COMP 110/401

Prasun Dewan[1]

14. Arrays

A string is one kind of sequence. In this chapter, we will see how we can create arbitrary sequences using arrays. We will use arrays to create several new kinds of types including sets, histories, and databases. Some of these types can be considered as special cases of other types. To get experience with processing arrays, we will study another kind of loop statement, called the *for* loop. The for loop has a different syntax than the *while* loop, but we will see that in fact they are equivalent – whatever you can do with a for loop, you can do with a while loop, and vice versa.

For Loop

The following program fragment illustrates string manipulation, printing the characters of a string on separate lines:

```
// String s declared and initialized earlier

int i = 0; // initialization of loop variables
while (i < s.length()) { // continuation condition
    System.out.println (s.charAt(i)); // real body
    i++; // resetting loop variables
}
```

This is an example of a counter-controlled loop, since the variable `i` serves as a counter that is incremented in each iteration of the loop. It is also an example of an *index-controlled loop*, which is a special case of a counter-controlled loop in which the counter is an index.

Let us look at this code in more depth to grasp the general nature of a loop. We can decompose it into the following components:

1. *Continuation condition*: the boolean expression that determines if the next iteration of the loop should be executed.
2. *Initialization of loop variables*: the first assignment of variables that appear in the continuation condition and are changed in the loop body.
3. *Resetting loop variables:* preparing for the next iteration by reassigning one or more loop variables.
4. *Real body:* the rest of the while body, which does the "real-work" of each iteration.

The while loop separates the continuing condition from the rest of the components, thereby ensuring we do not forget to enter it. However, it does not separate the remaining components – we have added comments to do so in the example above. As a result, we may forget to create one or more of these components. For instance, we may forget to reset the loop variables, thereby creating an infinite loop.

Java provides another loop, the *for loop* or *for statement,* illustrated below, which explicitly separates all four components:

```
for (int i = 1; i < s.length(); i++)
    System.out.println (s.charAt(i));
```

This loop is equivalent to the program fragment above. In general, a for statement is of the form:

```
for (S1;E;S2)
      S3
```

It is equivalent to:

```
S1;
while (E) {
      S3;
      S2;
}
```

In other words, the for loop first executes S1 and then essentially executes a while loop whose condition is E and whose body is S3 followed by S2. S1 and S2 are expected to initialize and reset, respectively, the loop variables.

In comparison to a while loop, a for loop is more compact. More important, its structure reminds us to enter S1 and S2, that is, initialize and reset the loop variables.

Any of the three parts of a for loop, S1, E, and S2 can, in fact, be missing. A missing S1 or S2 is considered a null statement, while a missing E is considered true. Thus:

```
for (;;) {
      S3;
}
```

is equivalent to:

```
while (true) {
      S3;
}
```

A *break* statement can be used to exit both kinds of loops. We will learn about break statements later.

Arrays

A string is a sequence of values of type `char`. What if we wanted a sequence of other types of values such as `int`, `double`, or `String` values? One could imagine restricted solutions to this problem. Java could provide an `IntSequence` type for defining sequences of `int` values. Similarly, it could define the types `DoubleSequence`, `StringSequence`, and so on. The problem with this approach is that no matter how many predefined sequence types we have, we might want a kind of sequence that was not anticipated by Java. For instance, what if we wanted a sequence of instances of the type `Loan` we defined earlier? This is not a type known to Java, so it cannot predefine such a sequence.

Therefore, instead, it lets us, as programmers, define our own indexable sequences, called *arrays*, which can contain elements of some type specified by us. Like other values, they can be stored in variables of the appropriate type. The following declaration illustrates how an array variable and array value are created:

```
String[] names = {"John Smith", "Jane Doe"};
```

Let us decompose this declaration to better understand it:

1. The variable definition:

   ```
   String[] names
   ```

   declared a new array variable, called `names`, whose type is `String[]`. `String[]` denotes the string-array type, that is, it is the type of an array with `String` elements. Thus the definition says that the variable `names` can store string-arrays.

2. The expression:

   ```
   {"John Smith", "Jane Doe"};
   ```

   creates a new string-array consisting of the two strings, `"John Smith"` and `"Jane Doe"`. This expression can be considered as an array literal; like other literals we have seen before, it directly indicates a value rather than identifying a variable storing the value.

3. The initialization assignment:

   ```
   names = {"John Smith", "Jane Doe"};
   ```

   assigns the string-array value on the RHS to the string-array variable on the LHS.

`String[]` is the type of all string arrays, regardless of their size. A string-array variable, thus, can be assigned string arrays of different sizes. For instance, we can reassign to `names` a 3-element array:

```
names = {"John Smith", "Joe Doe", "Jane Doe"};
```

Similarly, we can create and initialize other types of arrays:

```
int[] scores = {45, 32, 68};
```

```
Loan[] loans = {new ALoan(100000), new AnotherLoan (100)};
```

Like `String`, an array type is a variable but not a dynamic type, that is, its instances can have different sizes but the size of a particular array is fixed during its lifetime. Thus, when we reassigned `names` above, we did not extend the size of the array to which it was assigned; instead, we assigned it a new array with a larger size.

Like other variables, array variables need not be initialized when they are created. Instead, we can assign to them later in a separate assignment statement, as shown below[2]

```
String[] names;
names = {"John Smith", "Jane Doe"};
```

In fact, we have already seen the syntax for declaring uninitialized array variables while declaring a main method:

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

The formal parameter, `args`, is declared as an uninitialized string array. When a user enters a list of string arguments, Java stores the list in a string array, and assigns this value to the formal parameter.

Accessing and Modifying Array Elements

Like strings, arrays can be indexed, but the syntax for doing so is more convenient. Even though arrays are objects, we do not invoke a method such as `charAt` to index an array. Instead, we simply enclose the index within square brackets. Thus the 1st element of `names` is:

```
names[0]
```

and the nth element is

```
names[n-1]
```

The size of the array is given by:

```
names.length
```

---

[2] Java also accepts an alternative syntax for declaring array variables. In addition to the syntax:

```
<Element Type>[] <String Array Variable>
```

it allows:

```
<Element Type> <String Array Variable> []
```

Thus,

```
String names[];
```

is equivalent to:

```
String[] names;
```

The legal indices of the array, thus, are:

```
0..names.length - 1
```

If we use an index that is not in this range, Java will raise an `ArrayIndexOutOfBoundsException`.

Note that unlike the case of a string, `length` is not an instance method, and thus is not followed by parentheses. Instead, it is essentially a public instance variable of the array. Though we can read the value of this variable, we cannot assign to it, because the size of an array is fixed during its lifetime. It should be regarded as a special instance variable of an array that cannot be modified after it has been assigned the first time.

Uninitialized Array Elements

Unlike a string, an array can be modified. For instance, the following statement:

```
names[0] = "Johnny Smith"
```

assigns a new value to the first element of the array. Recall that it is not possible to similarly modify an element of a string.

In fact, when an array is created, typically, we do not know what the values of its elements would be. Often, as we will see later, they are assigned based on the user input. Therefore, Java allows us to create an array with uninitialized elements. For instance, the expression:

```
new Loan[MAX_LOANS]
```

creates a new loan array of size `MAX_LOANS` whose elements are uninitialized.

Since the size of a Java array is fixed during its lifetime, when it is created, we must tell Java how many elements it has, even if we do not know then what the values of these elements will be. Thus, the square brackets after an element type must enclose a size or *dimension* when an array instance is created, since the instance is fixed-size, but not when an array type is specified, since the type is variable-sized.

An array with uninitialized elements can be used to initialize an array variable:

```
Loan[] loans = new Loan[MAX_LOANS];
```

Here, the array variable, `loans`, is initialized with an array of size `MAX_LOANS`. This means that the value of the variable is a collection of `MAX_LOANS` slots that can store values of type Loan. However, these slots are themselves uninitialized, that is, have no values assigned to them. Later, we can initialize them:

```
loans[0] = new ALoan(2,3);
loans[1] = new AnotherLoan(3,1);
```

Thus, think of the array elements, `loans[0]` and `loans[1]`, as themselves variables, which should be initialized before accessing their values.

Figure 1. an uninitialized loan-array variable, loans

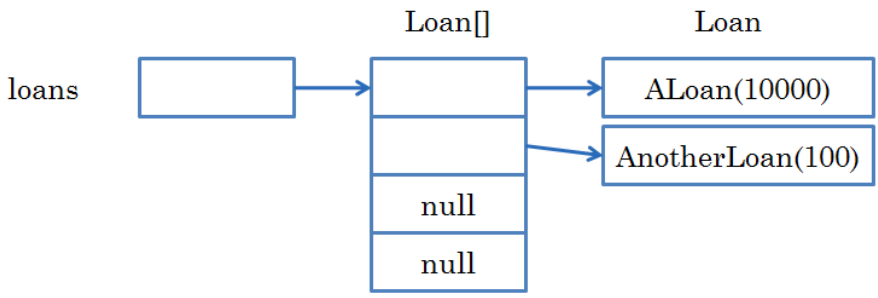


Figure 2. an initialized loan-array variable, loans, with first two elements initialized and the rest uninitialized

Accessing Uninitialized Variables

Arrays must often be created with uninitialized array elements. There is the danger, thus, of accidentally accessing an uninitialized array element. What exactly happens when we try and access an uninitialized array element? Actually, let us try and answer the more general question: What exactly happens when we try and access any uninitialized variable?

There are three cases:

- *Compiler error*: if the variable is a local variable, then the compiler flags such an access as an error, and refuses to generate executable code.
- *Default value*: if the variable is a global (class or instance) primitive-variable, then a default value of the type is stored in the variable. Therefore when we access this variable, we get this value. The default value, in the case of numeric types, is 0, and in the case of other primitive types, is a value whose computer encoding is 0. This means in the case of `char`, it is the null character, and in the case of `boolean`, it is false.
- *Special value*: if the variable is a global object-variable, Java does not store a default object of that type. Instead, it stores a special value, called `null`. If we try and access the variable, we get a `NullPointerException`. Beware of this exception - it is very easy to write programs that access uninitialized object variables!

The first approach of generating a compiler error seems the best because before we even run the program we are told about the mistake. Why does Java not use it also for global variables? The reason is that these can be accessed by multiple methods, and it is difficult, and sometimes impossible, to know, when compiling a class, whether these variables have been initialized before access.

Why use different approaches for primitive and object global variables, and which approach is better? The approach of storing a special value is probably better because it allows the program to know (at

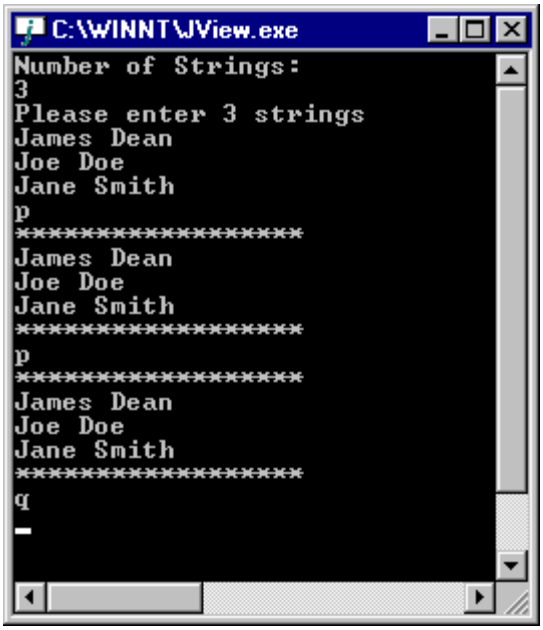


Figure 3. A fixed-size collection

runtime) that it has accessed an uninitialized variable. In the other approach, if the default value is one of the expected values of a variable in a particular program, then the program may not even know it has accessed an uninitialized variable. Storing and checking a special uninitialized value for too much of a cost to pay for primitive types, which are used extensively in programs. Therefore, all values that can be stored in a primitive variable are considered legal.

Let us return to array variables, and better understand what it means for them to be uninitialized.

Figure 1 and Figure 2 illustrate the two forms of uninitialization in a program with arrays. Figure 1, the array, `loans`, itself is uninitialized. In Figure 2, the array is initialized, but its 3rd and 4th elements are uninitialized.

Thus, if you get a `NullPointerException` in a program that uses arrays, you may have forgotten to initialize either an array variable or an array element.

Fixed-Size Collections

Arrays can be used to implement a variety of collections such as databases, histories, sets, and ordered lists. Consider the implementation of a simple, fixed-size, database shown in Figure 3.

It collects a series of input strings entered by the user, and prints them on request. The number of strings in the collection is specified before the first element is input.

We have done problems before that processed a list of items such as the problem of computing the sum of a list of input numbers. We did those problems without using arrays; so why do we need arrays here?

In the previous problems, after the sequence of input values was processed, it was not needed again. This is not the case here, because we might want to print the input values multiple times. Therefore, the values must be stored in memory in some variable. The type of the variable cannot define a fixed-size because the number of elements in the collection is specified at runtime. The only variable-size types we have seen so far are `String` and the array types. Since the items being collected are strings and not characters, we should use the array type, `String[]`, as the type of the collection.

We are now ready to give the top-level of our algorithm. This level invokes methods to get an instance of the database type and print it:

```
public static void main(String[] args){
      String[] names = getStrings();
      String command = Console.readString();
      while (command.length() > 0 && command.charAt(0) != 'q') {
            if (command.charAt(0) == 'p')
                  // print database if user enters 'p'
                  print(names);
            String command = Console.readString();
      }
}
```

The main method first calls `getStrings`, which returns the list of lines entered by the user as a string array. The program assigns this value to the string array-variable, `names`. It then processes each command string entered by the user. If the string is:

1) the quit command, that is, the first character of the string is 'q', then the program terminates.
2) the print command, that is, the first character of the string is 'p', then the program calls the `print` method to print the string-array collection.
3) any other string the program simply ignores it.

Before looking at the first character of the command, the program checks that it has at least one character. It is possible for it to have no characters – if the user presses the Enter key without entering anything in the line, then the `readString` method returns the null string. If the length check is not made, then the Java will throw a `StringIndexBounds` exception if we try to access the first character of a null string.

To complete this implementation, we must define the `print` and `getStrings` methods.

Processing Arrays of Different Sizes

The method, `print`, simply prints all elements of its argument. Since the argument can be a string array of arbitrary size, it uses the `length` variable to determine how many elements there are:[3]

```
static void print(String[] strings) {
      System.out.println("******************");
      for (int elementNum = 0; elementNum < strings.length; elementNum++)
            System.out.println(strings[elementNum]);
      System.out.println("******************");
}
```

Runtime Array-Size

The method, `getStrings`, is a dual of `print`. Instead of receiving an array as an argument from its caller, it returns an array to its caller.

```
static String[] getStrings() {
      System.out.println("Number of Strings:");
      int numElements = Console.readInt();
      System.out.println("Please enter " + numElements + " strings");
      String[] strings = new String[numElements];
      for (int elementNum = 0; elementNum < numElements; elementNum++)
            strings[elementNum] = Console.readString();
      return strings;

}
```

It reads the number of elements entered by the user, and uses this value in

```
            String[] strings = new String[numElements];
```

to create an array of the right size. As we see here, the array dimension specified in `new` does not have to be a constant. Even though an array is a fixed-sized instance, the size is not determined at compile time. It is determined when the array is instantiated.

After instantiating the array, this method stores each input value in the array and returns when the array is completely full. It does not access the `length` field of the array since it knows the array size (stored in `numElements`) having created the array itself.

[3] Methods such as this one that access arrays of different sizes were not possible in languages such as Pascal with fixed-size array types. In such languages, we would need a separate method for each array-size. Thus, we see here an important benefit of variable-sized array types.

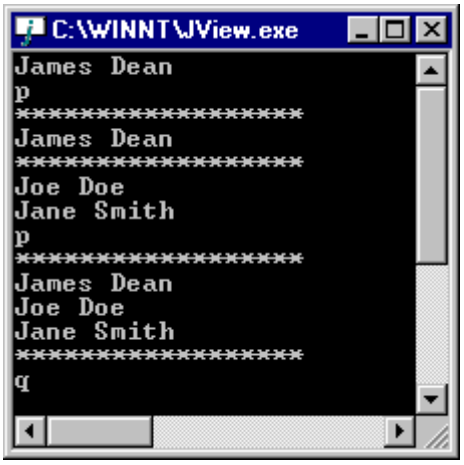


Figure 4. A variable-size collection

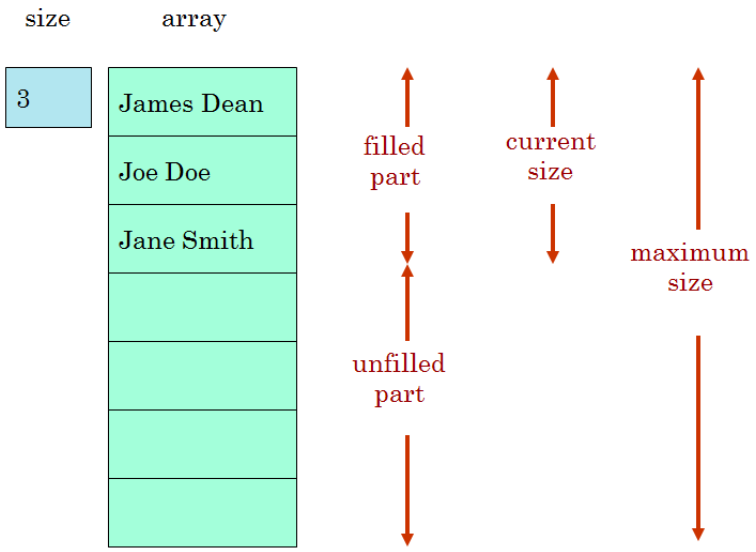


Figure 5.

Variable-Sized Collections

Let us consider a more interesting variation of the problem above, shown in Figure 4. In this problem, users do not specify the number of collection elements at the start of the interaction. They keep entering strings until they quit the program. Moreover, they do not have to wait for the entire database to be entered before printing it. They can print the database after a subset of it has been entered.

This is a more complicated problem. As before, we must collect the strings in memory so that they can be printed multiple times. But after they have been printed, a user can enter additional items. Thus, we need to create a collection in memory that can grow dynamically.

It is, in fact, possible to use a fixed-size array to store a variable-size collection, much as we write variable-sized lists in fixed-size pages. As the following figure illustrates, such a collection can be

simulated using an array and two `int` variables. The size of the array is the maximum size of the collection, given by one of the `int` variables. The filled part consists of elements of the variable-sized collection and the unfilled part consists of the unused elements of the array. The size of the filled part, thus, ranges from 0 to the array length. We store this value in the second `int` variable, and update it whenever we add or delete a collection element.

Thus, changing the size of the collection involves changing the boundary between the filled and unfilled parts of the array. Since the length of an array cannot change once it has been instantiated, the "variable" storing the value of the maximum size of the collection is usually declared as `final` to make it readonly.

Encapsulating Related Variables

It is considered good programming practice to relate the names of the three variables defining the variable-sized collection. The popular approach (used in most textbooks) is to declare these variables in the class that needs the collection, and name the current and maximum size variables `aSize`, and `A_MAX_SIZE`, respectively, if the array is named `a`.

```
final int A_MAX_SIZE = 50;
String[] a = new String[A_MAX_SIZE];
int aSize = 0;
//process collection a
…
```

If the class needs another collection of this kind, it similarly creates another three variables:

```
final int B_MAX_SIZE = 50;
String[] b = new String[B_MAX_SIZE];
int bSize = 0;
//process collection b
…
```

We can, in fact, do better than this, since each time we need a group of related variables, we should declare them as instance variables of a new class. Here is one way of doing so:

```
public class AVariableSizedStringCollection {
      public final int MAX_SIZE = 50;
      public String[] contents = new String [MAX_SIZE];
      public int size = 0;
}
```

In this declaration, the three variables are declared as public instance variables of the class. Each time we need a new dynamic collection of strings, we can create a new instance of this class and store it in some variable:

```
AVariableSizedStringCollection a = new AVariableSizedStringCollection();
```

```
AVariableSizedStringCollection b = new AVariableSizedStringCollection();
```

We can then refer to the array and size components of it, as shown below:

```
a.MAX_SIZE
a.contents
a.size
b.MAX_SIZE
 ….
```

This approach is superior to creating separate variables, such as `aSize` and `a`, because it creates a new type that better matches the problem than the separate array and `int` types do individually. As a result, it gives us the advantages of defining high-level types such as reuse of the declarations defining the variable size collection and allowing a dynamic collection to be returned by a function. However, it does not meet the principle of least privilege. Because both the non-final instance variables have been declared public, users of this class can manipulate them in arbitrary ways. For instance, they can add a new element to the array without incrementing the size field; or make the size field become larger than MAX_SIZE.

History

Therefore, we should *encapsulate* the variables, that is, make them non-public and provide access to them through public methods. These methods depend on the nature of the collection we want. In the problem of Figure 5, we need operations to:

- add an element to the collection,
- examine each element of the collection so that we can print it.

We do not need operations to modify or delete elements. Thus, the collection we need is really a *history* of strings – as in the case of a history of past events, its filled portion cannot be overwritten.

The following interface defines these operations:

```
public interface StringHistory {
      public void addElement(String element);
      public String elementAt (int index);
      public int size();
}
```

Like `String`, the interface provides a method to access an element at a particular index. It is, of course, more convenient to index a collection like an array using the square brackets, `[` and `]`. Unfortunately, in Java, such indexing cannot be used for programmer-defined types, requiring special support from the programming language.

Like `String`, this interface also provides a method to find the size of the collection. Finally, unlike the immutable `String`, it provides a method to add an element to the collection.

The three variables creating a variable-size storage for the history elements are defined, not in the interface, but in its implementation, `AStringHistory`:

```
public class AStringHistory implements StringHistory {
	public final int MAX_SIZE = 50;
	String[] contents = new String[MAX_SIZE];
	int size = 0;
	public int size() {
		return size;
	}
	public String elementAt (int index) {
		return contents[index];
	}
	boolean isFull() {
		return size == MAX_SIZE;
	}
	public void addElement(String element) {
		if (isFull())
			System.out.println("Adding item to a full history");
		else {
			contents[size] = element;
			size++;
		}
	}
}
```

Unlike `AVariableStringCollection`, `AStringHistory` does not make these instance variables public. As a result, it supports encapsulation – all access to these variables is through the public methods of the class. As a result, there is no danger these variables will be manipulated in an inconsistent manner from outside the class. Moreover, if we were to change the variables by, for instance, renaming them, increasing the maximum size, or replacing an array with another data structure (such as Vector, discussed later), the users of the class would not know the difference and thus could be reused with the new implementation.

Most of the methods of this class are trivial. The method `size` returns the current size, `elementAt` indexes the array to return its result, and `isFull` returns true if the current size has reached its maximum value.

The method `addElement` requires some thought. If the collection is not full, the value of `size` is the index of the first element of the unfilled part of the array. In this case, it simply assigns to this element the new value, and increments `size`. The tricky issue here is what should happen when the collection is full? If we have guessed the maximum size right, this condition should not arise. Nonetheless, we must decide how to handle this situation. Some choices are:

- *Subscript Exception*: We could ignore this situation, and let Java throw an `ArrayIndexOutOfBoundsException` when the method accesses the array beyond the last element. However, this may not be very illuminating to the user, who may not know that the implementation uses an array (rather than, say, a vector, discussed later).
- *Specialized Exception:* We could define a special exception for this situation, `CollectionIsFullException`, and throw it. However, defining new exceptions is beyond the scope of this course.
- *Print Message*: We would print a message for the user. Unfortunately, the caller of `addElement` gets no feedback with this message.
- *Increase Size*: If it is not an error to add more elements than the original array can accommodate, we could increase the size of the collection by creating a larger array, add all the elements of the previous array to it, and assign the new array to the contents field.

Our program assumes it is an error to add to a full collection and simply prints a message. As the discussion above points out, had we known how to define one, a specialized exception would have been a better alternative under this assumption.

The following main method illustrates how `StringHistory` and its implementation `AStringHistory`, are used to solve our problem:

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

      StringHistory names = new AStringHistory(); // create an empty history
      while (true) {
             String input = Console.readString();
             if (input.length > 0)
                   if (input.charAt(0) == 'q')
                         // exit loop if user enters 'q'
                         break;
                  else if (input.charAt(0) == 'p')
                         // print database if user enters 'p'
                         print(names);
            else // add input to history
                   names.addElement(input);
            }
      }
```

It is much like the main we used for the fixed-size collection with two main differences. Instead of the array type `String[]`, it uses our new type `StringHistory`. Second, instead of gathering all database entries before starting the command loop, it gathers database entries incrementally in the loop. If a non-null input line does not begin with any of the recognized commands, the loop assumes it is a database entry, and calls the `addElement` method to add it to the `StringHistory` instance.

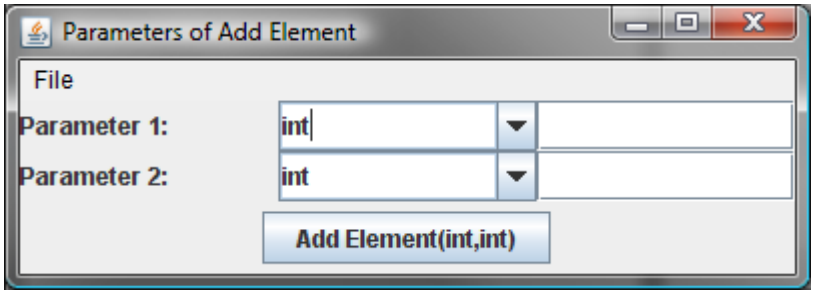


Figure 6. AddElement of APointHistory

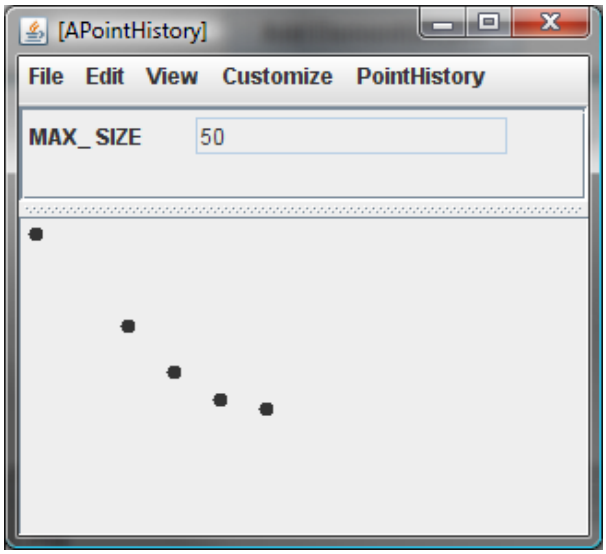


Figure 7. APointHistory with 5 points

Similarly, the `print` method is like the one we saw for a fixed-size collection except that it invokes `StringHistory` methods rather than the corresponding array operations.

```
static void print(StringHistory strings) {
      System.out.println("******************");
      for ( int elementNum = 0; elementNum < strings.size(); elementNum++)
            System.out.println(strings.elementAt(elementNum));
}
```

Database

A history is perhaps the simplest example of a variable-sized collection. Let us define a more sophisticated collection shown in Figure 8.

In addition to the commands to add to and print the collection, this application provides commands to delete an entry (d), check if an entry is a member of the collection (m), and clear the whole collection (c). Thus, the collection forms a simple string *database,* providing commands for searching, adding, and deleting entries. The following interface describes the new type:

```
public interface StringDatabase {
    // methods of StringHistory
    public String elementAt(int index);
    public void addElement(String element);
    public int size();
    // additional methods
    public void deleteElement(String element);
    public void clear();
    public boolean member(String element);
}
```

It retains all the methods of the `StringHistory`, including additional methods to delete an element, clear the collection, and check if an item is present in the collection.

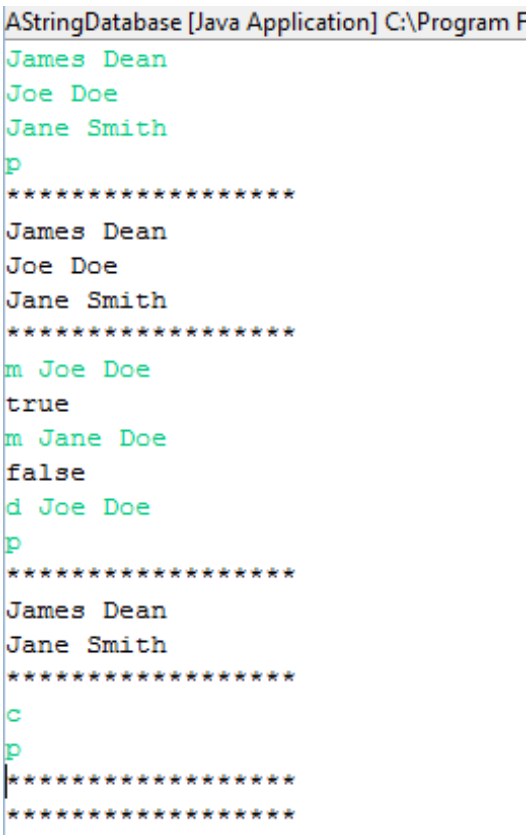


Figure 8. AStringDatabase

Similarly, the implementation of this interface retains the variables and methods of `StringHistory`.

```
public class AStringDatabase implements StringDatabase {
    // code from StringHistory
    …
    // additional code
    …
}
```

containing additional code for the three new operations, which is described below.

Deleting an Entry & Multi-Element List Window

Let us first consider the implementation of the `deleteElement` operation. Deleting an entry from a collection (implemented as an array) is more difficult than adding one. We have assumed that in a collection, the positions of the entries do not matter, or if they do matter, the order in which they are added determines their position (and not, say, the alphabetic order). Therefore, we always assigned the new entry at the first unfilled position of the array, and simply incremented the size field. The delete command allows us to remove an entry from the middle of the array. Our implementation strategy for a

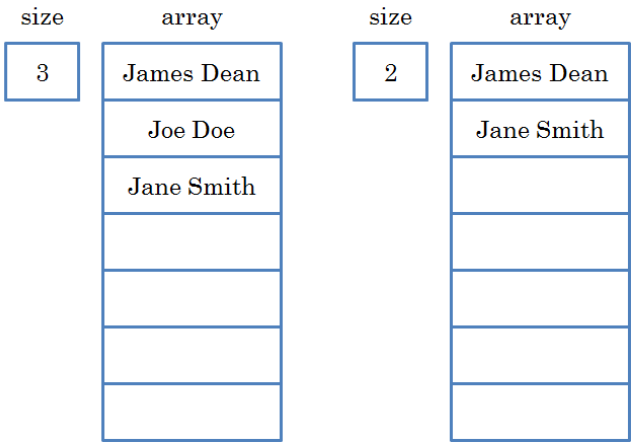


Figure 9. Deleting the second element from a history that contains three elements

variable-size collection assumes that all unfilled entries follow the filled entries. Thus, we cannot leave an unfilled "hole" in the middle of the filled area.

A simple approach would be to take the last element and put it in the slot of the deleted element. However, this does not preserve the order in which the entries were added. Assuming that `print` should list the elements in this order, we must, instead, shift all the elements following the deleted item up one position. Figure 9 shows the contents of the array before and after "Joe Doe" is deleted from the collection.

In order to do the move, we must examine successive pairs of slots in the array, starting from the deleted position, and store the contents of the slot at the larger index in the slot at the lower index. Thus, for each of these pairs, we must either perform the assignment:

```
contents[index] = contents[index + 1]
```

or

```
contents[index - 1] = contents[index]
```

Let us chose the first assignment since it allows us to start at the index at the position of the deleted item. We are now ready for the code required for deletion:

```
public void deleteElement(String element) {
    shiftUp(indexOf(element));
}

void shiftUp(int startIndex) {
    for (int index = startIndex; index + 1 < size; index++) {
        contents[index] = contents[index + 1];
    }
    size--;
}
```

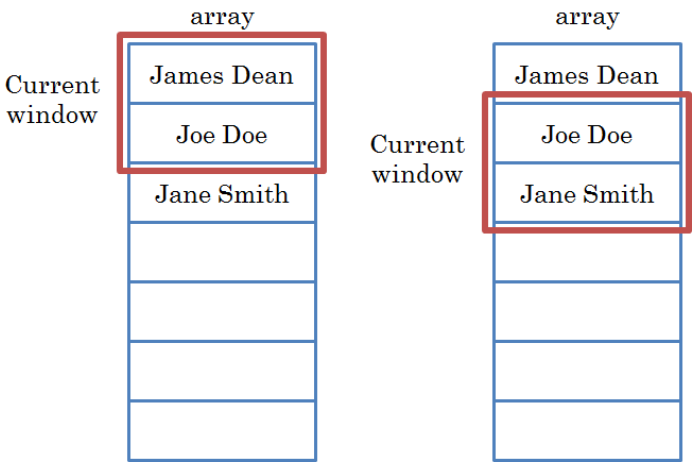


Figure 10. A 2-element window-based list traversal

In our previous list traversals, we examined one entry of the list at a time. In this traversal, we examine two consecutive list elements at a time. Both are examples of *window-based* list traversals, where each list traversal step (loop iteration or recursive step) accesses a fixed size window of neighboring elements in a list, and each subsequent step moves the window up or down by a fixed step as shown in Figure 10.

In a forward (backward) list traversal, the termination condition is the last (first) window element becoming the last (first) list element. This is the reason that our loop for the two-element window is:

```
index + 1 < size
```

rather than:

```
index < size
```

which was the condition for the one-element window traversal.

Searching for an Element

Our delete operation assumes we have an operation, `indexOf`, to find the position of a list element. The operation traverses the list using a one-element window, stopping when it finds the element or reaches the end of the list:

```
int indexOf(String element) {
    int index = 0;
    while ((index < size) && !element.equals(contents[index]))
        index++;
    return index;
}
```

To decide if the element at the current index is equal to the element being sought, the `indexOf` function uses the predefined `equals` operation in String. The `equals` operation is invoked on a String instance and takes a single formal parameter also of type String. The operation returns `true` if and only

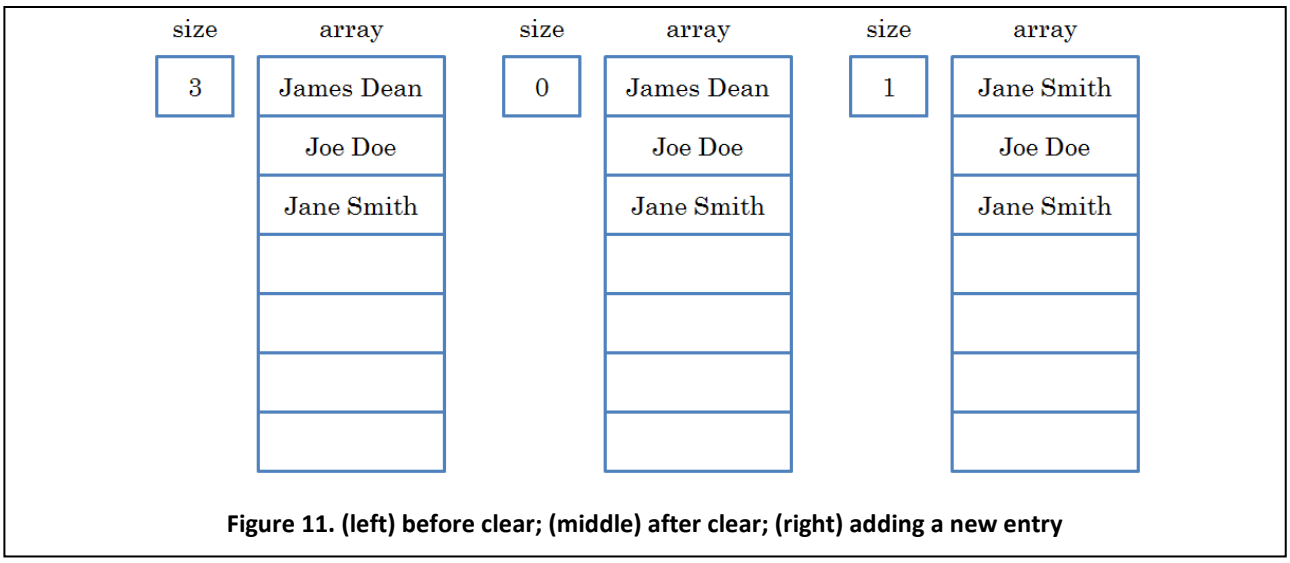


Figure 11. (left) before clear; (middle) after clear; (right) adding a new entry

if the String instance on which it is being invoked is identical (in terms of number of characters and the characters themselves) to the String instance passed as the actual parameter to the formal parameter.

Note that in the case the item exists multiple times in the collection, the `indexOf` function returns the position of the first element that matches the item for which we are searching. If the item is not in the array, it returns the value of the `size` variable. In this case, the loop in `shiftUp` will be skipped and no item will be deleted.

With this method, we can trivially implement the `member` operation:

```
public boolean member(String element) {
    return indexOf(element) < size;
}
```

That leaves us the `clear` method. We could clear the database by deleting each item of the array:

```
public void clear() {
    while (size > 0) {
        deleteElement(size - 1);
    }
}
```

At each iteration of the loop, the element at `size - 1` is the last element in the collection. We can thus clear the database by repeatedly deleting its last item until we have no more items left.

However, it is much simpler and more efficient to make the current size 0:

```
public void clear() {
    size = 0;
}
```

Thus, `clear` (and `deleteElement`) simply adjusts the size field without explicitly removing from the array any element. Figure 11 shows the array contents before and after the clear operation.

As we can see, the “deleted” elements still occupy space in the array. However, they will be replaced with any new items we add to the collection – in that sense they do not take space in the array. For instance, if we were to now add the string “Joe Doe” to the collection, it would replace “James Dean” in the first element of the array. Thus, the difference between deleted and undeleted elements is that the slots of the latter are in the “unfilled” space and are used to add new elements.

When the first slot is reassigned the string, “Joe Doe”, what happens to the old value of the slot, “James Dean”? Clearly, it is no longer in the array, but is it also no longer in memory? This string is essentially “garbage” in that we are no longer interested in it; thus it should be removed from memory. Java does *garbage collection* to automatically find and de-allocate any such unused space from memory. In languages such as Pascal and C++ that do not do garbage collection, we would have to explicitly dispose of this space. It is very easy to make mistakes by either accidentally de-allocating useful space or not de-allocating useless space. Java’s garbage collection, thus, makes our programs easier to write and more reliable. How Java does garbage collection is beyond the scope of this course. What is important is that, like garbage collection in the real world, Java’s garbage collection is not done each time some garbage is created, but later, according to criteria determined by the Java garbage collector, often when space is running low. You might see your program slow down when the garbage collector becomes active.

Switch and Ordinal Types

Now that we have seen how the various methods of `AStringDatabase` are implemented, let us complete the implementation of the database problem by giving the main method:

```
public static void main(String args[]) {
      StringDatabase names = new AStringDatabase();
      while (true) {
            String input = Console.readString();
            if (!(input.length() == 0))
                  if (input.charAt(0) == 'q')
                        break;
                  else if (input.charAt(0) == 'p')
                        print(names);
                  else if (input.charAt(0) == 'd')
                        names.deleteElement(input.substring(
                              2, input.length()));
                  else if (input.charAt(0) == 'm')
                        System.out.println(names.member(
                              input.substring(2, input.length())));
                  else if (input.charAt(0) == 'c')
                        names.clear();
                  else
                        names.addElement(input);
      }
}
```

Instead of creating an instance of `AStringHistory`, the method creates an instance of `AStringDatabase`. The loop of this method extends the loop of the main method we created for the history example by processing the additional commands for deleting an entry, clearing the database, and checking for membership. The clear command is processed like the other commands we have seen so far since it is also a one-character command. The other two commands take an operand after the command character. The program extracts this operand using the `substring` operation and passes it as an argument to the method for processing the command.

Notice that we have been careful to do all the branching in the else parts of the if statements that discriminate among the different characters. As it turns, we can further improve the code by using an alternative conditional, called a *switch statement*, illustrated below:

```
public static void main(String args[]) {
      StringDatabase names = new AStringDatabase();
      while (true) {
            String input = Console.readString();
            if (!(input.length() == 0))
                  if (input.charAt(0) == 'q')
                        break;
                  else switch (input.charAt(0)) {
                        case 'p':
                              print(names);
                              break;
                        case 'd':
                              names.deleteElement(
                                    input.substring(2,
                              input.length()));
                              break;
                        case 'm':
                              System.out.println(names.member(
                                    input.substring(2,
                              input.length())));
                              break;
                        case 'c':
                              names.clear();
                              break;
                        default:
                              names.addElement(input);
      }
}
```

The switch statement selects among different values of the expression following the **switch** keyword called the *switch expression*. Each of the listed values is a *case* of the switch expression, and the statement sequence following it is called an *arm* of the switch. If the value of the switch expression is equal to a particular case, then control transfers to the corresponding arm.

As shown above, Java does not require us to list all possible cases for the switch expression. The default clause stands for all unlisted cases.

It is possible to associate multiple cases with the same arm, as shown below:

```
switch (input.charAt(0)) {
      case 'p', 'P':
            print(names);
            break;
      case 'd', 'D':
            names.deleteElement(input.substring(2, input.length()));
            break;
            …
}
```

The switch conditional is shorter, easier to read, and as it turns out, more efficient than an if-else conditional. While an if-else conditional does a two-way branch to the then or the else part, a switch statement does a multi-way branch to its different arms.

A switch is, not, however, suitable for all selection problems, since the type of the switch expression must be an *ordinal type*. An ordinal type is a primitive type with the following properties:

1) The instances/literals of the type are ordered.
2) For each literal, there is a unique successor/predecessor unless it is the last/first literal.

For instance, it can be used to test an `int` but not a `String` or `float` expression. Thus, if-else statements are more general but less high-level conditionals than switch statements.

Notice that we put a break statement at the end of each arm of the switch. In general, once it finishes executing an arm, a switch statement executes *all* of the subsequent arms until it finds a break**.** Consider what happens when we forget to put break statements

```
switch (input.charAt(0)) {
      case 'p':
            print(names);
      case 'd':
            names.deleteElement(input.substring(2, input.length()));
      case 'm':
            System.out.println(names.member(
                  input.substring(2, input.length())));
      case 'c':
            names.clear();
      default:
            names.addElement(input);
```

and the input character is ‘m’. The statement will executes the ‘m’ arm and all of the arms of the cases below it, thus executing the statements:

```
names.deleteElement(input.substring(2, input.length()));
System.out.println(names.member(input.substring(2, input.length())));
names.clear();
names.addElement(input);
```

The break statement makes the program jump out of the immediately enclosing statement block, which may be a loop or a switch. It is not possible to use it to break out a statement block that is not immediately enclosing it. Consider the following alternative main method:

```
public static void main(String args[]) {
      StringDatabase names = new AStringDatabase();
      while (true) {
            String input = Console.readString();
            if (!(input.length() == 0))
                  switch (input.charAt(0)) {
                        case 'q':
                              break;
                        case 'p':
                              print(names);
                              break;
                        case 'd':
                              names.deleteElement(
                                    input.substring(2,
                              input.length()));
                              break;
                        case 'm':
                              System.out.println(names.member(
                                    input.substring(2,
                              input.length())));
                              break;
                        case 'c':
                              names.clear();
                              break;
                        default:
                              names.addElement(input);
      }
}
```

This solution is more elegant in that it avoids the if conditional that tests if the input character is ‘q’, replacing it with an extra arm in the switch. Unfortunately, it does not work, because the break in the new arm terminates the enclosing switch, not the loop. Since in this example, the main method does not execute any statement after the loop, we can execute return instead of break in the arm corresponding to ‘q’:

```
switch (input.charAt(0)) {
      case 'q':
            return;
      …
```

This solution will work since the main method will return (to the interpreter) when this arm is executed, terminating the program.

PointHistory

Suppose we wish to enter and view a series of points in some two-dimensional space. For instance, we wish to enter and view points on the trajectory of a rocket or the points in an exam curve. Figure 7 shows a user-interface that allows us to perform this task.

It provides an operation, `addElement` (Figure 6) for adding a new point whose x and y coordinates are arguments of the operation. Each added point is displayed in the ObjectEditor graphics window.

As before, we would like to create an object responsible only for creating and manipulating the shapes, leaving the details of displaying and editing the shapes to ObjectEditor. To create a point on the screen, we can use our `Point` interface and `ACartesianPoint` and `APolarPoint` classes, which, as we saw before, represent a point that is automatically displayed by ObjectEditor as a small, filled circle. In this example, we must create a dynamic number of points. We have seen how to create a dynamic collection of textual elements. We can create a dynamic number of graphics elements in much the same way.

Let us, then, derive the interface of the object being edited in Figure 7. From Figure 6, we can directly derive one of the methods of this interface:

```
public void addElement (int x, int y);
```

If this object is not going to actually display the points, it must expose them to another object, in this case ObjectEditor, that performs this function. As we discussed in above for `StringHistory`, it cannot use getter and setter methods to expose these elements, since they work only for objects with a static number of elements. To identify how a dynamic list of elements can be exposed, consider the `StringHistory` interface we defined to expose a dynamic collection of strings:

```
interface StringHistory {
      public void addElement(String element);
      public String elementAt (int index);
      public int size();
}
```

The methods `elementAt` and `size`, together, allow an external object to determine all the strings in the history. We can define the exact same method headers for this example, with the only difference that instead of strings we would expose values of type `Point`. Since this generalizes to indexable collections of arbitrary types of elements, ObjectEditor looks in an object for (a) a method named `elementAt` taking a single `int` parameter, and (b) a parameter-less method named `size` to determine if the object encapsulates an indexable dynamic collection; and uses these methods to access the elements of the collection. This is in the spirit of looking for methods whose names start with `get` to determine and access the static properties of an object. In fact, we will refer to methods such as `elementAt` and `size` that retrieve dynamic components of a collection as getter methods, and

methods such as `addElement` that change the dynamic components of a collection as setter methods. Even though their names do not begin with `get` and `set`, they do perform the task of getting and setting components of an object.

Thus, our interface becomes:

```
public interface PointHistory {
      public void addElement(int x, int y);
      public Point elementAt(int index);
      public int size();
}
```

We have named the interface `PointHistory` to because, like `StringHistory`, it also defines a history, allowing elements to be appended but not inserted in the middle or deleted. After implementing `StringHistory`, these three methods do not present any new implementation challenges. Instead of repeating the code we used in `StringHistory`, however, we could have used Java vectors, which make the implementations of these methods trivial. We will learn about vectors later.

Other Important Collections: Stacks and Queues

Based on the StringHistory and PointHistory examples above, we see that the notion of a history is independent of a specific class or interface. It is an ordered collection that allows addition and retrieval of elements, but does not allow removal of them. Similarly, we can classify collections as databases if they allow searching, retrieval, addition, insertion, modification, and unconstrained deletion of elements.

Two other kinds of collections are stacks and queues. 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());
```

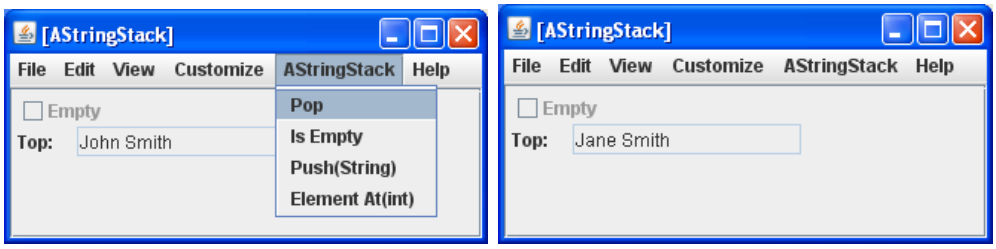
is:

James Dean
Joe Doe

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

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

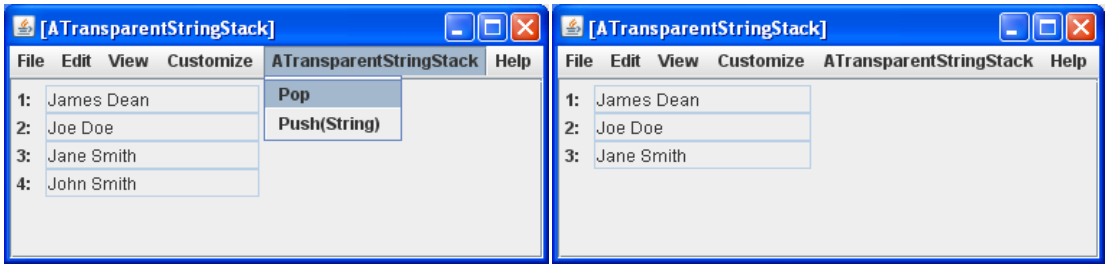
The nature of FIFO/LIFO objects can be better understood by considering the ObjectEditor display of an instance of StringStack.



Since the stack allows retrieval of only the top element of the collection, ObjectEditor cannot display other elements. However it is example 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.