Array Element Manipulations

Working With Arrays That Are Not Full

When an array is instantiated, the computer automatically fills its cells with default values. For example, each cell in an array of `int` initially contains the value 0. The application then replaces these values with new ones as needed. An application might not use all the cells available in an array, however. For example, one might create an array of 20 `ints`, but receive only 5 `ints` from interactive input. This array has a *physical size* of 20 cells, but a *logical size* of 5 cells currently used by the application. From the application's perspective, the remaining 15 cells contain garbage. Clearly, the application should only access the first five cells when asked to display the data, so using the array's physical size as an upper bound on a loop will not do. We solve this problem by tracking the array's logical size with a separate integer variable. The following code segment shows the initial state of an array and its logical size:

```
  int[] abc = new int[50];
  int size = 0;
```

Note that `abc.length` (the physical size) is 50, whereas `size` (the logical size) is 0.

Processing Elements In An Array That Is Not Full

We can use a loop to process all the data in an array of any size. The loop accesses each cell from position 0 to position `length` – 1, where `length` is the array's public instance variable. When the array is not full, one must replace the array's physical length with its logical size in the array `abc`:

```
  int[] abc = new int[50];
  int size = 0;

. . . code that puts values into some initial portion of the array and sets
the value of size . . .

  int sum = 0;
  for(int i  = 0; i < size; i++)
    sum += abc[i];
```

Adding Elements To An Array

The simplest way to add a data element to an array is to place it after the last available item. But it is critical to first check to see if there is a cell available. If there is not enough room to add more data, the array requires a larger physical size. When `size` equals `abc.length`, for instance, the array is full. Remember that Java arrays are of fixed size when they are instantiated. We can also insert an element at a position other than at the end. This process usually does not disrupt the order of existing elements, therefore, requires shifting of elements on one end of the insertion. In any of these cases, the array's logical size variable must be incremented to reflect the addition of an element into the array.

Removing Elements From An Array

Removing a data element from the end of an array requires no change to the array itself. We simply decrement the logical size, thus preventing the application from accessing the garbage elements beyond that point.

Removing a data element from an index that is not at the end requires shifting of some or all elements in order to maintain the list's order and contiguity. In either of these cases, the array's logical size variable must be decremented to reflect the removal of an element from the array.

Copy An Array

Copying an array must be done with care. Assigning one array variable to another does not do the job. It merely yields two variables referencing the same array. To make a true copy of an existing array, a new array must be created and set to the size desired. Then each element from the source array can be copied into the receiving array, index by index, using a for loop.