PARTIALLY FULL ARRAYS

Consider an application where the number of items that we want to store in an array changes as the program runs. Since the size of the array can't actually be changed, a separate counter variable must be used to keep track of how many spaces in the array are in use. (Of course, every space in the array has to contain something; the question is, how many spaces contain useful or valid items?)

Consider, for example, a program that reads positive integers entered by the user and stores them for later processing. The program stops reading when the user inputs a number that is less than or equal to zero. The input numbers can be kept in an array, `numbers`, of type `int[]`. Let's say that no more than 100 numbers will be input. Then the size of the array can be fixed at 100. But the program must keep track of how many numbers have actually been read and stored in the array. For this, it can use an integer variable, `numCount`. Each time a number is stored in the array, `numCount` must be incremented by one. As a rather silly example, let's write a program that will read the numbers input by the user and then print them in the reverse of the order in which they were entered. (This is, at least, a processing task that requires that the numbers be saved in an array. Remember that many types of processing, such as finding the sum or average or maximum of the numbers, can be done without saving the individual numbers.)

```
public class ReverseInputNumbers {

   public static void main(String[] args) {

     int[] numbers;  // An array for storing the input
values.
     int numCount;   // The number of numbers saved in the
array.
```

```
        int num;        // One of the numbers input by the
    user.

        numbers = new int[100];   // Space for 100 ints.
        numCount = 0;             // No numbers have been
    saved yet.

        TextIO.putln("Enter up to 100 positive integers;
    enter 0 to end.");

        while (true) {   // Get the numbers and put them in
    the array.
            TextIO.put("? ");
            num = TextIO.getlnInt();
            if (num <= 0)
               break;
            numbers[numCount] = num;
            numCount++;
        }

        TextIO.putln("\nYour numbers in reverse order
    are:\n");

        for (int i = numCount - 1; i >= 0; i--) {
            TextIO.putln( numbers[i] );
        }

      } // end main();

    }  // end class ReverseInputNumbers
```

It is especially important to note that the variable `numCount` plays a dual role. It is the number of items that have been entered into the array. But it is also the index of the next available spot in the array. For example, if 4 numbers have been stored in the

array, they occupy locations number 0, 1, 2, and 3. The next available spot is location 4. When the time comes to print out the numbers in the array, the last occupied spot in the array is location `numCount - 1`, so the `for` loop prints out values starting from location `numCount - 1` and going down to 0.

Let's look at another, more realistic example. Suppose that you write a game program, and that players can join the game and leave the game as it progresses. As a good object-oriented programmer, you probably have a class named *Player* to represent the individual players in the game. A list of all players who are currently in the game could be stored in an array, `playerList`, of type `Player[]`. Since the number of players can change, you will also need a variable, `playerCt`, to record the number of players currently in the game. Assuming that there will never be more than 10 players in the game, you could declare the variables as:

```
Player[] playerList = new Player[10];  // Up to 10 players.
int      playerCt = 0;  // At the start, there are no players.
```

After some players have joined the game, `playerCt` will be greater than 0, and the player objects representing the players will be stored in the array elements `playerList[0]`, `playerList[1]`, ...,`playerList[playerCt-1]`. Note that the array element `playerList[playerCt]` is **not** in use. The procedure for adding a new player, `newPlayer`, to the game is simple:

```
playerList[playerCt] = newPlayer; // Put new player in next
                                  //     available spot.
playerCt++;  // And increment playerCt to count the new player.
```

Deleting a player from the game is a little harder, since you don't want to leave a "hole" in the array. Suppose you want to delete the player at index `k` in `playerList`. If you are not worried about keeping the players in any

particular order, then one way to do this is to move the player from the last occupied position in the array into position `k` and then to decrement the value of `playerCt`:

```
playerList[k] = playerList[playerCt - 1];
playerCt--;
```

The player previously in position `k` is no longer in the array. The player previously in position `playerCt - 1` is now in the array twice. But it's only in the occupied or valid part of the array once, since `playerCt`has decreased by one. Remember that every element of the array has to hold some value, but only the values in positions 0 through `playerCt - 1` will be looked at or processed in any way. (By the way, you should think about what happens if the player that is being deleted is in the last position in the list. The code does still work in this case. What exactly happens?)

Suppose that when deleting the player in position `k`, you'd like to keep the remaining players in the same order. (Maybe because they take turns in the order in which they are stored in the array.) To do this, all the players in positions `k+1` and above must move down one position in the array. Player `k+1` replaces player `k`, who is out of the game. Player `k+2` fills the spot left open when player `k+1` is moved. And so on. The code for this is

```
for (int i = k+1; i < playerCt; i++) {
    playerList[i-1] = playerList[i];
}
playerCt--;
```

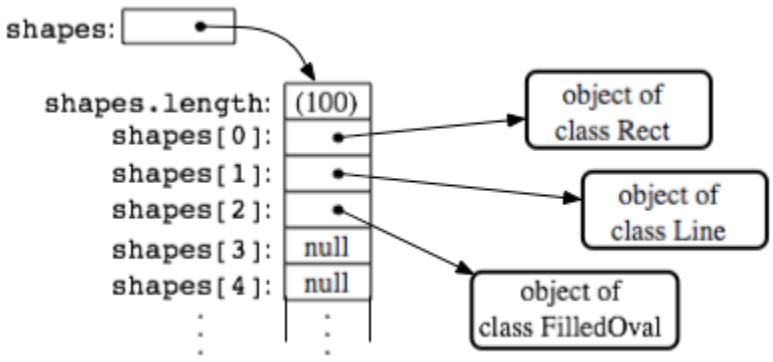
It's worth emphasizing that the *Player* example deals with an array whose base type is a class. An item in the array is either `null` or is a reference to an object belonging to the class, *Player*. The *Player* objects themselves are not really stored in the array, only references to them. Note that because of the rules for assignment in Java, the objects

can actually belong to subclasses of *Player*. Thus there could be different classes of players such as computer players, regular human players, players who are wizards, ..., all represented by different subclasses of *Player*.

As another example, suppose that a class *Shape* represents the general idea of a shape drawn on a screen, and that it has subclasses to represent specific types of shapes such as lines, rectangles, rounded rectangles, ovals, filled-in ovals, and so forth. (*Shape* itself would be an abstract class, as discussed in Subsection 5.5.5.) Then an array of type `Shape[]` can hold references to objects belonging to the subclasses of *Shape*. For example, the situation created by the statements

```
Shape[] shapes = new Shape[100]; // Array to hold up to 100
shapes.
shapes[0] = new Rect();            // Put some objects in the
array.
shapes[1] = new Line();
shapes[2] = new FilledOval();
int shapeCt = 3;  // Keep track of number of objects in array.
```

could be illustrated as:



Such an array would be useful in a drawing program. The array could be used to hold a list of shapes to be displayed. If the `Shape` class includes a method, "`void`

`redraw(Graphics g)`", for drawing the shape in a graphics context `g`, then all the shapes in the array could be redrawn with a simple for loop:

```
for (int i = 0; i < shapeCt; i++)
   shapes[i].redraw(g);
```

The statement "`shapes[i].redraw(g);`" calls the `redraw()` method belonging to the particular shape at index `i` in the array. Each object knows how to redraw itself, so that repeated executions of the statement can produce a variety of different shapes on the screen. This is nice example both of polymorphism and of array processing.

Source : http://math.hws.edu/javanotes/c7/s3.html