7 Arrays and Array Lists

Should array indices start at 0 or 1?
My compromise of 0.5 was rejected,
without, I thought, proper consideration.
—S. Kelly-Bootle

Learning Objectives

In this chapter, you will learn:

- One-dimensional arrays
- Array lists
- Two-dimensional arrays

One-Dimensional Arrays

A one-dimensional array is a data structure used to implement a list object, where the elements in the list are of the same type; for example, a class list of 25 test scores, a membership list of 100 names, or a store inventory of 500 items.

For an array of N elements in Java, index values ("subscripts") go from 0 to $N-1$. Individual elements are accessed as follows: If `arr` is the name of the array, the elements are `arr[0]`, `arr[1]`, ..., `arr[N-1]`. If a negative subscript is used, or a subscript k where $k \geq N$, an `ArrayIndexOutOfBoundsException` is thrown.

Initialization

In Java, an array is an object; therefore, the keyword `new` must be used in its creation. The one exception is an initializer list (discussed on the next page). The size of an array remains fixed once it has been created. As with `String` objects, however, an array reference may be reassigned to a new array of a different size.

Example

All of the following are equivalent. Each creates an array of 25 `double` values and assigns the reference `data` to this array.

1. `double[] data = new double[25];`
2. `double data[] = new double[25];`
3. `double[] data;`
   `data = new double[25];`

A subsequent statement like

```
data = new double[40];
```

reassigns `data` to a new array of length 40. The memory allocated for the previous `data` array is recycled by Java's automatic garbage collection system.

When arrays are declared, the elements are automatically initialized to zero for the primitive numeric data types (`int` and `double`), to `false` for boolean variables, or to `null` for object references.

It is possible to declare several arrays in a single statement. For example,

```
int[] intList1, intList2;   //declares intList1 and intList2 to
                            //contain int values
int[] arr1 = new int[15], arr2 = new int[30];  //reserves 15 slots
                                               //for arr1, 30 for arr2
```

Initializer List

Small arrays whose values are known can be conveniently declared with an *initializer list.* For example, instead of writing

```
int[] coins = new int[4];
coins[0] = 1;
coins[1] = 5;
coins[2] = 10;
coins[3] = 25;
```

you can write

```
int[] coins = {1, 5, 10, 25};
```

This construction is the one case where `new` is not required to create an array.

Length of Array

A one-dimensional array in Java has a final public instance variable (i.e., a constant), `length`, which can be accessed when you need the number of elements in the array. For example,

```
String[] names = new String[25];
< code to initialize names >

//loop to process all names in array
for (int i = 0; i < names.length; i++)
    < process names >
```

Note

1. The array subscripts go from `0` to `names.length-1`; therefore, the test on `i` in the `for` loop must be strictly less than `names.length`.
2. `length` is not a method and therefore is not followed by parentheses. Contrast this with `String` objects, where `length` *is* a method and *must* be followed by parentheses. For example,

```
String s = "Confusing syntax!";
int size = s.length();    //assigns 17 to size
```

Traversing a One-Dimensional Array

Do not use an enhanced `for` loop to remove or replace elements of an array.

Use an enhanced `for` loop whenever you need access to every element in an array without replacing or removing any elements, and without needing the position of any element. Use a `for` loop in all other cases: to access the index of any element, to replace or remove elements, or to access just some of the elements.

Note that if you have an array of objects (not primitive types), you can use the enhanced `for` loop and mutator methods of the object to modify the fields of any instance (see the `shuffleAll` method on p. 247).

Example 1

```
/** Returns the number of even integers in array arr of integers. */
public static int countEven(int[] arr)
{
    int count = 0;
    for (int num : arr)
        if ( num % 2 == 0)   //num is even
            count++;
    return count;
}
```

Example 2

```
/** Change each even-indexed element in array arr to 0.
 * Precondition: arr contains integers.
 * Postcondition: arr[0], arr[2], arr[4], ... have value 0.
 */
public static void changeEven(int[] arr)
{
    for (int i = 0; i < arr.length; i += 2)
        arr[i] = 0;
}
```

Arrays as Parameters

Since arrays are treated as objects, passing an array as a parameter means passing its object reference. No copy is made of the array. *Thus, the elements of the actual array can be accessed—and modified.*

Example 1

Array elements accessed but not modified:

```
/** Returns index of smallest element in array arr of integers. */
public static int findMin (int[] arr)
{
    int min = arr[0];
    int minIndex = 0;
    for (int i = 1; i < arr.length; i++)
        if (arr[i] < min)    //found a smaller element
        {
            min = arr[i];
            minIndex = i;
        }
    return minIndex;
}
```

To call this method (in the same class that it's defined):

```
int[] array;
< code to initialize array >
int min = findMin(array);
```

Example 2

Array elements modified:

```
/** Add 3 to each element of array b. */
public static void changeArray(int[] b)
{
    for (int i = 0; i < b.length; i++)
        b[i] += 3;
}
```

To call this method (in the same class):

```
int[] list = {1, 2, 3, 4};
changeArray(list);
System.out.print("The changed list is ");
for (int num : list)
    System.out.print(num + " ");
```

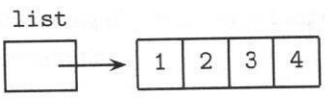
When an array is passed as a parameter, it is possible to alter the contents of the array.

The output produced is

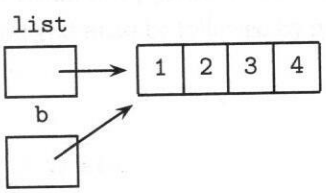
```
The changed list is 4 5 6 7
```

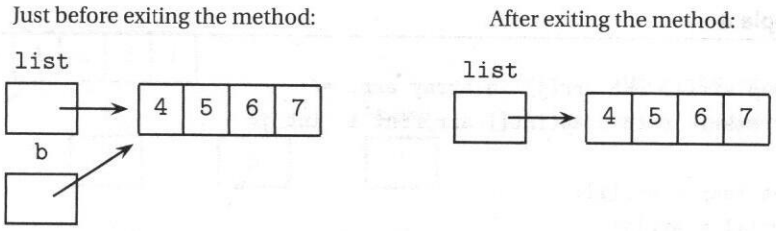
Look at the memory slots to see how this happens:

Before the method call:



At the start of the method call:





Example 3

Contrast the `changeArray` method with the following attempt to modify one array element:

```
/** Add 3 to an element. */
public static void changeElement(int n)
{ n += 3; }
```

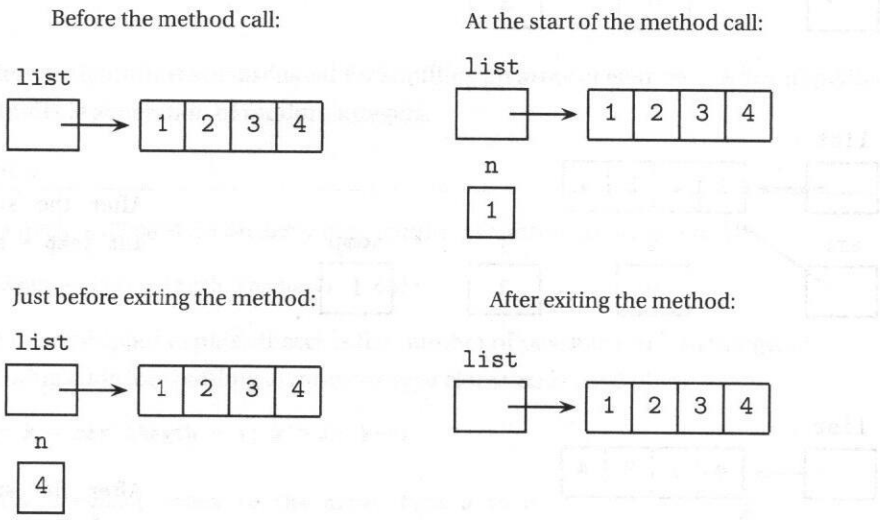
Here is some code that invokes this method:

```
int[] list = {1, 2, 3, 4};
System.out.print("Original array: ");
for (int num : list)
    System.out.print(num + " ");
changeElement(list[0]);
System.out.print("\nModified array: ");
for (int num : list)
    System.out.print(num + " ");
```

Contrary to the programmer's expectation, the output is

```
Original array: 1 2 3 4
Modified array: 1 2 3 4
```

A look at the memory slots shows why the list remains unchanged.



The point of this is that primitive types—including single array elements of type `int` or `double`—are passed by value. A copy is made of the actual parameter, and the copy is erased on exiting the method.

Example 4

```
/** Swap arr[i] and arr[j] in array arr. */
public static void swap(int[] arr, int i, int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

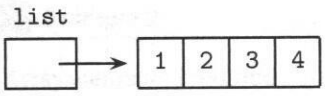
To call the swap method:

```
int[] list = {1, 2, 3, 4};
swap(list, 0, 3);
System.out.print("The changed list is: ");
for (int num : list)
    System.out.print(num + " ");
```

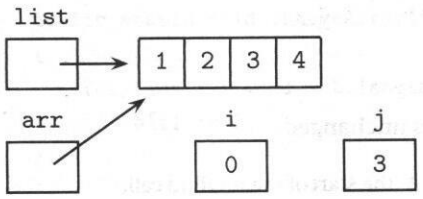
The output shows that the program worked as intended:

```
The changed list is: 4 2 3 1
```

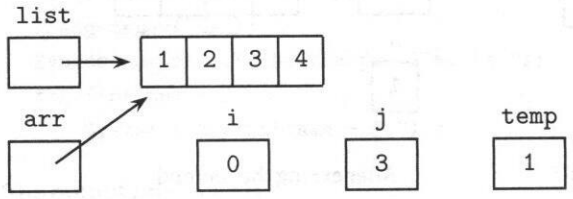
A look at the memory slots shows why the list changes.



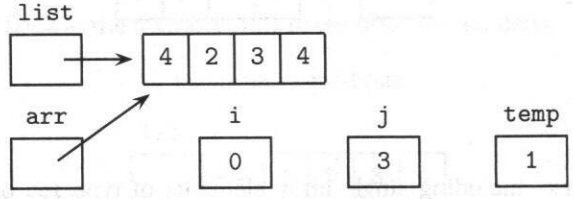
Before the method call



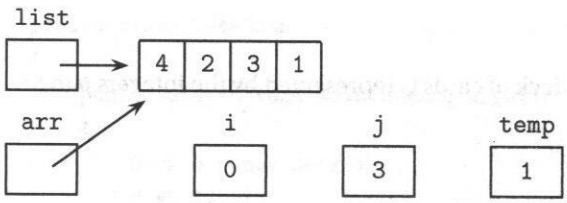
At the start of the method call



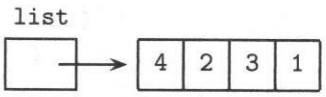
After the statement `int temp = arr[i];`



After the statement `arr[i] = arr[j];`



After the statement
`arr[j] = temp;`



After exiting the method

Example 5

```
/** Returns array containing NUM_ELEMENTS integers read from the keyboard.
 *  Precondition: Array undefined.
 *  Postcondition: Array contains NUM_ELEMENTS integers read from
 *                 the keyboard.
 */
public int[] getIntegers()
{
    int[] arr = new int[NUM_ELEMENTS];
    for (int i = 0; i < arr.length; i++)
    {
        System.out.println("Enter integer: ");
        arr[i] = ...;          //read user input
    }
    return arr;
}
```

To call this method:

```
int[] list = getIntegers();
```

Shuffling

Several different algorithms are discussed for shuffling an array of elements. A key ingredient of a good shuffle is generation of random integers.

Example 6

To shuffle a deck of 52 cards in an array may require a random `int` from 0 to 51:

```
int cardNum = (int) (Math.random() * 52);
```

(Recall that the multiplier in parentheses is the number of possible random integers.)

The following code for shuffling an array of `Type` elements is used often:

```
for (int k = arr.length - 1; k > 0; k--)
{
    //Pick a random index in the array from 0 to k
    int index = (int) (Math.random() * (k + 1));
    //Swap randomly selected element with element at position k
    Type temp = arr[k];
    arr[k] = arr[index];
    arr[index] = temp;
}
```

Array Variables in a Class

Consider a simple Deck class in which a deck of cards is represented by the integers 0 to 51.

```
public class Deck
{
    private int[] deck;
    public static final int NUMCARDS = 52;

    /** constructor */
    public Deck()
    {
        deck = new int[NUMCARDS];
        for (int i = 0; i < NUMCARDS; i++)
            deck[i] = i;
    }

    /** Write contents of Deck. */
    public void writeDeck()
    {
        for (int card : deck)
            System.out.print(card + " ");
        System.out.println();
        System.out.println();
    }

    /** Swap arr[i] and arr[j] in array arr. */
    private void swap(int[] arr, int i, int j)
    {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    /** Shuffle Deck: Generate a random permutation by picking a
     *    random card from those remaining and putting it in the
     *    next slot, starting from the right.
     */
    public void shuffle()
    {
        int index;
        for (int i = NUMCARDS - 1; i > 0; i--)
        {
            //generate an int from 0 to i
            index = (int) (Math.random() * (i + 1));
            swap(deck, i, index);
        }
    }
}
```

Here is a simple driver class that tests the Deck class:

```
public class DeckMain
{
    public static void main(String args[])
    {
        Deck d = new Deck();
        d.shuffle();
        d.writeDeck();
    }
}
```

Note

There is no evidence of the array that holds the deck of cards—deck is a private instance variable and is therefore invisible to clients of the Deck class.

Array of Class Objects

Suppose a large card tournament needs to keep track of many decks. The code to do this could be implemented with an array of Deck:

```
public class ManyDecks
{
    private Deck[] allDecks;
    public static final int NUMDECKS = 500;

    /** constructor */
    public ManyDecks()
    {
        allDecks = new Deck[NUMDECKS];
        for (int i = 0; i < NUMDECKS; i++)
            allDecks[i] = new Deck();
    }

    /** Shuffle the Decks. */
    public void shuffleAll()
    {
        for (Deck d : allDecks)
            d.shuffle();
    }

    /** Write contents of all the Decks. */
    public void printDecks()
    {
        for (Deck d : allDecks)
            d.writeDeck();
    }
}
```

Note

1. The statement

   ```
   allDecks = new Deck[NUMDECKS];
   ```

   creates an array, allDecks, of 500 Deck objects. The default initialization for these Deck objects is null. In order to initialize them with actual decks, the Deck constructor must

be called for each array element. This is achieved with the `for` loop of the `ManyDecks` constructor.

2. In the `shuffleAll` method, it's OK to use an enhanced `for` loop to modify each deck in the array with the mutator method `shuffle`.

Analyzing Array Algorithms

Example 1

Discuss the efficiency of the `countNegs` method below. What are the best and worst case configurations of the data?

```
/** Returns the number of negative values in arr.
 *  Precondition:  arr[0],...,arr[arr.length-1] contain integers.
 */
public static int countNegs(int[] arr)
{
    int count = 0;
    for (int num : arr)
        if (num < 0)
            count++;
    return count;
}
```

Solution

This algorithm sequentially examines each element in the array. In the best case, there are no negative elements, and `count++` is never executed. In the worst case, all the elements are negative, and `count++` is executed in each pass of the `for` loop.

Example 2

The code fragment below inserts a value, `num`, into its correct position in a sorted array of integers. Discuss the efficiency of the algorithm.

```
/** Precondition:
 *   - arr[0],...,arr[n-1] contain integers sorted in increasing order.
 *   - n < arr.length.
 *  Postcondition: num has been inserted in its correct position.
 */
{
    //find insertion point
    int i = 0;
    while (i < n && num > arr[i])
        i++;
    //if necessary, move elements arr[i]...arr[n-1] up 1 slot
    for (int j = n; j >= i + 1; j--)
        arr[j] = arr[j-1];
    //insert num in i-th slot and update n
    arr[i] = num;
    n++;
}
```

✓ Solution

In the best case, `num` is greater than all the elements in the array: Because it gets inserted at the end of the list, no elements must be moved to create a slot for it. The worst case has `num` less than all the elements in the array. In this case, `num` must be inserted in the first slot, `arr[0]`, and every element in the array must be moved up one position to create a slot.

This algorithm illustrates a disadvantage of arrays: Insertion and deletion of an element in an ordered list is inefficient, since, in the worst case, it may involve moving all the elements in the list.

Array Lists

An `ArrayList` provides an alternative way of storing a list of objects and has the following advantages over an array:

- An `ArrayList` shrinks and grows as needed in a program, whereas an array has a fixed length that is set when the array is created.
- In an `ArrayList` `list`, the last slot is always `list.size()-1`, whereas in a partially filled array, you, the programmer, must keep track of the last slot currently in use.
- For an `ArrayList`, you can do insertion or deletion with just a single statement. Any shifting of elements is handled automatically. In an array, however, insertion or deletion requires you to write the code that shifts the elements.
- It is easier to print the elements of an `ArrayList` than those of an array. For an `ArrayList` `list` and an array `arr`, the statement

```
System.out.print(list);
```

will output the elements of `list`, nicely formatted in square brackets, with the elements separated by commas. However, to print the elements of `arr`, an explicit piece of code that accesses and prints each element is needed. The statement

```
System.out.print(arr);
```

will produce weird output that includes an `@` symbol—not the elements of the array.

The `ArrayList` Class

The `ArrayList` class is part of the `java.util` package. An `import` statement makes this class available in a program. Java allows the generic type `ArrayList<E>`, where `E` is the type of the elements in the `ArrayList`. When a generic class is declared, the type parameter is replaced by an actual object type. For example,

```
private ArrayList<Clown> clowns;
```

Note

The `clowns` list must contain only `Clown` objects. An attempt to add an `Acrobat` to the list, for example, will cause a compile-time error.

The Methods of ArrayList<E>

Here are the methods you should know:

`ArrayList()`

Constructor constructs an empty list.

`int size()`

Returns the number of elements in the list.

`boolean add(E obj)`

Appends `obj` to the end of the list. Always returns `true`. If the specified element is not of type `E`, throws a run-time exception.

`void add(int index, E element)`

Inserts `element` at specified `index`. Elements from position `index` and higher have 1 added to their indices. Size of list is incremented by 1.

`E get(int index)`

Returns the element at the specified `index` in the list.

`E set(int index, E element)`

Replaces item at specified `index` in the list with specified `element`. Returns the element that was previously at `index`. If the specified element is not of type `E`, throws a run-time exception.

`E remove(int index)`

Removes and returns the element at the specified `index`. Elements to the right of position `index` have 1 subtracted from their indices. Size of list is decreased by 1.

Note

Each of these methods that has an `index` parameter—`add`, `get`, `remove`, and `set`—throws an `IndexOutOfBoundsException` if `index` is out of range. For `get`, `remove`, and `set`, `index` is out of range if

```
index < 0 || index >= size()
```

For `add`, however, it is OK to add an element at the end of the list. Therefore `index` is out of range if

```
index < 0 || index > size()
```

Autoboxing and Unboxing

An `ArrayList` cannot contain a primitive type like `double` or `int`: It must only contain *objects*. (It actually contains the references to those objects.) Numbers must therefore be boxed—placed in wrapper classes like `Integer` and `Double`—before insertion into an `ArrayList`.

Autoboxing is the automatic wrapping of primitive types in their wrapper classes (see p. 180).

To retrieve the numerical value of an `Integer` (or `Double`) stored in an `ArrayList`, the `intValue()` (or `doubleValue()`) method must be invoked (unwrapping). *Unboxing* is the automatic conversion of a wrapper class to its corresponding primitive type. This means that you don't need to explicitly call the `intValue()` or `doubleValue()` methods. Be aware that if a program tries to auto-unbox `null`, the method will throw a `NullPointerException`.

Note that while autoboxing and unboxing cut down on code clutter, these operations must still be performed behind the scenes, leading to decreased run-time efficiency. It is much more efficient to assign and access primitive types in an array than an `ArrayList`. You should therefore consider using an array for a program that manipulates sequences of numbers and does not need to use objects.

Note

1. Autoboxing and unboxing are part of the AP Java subset.
2. The `List<E>` interface and the methods of `List<E>` are no longer part of the AP Java subset.

Using `ArrayList<E>`

Example 1

```
//Create an ArrayList containing 0 1 4 9.
ArrayList<Integer> list = new ArrayList<Integer>();
for (int i = 0; i < 4; i++)
    list.add(i * i);  //example of autoboxing
                      //i*i wrapped in an Integer before insertion

Integer intOb = list.get(2); //assigns Integer with value 4 to intOb.
                             //Leaves list unchanged.

int n = list.get(3);  //example of auto-unboxing
                      //Integer is retrieved and converted to int
                      //n contains 9

Integer x = list.set(3, 5);  //list is 0 1 4 5
                             //x contains Integer with value 9

x = list.remove(2);          //list is 0 1 5
                             //x contains Integer with value 4

list.add(1, 7);              //list is 0 7 1 5

list.add(2, 8);              //list is 0 7 8 1 5
```

Example 2

```
/** Swap two values in list, indexed at i and j. */
public static void swap(ArrayList<E> list, int i, int j)
{
    E temp = list.get(i);
    list.set(i, list.get(j));
    list.set(j, temp);
}
```

Example 3

```
/** Returns an ArrayList of random integers from 0 to 100. */
public static ArrayList<Integer> getRandomIntList()
{
    ArrayList<Integer> list = new ArrayList<Integer>();
    System.out.print("How many integers? ");
    int length = ...;      //read user input
    for (int i = 0; i < length; i++)
    {
        int newNum = (int) (Math.random() * 101);
        list.add(newNum);   //autoboxing
    }
    return list;
}
```

Traversing an ArrayList

To traverse an `ArrayList` means to access all of the elements of the list using an iteration statement (`for` loop, `while` loop, or enhanced `for` loop).

Here are several examples to illustrate different types of traversals.

For simple accessing—for example, printing each element or adding each element to a running total, etc.—an enhanced `for` loop is a convenient method of traversal.

Example 4

```
/** Print all negatives in list.
 *  Precondition: list contains Integer values.
 */
public static void printNegs(ArrayList<Integer> list)
{
    System.out.println("The negative values in the list are: ");
    for (Integer i : list)
        if (i < 0)          //auto-unboxing
            System.out.println(i);
}
```

Note

Here's how to think of this algorithm: For each `Integer i` in `ArrayList list`, create a local copy of the element, test if it's negative, and print it if negative.

To access an element with a specific index—for example, to replace the element at that index, or insert an element at that index—use an index traversal. Since the indices for an `ArrayList` start at `0` and end at `list.size()-1`, trying to access an element with an index value outside of this range will cause an `IndexOutOfBoundsException` to be thrown.

Example 5

```
/** Precondition: ArrayList list contains Integer values sorted in increasing order.
 *  Postcondition: value inserted in its correct position in list.
 */
public static void insert(ArrayList<Integer> list, Integer value)
{
    int index = 0;
    //find insertion point
    while (index < list.size() &&
           value > list.get(index)      //unboxing
        index++;
    //insert value
    list.add(index, value);
}
```

Note

Suppose `value` is larger than all the elements in `list`. Then the `insert` method will throw an `IndexOutOfBoundsException` if the first part of the test is omitted, that is, `index < list.size()`.

Example 6

```
/** Change every even-indexed element of strList to the empty string.
 *  Precondition: strList contains String values.
 */
public static void changeEvenToEmpty(ArrayList<String> strList)
{
    boolean even = true;
    int index = 0;
    while (index < strList.size())
    {
        if (even)
            strList.set(index, "");
        index++;
        even = !even;
    }
}
```

Note

Deleting elements during the traversal of an `ArrayList` requires special care to avoid skipping elements.

Example 7

```
/* Remove all occurrences of value from list. */
public static void removeAll(ArrayList<Integer> list, int value)
{
    int index = 0;
    while (index < list.size())
    {
        if (list.get(index) == value)
            list.remove(index);
        else
            index++;
    }
}
```

Note

1. The statement

   ```
   list.remove(index);
   ```

   causes the elements to the right of the removed element to be shifted left to fill the "hole." In this case, if `index` is incremented, the current element will be skipped, and if two consecutive elements are equal to `value`, one will be missed and (mistakenly) remain in the list.
2. Trying to add or delete an element during a traversal with an enhanced `for` loop may result in a `ConcurrentModificationException` being thrown. Therefore, if you want to add or delete elements, don't use an enhanced `for` loop to traverse the `ArrayList`.

Example 8

```
ArrayList<Integer> list = new ArrayList<Integer>();
< code to initialize list >

for (Integer num : list)
{
    if (num < 0)
        list.add(0);       //WRONG!
}
```

Note

This code segment throws a `ConcurrentModificationException`.

It is okay, however, to use an enhanced `for` loop to *modify* objects that have a mutator method in their class definition.

Example 9

Consider a `Clown` class that has a `changeAct` method, and an `ArrayList<Clown>` that has been initialized with `Clown` objects. The following code is fine.

```
for (Clown c : clownList)
{
    if ( <some condition on Clown c > )
        clownList.changeAct();
}
```

Random Element Selection

You should know how to return a random element from an array or `ArrayList`.

Example 10

Suppose `responses` is an `ArrayList<String>` of surprised responses the computer may make to a user's crazy input. If the contents of `responses` are currently

0	1	2	3	4	5
Oh my!	Say what?	No!	Heavens!	You're kidding me.	Jumping Jellybeans!

you should be able to randomly return one of these responses. The key is to select a random index from 0 to 5, inclusive, and then return the string in the `responses` list that is at that index.

Recall that the expression `(int)(Math.random()*howMany)` generates a random `int` in the range `0...howMany-1`. In the given example, `howMany` is 6, the number of elements in the array. The piece of code that returns a random response is:

```
int randIndex = (int) (Math.random() * 6);
String response = responses.get(randIndex);
```

Simultaneous Traversal of an `ArrayList` and an Array

In the traversal of a list, if it's important to keep track of indices (positions) in the list, you must use an index traversal. Sometimes an algorithm requires the simultaneous traversal of an array and an `ArrayList`. Try your hand at writing code for the following problems.

Example 11

Consider an `ArrayList<Integer> list`, and an array `arr` of `int` that have both been initialized. A method `getProductSum` returns the sum of products of the values of corresponding elements. Thus, `prodArr[0]` will be the product of `arr[0]` and the first `Integer` value in `list`; `prodArr[1]` will be the product of `arr[1]` and the second `Integer` value in `list`; and so on. The algorithm stops when the end of the shorter list is reached.

Here are some examples:

list	arr	getProductSum
[2,1,4]	{5,0,3}	22
[1,3,5,7,9]	{2,4}	14
[7,6,5]	{1,2,3,4,5}	34
[]	{2,3,7}	0

The method `getProductSum`, whose header is given below, returns the sum of products as described above. Write code for the method.

```
public static int getProductSum(ArrayList<Integer> list, int[] arr)
```

Solution

```
public static int getProductSum(ArrayList<Integer> list, int[] arr)
{
    int sum = 0;
    int index = 0;

    //Traverse both arr and list, until the end of
    //one of the lists is reached.
    while(index < arr.length && index < list.size())
    {
        sum += arr[index] * list.get(index);      //auto-unboxing;
        index++;
    }
    return sum;
}
```

Note

Beware of going off the end of either list!

Example 12

Here is a trickier example.

Consider an `ArrayList<Integer>` `list` and an array `arr` of `int` that have both been initialized. An array called `productArr` is to be created that contains the products of the values of corresponding elements. Thus, `prodArr[0]` will be the product of `arr[0]` and the first `Integer` value in `list`; `prodArr[1]` will be the product of `arr[1]` and the second `Integer` value in `list`; and so on. When the end of the shorter list is reached, the algorithm should copy the remaining elements of the longer list into `productArr`.

Here are some examples:

list	arr	productArr
[2,1,4]	{5,0,3}	{10, 0, 12}
[1,3,5,7,9]	{2,4}	{2,12,5,7,9}
[7,6,5]	{1,2,3,4,5}	{7,12,15,4,5}
[]	{2,3,7}	{2,3,7}

The method `getProducts`, whose header is given below, returns an array of products as described above. Write code for the method.

```
public static int[] getProducts(ArrayList<Integer> list, int[] arr)
```

Solution

```
public static int[] getProducts(ArrayList<Integer> list, int[] arr)
{
    int prodArrSize, smallerCount;
    boolean arrayIsLonger;
    //Determine length of prodArray.
    if (list.size() < arr.length)
    {
        prodArrSize = arr.length;
        smallerCount = list.size();
        arrayIsLonger = true;
    }
    else
    {
        prodArrSize = list.size();
        smallerCount = arr.length;
        arrayIsLonger = false;
    }
    int [] prodArray = new int[prodArrSize];
    //Place all products in prodArray.
    for (int i = 0; i < smallerCount; i++)
        prodArray[i] = arr[i] * list.get(i);
    //How many elements must be transferred to prodArray?
    int numExtra = Math.abs(arr.length - list.size());
    //Transfer those final elements to prodArray.
    for (int i = 0; i <= numExtra - 1; i++)
    {
        if (arrayIsLonger)
            prodArray[prodArrSize - numExtra + i] =
                    arr[prodArrSize - numExtra + i];
        else
            prodArray[prodArrSize - numExtra + i] =
                    list.get(prodArrSize - numExtra + i);
    }
    return prodArray;
}
```

Note

1. Use `Math.abs` to get a positive value for the number of extra elements to be copied.
2. `prodArray` already has slots for the leftover elements that must be copied. But you must be careful in the indexes for these elements that are taken from the end of the longer list and placed in the end slots of the `prodArray`.

Two-Dimensional Arrays

A two-dimensional array (matrix) is often the data structure of choice for objects like board games, tables of values, theater seats, and mazes.

Look at the following 3 × 4 matrix:

2	6	8	7
1	5	4	0
9	3	2	8

If `mat` is the matrix variable, the row subscripts go from `0` to `2` and the column subscripts go from `0` to `3`. The element `mat[1][2]` is 4, whereas `mat[0][2]` and `mat[2][3]` are both 8. As with one-dimensional arrays, if the subscripts are out of range, an `ArrayIndexOutOfBoundsException` is thrown.

Declarations

Each of the following declares a two-dimensional array:

```
int[][] table;      //table can reference a 2D array of integers
                    //table is currently a null reference
double[][] matrix = new double[3][4];  //matrix references a 3 × 4
                                       //array of real numbers.
                                       //Each element has value 0.0
String[][] strs = new String[2][5]; //strs references a 2 × 5
                                    //array of String objects.
                                    //Each element is null
```

An *initializer list* can be used to specify a two-dimensional array:

```
int[][] mat = { {3, 4, 5},          //row 0
                {6, 7, 8} };        //row 1
```

This defines a 2 × 3 *rectangular* array (i.e., one in which each row has the same number of elements). All two-dimensional arrays on the AP exam are rectangular.

The initializer list is a list of lists in which each inside list represents a row of the matrix.

Matrix as Array of Row Arrays

A matrix is implemented as an array of rows, where each row is a one-dimensional array of elements. Suppose `mat` is the 3 × 4 matrix

```
2 6 8 7
1 5 4 0
9 3 2 8
```

Then `mat` is an array of three arrays:

`mat[0]`	contains	`{2, 6, 8, 7}`
`mat[1]`	contains	`{1, 5, 4, 0}`
`mat[2]`	contains	`{9, 3, 2, 8}`

The quantity `mat.length` represents the number of rows. In this case it equals 3 because there are three row arrays in `mat`. For any given row `k`, where $0 \le k < $ `mat.length`, the quantity `mat[k].length` represents the number of elements in that row—namely, the number of columns. (Java allows a variable number of elements in each row. Since these "jagged arrays" are not part of the AP Java subset, you can assume that `mat[k].length` is the same for all rows `k` of the matrix, i.e., that the matrix is rectangular.)

Processing a Two-Dimensional Array

There are three common ways to traverse a two-dimensional array:

- row-column (for accessing elements, modifying elements that are class objects, or replacing elements)
- enhanced `for` loop (for accessing elements or modifying elements that are class objects, but no replacement)
- row-by-row array processing (for accessing, modifying, or replacement of elements)

Note

A row-by-row traversal, starting in the top, leftmost corner and going from left to right is called a *row-major traversal*:

```
for (row = 0; row < mat.length; row++)
    for (col = 0; col < mat[0].length; col++)
        processElements();
```

A column-by-column traversal, starting in the top, leftmost corner and going from top to bottom is less common and is called a *column-major traversal*:

```
for (col = 0; col < mat[0].length; col++)
    for (row = 0; row < mat.length; row++)
        processElements();
```

Example 1

Find the sum of all elements in a matrix `mat`.

Solution

Here is a row-column traversal:

```
/** Precondition: mat is initialized with integer values. */
int sum = 0;
for (int r = 0; r < mat.length; r++)
    for (int c = 0; c < mat[r].length; c++)
        sum += mat[r][c];
```

Note

1. `mat[r][c]` represents the `r`th row and the `c`th column.
2. Rows are numbered from `0` to `mat.length-1`, and columns are numbered from `0` to `mat[r].length-1`. Any index that is outside these bounds will generate an `ArrayIndexOutOfBoundsException`.

Since elements are not being replaced, nested enhanced `for` loops can be used instead:

```
for (int[] row : mat)          //for each row array in mat
    for (int element : row)    //for each element in this row
        sum += element;
```

Note

You also need to know how to process a matrix as shown below, using a third type of traversal, row-by-row array processing. This traversal

- assumes access to a method that processes an array;
- passes a one-dimensional array reference as a parameter to a method that processes each row;
- traverses the rows using either a regular loop or an enhanced `for` loop.

So, continuing with the example to find the sum of all elements in `mat`: In the class where `mat` is defined, suppose you have the method `sumArray`.

```
/** Returns the sum of integers in arr. */
public int sumArray(int[] arr)
{ /* implementation not shown */ }
```

You could use this method to sum all the elements in `mat` as follows:

```
int sum = 0;
for (int row = 0; row < mat.length; row++) //for each row in mat,
    sum += sumArray(mat[row]);             //add that row's total to sum
```

Note how, since `mat[row]` is an array of `int` for $0 \le$ `row` $<$ `mat.length`, you can use the `sumArray` method for each row in `mat`. Alternatively, you can use an enhanced `for` loop traversal:

```
for (int [] rowArr: mat)          //for each row array in mat
    sum += sumArray(rowArr);      //add that row's total to sum
```

Example 2

Add 10 to each element in row 2 of matrix `mat`.

Solution

```
for (int c = 0; c < mat[2].length; c++)
    mat[2][c] += 10;
```

Note

1. In the `for` loop, you can use `c < mat[k].length`, where $0 \le$ `k` $<$ `mat.length`, since each row has the same number of elements.
2. You should not use an enhanced `for` loop here because elements are being replaced.
3. You can, however, use row-by-row array processing. Suppose you have method `addTen` shown below.

```
/** Add 10 to each int in arr */
public void addTen(int[] arr)
{
    for (int i = 0; i < arr.length; i++)
        arr[i] += 10;
}
```

You could add `10` to each element in row `2` with the single statement,

```
addTen(mat[2]);
```

You could also add 10 to every element in `mat`:

```
for (int row = 0; row < mat.length; row++)
    addTen(mat[row]);
```

Example 3

Suppose `Card` objects have a mutator method `changeValue`:

```
public void changeValue(int newValue)
{ value = newValue; }
```

Now consider the following declaration.

```
Card[][] cardMatrix;
```

Suppose `cardMatrix` is initialized with `Card` objects. A piece of code that traverses the `cardMatrix` and changes the value of each `Card` to `v` is

```
for (Card[] row : cardMatrix)  //for each row array in cardMatrix,
    for (Card c : row)         //for each Card in that row,
        c.changeValue(v);      //change the value of that card
```

Alternatively:

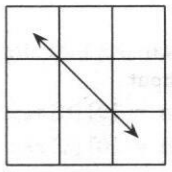
```
for (int row = 0; row < cardMatrix.length; row++)
    for (int col = 0; col < cardMatrix[0].length; col++)
        cardMatrix[row][col].changeValue(v);
```

Note

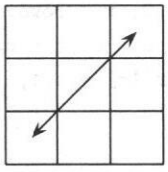
The use of the nested enhanced `for` loop is OK. Modifying the objects in the matrix with a mutator method is fine. What you shouldn't do is *replace* the `Card` objects with new `Card`s.

Example 4

The major and minor diagonals of a square matrix are shown below:



Major diagonal



Minor diagonal

You can process the diagonals as follows:

```
int[][] mat = new int[SIZE][SIZE];  //SIZE is a constant int value
for (int i = 0; i < SIZE; i++)
    Process mat[i][i];              //major diagonal
      OR
    Process mat[i][SIZE - i - 1];   //minor diagonal
```

Two-Dimensional Array as Parameter

Example 1

Here is a method that counts the number of negative values in a matrix:

```
/** Returns count of negative values in mat.
 *  Precondition: mat is initialized with integers.
 */
public static int countNegs (int[][] mat)
{
    int count = 0;
    for (int[] row : mat)
        for (int num : row)
            if (num < 0)
                count++;
    return count;
}
```

A method in the same class can invoke this method with a statement such as

```
int negs = countNegs(matrix);
```

Example 2

Reading elements into a matrix:

```
/** Returns matrix containing rows × cols integers
 *  read from the keyboard.
 *  Precondition: Number of rows and columns known.
 */
public static int[][] getMatrix(int rows, int cols)
{
    int[][] mat = new int[rows][cols];   //initialize slots
    System.out.println("Enter matrix, one row per line:");
    System.out.println();

    //read user input and fill slots
    for (int r = 0; r < rows; r++)
        for (int c = 0; c < cols; c++)
            mat[r][c] = ...;   //read user input
    return mat;
}
```

To call this method:

```
//prompt for number of rows and columns
int rows = ...;       //read user input
int cols = ...;       //read user input
int[][] mat = getMatrix(rows, cols);
```

Note

You should not use an enhanced `for` loop in `getMatrix` because elements in `mat` are being replaced. (Their current value is the initialized value of `0`. The new value is the input value from the keyboard.)

Mirror Images

A tricky multiple-choice question on two-dimensional arrays may involve an algorithm that creates some kind of mirror image of a matrix. The code could reflect a matrix across mirrors placed somewhere in the center of the matrix, horizontally, vertically, or diagonally.

Note that if a vertical mirror is placed down the center of a matrix, so that all elements to the left of the mirror are reflected across it, the element `mat[row][col]` reflects across to element `mat[row][numCols-col-1]`.

You should teach yourself to trace the following type of code:

```
public static void matrixMethod(int[][] mat)
{
    int height = mat.length;
    int numCols = mat[0].length;
    for (int col = 0; col < numCols; col++)
        for (int row = 0; row < height/2; row++)
            mat[height - row - 1][col] = mat[row][col];
}
```

What does it do? How does it transform the matrix below?

```
2 3 4
5 6 7
8 9 0
1 1 1
```

The answer is that the algorithm reflects the matrix from top to bottom across a horizontal mirror placed at its center.

```
height = 4, numCols = 3
col takes on values 0, 1, and 2
row takes on values 0 and 1
```

Here are the replacements that are made:

```
col = 0, row = 0: mat[3][0] = mat[0][0]
         row = 1: mat[2][0] = mat[1][0]

col = 1, row = 0: mat[3][1] = mat[0][1]
         row = 1: mat[2][1] = mat[1][1]

col = 2, row = 0: mat[3][2] = mat[0][2]
         row = 1: mat[2][2] = mat[1][2]
```

This transforms the matrix into:

```
2 3 4
5 6 7
5 6 7
2 3 4
```

Note that an enhanced `for` loop was not used in the traversal, because elements in the matrix are being replaced.

Chapter Summary

Manipulation of one-dimensional arrays, two-dimensional arrays, and array lists should be second nature to you by now. Know the Java subset methods for the `ArrayList<E>` class. You must also know when these methods throw an `IndexOutOfBoundsException` and when an `ArrayIndexOutOfBoundsException` can occur.

When traversing an `ArrayList`:

- Use an enhanced `for` loop to access each element without changing it, or to modify each object in the list using a mutator method.
- Take special care with indices when removing multiple elements from an `ArrayList`.

A two-dimensional array is an array of row arrays. The number of rows is `mat.length`. The number of columns is `mat[0].length`.

When traversing a two-dimensional array:

- Use a row-column traversal to access, modify, or replace elements. Know the difference between row-major order and column-major order.
- Use a nested `for` loop to access or modify elements, but not replace them.
- Know how to do row-by-row array processing if you have an appropriate method that takes an array parameter.