Arrays and Array Lists

CHAPTER 6

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

Chapter Goals

- One-dimensional arrays
- The `ArrayList<E>` class
- Two-dimensional arrays
- The `List<E>` interface

ONE-DIMENSIONAL ARRAYS

An 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 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];
```

YOU NEED TO USE new

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 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 Java array 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 an Array

Use a for-each loop whenever you need access to every element in an array without replacing or removing any elements. 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.

Do not use a for-each loop to remove or replace elements of an array.

Note that if you have an array of objects (not primitive types), you can use the for-each loop and mutator methods of the object to modify the fields of any instance (see the `shuffleAll` method on p. 239).

Example 1

```
/** @return 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;
}
```

Important

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:

```
/** @return 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):

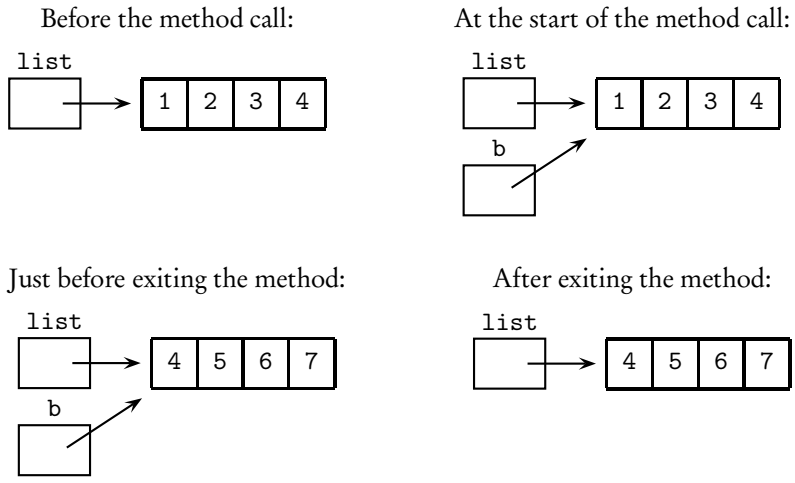
```
int[] list = {1, 2, 3, 4};
changeArray(list);
System.out.print("The changed list is ");
for (int num : list)
    System.out.print(num + " ");
```

The output produced is

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

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

Look at the memory slots to see how this happens:



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; }
```

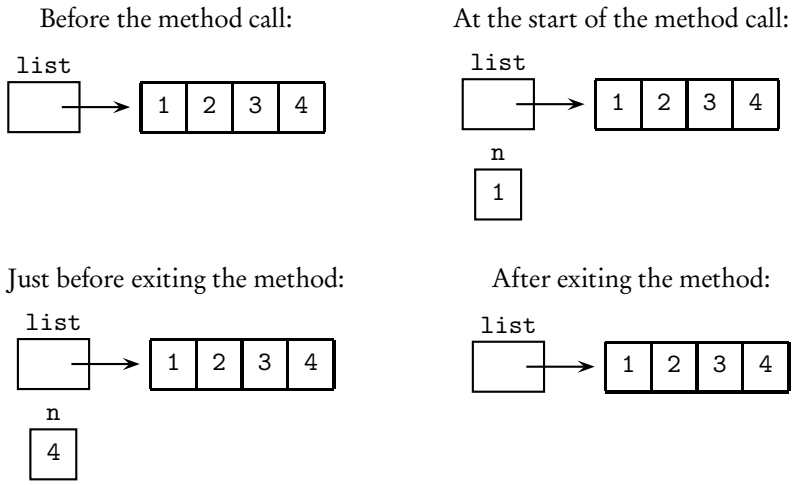
Here is some code that invokes this method:

```
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
```

Example 5

```
/** @return 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] = IO.readInt();         //read user input
    }
    return arr;
}
```

To call this method:

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

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 a for-each 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?

```
/** Precondition:  arr[0],...,arr[arr.length-1] contain integers.
 *  @return the number of negative values in arr
 */
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

Great

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.

Strings:
n.length();
Arrays:
n.length;
Array Lists:
n.size();

The Collections API

The `ArrayList` class is in the Collections API (Application Programming Interface), which is a library provided by Java. Most of the API is in `java.util`. This library gives the programmer access to prepackaged data structures and the methods to manipulate them. The implementations of these *container classes* are invisible and should not be of concern to the programmer. The code works. And it is reusable.

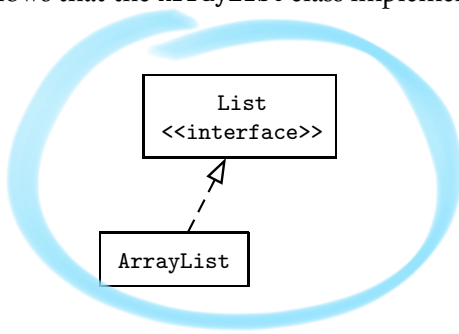
All of the collections classes, including `ArrayList`, have the following features in common:

- They are designed to be both memory and run-time efficient.
- They provide methods for insertion and removal of items (i.e., they can grow and shrink).
- They provide for iteration over the entire collection.

The Collections Hierarchy

Inheritance is a defining feature of the Collections API. The interfaces that are used to manipulate the collections specify the operations that must be defined for any container class that implements that interface.

The diagram below shows that the `ArrayList` class implements the `List` interface.



Collections and Generics

The collections classes are generic, with type parameters. Thus, `List<E>` and `ArrayList<E>` contain elements of type `E`.

When a generic class is declared, the type parameter is replaced by an actual object type. For example,

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

NOTE

1. 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.
2. Since the type of objects in a generic class is restricted, the elements can be accessed without casting.
3. All of the type information in a program with generic classes is examined at compile time. After compilation the type information is erased. This feature of generic classes is known as *erasure*. During execution of the program, any attempt at incorrect casting will lead to a `ClassCastException`.

Auto-Boxing and -Unboxing

There are no primitive types in collections classes. An `ArrayList` must contain *objects*, not types like `double` and `int`. Numbers must therefore be boxed—placed in wrapper classes like `Integer` and `Double`—before insertion into an `ArrayList`.

Auto-boxing is the automatic wrapping of primitive types in their wrapper classes.

To retrieve the numerical value of an `Integer` (or `Double`) stored in an `ArrayList`, the `intValue()` (or `doubleValue()`) method must be invoked (unwrapping). *Auto-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 auto-boxing 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

Auto-boxing and -unboxing is a feature in Java 5.0 and later versions and will not be tested on the AP exam. It is OK, however, to use this convenient feature in code that you write in the free-response questions.

THE List<E> INTERFACE

A class that implements the List<E> interface—ArrayList<E>, for example—is a list of elements of type E. In a list, duplicate elements are allowed. The elements of the list are indexed, with 0 being the index of the first element.

A list allows you to

- Access an element at any position in the list using its integer index.
- Insert an element anywhere in the list.
- Iterate over all elements using ListIterator or Iterator (not in the AP subset).

The Methods of List<E>

Here are the methods you should know.

```
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 ClassCastException.

```
int size()
```

Returns the number of elements in the list.

```
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. Throws a ClassCastException if the specified element is not of type E.

```
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 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.

Optional topic

```
Iterator<E> iterator()
```

Returns an iterator over the elements in the list, in proper sequence, starting at the first element.

The `ArrayList<E>` Class

This is an array implementation of the `List<E>` interface. The main difference between an array and an `ArrayList` is that an `ArrayList` is resizable during run time, whereas an array has a fixed size at construction.

Shifting of elements, if any, caused by insertion or deletion, is handled automatically by `ArrayList`. Operations to insert or delete at the end of the list are very efficient. Be aware, however, that at some point there will be a resizing; but, on average, over time, an insertion at the end of the list is a single, quick operation. In general, insertion or deletion in the middle of an `ArrayList` requires elements to be shifted to accommodate a new element (`add`), or to close a "hole" (`remove`).

THE METHODS OF `ArrayList<E>`

In addition to the two `add` methods, and `size`, `get`, `set`, and `remove`, you must know the following constructor.

```
ArrayList()
```

Constructs an empty list.

NOTE

Each method above 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()
```

Using ArrayList<E>

Example 1

```
//Create an ArrayList containing 0 1 4 9.
List<Integer> list = new ArrayList<Integer>(); //An ArrayList is-a List
for (int i = 0; i < 4; i++)
    list.add(i * i);  //example of auto-boxing
                      //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

```
//Traversing an ArrayList of Integer.
//Print the elements of list, one per line.
for (Integer num : list)
    System.out.println(num);
```

Example 3

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

Example 4

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

NOTE

1. The variable `list` is declared to be of type `List<Integer>` (the interface) but is instantiated as type `ArrayList<Integer>` (the implementation).
2. The `add` method in `getRandomIntList` is the `List` method that appends its parameter to the end of the list.

Example 5

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

Example 6

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

Example 7

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

Optional topic

COLLECTIONS AND ITERATORS

Definition of an Iterator

An *iterator* is an object whose sole purpose is to traverse a collection, one element at a time. During iteration, the iterator object maintains a current position in the collection, and is the controlling object in manipulating the elements of the collection.

The `Iterator<E>` Interface

The package `java.util` provides a generic interface, `Iterator<E>`, whose methods are `hasNext`, `next`, and `remove`. The Java Collections API allows iteration over each of its collections classes.

THE METHODS OF `Iterator<E>`

`boolean hasNext()`

Returns `true` if there's at least one more element to be examined, `false` otherwise.

`E next()`

Returns the next element in the iteration. If no elements remain, the method throws a `NoSuchElementException`.

`void remove()`

Deletes from the collection the last element that was returned by `next`. This method can be called only once per call to `next`. It throws an `IllegalStateException` if the `next` method has not yet been called, or if the `remove` method has already been called after the last call to `next`.

Using a Generic Iterator

To iterate over a parameterized collection, you must use a parameterized iterator whose parameter is the same type.

Example 1

```
List<String> list = new ArrayList<String>();
< code to initialize list with strings>
//Print strings in list, one per line.
Iterator<String> itr = list.iterator();
while (itr.hasNext())
    System.out.println(itr.next());
```

NOTE

1. Only classes that allow iteration can use the for-each loop. This is because the loop operates by using an iterator. Thus, the loop in the above example is equivalent to

(continued)

```
for (String str : list)    //no iterator in sight!
    System.out.println(str);
```

2. Recall, however, that a for-each loop cannot be used to remove elements from the list. The easiest way to "remove all occurrences of . . . " from an `ArrayList` is to use an iterator.

Example 2

```
/** Remove all 2-character strings from strList.
 *  Precondition:  strList initialized with String objects.
 */
public static void removeTwos(List<String> strList)
{
    Iterator<String> itr = strList.iterator();
    while (itr.hasNext())
        if (itr.next().length() == 2)
            itr.remove();
}
```

Example 3

```
/** Assume a list of integer strings.
 *  Remove all occurrences of "6" from the list.
 */
Iterator<String> itr = list.iterator();
while (itr.hasNext())
{
    String num = itr.next();
    if (num.equals("6"))
    {
        itr.remove();
        System.out.println(list);
    }
}
```

If the original list is `2 6 6 3 5 6` the output will be

```
[2, 6, 3, 5, 6]
[2, 3, 5, 6]
[2, 3, 5]
```

Example 4

```
/** Illustrate NoSuchElementException. */
Iterator<SomeType> itr = list.iterator();
while (true)
    System.out.println(itr.next());
```

The list elements will be printed, one per line. Then an attempt will be made to move past the end of the list, causing a `NoSuchElementException` to be thrown. The loop can be corrected by replacing `true` with `itr.hasNext()`.

(continued)

Example 5

```
/** Illustrate IllegalStateException. */
Iterator<SomeType> itr = list.iterator();
SomeType ob = itr.next();
itr.remove();
itr.remove();
```

Every `remove` call must be preceded by a `next`. The second `itr.remove()` statement will therefore cause an `IllegalStateException` to be thrown.

NOTE

In a given program, the declaration

```
Iterator<SomeType> itr = list.iterator();
```

must be made every time you need to initialize the iterator to the beginning of the list.

Example 6

```
/** Remove all negatives from intList.
 *  Precondition:  intList contains Integer objects.
 */
public static void removeNegs(List<Integer> intList)
{
    Iterator<Integer> itr = intList.iterator();
    while (itr.hasNext())
        if (itr.next().intValue() < 0)
            itr.remove();
}
```

NOTE

Every call to `remove` must be preceded by `next`.

1. In Example 6 on p. 246 a for-each loop is used because each element is accessed without changing the list. An iterator operates unseen in the background. Contrast this with Example 6 above, where the list is changed by removing elements. Here you cannot use a for-each loop.
2. To test for a negative value, you could use

   ```
   if (itr.next() < 0)
   ```

   because of auto-unboxing.
3. Use a for-each loop for accessing and modifying objects in a list. Use an iterator for removal of objects.

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:

$$\begin{matrix} 2 & 6 & 8 & 7 \\ 1 & 5 & 4 & 0 \\ 9 & 3 & 2 & 8 \end{matrix}$$

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 2-D 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).

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 $0 \le k < \text{mat.length}$, 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)
- for-each loop (for accessing elements or modifying elements that are class objects, but no replacement)
- row-by-row array processing (for accessing, modifying, or replacement)

Example 1

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

Starting in 2015, you will need to know how to use a nested for-each traversal. You will also need to know how to process a matrix as shown below, using the third type of traversal, row-by-row array processing. This traversal assumes access to a method that processes an array. 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`.

```
/** @return 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 $0 \leq$ `row` $<$ `mat.length`, you can use the `sumArray` method for each row in `mat`.

Example 2

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

```
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 $0 \leq k < mat.length$, since each row has the same number of elements.
2. You cannot use a for-each 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 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:

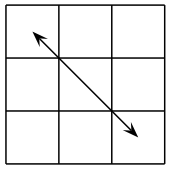
```
for (int row = 0; row < cardMatrix.length; row++)
    for (int col = 0; col < cardMatrix[0].length; col++)
        cardMatrix[row][col].changeValue(v);
```

NOTE

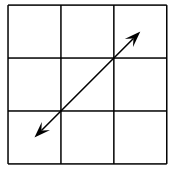
The use of the nested for-each loop is OK. Modifying the objects in the matrix with a mutator method is fine. What you can't do is *replace* the `Card` objects with new `Card`s.

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.

```
/** Precondition:  mat is initialized with integers.
 *  @return count of negative values in mat
 */
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(mat);
```

Example 2

Reading elements into a matrix:

```
/** Precondition:  Number of rows and columns known.
 *  @return matrix containing rows × cols integers
 *  read from the keyboard
 */
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] = IO.readInt();   //read user input
    return mat;
}
```

To call this method:

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

NOTE

You cannot use a for-each 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.)

There is further discussion of arrays and matrices, plus additional questions, in Chapter 9 (The AP Computer Science Labs).

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 `List<E>` class. You must also know when these methods throw an `IndexOutOfBoundsException` and when an `ArrayIndexOutOfBoundsException` can occur.

When traversing an `ArrayList`:

- Use a for-each loop to access each element without changing it, or to modify each object in the list using a mutator method.
- Use an `Iterator` to remove elements. (This is not in the AP subset, but it is the easiest way to remove elements from an `ArrayList`.)

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

When traversing a matrix:

- Use a row-column traversal to access, modify, or replace elements.
- 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.