Arrays and Array Lists

7

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

- → **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];
```

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.

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. 254).

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

➥ 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;
}
```

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):

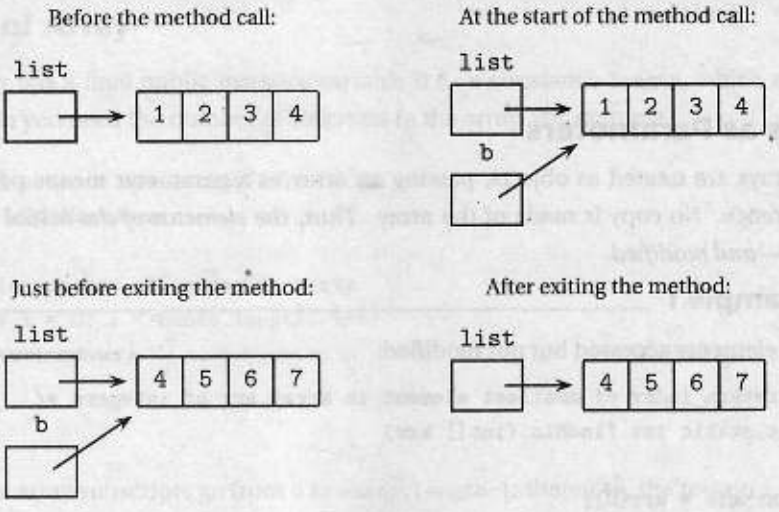
```
int[] list = {1, 2, 3, 4};
changeArray(list);
System.out.print("The changed list is ");
for (int num : list)
    System.out.print(num + " ");
```

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:



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; }
```

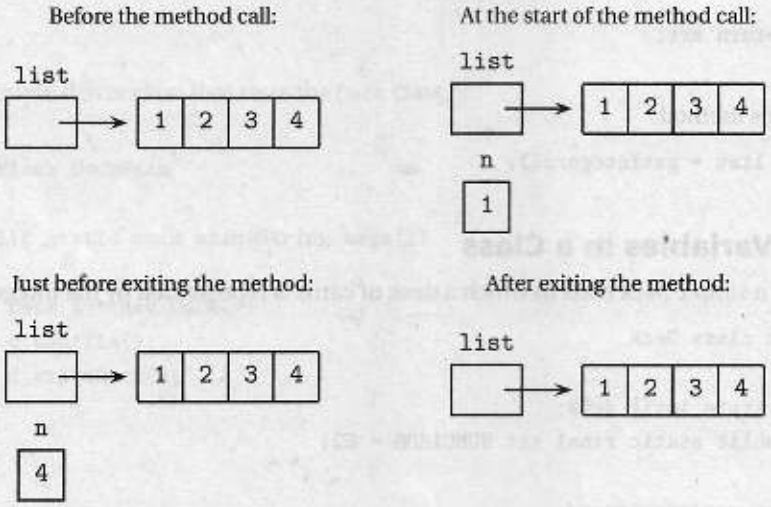
Here is some code that invokes this method:

```
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] = ...;              //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

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. Whereas 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 and the hashcode of `arr` in hexadecimal.

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.

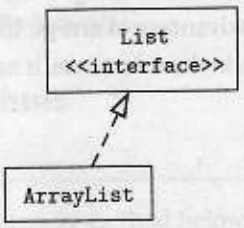
All of the collections classes, including `ArrayList`, have the following features in common:

- 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 or comparisons will lead to run-time errors.

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 an exception (`ClassCastException`, not part of the AP Java subset).

```
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`. If the specified element is not of type `E`, throws an exception (`ClassCastException`, not part of the AP Java subset).

`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 = ...;      //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.

(continued)

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

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

2. Recall, however, that a for-each loop should not 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 *(continued)*

```
/** 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()`.

➥ 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. 260 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:

```
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 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 $0 \le k <$ `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

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 1-dimensional array reference as a parameter to a method that processes each row;
- traverses the rows using either a regular loop or a for-each 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`.

```
/** @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 $0 \le$ `row` $<$ `mat.length`, you can use the `sumArray` method for each row in `mat`. Alternatively, you can use a for-each 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`.

```
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 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:

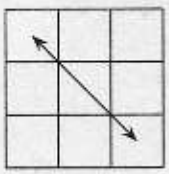
```
for (int row = 0; row < cardMatrix.length; row++)
    for (int col = 0; col < cardMatrix[0].length; col++)
        cardMatrix[row][col].changeValue(v);
```

NOTE

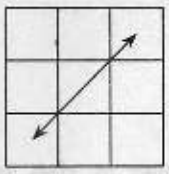
The use of the nested for-each loop is OK. Modifying the objects in the matrix with a mutator method is fine. What you shouldn't do is *replace* the `Card` objects with new `Card`s.

➥ 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.

```
/** 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] = ...;   //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 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 10 (The AP Computer Science A 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.

MULTIPLE-CHOICE QUESTIONS ON ARRAYS AND ARRAY LISTS

1. Which of the following correctly initializes an array `arr` to contain four elements each with value `0`?

   I
   ```
   int[] arr = {0, 0, 0, 0};
   ```
   II
   ```
   int[] arr = new int[4];
   ```
   III
   ```
   int[] arr = new int[4];
   for (int i = 0; i < arr.length; i++)
       arr[i] = 0;
   ```

   (A) I only
   (B) III only
   (C) I and III only
   (D) II and III only
   (E) I, II, and III

2. The following program segment is intended to find the index of the first negative integer in `arr[0] ... arr[N-1]`, where `arr` is an array of `N` integers.

   ```
   int i = 0;
   while (arr[i] >= 0)
   {
       i++;
   }
   location = i;
   ```

   This segment will work as intended
   (A) always.
   (B) never.
   (C) whenever `arr` contains at least one negative integer.
   (D) whenever `arr` contains at least one nonnegative integer.
   (E) whenever `arr` contains no negative integers.

3. Refer to the following code segment. You may assume that `arr` is an array of `int` values.

```
int sum = arr[0], i = 0;
while (i < arr.length)
{
    i++;
    sum += arr[i];
}
```

Which of the following will be the result of executing the segment?

(A) Sum of `arr[0], arr[1], ..., arr[arr.length-1]` will be stored in `sum`.
(B) Sum of `arr[1], arr[2], ..., arr[arr.length-1]` will be stored in `sum`.
(C) Sum of `arr[0], arr[1], ..., arr[arr.length]` will be stored in `sum`.
(D) An infinite loop will occur.
(E) A run-time error will occur.

4. Refer to the following code segment. You may assume that array `arr1` contains elements `arr1[0], arr1[1], ..., arr1[N-1]`, where `N = arr1.length`.

```
int count = 0;
for (int i = 0; i < N; i++)
    if (arr1[i] != 0)
    {
        arr1[count] = arr1[i];
        count++;
    }
int[] arr2 = new int[count];
for (int i = 0; i < count; i++)
    arr2[i] = arr1[i];
```

If array `arr1` initially contains the elements 0, 6, 0, 4, 0, 0, 2 in this order, what will `arr2` contain after execution of the code segment?

(A) 6, 4, 2
(B) 0, 0, 0, 0, 6, 4, 2
(C) 6, 4, 2, 4, 0, 0, 2
(D) 0, 6, 0, 4, 0, 0, 2
(E) 6, 4, 2, 0, 0, 0, 0

5. Consider this program segment:

```
for (int i = 2; i <= k; i++)
    if (arr[i] < someValue)
        System.out.print("SMALL");
```

What is the maximum number of times that `SMALL` can be printed?

(A) `0`
(B) `1`
(C) `k - 1`
(D) `k - 2`
(E) `k`

6. What will be output from the following code segment, assuming it is in the same class as the `doSomething` method?

```
int[] arr = {1, 2, 3, 4};
doSomething(arr);
System.out.print(arr[1] + " ");
System.out.print(arr[3]);
    ...
public void doSomething(int[] list)
{
    int[] b = list;
    for (int i = 0; i < b.length; i++)
        b[i] = i;
}
```

(A) `0 0`
(B) `2 4`
(C) `1 3`
(D) `0 2`
(E) `0 3`

7. Consider writing a program that reads the lines of any text file into a sequential list of lines. Which of the following is a good reason to implement the list with an `ArrayList` of `String` objects rather than an array of `String` objects?
   (A) The `get` and `set` methods of `ArrayList` are more convenient than the `[]` notation for arrays.
   (B) The `size` method of `ArrayList` provides instant access to the length of the list.
   (C) An `ArrayList` can contain objects of any type, which leads to greater generality.
   (D) If any particular text file is unexpectedly long, the `ArrayList` will automatically be resized. The array, by contrast, may go out of bounds.
   (E) The `String` methods are easier to use with an `ArrayList` than with an array.

8. Consider writing a program that produces statistics for long lists of numerical data. Which of the following is the best reason to implement each list with an array of `int` (or `double`), rather than an `ArrayList` of `Integer` (or `Double`) objects?
   (A) An array of primitive number types is more efficient to manipulate than an `ArrayList` of wrapper objects that contain numbers.
   (B) Insertion of new elements into a list is easier to code for an array than for an `ArrayList`.
   (C) Removal of elements from a list is easier to code for an array than for an `ArrayList`.
   (D) Accessing individual elements in the middle of a list is easier for an array than for an `ArrayList`.
   (E) Accessing all the elements is more efficient in an array than in an `ArrayList`.

Refer to the following classes for Questions 9–12.

```
public class Address
{
    private String name;
    private String street;
    private String city;
    private String state;
    private String zip;

    //constructors
       ...

    //accessors
    public String getName()
    { return name; }
    public String getStreet()
    { return street; }
    public String getCity()
    { return city; }
    public String getState()
    { return state; }
    public String getZip()
    { return zip; }
}

public class Student
{
    private int idNum;
    private double gpa;
    private Address address;

    //constructors
       ...

    //accessors
    public Address getAddress()
    { return address; }
    public int getIdNum()
    { return idNum; }
    public double getGpa()
    { return gpa; }
}
```

9. A client method has this declaration, followed by code to initialize the list:

```
Address[] list = new Address[100];
```

Here is a code segment to generate a list of *names only*.

```
for (Address a : list)
    /* line of code */
```

Which is a correct `/* line of code */`?

(A) `System.out.println(Address[i].getName());`
(B) `System.out.println(list[i].getName());`
(C) `System.out.println(a[i].getName());`
(D) `System.out.println(a.getName());`
(E) `System.out.println(list.getName());`

10. The following code segment is to print out a list of addresses:

```
for (Address addr : list)
{
    /* more code */
}
```

Which is a correct replacement for `/* more code */`?

I
```
System.out.println(list[i].getName());
System.out.println(list[i].getStreet());
System.out.print(list[i].getCity() + ", ");
System.out.print(list[i].getState() + " ");
System.out.println(list[i].getZip());
```

II
```
System.out.println(addr.getName());
System.out.println(addr.getStreet());
System.out.print(addr.getCity() + ", ");
System.out.print(addr.getState() + " ");
System.out.println(addr.getZip());
```

III
```
System.out.println(addr);
```

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

11. A client method has this declaration:

```
Student[] allStudents = new Student[NUM_STUDS];  //NUM_STUDS is
                                                 //an int constant
```

Here is a code segment to generate a list of `Student` names only. (You may assume that `allStudents` has been initialized.)

```
for (Student student : allStudents)
    /* code to print list of names */
```

Which is a correct replacement for `/* code to print list of names */`?

(A) `System.out.println(allStudents.getName());`

(B) `System.out.println(student.getName());`

(C) `System.out.println(student.getAddress().getName());`

(D) `System.out.println(allStudents.getAddress().getName());`

(E) `System.out.println(student[i].getAddress().getName());`

12. Here is a method that locates the `Student` with the highest `idNum`:

```
/** Precondition: Array stuArr of Student is initialized.
 *  @return Student with highest idNum
 */
public static Student locate(Student[] stuArr)
{
    /* method body */
}
```

Which of the following could replace `/* method body */` so that the method works as intended?

I
```
int max = stuArr[0].getIdNum();
for (Student student : stuArr)
    if (student.getIdNum() > max)
    {
        max = student.getIdNum();
        return student;
    }
return stuArr[0];
```

II
```
Student highestSoFar = stuArr[0];
int max = stuArr[0].getIdNum();
for (Student student : stuArr)
    if(student.getIdNum() > max)
    {
        max = student.getIdNum();
        highestSoFar = student;
    }
return highestSoFar;
```

III
```
int maxPos = 0;
for(int i = 1; i < stuArr.length; i++)
    if(stuArr[i].getIdNum() > stuArr[maxPos].getIdNum())
        maxPos = i;
return stuArr[maxPos];
```

(A) I only
(B) II only
(C) III only
(D) I and III only
(E) II and III only

Questions 13–15 refer to the `Ticket` and `Transaction` classes below.

```
public class Ticket
{
    private String row;
    private int seat;
    private double price;

    //constructor
    public Ticket(String aRow, int aSeat, double aPrice)
    {
        row = aRow;
        seat = aSeat;
        price = aPrice;
    }

    //accessors getRow(), getSeat(), and getPrice()
        ...
}

public class Transaction
{
    private int numTickets;
    private Ticket[] tickList;

    //constructor
    public Transaction(int numTicks)
    {
        numTickets = numTicks;
        tickList = new Ticket[numTicks];
        String theRow;
        int theSeat;
        double thePrice;
        for (int i = 0; i < numTicks; i++)
        {
            < read user input for theRow, theSeat, and thePrice >
                ...

            /* more code */
        }
    }

    /** @return total amount paid for this transaction */
    public double totalPaid()
    {
        double total = 0.0;
        /* code to calculate amount */
        return total;
    }
}
```

13. Which of the following correctly replaces /* *more code* */ in the Transaction constructor to initialize the tickList array?

(A)
```
tickList[i] = new Ticket(getRow(), getSeat(), getPrice());
```

(B)
```
tickList[i] = new Ticket(theRow, theSeat, thePrice);
```

(C)
```
tickList[i] = new tickList(getRow(), getSeat(), getPrice());
```

(D)
```
tickList[i] = new tickList(theRow, theSeat, thePrice);
```

(E)
```
tickList[i] = new tickList(numTicks);
```

14. Which represents correct /* *code to calculate amount* */ in the totalPaid method?

(A)
```
for (Ticket t : tickList)
    total += t.price;
```

(B)
```
for (Ticket t : tickList)
    total += tickList.getPrice();
```

(C)
```
for (Ticket t : tickList)
    total += t.getPrice();
```

(D)
```
Transaction T;
for (Ticket t : T)
    total += t.getPrice();
```

(E)
```
Transaction T;
for (Ticket t : T)
    total += t.price;
```

15. Suppose it is necessary to keep a list of all ticket transactions. Assuming that there are NUMSALES transactions, a suitable declaration would be

(A)
```
Transaction[] listOfSales = new Transaction[NUMSALES];
```

(B)
```
Transaction[] listOfSales = new Ticket[NUMSALES];
```

(C)
```
Ticket[] listOfSales = new Transaction[NUMSALES];
```

(D)
```
Ticket[] listOfSales = new Ticket[NUMSALES];
```

(E)
```
Transaction[] Ticket = new listOfSales[NUMSALES];
```

16. The following code fragment is intended to find the smallest value in `arr[0] ...arr[n-1]`.

```
/** Precondition:
 *   - arr is an array, arr.length = n.
 *   - arr[0]...arr[n-1] initialized with integers.
 *  Postcondition: min = smallest value in arr[0]...arr[n-1].
 */
int min = arr[0];
int i = 1;
while (i < n)
{
    i++;
    if (arr[i] < min)
        min = arr[i];
}
```

This code is incorrect. For the segment to work as intended, which of the following modifications could be made?

I Change the line

```
int i = 1;
```

to

```
int i = 0;
```

Make no other changes.

II Change the body of the `while` loop to

```
{
    if (arr[i] < min)
        min = arr[i];
    i++;
}
```

Make no other changes.

III Change the test for the `while` loop as follows:

```
while (i <= n)
```

Make no other changes.

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

17. Refer to method `match` below:

```
/** @param v an array of int sorted in increasing order
 *  @param w an array of int sorted in increasing order
 *  @param N the number of elements in array v
 *  @param M the number of elements in array w
 *  @return true if there is an integer k that occurs
 *  in both arrays; otherwise returns false
 *  Precondition:
 *    v[0]..v[N-1] and w[0]..w[M-1] initialized with integers.
 *    v[0] < v[1] < .. < v[N-1] and w[0] < w[1] < .. < w[M-1].
 */
public static boolean match(int[] v, int[] w, int N, int M)
{
    int vIndex = 0, wIndex = 0;
    while (vIndex < N && wIndex < M)
    {
        if (v[vIndex] == w[wIndex])
            return true;
        else if (v[vIndex] < w[wIndex])
            vIndex++;
        else
            wIndex++;
    }
    return false;
}
```

Assuming that the method has not been exited, which assertion is true at the end of every execution of the `while` loop?

(A) `v[0]..v[vIndex-1]` and `w[0]..w[wIndex-1]` contain no common value, `vIndex` ≤ `N` and `wIndex` ≤ `M`.

(B) `v[0]..v[vIndex]` and `w[0]..w[wIndex]` contain no common value, `vIndex` ≤ `N` and `wIndex` ≤ `M`.

(C) `v[0]..v[vIndex-1]` and `w[0]..w[wIndex-1]` contain no common value, `vIndex` ≤ `N-1` and `wIndex` ≤ `M-1`.

(D) `v[0]..v[vIndex]` and `w[0]..w[wIndex]` contain no common value, `vIndex` ≤ `N-1` and `wIndex` ≤ `M-1`.

(E) `v[0]..v[N-1]` and `w[0]..w[M-1]` contain no common value, `vIndex` ≤ `N` and `wIndex` ≤ `M`.

18. Consider this class:

```
public class Book
{
    private String title;
    private String author;
    private boolean checkoutStatus;

    public Book(String bookTitle, String bookAuthor)
    {
        title = bookTitle;
        author = bookAuthor;
        checkoutStatus = false;
    }

    /** Change checkout status. */
    public void changeStatus()
    { checkoutStatus = !checkoutStatus; }

    //Other methods are not shown.
}
```

A client program has this declaration:

```
Book[] bookList = new Book[SOME_NUMBER];
```

Suppose `bookList` is initialized so that each `Book` in the list has a title, author, and checkout status. The following piece of code is written, whose intent is to change the checkout status of each book in `bookList`.

```
for (Book b : bookList)
    b.changeStatus();
```

Which is *true* about this code?

(A) The `bookList` array will remain unchanged after execution.

(B) Each book in the `bookList` array will have its checkout status changed, as intended.

(C) A `NullPointerException` may occur.

(D) A run-time error will occur because it is not possible to modify objects using the for-each loop.

(E) A logic error will occur because it is not possible to modify objects in an array without accessing the indexes of the objects.

Consider this class for Questions 19 and 20:

```
public class BingoCard
{
    private int[] card;

    /** Default constructor: Creates BingoCard with
     *  20 random digits in the range 1 - 90.
     */
    public BingoCard()
    { /* implementation not shown */ }

    /* Display BingoCard. */
    public void display()
    { /* implementation not shown */ }
       ...
}
```

A program that simulates a bingo game declares an array of `BingoCard`. The array has `NUMPLAYERS` elements, where each element represents the card of a different player. Here is a code segment that creates all the bingo cards in the game:

```
/* declare array of BingoCard */
/* construct each BingoCard */
```

19. Which of the following is a correct replacement for

```
/* declare array of BingoCard */?
```

(A) `int[] BingoCard = new BingoCard[NUMPLAYERS];`

(B) `BingoCard[] players = new int[NUMPLAYERS];`

(C) `BingoCard[] players = new BingoCard[20];`

(D) `BingoCard[] players = new BingoCard[NUMPLAYERS];`

(E) `int[] players = new BingoCard[NUMPLAYERS];`

20. Assuming that `players` has been declared as an array of `BingoCard`, which of the following is a correct replacement for

```
/* construct each BingoCard */
```

I
```
for (BingoCard card : players)
    card = new BingoCard();
```

II
```
for (BingoCard card : players)
    players[card] = new BingoCard();
```

III
```
for (int i = 0; i < players.length; i++)
    players[i] = new BingoCard();
```

(A) I only
(B) II only
(C) III only
(D) I and III only
(E) I, II, and III

21. Which declaration will cause an error?

I `List<String> stringList = new ArrayList<String>();`

II `List<int> intList = new ArrayList<int>();`

III `ArrayList<String> compList = new ArrayList<String>();`

(A) I only
(B) II only
(C) III only
(D) I and III only
(E) II and III only

22. Consider these declarations:

```
List<String> strList = new ArrayList<String>();
String ch = " ";
Integer intOb = new Integer(5);
```

Which statement will cause an error?

(A) `strList.add(ch);`
(B) `strList.add(new String("handy andy"));`
(C) `strList.add(intOb.toString());`
(D) `strList.add(ch + 8);`
(E) `strList.add(intOb + 8);`

23. Let `list` be an `ArrayList<Integer>` containing these elements:

```
2 5 7 6 0 1
```

Which of the following statements would *not* cause an error to occur? Assume that each statement applies to the given list, independent of the other statements.

(A) `Object ob = list.get(6);`
(B) `Integer intOb = list.add(3.4);`
(C) `list.add(6, 9);`
(D) `Object x = list.remove(6);`
(E) `Object y = list.set(6, 8);`

24. Refer to method `insert` below:

```
/** @param list an ArrayList of String objects
 *  @param element a String object
 *  Precondition: list contains String values sorted
 *                in decreasing order.
 *  Postcondition: element inserted in its correct position in list.
 */
public void insert(List<String> list, String element)
{
    int index = 0;
    while (element.compareTo(list.get(index)) < 0)
        index++;
    list.add(index, element);
}
```

Assuming that the type of `element` is compatible with the objects in the list, which is a *true* statement about the `insert` method?

(A) It works as intended for all values of `element`.
(B) It fails for all values of `element`.
(C) It fails if `element` is greater than the first item in `list` and works in all other cases.
(D) It fails if `element` is smaller than the last item in `list` and works in all other cases.
(E) It fails if `element` is either greater than the first item or smaller than the last item in `list` and works in all other cases.

25. Consider the following code segment, applied to `list`, an `ArrayList` of `Integer` values.

```
int len = list.size();
for (int i = 0; i < len; i++)
{
    list.add(i + 1, new Integer(i));
    Object x = list.set(i, new Integer(i + 2));
}
```

If `list` is initially `6 1 8`, what will it be following execution of the code segment?

(A) `2 3 4 2 1 8`
(B) `2 3 4 6 2 2 0 1 8`
(C) `2 3 4 0 1 2`
(D) `2 3 4 6 1 8`
(E) `2 3 3 2`

Questions 26 and 27 are based on the `Coin` and `Purse` classes given below:

```
/* A simple coin class */
public class Coin
{
    private double value;
    private String name;

    //constructor
    public Coin(double coinValue, String coinName)
    {
        value = coinValue;
        name = coinName;
    }

    /** @return the value of this coin */
    public double getValue()
    { return value; }

    /** @return the name of this coin */
    public String getName()
    { return name; }

    /** @param obj a Coin object
     *  @return true if this coin equals obj; otherwise false
     */
    public boolean equals(Object obj)
    { return name.equals(((Coin) obj).name); }

    //Other methods are not shown.
}

/* A purse holds a collection of coins */
public class Purse
{
    private List<Coin> coins;

    /** Creates an empty purse. */
    public Purse()
    { coins = new ArrayList<Coin>(); }

    /** Adds aCoin to the purse.
     *  @param aCoin the coin to be added to the purse
     */
    public void add(Coin aCoin)
    { coins.add(aCoin); }

    /** @return the total value of coins in purse */
    public double getTotal()
    { /* implementation not shown */}

}
```

26. Here is the `getTotal` method from the `Purse` class:

```
/** @return the total value of coins in purse */
public double getTotal()
{
    double total = 0;
    /* more code */
    return total;
}
```

Which of the following is a correct replacement for `/* more code */`?

(A)
```
for (Coin c : coins)
{
    c = coins.get(i);
    total += c.getValue();
}
```

(B)
```
for (Coin c : coins)
{
    Coin value = c.getValue();
    total += value;
}
```

(C)
```
for (Coin c : coins)
{
    Coin c = coins.get(i);
    total += c.getValue();
}
```

(D)
```
for (Coin c : coins)
{
    total += coins.getValue();
}
```

(E)
```
for (Coin c : coins)
{
    total += c.getValue();
}
```

27. Two coins are said to *match* each other if they have the same name or the same value. You may assume that coins with the same name have the same value and coins with the same value have the same name. A boolean method `find` is added to the `Purse` class:

```
/** @return true if the purse has a coin that matches aCoin,
 *  false otherwise
 */
public boolean find(Coin aCoin)
{
   for (Coin c : coins)
   {
      /* code to find match */
   }
   return false;
}
```

Which is a correct replacement for `/* code to find match */`?

I
```
if (c.equals(aCoin))
   return true;
```

II
```
if ((c.getName()).equals(aCoin.getName()))
   return true;
```

III
```
if ((c.getValue()).equals(aCoin.getValue()))
   return true;
```

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

28. Which of the following initializes an 8 × 10 matrix with integer values that are perfect squares? (0 is a perfect square.)

I
```
int[][] mat = new int[8][10];
```

II
```
int[][] mat = new int[8][10];
for (int r = 0; r < mat.length; r++)
   for (int c = 0; c < mat[r].length; c++)
      mat[r][c] = r * r;
```

III
```
int[][] mat = new int[8][10];
for (int c = 0; c < mat[r].length; c++)
   for (int r = 0; r < mat.length; r++)
      mat[r][c] = c * c;
```

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

29. Consider a class that has this private instance variable:

```
private int[][] mat;
```

The class has the following method, `alter`.

```
public void alter(int c)
{
    for (int i = 0; i < mat.length; i++)
        for (int j = c + 1; j < mat[0].length; j++)
            mat[i][j-1] = mat[i][j];
}
```

If a 3 × 4 matrix `mat` is

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

then `alter(1)` will change `mat` to

(A)
```
1 5 7 7
2 6 8 8
3 7 9 9
```

(B)
```
1 5 7
2 6 8
3 7 9
```

(C)
```
1 3 5 7
3 5 7 9
```

(D)
```
1 3 5 7
3 5 7 9
3 5 7 9
```

(E)
```
1 7 7 7
2 8 8 8
3 9 9 9
```

30. Consider the following method that will alter the matrix `mat`:

```
/** @param mat the initialized matrix
 *  @param row the row number
 */
public static void matStuff(int[][] mat, int row)
{
    int numCols = mat[0].length;
    for (int col = 0; col < numCols; col++)
        mat[row][col] = row;
}
```

Suppose `mat` is originally

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

After the method call `matStuff(mat,2)`, matrix `mat` will be

(A)

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

(B)

1	4	9	0
2	2	2	2
5	1	4	3

(C)

2	2	2	2
2	2	2	2
2	2	2	2

(D)

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

(E)

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

31. Assume that a square matrix `mat` is defined by

```
int[][] mat = new int[SIZE][SIZE];
//SIZE is an integer constant >= 2
```

What does the following code segment do?

```
for (int i = 0; i < SIZE - 1; i++)
    for (int j = 0; j < SIZE - i - 1; j++)
        swap(mat, i, j, SIZE - j - 1, SIZE - i - 1);
```

You may assume the existence of this `swap` method:

```
/** Interchange mat[a][b] and mat[c][d]. */
public void swap(int[][] mat, int a, int b, int c, int d)
```

(A) Reflects `mat` through its major diagonal. For example,

```
2  6        2  4
      ⟶
4  3        6  3
```

(B) Reflects `mat` through its minor diagonal. For example,

```
2  6        3  6
      ⟶
4  3        4  2
```

(C) Reflects `mat` through a horizontal line of symmetry. For example,

```
2  6        4  3
      ⟶
4  3        2  6
```

(D) Reflects `mat` through a vertical line of symmetry. For example,

```
2  6        6  2
      ⟶
4  3        3  4
```

(E) Leaves `mat` unchanged.

32. Consider a class `MatrixStuff` that has a private instance variable:

```
private int[][] mat;
```

Refer to method `alter` below that occurs in the `MatrixStuff` class. (The lines are numbered for reference.)

```
Line 1:  /** @param mat the matrix initialized with integers
Line 2:   *  @param c the column to be removed
Line 3:   *  Postcondition:
Line 4:   *    - Column c has been removed.
Line 5:   *    - The last column is filled with zeros.
Line 6:   */
Line 7:  public void alter(int[][] mat, int c)
Line 8:  {
Line 9:      for (int i = 0; i < mat.length; i++)
Line 10:         for (int j = c; j < mat[0].length; j++)
Line 11:             mat[i][j] = mat[i][j+1];
Line 12:     //code to insert zeros in rightmost column
Line 13:         ...
Line 14: }
```

The intent of the method `alter` is to remove column `c`. Thus, if the input matrix `mat` is

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

the method call `alter(mat, 1)` should change `mat` to

2	8	9	0
1	4	3	0
0	3	2	0

The method does not work as intended. Which of the following changes will correct the problem?

I Change line 10 to

```
for (int j = c; j < mat[0].length - 1; j++)
```

and make no other changes.

II Change lines 10 and 11 to

```
for (int j = c + 1; j < mat[0].length; j++)
    mat[i][j-1] = mat[i][j];
```

and make no other changes.

III Change lines 10 and 11 to

```
for (int j = mat[0].length - 1; j > c; j--)
    mat[i][j-1] = mat[i][j];
```

and make no other changes.

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

33. This question refers to the following method:

```
public static boolean isThere(String[][] mat, int row, int col,
    String symbol)
{
    boolean yes;
    int i, count = 0;
    for (i = 0; i < SIZE; i++)
        if (mat[i][col].equals(symbol))
            count++;
    yes = (count == SIZE);
    count = 0;
    for (i = 0; i < SIZE; i++)
        if (mat[row][i].equals(symbol))
            count++;
    return (yes || count == SIZE);
}
```

Now consider this code segment:

```
public final int SIZE = 8;
String[][] mat = new String[SIZE][SIZE];
```

Which of the following conditions on a matrix `mat` of the type declared in the code segment will by itself guarantee that

```
isThere(mat, 2, 2, "$")
```

will have the value `true` when evaluated?

I The element in row 2 and column 2 is `"$"`
II All elements in both diagonals are `"$"`
III All elements in column 2 are `"$"`

(A) I only
(B) III only
(C) I and II only
(D) I and III only
(E) II and III only

34. The method `changeNegs` below should replace every occurrence of a negative integer in its matrix parameter with 0.

```
/** @param mat the matrix
 *  Precondition: mat is initialized with integers.
 *  Postcondition: All negative values in mat replaced with 0.
 */
public static void changeNegs(int[][] mat)
{
    /* code */
}
```

Which is correct replacement for `/* code */`?

I
```
for (int r = 0; r < mat.length; r++)
    for (int c = 0; c < mat[r].length; c++)
        if (mat[r][c] < 0)
            mat[r][c] = 0;
```

II
```
for (int c = 0; c < mat[0].length; c++)
    for (int r = 0; r < mat.length; r++)
        if (mat[r][c] < 0)
            mat[r][c] = 0;
```

III
```
for (int[] row : mat)
    for (int element : row)
        if (element < 0)
            element = 0;
```

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

35. A two-dimensional array of `double`, `rainfall`, will be used to represent the daily rainfall for a given year. In this scheme, `rainfall[month][day]` represents the amount of rain on the given day and month. For example,

`rainfall[1][15]` is the amount of rain on Jan. 15
`rainfall[12][25]` is the amount of rain on Dec. 25

The array can be declared as follows:

```
double[][] rainfall = new double[13][32];
```

This creates 13 rows indexed from 0 to 12 and 32 columns indexed from 0 to 31, all initialized to 0.0. Row 0 and column 0 will be ignored. Column 31 in row 4 will be ignored, since April 31 is not a valid day. In years that are not leap years, columns 29, 30, and 31 in row 2 will be ignored since Feb. 29, 30, and 31 are not valid days.

Consider the method `averageRainfall` below:

```
/** Precondition:
 *   - rainfall is initialized with values representing amounts
 *     of rain on all valid days.
 *   - Invalid days are initialized to 0.0.
 *   - Feb 29 is not a valid day.
 *  Postcondition: Returns average rainfall for the year.
 */
public double averageRainfall(double rainfall[][])
{
    double total = 0.0;
    /* more code */
}
```

Which of the following is a correct replacement for `/* more code */` so that the postcondition for the method is satisfied?

I
```
for (int month = 1; month < rainfall.length; month++)
    for (int day = 1; day < rainfall[month].length; day++)
        total += rainfall[month][day];
return total / (13 * 32);
```

II
```
for (int month = 1; month < rainfall.length; month++)
    for (int day = 1; day < rainfall[month].length; day++)
        total += rainfall[month][day];
return total / 365;
```

III
```
for (double[] month : rainfall)
    for (double rainAmt : month)
        total += rainAmt;
return total / 365;
```

(A) None
(B) I only
(C) II only
(D) III only
(E) II and III only

36. This question is based on the `Point` class below:

```
public class Point
{
    /** The coordinates. */
    private int x;
    private int y;

    public Point (int xValue, int yValue)
    {
        x = xValue;
        y = yValue;
    }

    /** @return the x-coordinate of this point */
    public int getx()
    { return x; }

    /** @return the y-coordinate of this point */
    public int gety()
    { return y; }

    /** Set x and y to new_x and new_y. */
    public void setPoint(int new_x, int new_y)
    {
        x = new_x;
        y = new_y;
    }

    //Other methods are not shown.
}
```

The method `changeNegs` below takes a matrix of `Point` objects as parameter and replaces every `Point` that has as least one negative coordinate with the `Point` (0,0).

```
/** @param pointMat the matrix of points
 *  Precondition: pointMat is initialized with Point objects.
 *  Postcondition: Every point with at least one negative coordinate
 *                 has been changed to have both coordinates
 *                 equal to zero.
 */
public static void changeNegs (Point [][] pointMat)
{
    /* code */
}
```

Which is a correct replacement for `/* code */`?

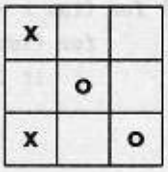
I
```
for (int r = 0; r < pointMat.length; r++)
    for (int c = 0; c < pointMat[r].length; c++)
        if (pointMat[r][c].getx() < 0
            || pointMat[r][c].gety() < 0)
            pointMat[r][c].setPoint(0, 0);
```

II
```
for (int c = 0; c < pointMat[0].length; c++)
    for (int r = 0; r < pointMat.length; r++)
        if (pointMat[r][c].getx() < 0
            || pointMat[r][c].gety() < 0)
            pointMat[r][c].setPoint(0, 0);
```

III
```
for (Point[] row : pointMat)
    for (Point p : row)
        if (p.getx() < 0 || p.gety() < 0)
            p.setPoint(0, 0);
```

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

37. A simple Tic-Tac-Toe board is a 3 × 3 array filled with either X's, O's, or blanks. Here is a class for a game of Tic-Tac-Toe:




```
public class TicTacToe
{
    private String[][] board;
    private static final int ROWS = 3;
    private static final int COLS = 3;

    /** Construct an empty board. */
    public TicTacToe()
    {
        board = new String[ROWS][COLS];
        for (int r = 0; r < ROWS; r++)
            for (int c = 0; c < COLS; c++)
                board[r][c] = " ";
    }

    /** @param r the row number
     *  @param c the column number
     *  @param symbol the symbol to be placed on board[r][c]
     *  Precondition: The square board[r][c] is empty.
     *  Postcondition: symbol placed in that square.
     */
    public void makeMove(int r, int c, String symbol)
    {
        board[r][c] = symbol;
    }

    /** Creates a string representation of the board, e.g.
     *    |o  |
     *    |xx |
     *    |  o|
     * @return the string representation of board
     */
    public String toString()
    {
        String s = "";      //empty string
        /* more code */
        return s;
    }
}
```

Which segment represents a correct replacement for /* *more code* */ for the `toString` method?

(A)
```
for (int r = 0; r < ROWS; r++)
{
    for (int c = 0; c < COLS; c++)
    {
        s = s + "|";
        s = s + board[r][c];
        s = s + "|\n";
    }
}
```

(B)
```
for (int r = 0; r < ROWS; r++)
{
    s = s + "|";
    for (int c = 0; c < COLS; c++)
    {
        s = s + board[r][c];
        s = s + "|\n";
    }
}
```

(C)
```
for (int r = 0; r < ROWS; r++)
{
    s = s + "|";
    for (int c = 0; c < COLS; c++)
        s = s + board[r][c];
}
s = s + "|\n";
```

(D)
```
for (int r = 0; r < ROWS; r++)
    s = s + "|";
for (int c = 0; c < COLS; c++)
{
    s = s + board[r][c];
    s = s + "|\n";
}
```

(E)
```
for (int r = 0; r < ROWS; r++)
{
    s = s + "|";
    for (int c = 0; c < COLS; c++)
        s = s + board[r][c];
    s = s + "|\n";
}
```

ANSWER KEY

1. **E**
2. **C**
3. **E**
4. **A**
5. **C**
6. **C**
7. **D**
8. **A**
9. **D**
10. **B**
11. **C**
12. **E**
13. **B**
14. **C**
15. **A**
16. **B**
17. **A**
18. **B**
19. **D**
20. **C**
21. **B**
22. **E**
23. **C**
24. **D**
25. **A**
26. **E**
27. **D**
28. **D**
29. **A**
30. **A**
31. **B**
32. **D**
33. **B**
34. **D**
35. **E**
36. **E**
37. **E**

ANSWERS EXPLAINED

1. **(E)** Segment I is an initializer list which is equivalent to

```
int[] arr = new int[4];
arr[0] = 0;
arr[1] = 0;
arr[2] = 0;
arr[3] = 0;
```

   Segment II creates four slots for integers, which by default are initialized to `0`. The `for` loop in segment III is therefore unnecessary. It is not, however, incorrect.

2. **(C)** If `arr` contains no negative integers, the value of `i` will eventually exceed `N-1`, and `arr[i]` will cause an `ArrayIndexOutOfBoundsException` to be thrown.

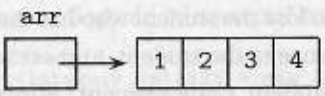
3. **(E)** The intent is to sum elements `arr[0]`, `arr[1]`, ..., `arr[arr.length-1]`. Notice, however, that when `i` has the value `arr.length-1`, it is incremented to `arr.length` in the loop, so the statement `sum += arr[i]` uses `arr[arr.length]`, which is out of range.

4. **(A)** The code segment has the effect of removing all occurrences of 0 from array `arr1`. The algorithm copies the nonzero elements to the front of `arr1`. Then it transfers them to array `arr2`.

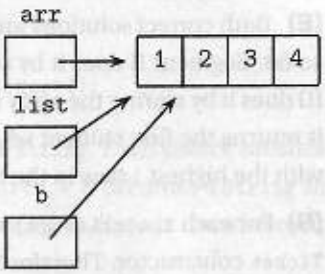
5. **(C)** If `arr[i] < someValue` for all `i` from 2 to `k`, `SMALL` will be printed on each iteration of the `for` loop. Since there are `k - 1` iterations, the maximum number of times that `SMALL` can be printed is `k - 1`.

6. **(C)** Array `arr` is changed by `doSomething`. Here are the memory slots:

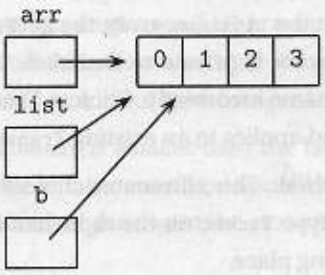
Just before doSomething is called:



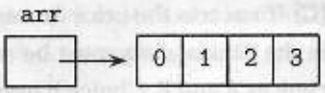
Just after doSomething is called,
but before the for loop is executed:



Just before exiting doSomething:



Just after exiting doSomething:



7. **(D)** Arrays are of fixed length and do not shrink or grow if the size of the data set varies. An `ArrayList` automatically resizes the list. Choice A is false: The [] notation is compact and easy to use. Choice B is not a valid reason because an array `arr` also provides instant access to its length with the quantity `arr.length`. Choice C is invalid because an array can also contain objects. Also, generality is beside the point in the given program: The list *must* hold `String` objects. Choice E is false: Whether a `String` object is `arr[i]` or `list.get(i)`, the `String` methods are equally easy to invoke.
8. **(A)** In order for numerical elements to be added to an `ArrayList`, each element must be wrapped in a wrapper class before insertion into the list. Then, to retrieve a numerical value from the `ArrayList`, the element must be unboxed using the `intValue` or `doubleValue` methods. Even though these operations can be taken care of with autoboxing and -unboxing, there are efficiency costs. In an array, you simply use the [] notation for assignment (as in `arr[i] = num`) or retrieval (`value = arr[i]`). Note that choices B and C are false statements: Both insertion and deletion for an array involve writing code to shift elements. An `ArrayList` automatically takes care of this through its `add` and `remove` methods. Choice D is a poor reason for choosing an array. While the `get` and `set` methods of `ArrayList` might be slightly more awkward than using the [] notation, both mechanisms work pretty easily. Choice E is false: Efficiency of access is roughly the same.
9. **(D)** For each `Address` object `a` in `list`, access the name of the object with `a.getName()`.
10. **(B)** Since the `Address` class does not have a `toString` method, each data field must explicitly be printed. Segment III would work if there *were* a `toString` method for the class (but there isn't, so it doesn't!). Segment I fails because of incorrect use of the for-each loop: The array index should not be accessed.
11. **(C)** Each `Student` name must be accessed through the `getName()` accessor of the `Address` class. The expression `student.getAddress()` accesses the entire address of that

student. The `name` field is then accessed using the `getName()` accessor of the `Address` class.

12. **(E)** Both correct solutions are careful not to lose the student who has the highest `idNum` so far. Segment II does it by storing a reference to the student, `highestSoFar`. Segment III does it by storing the array index of that student. Code segment I is incorrect because it returns the first student whose `idNum` is greater than `max`, not necessarily the student with the highest `idNum` in the list.

13. **(B)** For each `i`, `tickList[i]` is a new `Ticket` object that must be constructed using the `Ticket` constructor. Therefore eliminate choices C, D, and E. Choice A is wrong because `getRow()`, `getSeat()`, and `getPrice()` are accessors for values *that already exist* for some `Ticket` object. Note also the absence of the dot member construct.

14. **(C)** To access the price for each `Ticket` in the `tickList` array, the `getPrice()` accessor in the `Ticket` class must be used, since `price` is private to that class. This eliminates choices A and E. Choice B uses the array name incorrectly. Choices D and E incorrectly declare a `Transaction` object. (The method applies to an existing `Transaction` object.)

15. **(A)** An array of type `Transaction` is required. This eliminates choices C and D. Additionally, choices B and D incorrectly use type `Ticket` on the right-hand side. Choice E puts the identifier `listOfSales` in the wrong place.

16. **(B)** There are two problems with the segment as given:

    1. `arr[1]` is not tested.
    2. When `i` has a value of `n-1`, incrementing `i` will lead to an out-of-range error for the `if(arr[i] < min)` test.

    Modification II corrects both these errors. The change suggested in III corrects neither of these errors. The change in I corrects (1) but not (2).

17. **(A)** Notice that either `vIndex` or `wIndex` is incremented at the end of the loop. This means that, when the loop is exited, the current values of `v[vIndex]` and `w[wIndex]` have not been compared. Therefore, you can only make an assertion for values `v[0]..v[vIndex-1]` and `w[0]..w[wIndex-1]`. Also, notice that if there is no common value in the arrays, the exiting condition for the `while` loop will be that the end of one of the arrays has been reached, namely `vIndex` equals `N` or `wIndex` equals `M`.

18. **(B)** Objects in an array can be changed in a for-each loop by using mutator methods of the objects' class. The `changeStatus` method, a mutator in the `Book` class, will work as intended in the given code. Choice C would be true if it were not given that each `Book` in `bookList` was initialized. If any given `b` had a value of `null`, then a `NullPointerException` would be thrown.

19. **(D)** The declaration must start with the type of value in the array, namely `BingoCard`. This eliminates choices A and E. Eliminate choice B: The type on the right of the assignment should be `BingoCard`. Choice C is wrong because the number of slots in the array should be `NUMPLAYERS`, not 20.

20. **(C)** Segment III is the only segment that works, since the for-each loop should not be used to replace elements in an array. After the declaration

```
BingoCard[] players = new BingoCard[NUMPLAYERS];
```

each element in the `players` array is `null`. The intent in the given code is to replace each

null reference with a newly constructed `BingoCard`.

21. **(B)** The type parameter in a generic `ArrayList` must be a class type, not a primitive. Declaration II would be correct if it were

```
List<Integer> intList = new ArrayList<Integer>();
```

22. **(E)** All elements added to `strList` must be of type `String`. Each choice satisfies this except choice E. Note that in choice D, the expression `ch + 8` becomes a `String` since `ch` is a `String` (just one of the operands needs to be a `String` to convert the whole expression to a `String`). In choice E, neither `intOb` nor `8` is a `String`.

23. **(C)** The effect of choice C is to adjust the size of the list to 7 and to add the `Integer` `9` to the last slot (i.e., the slot with index `6`). Choices A, D, and E will all cause an `IndexOutOfBoundsException` because there is no slot with index `6`: the last slot has index `5`. Choice B will cause a compile-time error, since it is attempting to add an element of type `Double` to a list of type `Integer`.

24. **(D)** If `element` is smaller than the last item in the list, it will be compared with every item in the list. Eventually `index` will be incremented to a value that is out of bounds. To avoid this error, the test in the `while` loop should be

```
while(index < list.size() && element.compareTo(list.get(index)) < 0)
```

Notice that if `element` is greater than or equal to at least one item in `list`, the test as given in the problem will eventually be false, preventing an out-of-range error.

25. **(A)** Recall that `add(index, obj)` shifts all elements, starting at `index`, one unit to the right, then inserts `obj` at position `index`. The `set(index, obj)` method replaces the element in position `index` with `obj`. So here is the state of `list` after each change:

```
i = 0    6 0 1 8
         2 0 1 8
i = 1    2 0 1 1 8
         2 3 1 1 8
i = 2    2 3 1 2 1 8
         2 3 4 2 1 8
```

26. **(E)** The value of each `Coin c` in `coins` must be accessed with `c.getValue()`. This eliminates choice D. Eliminate choices A and B: The loop accesses each `Coin` in the `coins` `ArrayList`, which means that there should not be any statements attempting to get the next `Coin`. Choice B would be correct if the first statement in the loop body were

```
double value = c.getValue();
```

27. **(D)** Code segment III is wrong because the `equals` method is defined for objects only. Since `getValue` returns a `double`, the quantities `c.getValue()` and `aCoin.getValue()` must be compared either using `==`, or as described in the box on p. 75 (better).

28. **(D)** Segment II is the straightforward solution. Segment I is correct because it initializes all slots of the matrix to 0, a perfect square. (By default, all arrays of `int` or `double` are initialized to 0.) Segment III fails because `r` is undefined in the condition `c < mat[r].length`. In order to do a column-by-column traversal, you need to get the number of columns in each row. The outer `for` loop could be

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

Now segment III works. Note that since the array is rectangular, you can use any index `k` in the conditional `c < mat[k].length`, provided that `k` satisfies the condition $0 \le k <$ `mat.length` (the number of rows).

29. **(A)** Method `alter` shifts all the columns, starting at column `c+1`, one column to the left. Also, it does it in a way that overwrites column `c`. Here are the replacements for the method call `alter(1)`:

```
mat[0][1] = mat[0][2]
mat[0][2] = mat[0][3]
mat[1][1] = mat[1][2]
mat[1][2] = mat[1][3]
mat[2][1] = mat[2][2]
mat[2][2] = mat[2][3]
```

30. **(A)** `matStuff` processes the row selected by the row parameter, 2 in the method call. The row value, 2, overwrites each element in row 2. Don't make the mistake of selecting choice B—the row labels are 0, 1, 2.

31. **(B)** Hand execute this for a 2×2 matrix. `i` goes from 0 to 0, `j` goes from 0 to 0, so the only interchange is swap `mat[0][0]` with `mat[1][1]`, which suggests choice B. Check with a 3×3 matrix:

```
i = 0   j = 0   swap mat[0][0] with mat[2][2]
        j = 1   swap mat[0][1] with mat[1][2]
i = 1   j = 0   swap mat[1][0] with mat[2][1]
```

The elements to be interchanged are shown paired in the following figure. The result will be a reflection through the minor diagonal.

32. **(D)** The method as given will throw an `ArrayIndexOutOfBoundsException`. For the matrix in the example, `mat[0].length` is 4. The call `mat.alter(1)` gives `c` a value of 1. Thus, in the inner `for` loop, `j` goes from 1 to 3. When `j` is 3, the line `mat[i][j] = mat[i][j+1]` becomes `mat[i][3] = mat[i][4]`. Since columns go from 0 to 3, `mat[i][4]` is out of range. The changes in segments I and II both fix this problem. In each case, the correct replacements are made for each row `i`: `mat[i][1] = mat[i][2]` and `mat[i][2] = mat[i][3]`. Segment III makes the following incorrect replacements as `j` goes from 3 to 2: `mat[i][2] = mat[i][3]` and `mat[i][1] = mat[i][2]`. This will cause both columns 1 and 2 to be overwritten. Before inserting zeros in the last column, `mat` will be

```
2  9  9  9
1  3  3  3
0  2  2  2
```

This does not achieve the intended postcondition of the method.

33. **(B)** For the method call `isThere(mat, 2, 2, "$")`, the code counts how many times `"$"` appears in row 2 and how many times in column 2. The method returns `true` only if `count == SIZE` for either the row or column pass (i.e., the whole of row 2 or the whole of column 2 contains the symbol `"$"`). This eliminates choices I and II.

34. **(D)** Segment I is a row-by-row traversal; segment II is a column-by-column traversal. Each achieves the correct postcondition. Segment III traverses the matrix but does not alter it. All that is changed is the local variable `element`. You cannot use this kind of loop to replace elements in an array.

35. **(E)** Since there are 365 valid days in a year, the divisor in calculating the average must be 365. It may appear that segments II and III are incorrect because they include rainfall for invalid days in `total`. Since these values are initialized to `0.0`, however, including them in the total won't affect the final result.

36. **(E)** This is similar to the previous question, but in this case segment III is also correct. This is because instead of *replacing* a matrix element, you are *modifying* it using a mutator method.

37. **(E)** There are three things that must be done in each row:

    - Add an opening boundary line:

      ```
      s = s + "|";
      ```

    - Add the symbol in each square:

      ```
      for (int c = 0; c < COLS; c++)
          s = s + board[r][c];
      ```

    - Add a closing boundary line and go to the next line:

      ```
      s = s + "|\n";
      ```

    All of these statements must therefore be enclosed in the outer `for` loop, that is,

    ```
    for (int r = ...)
    ```