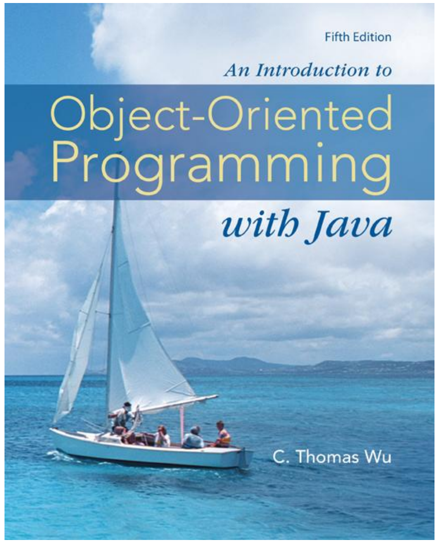


Chapter 10

Arrays
and
Collections

Objectives

- After you have read and studied this chapter, you should be able to
  - Manipulate a collection of data values, using an array.
  - Declare and use an array of primitive data types in writing a program.
  - Declare and use an array of objects in writing a program
  - Define a method that accepts an array as its parameter and a method that returns an array
  - Describe how a two-dimensional array is implemented as an array of arrays
  - Manipulate a collection of objects, using lists and maps

Array Basics

- An array is a collection of data values.
- If your program needs to deal with 100 integers, 500 Account objects, 365 real numbers, etc., you will use an array.
- In Java, an array is an indexed collection of data values of the same type.

Arrays of Primitive Data Types

- Array Declaration

```
<data type> [ ] <variable>        //variation 1
<data type>   <variable>[ ]       //variation 2
```

- Array Creation

```
<variable>  = new <data type> [ <size> ]
```

- Example

Variation 1

```
double[ ] rainfall;
rainfall
   = new double[12];
```

Variation 2

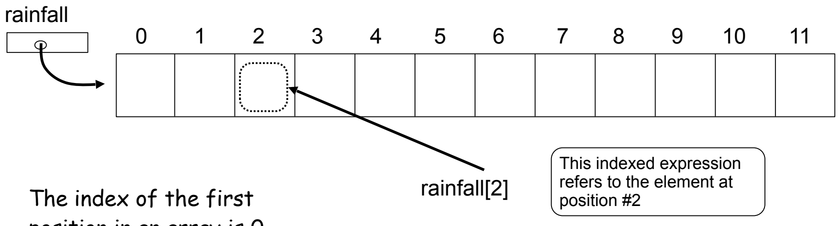
```
double rainfall [ ];
rainfall
   = new double[12];
```

An array is like an object!

Accessing Individual Elements

- Individual elements in an array accessed with the indexed expression.

```
double[] rainfall = new double[12];
```



This indexed expression refers to the element at position #2

The index of the first position in an array is 0.

Array Processing – Sample1

```
Scanner scanner = new Scanner(System.in);
double[] rainfall = new double[12];

double   annualAverage,
          sum = 0.0;

for (int i = 0; i < rainfall.length; i++) {

    System.out.print("Rainfall for month " + (i+1));
    rainfall[i] = scanner.nextDouble( );
    sum += rainfall[i];
}

annualAverage = sum / rainfall.length;
```

The public constant length returns the capacity of an array.

Array Processing – Sample 2

```
Scanner scanner = new Scanner(System.in);
double[] rainfall = new double[12];
String[] monthName = new String[12];
monthName[0] = "January";
monthName[1] = "February";
…
double   annualAverage, sum = 0.0;

for (int i = 0; i < rainfall.length; i++) {
    System.out.print("Rainfall for " + monthName[i] + ": ");
    rainfall[i] = scanner.nextDouble();
    sum += rainfall[i];
}
annualAverage = sum / rainfall.length;
```

The same pattern for the remaining ten months.

The actual month name instead of a number.

Array Processing – Sample 3

- Compute the average rainfall for each quarter.

```
//assume rainfall is declared and initialized properly

double[] quarterAverage = new double[4];

for (int i = 0; i < 4; i++) {
    sum = 0;
    for (int j = 0; j < 3; j++) {
                                        //compute the sum of
        sum += rainfall[3*i + j];       //one quarter
    }
    quarterAverage[i] = sum / 3.0;    //Quarter (i+1) average
}
```

Array Initialization

- Like other data types, it is possible to declare and initialize an array at the same time.

```
int[] number = { 2, 4, 6, 8 };

double[] samplingData = { 2.443, 8.99, 12.3, 45.009, 18.2,
                           9.00, 3.123, 22.084, 18.08 };

String[] monthName = { "January", "February", "March",
                       "April", "May", "June", "July",
                       "August", "September", "October",
                       "November", "December"  };
```

number.length → **4**

samplingData.length → **9**

monthName.length → **12**

Variable-size Declaration

- In Java, we are not limited to fixed-size array declaration.
- The following code prompts the user for the size of an array and declares an array of designated size:

```
Scanner scanner = new Scanner(System.in);
int size;
int[] number;

System.out.print("Size of an array:"));
size= scanner.nextInt( );

number = new int[size];
```

Arrays of Objects

- In Java, in addition to arrays of primitive data types, we can declare arrays of objects
- An array of primitive data is a powerful tool, but an array of objects is even more powerful.
- The use of an array of objects allows us to model the application more cleanly and logically.

The Person Class

- We will use Person objects to illustrate the use of an array of objects.

```
Person latte;

latte = new Person( );
latte.setName("Ms. Latte");
latte.setAge(20);
latte.setGender('F');

System.out.println( "Name: " + latte.getName()   );
System.out.println( "Age : " + latte.getAge()    );
System.out.println( "Sex : " + latte.getGender() );
```

The Person class supports the set methods and get methods.

Creating an Object Array - 1

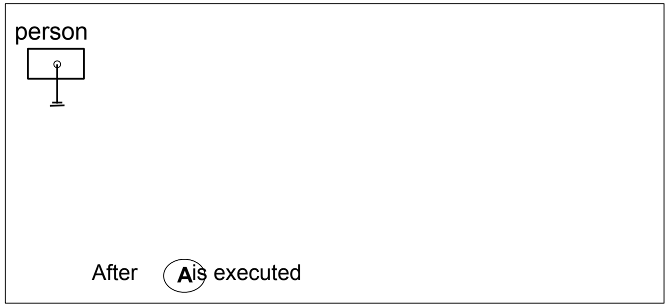
Code

A

```
Person[ ]  person;
person = new Person[20];
person[0] = new Person( );
```

Only the name person is declared, no array is allocated yet.

State of Memory



After A is executed

Creating an Object Array - 2

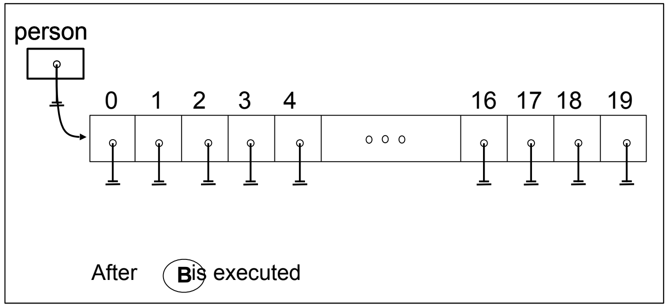
Code

B

```
Person[ ]  person;
person = new Person[20];
person[0] = new Person( );
```

Now the array for storing 20 Person objects is created, but the Person objects themselves are not yet created.

State of Memory



After B is executed

Creating an Object Array - 3

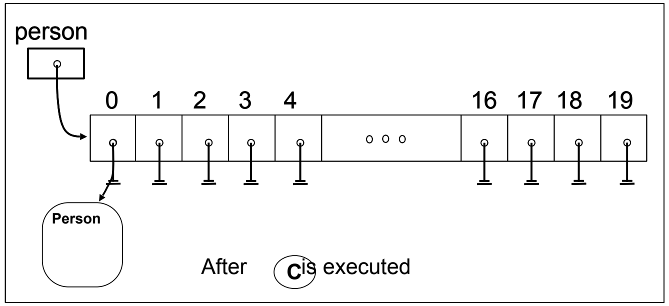
Code

C

```
Person[ ]  person;
person = new Person[20];
person[0] = new Person( );
```

One Person object is created and the reference to this object is placed in position 0.

State of Memory



After C is executed

Person Array Processing – Sample 1

- Create Person objects and set up the person array.

```
String     name, inpStr; int age; char gender;
Scanner scanner = new Scanner(System.in);

for (int i = 0; i < person.length; i++) {
    System.out.print("Enter name:"); name = scanner.next ( );
    System.out.print("Enter age:");  age = scanner.nextInt( );
    System.out.print("Enter gender:"); inpStr = scanner.next( );
    gender = inpStr.charAt(0);

    person[i] = new Person( );    //create a new Person and assign values

    person[i].setName  ( name   );
    person[i].setAge   (  age   );
    person[i].setGender( gender );
}
```

Person Array Processing – Sample 2

- Find the youngest and oldest persons.

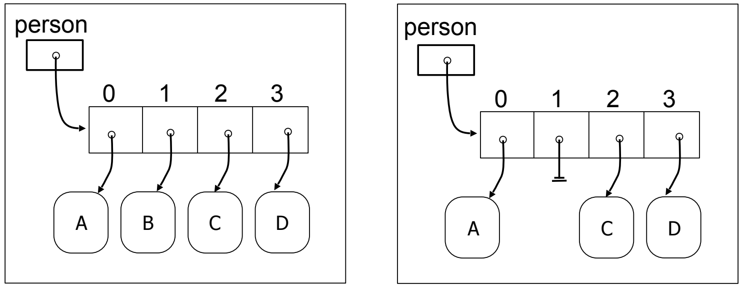
```
int     minIdx = 0;      //index to the youngest person
int     maxIdx = 0;      //index to the oldest person

for (int i = 1; i < person.length; i++) {

    if ( person[i].getAge() < person[minIdx].getAge() ) {
        minIdx      = i;        //found a younger person

    } else if (person[i].getAge() > person[maxIdx].getAge() ) {

        maxIdx       = i;       //found an older person
    }
}

//person[minIdx] is the youngest and person[maxIdx] is the oldest
```

Object Deletion – Approach 1

A

```
int delIdx = 1;

person[delIdx] = null;
```

Delete Person B by setting the reference in position 1 to null.



Before A is executed

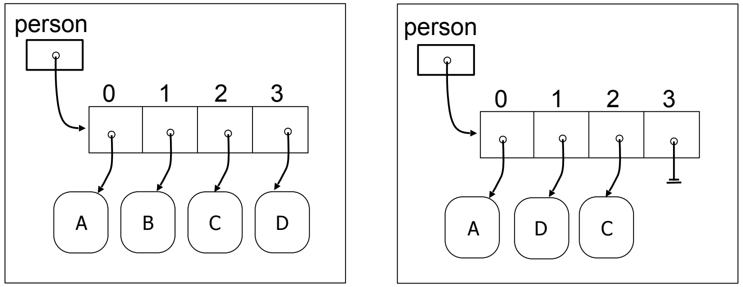
After A is executed

Object Deletion – Approach 2

A

```
int delIdx = 1, last = 3;

person[delIndex] = person[last];
person[last]     = null;
```

Delete Person B by setting the reference in position 1 to the last person.



Before A is executed

After A is executed

Person Array Processing – Sample 3

- Searching for a particular person. Approach 2 Deletion is used.

```
int i = 0;

while ( person[i] != null && !person[i].getName().equals("Latte") ) {
    i++;
}

if ( person[i] == null ) {
    //not found - unsuccessful search
    System.out.println("Ms. Latte was not in the array");

} else {
    //found - successful search
     System.out.println("Found Ms. Latte at position " + i);
}
```

The For-Each Loop

- This new for loop is available from Java 5.0
- The for-each loop simplifies the processing of elements in a collection
- Here we show examples of processing elements in an array

```
int sum = 0;

for (int i = 0; i < number.length; i++) {
    sum = sum + number[i];
}
```

standard for loop

```
int sum = 0;

for (int value : number) {
    sum = sum + value;
}
```

for-each loop

Processing an Array of Objects with For-Each

```
Person[] person = new Person[100];
//create person[0] to person[99]
```

```
for (int i = 0; i < person.length; i++) {
    System.out.println(person[i].getName());
}
```

standard for loop

```
for (Person p : person) {
    System.out.println(p.getName());
}
```

for-each loop

For-Each: Key Points to Remember

- A for-each loop supports read access only. The elements cannot be changed.
- A single for-each loop allows access to a single array only, i.e., you cannot access multiple arrays with a single for-each loop.
- A for-each loop iterates over every element of a collection from the first to the last element. You cannot skip elements or iterate backward.

Passing Arrays to Methods - 1

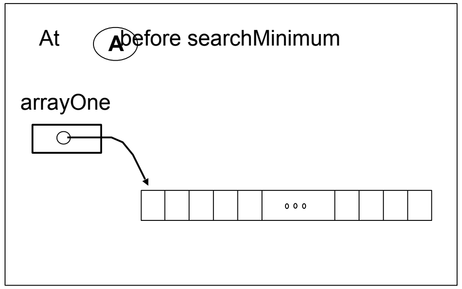
Code

A

```
minOne
= searchMinimum(arrayOne);
```

```
public int searchMinimum(float[] number))
{

    …

}
```

State of Memory



At A before searchMinimum

A. Local variable number does not exist before the method execution

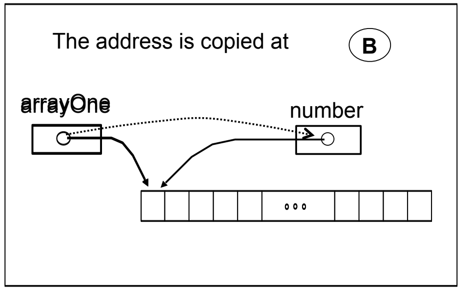
Passing Arrays to Methods - 2

Code

```
minOne
= searchMinimum(arrayOne);
```

```
public int searchMinimum(float[] number))
{

    …

}
```

B



State of Memory

B. The value of the argument, which is an address, is copied to the parameter.

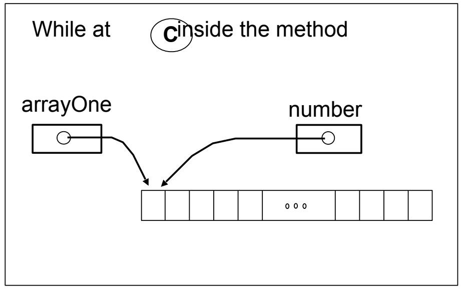
Passing Arrays to Methods - 3

Code

```
minOne
= searchMinimum(arrayOne);
```

```
public int searchMinimum(float[] number))
{

    …

}
```

C



State of Memory

C. The array is accessed via number inside the method.

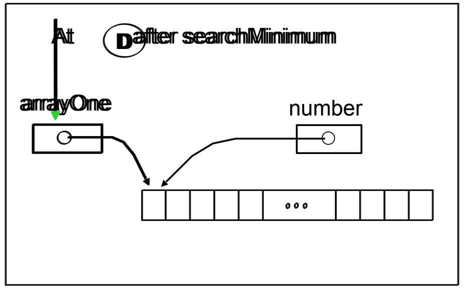
Passing Arrays to Methods - 4

Code

```
minOne
= searchMinimum(arrayOne);
```

D

```
public int searchMinimum(float[] number))
{

    …

}
```



State of Memory

D. The parameter is erased. The argument still points to the same object.

Two-Dimensional Arrays

- Two-dimensional arrays are useful in representing tabular information.

Distance Table (in miles)

	Los Angeles	San Francisco	San Jose	San Diego	Monterey
Los Angeles	—	600	500	150	450
San Francisco	600	—	100	750	150
San Jose	500	100	—	650	50
San Diego	150	750	650	—	600
Monterey	450	150	50	600	—

Multiplication Table

	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

Tuition Table

	Day Students	Boarding Students
Grades 1 – 6	$ 6,000.00	$ 18,000.00
Grades 7 – 8	$ 9,000.00	$ 21,000.00
Grades 9 – 12	$ 12,500.00	$ 24,500.00

Declaring and Creating a 2-D Array

Declaration

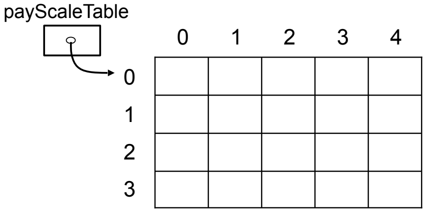
```
<data type> [][] <variable>     //variation 1
<data type>   <variable>[][]    //variation 2
```

Creation

```
<variable>   = new <data type> [ <size1> ]
[ <size2> ]
```

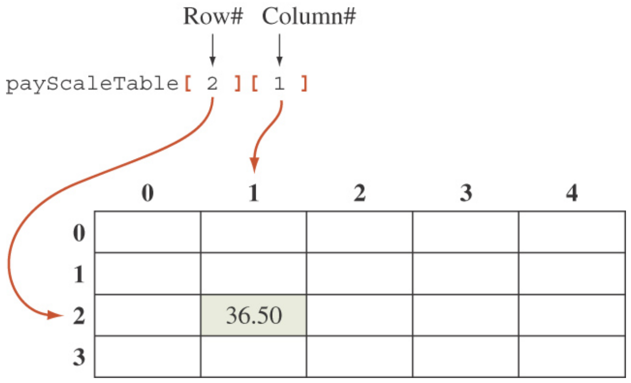
Example

```
double[][] payScaleTable;
payScaleTable
    = new double[4][5];
```



Accessing an Element

- An element in a two-dimensional array is accessed by its row and column index.



Sample 2-D Array Processing

- Find the average of each row.

```
double[ ] average = { 0.0, 0.0, 0.0, 0.0 };

for (int i = 0; i < payScaleTable.length; i++) {

    for (int j = 0; j < payScaleTable[i].length; j++) {

        average[i] += payScaleTable[i][j];
    }

    average[i] = average[i] / payScaleTable[i].length;
}
```

Java Implementation of 2-D Arrays

- The sample array creation

```
payScaleTable = new double[4][5];
```

is really a shorthand for

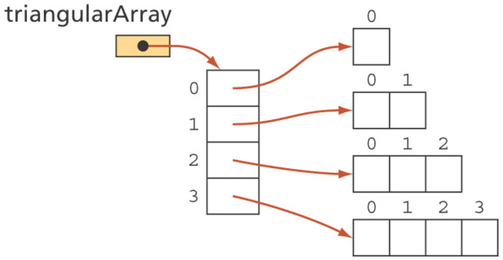
```
payScaleTable = new double [4][ ];

payScaleTable[0] = new double [5];
payScaleTable[1] = new double [5];
payScaleTable[2] = new double [5];
payScaleTable[3] = new double [5];
```

Two-Dimensional Arrays

- Subarrays may be different lengths.
- Executing

```
triangularArray = new double[4][ ];
for (int i = 0; i < 4; i++)
   triangularArray[i] = new double [i + 1];
```

results in an array that looks like:



Collection Classes: Lists and Maps

- The **java.util** standard package contains different types of classes for maintaining a collection of objects.
- These classes are collectively referred to as the *Java Collection Framework (JCF).*
- JCF includes classes that maintain collections of objects as sets, lists, or maps.

Java Interface

- A Java interface defines only the behavior of objects
  - It includes only public methods with no method bodies.
  - It does not include any data members except public constants
  - No instances of a Java interface can be created

JCF Lists

- JCF includes the **List** interface that supports methods to maintain a collection of objects as a linear list

$$L = (l_0, l_1, l_2, . . . , l_N)$$

- We can add to, remove from, and retrieve objects in a given list.
- A list does not have a set limit to the number of objects we can add to it.

List Methods

- Here are five of the 25 list methods:

`boolean add    ( E   o )`
Adds an object o to the list
`void    clear  (        )`
Clears this list, i.e., make the list empty
`E      get    ( int idx  )`
Returns the element at position idx
`boolean remove ( int idx  )`
Removes the element at position idx
`int      size   (           )`
Returns the number of elements in the list

E is a generic class.
Replace E with a concrete class.

Using Lists

- To use a list in a program, we must create an instance of a class that implements the List interface.
- Two classes that implement the **List** interface:
  - **ArrayList**
  - **LinkedList**
- The **ArrayList** class uses an array to manage data.
- The **LinkedList** class uses a technique called *linked-node representation*.

Homogeneous vs. Heterogeneous Collections

- Heterogeneous collections can include any types of objects (Person, Integer, Dog, etc.)
- Homogenous collections can include objects from a designated class only.
  - Designate the class in the collection declaration.
  - For example, to declare and create a list (ArrayList) of Person objects, we write

```
List<Person> friends;
…
friends = new ArrayList<Person>( ) ;
```

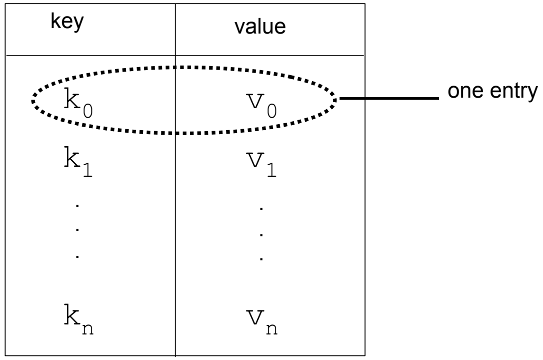
Sample List Usage

- Here's an example of manipulating a list of Person objects:

```
import java.util.*;

List<Person>  friends;
Person        person;

friends = new ArrayList<Person>( );

person = new Person("jane", 10, 'F');
friends.add( person );
person = new Person("jack",  6, 'M');
friends.add( person );

Person p = friends.get( 1 );
```

JCF Maps

- JCF includes the **Map** interface that supports methods to maintain a collection of objects (key, value) pairs called map entries.



key	value
k_0	v_0
k_1	v_1
.	.
.	.
.	.
k_n	v_n

one entry

Map Methods

- Here are five of the 14 list methods:

`void clear ( )`
Clears this list, i.e., make the map empty
`boolean containsKey ( Object key )`
Returns true if the map contains an entry with a given key
`V put (K key, V value)`
Adds the given (key, value) entry to the map
`V remove ( Object key )`
Removes the entry with the given key from the map
`int size ( )`
Returns the number of elements in the map

Using Maps

- To use a map in a program, we must create an instance of a class that implements the Map interface.
- Two classes that implement the **Map** interface:
  - **HashMap**
  - **TreeMap**

Sample Map Usage

- Here's an example of manipulating a map:

```
import java.util.*;

Map     catalog;
catalog = new TreeMap<String, String>( );

catalog.put("CS101", "Intro Java Programming");
catalog.put("CS301", "Database Design");
catalog.put("CS413", "Software Design for Mobile Devices");

if (catalog.containsKey("CS101")) {
    System.out.println("We teach Java this semester");
} else {
    System.out.println("No Java courses this semester");
}
```

Problem Statement

Write an AddressBook class that manages a collection of Person objects. An AddressBook object will allow the programmer to add, delete, or search for a Person object in the address book.

Overall Plan / Design Document

- Since we are designing a single class, our task is to identify the public methods.

Public Method	Purpose
`AddressBook`	A constructor to initialize the object. We will include multiple constructors as necessary.
`add`	Adds a new Person object to the address book.
`delete`	Deletes a specified Person object from the address book.
`search`	Searches a specified Person object in the address book and returns this person if found.

Development Steps

- We will develop this program in five steps:

1. Implement the constructor(s).
2. Implement the add method.
3. Implement the search method.
4. Implement the delete method.
5. Finalize the class.

Step 1 Design

- Start the class definition with two constructors
- The zero-argument constructor will create an array of default size
- The one-argument constructor will create an array of the specified size

Step 1 Code

Program source file is too big to list here. From now on, we ask you to view the source files using your Java IDE.

Directory: Chapter10/Step1

Source Files: AddressBook.java

Step 1 Test

- The purpose of Step 1 testing is to verify that the constructors work as expected.

Argument to Constructor	Purpose
`Negative numbers`	Test the invalid data.
`0`	Test the end case of invalid data.
`1`	Test the end case of valid data.
`>= 1`	Test the normal cases.

Step 2 Design

- Design and implement the add method
- The array we use internal to the AddressBook class has a size limit, so we need consider the overflow situation
  - Alternative 1: Disallow adds when the capacity limit is reached
  - Alternative 2: Create a new array of bigger size
- We will adopt Alternative 2

Step 2 Code

Directory: Chapter10/Step2

Source Files: AddressBook.java

Step 2 Test

- The purpose of Step 2 test is to confirm that objects are added correctly and the creation of a bigger array takes place when an overflow situation occurs.

Test Sequence	Purpose
`Create the array of size 4`	Test that the array is created correctly.
`Add four Person objects`	Test that the Person objects are added correctly.
`Add the fifth Person object`	Test that the new array is created and the Person object is added correctly (to the new array).

Step 3 Design

- Design and implement the search method.

```
loc = 0;
while ( loc < count &&
        name of Person at entry[loc] is not equal to
        the given search name ) {
    loc++;
}
if (loc == count) {
    foundPerson = null;
} else {
    foundPerson = entry[loc];
}

return foundPerson;
```

Step 3 Code

Directory: Chapter10/Step3

Source Files: AddressBook.java

Step 3 Test

- To test the correct operation of the search method, we need to carry out test routines much more elaborate than previous tests.

Test Sequence	Purpose
`Create the array of size 5 and add five Person objects with unique names.`	Test that the array is created and set up correctly. Here, we will test the case where the array is 100 percent filled.
`Search for the person in the first position of the array`	Test that the successful search works correctly for the end case.
`Search for the person in the last position of the array`	Test another version of the end case.
`Search for a person somewhere in the middle of the array.`	Test the normal case.
`Search for a person not in the array.`	Test for the unsuccessful search.
`Repeat the above steps with an array of varying sizes, especially the array of size 1.`	Test that the routine works correctly for arrays of different sizes.
`Repeat the testing with the cases where the array is not fully filled, say, array length is 5 and the number of objects in the array is 0 or 3.`	Test that the routine works correctly for other cases.

Step 4 Design

- Design and implement the delete method.

```
boolean status;
int       loc;
loc = findIndex( searchName );

if ( loc is not valid ) {
    status = false;
} else { //found, pack the hole
    replace the element at index loc+1 by the last element
    at index count;

    status = true;

    count--;     //decrement count, since we now have one less element
    assert 'count' is valid;
}
return status;
```

Step 4 Code

Directory: Chapter10/Step4

Source Files: AddressBook.java

Step 4 Test

- To test the correct operation of the delete method, we need to carry out a detailed test routine.

Test Sequence	Purpose
Create the array of size 5 and add five Person objects with unique names.	Test the array is created and set up correctly. Here, we will test the case where the array is 100 percent filled.
Search for a person to be deleted next.	Verify that the person is in the array before deletion.
Delete the person in the array	Test that the delete method works correctly.
Search for the deleted person.	Test that the delete method works correctly by checking the value null is returned by the search.
Attempt to delete a nonexisting person.	Test that the unsuccessful operation works correctly.
Repeat the above steps by deleting persons at the first and last positions.	Test that the routine works correctly for arrays of different sizes.
Repeat testing where the array is not fully filled, say, an array length is 5 and the number of objects in the array is 0 or 3.	Test that the routine works correctly for other cases.

Step 5: Finalize

- Final Test
  - Since the three operations of add, delete, and search are interrelated, it is critical to test these operations together. We try out various combinations of add, delete, and search operations.
- Possible Extensions
  - One very useful extension is scanning. Scanning is an operation to visit all elements in the collection.
  - Scanning is useful in listing all Person objects in the address book.