Objectives

After this week, you should be able to

- Manipulate a collection of data values, using an array.
- Declare and use arrays
- Define a method that accepts an array as its parameter and a method that returns an array
- Understand two-dimensional arrays.
- Manipulate a collection of objects, using lists and maps
Array Basics

- Suppose your program needs to deal with 100 integers, 500 Account objects, 365 real numbers, etc., having a separate variable for each of these is cumbersome.

- Instead, we can use an array for each collection of similar values or objects.

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

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

The index of the first position in an array is 0.
```java
double[] rainfall = new double[12];
double annualAverage,
    sum = 0.0;
for (int i = 0; i < rainfall.length; i++) {
    rainfall[i] = Double.parseDouble(
        JOptionPane.showInputDialog(null,
            "Rainfall for month "+(i+1)));
    sum += rainfall[i];
}
annualAverage = sum / rainfall.length;
```

The public constant `length` returns the size of an array.
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++) {
    rainfall[i] = Double.parseDouble(
        JOptionPane.showInputDialog(null,
            "Rainfall for "
            + monthName[i] ));
    sum += rainfall[i];
}

annualAverage = sum / rainfall.length;

The same pattern for the remaining ten months.
The actual month name instead of a number.
Compute the average rainfall for each quarter.

```java
//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
}
```
Index out of bounds

- The index for an array `a`, must evaluate to a value between `0` and `a.length-1`.
- If it does not, then a run time exception called `ArrayIndexOutOfBoundsException`.
- This does not need to be caught -- but will cause the program to terminate if not caught.
Array Initialization

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

```java
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 };
```

- `number.length` → 4
- `samplingData.length` → 9
- `monthName.length` → 12
Initializing arrays

- If we do not initialize values at creation time, then the elements are initialized to the default value of the corresponding type.
- It is also common to initialize an array using a for loop.
In Java, we are not required to declare the size at compile time.

The following code prompts the user for the size of an array and declares an array of the designated size:

```java
int size;
int[] number;
size = Integer.parseInt(JOptionPane.showInputDialog(null, "Size of an array:"));
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.

```java
Person student;
student = new Person();
student.setName("Doe");
student.setAge(20);
student.setGender('F');

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

The Person class supports the set methods and get methods.
Creating an Object Array - 1

**Code**

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

**State of Memory**

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

After [A] is executed
Creating an Object Array - 2

**Code**

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

**State of Memory**

After **B** is executed

Now the array for storing 20 Person objects is created, but the Person objects themselves are not yet created.
Creating an Object Array - 3

**Code**

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

**State of Memory**

After code is executed

One Person object is created and the reference to this object is placed in position 0.
Create Person objects and set up the person array.

```java
String name, inpStr;
int age;
char gender;

for (int i = 0; i < person.length; i++) {
    name  = inputBox.getString("Enter name:"); //read in data values
    age  = inputBox.getInteger("Enter age:");
    inpStr = inputBox.getString("Enter gender:");
    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 );
}
```
Find the youngest and oldest persons.

```java
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

```java
int delIdx = 1;
person[delIdx] = null;
```

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

Before Person B is executed:

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

After Person B is executed:

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Object Deletion – Approach 2

```c
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
Searching for a particular person. Approach 2
Deletion is used.

```java
int i = 0
while (person[i] != null && !person[i].getName().equals("Doe")) {
    i++;
}

if (person[i] == null) {
    //not found - unsuccessful search
    System.out.println("Doe was not in the array");
} else {
    //found - successful search
    System.out.println("Found Doe at position " + i);
}
```
Array data type

- An array with elements of type T is a new data type represented as T[].
  - `int[][] age;`
  - `double[] salary[];`
  - `Person[] student[];`
    - `age` is of type `int[]`
    - `salary` is of type `double[]`
    - `student` is of type `Person[]`

- Each element of this array is of type T
  - `age[0]`, `salary[0]` are int data types
  - `student[1]` is a Person object.
Passing Arrays to Methods - 1

Code

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

State of Memory

At before `searchMinimum`

Array `arrayOne`

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

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

Purdue University
The address is copied at B.

The value of the argument, which is an address, is copied to the parameter.
minOne = searchMinimum(arrayOne);

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

public int searchMinimum(float[] number))
{
    ...
}

While at C inside the method
minOne = searchMinimum(arrayOne);

public int searchMinimum(float[] number))
{
    ...
}

D. The parameter is erased. The argument still points to the same object.
Arguments and return values

- An array can be returned by a method.
- The return type must be an array in this case.
  ```java
  public int[] doubleValues(int[] inArray)
  ```
- An element can be passed to any method that accepts an argument of the base type of the array.
  ```java
  double x[] = new double[5];
y = Math.exp(x[2]);
  ```
The main method

- Recall the only argument to main:
  `public static void main(String[] args)`
- The argument is an array of strings. Each element of this array is set to the words that follow the program name when executing:
  `%java Test one two three`
- In main: `args[0]` is “one” `args[1]` is “two” and `args[2]` is three.
- Also, `args.length` will be 3 for this case.
Two-Dimensional Arrays

Two-dimensional arrays are useful in representing tabular information.

<table>
<thead>
<tr>
<th>Distance Table (in miles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Los Angeles</td>
</tr>
<tr>
<td>Los Angeles</td>
</tr>
<tr>
<td>San Francisco</td>
</tr>
<tr>
<td>San Jose</td>
</tr>
<tr>
<td>San Diego</td>
</tr>
<tr>
<td>Monterey</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multiplication Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tuition Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grades 1 – 6</td>
</tr>
<tr>
<td>Grades 7 – 8</td>
</tr>
<tr>
<td>Grades 9 – 12</td>
</tr>
</tbody>
</table>
Declaring and Creating a 2-D Array

Declaration

\[
\text{<data type> [][] } \text{<variable>} \quad //\text{variation 1}
\]

\[
\text{<data type> } \text{<variable>[][]} \quad //\text{variation 2}
\]

Creation

\[
\text{<variable> } = \textbf{new} \text{ <data type> } [ \text{<size1>} ][ \text{<size2>} ]
\]

Example

```java
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.

```java
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;
}
```
Sample 2-D Array Processing

- Find the average of each column.

```java
double[] average = { 0.0, 0.0, 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[j] += payScaleTable[i][j];
    }
}
for (int i = 0; i < payScaleTable[0].length; i++)
    average[i] = average[i] / payScaleTable.length;
```
Sample 2-D Array Processing

- Find the average of each column -- alternative.

```java
double[] average = { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 };  
for (int i = 0; i < payScaleTable[0].length; i++) {
    for (int j = 0; j < payScaleTable.length; j++) {
        average[i] += payScaleTable[j][i];
    }
    average[i] = average[i] / payScaleTable.length;
}
```
Java Implementation of 2-D Arrays

- The sample array creation

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

is really a shorthand for

```java
payScaleTable = new double [4][ ];
payScaleTable[0] = new double [5];
payScaleTable[1] = new double [5];
payScaleTable[2] = new double [5];
payScaleTable[3] = new double [5];
```
Java Implementation

payScaleTable = new double[4][5];

payScaleTable = new double [4][];
payScaleTable[0] = new double [5];
Java Implementation

```java
payScaleTable.length → 4
payScaleTable[1].length → 5
payScaleTable[1][2].length → ERROR!
```

![Diagram of payScaleTable]

Pay Scale Table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Two-Dimensional Arrays

- In Java, subarrays may have different lengths.
- Executing
  ```java
  triangularArray = new double[4][ ];
  for (int i = 0; i < 4; i++)
      triangularArray[i] = new double [i + 1];
  ```
  results in an array that looks like:

```
0   0   0   0
0   1   0
0   0   1
0   0   0
```

triangularArray.length

triangularArray[1].length
Limitation of Arrays

- Once an array object is created, its size is fixed -- it cannot be changed.

- If we need to store more elements than the size with which an array was created, we have to
  - Create a new larger array
  - Copy all elements from current to new array
  - Change the reference to the new array

- Alternatively, we can use Lists or Maps.
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.
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.
The **List** interface

- JCF includes the **List** interface that supports methods to maintain a collection of objects as a linear list $L = (l_0, l_1, l_2, \ldots, l_N)$
- 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.
Here are five of the 25 list methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean add</td>
<td>(Object o) Adds an object o to the list</td>
</tr>
<tr>
<td>void clear</td>
<td>() Clears this list, i.e., make the list empty</td>
</tr>
<tr>
<td>Object get</td>
<td>(int idx) Returns the element at position idx</td>
</tr>
<tr>
<td>boolean remove</td>
<td>(int idx) Removes the element at position idx</td>
</tr>
<tr>
<td>int size</td>
<td>() Returns the number of elements in the list</td>
</tr>
</tbody>
</table>
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.
Sample List Usage

Here's an example of manipulating a list:

```java
import java.util.*;
List friends;
Person person;
friends = new ArrayList();

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

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

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

Note: `List` is the interface, `ArrayList` implements it.

Get returns an object of class `Object` -- need to typecast to `Person`.
JCF includes the **Map** interface that supports methods to maintain a collection of objects (key, value) pairs called map entries.

Here is a table to illustrate:

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_0$</td>
<td>$v_0$</td>
</tr>
<tr>
<td>$k_1$</td>
<td>$v_1$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$k_n$</td>
<td>$v_n$</td>
</tr>
</tbody>
</table>
Map Methods

- Here are five of the 14 list methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void clear()</code></td>
<td>Clears this list, i.e., make the map empty</td>
</tr>
<tr>
<td><code>boolean containsKey(Object key)</code></td>
<td>Returns true if the map contains an entry with a given key</td>
</tr>
<tr>
<td><code>Object put(Object key, Object value)</code></td>
<td>Adds the given (key, value) entry to the map</td>
</tr>
<tr>
<td><code>boolean remove(Object key)</code></td>
<td>Removes the entry with the given key from the map</td>
</tr>
<tr>
<td><code>int size()</code></td>
<td>Returns the number of elements in the map</td>
</tr>
</tbody>
</table>
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
Here's an example of manipulating a map:

```java
import java.util.*;

Map catalog;
catalog = new TreeMap();

catalog.put("CS180", "Intro Java Programming");
catalog.put("CS348", "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");
}
```
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.
Since we are designing a single class, our task is to identify the public methods.

<table>
<thead>
<tr>
<th>Public Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddressBook</td>
<td>A constructor to initialize the object. We will include multiple constructors as necessary.</td>
</tr>
<tr>
<td>add</td>
<td>Adds a new Person object to the address book.</td>
</tr>
<tr>
<td>delete</td>
<td>Deletes a specified Person object from the address book.</td>
</tr>
<tr>
<td>search</td>
<td>Searches a specified Person object in the address book and returns this person if found.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Argument to Constructor</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative numbers</td>
<td>Test the invalid data.</td>
</tr>
<tr>
<td>0</td>
<td>Test the end case of invalid data.</td>
</tr>
<tr>
<td>1</td>
<td>Test the end case of valid data.</td>
</tr>
<tr>
<td>$\geq 1$</td>
<td>Test the normal cases.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Test Sequence</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create the array of size 4</td>
<td>Test that the array is created correctly.</td>
</tr>
<tr>
<td>Add four Person objects</td>
<td>Test that the Person objects are added correctly.</td>
</tr>
<tr>
<td>Add the fifth Person object</td>
<td>Test that the new array is created and the Person object is added correctly (to the new array).</td>
</tr>
</tbody>
</table>
Step 3 Design

Design and implement the **search** method.

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

### Test Sequence

<table>
<thead>
<tr>
<th>Test Sequence</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create the array of size 5 and add five Person objects with unique names.</td>
<td>Test that the array is created and set up correctly. Here, we will test the case where the array is 100 percent filled.</td>
</tr>
<tr>
<td>Search for the person in the first position of the array</td>
<td>Test that the successful search works correctly for the end case.</td>
</tr>
<tr>
<td>Search for the person in the last position of the array</td>
<td>Test another version of the end case.</td>
</tr>
<tr>
<td>Search for a person somewhere in the middle of the array.</td>
<td>Test the normal case.</td>
</tr>
<tr>
<td>Search for a person not in the array.</td>
<td>Test for the unsuccessful search.</td>
</tr>
<tr>
<td>Repeat the above steps with an array of varying sizes, especially the array of size 1.</td>
<td>Test that the routine works correctly for arrays of different sizes.</td>
</tr>
<tr>
<td>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.</td>
<td>Test that the routine works correctly for other cases.</td>
</tr>
</tbody>
</table>
Design and implement the **delete** method.

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

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