Chapter 6

Arrays

- Array Basics
- Arrays in Classes and Methods
- Programming with Arrays and Classes
- Sorting Arrays
- Multidimensional Arrays

Overview

- An array: a single name for a collection of data values, all of the same data type
  - » subscript notation identifies precisely one of the values
- Arrays are a carryover from earlier programming languages
- Array: more than a primitive type, less than an object
  - » their methods are invoked with a special subscript notation
    - – most programmers do not even think of them as methods
  - » they work like objects when used as method arguments and return types
  - » they do not have or use inheritance
  - » they are sort of like a Java class that is not fully implemented
- Arrays are a natural fit for loops, especially `for` loops

Creating Arrays

- General syntax for declaring an array:

```
Base_Type[] Array_Name = new Base_Type[Length];
```

- Examples:
  80-element array with base type `char`:
  ```
  char[] symbol = new char[80];
  ```
  100-element array of `doubles`:
  ```
  double[] reading = new double[100];
  ```
  80-element array of `Species`:
  ```
  Species[] specimen = new Species[100];
  ```

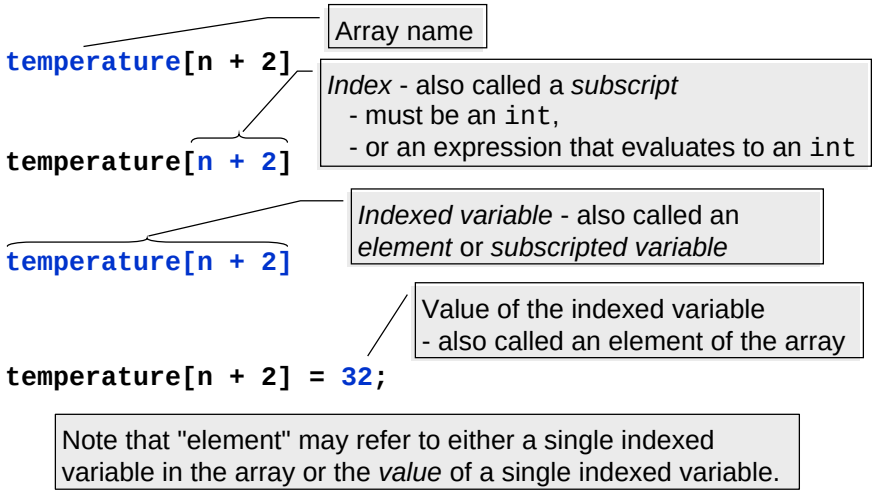

Three Ways to Use [ ] (Brackets) with an Array Name

1. To create a type name, e.g. `int[] pressure;` creates a name with the type "`int` array"
   - » note that the types `int` and `int` array are different
   - » it is the type of the name, not the type of the data

2. To create a new array, e.g. `pressure = new int[100];`

3. To name a specific element in the array
   - also called *an indexed variable*, e.g.
   ```
   pressure[3] = SavitchIn.readLineInt();
   System.out.println("You entered" + pressure[3]);
   ```

Some Array Terminology



Note that "element" may refer to either a single indexed variable in the array or the *value* of a single indexed variable.

Array Length

- Length of an array is specified by the number in brackets when it is created with new
  - » it determines the amount of memory allocated for the array elements (values)
  - » it determines the *maximum* number of elements the array can hold
    - – storage is allocated whether or not the elements are assigned values
- The array length can be read with the instance variable `length`, e.g. the following code displays the number 20 (the *size*, or *length* of the `Species` array, `entry`):

```
Species[] entry = new Species[20];
System.out.println(entry.length);
```

- The length attribute is established in the declaration and cannot be changed unless the array is redeclared

Initializing an Array's Values in Its Declaration

- Array elements can be initialized in the declaration statement by putting a comma-separated list in braces
- Uninitialized elements will be assigned some default value, e.g. 0 for `int` arrays
- The length of an array is automatically determined when the values are explicitly initialized in the declaration
- For example:

```
double[] reading = {5.1, 3.02, 9.65};
System.out.println(readings.length);
```

  - displays 3, the length of the array `readings`

Subscript Range

- Array subscripts use zero-numbering
  - » the first element has subscript 0
  - » the second element has subscript 1
  - » etc. - the n^{th} element has subscript n-1
  - » the last element has subscript `length-1`
- For example:

```
int[] scores = {97, 86, 92, 71};
```

Subscript:	0	1	2	3
Value:	97	86	92	71

Subscript out of Range Error

- Using a subscript larger than `length-1` causes a *run time* (not a compiler) error
  - » an `ArrayOutOfBoundsException` is thrown
    - – you do not need to catch it or declare it in a throws-clause
    - – you need to fix the problem and recompile your code
- Other programming languages, e.g. C and C++, do not even cause a run time error!
  - » one of the most dangerous characteristics of these languages is that they allow out of bounds array indexes.

Initializing Array Elements in a Loop

- Array processing is easily done in a loop
- For example, a `for` loop is commonly used to initialize array elements
- For example:

```
int i;//loop counter/array index
int[] a = new int[10];
for(i = 0; i < a.length; i++)
   a[i] = 0;
```

  - » note that the loop counter/array index goes from `0` to `length - 1`
  - » it counts through `length = 10` iterations/elements using the zero-numbering of the array index

Arrays, Classes, and Methods

An array of a class can be declared and the class's methods applied to the elements of the array.

This excerpt from Sales Report program in the text uses the `SalesAssociate` class to create an array of sales associates:

create an array of `SalesAssociates`

each array element is a `SalesAssociate` instance variable

use the `readInput` method of `SalesAssociate`

```
public void getFigures()
{
   System.out.println("Enter number of sales associates:");
   numberOfAssociates = SavitchIn.readLineInt();
   record = new SalesAssociate[numberOfAssociates];
   int i;
   for (i = 0; i < numberOfAssociates; i++)
   {
      record[i] = new SalesAssociate();
      System.out.println("Enter data for associate " + (i + 1));
      record[i].readInput();
      System.out.println();
   }
}
```


Arrays and Array Elements as Method Arguments

Arrays and array elements can be used with classes and methods just like other objects

- both an indexed element and an array name can be an argument in a method
- methods can return an array value or an array name

Indexed Variables as Method Arguments

`nextScore` is an array of `ints`

an element of `nextScore` is an argument of method `average`

`average` method definition

```
public static void main(String arg[])
{
   System.out.println("Enter your score on exam 1:");
   int firstScore = SavitchIn.readLineInt();
   int[ ] nextScore = new int[3];
   int i;
   double possibleAverage;
   for (i = 0; i < nextScore.length; i++)
      nextScore[i] = 80 + 10*i;
   for (i = 0; i < nextScore.length; i++)
   {
      possibleAverage = average(firstScore, nextScore[i]);
      System.out.println("If your score on exam 2 is "
                         + nextScore[i]);
      System.out.println("your average will be "
                         + possibleAverage);
   }
}
public static double average(int n1, int n2)
{
  return (n1 + n2)/2.0;
}
```

Excerpt from ArgumentDemo program in text.

When Can a Method Change an Indexed Variable Argument?

Remember:

- primitive types are call-by-value
  - » only a copy of the value is passed as an argument in a method call
  - » so the method *cannot* change the value of the indexed variable
- class types are reference types; they pass the address of the object when they are an argument in a method call
  - » the corresponding argument in the method definition becomes another name for the object
  - » the method has access to the actual object
  - » so the method *can* change the value of the indexed variable if it is a class (and not a primitive) type

Array Names as Method Arguments

When using an entire array as an argument to a method:

- » use just the array name and no brackets
- » as described in the previous slide, the method has access to the original array and can change the value of the elements
- » the length of the array passed can be different for each call
  - – when you define the function you do not know the length of the array that will be passed
  - – so use the `length` attribute inside the method to avoid `ArrayIndexOutOfBoundsExceptions`

Example: An Array as an Argument in a Method Call

```
public static void
   showArray(char[] a)
{
    int i;
    for(i = 0; i < a.length;
   i++)
        System.out.println(a[i]);
}
```

the method's argument is the name of an array of characters

uses the `length` attribute to control the loop
allows different size arrays and avoids index-out-of-bounds exceptions

Arguments for the Method **`main`**

- The heading for the `main` method shows a parameter that is an array of `Strings`:

  `public static void main(`**`String[] args`**`)`
- When you run a program from the command line, all words after the class name will be passed to the main method in the args array.

  `Java TestProgram` **`Josephine Student`**
- The following `main` method in the class `TestProgram` will print out the first two arguments it receives:

```
Public static void main(String[] args)
{
   System.out.println("Hello " + args[0] + " " + args[1]);
}
```

- In this example, the output from the command line above will be:

  `Hello Josephine Student`

Using = with Array Names: Remember They Are Reference Types

```
int[] a = new int[3];
int[] b = new int[3];
for(int i; i < a.length; i++)
   a[i] = i;
b = a;
System.out.println(a[2] + " " + b[2]);
a[2] = 10;
System.out.println(a[2] + " " + b[2]);
```

This does not create a copy of array a; it makes b another *name* for array a.

The output for this code will be:

```
2 2
10 10
```

A value changed in a is the same value obtained with b

Using == with array names: remember they are reference types

```
int i;
int[] a = new int[3];
int[] b = new int[3];
for(i; i < a.length; i++)
   a[i] = i;
for(i; i < b.length; i++)
   b[i] = i;
if(b == a)
   System.out.println("a equals b");
else
 System.out.println("a does not equal b");
```

a and b are both 3-element arrays of `int`s

all elements of a and b are assigned the value `0`

tests if the *addresses* of a and b are equal, not if the array values are equal

The output for this code will be "`a does not equal b`" because the *addresses* of the arrays are not equal.

Testing Two Arrays for Equality

- To test two arrays for equality you need to define an `equals` method that returns true if and only the arrays have the same length and all corresponding values are equal
- This code shows an example of an `equals` method.

```
public static boolean equals(int[] a, int[] b)
 {
    boolean match;
    if (a.length != b.length)
       match = false;
    else
    {
       match = true; //tentatively
       int i = 0;
       while (match && (i < a.length))
       {
          if (a[i] != b[i])
             match = false;
          i++;
       }
    }
    return match;
 }
```

Methods that Return an Array

- Yet another example of passing a reference
- Actually, the array is not passed, the address of the array is passed
- The local array name within the method is just another name for the original array
- The code at right shows an example of returning an array

```
public class returnArrayDemo
 {
    public static void main(String arg[])
    {
       char[] c;
       c = vowels();
       for(int i = 0; i < c.length; i++)
         System.out.println(c[i]);
    }
    public static char[] vowels()
    {
       char[] newArray = new char[5];
       newArray[0] = 'a';
       newArray[0] = 'e';
       newArray[0] = 'i';
       newArray[0] = 'o';
       newArray[0] = 'u';
       return newArray;
    }
 }
```

c, `newArray`, and the return type of `vowels` are all the same type: `char array name`

Good Programming Practice

- Using singular rather than plural names for arrays improves readability
  - » although the array contains many elements the most common use of the name will be with a subscript, which references a *single* value
- Do not count on default initial values for array elements
  - » explicitly initialize elements in the declaration or in a loop

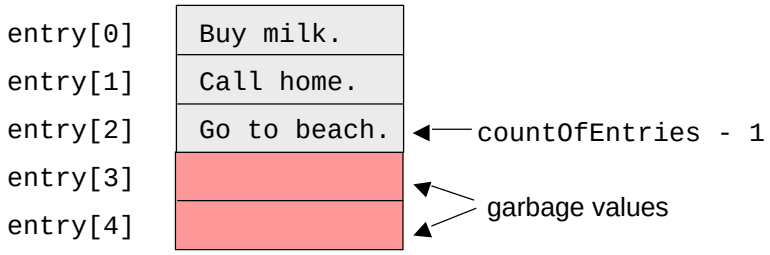
Wrapper Classes for Arrays

- Arrays can be made into objects by creating a wrapper class
  - » similar to wrapper classes for primitive types
- In the wrapper class:
  - » make an array an instance variable
  - » define constructors
  - » define accessor methods to read and write element values and parameters
- The text shows an example of creating a wrapper class for an array of objects of type `OneWayNoRepeatsList`
  - » the wrapper class defines two constructors plus the following methods:
    - – `addItem`, `full`, `empty`, `entryAt`, `atLastEntry`, `onList`, `maximumNumberOfEntries`, `numberOfEntries`, and `eraseList`

Partially Filled Arrays

- Sometimes only part of an array has been filled with data
- Array elements always contain something, whether you have written to them or not
  - » elements which have not been written to contain unknown (*garbage*) data so you should avoid reading them
- There is no automatic mechanism to detect how many elements have been filled - *you*, the programmer need to keep track!
- An example: the instance variable `countOfEntries` (in the class `OneWayNoRepeatsList`) is incremented every time `addItem` is called (see the text)

Example of a Partially Filled Array



`countOfEntries` has a value of 3.
`entry.length` has a value of 5.

Searching an Array

- There are many techniques for searching an array for a particular value
- *Sequential search*:
  - » start at the beginning of the array and proceed in sequence until either the value is found or the end of the array is reached*
    - – if the array is only partially filled, the search stops when the last meaningful value has been checked
  - » it is not the most efficient way
  - » but it works and is easy to program

* Or, just as easy, start at the end and work backwards toward the beginning

Example: Sequential Search of an Array

The `onList` method of `OneWayNoRepeatsList` sequentially searches the array `entry` to see it the parameter `item` is in the array

```
public boolean onList(String item)
{
    boolean found = false;
    int i = 0;
    while ((! found) && (i < countOfEntries))
    {
       if (item.equals(entry[i]))
          found = true;
       else
          i++;
    }

   return found;
}
```


Sorting an Array

- Sorting a list of elements is another very common problem (along with searching a list)
  - » sort numbers in ascending order
  - » sort numbers in descending order
  - » sort strings in alphabetic order
  - » etc.
- There are many ways to sort a list, just as there are many ways to search a list
- *Selection sort*
  - » one of the easiest
  - » not the most efficient, but easy to understand and program

Selection Sort Algorithm for an Array of Integers

To sort an array on integers in ascending order:

- search the array for the smallest number and record its index
- swap (interchange) the smallest number with the first element of the array
  - » the sorted part of the array is now the first element
  - » the unsorted part of the array is the remaining elements
- search the remaining unsorted part of the array for the next smallest element and record that element's index
- swap the next smallest element with the second element of the array
- repeat the search and swap until all elements have been placed
  - » each iteration of the search/swap process increases the length of the sorted part of the array by one, and reduces the unsorted part of the array by one

Selection Sort Code

```
/**************************
*Precondition:
*Every indexed variable of the array a has a value.
*Action: Sorts the array a so that
*a[0] <= a[1] <= ... <= a[a.length - 1].
*************************************************/
public static void sort(int[] a)
{
   int index, indexOfNextSmallest;
   for (index = 0; index < a.length; index++)
   {//Place the correct value in a[index]:
       indexOfNextSmallest = indexOfSmallest(index, a);
       interchange(index,indexOfNextSmallest, a);
       //a[0] <= a[1] <=...<= a[index] and these are
       //the smallest of the original array elements.
       //The remaining positions contain the rest of
       //the original array elements.
   }
}
```

Example: Selection Sort

- The `SelectionSort` program in the text shows a class for sorting an array of `int`s in ascending order
- Notice the precondition: every indexed variable has a value
- Also notice that the array may have duplicate values and the class handles them in a reasonable way - they are put in sequential positions
- Finally, notice that the problem was broken down into smaller tasks, such as "find the index of the smallest value" and "interchange two elements"
  - » these subtasks are written as separate methods and are `private` because they are helper methods (users are not expected to call them directly)

Selection Sort: Diagram of an Example

Key:
- smallest remaining value
- sorted elements

Problem: sort this 10-element array of integers in ascending order:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
7	6	11	17	3	15	5	19	30	14

1st iteration: smallest value is 3, its index is 4, swap a[0] with a[4]

before:	7	6	11	17	3	15	5	19	30	14
after:	3	6	11	17	7	15	5	19	30	14

2nd iteration: smallest value in remaining list is 5, its index is 6, swap a[1] with a[6]

3	6	11	17	7	15	5	19	30	14
3	6	11	17	7	15	5	19	30	14

Etc. - only nine iterations are required since the last one will put the last *two* entries in place by swapping them if necessary.

Multidimensional Arrays

- Arrays with more than one index
  - » number of dimensions = number of indexes
- Arrays with more than two dimensions are a simple extension of two-dimensional (2-D) arrays
- A 2-D array corresponds to a table or grid
  - » one dimension is the row
  - » the other dimension is the column
  - » cell: an intersection of a row and column
  - » an array element corresponds to a cell in the table

Table as a 2-Dimensional Array

- The table assumes a starting balance of $1000
- First dimension: row identifier - Year
- Second dimension: column identifier - percentage
- Cell contains balance for the year (row) and percentage (column)
- Balance for year 4, rate 7.00% = $1311

Balances for Various Interest Rates Compounded Annually (Rounded to Whole Dollar Amounts)

Year	5.00%	5.50%	6.00%	6.50%	7.00%	7.50%
1	$1050	$1055	$1060	$1065	$1070	$1075
2	$1103	$1113	$1124	$1134	$1145	$1156
3	$1158	$1174	$1191	$1208	$1225	$1242
4	$1216	$1239	$1262	$1286	$1311	$1335
5	$1276	$1307	$1338	$1370	$1403	$1436
...	…	…	…	…	…	…

Table as a 2-D Array

Column Index 4
(5th column)

Row Index 3
(4th row)

Indexes	0	1	2	3	4	5
0	$1050	$1055	$1060	$1065	$1070	$1075
1	$1103	$1113	$1124	$1134	$1145	$1156
2	$1158	$1174	$1191	$1208	$1225	$1242
3	$1216	$1239	$1262	$1286	$1311	$1335
4	$1276	$1307	$1338	$1370	$1403	$1436
...	…	…	…	…	…	…

- Generalizing to two indexes: [row][column]
- First dimension: row index
- Second dimension: column index
- Cell contains balance for the year/row and percentage/column
- All indexes use zero-numbering
  - » Balance[*3*][*4*] = cell in 4th row (year = 4) and 5th column (7.50%)
  - » Balance[3][4] = $1311 (shown in yellow)

Java Code to Create a 2-D Array

- Syntax for 2-D arrays is similar to 1-D arrays
- Declare a 2-D array of `ints` named `table`
  - » the table should have ten rows and six columns

```
int[][] table = new int[10][6];
```

Method to Calculate the Cell Values

Each array element corresponds to the balance for a specific number of years and a specific interest rate (assuming a starting balance of $1000):

balance(starting, years, rate) = (starting) x $(1 + \text{rate})^{\text{years}}$

The repeated multiplication by (1 + rate) can be done in a `for` loop that repeats `years` times.

```
public static int balance(double startBalance, int years, double rate)
 {
    double runningBalance = startBalance;
    int count;
    for (count = 1; count <= years; count++)
       runningBalance = runningBalance*(1 + rate/100);
    return (int) (Math.round(runningBalance));
 }
```

`balance` method in class InterestTable

Processing a 2-D Array: `for` Loops Nested 2-Deep

- Arrays and `for` loops are a natural fit
- To process all elements of an *n*-D array nest *n* for loops
  - » each loop has its own counter that corresponds to an index
- For example: calculate and enter balances in the interest table
  - » inner loop repeats 6 times (six rates) for every outer loop iteration
  - » the outer loop repeats 10 times (10 different values of `years`)
  - » so the inner repeats 10 x 6 = 60 times = # cells in `table`

```
int[][] table = new int[10][6];
int row, column;
for (row = 0; row < 10; row++)
    for (column = 0; column < 6; column++)
       table[row][column] = balance(1000.00, row + 1, (5 + 0.5*column));
```

Excerpt from `main` method of `InterestTable`

Multidimensional Array Parameters and Returned Values

- Methods may have multi-D array parameters
- Methods may return a multi-D array as the value returned
- The situation is similar to 1-D arrays, but with more brackets
- Example: a 2-D `int` array as a method argument

```
public static void showTable(int[][] displayArray)
{
  int row, column;
  for (row = 0; row < displayArray.length; row++)
  {
     System.out.print((row + 1) + "      ");
     for (column = 0; column < displayArray[row].length; column++)
        System.out.print("$" + displayArray[row][column] + "  ");
     System.out.println();
  }
}
```

Notice how the number of rows is obtained

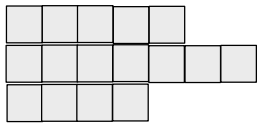
Notice how the number of columns is obtained

`showTable` method from class `InterestTable2`

Ragged Arrays

- Ragged arrays have rows of unequal length
  - » each row has a different number of columns, or entries
- Ragged arrays are allowed in Java
- Example: create a 2-D `int` array named b with 5 elements in the first row, 7 in the second row, and 4 in the third row:

```
int[][] b;
b = new int[3][];
b[0] = new int[5];
b[1] = new int[7];
b[2] = new int[4];
```



Summary
Part 1

- An array may be thought of as a collection of variables, all of the same type.
- An array is also may be thought of as a single object with a large composite value of all the elements of the array.
- Arrays are objects created with *new* in a manner similar to objects discussed previously.

Summary
Part 2

- Array indexes use zero-numbering:
  - » they start at 0, so index *i* refers to the *(i+1)th* element;
  - » the index of the last element is `(length-of-the-array - 1)`.
  - » Any index value outside the valid range of 0 to length-1 will cause an **array index out of bounds error** when the program *runs.*
- A method may return an array.
- A "partially filled array" is one in which values are stored in an initial segment of the array:
  - » use an `int` variable to keep track of how many variables are stored.

Summary
Part 3

- An array indexed variable can be used as an argument to a method anyplace the base type is allowed:
  - » if the base type is a primitive type then the method cannot change the value of the indexed variable;
  - » but if the base type is a class, then the method *can* change the value of the indexed variable.
- When you want to store two or more different values (possibly of different data types) for each index of an array, you can use parallel arrays (multiple arrays of the same length).
- An accessor method that returns an array corresponding to a private instance variable of an array type should be careful to return a copy of the array, and not return the private instance variable itself.
- The selection sort algorithm can be used to sort an array of numbers into increasing or decreasing order.

Summary
Part 4

- Arrays can have more than one index.
- Each index is called a *dimension*.
- Hence, *multidimensional* arrays have multiple indexes,
  - » e.g. an array with two indexes is a two-dimensional array.
- A two-dimensional array can be thought of as a grid or table with rows and columns:
  - » one index is for the row, the other for the column.
- Multidimensional arrays in Java are implemented as arrays of arrays,
  - » e.g. a two-dimensional array is a one-dimensional array of one-dimensional arrays.