Chapter 6
Arrays

Prof. Yongsu Park

Dept. of Computer Science and Engineering
Hanyang University

Contents

- Introduction
- Creating & accessing arrays
- length instance variable
- Initializing arrays
- Arrays & references
- Array parameters
- Arguments for the method main
- Privacy leaks with arrays
- Enumerated Type
- Multidimensional arrays

Introduction to Arrays

- An *array* is a data structure used to process a collection of data that is all of the same type
  - An array behaves like a numbered list of variables with a uniform naming mechanism
  - It has a part that does not change: the name of the array
  - It has a part that can change: an integer in square brackets
  - For example, given five scores:

```
score[0], score[1], score[2], score[3], score[4]
```

Creating and Accessing Arrays

- An array that behaves like this collection of variables, all of type **`double`**, can be created using one statement as follows:

```
double[] score = new double[5];
```

- Or using two statements:

```
double[] score;
score = new double[5];
```

  - The first statement declares the variable **`score`** to be of the array type **`double[]`**
  - The second statement creates an array with five numbered variables of type **`double`** and makes the variable **`score`** a name for the array

Creating and Accessing Arrays

- The individual variables that together make up the array are called *indexed variables*
  - They can also be called *subscripted variables* or *elements* of the array
  - The number in square brackets is called an *index* or *subscript*
  - In Java, *indices must be numbered starting with **0**, and nothing else*

`score[0], score[1], score[2], score[3], score[4]`

Declaring and Creating an Array

- An array is declared and created in almost the same way that objects are declared and created:

```
BaseType[] ArrayName = new BaseType[size];
```

  - The *size* may be given as an expression that evaluates to a nonnegative integer, for example, an `int` variable

```
char[] line = new char[80];
double[] reading = new double[count];
Person[] specimen = new Person[100];
```

Display 6.1 An Array Used in a Program

```
import java.util.Scanner;

public class ArrayOfScores
{
    /**
     Reads in 5 scores and shows how much each
     score differs from the highest score.
    */
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        double[] score = new double[5];
        int index;
        double max;

        System.out.println("Enter 5 scores:");
        score[0] = keyboard.nextDouble();
        max = score[0];
        for (index = 1; index < 5; index++)
        {
            score[index] = keyboard.nextDouble();
            if (score[index] > max)
                max = score[index];
        //max is the largest of the values score[0],..., score[index].
        }

        System.out.println("The highest score is " + max);
        System.out.println("The scores are:");
        for (index = 0; index < 5; index++)
            System.out.println(score[index] + " differs from max by "
                                            + (max - score[index]));
    }
}
```

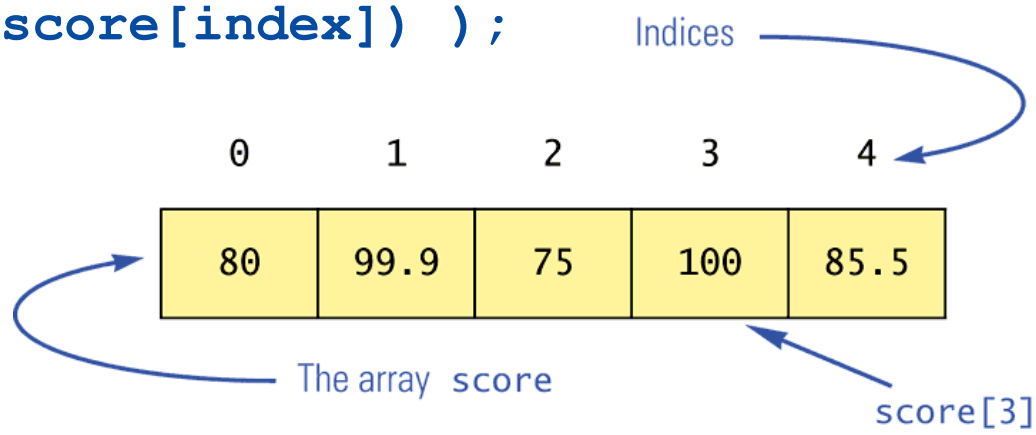
Sample Dialogue

```
Enter 5 scores:
90  99.9  75  100  85.5
The highest score is 100
The scores are:
90.0 differs from max by 20
99.9 differs from max by 0.1
75.0 differs from max by 25
100.0 differs from max by 0.0
85.5 differs from max by 14.5
```

Using the `score` Array in a Program

- The `for` loop is ideally suited for performing array manipulations:

```
for (index = 0; index < 5; index++)
  System.out.println(score[index] +
      " differs from max by " +
      (max - score[index]) );
```



The `length` Instance Variable

- An array is considered to be an object
- Since other objects can have instance variables, so can arrays
- Every array has exactly one instance variable named ***`length`***
  - When an array is created, the instance variable **`length`** is automatically set equal to its size
  - The value of **`length`** cannot be changed (other than by creating an entirely new array with **`new`**)

```
double[] score = new double[5];
```

  - Given **`score`** above, **`score.length`** has a value of 5

Initializing Arrays

- An array can be initialized when it is declared
  - Values for the indexed variables are enclosed in braces, and separated by commas
  - The array size is automatically set to the number of values in the braces

```
int[] age = {2, 12, 1};
```

  - Given **`age`** above, **`age.length`** has a value of 3

Initializing Arrays

- Another way of initializing an array is by using a **`for`** loop

```
double[] reading = new double[100];
int index;
for (index = 0;
     index < reading.length; index++)
  reading[index] = 42.0;
```

- If the elements of an array are not initialized explicitly, they will automatically be initialized to the default value for their base type

Pitfall: An Array of Characters Is Not a String

- An array of characters is conceptually a list of characters, and so is conceptually like a string
- However, an array of characters is not an object of the class **`String`**

```
char[] a = {'A', 'B', 'C'};
String s = a; //Illegal!
```

- An array of characters can be converted to an object of type **`String`**, however

Pitfall: An Array of Characters Is Not a String

- The class **`String`** has a constructor that has a single parameter of type **`char[]`**

```
String s = new String(a);
```

  - The object **`s`** will have the same sequence of characters as the entire array **`a`** (**`"ABC"`**), but is an *independent* copy
- Another **`String`** constructor uses a subrange of a character array instead

```
String s2 = new String(a,0,2);
```

  - Given **`a`** as before, the new string object is **`"AB"`**

Arrays and References

- Like class types, a variable of an array type holds a *reference*
  - Arrays are objects
  - A variable of an array type holds the address of where the array object is stored in memory
  - Array types are (usually) considered to be class types

Arrays are Objects

- An array can be viewed as a collection of indexed variables
- An array can also be viewed as a single item whose value is a collection of values of a base type
  - An array variable names the array as a single item

    ```
    double[] a;
    ```

  - A **new** expression creates an array object and stores the object in memory

    ```
    new double[10]
    ```

  - An assignment statement places a reference to the memory address of an array object in the array variable

    ```
    a = new double[10];
    ```

Pitfall: Arrays with a Class Base Type

- The base type of an array can be a class type

```
Date[] holidayList = new Date[20];
```

- The above example creates 20 indexed variables of type **`Date`**
  - It does not create 20 objects of the class **`Date`**
  - Each of these indexed variables are automatically initialized to **`null`**
  - Any attempt to reference any of them at this point would result in a "null pointer exception" error message

Pitfall: Arrays with a Class Base Type

- Like any other object, each of the indexed variables requires a separate invocation of a constructor using **new** (singly, or perhaps using a **for** loop) to create an object to reference

```
holidayList[0] = new Date();
                 . . .
holidayList[19] = new Date();
```

OR

```
for (int i = 0; i < holidayList.length; i++)
  holidayList[i] = new Date();
```

- Each of the indexed variables can now be referenced since each holds the memory address of a **Date** object

Array Parameters

- Both array indexed variables and entire arrays can be used as arguments to methods
  - An indexed variable can be an argument to a method in exactly the same way that any variable of the array base type can be an argument

Array Parameters

```
double n = 0.0;
double[] a = new double[10];//all elements
                    //are initialized to 0.0
int i = 3;
```

- Given **myMethod** which takes one argument of type **double**, then all of the following are legal:

```
myMethod(n);//n evaluates to 0.0
myMethod(a[3]);//a[3] evaluates to 0.0
myMethod(a[i]);//i evaluates to 3,
                  //a[3] evaluates to 0.0
```

Array Parameters

- An argument to a method may be an entire array
- Array arguments behave like objects of a class
  - Therefore, a method can change the values stored in the indexed variables of an array argument
- A method with an array parameter must specify the base type of the array only

```
BaseType[]
```

  - It does not specify the length of the array

Array Parameters

- The following method, **doubleElements**, specifies an array of **double** as its single argument:

```
public class SampleClass
{
  public static void doubleElements(double[] a)
  {
    int i;
    for (i = 0; i < a.length; i++)
      a[i] = a[i]*2;
    . . .
  }
. . .
}
```

Array Parameters

- Arrays of double may be defined as follows:

```
double[] a = new double[10];
double[] b = new double[30];
```

- Given the arrays above, the method **doubleElements** from class **SampleClass** can be invoked as follows:

```
SampleClass.doubleElements(a);
SampleClass.doubleElements(b);
```

  - Note that no square brackets are used when an entire array is given as an argument
  - Note also that a method that specifies an array for a parameter can take an array of any length as an argument

Pitfall: Use of = and == with Arrays

- Because an array variable contains the memory address of the array it names, the assignment operator (=) only copies this memory address
  - It does not copy the values of each indexed variable
  - Using the assignment operator will make two array variables be different names for the same array

```
b = a;
```

  - The memory address in `a` is now the same as the memory address in `b`: They reference the same array

Pitfall: Use of = and == with Arrays

- For the same reason, the equality operator **(==)** only tests two arrays to see if they are stored in the same location in the computer's memory
  - It does not test two arrays to see if they contain the same values

    ```
    (a == b)
    ```

  - The result of the above **boolean** expression will be **true** if **a** and **b** share the same memory address (and, therefore, reference the same array), and **false** otherwise

Arguments for the Method **`main`**

- The heading for the **`main`** method of a program has a parameter for an array of **`String`**
  - It is usually called ***`args`*** by convention

```
public static void main(String[] args)
```

  - Note that since ***`args`*** is a parameter, it could be replaced by any other non-keyword identifier
- If a Java program is run without giving an argument to **`main`**, then a default empty array of strings is automatically provided

Arguments for the Method `main`

- Here is a program that expects three string arguments:

```
public class SomeProgram
{
  public static void main(String[] args)
  {
    System.out.println(args[0] + " " +
                       args[2] + args[1]);
  }
}
```

- Note that if it needed numbers, it would have to convert them from strings first

Arguments for the Method `main`

- If a program requires that the `main` method be provided an array of strings argument, each element must be provided from the command line when the program is run

```
java SomeProgram Hi ! there
```

  - This will set `args[0]` to "Hi", `args[1]` to "!", and `args[2]` to "there"
  - It will also set `args.length` to 3
- When `SomeProgram` is run as shown, its output will be:

```
Hi there!
```

Methods That Return an Array

- In Java, a method may also return an array
  - The return type is specified in the same way that an array parameter is specified

```
public static int[]
    incrementArray(int[] a, int increment)
{
  int[] temp = new int[a.length];
  int i;
  for (i = 0; i < a.length; i++)
    temp[i] = a[i] + increment;
  return temp;
}
```

Privacy Leaks with Array Instance Variables

- If an accessor method does return the contents of an array, special care must be taken
  - Just as when an accessor returns a reference to any private object

```
public double[] getArray()
{
   return anArray;//BAD!
}
```

  - The example above will result in a *privacy leak*

Privacy Leaks with Array Instance Variables

- The previous accessor method would simply return a reference to the array **anArray** itself
- Instead, an accessor method should return a reference to a *deep copy* of the private array object
  - Below, both **a** and **count** are instance variables of the class containing the **getArray** method

```
public double[] getArray()
{
  double[] temp = new double[count];
  for (int i = 0; i < count; i++)
    temp[i] = a[i];
  return temp
}
```

Privacy Leaks with Array Instance Variables

- If a private instance variable is an array that has a class as its base type, then copies must be made of each class object in the array when the array is copied:

```
public ClassType[] getArray()
{
  ClassType[] temp = new ClassType[count];
  for (int i = 0; i < count; i++)
    temp[i] = new ClassType(someArray[i]);
  return temp;
}
```

Enumerated Types

- Starting with version 5.0, Java permits enumerated types
  - An enumerated type is a type in which all the values are given in a (typically) short list
- The definition of an enumerated type is normally placed outside of all methods in the same place that named constants are defined:

```
enum TypeName {VALUE_1, VALUE_2, …, VALUE_N};
```

  - Note that a value of an enumerated type is a kind of named constant and so, by convention, is spelled with all uppercase letters
  - As with any other type, variables can be declared of an enumerated type

Enumerated Types Usage

- Just like other types, variable of this type can be declared and initialized at the same time:

```
WorkDay meetingDay = WorkDay.THURSDAY;
```

  - Note that the value of an enumerated type must be prefaced with the name of the type
- The value of a variable or constant of an enumerated type can be output using **println**
  - The code:

```
System.out.println(meetingDay);
```

  - Will produce the following output:

```
THURSDAY
```

  - As will the code:

```
System.out.println(WorkDay.THURSDAY);
```

  - Note that the type name **WorkDay** is not output

Enumerated Types Usage

- Two variables or constants of an enumerated type can be compared using the **`equals`** method or the **`==`** operator
- However, the **`==`** operator has a nicer syntax

```
if (meetingDay == availableDay)
  System.out.println("Meeting will be on
  schedule.");
if (meetingDay == WorkDay.THURSDAY)
  System.out.println("Long weekend!");
```

An Enumerated Type

Display 6.13 An Enumerated Type

```
1  public class EnumDemo
2  {
3      enum WorkDay {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};

4      public static void main(String[] args)
5      {
6          WorkDay startDay = WorkDay.MONDAY;
7          WorkDay endDay = WorkDay.FRIDAY;

8          System.out.println("Work starts on " + startDay);
9          System.out.println("Work ends on " + endDay);
10     }
11 }
```

SAMPLE DIALOGUE

```
Work starts on MONDAY
Work ends on FRIDAY
```

Some Methods Included with Every Enumerated Type (Part 1 of 3)

Display 6.14 Some Methods Included with Every Enumerated Type

`public boolean equals(`*Any_Value_Of_An_Enumerated_Type*`)`

Returns `true` if its argument is the same value as the calling value. While it is perfectly legal to use `equals`, it is easier and more common to use ==.

EXAMPLE

For enumerated types, (*Value1*`.equals(`*Value2*`)`) is equivalent to (*Value1* == *Value2*).

`public String toString()`

Returns the calling value as a string. This is often invoked automatically. For example, this method is invoked automatically when you output a value of the enumerated type using `System.out.println` or when you concatenate a value of the enumerated type to a string. See Display 6.15 for an example of this automatic invocation.

EXAMPLE

`WorkDay.MONDAY.toString()` returns "MONDAY".
The enumerated type `WorkDay` is defined in Display 6.13.

(continued)

Some Methods Included with Every Enumerated Type (Part 2 of 3)

Display 6.14 Some Methods Included with Every Enumerated Type

```
public int ordinal()
```

Returns the position of the calling value in the list of enumerated type values. The first position is 0.

EXAMPLE

`WorkDay.MONDAY.ordinal()` returns 0, `WorkDay.TUESDAY.ordinal()` returns 1, and so forth. The enumerated type `WorkDay` is defined in Display 6.13.

```
public int compareTo(Any_Value_Of_The_Enumerated_Type)
```

Returns a negative value if the calling object precedes the argument in the list of values, returns 0 if the calling object equals the argument, and returns a positive value if the argument precedes the calling object.

EXAMPLE

`WorkDay.TUESDAY.compareTo(WorkDay.THURSDAY)` returns a negative value. The type `WorkDay` is defined in Display 6.13.

```
public EnumeratedType[] values()
```

(continued)

Some Methods Included with Every Enumerated Type (Part 3 of 3)

Display 6.14 Some Methods Included with Every Enumerated Type

Returns an array whose elements are the values of the enumerated type in the order in which they are listed in the definition of the enumerated type.

EXAMPLE

See Display 6.15.

```
public static EnumeratedType valueOf(String name)
```

Returns the enumerated type value with the specified name. The string `name` must be an exact match.

EXAMPLE

`WorkDay.valueOf("THURSDAY")` returns `WorkDay.THURSDAY`. The type `WorkDay` is defined in Display 6.13.

The `values` Method

- To get the full potential from an enumerated type, it is often necessary to cycle through all the values of the type
- Every enumerated type is automatically provided with the static method `values()` which provides this ability
  - It returns an array whose elements are the values of the enumerated type given in the order in which the elements are listed in the definition of the enumerated type
  - The base type of the array that is returned is the enumerated type

The Method `values` (Part 1 of 2)

Display 6.15 The Method values

```
1  import java.util.Scanner;
2
3  public class EnumValuesDemo
4  {
5      enum WorkDay {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};
6      public static void main(String[] args)
7      {
8          WorkDay[] day = WorkDay.values();
9          Scanner keyboard = new Scanner(System.in);
10         double hours = 0, sum = 0;
11         for (int i = 0; i < day.length; i++)
12         {
13             System.out.println("Enter hours worked for " + day[i]);
14             hours = keyboard.nextDouble();
15              sum = sum + hours;
16         }
17         System.out.println("Total hours work = " + sum);
18     }
19 }
```

This is equivalent to `day[i].toString()`.

(continued)

The Method **values** (Part 2 of 2)

Display 6.15 **The Method values**

SAMPLE DIALOGUE

```
Enter hours worked for MONDAY
8
Enter hours worked for TUESDAY
8
Enter hours worked for WEDNESDAY
8
Enter hours worked for THURSDAY
8
Enter hours worked for FRIDAY
7,5
Total hours work = 39.5
```

Multidimensional Arrays

- It is sometimes useful to have an array with more than one index
- Multidimensional arrays are declared and created in basically the same way as one-dimensional arrays
  - You simply use as many square brackets as there are indices
  - Each index must be enclosed in its own brackets

```
double[][]table = new double[100][10];
int[][][] figure = new int[10][20][30];
Person[][] entry = new Person[10][100];
```

Multidimensional Arrays

- Multidimensional arrays may have any number of indices, but perhaps the most common number is two
  - Two-dimensional array can be visualized as a two-dimensional display with the first index giving the row, and the second index giving the column

```
char[][] a = new char[5][12];
```

  - Note that, like a one-dimensional array, each element of a multidimensional array is just a variable of the base type (in this case, **`char`**)

Multidimensional Arrays

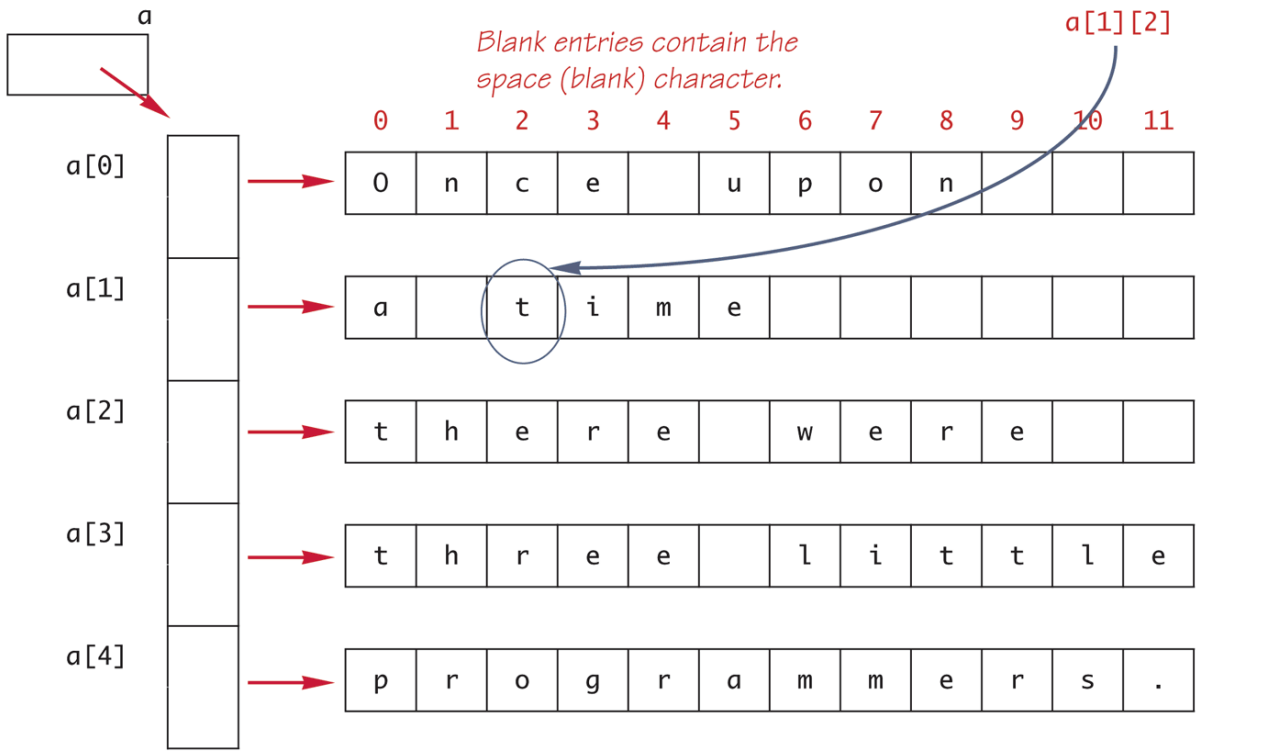
- In Java, a two-dimensional array, such as `a`, is actually an array of arrays
  - The array `a` contains a reference to a one-dimensional array of size 5 with a base type of `char[]`
  - Each indexed variable (`a[0]`, `a[1]`, etc.) contains a reference to a one-dimensional array of size 12, also with a base type of `char[]`
- A three-dimensional array is an array of arrays of arrays, and so forth for higher dimensions

Two-Dimensional Array as an Array of Arrays (Part 1 of 2)

Display 6.17 Two-Dimensional Array as an Array of Arrays

```
char[][] a = new char[5][12];
```

Code that fills the array is not shown.



(continued)

Two-Dimensional Array as an Array of Arrays (Part 2 of 2)

Display 6.17 Two-Dimensional Array as an Array of Arrays

We will see that these can and should be replaced with expressions involving the `length` *instance variable.*

```
int row, column;
for (row = 0; row < 5; row++)
{
    for (column = 0; column < 12; column++)
       System.out.print(a[row][column]);
    System.out.println();
}
```

Produces the following output:

```
Once upon
a time
there were
three little
programmers.
```

Using the `length` Instance Variable

```
char[][] page = new char[30][100];
```

- The instance variable **`length`** does not give the total number of indexed variables in a two-dimensional array
  - Because a two-dimensional array is actually an array of arrays, the instance variable **`length`** gives the number of first indices (or "rows") in the array
    - **`page.length`** is equal to 30
  - For the same reason, the number of second indices (or "columns") for a given "row" is given by referencing **`length`** for that *"row" variable*
    - **`page[0].length`** is equal to 100

Using the `length` Instance Variable

- The following program demonstrates how a nested `for` loop can be used to process a two-dimensional array
  - Note how each `length` instance variable is used

```
int row, column;
for (row = 0; row < page.length; row++)
  for (column = 0; column < page[row].length;
                                        column++)
    page[row][column] = 'Z';
```