

Comp 248

Introduction to Programming

Chapter 6 *Arrays*

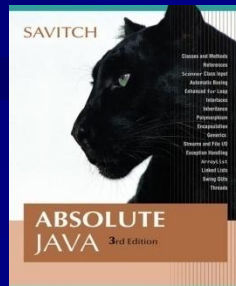
Part C

Dr. Aiman Hanna

**Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada**

These slides has been extracted, modified and updated from original slides of Absolute Java 3rd Edition by Savitch; which has originally been prepared by Rose Williams of Binghamton University. Absolute Java is published by Pearson Education / Addison-Wesley.

Copyright © 2007 Pearson Addison-Wesley
Copyright © 2007-2016 Aiman Hanna
All rights reserved



Initializer Lists

- An *initializer list* can be used to instantiate and initialize an array in one step
- The values are delimited by braces and separated by commas
- Examples:

```
int[] units = {147, 323, 89, 933, 540,  
              269, 97, 114, 298, 476};
```

```
char[] letterGrades = {'A', 'B', 'C', 'D',  
                       'F'};
```

Array of Characters

- An Array of Characters Is Not a String

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

- However, an array of characters can be converted to an object of type **String**

- [CharArrays1.java](#) (MS-Word file)

Arrays of Objects

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

```
Car[] carArr = new Car[20];
```

- VERY IMPOTRANT:

However, this will NOT create 20 Car objects; since each of the 20 elements of the array are initialized to **null**

- Any attempt to reference any them at this point would result in a "null pointer exception" error message

- [ObjectArrays1.java](#) (MS-Word file)

- [ObjectArrays2.java](#) (MS-Word file)

Array As Method 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
- [ArrayOperations9.java](#) ([MS-Word file](#))

Pitfall: Use of = and == with Arrays

- 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
- The result of **if(a == b)** will be **true** if **a** and **b** point to the same memory address (and, therefore, reference the same array), and **false** otherwise

Pitfall: Use of = and == with Arrays

- In the same way that an **equals** method can be defined for a class, an **equalsArray** method (notice that this is just any name) can be defined for a type of array
 - The following method tests two integer arrays to see if they contain the same integer values

Pitfall: Use of = and == with Arrays

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


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;  
}
```

- [ArrayOperations10.java](#) ([MS-Word file](#))

Partially Filled Arrays

- The exact size needed for an array is not always known when a program is written, or it may vary from one run of the program to another
- A common way to handle this is to declare the array to be of the largest size that the program could possibly need
- Care must then be taken to keep track of how much of the array is actually used
 - An indexed variable that has not been given a meaningful value must never be referenced

Partially Filled Arrays

- A variable should be used to keep track of how many elements are currently stored in an array

- [ArrayOperations11.java](#) (MS-Word file)

Privacy Leaks with Array Instance Variables

- If a method return the contents of an array, special care must be taken

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

- The example above will result in a *privacy leak*
- Instead, an accessor method should return a reference to a *deep copy* of the private array object
- [ArrayOperations12.java](#) (MS-Word file)
- [ArrayOperations13.java](#) (MS-Word file)

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;  
}
```

Passing Multidimensional Arrays as Method Parameters

Multidimensional arrays can be passed as parameters to methods in the same fashion as for one-dimensional arrays.

- [ArrayOperations16.java](#) (MS-Word file)

Multidimensional Array as Returned Values

- Methods may have a multidimensional array type as their return type
 - They use the same kind of type specification as for a multidimensional array parameter

```
public double[][] aMethod()  
{  
    . . .  
}
```

- The method **aMethod** returns an array of **double**

Ragged Arrays

- Each row in a two-dimensional array need not have the same number of elements
 - Different rows can have different numbers of columns
- An array that has a different number of elements per row it is called a *ragged array*

Ragged Arrays

```
double[][] a = new double[3][5];
```

- The above line is equivalent to the following:

```
double [][] a;
```

```
a = new double[3][]; //Note below
```

```
a[0] = new double[5];
```

```
a[1] = new double[5];
```

```
a[2] = new double[5];
```

- Note that the second line makes **a** the name of an array with room for 3 entries, each of which can be an array of **doubles** *that can be of any length*
- The next 3 lines each create an array of doubles of size 5

Ragged Arrays

```
double [][] a;  
a = new double[3][];
```

- Since the above line does not specify the size of **a[0]**, **a[1]**, or **a[2]**, each could be made a different size instead:

```
a[0] = new double[5];  
a[1] = new double[10];  
a[2] = new double[4];
```

- [RaggedArrays1.java](#) (MS-Word file)
- [RaggedArrays2.java](#) (MS-Word file)

Enumerated Types

- An enumerated type is a type in which all the values are given in a (typically) short list

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

Example:

```
enum WorkDays {MONDAY, TUESDAY, WEDNESDAY, THURSDAY,  
FRODAY};
```

- The definition of an enumerated type is normally placed outside of all methods
- Once an enumerated type is defined, variables can be declared from this enumerated type
 - Note that a value of an enumerated type is a kind of named constant and so, by convention, is spelled with all uppercase letters

Enumerated Types Example

- A variable of this type can be declared as follows:

```
WorkDay meetingDay, availableDay;
```

- The value of a variable of this type can be set to one of the values listed in the definition of the type, or else to the special value **null**:

```
meetingDay = WorkDay.THURSDAY;  
availableDay = null;
```

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 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
5     public static void main(String[] args)
6     {
7         WorkDay startDay = WorkDay.MONDAY;
8         WorkDay endDay = WorkDay.FRIDAY;
9
10        System.out.println("Work starts on " + startDay);
11        System.out.println("Work ends on " + endDay);
12    }
13 }
```

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
6      enum WorkDay {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};
7
8      public static void main(String[] args)
9      {
10         WorkDay[] day = WorkDay.values();
11
12         Scanner keyboard = new Scanner(System.in);
13         double hours = 0, sum = 0;
14
15         for (int i = 0; i < day.length; i++)
16         {
17             System.out.println("Enter hours worked for " + day[i]);
18             hours = keyboard.nextDouble();
19             sum = sum + hours;
20         }
21
22         System.out.println("Total hours work = " + sum);
23     }
24 }
```

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