Lecture Feb 14 - Array Examples and Reference Types

Bentley James Oakes

February 13, 2018

Assignment 2

- Reminder that it's due **Thursday, February 15th** (tomorrow)
- Office hours today Room 233, 13:30 to 15:00

Assignment 2

- I've seen a common error on assignments
- **TAs will take marks off for this error**

```
int x = 3;
boolean a = x == 3;
System.out.println("Can use == for ints/doubles: " + a);
//Can use == for ints/doubles: true

String phrase = "Hello World";

String world = "World";
String test = "Hello " + world;

boolean b = phrase == test;
System.out.println("Can use == for Strings: " + b);
//Can use == for Strings: false

boolean c = phrase.equals(test);
System.out.println("Must use .equals for Strings: " + c);
//Must use .equals for Strings: true
```

This Lecture

Section 1

Nested For Loops

Multiplication Table

Let's create another example for nested *for-loops*, a multiplication table

```
          B:0     B:1     B:2     B:3     B:4     B:5
A: 0|       0       0       0       0       0       0
A: 1|       0       1       2       3       4       5
A: 2|       0       2       4       6       8      10
A: 3|       0       3       6       9      12      15
A: 4|       0       4       8      12      16      20
A: 5|       0       5      10      15      20      25
```

Multiplication Table

```
//method to print out a times table
//note that the min and max values of a and b are set differently
public static void printMultiTable(int minA, int maxA, int minB, int maxB){

    //iterate over the rows
    for (int a = minA; a < maxA; a++){


        //for each value of b, print the multiplication
        for (int b = minB; b < maxB; b++){

            System.out.print((a*b) + "    ");

        }

        System.out.println();
    }

}
```

Multiplication Table

```java
//method to print out a times table
//note that the min and max values of a and b are set differently
public static void printMultiTable(int minA, int maxA, int minB, int maxB){

    //this code creates the header to display the values of b
    System.out.print("      ");
    for (int b = minB; b < maxB; b++){
        System.out.print(String.format("%8s", "B:" + b));
    }
    System.out.println();

    //iterate over the rows
    for (int a = minA; a < maxA; a++){

        //print the header for each row
        System.out.print("A: " + a + "| ");

        //for each value of b, print the multiplication
        for (int b = minB; b < maxB; b++){

            //formats the results nicely (not testable)
            System.out.print(String.format("%8d", a*b));

        }

        System.out.println();
    }

}
```

- This version of the code looks much better
- **The String.format method isn't testable**

Section 2

Arrays

Creating an Array

- Let's create an array and fill it with values

```
String[] threeMonths = new String[3];
threeMonths[0] = "January";
threeMonths[1] = "February";
threeMonths[2] = "March";
```

- Arrays are created with the `new` keyword
- `threeMonths` will be created with a length of three
  - The length of an array can't be changed after it's created
- The entries are then filled after array creation

Iterating an Array

- Let's use a *for-loop* to iterate through an array

```
String[] catNames = {"Jack Bauer", "Lord Fuzzykins", "Mrs. Whiskers"};
System.out.println("Length: " + catNames.length); //prints 3

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

Notice the difference:

- For String s the length is `s.length()`
- For arrays, the length is `catNames.length`

Creating an Array

- Let's declare and initialize an array in one step

```
String[] daysOfWeek = {"Monday", "Tuesday",
    "Wednesday", "Thursday", "Friday",
    "Saturday", "Sunday"};
```

- `daysOfWeek` will have a length of seven
- Index 0 will contain "Monday"
- Index 6 will contain "Sunday"

Creating an Array Examples

```
13          boolean[] values = {true, true, false};
14
15          values = {false, false};
16      }
17  }
```

Interactions | Console | Compiler Output

```
File: /home/dcx/Dropbox/COMP 202/Lecture 10 - Case Studies/ArrayExamples.java
[line: 15]
Error: illegal start of expression
```

- To repeat, we can't use the braces to assign values to an array after it is created

Section 3

Printing/Comparing/Reversing Arrays

Printing Out an Array

```
int[] a = {1, 2, 3};

System.out.println("Array a: " + a);
```

What prints?

```
Array a: [I@7347c5f3
```

You can't print arrays directly

We'll see why when we talk about *reference types*

Array Printing

- It's very helpful to have a method just for printing out an array

```
public static void printArray(double[] arr){
    for (int i=0; i < arr.length; i++){
        System.out.print(arr[i] + ", ");
    }
    System.out.println();
}
```

- Note that this method can only accept `double` arrays as a parameter
- You can't pass `integer` arrays

Comparing Two Arrays

Write a method that takes as input two `integer` arrays and tests whether they contain the same elements

```
int[] a = {1, 2, 3};
int[] b = {1, 2, 3};

boolean areSame = (a==b);
System.out.println("Are same: " + areSame);
//Are same: false
```

Comparing Arrays

- Let's try to use `.equals()`

```
int[] a = {1, 2, 3};
int[] b = {1, 2, 3};

boolean areSame = (a==b);
System.out.println("Are same: " + areSame);
//Are same: false

boolean areSameEquals = a.equals(b);
System.out.println("Are same using equals: " + areSameEquals);
//Are same using equals: false
```

This prints *false* again

This worked with `Strings`!

Let's write a method to loop through both arrays and compare them

Comparing Two Arrays

```
public static boolean areEqual(int[] a, int[] b){
```

- Must return a boolean whether the arrays contain the same elements or not

Planning the Method

- As methods get more complex, it's necessary to take time to plan them before you write them
- For comparison, we need to know in what cases the array are equal, and when they're not
- For example, the arrays are not equal if they have different lengths
- They are equal if every element in one is found in the same position in the other

Comparing Two Arrays

```
public static boolean areEqual(int[] a, int[] b){

    //if the arrays have different lengths, they are not equal
    if (a.length != b.length){
        return false;
    }

    //loop the variable from 0 to the length of the arrays
    for (int i=0; i < a.length; i++){

        //if the elements are not equal, the arrays are not equal
        if (a[i] != b[i]){
            return false;
        }

        //use this condition for String arrays
        //if (!a[i].equals(b[i])){
        //    return false;
        //}
    }
    //everything's the same, return true
    return true;
}
```

Reversing Arrays

- Let's take an array of `integers`
- and *reverse* it

Steps:

1. Create a new array of the same size
2. For the length of the old array:
   - Find the spot to insert the element in the new array
   - Place it in the new array
3. Return the new array

Reversing an Array

```java
public static int[] reverseArray(int[] arr){

    //create a new array of the same size
    int[] result = new int[arr.length];

    for (int index = 0; index < arr.length; index++){

        //get the element
        int x = arr[index];

        //get the index where to place the element in the new array
        //for example, when index is 0, otherIndex is result.length - 1
        //when index is result.length - 1, otherIndex is 0
        int otherIndex = result.length - index - 1;

        //place the element in the new array
        result[otherIndex] = x;
    }
    return result;
}
```

Section 4

Arrays Methods and Imports

Built-in Methods

- There are built-in versions for printing and comparing arrays
- But you'll have to ask Java to use them

Array.equals()

```java
import java.util.Arrays; //need to have this "import"

public class EqualsMethodExample{

    public static void main(String[] args){

        int[] a = {1, 2, 3};
        int[] b = {1, 2, 3};

        boolean areEqual = Arrays.equals(a, b);
        System.out.println("Are they equal now?! " + areEqual);
        //Are they equal now?! true
    }
}
```

- This will compare the contents of two arrays and return a boolean value
- At the top of our .java file, we have to add `import java.util.Arrays;`

Import Statements

```java
import java.util.Arrays; //need to have this "import"

public class EqualsMethodExample{

    public static void main(String[] args){

        int[] a = {1, 2, 3};
        int[] b = {1, 2, 3};

        boolean areEqual = Arrays.equals(a, b);
        System.out.println("Are they equal now?! " + areEqual);
        //Are they equal now?! true
    }
}
```

- The top line is an example of an **import statement**
- Required to use methods in the Arrays class
- All your import statements go at the top of your .java file, before your `public class`

Other Arrays Methods

Here are some useful methods in the Arrays class:

- `Arrays.equals(a,b)` → returns a boolean indicating whether or not the contents of the two arrays are the same.
- `Arrays.toString(arr)` → returns the contents of an array as a `String` value. This allows us to print the contents of an array without using a *for-loop*.
- `Arrays.sort(arr)` → sorts the input array in increasing order

- Note: It would be a **perfect test question** to ask you to write the `.equals()` and `toString()` methods by hand
- Writing the `sort()` method is more difficult, and won't be testable material, but it is also great practice

Section 5

Primitive versus Reference Types Intro

Type Division

- Arrays are different than the usual variable types `int`, `double`, `boolean`, `char`
- Arrays are more complicated, like `Strings`

We separate variable types in Java into two groups:

- Primitive types
- Reference types

Primitive vs Reference Types

- Primitive types:
  - `int, double, boolean, char`
- Reference types:
  - `String`
  - Arrays
  - Objects

What's different about reference types?

- Can't use == for comparisons
- Variables store **addresses** instead of values
- We can call methods and access members of variables
  - `.equals()` for Strings, `.length` for arrays

Reference Type Variable

```
int[] a = {1, 2, 3};

System.out.println("Array a: " + a);
```

What prints?

```
Array a: [I@7347c5f3
```

This is the **address** of the data within a

Addresses in Variables

```
int[] a = {1, 2, 3};
```

- The value stored in a is the address in the computer's memory where we can find those numbers

Analogy:

- A reference variable stores the website address where the data can be found

Addresses Figure

```
int[] a = {1, 2, 3};
```

Address	Variable Type	ID	Value
1171...	int[]	a	@7347...
...			
7347...	int	a[0]	1
7347...	int	a[1]	2
7347...	int	a[2]	3

- The array variable stores the address where the data starts
- When we index, we are looking up the address in the computer's memory

Consequences

- **Reference type variables store addresses**
- This means that printing out their value might not work
- Also means that we have to compare them a different way
  - This is why we can't use == to compare Strings
- Another consequence: we can have two variables storing the same address

Comparing Strings

- We talked about comparing `Strings` with `.equals()`, instead of the == we use for primitive types
- This is because a `String` variable is actually storing an address
- And == is comparing the addresses

- You can print out a `String` directly because Java does some work behind the scenes

Printing an Array

Let's make sure we know how to properly print an array

```
public static void print(int[] arr){
    for(int i=0; i < arr.length; i++){
        System.out.print(arr[i] + ", ");
    }
    System.out.println();
}
```

Aliasing

```
//create a
int[] a = {1, 2, 3};
print(a);

//assign the address in a
//into b
int[] b = a;
print(b);

//change the first element in b
b[0] = 5;

//print out a again
print(a);
```

What prints?

1, 2, 3

1, 2, 3

5, 2, 3

- Here we have two array variables pointing to the **same address**
- A change in one affects the other
- This is called **aliasing**
- Analogy: They both contain the same website address, so changes are seen for both

Conclusion

- **Primitive types store values**
  - `int`, `double`, `boolean`, `char`
- **Reference types store addresses**
  - `Strings`, arrays, Objects
- We'll examine consequences for comparisons, aliasing, and swapping

Section 6

Null

null

- Reference type variables can also store the **null** value.
- `null` means *no address*
  - Analogy: The website address is a big red X
- Null is useful to check if something has not been initialized yet
- We'll see examples of using `null` later

NullPointerException

```
30          int[] c = null;
31          System.out.println("C Length: " + c.length);
32
```

Interactions | Console | Compiler Output

```
Welcome to DrJava.  Working directory is /home/dcx/Dropbo
> run RandEntry
java.lang.NullPointerException
     at RandEntry.main(RandEntry.java:31)
```

- This is a common run-time error
- Occurs when a reference variable has the value *null* and you try to access it
  - Example: Trying to access `c.length` if `c` is `null`
  - Or trying to print out the first element in `c`

Section 7

Swapping Values

Swapping Values

- Now we'll look at how to swap values around
- How do we take the value in variable `a` and put it in variable `b`?
- What if `a` and `b` are primitive types? What if they are reference types?

- This will help you understand reference types
- But it's a bit tricky

Swapping Basics

```
int x = 5;
int y = 7;
```

How can we switch around these values?

```
int temp = x;
    x = y;
  y = temp;
```

Use a temporary variable to store the value of `x`
Then shift around the values

Recall

- If you remember when we were talking about passing parameters to methods, parameters are **copied** to the method
- Here, the x variable is not changed in the main method

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

    int x = 0;

    System.out.println("X in main first: " + x);
    modify(x);
    System.out.println("X in main second: " + x);
}

public static void modify(int x){

    System.out.println("X in method first: " + x);
    x = x + 1;
    System.out.println("X in method second: " + x);
}
```

```
X in main first: 0
X in method first: 0
X in method second: 1
X in main second: 0
```

Swapping Primitive Types

```
public static void main(String[] args){
    int x = 5;
    int y = 6;
    swap(x, y);
    System.out.println(x + " and " + y);
}
public static void swap(int x, int y){
    int temp = x;
    x = y;
    y = temp;
}
```

What prints? 5 and 6 Why?

- Value of x is copied to the first parameter of the method
- Value of y is copied to the second parameter of the method
- Values in the method's variables are changed, not in main's variables

Swap with Return

```
public static void main(String[] args){
    int x = 5;
    int y = 6;
    swap(x, y);
    System.out.println(x + " and " + y);
}
public static int swap(int x, int y){
    int temp = x;
    x = y;
    y = temp;
    return x;
}
```

What prints? 5 and 6 Why?

- We are returning a value from swap, but it is not assigned to x or y
- Because we **copy** primitive variables when passing them, main's variables are different than swap's variables

Incorrect Swapping Values in Arrays

```
public static void main(String[] args){
    int[] a = {1,2,3,4,5};
    System.out.println(a[1]+ " and " + a[3]);

    swap(a[1], a[3]);
    System.out.println(a[1]+ " and " + a[3]);
}
public static void swap(int x, int y){
    int temp = x;
    x = y;
    y = temp;
}
```

What prints? 2 and 4 2 and 4 Why?

- We are passing two int values, which are copied to the parameters of swap
- Only the variables in `swap` are being swapped

Modifying Array Values

- Let's modify an element in an array
- We pass the value of a, which is an **address**
- Then we can change around the data in the array
- The `modifyArray` method is modifying data (in the computer's memory)

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

    int[] a = {1, 2, 3, 4};

    System.out.println("a[0] first: " + a[0]); //a[0] first: 1
    modifyArray(a);
    System.out.println("a[0] second: " + a[0]); //a[0] second: 2
}

public static void modifyArray(int[] a){
    a[0] = a[0] + 1;
}
```

Properly Swapping Array Values

```
public static void main(String[] args){
    int[] a = {1,2,3,4,5};
    System.out.println(a[1]+ " and " + a[3]);
    swapArray(a, 1, 3);
    System.out.println(a[1]+ " and " + a[3]);
}
public static void swapArray(int[] b, int i, int j){
    int temp = b[i];
    b[i] = b[j];
    b[j] = temp;
}
```

What prints? 2 and 4 4 and 2 Why?

- We pass the value of a, which is an **address**
- Then we can change around the data in the array
- The `swapArrays` method is moving around data (in the computer's memory)

Conclusion

Bottom line:

- Values are **copied** when they are passed as parameters
- Arrays store addresses
- You must pass an array to a method if you want to change values within the array inside the method

Creating a New Array

```
public static void changeContent(int[] arr) {
    // If we change the content of arr.
    arr[0] = 10; // The address of arr is passed, and we change the data inside
    //This changes main's array
}
public static void changeRef(int[] arr) {
    // If we change the reference
    arr = new int[2]; // this changes the address stored in the method's variable
    arr[0] = 15; //this refers to different data
}
public static void main(String[] args) {
    int [] arr = {4,5};
    System.out.println(arr[0]); //Will print 4.
    changeContent(arr); //pass address, data inside changes
    System.out.println(arr[0]); // Will print 10.
    changeRef(arr); //pass address, but data inside does not change
    System.out.println(arr[0]); // Will still print 10.
}
```