Array Advanced Topics

CSCI 161 – Introduction to Programming I

Professor Thomas Rogers

Overview

- Chapter 7 in the textbook *"Building Java Programs"*, by Reges & Stepp.
- Reference Semantics
- References vs Value Semantics
- Why Have Both Semantics?
- Advanced Array Techniques
- Multidimensional Arrays

Reference Semantics

- Arrays are powerful because they can be passed to methods and their values changed within the method and those changes visible to the caller. This is due to **Reference Semantics.**

- To understand Reference Semantics, we should first look at its counterpart, **Value Semantics**:

  - **Value Semantics** (Value Types) - A ***system*** in which values are stored directly and copying is achieved by creating independent copies of values. Types that use value semantics are called **value types**.

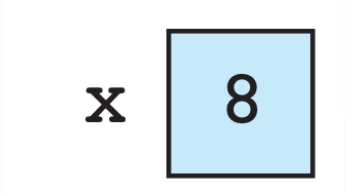
  - Primitive data types, like ***int***, ***double*** are stored as value types and passed to methods via value semantics.

  - Objects, such as those variables declared from classes (String, Scanner, File, ...) and arrays are stored as reference types.

Reference vs Value Semantics

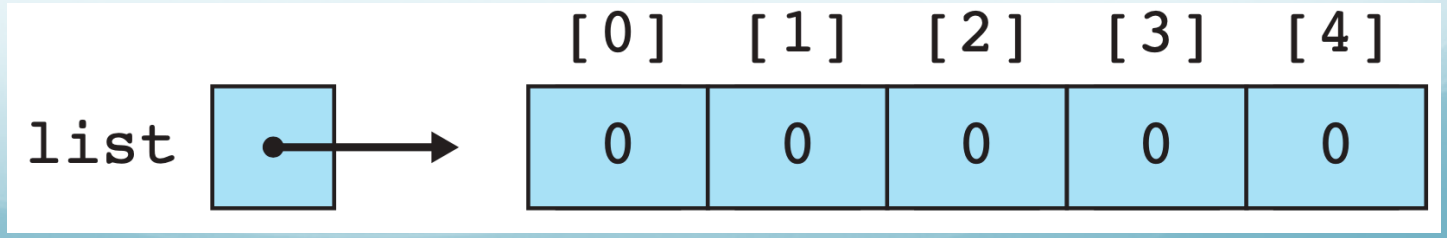
- A primitive data type variable such as int ***x*** looks like the following in memory:

```
int x = 8;
```



- However, an array of integers named ***list*** is represented in memory as follows:

```
int[] list = new int[5];
```



*** Note:**

list *points* to the memory holding the array

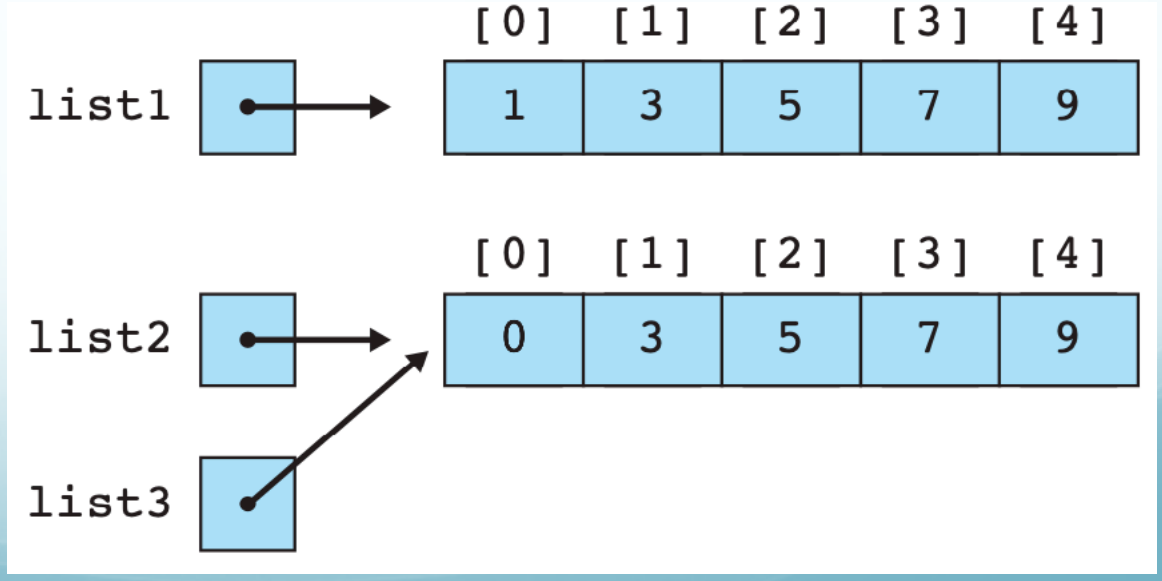
Why Have Both Semantics?

- **Why Reference Types?** - Objects can be complex and they can hold a lot of data. For this reason, the ability to pass them around and ***sharing*** them in your program as a reference increases ***efficiency*** (uses less memory, and less resources such as heap and the stack, etc.).
- **Why Value (aka Primitive) Types?** - Again, for efficiency. For certain types of data (integers, boolean, etc.) that are small and without complexity, the overhead of references just adds bloat and decreases efficiency.

Example

- Consider the following sample code:

```
int[] list1 = new int[5];
int[] list2 = new int[5];
for (int i = 0; i < list1.length; i++) {
    list1[i] = 2 * i + 1;
    list2[i] = 2 * i + 1;
}
int[] list3 = list2;
list3[0] = 0;
```



*** Note:**

list3 and ***list2*** point to the same memory locations

Be Aware, Be Careful…

- Things to be aware of:
  - **Passing arrays to a method** - When you pass an array to a method it is passed by reference and you are able to change the contents of the array.
  - **.equals() method** - Just like the same named method of the String object, this method can tell whether two array objects are the same (two different objects, but each object is an array with the same number of elements and each respective element identical.)
  - **null** - Java keyword signifying no object. Arrays of objects (like String[] are auto-initialized to null when the array is declared with no initial values.

Advanced Array Techniques

- **Shifting Values in an Array** - A complex subject, but boils down to ***rotateLeft()*** and ***rotateRight*** methods:
  - **rotateLeft** - method for integer arrays:

```
public static void rotateLeft(int[] list) {
  int first = list[0];
  for (int i = 0; i < list.length -1; i++) {
    list[i] = list[ i + 1];
  }
  list[list.length - 1] = first;
}
```

Advanced Array Techniques

- **rotateRight** - method for integer arrays:

```
public static void rotateRight(int[] list) {
  int last = list[list.length - 1];
  for (int i = list.length-1; i >= 1; i--) {
    list[i] = list[ i - 1];
  }
  list[0] = last;
}
```

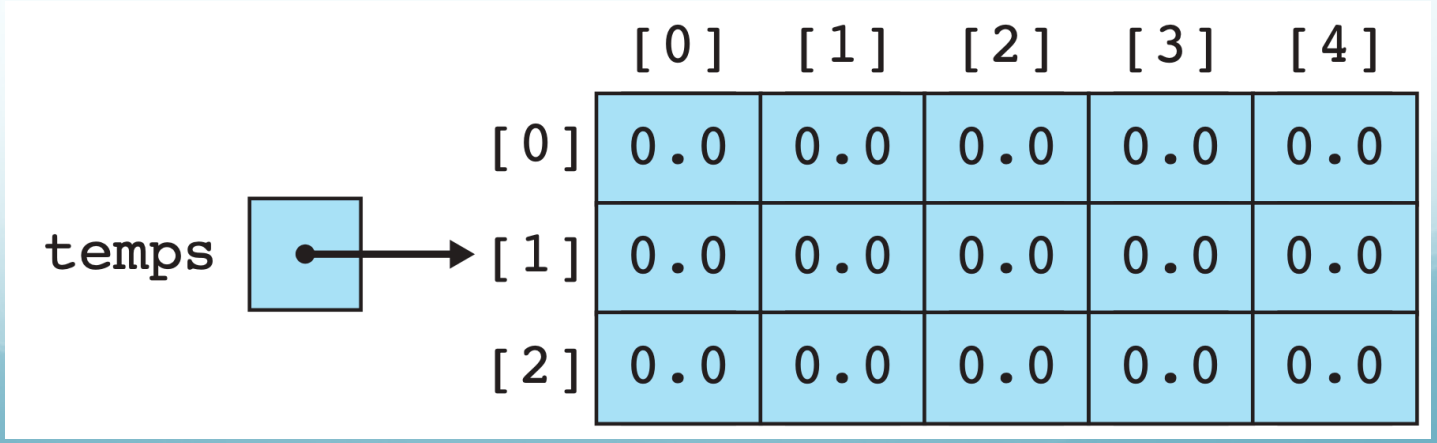
Advanced Array Techniques

- **Arrays of Objects:** Your program can declare and use arrays of primitive data types, or of objects (String, your own object, etc.).
- **Determine Size, Declare, Fill:** It is important to remember that declaring and filling arrays is often a multi-step process:
  1. ***Determine the needed size*** of the array (possibly by reading the number entries or lines in a file).
  2. ***Declare the array given its size***.
  3. ***Fill the array*** by looping through the data and creating and/or assigning **new** objects/values to the array elements.

Multidimensional Arrays

- **Multidimensional Array** - An array of arrays with the elements of which accessed through multiple integer indexes (one per dimension).
  - All dimensions must be arrays of the same data type and all dimensions have arrays that are zero-based.
  - **Two-Dimensional Array:** Think of a spreadsheet with rows and columns (first index rows, second index columns):

```
// 3 rows by 5 columns
double[][] temps = new double[3][5];
```



Multidimensional Arrays

- **Three-Dimensional Array:** It is a bit harder to visualize a three-dimension array, but a bit easier if you think of multiple sheets, with each sheet on a different ***"plane"***, with the general recommendation to think of the first dimension as the plane #, followed by row, followed by column:

```
int[][][] numbers =
  new int[numPlanes][numRows][numColumns];
```

Jagged Arrays

- Two-dimensional Arrays need not be rectangular (same width for each row). They can be jagged, instead:

```
int[][] jagged = new int[3][];
jagged[0] = new int[2];
jagged[1] = new int[4];
jagged[3] = new int[3];
```