Lecture 29:
1-D Arrays, Part III

Sierra College
CSCI-12
Spring 2015
Weds 05/13/15

Announcements

- **General**
- **Schedule**
  - Last lecture today, then finals week (final exam Weds 5/20)
    - Review session next Monday, study guide will be posted this week
    - In-class exam Weds (like last time)
  - Added lab hours, tentatively:
    - **Friday 5/15 9am-noon** (for finish-up work on Dam class, and beginning LAST program)
    - **Friday 5/22 10am-2pm** (for work on LAST program)
- **Current assignments**
  - PRGM25: Dam (due Thurs 5/14 @ 11pm)
    - Create a new class which models a water storage dam
    - Use the systematic procedure we have gone thru in lectures
    - Refer back to prior lecture notes
- **Last assignment**
  - **PRGM29: CA Water Project (due Sunday 5/24 @ 11pm, after midterm)**
    - Create a new class which is an ARRAY of Dams
    - Discussed in class today, and **starter code framework is provided** (scaled-back version)

Lecture Topics

- **Last time**:
  - More operations on 1-D arrays using looping
- **Today**
  - 1-D Arrays
    - Parsing strings (from last time)
    - Creating objects from strings and files (from last time)
    - 1-D arrays as objects, with methods upon them
    - Discussion of last program, API, starter code, etc.

For Next Time

- **Lecture Prep**
  - Final exam study guide
- **Program**
  - Get underway on your last program
    - ***For starters, get the provided starter code working with YOUR Dam class***

Arrays In Classes

- Arrays are just another type of data/variable that may be present in a Java class
- Arrays can be **data**
  - Local variables inside a method
  - Instance variables of a class
- Arrays can be **method input/output arguments**
  - A calling parameter passed into a class method, using a mutator or constructor
  - A return value returned from a class method, using an accessor

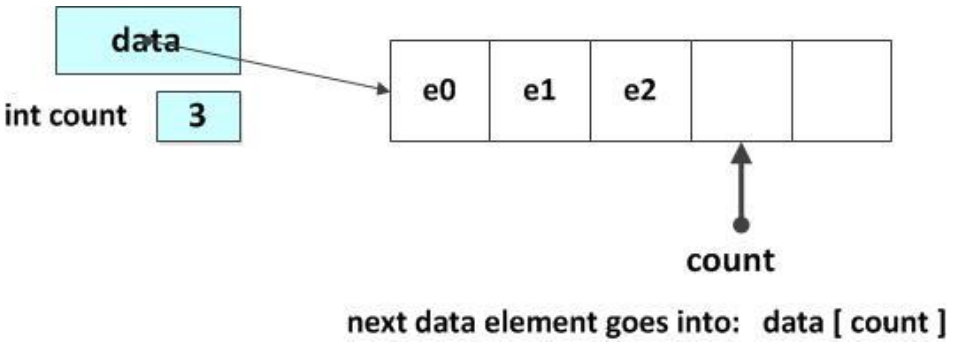
Arrays As Local Variables

- An array may be declared and used within a method, just like any other variable
- It is declared and instantiated using any of the ways already discussed
- Like any other variable, it has **local scope**, and "disappears" as soon as its containing method is exited

```
public class MyClass {

   public void doSomething() {
      int [ ] highTemps = new int[365];

      highTemps[0] = 45;
      highTemps[1] = 49;
      // etc…

   }
} // end class
```

Array-Based Classes

- We want to begin building a class which is centered around an array of ints
  - **Can easily be generalized to a class based upon an array of objects**
    - **An array of *Dams***



- The array itself is the **instance variable**
  - We also need to keep track of the **count** of how many elements are actually used
  - This becomes the other **instance variable**
- The **methods** are what we can do to the array
  - How big is it? How many are being used?
  - Printing the entire array
  - Adding elements to the end of the (used portion of) the array
  - Getting or setting individual elements
    - But first, is a specific index valid or not?
  - Actions upon specific elements of the array

Arrays As Instance Variables

- Arrays are declared and instantiated as usual
- Their sizes may be known or unknown at declaration
  - **If size is known**: could instantiate size where instance variable is declared
  - **If size is unknown**: instantiate size in the constructor(s)
- Like any instance variables, arrays have **class scope** (are globally visible throughout the class)
- The class will provide the methods needed to manipulate the array
- Here, we also want to keep track of the **count** of actually-used elements

```
public class Array1DIntsClass {

    //-------------------------------------------
    // instance variables
    //-------------------------------------------

    // the primary data for this class is an ARRAY
    private int [] data;   // declaration only

    // this tells us how many are used, AND index of *next* one
    private int count;
```

See ***Array1DIntsClass.java*** in **Example Source Code**

Arrays With Constructors

- Any **constructor** that accepts an array parameter should:
  - Instantiate its own version of the internal instance variable array (if the size is variable or unknown)
- In this example, we have 3 overloaded constructors:
  - Default size
  - Specified size
  - Sized according to input array
- We are also keeping track of the **count** of ints actually used

```
//---------------------------------------------
// constructors
//---------------------------------------------

// default constructor, fixed size, default values
public Array1DIntsClass() {

    // declare and initialize to default size, gets default values
    data = new int [DEF_SIZE];
    count = 0;
}

// constructor, user-specified size, default values
public Array1DIntsClass(int size) {

    // declare and initialize to specified size, gets default values
    data = new int [size];
    count = 0;
}

// constructor which initializes using provided array
public Array1DIntsClass(int [] data) {

    // initialize to size of provided data
    this.data = new int [data.length];
    count = data.length;

    // transfer over all input data, int by int
    for (int i=0; i < data.length; i++) {
        this.data[i] = data[i];
    }
}
```

See ***Array1DIntsClass.java*** in **Example Source Code**

Size Utilities For Arrays

- If we are keeping track of the used portion of the array ourselves, we may want to know:
  - How much space do we have?
  - How much space is used?
  - Is there space to add more elements?
- Since these will be common questions, we may want to create **size utilities** to use in later methods

```
//-----------------------------------------------
// derived data accessors and utilities
//-----------------------------------------------

// what is the maximum size?
public int getMaxSize() {
    return data.length;
}

// what is the current size?
public int getUsedSize() {
    return count;
}

// is there room to add more?
public boolean isRoom() {
    if (getUsedSize() < getMaxSize()) {
        return true;
    }
    else {
        return false;
    }
}
```

See ***Array1DIntsClass.java*** in **Example Source Code**

Index Utilities For Arrays

- If we are planning to manipulate individual array elements, we will want to know things like:
  - Is a given index valid (in the "used" region) or not?
  - What are my choices for a valid index?
- Again, we can create **index utilities** for such common operations, to use in other methods

See ***Array1DIntsClass.java*** in **Example Source Code**

```
// does a given index contain valid data?
public boolean isValidIndex(int index) {
    // we can do this as 1-based or 0-based
    boolean status;

    if (index < 0) {
        // input can't be < 0
        status = false;
    }
    else if (count == 0) {
        // there are no valid elements
        status = false;
    }
    else if (index < count) {
        // any index less than size is OK
        status = true;
    }
    else {
        // index is past last element
        status = false;
    }
    return status;
}

// common across multiple operations:
// obtain a desired (valid) index from user,
// note that if empty array, there IS no valid index
public int promptValidIndex() {
    int index;

    if (getUsedSize() == 0) {
        index = -1;  // gotta return SOMETHING
    }

    else {
        // prompt user for element index
        System.out.println("Which element?");

        // print some sort of menu
        for (int i=0; i<getUsedSize() ; i++) {
            System.out.println("[" + i + "]: " + getElement(i));
        }

        index = UtilsRL.readInt("Enter array index > ", false);
        while (!isValidIndex(index)) {
            index = UtilsRL.readInt("Enter array index > ", false);
        }
        return index;
    }
    return index;
}
```

Display Methods For Arrays

- As we know, it's a standard programming convention to always override ***toString()*** with our own version
  - Here, we create a comma separated list of the array
- For the ***print()***, we might want to also incorporate the index values, as well as display some overall status
- All of the above is "designer's choice"
  - Another overloaded method is also provided
  - See the example file

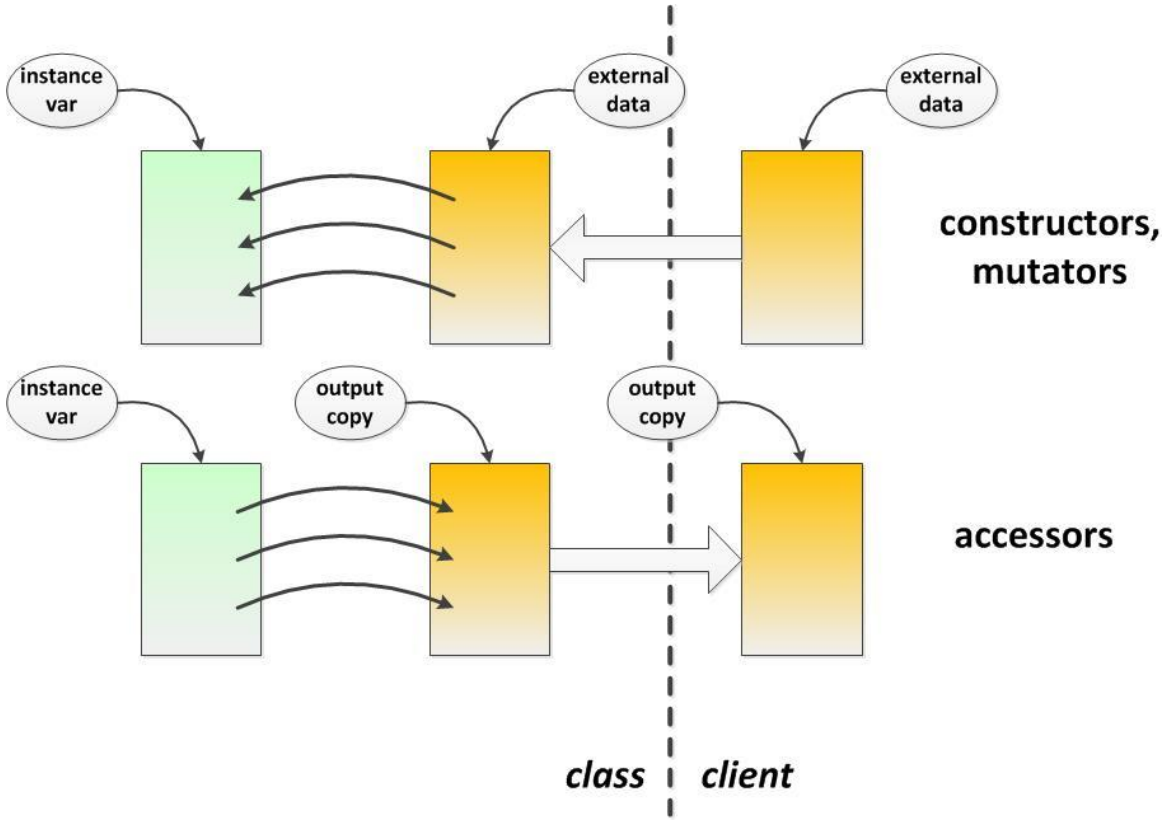
See ***Array1DIntsClass.java***
in **Example Source Code**

```
//-------------------------------------------
// display methods
//-------------------------------------------

// prints the entire array as comma-separated data
public String toString() {

    // the initial empty string keeps compiler happy
    String temp = "" + data[0];

    // append all other data, comma-separated
    for (int i=1; i<data.length; i++) {
        temp += (", " + data[i]);
    }
    return temp;
}

// prints the entire array in a more labeled way
public void print() {

    System.out.println("max size:\t" + getMaxSize());
    System.out.println("used size:\t" + getUsedSize());
    System.out.println("room to add:\t" + isRoom());

    // labeled data
    for (int i=0; i<data.length; i++) {
        System.out.println("data[" + i + "] =\t" + data[i]);
    }
}

// overloaded print, with a text label
public void print(String message) {
    separator(DEF_SEP, message.length() + DEF_OHANG);
    System.out.println(message);
    separator(DEF_SEP, message.length() + DEF_OHANG);
    print();
}

// PRIVATE display utility to print a separator line
private void separator(char sep, int num) {
    for (int i=0; i<num; i++) {
        System.out.print(sep);
    }
    System.out.println();
}
```

Arrays As Method Input/Output

- When arrays are passed into/out from a method, remember that what's really being passed is the **array reference**
  - It "points to" the real data somewhere in memory
- Methods such as constructors, accessors, and mutators must take care not to share any array references to instance variables with any outside client program
  - To do so would permit a calling client program to directly change instance array elements, using an externally known array reference
  - This would **violate encapsulation**, because the intended accessors/mutators could be bypassed

Array/Object Data-Safe Transfer



- Whenever passing arrays or objects via a method call, always COPY all the data internally to an instance variable, so to not violate ***encapsulation***
  - Looping over all elements, for an array
  - Field-by-field transfer, for an object

Arrays As Method Input/Output

To **define a method that uses an array as a parameter**, use this syntax:

general: accessModifier returnType methodName(dataType [] arrayName)

example: public void setTemperatures (double [ ] dailyTemps)

To **define a method that returns an array**, use this syntax:

general: accessModifier dataType [] methodName(parameterList)

example: public double [ ] getTemperatures (SimpleDate date)

To **use an array as an argument when calling a method**, use the array name without brackets:

general: methodName(arrayName)

examples: setTemperatures(todayTemps)

double [ ] tempData = getTemperatures(UtilsRL.today())

Mutators For Arrays

- Any **mutator** that accepts an array parameter should:
  - Copy over the elements from the parameter into the internal instance variable array
  - The object thus maintains its own private version of data (the instance variable array)
  - Encapsulation is preserved
- Here we have 2 overloaded mutators:
  - Adding an int to next unused index
  - Writing over data with new array

```
//-----------------------------------------
// mutators
//-----------------------------------------

// mutator for one element
// assume new elements are added to first open space
public void setData(int value) {

    if (isRoom()) {
        data[count] = value;
        count++;
    }
    else {
        System.out.println(ERROR_FULL);
    }

}

// mutator for entire array
public void setData(int [] values) {

    // resize the array
    data = new int [values.length];
    count = data.length;

    // copy over all data
    for (int i=0; i<data.length; i++) {
        data[i] = values[i];
    }
}
```

See ***Array1DIntsClass.java*** in **Example Source Code**

Accessors For Arrays

- Any **accessor** that returns an array parameter should:
  - Return an array reference to a copy of the instance variable array
  - The object thus maintains its own private version of the data (the instance variable array)
  - Encapsulation is preserved
- Here we have 2 overloaded accessors:
  - Getting an int from a specified index, if valid
  - Returning a copy of the entire int array

```
//-----------------------------------------------
// accessors
//-----------------------------------------------

// accessor for one element
public int getData(int index) {

    // check for valid index first
    if (isValidIndex(index)) {
        return data[index];
    }
    else {
        // no action taken, but we gotta return something
        System.out.println(ERROR_INDEX + index);
        return 0;
    }
}

// accessor for entire array
public int [] getData() {

    // set up a bogus return array
    int [] temp = new int [data.length];

    // transfer over data and return bogus copy
    for (int i=0; i<data.length; i++) {
        temp[i] = data[i];
    }
    return temp;
}
```

See ***Array1DIntsClass.java*** in **Example Source Code**

Equality For Arrays

- To check if two array classes are equal:
  - Checks types using instanceof
  - Check array lengths
  - Check element by element for equality
  - If ALL checks pass, the array objects are equal

```
//-----------------------------------------
// equality
//-----------------------------------------

// equals check
public boolean equals(Object obj) {
    boolean result = true;

    if (obj instanceof Array1DIntsClass) {
        // apples and apples, keep checking
        Array1DIntsClass a = (Array1DIntsClass) obj;

        // checks sizes next
        if (this.data.length == a.getMaxSize()) {

            // finally, check element by element
            for (int i=0; i<this.data.length; i++) {

                // if ANY one element is not identical, not equal
                if (this.data[i] != a.getData(i)) {
                    result = false;
                }
            }
        }
        else {
            result = false;
        }
    }
    else {
        // apples and bananas
        result = false;
    }

    return result;
}
```

See ***Array1DIntsClass.java*** in **Example Source Code**

Array Utilities: Adding Values

- We want to add new “elements” to the end of the array
- The source of data could be any one of:
  - User inputs from prompts
  - Random data
  - File data
- Provide overloaded add methods
  - Different data sources
  - Use one common setData() mutator to actually add the data, no matter what the source
- Each method is “self-contained”
  - No inputs
  - No return value (void)
  - Data generation or prompting is all internal

```
//---------------------------------------------
// utility methods: "do something" to one specific element
// note these next methods are all "self-contained"
// they take action w/o any input or return values
//---------------------------------------------

// add element from the command line
public void setDataFromUser() {
    int value = UtilsRL.readInt("Enter new int data > ", false);
    // next method checks for space
    setData(value);
}

// add element using a random element
public void setDataRandom() {
    int value = UtilsRL.randomInt(MIN_RANDOM, MAX_RANDOM);
    // next method checks for space
    setData(value);
}

// add elements from a file
public void setDataFromFile() {
    // STUDENT TO ADD IN LAST PROGRAM... SEE PRIOR LECTURE EXAMPLE
    // open file connection
    // read all lines in as Strings
    // parse Strings into scalars
    // next method checks for space
    //setData(value);
}
```

See ***Array1DIntsClass.java*** in **Example Source Code**

Array Utilities: Updating Values

- We might want to modify specific values of the array
  - Obtain desired index
    - Keep reprompting
  - Is the index valid?
    - Notify user if not
  - Obtain updated value from user
  - Modify value at specified array index
- **Each method is "self-contained"**
  - No inputs
  - No return value (void)
  - Index and data prompting are all internal

```
// update one specific element of the array
public void updateElement() {

    int index, value;

    // determine which element to update
    index = promptValidIndex();

    if (index != -1) {
        // prompt for update value
        value = UtilsRL.readInt("Updated value? > ", false);

        // perform the array element update
        data[index] = value;
    }
    else {
        System.out.println(ERROR_EMPTY);
    }
}

// increment one specific element of the array
public void incrementElement() {

    int index, value;

    // determine which element to update
    index = promptValidIndex();

    if (index != -1) {
        // prompt for update value
        value = UtilsRL.readInt("Increment value? > ", false);

        // perform the array element update
        data[index] += value;
    }
    else {
        System.out.println(ERROR_EMPTY);
    }
}
```

See ***Array1DIntsClass.java*** in **Example Source Code**