Lecture 28:
1-D Arrays, Part II

Sierra College
CSCI-12
Spring 2015
Mon 05/11/15

Announcements

- **General**
  - An ongoing router issue has slowed my grading progress the past few days, but I'm working to finish up (apologies!)
- **Schedule**
  - 2 more lectures, 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)
  - Current program, then one LAST one (due after final)
  - Added lab hours, tentatively:
    - **Friday 5/15   9am-noon** (for finish-up work on Dam class, or 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

Lecture Topics

- **Last time**:
  - 1-D arrays
    - Definitions
    - Creating and initializing
    - Array mechanics
    - Common array operations
- **Today**
  - 1-D Arrays
    - Copying, resizing, equality
    - Counting with arrays
    - Object arrays
    - Command line execution
    - Parsing strings
    - Creating objects from strings and files

For Next Time

- **Lecture Prep**
  - Text readings and lecture notes
- **Program**
  - Continue building up your ***Dam*** class
    - Build a little, test a little (add test code to *main*() as you go)
    - Implement the entire starter class for ONE instance variable
    - Then repeat the steps for instance variables 2-7
    - Then add the utility methods in the API

Quick Recap Of 1-D Arrays

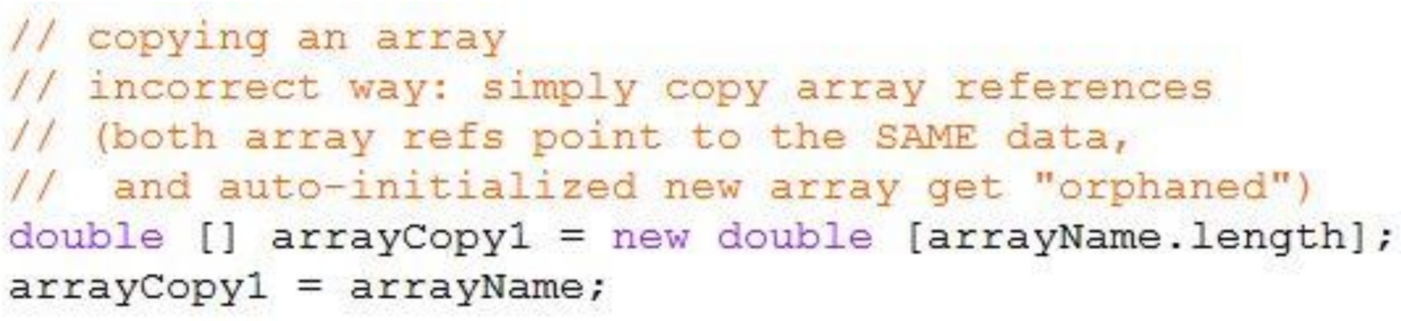
- Array terminology
  - Elements, indices, array reference
- Creating arrays
- Looping thru an array
  - Array bounds
- Indexing into/from an array
  - arrayName[i]

- See ***Arrays1DExamples.java*** in **Example Source Code** for all following examples

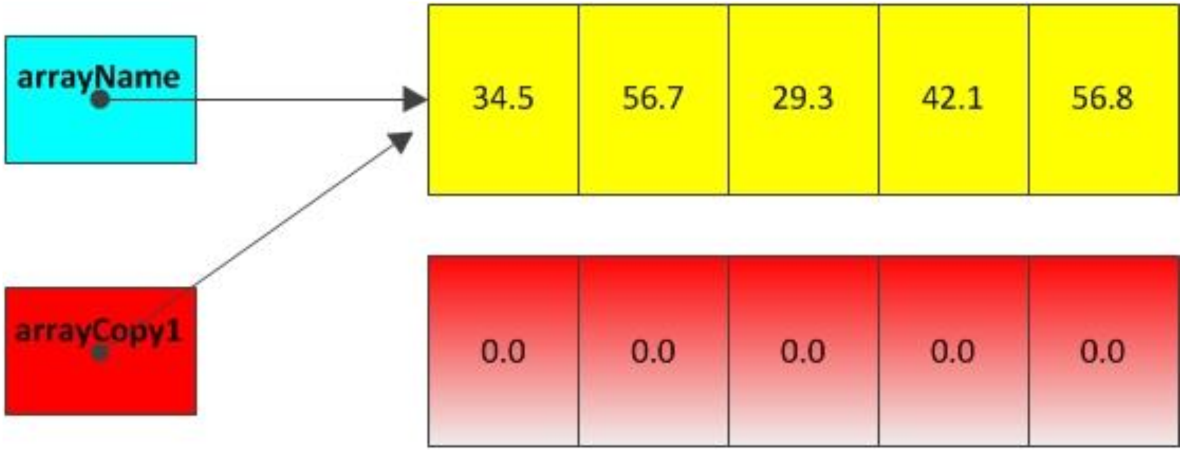
Copying An Array

- There are two ways of copying an array
- The incorrect way:
  - Set up a new array of the same size as the original array
  - Copy the original array reference to the new array reference
  - This results in two array references pointing to the same data
- The correct way:
  - Set up a new array of the same size as the original array
  - In a loop, transfer the values oldArray[i] → newArray[i]
  - This results in two distinct sets of identical data

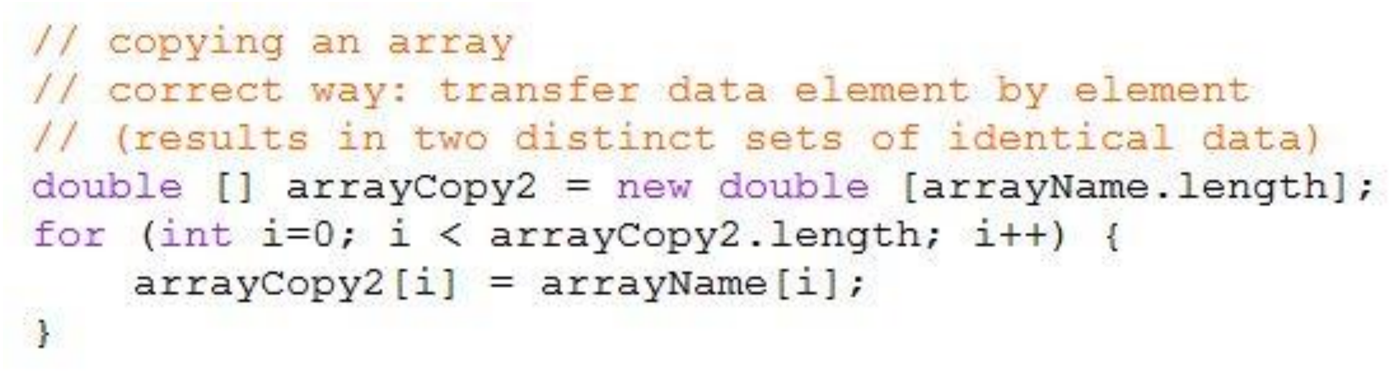
Copying An Array (Incorrect)



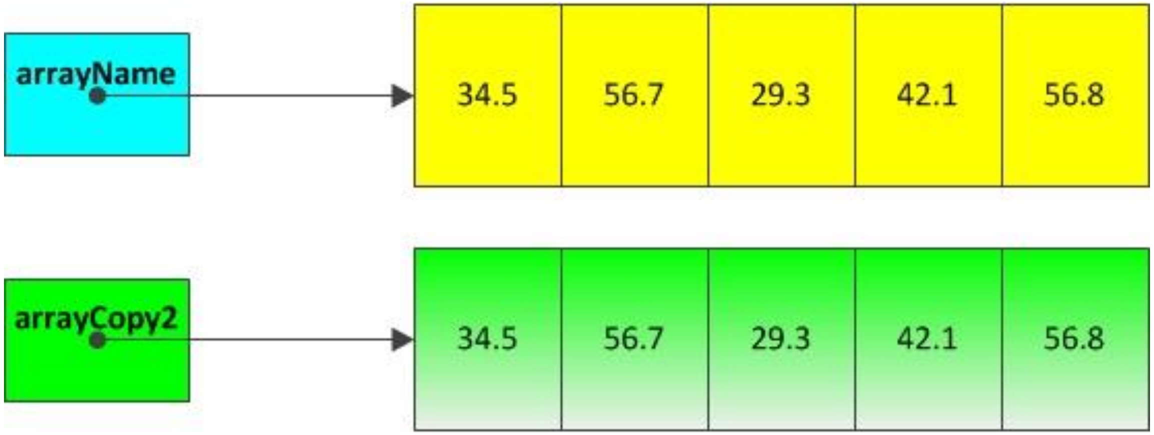
```
// copying an array
// incorrect way: simply copy array references
// (both array refs point to the SAME data,
//  and auto-initialized new array get "orphaned")
double [] arrayCopy1 = new double [arrayName.length];
arrayCopy1 = arrayName;
```



Copying An Array (Correct)



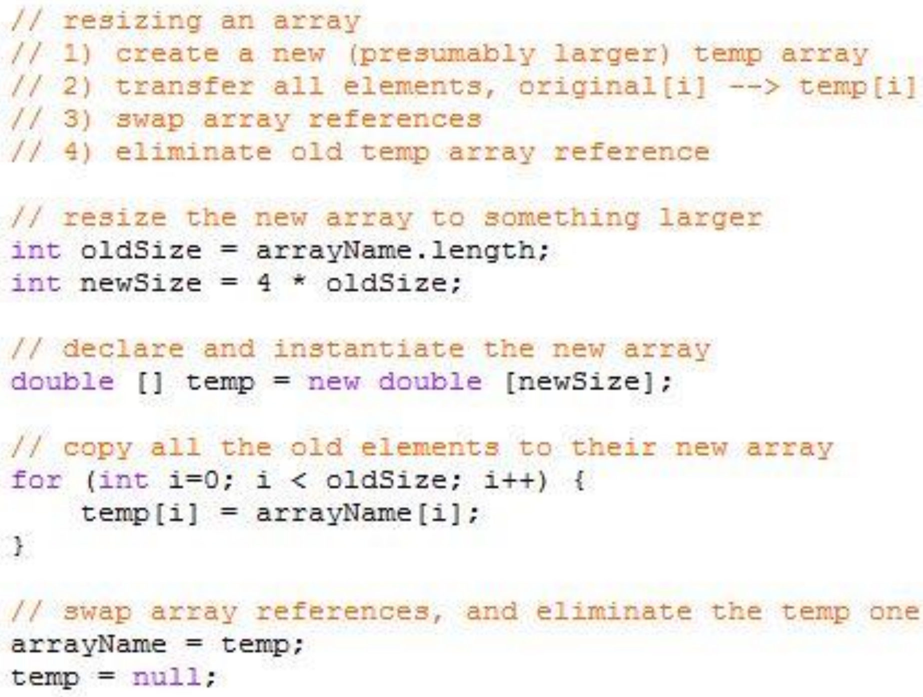
```
// copying an array
// correct way: transfer data element by element
// (results in two distinct sets of identical data)
double [] arrayCopy2 = new double [arrayName.length];
for (int i=0; i < arrayCopy2.length; i++) {
    arrayCopy2[i] = arrayName[i];
}
```



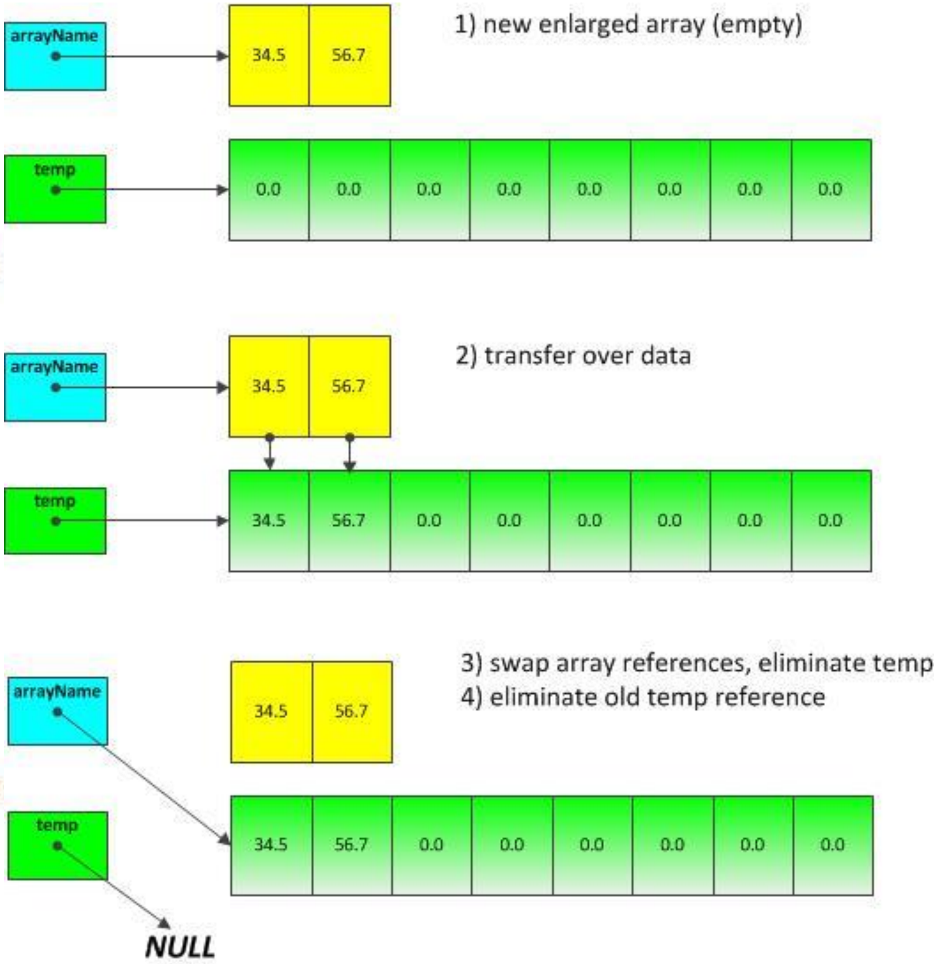
34.5 56.7 29.3 42.1 56.8

34.5 56.7 29.3 42.1 56.8

Resizing An Array



```
// resizing an array
// 1) create a new (presumably larger) temp array
// 2) transfer all elements, original[i] --> temp[i]
// 3) swap array references
// 4) eliminate old temp array reference

// resize the new array to something larger
int oldSize = arrayName.length;
int newSize = 4 * oldSize;

// declare and instantiate the new array
double [] temp = new double [newSize];

// copy all the old elements to their new array
for (int i=0; i < oldSize; i++) {
    temp[i] = arrayName[i];
}

// swap array references, and eliminate the temp one
arrayName = temp;
temp = null;
```



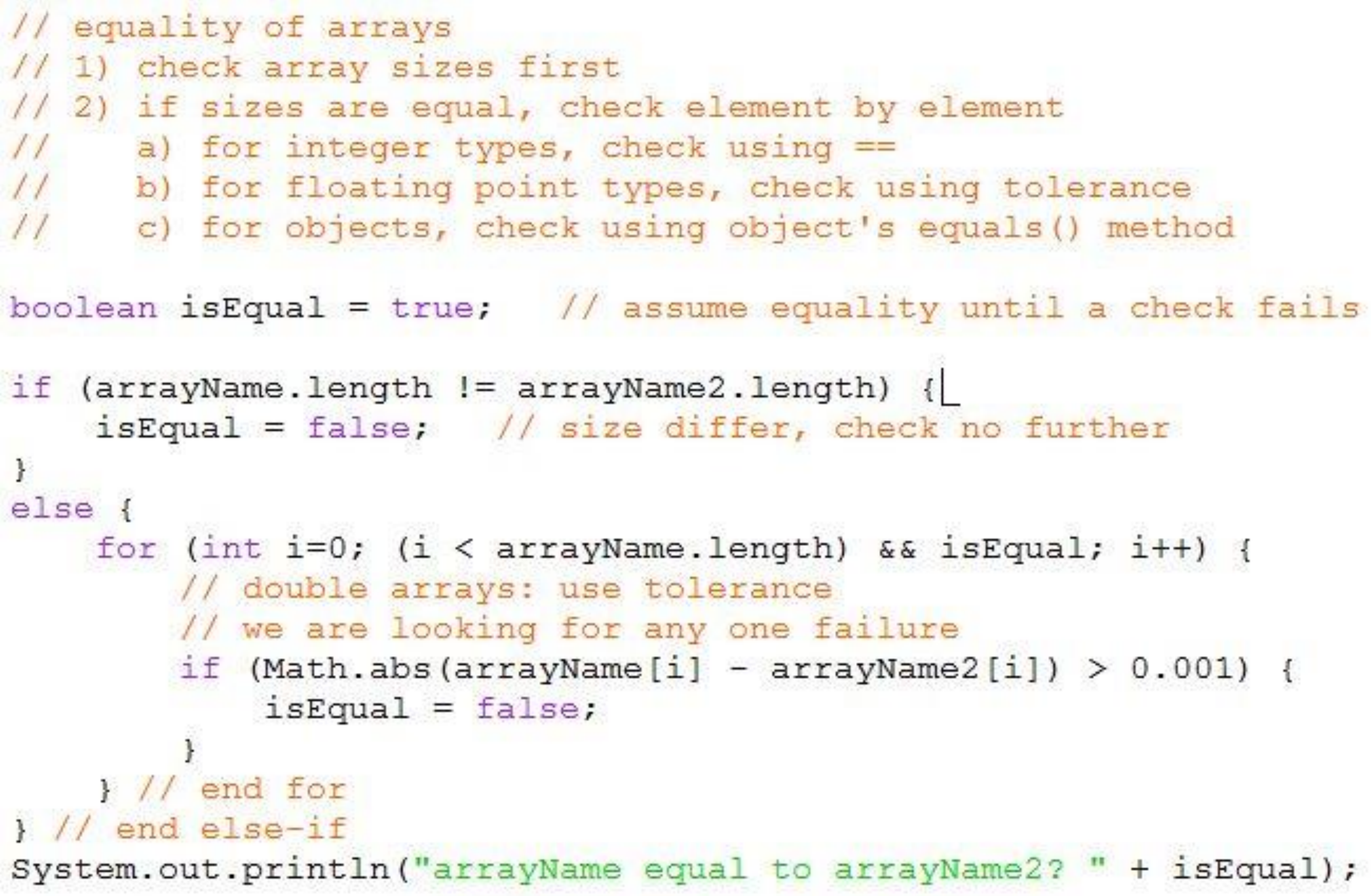
1) new enlarged array (empty)

2) transfer over data

3) swap array references, eliminate temp

4) eliminate old temp reference

Equality Of Arrays

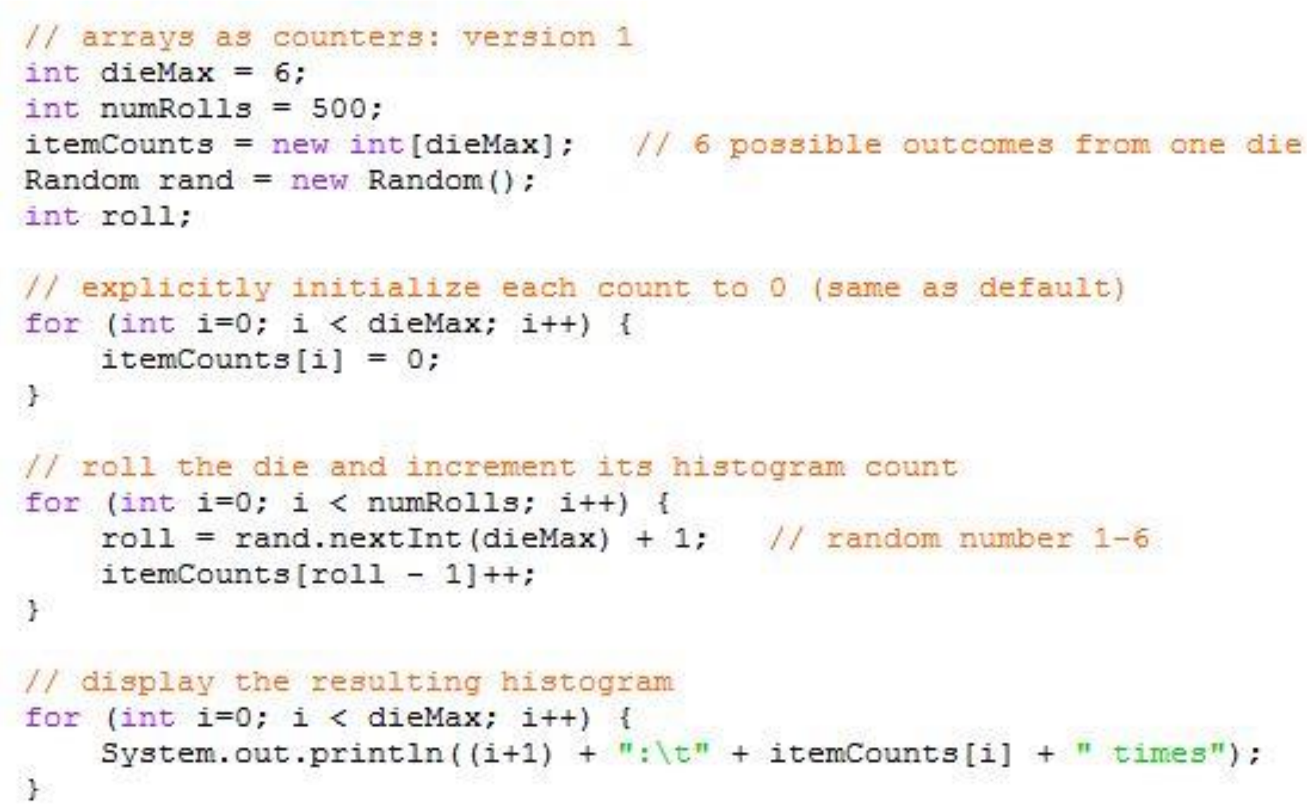


```
// equality of arrays
// 1) check array sizes first
// 2) if sizes are equal, check element by element
//    a) for integer types, check using ==
//    b) for floating point types, check using tolerance
//    c) for objects, check using object's equals() method

boolean isEqual = true;   // assume equality until a check fails

if (arrayName.length != arrayName2.length) {
    isEqual = false;   // size differ, check no further
}
else {
    for (int i=0; (i < arrayName.length) && isEqual; i++) {
        // double arrays: use tolerance
        // we are looking for any one failure
        if (Math.abs(arrayName[i] - arrayName2[i]) > 0.001) {
            isEqual = false;
        }
    } // end for
} // end else-if
System.out.println("arrayName equal to arrayName2? " + isEqual);
```

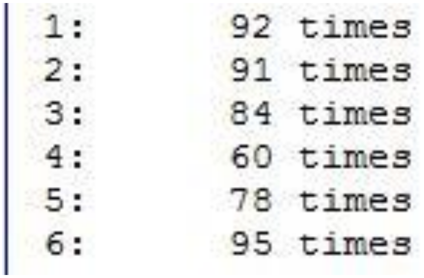
Counters With An Array

- To count outcome occurences using an array (**histogram**):
  - Define an *int* array, sized by the number of “possibilities” to be counted
  - Each array element is a counter for a specific item or outcome
- Example: counting dice rolls
  - Throw a die, how many times does each side come up?
  - Set up an *int* array of 6 elements, one for each of 1 to 6
  - Initialize each count to 0 (the auto-initialization default)
  - For a roll, use *Random* to generate a “**roll**” from 1-6
  - For each roll, increment the (**roll-1**) element

Counter Example, ver.1



```java
// arrays as counters: version 1
int dieMax = 6;
int numRolls = 500;
itemCounts = new int[dieMax];   // 6 possible outcomes from one die
Random rand = new Random();
int roll;

// explicitly initialize each count to 0 (same as default)
for (int i=0; i < dieMax; i++) {
    itemCounts[i] = 0;
}

// roll the die and increment its histogram count
for (int i=0; i < numRolls; i++) {
    roll = rand.nextInt(dieMax) + 1;   // random number 1-6
    itemCounts[roll - 1]++;
}

// display the resulting histogram
for (int i=0; i < dieMax; i++) {
    System.out.println((i+1) + ":\t" + itemCounts[i] + " times");
}
```

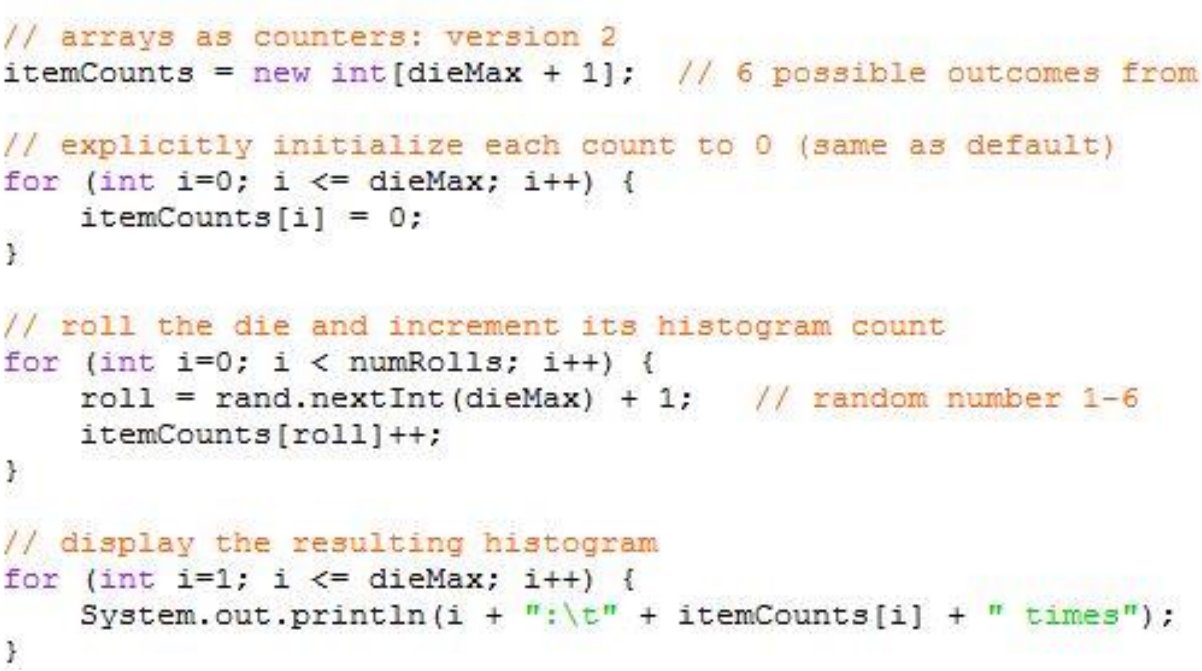


```
1:      92 times
2:      91 times
3:      84 times
4:      60 times
5:      78 times
6:      95 times
```

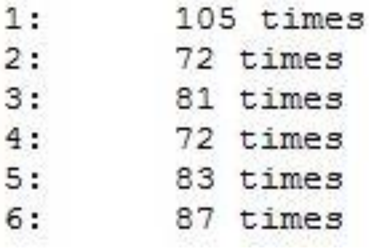
Counter Improvements?

- The prior approach might not be ideal:
  - Need to subtract 1 from the die roll to store count in array
  - Need to add 1 to the array index to display die roll counts
- Another approach:
  - Use an array of size 7, rather than size 6
  - Elements 1-6 directly are the counts for die rolls 1-6
  - The die roll and the array index are now equal
  - Index 0 is now wasted space, but we just ignore it

Counter Example, ver.2



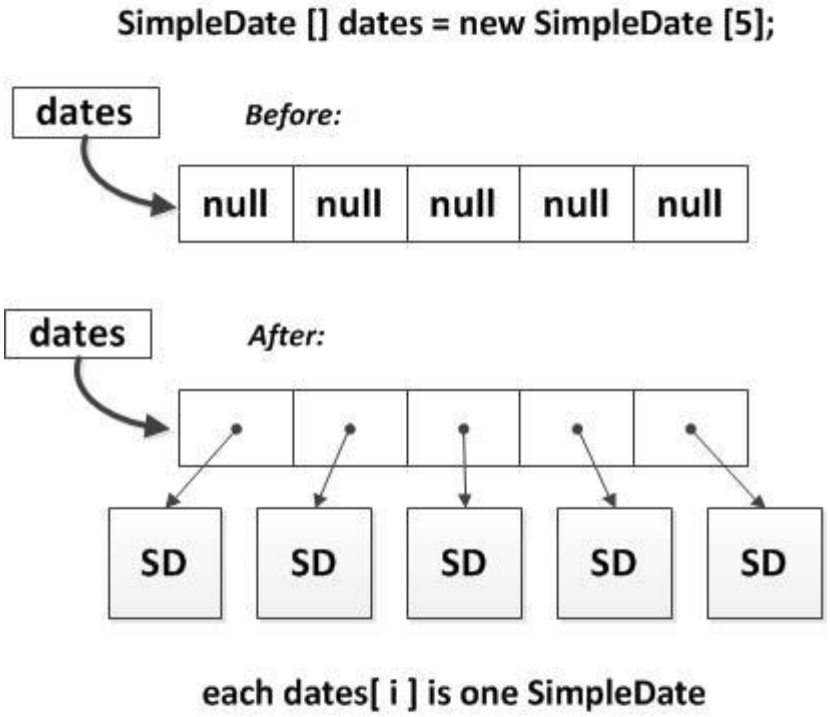
```
// arrays as counters: version 2
itemCounts = new int[dieMax + 1];  // 6 possible outcomes from one die, plus 0

// explicitly initialize each count to 0 (same as default)
for (int i=0; i <= dieMax; i++) {
    itemCounts[i] = 0;
}

// roll the die and increment its histogram count
for (int i=0; i < numRolls; i++) {
    roll = rand.nextInt(dieMax) + 1;   // random number 1-6
    itemCounts[roll]++;
}

// display the resulting histogram
for (int i=1; i <= dieMax; i++) {
    System.out.println(i + ":\t" + itemCounts[i] + " times");
}
```



```
1:      105 times
2:      72 times
3:      81 times
4:      72 times
5:      83 times
6:      87 times
```

Object Arrays

- Everything we have discussed thus far is equally applicable to **object arrays** (arrays of objects)
- Each array element of an object array is now an **object reference** to an object of the specified class datatype
  - *null* upon array instantiation, by default
  - an object reference, upon data instantiation
- The array element **objArray[i]** may then be manipulated just like any other object of that class datatype

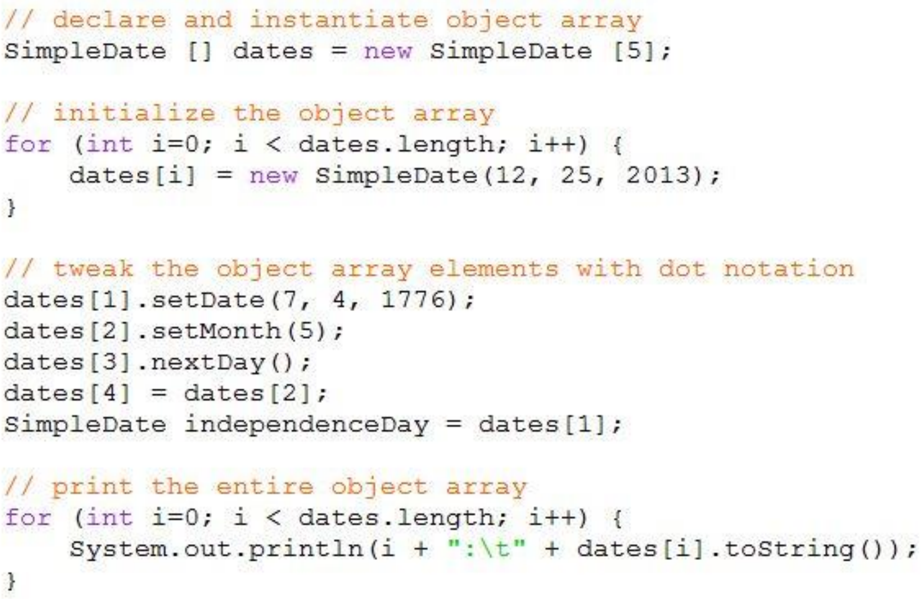

SimpleDate [] dates = new SimpleDate [5];
dates
Before:
null
null
null
null
null
dates
After:
SD
SD
SD
SD
SD
each dates[i] is one SimpleDate

- See ***Arrays1DExamples.java*** in **Example Source Code**

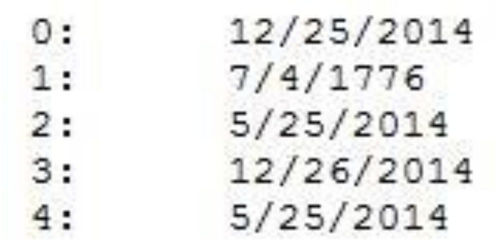
Object Array Elements

- Each element **objArray[i]** is simply an object (reference) of the specified class type
  - For our purposes, it “is” an object, and may be treated as such
  - It may appear or be used anywhere an object of that type may appear or be used
- All of the object’s API methods are available via its index and **dot notation**:

  objArray[i].methodName()



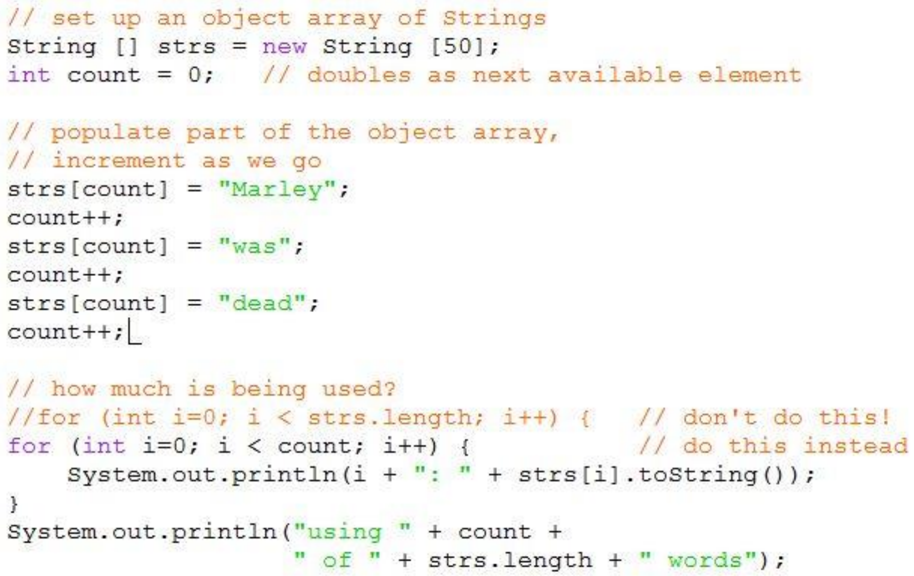
```
// declare and instantiate object array
SimpleDate [] dates = new SimpleDate [5];

// initialize the object array
for (int i=0; i < dates.length; i++) {
    dates[i] = new SimpleDate(12, 25, 2013);
}

// tweak the object array elements with dot notation
dates[1].setDate(7, 4, 1776);
dates[2].setMonth(5);
dates[3].nextDay();
dates[4] = dates[2];
SimpleDate independenceDay = dates[1];

// print the entire object array
for (int i=0; i < dates.length; i++) {
    System.out.println(i + ":\t" + dates[i].toString());
}
```



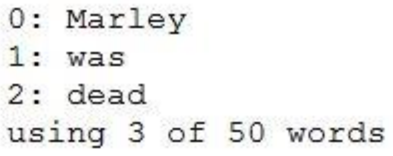
```
0:      12/25/2014
1:      7/4/1776
2:      5/25/2014
3:      12/26/2014
4:      5/25/2014
```

Object Array Size vs. Usage

- One “gotcha” with object arrays is that each element starts out as *null*, until it gets updated
  - Unlike integers or floating points, which default to 0 or 0.0
- Trying to call a method of an object element not yet instantiated will result in a *NullPointerException*
- Need to distinguish between:
  - The **max size** of an object array
    - **objArray.length**
  - The **actual usage** of an object array
    - Need to keep track yourself



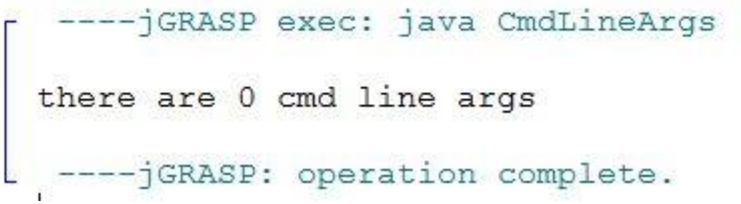
```
// set up an object array of Strings
String [] strs = new String [50];
int count = 0;   // doubles as next available element

// populate part of the object array,
// increment as we go
strs[count] = "Marley";
count++;
strs[count] = "was";
count++;
strs[count] = "dead";
count++;

// how much is being used?
//for (int i=0; i < strs.length; i++) {    // don't do this!
for (int i=0; i < count; i++) {            // do this instead
    System.out.println(i + ": " + strs[i].toString());
}
System.out.println("using " + count +
                   " of " + strs.length + " words");
```



```
0: Marley
1: was
2: dead
using 3 of 50 words
```

Command Line Execution

- Thus far, we have only executed Java programs from the jGRASP IDE
- But "under the hood", Java applications run as follows:
  - ***MyJavaClass.java*** → ***MyJavaClass.class***, using OS process "javac"
  - ***MyJavaClass.class*** is executed by the JVM, using OS process "java"



```
 ----jGRASP exec: java CmdLineArgs

there are 0 cmd line args

 ----jGRASP: operation complete.
```

- Java applications themselves may be part of some larger software application
  - The main() method may itself need some data at startup
  - May want to pass in I/O files, config settings, etc.
  - Perhaps a Java program is controlled by some larger program

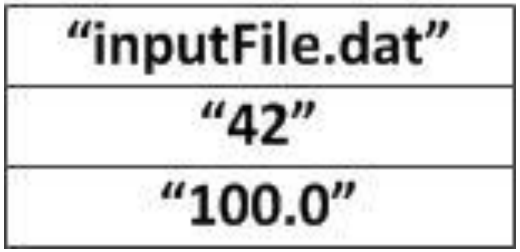
Command Line Arguments

- The **standard main() method interface** provides for passing in runtime arguments:

  public static void main(String [ ] args) { … }

- In the above interface:
  - *args* is an array of *String* arguments (no matter what the data)
  - Strings are “generic”: they can be converted into their numerical equivalents, using the **wrapper methods** (parseInt(), parseDouble(), etc.)
- Example:

args =



“inputFile.dat”
“42”
“100.0”

Executing With Command Line Arguments

From an OS command line

- In a command shell window:

java MyJavaClass <argList>

where:

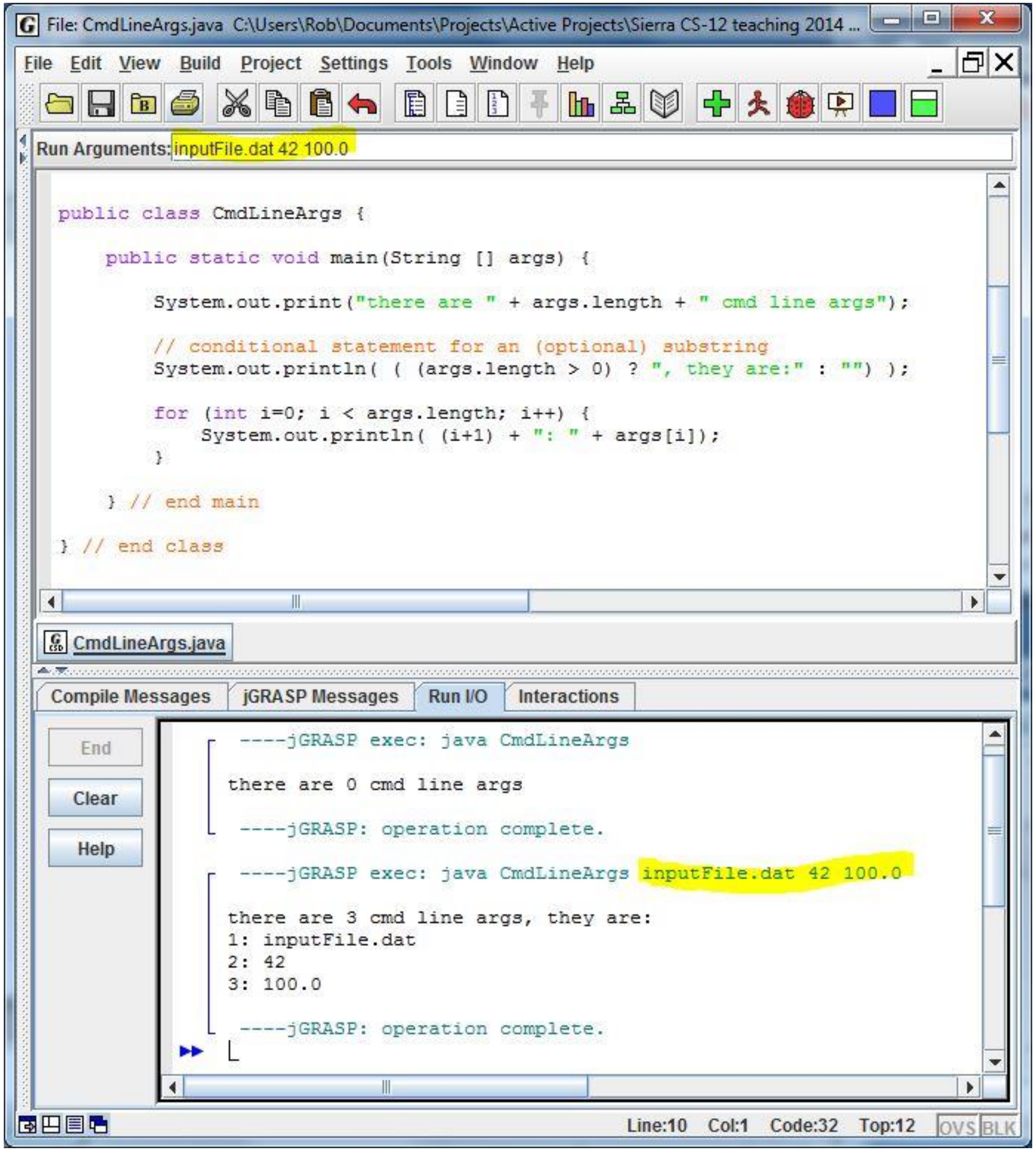
MyJavaClass← ***MyJavaClass.class***

(compiled Java bytecode)

From jGRASP

- Set **Build: Run Arguments**
  - A **Run Arguments** text box appears above the editor
- Specify a space-separated list of arguments (optional)
- Run program as usual (either with or w/o debugger)

Example: Echo Command Line Arguments



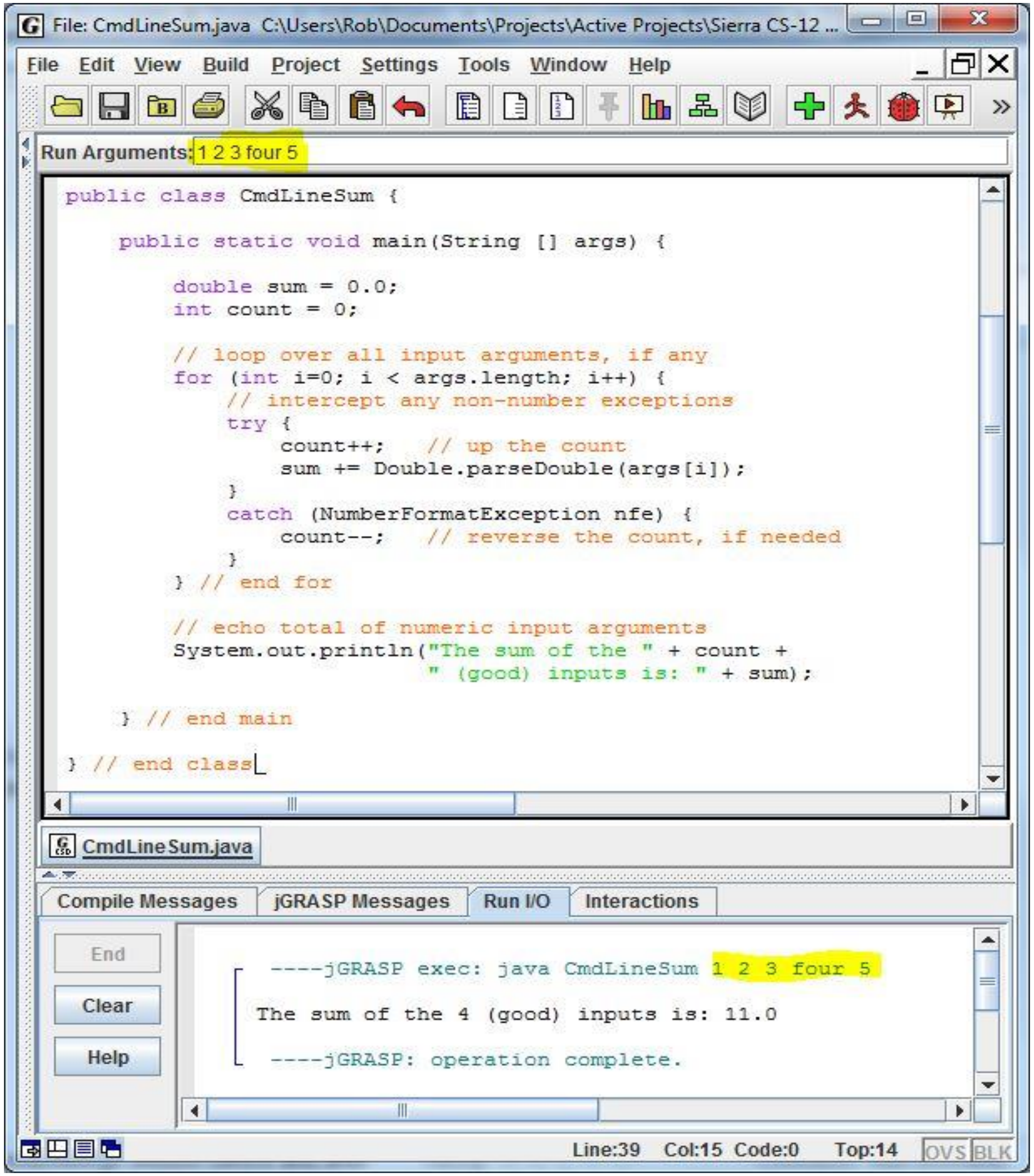
File: CmdLineArgs.java C:\Users\Rob\Documents\Projects\Active Projects\Sierra CS-12 teaching 2014 ...
File Edit View Build Project Settings Tools Window Help
CmdLineArgs.java
Compile Messages
jGRASP Messages
Run I/O
Interactions
End
Clear
Help
Line:10 Col:1 Code:32 Top:12 OVS BLK

Run Arguments: inputFile.dat 42 100.0

```java
public class CmdLineArgs {

    public static void main(String [] args) {

        System.out.print("there are " + args.length + " cmd line args");

        // conditional statement for an (optional) substring
        System.out.println( ( (args.length > 0) ? ", they are:" : "") );

        for (int i=0; i < args.length; i++) {
            System.out.println( (i+1) + ": " + args[i]);
        }

    } // end main

} // end class
```

```
 ----jGRASP exec: java CmdLineArgs

there are 0 cmd line args

 ----jGRASP: operation complete.

 ----jGRASP exec: java CmdLineArgs inputFile.dat 42 100.0

there are 3 cmd line args, they are:
1: inputFile.dat
2: 42
3: 100.0

 ----jGRASP: operation complete.
```

Example: Sum Command Line Arguments



Run Arguments: 1 2 3 four 5

```
public class CmdLineSum {

    public static void main(String [] args) {

        double sum = 0.0;
        int count = 0;

        // loop over all input arguments, if any
        for (int i=0; i < args.length; i++) {
            // intercept any non-number exceptions
            try {
                count++;    // up the count
                sum += Double.parseDouble(args[i]);
            }
            catch (NumberFormatException nfe) {
                count--;    // reverse the count, if needed
            }
        } // end for

        // echo total of numeric input arguments
        System.out.println("The sum of the " + count +
                           " (good) inputs is: " + sum);

    } // end main

} // end class
```

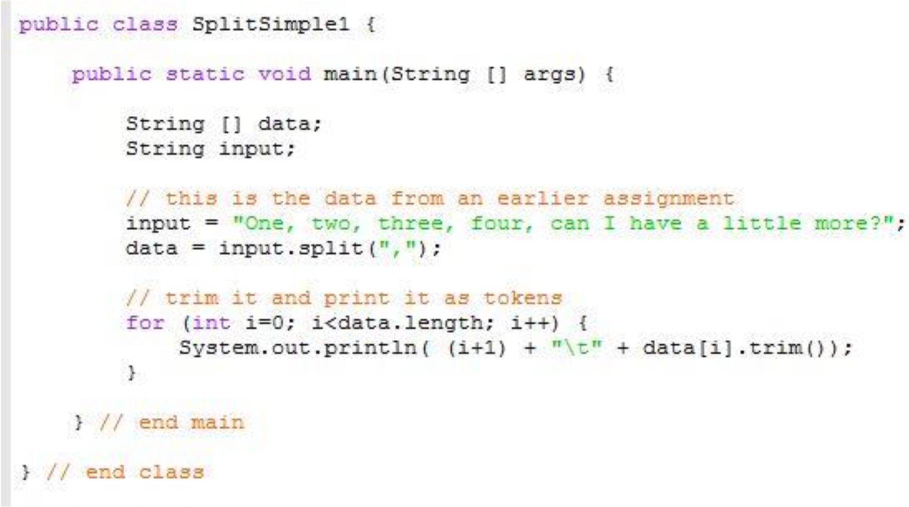
```
 ----jGRASP exec: java CmdLineSum 1 2 3 four 5

The sum of the 4 (good) inputs is: 11.0

 ----jGRASP: operation complete.
```

Parsing Strings, Revisited

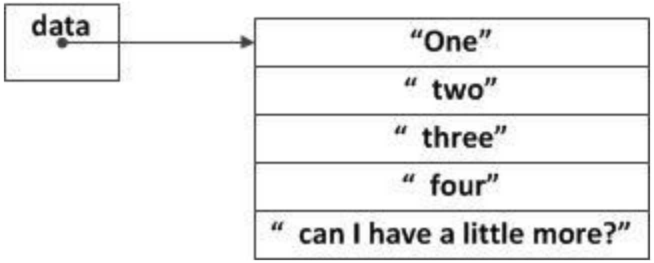
- Recall the manual *String* parsing done in an earlier program?? (HW12: Strings)
- Does Java provide a better way to do this? Of course!
- The ***String* class split() method** splits up an existing *String* object on a specified *String* delimiter:
  - **String [] stringArray stringVar.split(stringDelimiter)**
  - See the details in the *String* Java API
  - Returns an array of *Strings*
  - Works similar to what *args* provides for command line arguments
  - See examples on next slides

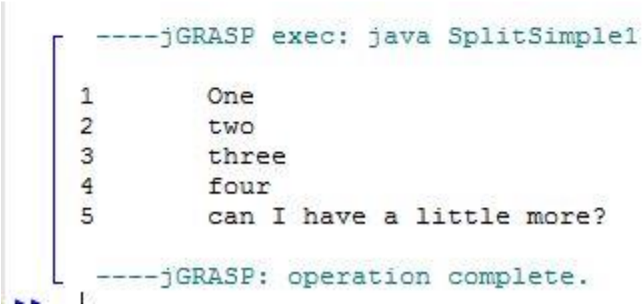
Splitting Simple Strings

- Invoke the **split()** API method of an existing *String*
  - This is nothing but your parsing exercise from a prior HW!
  - Specify the delimiter
- Result is an <u>array</u> of *Strings*
  - Notice most of the strings have some leading whitespace
  - Might need some trim() cleanup



```java
public class SplitSimple1 {

    public static void main(String [] args) {

        String [] data;
        String input;

        // this is the data from an earlier assignment
        input = "One, two, three, four, can I have a little more?";
        data = input.split(",");

        // trim it and print it as tokens
        for (int i=0; i<data.length; i++) {
            System.out.println( (i+1) + "\t" + data[i].trim());
        }

    } // end main

} // end class
```



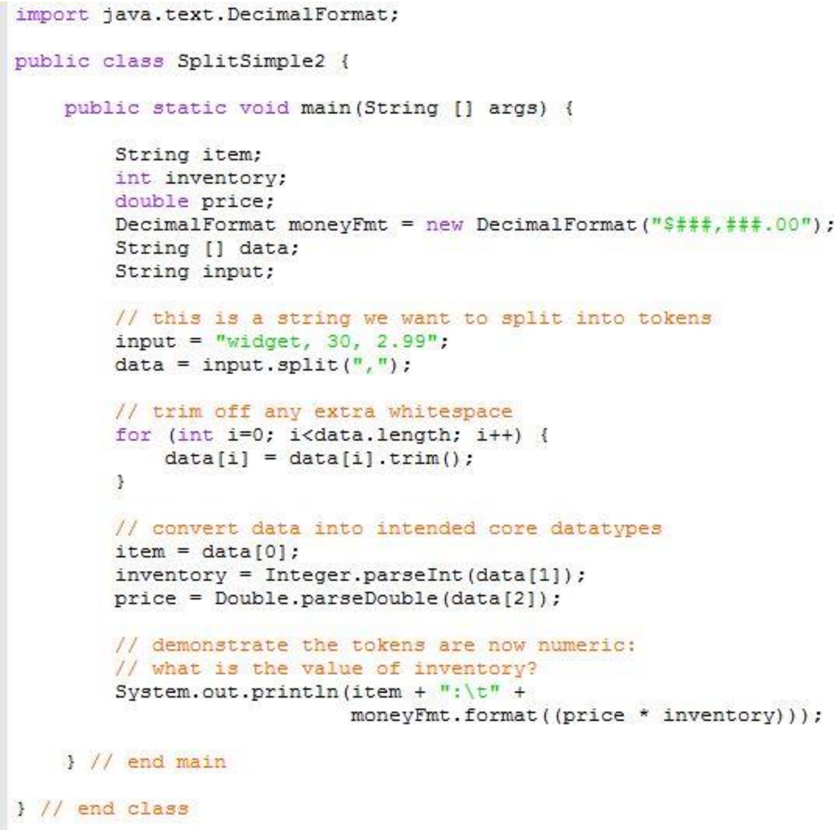


```
----jGRASP exec: java SplitSimple1

1       One
2       two
3       three
4       four
5       can I have a little more?

----jGRASP: operation complete.
```

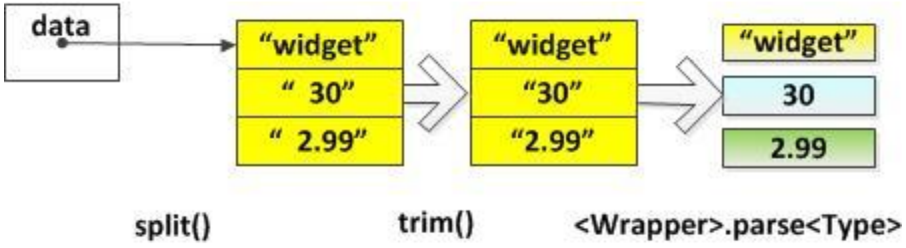
See ***SplitSimple1.java*** in **Example Source Code**

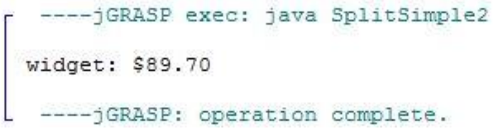
Split With Data Conversions

- In this example, a given string is:
  - ***split()*** into tokens
  - Stripped of whitespace using ***trim()***
  - Separated into individual scalars using **wrapper class parse methods**



```
import java.text.DecimalFormat;

public class SplitSimple2 {

    public static void main(String [] args) {

        String item;
        int inventory;
        double price;
        DecimalFormat moneyFmt = new DecimalFormat("$###,###.00");
        String [] data;
        String input;

        // this is a string we want to split into tokens
        input = "widget, 30, 2.99";
        data = input.split(",");

        // trim off any extra whitespace
        for (int i=0; i<data.length; i++) {
            data[i] = data[i].trim();
        }

        // convert data into intended core datatypes
        item = data[0];
        inventory = Integer.parseInt(data[1]);
        price = Double.parseDouble(data[2]);

        // demonstrate the tokens are now numeric:
        // what is the value of inventory?
        System.out.println(item + ":\t" +
                           moneyFmt.format((price * inventory)));

    } // end main

} // end class
```



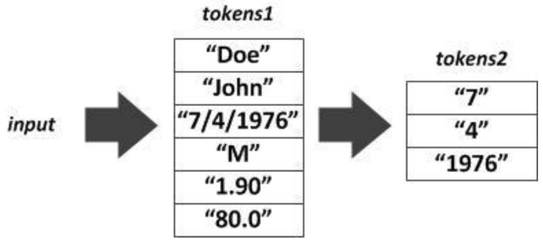


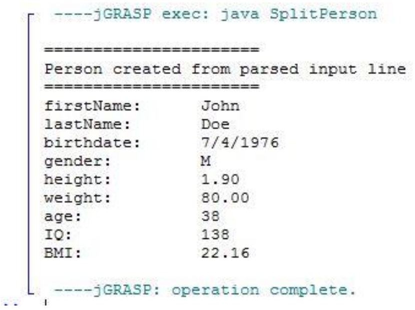
```
 ----jGRASP exec: java SplitSimple2

widget: $89.70

 ----jGRASP: operation complete.
```

See ***SplitSimple2.java*** in **Example Source Code**

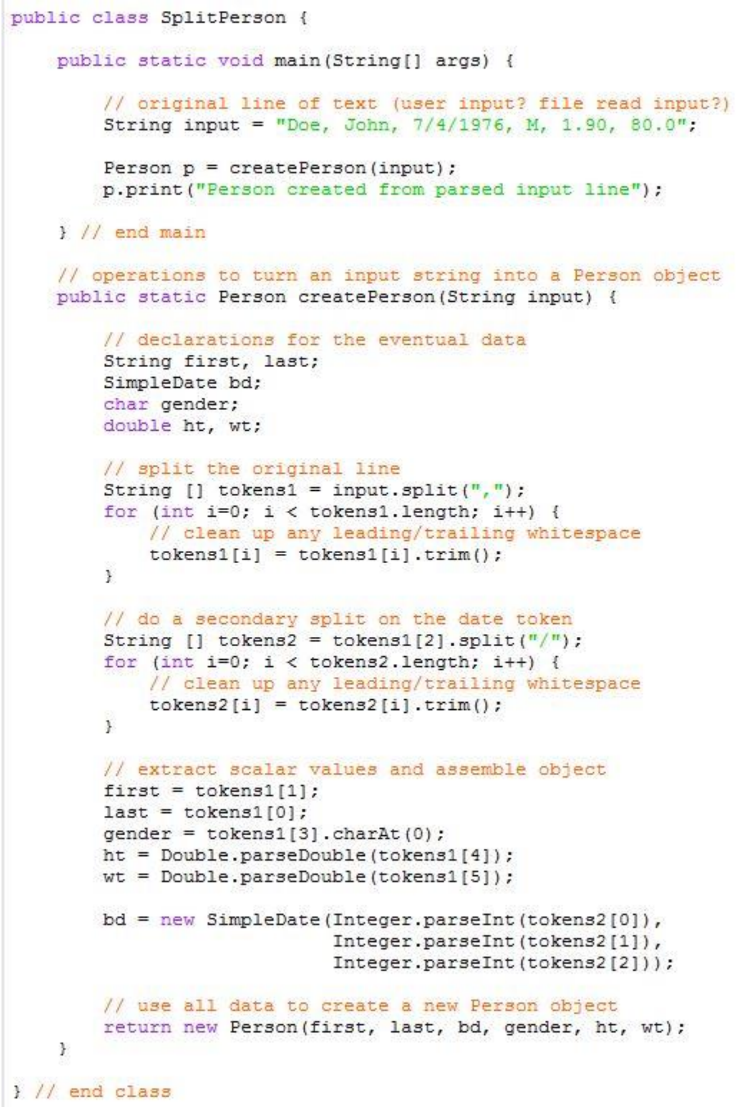
Secondary Split With Object Creation

- Starting from a single line of text, call method to do:
  - Original string is parsed
  - A secondary split is done using a 2nd delimiter
  - Individual variables are extracted
  - New *Person* object returned:





```
 ----jGRASP exec: java SplitPerson

=======================
Person created from parsed input line
=======================
firstName:      John
lastName:       Doe
birthdate:      7/4/1976
gender:         M
height:         1.90
weight:         80.00
age:            38
IQ:             138
BMI:            22.16

 ----jGRASP: operation complete.
```

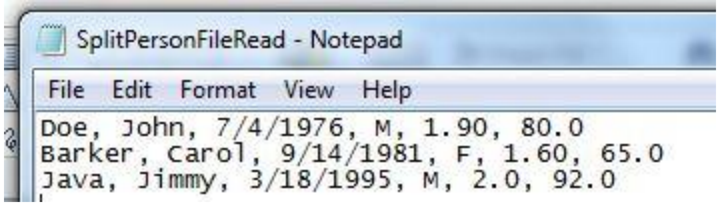


```
public class SplitPerson {

    public static void main(String[] args) {

        // original line of text (user input? file read input?)
        String input = "Doe, John, 7/4/1976, M, 1.90, 80.0";

        Person p = createPerson(input);
        p.print("Person created from parsed input line");

    } // end main

    // operations to turn an input string into a Person object
    public static Person createPerson(String input) {

        // declarations for the eventual data
        String first, last;
        SimpleDate bd;
        char gender;
        double ht, wt;

        // split the original line
        String [] tokens1 = input.split(",");
        for (int i=0; i < tokens1.length; i++) {
            // clean up any leading/trailing whitespace
            tokens1[i] = tokens1[i].trim();
        }

        // do a secondary split on the date token
        String [] tokens2 = tokens1[2].split("/");
        for (int i=0; i < tokens2.length; i++) {
            // clean up any leading/trailing whitespace
            tokens2[i] = tokens2[i].trim();
        }

        // extract scalar values and assemble object
        first = tokens1[1];
        last = tokens1[0];
        gender = tokens1[3].charAt(0);
        ht = Double.parseDouble(tokens1[4]);
        wt = Double.parseDouble(tokens1[5]);

        bd = new SimpleDate(Integer.parseInt(tokens2[0]),
                            Integer.parseInt(tokens2[1]),
                            Integer.parseInt(tokens2[2]));

        // use all data to create a new Person object
        return new Person(first, last, bd, gender, ht, wt);
    }

} // end class
```

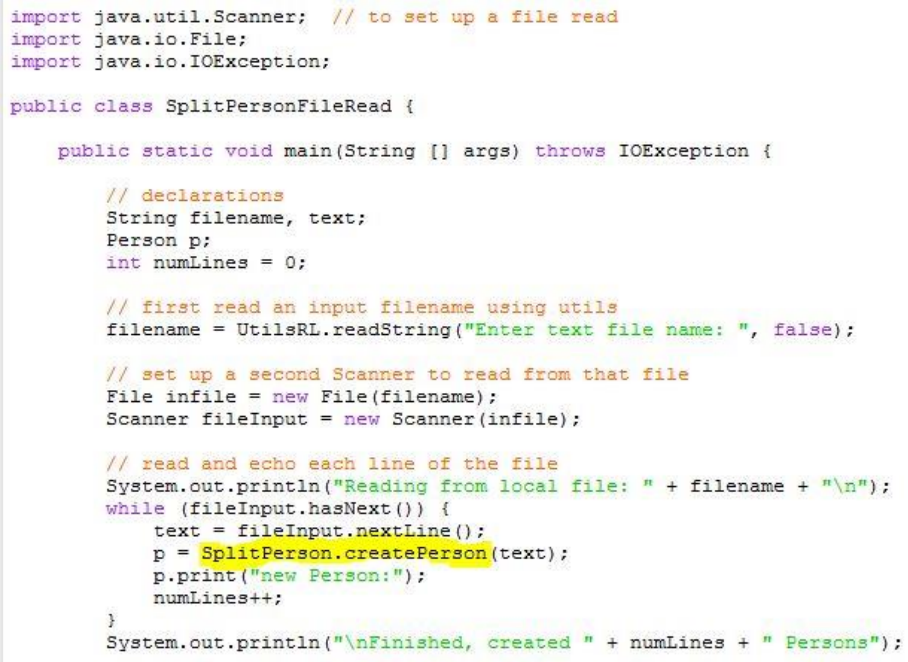
See ***SplitPerson.java*** in **Example Source Code**

Creating Objects From File Input

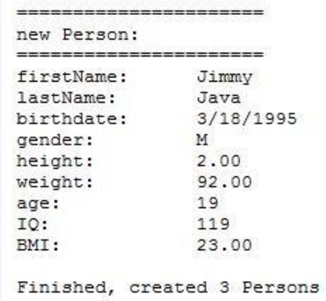
- In the prior example, the steps to create ONE Person object from a String were “carved out” into a method
- Here, use that (static) methoc for an entire file’s worth of input data



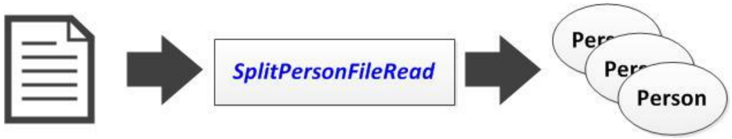
```
Doe, John, 7/4/1976, M, 1.90, 80.0
Barker, Carol, 9/14/1981, F, 1.60, 65.0
Java, Jimmy, 3/18/1995, M, 2.0, 92.0
```



```
import java.util.Scanner;   // to set up a file read
import java.io.File;
import java.io.IOException;

public class SplitPersonFileRead {

    public static void main(String [] args) throws IOException {

        // declarations
        String filename, text;
        Person p;
        int numLines = 0;

        // first read an input filename using utils
        filename = UtilsRL.readString("Enter text file name: ", false);

        // set up a second Scanner to read from that file
        File infile = new File(filename);
        Scanner fileInput = new Scanner(infile);

        // read and echo each line of the file
        System.out.println("Reading from local file: " + filename + "\n");
        while (fileInput.hasNext()) {
            text = fileInput.nextLine();
            p = SplitPerson.createPerson(text);
            p.print("new Person:");
            numLines++;
        }
        System.out.println("\nFinished, created " + numLines + " Persons");

    } // end main

} // end class
```



```
=======================
new Person:
=======================
firstName:       Jimmy
lastName:        Java
birthdate:       3/18/1995
gender:          M
height:          2.00
weight:          92.00
age:             19
IQ:              119
BMI:             23.00

Finished, created 3 Persons
```



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