Today’s Topics

• Comments and/or Questions?
• Programmer defined methods
  – Parameters of primitive types
• Sorting
Exam 1 grading curve

- Grades ranged from C- to A+
- Avg = 81.7%
- SD = 10.7%
- Your grade on 0-4.33 scale computed by:
  - \( \text{Min}(4.33, 3.3 + \frac{(\text{yourScore} - \text{avg})}{SD}) \)
  - \( \text{Min}(4.33, 3.3 + \frac{((\text{raw}/64) - 0.817)}{0.107}) \)
Exam 1 grading curve

• Min(4.33, 3.3 + ((raw/64) – 0.817)/0.107)

• Example:
  – suppose you see -11 on your exam
  – your raw score is 64-11 = 53 points
  – 53/64 = 0.828

• 3.3 + (0.828 – 0.817) / 0.107 = 3.4 which is a B+
variables passed in as arguments when calling a method

• the values of variables (of primitive types) that are passed into a method as arguments remain unchanged after the method ends its execution

• even if the corresponding parameter in the method changes its value, this change is only local to the method, and the value is not “passed back out” of the method.

• we’ll see this in the factorial method on the next slide.
computeFactorial method

class Example {
    public static int computeFactorial(int num) {
        int tempFactorial = 1;
        while (num > 0) {
            tempFactorial = tempFactorial * num;
            num--;
        }
        return tempFactorial;
    }
}

computeFactorial method

- this method is a programmer-defined method (that is, we make up the name and the purpose of the method.)
- the *name* of the method is: computeFactorial
- the *return type* of the method is: int
- there is one *parameter* to the method, which is: num (of type int)
- note that the value of num inside the method changes (it gets decremented in the loop) but this is only a *local* change
- the other *local variable* (tempFactorial) is used to compute the factorial and its value is the one that is returned to the caller.
computeFactorial method

• note that the value of num inside the method changes (it gets decremented in the loop) but this is only a local change the variable sent in as an argument does NOT change its value

• Let’s write a quick program to show how the variable that is “passed in” to the method as an argument doesn’t actually get changed.

• We’ll print the argument’s value before and after the call to the method.
promotion of argument types

- if a method is called with arguments of types different from those specified in the parameter list of the method definition, then Java will try to convert the value to the required type.
- The conversion is done according to Java’s promotion rules.
- Java only allows conversions among the primitive types from smaller to larger.
- primitive types from smaller to larger are: byte, short, int, long, float, double
promotion of argument types

some valid type conversions:
  – a float can be passed in to a method requiring a double
  – an int can be passed in to a method requiring a long, or a float or a double

some type conversions that would cause compiler errors:
  – a float cannot be passed in to a method requiring an int
  – a double cannot be passed in to a method requiring a float
public int test_value(double input_val) {
    if (input_val < 0)
        return -1;
    else
        return 1;
}

Example call in some other method:
int length = 3;
int returned_val;
returned_val = test_value(length);
// call is allowed: int length is promoted to double;
Sorting

• Sorting is an important topic in computer science.
• We'll discuss what sorting is and several algorithms for doing it.
• *There are a myriad of ways to sort.*
• Arrays of data lend themselves easily to being sorted.
• Sorting can be done in ascending or descending order.
• Original unsorted: 38, 41, 22, 12, 67
• We want the list to be: 12, 22, 38, 41, 67
• To accomplish this, one way would be to:
  – Go through the whole list once and find the lowest value.
  – Take it out and put it in a new list.
  – Go through the remaining list of numbers and find the lowest
  – Take that one out and put it at the end of the new list.
  – And so on ... until there's nothing left in the list.
• Does everyone agree that this will achieve a sorted list.
• We didn't say how exactly we'd find the lowest value each time ---
the next slide describes a different way to sort and is more detailed.
Sorting

• Another way would be to:
  – compare the item in the first position to all the other items in the list and swap them *in place* if the item in the first position is greater than the item in the other position.
  – the second pass does the same thing but with the item in the second position.
  – third pass does the same but with the 3rd item.
  – and so on, until the last position is reached.

• Why would this work or not work?
Sorting

• This algorithm doesn’t require a new list like the first one mentioned. All the sorting is done within the one list.

• After the first pass, what is guaranteed about the list?
• After the second pass, what is guaranteed about the list?
• Etc.
• **Original unsorted:** 38, 41, 22, 12, 67

• compare 38 to 41. 38 is not > 41, so leave them in place.

• compare 38 to 22. Swap them. So, now list is 22, 41, 38, 12, 67

• compare 22 to 12. Swap them. So, now list is 12, 41, 38, 22, 67

• compare 12 to 67. 12 is not > 67, so leave them in place.

• **After first pass:** 12, 41, 38, 22, 67
Sorting

• After first pass: 12, 41, 38, 22, 67
• Now start at second position.

• compare 41 to 38. Swap them. So, now list is 12, 38, 41, 22, 67
• compare 38 to 22. Swap them. So, now list is 12, 22, 41, 38, 67
• compare 22 to 67. Leave them.

• After second pass: 12, 22, 41, 38, 67
Sorting

• After second pass: 12, 22, 41, 38, 67

• Now start at third position.

• compare 41 to 38. Swap them. So, now list is 12, 22, 38, 41, 67

• compare 38 to 67. Leave them.

• After third pass: 12, 22, 38, 41, 67
Sorting

• After third pass: 12, 22, 38, 41, 67

• Now start at fourth position.

• compare 41 to 67. Leave them.

• After fourth pass: 12, 22, 38, 41, 67

• Done now.
Another algorithm for sorting is the BubbleSort:

- Compares consecutive numbers in the list
- Swaps them if the two numbers are not in ascending order
- Compare each consecutive pair of numbers in the first pass.
- Next pass, start at first again but go only up until the next to last element, and so on…
- Do \( n - 1 \) passes, where \( n \) is length of the list
Sorting (BubbleSort)

• Original unsorted: 38, 41, 22, 12, 67

• compare 38 to 41. 38 is not > 41, so leave them in place.
• compare 41 to 22. Swap them. So, now list is 38, 22, 41, 12, 67
• compare 41 to 12. Swap them. So, now list is 38, 22, 12, 41, 67
• compare 41 to 67. Leave them.

• After first pass: 38, 22, 12, 41, 67
Sorting (BubbleSort)

• After first pass: 38, 22, 12, 41, 67

• Now start at first position again.
• compare 38 to 22. Swap them. So, now list is 22, 38, 12, 41, 67
• compare 38 to 12. Swap them. So, now list is 22, 12, 38, 41, 67
• compare 38 to 41. Leave them.
• (don't need to compare the 4th and 5th positions) – why???
• After second pass: 22, 12, 38, 41, 67
Sorting (BubbleSort)

• After second pass: 22, 12, 38, 41, 67

• Now start at first position again.

• compare 22 to 12. Swap them. So, now list is 12, 22, 38, 41, 67

• compare 22 to 38. Leave them.

• After third pass: 12, 22, 38, 41, 67
Sorting (BubbleSort)

- After third pass: 12, 22, 38, 41, 67

- Now start at first position again.
  
  - compare 12 to 22. Leave them.

- Done.

- After fourth pass: 12, 22, 38, 41, 67
Sorting (BubbleSort)

• Here's a graphical bubble sort algorithm.

• http://math.hws.edu/TMCM/java/xSortLab/

• There are many, many more sorting algorithms that are more efficient in that they work by making fewer comparisons. But these should give you a basic idea of sorting and how it could be accomplished.