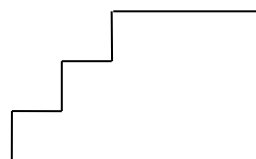


FCPS Java Packets



Unit Five – Data Processing

July 2019

Developed by Shane Torbert
edited by Marion Billington
under the direction of Gerry Berry
[Thomas Jefferson High School for Science and Technology](#)
Fairfax County Public Schools
Fairfax, Virginia

Contributing Authors

The author is grateful for additional contributions from Marion Billington, Charles Brewer, Margie Cross, Cathy Eagen, Anne Little, John Mitchell, John Myers, Steve Rose, John Totten, Ankur Shah, and Greg W. Price.

The students' supporting web site can be found at <http://academics.tjhsst.edu/compsci/>

The teacher's (free) FCPS Computer Science CD is available from Stephen Rose (srose@fcps.edu)

License Information

This work is licensed under the Creative Commons Attribution-Noncommercial-No Derivative Works 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

You are free:

- * to Share -- to copy, distribute, display, and perform the work

Under the following conditions:

- * Attribution. You must attribute the work in the manner specified by the author or licensor.
- * Noncommercial. You may not use this work for commercial purposes.
- * No Derivative Works. You may not alter, transform, or build upon this work.
- * For any reuse or distribution, you must make clear to others the license terms of this work.
- * Any of these conditions can be waived if you get permission from the copyright holder, smtorbert@fcps.edu

You are free to alter and distribute these materials within your educational institution provided that appropriate credit is given and that the nature of your changes is clearly indicated. As stipulated above, *you may not distribute altered versions of these materials to any other institution*. If you have any questions about this agreement please e-mail smtorbert@fcps.edu

Java Instruction Plan—Unit Five

Section One – Sorting Numbers

Page

Lab00: Find Min, Find Max	Five-3 to 5
Lab01: Selection Sort Algorithm	Five-6 to 9
Lab02: Modular Design	Five-10 to 11
Lab03: Scramble and Sort	Five-12 to 14

Section Two – Sorting Objects

Lab04: Sorting Weights	Five-15 to 16
Lab05: Sorting Distances	Five-17
Lab06: Sorting Strings	Five-18
Lab07: Stu's Used Car Lot	Five-19 to 20

Section Three – String Parsing

Lab08: EMail Addresses	Five-21 to 24
Lab09: Package Names	Five-25
Lab10: Musical Strings	Five-26

Section Four – Recursion

Lab11: Recursive Computations	Five-27 to 29
Lab12: Folder Listing	Five-30
Lab13: Towers of Hanoi	Five-33
Lab14: Fractal Trees	Five-34 to 35
Lab15: Binary Search	Five-36

Discussion

Index versus Value

Let's identify the features of an array declared as `double[] array = new double[100];`

	[0]	[1]	[2]	[3]	[4]		[99]
array →	138.35	748.15	20.51	800.01	390.02	...	680.41

array is a **reference**, or a pointer, to a block of 100 cells, numbered from 0 to 99.

The **index** is the location or position of the cell. The **value** is the data itself.

By using the **index** numbers and the array's subscript `[]` notation, you can access the array's **values**, for example, `array[index]`. These `array[subscript]` values can be used like any other values. In sorting and searching, we will often be comparing such subscripted values.

Looking at the array above, what is the output of `(array[0] < array[2])`? _____

Of `(array[1] > array[2])`? _____

Below are two loops. Describe in words what each one does.

```
int index = 0;
for (int k = 0; k < array.length; k++)
    index = k;
```

```
double d = 0;
for (int k = 0; k < array.length; k++)
    d = array[k];
```

This is a good place to recall Unit 4, Lab08. Notice the difference in use between `wordlist[x]` and `x`.

```
for (int x = 0; x < array.length; x++)
    if (myWord.equals(wordlist[x])
        System.out.println(wordlist[x] + " was found at " + (x+1));
```

If all you know is the value, then you cannot quickly determine its index. For one thing, there may be duplicate copies of that value and thus no unique index. Even if the value is unique, finding the index would require a loop, which could take a long time to run. Therefore, given the choice, you'd always rather know just the **index** than the **value**—because if you have the index, you can always get the value whenever you need it.

Write the code to locate the minimum value in the array `array` (above) by using only the indexes (indices):

Lab00

Find Min, Find Max

Objective

Using index numbers to search an array.

Background

In the methods we do not explicitly use the **values**, but instead we find and save the **index**. In other words, in the methods, we don't care what the min and max are, but we do care where they are. Later, in the *main*, we use the index to display the actual **values** of the max and min.

```

1 import java.io.*;           //the File class
2 import java.util.*;         //the Scanner class
3 public class Driver00
4 {
5     public static void main(String[] args) throws Exception
6     {
7         Scanner infile = new Scanner(new File("data.txt"));
8         int numitems = infile.nextInt();
9         double[] array = new double[numitems];
10        for(int k = 0; k < numitems; k++)
11            array[k] = infile.nextDouble();
12        infile.close();
13        int minPos, maxPos;
14        minPos = findMin(array);
15        maxPos = findMax(array);
16        System.out.println("Minimum value: " + "?????");
17        System.out.println("Maximum value: " + "?????");
18    }

19    private static int findMin(double[] apple)
20    {
21        /*****
22        /*
23        /* Your code goes here. */
24        /*
25        /*****
26    }
27    private static int findMax(double[] banana)
28    {
29
30
31
--

```

Lines 7-12: As in Unit 4, the data comes from a text file named *data.txt*. The first line of data (Line 8) tells the size of the array. Line 9 creates that array. Lines 10-11 store the data from the text file into the array.

Line 13: The **values** are doubles but we will find them through the **index**, which is an integer.

Line 14: call the return method and pass the data through the arguments.

Lines 16-17: Your solution should not display question marks. It should output the **value** of the max and min.

Line 19: The method is private because we don't want outsiders to use it.

Line 19: The argument to *findMin* is named *apple*. "*apple*" is a *local*

reference to "*array*". In other words, "*apple*" is an *alias* for "*array*." What it means is that *apple* and *array* point to the very same array! The same situation applies in Line 27 for *banana*.

Specification

Create *Unit5\Lab00\Driver00.java*. Enter the source code shown above. Write the methods *findMin* and *findMax*, using index numbers, not values. Then display the values on Lines 16 and 17.

Unit5\Lab00\data.txt is given to you.

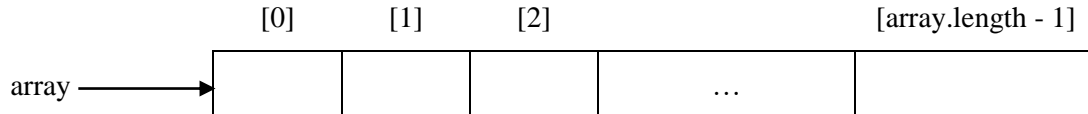
Sample Run

Unit5Lab00.jar

Exercises

Lab00

These code fragments all manipulate the array's **values** through the array's **index** numbers. The only way to figure out what's happening to the values and the indexes is to draw a picture of each array:



1) Write the contents of `circle` after this code has run:

```
double[] circle = {1.0, 10.0, 1.0, 0.0};
for (int index=0; index < circle.length; index++)
    circle[index] =
        circle[index] * circle[index] * Math.PI;
```

[0]	[1]	[2]	[3]
1.0	10.0	1.0	0.0

--	--	--	--

2) Write the contents of `array` after this code has run:

```
double[] array = {1.0, 5.0, 2.0, 3.0, 5.0};
for (int pos = array.length - 1; pos > 0; pos--)
    array[pos] = array[pos - 1];
```

[0]	[1]	[2]	[3]	[4]
1.0	5.0	2.0	3.0	5.0

--	--	--	--	--

3) Write the contents of `myArray` after this (useless) code has run:

```
int[] myArray = {3, 2, 6, 1, 9, 2, 1};
for (int i=0; i < myArray.length - 1; i++)
    if (myArray[i] > myArray[i + 1])
    {
        int temp = myArray[i];
        myArray[i] = myArray[i + 1];
        myArray[i + 1] = temp;
    }
```

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	2	6	1	9	2	1

--	--	--	--	--	--	--

temp

4) What is the output of this (useless) code?

```
int[] myList = {1, 2, 3, 4, 5};
changeArray(myList);
System.out.print("The changed list is ");
for (int i=0; i < myList.length; i++)
    System.out.println(myList[i] + " ");

public static void changeArray(int[] tempList)
{
    for (int i=0; i < tempList.length; i++)
        tempList[i] = tempList[i] + tempList[2];
}
```

[0]	[1]	[2]	[3]	[4]
1	2	3	4	5

--	--	--	--	--

Discussion

Selection Sort Algorithm

Searching and sorting are important and well-studied topics in computer science. In the last lab, you learned how to find a max or a min in an array, which is an example of searching. In Unit 4 you searched a list of words for a match. This lab shows you a sorting algorithm called the Selection Sort. Here is the algorithm:

Repeat $N - 1$ times, where N is the size of the data set:

- 1) Find the index of the maximum value in the unsorted part of the array. (Remember Lab00.)
- 2) Swap the value of the max and the value at the end of the unsorted part of the array.

Here is a table showing how the Selection Sort moves the values around. Only underlined values are being swapped. The bold values are known to be in their proper position.

<i>begin:</i>	3	1	4	1	5	<u>9</u>	2	<u>6</u>
<i>pass 1:</i>	3	1	4	1	5	<u>6</u>	<u>2</u>	9
<i>pass 2:</i>	3	1	4	1	<u>5</u>	<u>2</u>	6	9
<i>pass 3:</i>	3	1	<u>4</u>	1	<u>2</u>	5	6	9
<i>pass 4:</i>	<u>3</u>	1	2	<u>1</u>	4	5	6	9
<i>pass 5:</i>	1	1	<u>2</u>	<u>3</u>	4	5	6	9
<i>pass 6:</i>	1	<u>1</u>	<u>2</u>	3	4	5	6	9
<i>pass 7:</i>	<u>1</u>	<u>1</u>	2	3	4	5	6	9

After values are put at the end of the array, they are in sorted order, and are not considered further. It is possible that an item is swapped with itself. Notice that if we have N items, the selection sort makes $N-1$ passes in the outer loop. The inner loop at first visits and compares each element N times, but then visits one fewer element on each pass. In the last pass it visits and compares 0 times, which means the inner loop visits each element an average of $\frac{N}{2}$ times. Computer scientists express the relative **efficiency** of these nested loops by multiplying the outer passes times the inner visits. What is the efficiency of the Selection Sort? _____

In your own words write the algorithm for Selection Sort.

Lab01

Selection Sort Algorithm

Objective

Code the Selection Sort.

Background

Repeat $N - 1$ times, where N is the size of the data set:

1. Find the position of the max in the unsorted part of the array.
2. Swap the max in the unsorted part with the last item in the unsorted part.

Trace this algorithm for the data shown below.

<i>begin:</i>	2.0	3.7	9.9	8.1	8.5	7.4	1.0	6.2
<i>pass 1:</i>								
<i>pass 2:</i>								
<i>pass 3:</i>								
<i>pass 4:</i>								
<i>pass 5:</i>								
<i>pass 6:</i>								
<i>pass 7:</i>								

How many for-loops do you need? ____ Are they nested? Y / N

Assume they both start at index 0 (of course!). At what index does the outer loop stop? _____

The inner loop needs to stop before the sorted part of the array. What index is that? _____

How will you accomplish the "swap" routine? You will need a helper variable. You will need three assignment statements. Write them below, using the variables shown.

_____ array[maxPos] 9.9 array[index-1] 6.2

_____ temp

Specification

Open Unit5\Lab01\Driver01.java. An unsorted array is given to you. Sort the array using the Selection Sort algorithm. Print the sorted array. JGrasp's debugger lets you see the values moving around.

Extension

Open Unit5\Lab01\Driver01ext.java. Use a Scanner and a File to read **Lab00**'s data into an array by using the relative filepath ". . \Lab00\data.txt" (Apple computers use ". . /Lab00/data.txt"). Sort it. As you did in Unit 4, Lab06, use a PrintWriter and a FileWriter to write the sorted data to the file "output.txt".

Sample Run

Unit5Lab01.jar and Unit5Lab01ext.jar

Exercises

Lab01

Answer these questions.

```
1) private static double findMax(double[] arg)
{
    double max = arg[0];
    for(int k = 1; k < arg.length; k++)
        max = Math.max(max, arg[k]);
    return max;
}
```

- a) This findMax returns the max **value** in a given array. Why is this not helpful to a selection sort?
- b) What would we then have to do, knowing the max value, to actually make a selection sort work?
- c) Re-write the method so that it returns the index of the maximum value.

2. Here is a student's code for a selection sort to order the array from greatest to least. The student is trying to find the minimum value and swap it to the end of the array.

```
21     for(int k = 0; k < numitems; k++)
22     {
23         int minPos = 0;
24         for(int j = 1; j < numitems; j++)
25             if(array[j] < array[minPos])
26                 minPos = j;
27         double temp = array[numitems - k - 1];
28         array[numitems - k - 1] = array[minPos];
29         array[minPos] = temp;
30     }
```

- a) Which lines contain the code for finding the location of the minimum value? _____
- b) Which lines contain the code for swap? _____
- c) This selection sort method has a bug. The bug results in a logic error, not a syntax error. What is the problem and what do we need to change in order to make this sort work?

3. Write from memory the code to swap values at two given locations, i and j, in an array.

Exercises

Lab01 continued

We need to review some concepts and terminology regarding methods. Look at Unit 5, Lab00 and answer:

1. How do you recognize a *void method*? Give an example from Unit 5, Lab00.
2. How do you recognize a *return method*? (There are two indicators.) Give an example from Lab00.
3. How do you *call* a void method? Give an example from karel.
4. How do you *call* a method that returns an int? Give an example from Lab00.
5. In Lab00, what is the *type* of the argument to `findMin(double[] apple)`? _____
6. How do you *pass an argument* into `findMin(double[] apple)`?
7. In Lab00, what is the *global name* of the array outside `findMin`? _____
8. In Lab00, what is the *local name* of the array inside `findMin`? _____
9. If the argument to `findMin` is an array of doubles, why are we returning an int?
10. Does the loop in `findMin` go `<= array.length` or just `< array.length`? _____ Why?
11. Headers communicate lots of information. Explain everything communicated by this header:

```
private static void scramble(double[] array, int howManyTimes)
```

12. Explain everything communicated by this header:

```
public int compareTo(Object arg)
```

Lab02

Modular Design

Objective

Selection sort with methods.

Background

Top down programming is a way to break a complicated problem into parts. It means you break a problem into its tasks, which later become the methods. Then you break each task into sub-tasks. If necessary, you keep breaking the tasks into smaller pieces, and finally each piece is small enough to write the code directly. Then you put the pieces together.

Actually, you do this in your daily life all the time. One common daily task, for example, is to go to school. But if you think about it, that task is actually several sub-tasks. Subtask 1 is to get out of bed. Subtask 2 is to eat breakfast. Subtask 3 is to drive to school. But wait—each of those subtasks can themselves be divided into several sub-sub-tasks. For example, "to eat breakfast" is to get the milk, cereal, bowl, and spoon. But how do you get the milk? Clearly, we could turn each task into ever more detailed subtasks, until it becomes something that you "just know" how to do. At that point, you are ready to write the code.

Suppose we were writing a go-to-school program in Java. We would turn the list of tasks into the main:

```
public static void main(String[] args)
{
    Person me = new Person();      //instantiate the objects
    Bus bus = new SchoolBus();
    getOutOfBed(me);                //call the class methods
    eatBreakfast(me);
    driveToSchool(bus, me);
}
```

This is a Level 1 design. The next step is to write Level 2, the subtasks. For `eatBreakfast` we have:

```
private static void eatBreakfast(Person p)
{
    p.getMilk(refrigerator);        //call Person's instance methods
    p.getCereal(cabinet);
    p.getBowl(cabinet2);
    //etc.
}
```

Computer scientists follow this divide-and-conquer strategy all the time. Let's use the top-down design strategy to solve the problem of sorting an array of numbers into order.

Specification

Create Unit5\Lab02\Driver02.java. Read a given number of doubles from a text file. Use a Selection Sort to sort them into ascending order. Output the sorted array into another text file.

The specification describes three major tasks. What are they? These tasks become the method calls in the main. After that, you turn your attention to the subtasks. What are the subtasks for, e.g. `sort` ?

Sample Run

Unit5Lab02.jar

Exercises

Lab02

Answer each question about Lab02.

- 1) Write the header of the `swap` method.
- 2) What are the *types* of the three *arguments* in the *swap method*?
- 3) Give three reasons why programmers break up a program into methods.

Do a Level 1 top-down design for the problems below. That is, write the main method only.

- 3) Read in an unknown number of integers from a text file and print out all the numbers that are strictly greater than the average value. The three tasks are _____, _____, and _____.

```
public static void main(String[] args) throws Exception
{
    int[] array = _____("input.txt"); //reads in data from file

    double avg = _____( _____); //finds the _____

    _____(array, avg); //displays those values in array that are
                          // greater than avg
}
```

- 4) Write the method calls in `main` to: read in a month's worth of daily high and low temperatures, find the maximum and the minimum, and output those two numbers. *Do not write the methods themselves.*

- 5) Write the method calls in `main` to: read in an arithmetic expression, e.g. $2+3*4$, check to make sure that it is well-formed, and if the check is good, then evaluate and output the answer. If the check is not good, output an error message "arithmetic expression parse error." *Do not write the methods themselves.*

Discussion Bubble Sort

Imagine air bubbles bubbling up to the top of a water cooler. The bubble sort makes the smaller items "bubble up" to the top of the array. Conversely, the larger items "bubble down" to the bottom of the array.

We can start from the end (or the bottom) of the array, which means that `k` begins at `array.length-1`. Compare elements side-by-side (that is, compare `array[k-1]` and `array[k]`) and if the second one is smaller, swap them. Compare the next pair and swap if necessary. Keep going. Then make another pass.

	<i>begin:</i>	<i>after pass 1:</i>	<i>after pass 2:</i>	<i>after pass 3:</i>	<i>after pass 4:</i>	<i>after pass 5:</i>	<i>after pass 6:</i>	<i>after pass 7:</i>
	6	1	1	1	1	1	1	1
	2	6	1	1	1	1	1	1
	9	2	6	2	2	2	2	2
	5	9	2	6	3	3	3	3
	1	5	9	3	6	4	4	4
	4	1	5	9	4	6	5	5
array[k-1]	1	4	3	5	9	5	6	6
array[k]	3	3	4	4	5	9	9	9

In the Bubble Sort the data moves around by only one cell at a time, while in the Selection Sort the data jumps right to its final place. In addition, the Bubble Sort checks every item on every loop, while the inner loop of the Selection Sort checks one fewer item on each loop. For both these reasons, the Bubble Sort is slower than the Selection Sort.

How many for-loops does the Bubble Sort require? _____ Are they nested? Y / N

For N items, the outer loop repeats $N-1$ times. The inner loop makes N comparisons. What is the efficiency of the Bubble Sort? _____

In your own words, write the algorithm for the Bubble Sort.

Discussion Insertion Sort

How do you normally sort a hand of cards? Most people use the Insertion Sort. They get a couple of cards sorted, then put the next card in its proper place, and the next, and so on.

Here is a more formal algorithm: loop forward across the data. For each new item of data (the underlined item), take the item out, compare to its neighbor, and if necessary slide the neighbor over. Compare to the next neighbor and slide if necessary. Eventually the item will be less than a neighbor, meaning that it has reached the correct place, so insert it.

<i>begin:</i>	3	<u>1</u>	4	1	5	9	2	6
<i>pass 1:</i>	1	3	<u>4</u>	1	5	9	2	6
<i>pass 2:</i>	1	3	4	<u>1</u>	5	9	2	6
<i>pass 3:</i>	1	1	3	4	<u>5</u>	9	2	6
<i>pass 4:</i>	1	1	3	4	5	<u>9</u>	2	6
<i>pass 5:</i>	1	1	3	4	5	9	<u>2</u>	6
<i>pass 6:</i>	1	1	2	3	4	5	9	<u>6</u>
<i>pass 7:</i>	1	1	2	3	4	5	6	9

The Insertion Sort orders all the data it has processed. Once it has visited all the data, then the entire set is sorted. Processing the data takes 2 nested loops, the outer which loops $N-1$ times and the inner which loops $\frac{N}{2}$ times. What is the efficiency of the Insertion Sort? _____

What was the efficiency of the Bubble Sort? _____

What was the efficiency of the Selection Sort? _____

Since the efficiency of these three sorts are all of the same order of magnitude, computer scientists classify them as **quadratic sorts**. In a quadratic sort, if we double the number of items, we quadruple the work (or time) to sort them. Fill in this table, showing how the time increases as the square of the number N of items.

N	milliseconds
500	3
1000	12
2000	
4000	
8000	

In your own words, write the algorithm for the Insertion Sort.

Lab03

Scramble and Sort

Objective

Scramble and sort an array of integers.

Background

The task is to print an ordered array, scramble it, sort it into order, and print the ordered array. The main method is planned out in good top-down style:

```
4  public class Driver03
5  {
6      public static void main(String[] args)
7      {
8          int[] array = {100, 101, 102, 103, 104, 105, 106, 107, 108, 109};
9          print(array);
10         scramble(array);
11         print(array);
12         sort(array);
13         print(array);
14     }
```

Notice the main is just a list of methods with arguments. Of course, all the real work goes on in the methods.

It is important to realize that the *call* to the method *passes data* through its *argument(s)*. For example, the call in main at Line 10 to `scramble(array)` instructs the computer to run the method `scramble` using the data in the array.

The method `scramble` somehow has to receive the data. We specify that the method will be receiving data by coding its header something like `scramble(int[] grape)` .

Java checks to make sure that the type of the arguments always match up. That is, if you wrote the method with the header `scramble(double[] grape)`, you would get a compile error, something like "cannot apply `int[]` to `double[]`". You will learn in the next several labs that whenever you write a method, you must make sure that the data types in the argument list match the data types in the calling method.

Specification

Create `Unit5\Lab03\Driver03.java` as described above. as given above. Use the Selection sort from Lab02, including its helper methods, but change it so that it handles integers, not doubles. Then write the methods `scramble` and `print`. You will need to devise your own algorithm to scramble an array of numbers.

Extensions

- 1) Suppose we wanted to sort in reverse order, that is, from largest to smallest. What changes would we have to make in the code? Do so.
- 2) Implement the Bubble Sort.
- 3) Implement the Insertion Sort.

Sample Run

`Unit5Lab03.jar`

Discussion

The Comparable<E> Interface

Objective

Sorting objects, Java-style.

Background

Since we have just sorted numbers, we turn our attention to sorting objects. We will use the same Selection Sort algorithm, but we must specify what we mean when we say that one object is "bigger than" another. The standard Java way to do so is to write a `compareTo` method. Put more abstractly, we say that the class *implements the Comparable<E> interface*. (The *E* is a placeholder for a class. The *E* is used only in the API. Look at Line 1 on the next page for an example where *<E>* is replaced by a concrete class *<Weight>*.)

The Comparable<E> interface contains a single abstract method `compareTo`. All the work goes on in the concrete subclass's `compareTo`, which you will implement in Labs04, 05, and 06. The method `compareTo` takes one argument of type E and by convention returns a number. Your job as the programmer is to return the correct number. By convention, the returned number indicates either `<`, `==`, or `>` as follows:

if `x.compareTo(y)` returns a **negative** number, it is equivalent to `x < y`.
 if `x.compareTo(y)` returns **zero**, it is equivalent to `x == y`.
 if `x.compareTo(y)` returns a **positive** number, it is equivalent to `x > y`.

Often the number returned is `-1`, `0`, or `1`, but this is not guaranteed. It could be any negative number or any positive number. In Lab05 and Lab06, we will return other numbers.

To repeat, Java makes us do all this because objects cannot use the built-in arithmetic comparative operators. The conditional `if(x < y)` is undefined if `x` and `y` are objects. `Math.max` is also undefined for objects. Java's solution is to use `compareTo` to compare objects `x` and `y`:

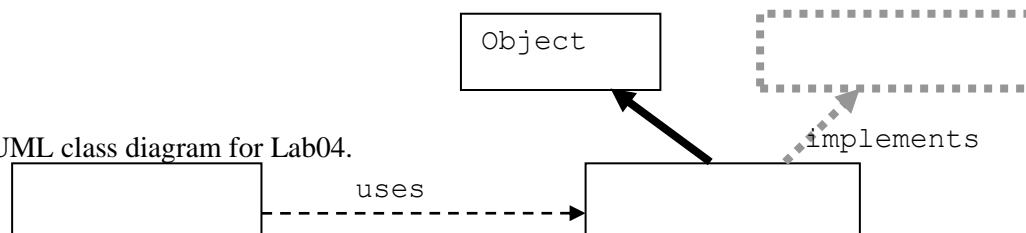
```
if( x.compareTo(y) < 0 )
    doSomething();

if( x.compareTo(y) == 0 )
    doSomething();

if( x.compareTo(y) > 0 )
    doSomething();
```

But not all objects can be compared. Only those objects that *implement the Comparable<E> interface* are guaranteed to have a `compareTo` method. 1) How does Java check for this requirement?

2) Fill in the UML class diagram for Lab04.



3) How does the method `findMax(Comparable[] array)` illustrate the power of polymorphism?

Lab04 Sorting Weights

```

1 public class Weight
    implements Comparable<Weight>
2 {
3     private int myPounds, myOunces;
4     public Weight()
5     {
6         myPounds = myOunces = 0;
7     }
8     public Weight(int x, int y)
9     {
10        myPounds = x;
11        myOunces = y;
12    }
13    public int getPounds()
14    {
15        return myPounds;
16    }
17    public int getOunces()
18    {
19        return myOunces;
20    }
21    public void setPounds(int x)
22    {
23        myPounds = x;
24    }
25    public void setOunces(int x)
26    {
27        myOunces = x;
28    }
29    public int compareTo(Weight w)
30    {
31        if(myPounds < w.getPounds())
32            return -1;
33        if(myPounds > w.getPounds())
34            return 1;
35        if(myOunces < w.myOunces)
36            return -1;
37        if(myOunces > w.getOunces())
38            return 1;
39        return 0;
40    }
41    public String toString()
42    {
43        return myPounds + " lbs. " +
44            myOunces + " oz.";
45    }

```

Sample Run
Unit5Lab04.jar

The Weight class in Lab04 is a typical example of a class. You should be able to identify the data fields, the constructors, the accessors, the modifiers, and the instance methods. Mark each of these with a letter "F", "C", "A", "M", or "I".
Line 1 says "implements Comparable." What does that mean?

Why does line 1 have "<Weight>"?
Why does line 31 have "w.getPounds()"?
When does line 32 return "-1"?

The w.myOunces on Line 35 works!
private means private at the class level, not at the object level.

When does line 38 return "1"?
When does line 39 return "0"?
What does this command return?
array[x].toString();

Specification

Use Unit5\Lab04\data.txt. Run MakeDataFile to create a file of pounds and ounces. The partial file at right has weights for 57 objects. The first object weighs 64 lbs 13 oz., the second is 15 lbs 12 oz., etc.

57
64
13
15
12
67
3
10

Create Unit5\Lab04\Weight.java.
Copy the code. Study compareTo carefully!

Complete Unit5\Lab04\Driver04.java.
You will read the file for pounds and ounces, instantiate a Weight object, and assign it to its place in the array. Also, sort, swap, and findMax all need to work with arrays of type Comparable. (Why?)

Partial Output

0 lbs. 4 oz.
3 lbs. 5 oz.
4 lbs. 0 oz.
7 lbs. 1 oz.

Lab05

Sorting Distances

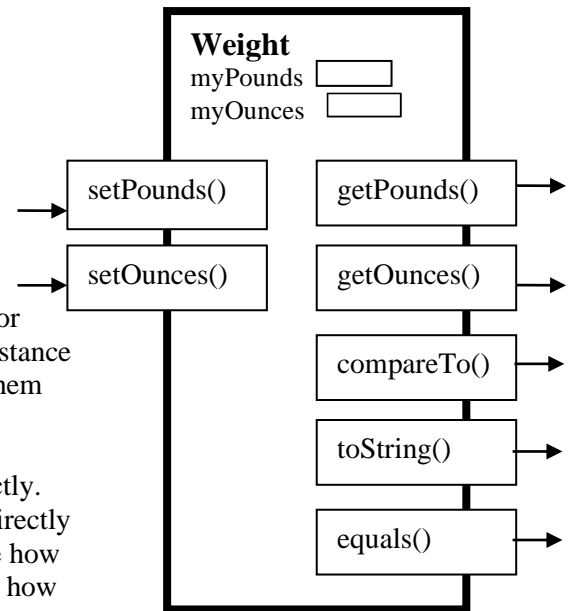
Objective

Write a Distance class that implements Comparable<E>.

Background

A diagram of the Weight class is shown to the right. Note the private data fields are totally inside the class. (Why?) The modifier methods show arrows going into the class. The accessor methods show arrows leaving the class. Since the three other instance methods are public, they extend outside the class. Since all of them return a result, they all have arrows leaving the class.

An object's private data should not, in general, be accessed directly. Rather, you should use accessor and modifier methods to deal directly with the private data. Outside methods should not know or care how the private data is actually stored. Only the class needs to know how to store its own data. Encapsulation!



Here is a different way to write Weight's compareTo method. This version of compareTo does not return the signals -1, 0, or 1, but will return the actual difference in ounces:

```
public int compareTo(Weight w)
{
    int myTotal = myPounds * 16 + myOunces; //access my data directly
    int wTotal = w.getPounds() * 16 + w.getOunces(); //use w's get() methods
    return myTotal - wTotal; //return the difference in ounces
}
```

When implementing compareTo it is a good idea also to override the equals method originally defined in Object<E>. This guarantees that your compareTo and your equals method will not give different answers.

```
public boolean equals(Weight arg)
{
    return compareTo(arg) == 0;
}
```

Specification

Copy MakeDataFile from Lab04. Modify it so that it generates random feet and inches. Run it to produce the data.txt file. The partial file at the right has data to make 28 Distance objects. The first object will be 10 feet 9 inches, the second is 15 feet 2 inches, etc.

Write Unit5\Lab05\Distance.java. Use Weight as an example. Write compareTo and equals methods modeled on those given above.

Write Unit5\Lab05\Driver05.java. Use Driver04 as an example.

Extension

Suppose we wanted to sort the distance objects in reverse order, that is, from largest to smallest. What changes would we have to make in the code? Do so.

```
28
10
9
15
2
31
5
6
11
.
.
.
```

Sample Run After you have made data.txt, run Unit5Lab05.jar

Lab06

Sorting Strings

Objective

To sort strings.

Background

In Java, strings are represented by the `String` class which implements the `Comparable<E>` interface. (Wasn't that nice of the Java developers?) As with single characters, a string is "less than" another if it comes before or earlier, according to the alphabet. What about capital letters and characters that are not in the alphabet? To handle these, programmers have come up with several possible orderings. Java uses an ordering called Unicode (www.unicode.org), which is a superset of the older ASCII ordering. In Unicode, all capital letters come before all lowercase letters. Capital 'A' happens to be number 65 and lowercase 'a' happens to be number 97. Therefore, `"A".equals("a")` returns `false`. Also, `"A".compareTo("a")` returns a negative number, which in this case happens to be -32.

Did you write a separate sort for `Weight` and `Distance`, and are you planning to write a new one for `String`? If so, you wasted your time. Since all three of these classes implement the `Comparable` interface, i.e., all three are (also) objects of type `Comparable`, we don't need three sorts, we only need one sort, for all `Comparables`:

```
public static void sort(Comparable[] array)
```

In both the `sort` and the `findMax` methods, the general superclass reference to `Comparable[]` points to a specific subclass object, either `Weight`, `Distance`, or `String`. The code that actually gets executed is in the subclass object, either `Weight`, `Distance`, or `String` as appropriate. Writing such general methods takes advantage of the power of polymorphism.

In Unit4, Lab08, when you searched for a word in a dictionary, you were looking for "equal" strings. If you remember, you could not use the `==` symbol. Instead, you had to use the `dot-equals` method for Objects:

```
if( s.equals(wordlist[x]) )
```

For the same reasons, in this lab you cannot use the less-than or greater-than symbols, `<` or `>`, to sort the list of words. You must use `dot-compareTo` because `Strings` are `Comparables`, like this:

```
if( array[j].compareTo( array[maxPos] ) > 0 )
```

Specification

Use Unit5\Lab06\data.txt. A file of 222 words is provided, beginning like this:

Write Unit5\Lab06\Driver06.java. Read the file of words into the array. (Java doesn't have many exceptions to its rules, but here is one: even though a string is an object, it doesn't need to use the `new` operator.) Then sort the words, using the general Selection Sort. Last, print the sorted array to "output.txt." Why are words with capital letters first in the list?

Sample Run

Unit5Lab06.jar

```
222
album
vampire
bomb
Triangle
pendulum
record
Film
Tiger
.
.
.
```

Lab07 Stu's Used Car Lot

Objective

To implement several `Comparator<E>` interfaces.

Background

What happens if you want to sort the same set of data based on differing criteria? For instance, you may want to sort objects by name sometimes and by size other times. The `Comparable` interface cannot do this because `compareTo` defines a single ordering for the class. The solution is to instantiate several `Comparator` objects whose `compare` method accepts two objects as arguments and that defines what it means to compare those two objects. The same Selection Sort code, when used with different comparator objects, will produce different ordered lists. Polymorphic behavior!

Each comparator, because it implements the `Comparator<E>` interface, must define a single method `compare` that accepts two objects as arguments. Here is an example `Comparator` that compares the sizes of two `Widgets`:

```
public class BySize implements Comparator<Widget>
{
    public int compare(Widget arg1, Widget arg2)
    {
        return arg1.getSize() - arg2.getSize(); //a negative number
                                                //means arg1 is smaller
    }
}
```

Recall the standard Selection Sort algorithm: find max and swap, find max and swap. In this case, `findMax` uses the `BySize` object's `compare` method to find the `Widget` with the largest size. `swap` then swaps the `Widgets`. At the end, we will have ordered the `Widget` objects by their sizes. A different comparator object, with a different `compare` method, would order the `Widget` objects differently. For example, a `ByName` object would order the `Widget` objects alphabetically and a `ByPrice` object would order the `Widget` objects by price. In this lab, we will be ordering `Salespersons` in different ways.

Specification

Use `Unit5\Lab07\data.txt`. It is a file of 10 names, each followed by two integers, as shown.

Create `Unit5\Lab07\Salesperson.java`. Write the constructors, accessors, and modifiers. In contrast to the previous labs, `Salesperson` is not a `Comparable`.

Open `Unit5\Lab07\Driver07.java`. Study this code. You have seven methods to complete, first `input`, `display`, `findMin`, and `swap`, then `add`, `search`, and `save`. Notice the clever way to produce the long string message used by `JOptionPane.showInputDialog`. Notice that `save` saves the file using a `PrintWriter` object and a try-catch block.

Open `Unit5\Lab07\ByCars.java`. This is an example `Comparator`. Sorting by cars gives the output as shown:

Write `ByTrucks.java` and `ByTotalSales.java`.

Write `ByName.java`. Hint: a `Salesperson`'s name is a `String`, which is a `Comparable`.

Sample Run `Unit5Lab07.jar`

Name	Cars	Trucks	Total
-----	-----	-----	-----
Choi	14	1	15
George	13	12	25
Adams	6	8	14
Brown	4	8	12
Miller	5	4	9
Jones	3	6	9
Smith	3	4	7
Nguyen	3	3	6
Thomas	2	20	22
Franks	3	1	4

```
10
Adams
6
8
Brown
4
8
Choi
14
1
Franks
3
1
George
13
12
.
.
.
```

Lab07

Exercises

1. Labs04, 05, and 06 all used the same `sort`, `findMax`, and `swap` methods, although the data being sorted were different objects. How can the same methods work with three different types of objects?

2. Lab06 on sorting strings contained the line: `if(array[j].compareTo(array[maxPos]) > 0)`

`j` is a(n) _____

`array[j]` refers to a(n) _____ that is-a _____ and a(n) _____

`compareTo()` is a(n) _____ of class _____ and of interface _____

`compareTo()` returns a(n) _____

the relational operator `>` returns a(n) _____

3. Lab07 on Salespersons contained the line: `if(c.compare(array[j], array[minPos]) < 0)`

`minPos` is a(n) _____

`array[minPos]` is-a _____ and a(n) _____

if we sort by cars, `c` is a(n) _____ of class _____ which is-a _____

`compare()` is a(n) _____ required by the _____ interface

`compare()` returns a(n) _____

the relational operator `<` returns a(n) _____

4. One student, trying to implement a Selection Sort, wrote this line in the `findMin` method:

```
if( array[j].compareTo( array[minPos] ) == -1 )
```

Will this always work? _____ Why or why not?

5. One student, trying to implement a Selection Sort, wrote:

```
if(array[j] > array[maxPos])
```

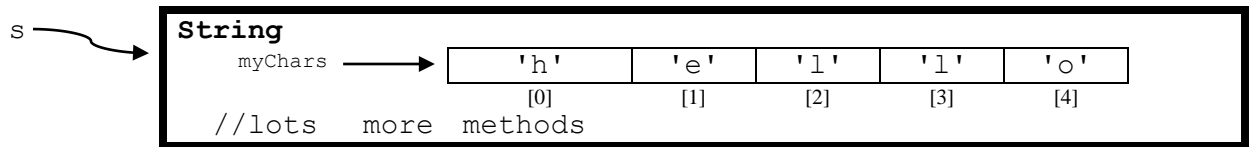
Will this always work? _____ Why or why not?

6. Write all the code to read an array of doubles from a disk file into an array. The file is named "thomas.txt" and the first line of the file is an integer telling how many doubles are in the file.

Discussion

String Methods

As we said in Unit 4, strings are sequences of characters and characters are single letters, symbols, or punctuation marks. Java has a primitive type `char` that stores single characters. Java denotes a `char` value with single quotation marks, such as `'A'`, `'>'`, or `'.'`. Every `String` object has a private field holding an array of characters. You should picture `String s` as a reference pointing to a `String` object that stores an array of characters, like this:



The reason programmers go to the trouble of making classes is not only to store data but also to provide methods to work on that data. Java's `String` class has many methods which are all, of course, listed in the `String` API. The effects of calling some commonly-used methods are shown below:

1. <code>String abc = "abcdefghijklmnopqrstuvwxyz";</code>	abc	→	"abcdefghijklmnopqrstuvwxyz"
2. <code>int len = abc.length();</code>	len		26
3. <code>String str1 = abc.substring(20);</code>	str1	→	"uvwxyz"
4. <code>String str2 = abc.substring(20, 21);</code>	str2	→	"u"
//from 20 up to, but not including, 21			
5. <code>char ch = abc.charAt(20);</code>	ch		'u'
6. <code>int pos1 = abc.indexOf('f');</code>	pos1		5
7. <code>int pos2 = abc.indexOf(ch);</code>	pos2		20
8. <code>int pos3 = abc.indexOf('F');</code>	pos3		-1 (signal for "not found")
9. <code>int comp1 = str1.compareTo(str2);</code>	comp1		some positive number, because "uvwxyz" comes after "u"
10. <code>int comp2 = abc.compareTo("abcdefghijklmnopqrstuvwxyz");</code>	comp2		0
11. <code>boolean b = "cat".equals("dog");</code>	b		
12. <code>boolean b2 = "ABCD".equalsIgnoreCase("abcd");</code>	b2		
13. <code>String s = "123.456.789";</code>	s	→	
14. <code>int pos = s.indexOf(".");</code>	pos		
15. <code>int a = Integer.parseInt(s.substring(0, pos));</code>	a		
16. <code>s = s.substring(pos + 1);</code> //Notice that <i>s</i> is <i>reassigned</i> as a shorter string.	b		
17. <code>pos = s.indexOf(".");</code>	c		
18. <code>int b = Integer.parseInt(s.substring(0, pos));</code>			
19. <code>s = s.substring(pos + 1);</code> //reassign a shorter <i>s</i>			
20. <code>int c = Integer.parseInt(s);</code>			

Lab08

EMail Address

Objective

Parsing a string.

Background

The shell for the Email class is shown here to the right. The address argument to the constructor is a complete email address in the form:

```
username@hostname.extension
```

Notice the three data fields in this class.

Your constructor must break that one string down (or "parse" it) into its component parts. Assign just the username to the field myUserName, just the host name to the field myHostName, and just the extension to the field myExtension.

Look at the String API. The string methods indexOf and substring may interest you.

Your toString should take those three components and return the original complete email address.

Specification

Open Unit5\Lab08\EMail.java. Complete the constructor and the toString method.

Open Unit5\Lab08\Driver08.java.

Extensions

Turn Lab08 into a duplicate of Lab07, Stu's Used Car Lot. Read a list of email addresses from a file, make an array of EMail objects, let the user choose different ways of sorting them, and display the sorted addresses to the screen. Finally, let the user change (update) the email addresses and save the new list back to the file.

Sample Run

Unit5Lab08.jar

//Name _____ Date _____

```
public class EMail
{
    private String myUserName;
    private String myHostName;
    private String myExtension;
    public EMail(String address)
    {
        /*****
        /*
        /* Your code goes here. */
        /*
        *****/
    }
    public String getUser_name()
    {
        return myUserName;
    }
    public String getHostName()
    {
        return myHostName;
    }
    public String getExtension()
    {
        return myExtension;
    }
    public String toString()
    {
        /*****
        /*
        /* Your code goes here. */
        /*
        *****/
    }
}
```

Exercises

Lab08

Indicate the output of the following code:

```

5   String str = "The rain in Spain falls.";
6   System.out.println( str.length() );           // _____
7   System.out.println( str.indexOf(' ') );        // _____
8   System.out.println( str.indexOf("rain") );     // _____
9   System.out.println( str.indexOf("in") );       // _____
10  System.out.println( str.indexOf('R') );        // _____
11  System.out.println( str.lastIndexOf(' ') );    // _____
12  System.out.println( str.charAt(str.indexOf('S')) ); // _____
13  System.out.println( str.charAt(str.indexOf(' ')+1) ); // _____
14  System.out.println( str.substring(str.indexOf(' ')+1)); // _____
15  System.out.println( str.substring(
16      str.indexOf(' ') + 1, str.indexOf(' ') + 5) ); // _____
17  System.out.println( str.substring(str.indexOf('S')) ); // _____
18  System.out.println(str.substring(str.lastIndexOf('S'))); // _____
19  System.out.println( str.contains("r") );       // _____
20  System.out.println( str.contains("R") );       // _____
21  int i = str.compareTo( "THE" );                // _____
22  boolean b = str.compareTo( str.substring( 0, 3 ) ) < 0; // _____
23  boolean b2 = str.substring(0, str.indexOf(" ")).compareTo(str)< 0; // _____
24  System.out.println( str );                     // _____
25  int count = 0;
26  while( str.contains(" ") )
27  {
28      str = str.substring( str.indexOf(' ') + 1 );
29      System.out.println(str);                    // _____
                                                    // _____
                                                    // _____
                                                    // _____
30      count++;
31  }
32  System.out.println(count + " words.");          // _____

```

-
1. To change a char `ch` into a String, use _____. Where have you seen this before?
 2. To change a single-character String `s` into a char, use _____.

Discussion Characters

As you know, strings are arrays of characters. Characters are single letters, digits, or keyboard symbols. You can, of course, manipulate single characters using the methods in the Character class. (Look at the current Java API, especially at the boolean methods.) Characters are also mapped to integers. In the ASCII code, 'A' is 65 (or 01000001 or 41₁₆), 'a' is 97, 'B' is 66, and 'z' is 122. A space, ' ', is number 32. Here is a simple way to print the upper and lower case letters and their corresponding integers. Notice the casting from `int` to `char`.

```
for(int n = 65; n <= 122; n++)
    System.out.println( n + "=" + (char)n );
```

The older ASCII system (<http://www.asciitable.com/>) has become a subset of Unicode (<http://unicode-table.com/en/>), which includes letters from languages other than English. The code below also prints the upper and lower case letters (plus a few more), this time casting characters to integers:

```
for(char ch = 'A'; ch <= 'z'; ch++)
    System.out.println( (int)ch + "=" + ch );
```

Characters are primitives and can often be manipulated like integers. These three comparisons accomplish much the same thing: ('A'<'a'), (int('A')<int('a')), and (char(65)<char(97)). When you compare primitives for equality, you use `==`, not `.equals()` or `.compareTo()`.

For any string `str`, `str.charAt(index)` returns a character. Since you know that Strings are stored in an underlying array of chars, you might think that `str[index]` would work. However, it does not. The technical explanation is that Strings are *immutable*, meaning that it is impossible to assign directly a character to `str[index]`, i.e., `str[index] = 'A';` does not work. Because strings are immutable, Java makes new strings every time you concatenate the character `ch` with an existing string `str`. There are two ways to do this, `str = str + ch;` and `str = str.concat(ch)`. Note that you must reassign the `str` reference.

Cyptography is an important subject in computer science. Supposedly Julius Caesar used the following cipher (in Latin, without Java) to communicate with his legati, praetores, or quaestores. The Caesar Cipher algorithm is: for each character, find the alphabetic position, add a constant, modulus the result by (the number of letters in the alphabet plus 1), and write the new letter.

```
public final String alphabet = "_ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Try it out: if the English alphabet is numbered as shown, the constant is 9, and the plaintext message is ATTACK_AT_DAWN, what is the encrypted message? _ _ _ _ _

Write a method that returns the encrypted message. The `k` is the constant, in this case, 9.

```
public static String encrypt(String alphabet, String plainText, int k)
{
```

Lab09 Package Names

Objective

String manipulation.

Background

A Java application is made up of many interacting classes. It is obviously important that every class have a unique name. In past labs, when we wrote

```
String s = "hello";
```

we were actually abbreviating the unique, fully-qualified package name for `String`, which is:

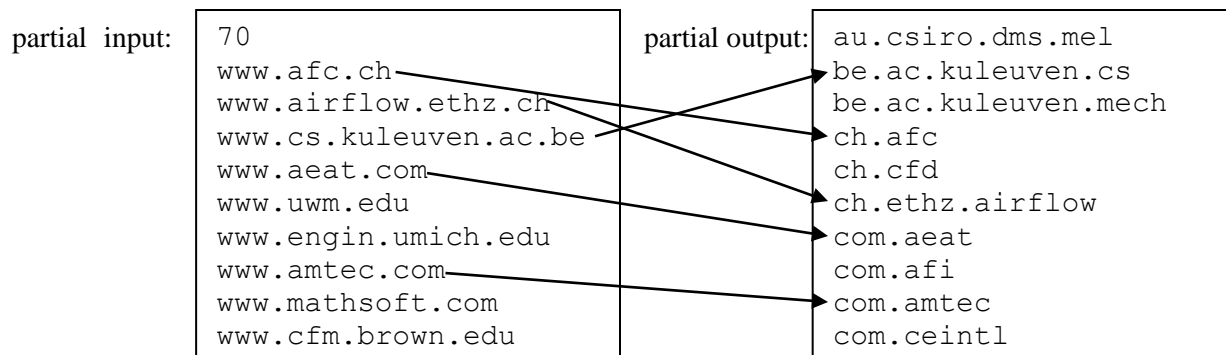
```
java.lang.String s = "hello";
```

If you or anyone else decided to write another class named `String`, you would have to put it in its own, unique package name, for example, `edu.fcps.student.String`. If you import both your package and `java.lang.String`, then Java will have a way to determine which `String` you mean to use.

How can everyone make sure that all package names are unique? Someone noticed that web addresses are unique because they are assigned by a central office. Thus, package names will also be unique if everyone follows the algorithm: take the web address, drop the "www.", and reverse the parts.

In this lab you will read in a list of web addresses, convert them to package names, sort the package names, and print them. What are the four tasks in `main`?

In this lab, we will sort the names, but we will not use our old friends `sort`, `findMax`, and `swap`. In that case, how can we sort? We will take advantage of a method in Java's `Arrays` class. Look at the Java API.



Specification

Use `Unit5\Lab09\data.txt`. This file contains a list of web addresses. Notice that web addresses may have 3, 4, or 5 dots. To handle this uncertainty, you'll need a loop. Which kind of loop is appropriate?

Open `Unit5\Lab09\Driver09.java`. The methods to input, sort (a call to Java's `Arrays.sort`), and output have been completed for you. You are to finish `convert`.

Sample Run

`Unit5Lab09.jar`

Lab10 Musical Strings

Objective

Reading and parsing lines of a text file.

Background

Up to now, the text file has always contained one item (either a word or a number) per line. You used `next` or `nextInt` to read that item (up to white space) and return it.

This time, we will have mixed data, both words and numbers—this can be tricky to read correctly. Each line will provide data for three different fields in the Song object. We have to read the entire line as a single string and then parse the string into its parts, either words or numbers. Scanner's method `nextLine` is the method we want. The file at the right is "dylan.txt". Study it.

```
7
3:45 Blowin' In The Wind
4:12 The Times They Are A-Changin'
2:57 It Ain't Me, Babe
4:08 Like A Rolling Stone
5:14 Mr. Tambourine Man
2:58 Subterranean Homesick Blues
3:00 Positively 4th Street
```

The first line, which contains an integer, tells how many songs are to be processed. Here is good advice: If you have to use `nextLine` in one part of the program, use `nextLine` all the way through. It is better not to mix `infile.nextInt` and `infile.nextLine`, although the lab can be done that way.

Read each subsequent line and process it into two integers and a string that represent the song's minutes, seconds, and title. The number of minutes might be either one or two digits (or three digits, ahh!). Therefore, you will have to search for the colon to figure out where the minutes end. The number of seconds will always be two digits. The title always starts one space after the seconds end. Clearly the format of the input file is very specific.

The output is the total time of all the songs in the list and the name of the longest song, followed by its time:

```
Total Time: 26' 14"
Longest Song: Mr. Tambourine Man (5' 14")
```

Do some top-down planning. What will be the methods in `main`?

As is typical in object-oriented programming, we decide to make an object that accomplishes some of our tasks. In this case, we will be working with Song objects. What private data should a Song object know? What methods would be useful in a Song object? Would it be useful if the Song class implemented the Comparable interface?

Specification

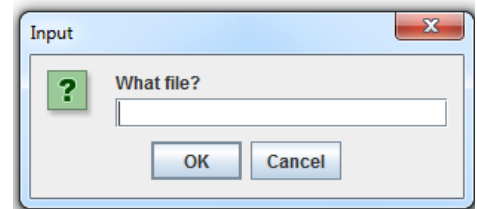
Open `Unit5\Lab10\Driver10.java`. Complete the methods. In the `input` method, use a `JOptionPane` to prompt the user to enter the file name.

Open `Unit5\Lab10\Song.java`. Complete the class.

Three data files have been provided: `dylan.txt`, `tchaikovsky.txt`, and `beethoven.txt`

Sample Run

`Unit5Lab10.jar`



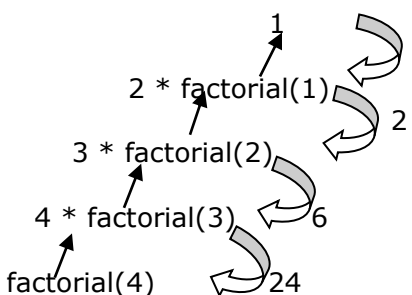
Discussion Recursion and Computations

In general, a *recursive structure* has substructures that have the same form as the original structure. In computer science, *recursion* is a technique in which a method calls a "one-step smaller" copy of itself. A recursive method calls itself, but recursion is not a loop! Instead, the recursive calls stack up until they reach the *base case*, which is where the recursion stops. Then the stacked up calls *return* to the previous calls, in reverse order, until the original method ends.

Calculating factorials is an example of recursion. Recall that $4!$ means $4 \times 3 \times 2 \times 1 = 24$. In math, the recursive definition is $n! = n * (n - 1)!$. Notice that $n!$ is defined in terms of $(n-1)!$

```
public static long factorial(int n)
{
    if (n==1)
        return 1;
    else
        return n * factorial(n-1);
}

System.out.println(factorial(4));
```



- Notice the *if-else* construction. All recursive methods have it, sometimes implicitly.
- Notice the call, at the end, to a "one-step smaller" copy of itself. Learn to look for it.
- Notice the *base case*, in the *if*-statement, which is where the recursion stops.
- The recursive calls stack up on top of one another. Until you get good at "seeing" the recursive structure, learn to trace the calls up and down the levels. The trace above shows how $4!$ returns 24.

Another common example of recursion is generating the Fibonacci sequence, which is defined by:

- The first two numbers are both one.
- Each subsequent number is equal to the sum of the two numbers directly preceding it.

This gives us the familiar sequence:

	1	1	2	3	5	8	13			
0	1	2	3	4	5	6	7	8	9	10

- Why is the Fibonacci function naturally recursive?
- What are the base cases?
- What are the recursive calls?

You should understand that in practice, calculating the Fibonacci sequence recursively is terribly inefficient because we end up calculating the first two values way more than we need to. In fact, it would be more efficient to use *iteration* and an array:

```
int[] array = new int[n];
array[1] = array[2] = 1;
for (int k = 3; k < n; k++)
    array[k] = array[k - 1] + array[k - 2];
```

Nevertheless, in other problem-spaces, recursion is faster and more convenient than iteration. Recursion is an extremely powerful tool for dealing with problems that are branching, tree-like, or nested.

Lab11

Recursive Computations

Objective

To use recursion in a computation.

Background

It's actually quite easy to make up a function that is recursive. Let's define an arbitrary function f such that $f(1) = 7$ and $f(n) = f(n - 1) + 9$. Then f generates a table:

n	1	2	3	4	5	6	7
f(n)	7	16	25	34	43	52	61

Our function f could of course be calculated explicitly as:

```
public static int f(int n)
{
    return 7 + (n - 1) * 9;
}
```

The recursive version of function f is given by:

```
public static int f(int n)
{
    if (n == 1)
        return 7;
    else
        return 9 + f(n - 1);
}
```

As an exercise, trace $f(5)$. Draw the arrows up and down and end up with 43.

Powers: How do you calculate powers, for example 2^3 ? $2^3 = 2 * 2^2$. But $2^2 = 2 * 2^1$. Finally, $2^1 = 2$. This leads to a recursive definition for powers:

$$\text{base}^{\text{exp}} = \begin{cases} \text{base} & \text{if exp} = 1 \\ \text{base} * \text{base}^{\text{exp}-1} & \text{if exp} > 1 \end{cases}$$

Trace the recursion
for power (3, 4)

GCD: Euclid's gcd algorithm uses recursion to find the Greatest Common Divisor of two integers. (The gcd is the largest integer that divides evenly into both integers.) For example, gcd(12,8) returns 4 and gcd(5,3) returns 1. Here is Euclid's algorithm to calculate the gcd of two integers x and y :

```
gcd(x,y) → x if y == 0
else
gcd(x,y) → gcd(y, remainder of x/y);
```

Trace the recursion
for gcd (12, 8)

Trace the recursion
for gcd (5, 3)

Specification

Open Unit5\Lab11\Driver11.java. Write methods to calculate 1) powers, 2) factorials, 3) Fibonacci numbers, and 4) GCD.

Extension

Other good recursive projects include 5) Superprimes, 6) white-black erase, 7) a palindrome checker, 8) the Knight's Tour, and 9) the 8-Queens problem.

Sample Run

Unit5Lab11.jar

Exercises

Lab11

1. Evaluate $f(6)$, given that:

$$f(x) = \begin{cases} f(x-2) + 3, & \text{when } x > 1 \\ 2, & \text{when } x = 1 \\ 1, & \text{when } x = 0 \end{cases} \quad f(6) = \underline{\hspace{2cm}}$$

2. Evaluate $f(12, 6)$, given that:

$$f(x, y) = \begin{cases} f(x-y, y-1) + 2, & \text{when } x > y \\ x + y, & \text{otherwise} \end{cases} \quad f(12, 6) = \underline{\hspace{2cm}}$$

3. Given the method,

```
public static int result( int num)
{
    if( num == 1 )
        return 1;
    else
        return num + result(num - 1);
}
```

what is the output of the following calls?

- a. `System.out.println(result(1));`
- b. `System.out.println(result(2));`
- c. `System.out.println(result(3));`
- d. `System.out.println(result(4));`

4. Consider the following function

```
public void mystery(int x)
{
    System.out.print((x % 10) + " ");
    if ((x / 10) != 0)
        mystery(x / 10);
}
```

What is printed as the result of the call `mystery(1234)`?

5. Consider the recursive method whose definition appears below.

```
public static String mysteryString(String s)
{
    if (s.length() == 1)
        return s;
    else
        return s.substring(s.length() - 1) +
               mysteryString(s.substring(0, s.length() - 1));
}
```

What is the result of the following call?

`System.out.println(mysteryString("computer"));`

Lab12 Folder Listing

Objective

To print files and directories recursively.

Background

The directories (folders) on your hard drive also demonstrate a recursive structure. Each subdirectory may be either a directory or a file. That's the recursive part.

This lab will print all the files and their pathnames in any given directory structure. First, prompt the user to enter the complete path of a directory, for instance, "C:\temp". Instantiate "C:\temp" as a File object `f`. Then call `f.oo`. The method `f.oo` implements a recursive algorithm for printing all the files in the directory structure, namely, if `f` is a directory, then make an array of the files, and recursively test whether each element in the array is a directory. If `f` is not a directory (that's the base case), then print the string representation of `f`.

How do you do all that? The File class has several helpful methods:

	<u>File</u> (String pathname) Instantiates a new File object from given pathname string.
boolean	<u>isDirectory</u> () Tests whether the File object is a directory.
File[]	<u>listFiles</u> () Returns an array of the files in the directory.
String	<u>getPath</u> () Returns the pathname string denoting the abstract pathname of this File.

You can copy and paste the path that is showing in the jGrasp window. If you are on a Windows computer at home, try entering C:\Windows or C:\Program Files.

At the right is a partial listing of C:\temp in my computer. Warning, you may want to test your program with a directory that will have fewer total files than C:\temp.

```
c:\temp\System32\drivers\drivers\vsapint.sys
c:\temp\System32\drivers\TMFilter.sys
c:\temp\System32\drivers\tmpreflt.sys
c:\temp\System32\drivers\tmtdi.sys
c:\temp\System32\drivers\VSAPINT.SYS
c:\temp\!db_pcc.dat
c:\temp\license.rtf
c:\temp\PCC.exe
c:\temp\setup.exe
```

Specification

Open Unit5\Lab12\Driver12.java. Complete the `f.oo` method. Since there may be quite a lot of files to list, the driver prints the output to a text file rather than to the command window.

Sample Run

Unit5Lab12.jar. You can copy and paste the path that is showing in the jGrasp window. If you are on a Windows computer at home, try entering C:\Windows or C:\Program Files.

Exercises

Lab12

1. What is output by the call `fun(3)`? _____

```
public void fun(int x)
{
    if (x>=1)
    {
        System.out.print(x);
        fun(x-1);
    }
}
```

2. Consider the following method:

```
public void mystery(int a, int b)
{
    System.out.print (a + " ");
    if (a <= b)
        mystery(a + 5, b - 1);
}
```

What is the output when `mystery(0, 16)` is called? _____

3. Consider the following method:

```
public int change(int value)
{
    if(value < 3)
        return value % 3;
    return value % 3 + 10 * change(value/3);
}
```

What will be returned by the call `change(45)`? _____

4. What is output by the call `fun(3, 6)`? _____

```
public int fun(int x, int y)
{
    if (y == 2)
        return x;
    else
        return fun(x, y-1) + x;
}
```

5. Consider the following method:

```
public int arithSeq(int n, int a, int d)
{
    if (n == 1)
        return a;
    return d + arithSeq(n-1, a, d);
}
```

What value is returned by the call `arithSeq(3, 2, 6)`? _____

Lab13

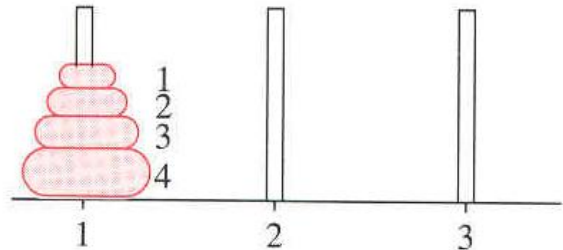
Towers of Hanoi

Objective

To solve a classic problem in recursive thinking.

Background

We have 3 pins and a tower of disks, smaller on top of bigger, on the first pin. The purpose of the puzzle is to move the whole tower from the first pin to the second, by moving only one disk every time and by taking care not to put a bigger disk on top of a smaller one.



Thinking recursively, we would say that moving 3 disks means we need to move 2 disks. Moving 2 disks means we need to move 1 disk. Moving 1 disk is easy, so let's print that move. So far, we have stacked up three levels. Then the recursion, all by itself, goes back down the stored levels, moving each disk in its turn. Moving a tower of disks is just like moving the one-step smaller tower. Recursion!

Print out the sequence of movements necessary to move N disks from one pin to the other. Your code will contain exactly one `System.out.println` statement, which happens to be called every time you move one disk. Your code will call `tower` twice, each time decreasing the number of disks. The base case can be, if the number of disks equals 0, then return. Alternatively, the implied base case can be, if $N \geq 1$, move.

The first call to `tower` is: `tower(1, 3, 2, number);`

The header of the `tower` method is:

```
public static void tower(int start, int finish, int helper, int number)
```

so that the start pin is 1, the finish pin is 3, and the helper pin is 2. Here are two sample runs:

Specification

Open `Unit5\Lab13\Lab13.java`.
Using recursion, print the sequence of moves.

Extension

Display the solution graphically.

Sample Run

`Unit5Lab13.jar`
`Unit5Lab13ext.jar`

Sample run, n = 3

```
Move disk 1 from 1 to 3.
Move disk 2 from 1 to 2.
Move disk 1 from 3 to 2.
Move disk 3 from 1 to 3.
Move disk 1 from 2 to 1.
Move disk 2 from 2 to 3.
Move disk 1 from 1 to 3.
```

Sample run, n = 4

```
Move disk 1 from 1 to 2.
Move disk 2 from 1 to 3.
Move disk 1 from 2 to 3.
Move disk 3 from 1 to 2.
Move disk 1 from 3 to 1.
Move disk 2 from 3 to 2.
Move disk 1 from 1 to 2.
Move disk 4 from 1 to 3.
Move disk 1 from 2 to 3.
Move disk 2 from 2 to 1.
Move disk 1 from 3 to 1.
Move disk 3 from 2 to 3.
Move disk 1 from 1 to 2.
Move disk 2 from 1 to 3.
Move disk 1 from 2 to 3.
```

Exercises

Lab13

1. What is output by the call `fun(3)`? _____

```
public int fun(int x)
{
    if (x<1)
        return x;
    else
        return x + fun(x-1);
}
```

2. What is the output when `mysterySum(6)` is called? _____

```
public int mysterySum(int a)
{
    if (a <= 1)
        return 1;
    else
        return 10 + mysterySum(a - 2);
}
```

3. Consider the following method:

```
public int getSomething(int value)
{
    if(value < 1)
        return 0;
    else
        return 1 + getSomething(value - 1) + getSomething(value - 2);
}
```

What is returned by the call `getSomething(4)`? _____

4. Consider the following method:

```
public void change(int value)
{
    if(value < 5)
        System.out.print("" + value % 5);
    else
    {
        System.out.print("" + value % 5);
        change(value/5);
    }
}
```

What will be printed as a result of the call `change(29)`? _____

5. What is output by the call `fun(3)`? _____

```
public void fun(int x)
{
    if (x<1)
        System.out.print(x+" ");
    else
    {
        System.out.print(x+" ");
        fun(x-1);
    }
}
```

Lab14

Fractal Trees

Objective

To make the turtle draw fractal trees using *recursion*.

Background

Study the driver class.

Line 9 instantiates a TurtlePanel, which has a listener and a timer that repaint the screen.

Line 11: Recall that Turtle is an abstract class. Since we can't normally instantiate an object from an abstract class, we use a trick on lines 13 and 14 to get around that restriction.

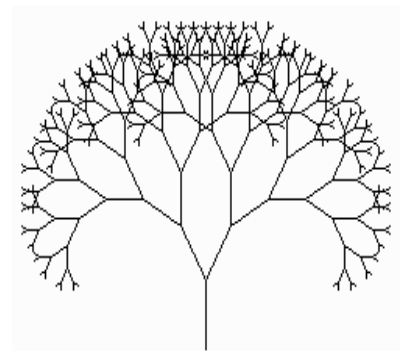
Line 19: The tree method, which you will have to write, uses the Turtle's instance methods forward, back, turnLeft, and turnRight.

```
1 public class Driver11
2 {
3     public static void main(String[] args)
4     {
5         JFrame frame = new JFrame("Tree Turtles");
6         frame.setSize(400, 400);
7         frame.setLocation(200, 100);
8         frame.setDefaultCloseOperation(
9             JFrame.EXIT_ON_CLOSE);
10        frame.setContentPane(new TurtlePanel());
11        frame.setVisible(true);
12        Turtle smidge = new Turtle(300.0, 500.0, 90.0)
13        {
14            public void drawShape()
15            {}
16        };
17        smidge.setCrawl(true);
18        smidge.setSpeed(1);
19        int level = Integer.parseInt(
20            JOptionPane.showInputDialog(
21                "How many levels?"));
22        tree(smidge, 70.0, 30.0, level);
23        //treeExt(smidge, 70.0, 25.0);    //extension
24    }
25 }
```

Recursion is a technique in which a method calls a "one-step smaller" copy of itself. Recursion is not a loop! The key is to see the "tree" as a line with two smaller "trees" coming out at different angles from the top. That is, the trunk, branches, and leaves are all just "trees" that get progressively smaller. Therefore we can program the whole tree-shape recursively as follows:

a. If the level reaches 0, then return;, which stops the recursion. This is the "base case," where the recursion stops.

b. In all other cases, have the turtle draw the trunk and call tree again, which will draw a smaller trunk coming out of it to the left. Somehow, call tree a second time to draw the trunks to the right. Try starting with a size of 70, each new trunk getting smaller by 10, and angles equal to 30°.



Specification

Open Unit5\Lab14\Driver14.java. Write the recursive tree method.

Use Unit5\Lab14\TurtlePanel.java. Don't change this class!

Extension

Change the code so that the tree stops when the size of the branches is less than 5. Don't use levels. Make the tree more realistic by using random (20° to 30°) angles and randomly decreasing (-5 to -15) branches. Write a setColor method to make the trunk black, the branches green, and the berries (i.e. the smallest branches) red.

Sample Run

Unit5Lab14.jar





Discussion

Making Fractals

The tree in Lab11 is an example of a fractal, defined as "a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole." The definition of fractals makes clear that fractals are naturally recursive.

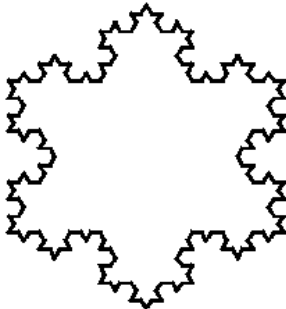
Some famous fractals include the Mandelbrot set, the Julia set, the Sierpinski triangle, and the Koch snowflake. With our Logo turtle, it is easier to draw a fractal Koch line, which can be repeated to make the Koch snowflake. Here is the algorithm: begin with a Koch segment. Replace it with 4 smaller (one-third as big) Koch segments. Turn so that two of those smaller Koch segments form a "spike." Then recur on each and every Koch segment, ad infinitum.

In other words, your program should produce the following fractal shapes at the given level:

Level = 0		A Koch segment
Level = 1		4 smaller (one-third the size) Koch segments
Level = 2		Each and every segment has 4 smaller Koch segments.
Level = 3		Recur ad infinitum.

Sample Run
Unit5Lab14Koch.jar

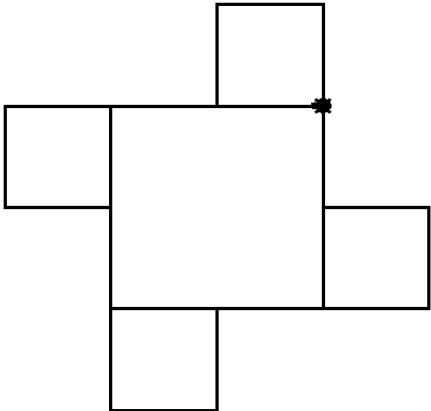
When that is working, write a snowflake method. To draw the Koch snowflake shown at the right, you just call the level 3 Koch line three times, turning the turtle between each call.



Did you know that Koch snowflakes have a finite area and an infinite perimeter? Amazing!

How about producing this fractal? It looks like the large square has half-size squares at all four corners.

Level 0, the base case, produces nothing.
Level 1 produces only the large square.
Level 2 is shown to the right.
What would the Level 3 fractal look like? Draw it here.



Sample Run
Unit5Lab14Squares.jar

Lab15 Binary Search

Objective

To implement the linear search and the binary search.

Background

A *linear search* uses a loop to examine each item in an array until a certain target is found. For instance:

```
public static int search(String[] array, String target)
{
    for(int k = 0; k < array.length; k++)
        if( array[k].compareTo( target ) == 0 )
            return k;
    return -1;
}
```

For n items, this requires n operations in the worst case, i.e. when the target is not in the array. Whether the array is sorted or not, the linear search still has a worst case of n operations. Fortunately, *if the array is already sorted*, we can get a much better speed than n operations, if we use the Binary Search.

Most people rely on the Binary Search when looking up a name in a phone book, or a word in a dictionary. They open the book near the middle, compare, then choose the half of the book in which to look again. In other words, they discard half of the data each time by continually checking the middle item from the data set still needing to be searched. At each step, half the data is discarded. The number of steps k is related to n elements by the equation $n = 2^k$. Take the log of both sides to get $k = \log_2 n$.

Length of list n	Maximum number of steps k	
	Linear	Binary
1	1	1
3	3	2
7	7	3
15	15	4
31	31	5
63	63	6
127	127	7
255	255	8
511	511	9
1023	1023	10
n	n	$\log_2 n$

Therefore in the worst case a binary search needs only $\log n$ comparisons while a linear search requires n comparisons. Examine the table at the left. Binary Searches are much faster than linear searches.

Given 42 different presidents, in alphabetical order, how many steps does it take to find "Madison"? To find "Fillmore"? How will the computer decide that "Obama" is not in the list?

In your own words, write the algorithm for the Binary Search.

1	Adams, J
2	Adams, J. Q.
3	Arthur
4	Buchanan
5	Bush, G.
6	Bush, G. W.
7	Carter
8	Cleveland
9	Clinton
10	Coolidge
11	Eisenhower
12	Fillmore
13	Ford
14	Garfield
15	Grant
16	Harding
17	Harrison, B.
18	Harrison, W.
19	Hayes
20	Hoover
21	Jackson
22	Jefferson
23	Johnson, A.
24	Johnson, L.
25	Kennedy
26	Lincoln
27	Madison
28	McKinley
29	Monroe
30	Nixon
31	Pierce
32	Polk
33	Reagan
34	Roosevelt, F
35	Roosevelt, T
36	Taft
37	Taylor
38	Truman
39	Tyler
40	Van Buren
41	Washington
42	Wilson

Specification

Use Unit5\Lab15\data.txt. The data is decimal numbers formatted to one decimal place, $0.0 \leq x < 10.0$. You may generate a different set of numbers if you run MakeDataFile.java.

Open Unit5\Lab15\Driver15.java. Implement both the linear search and the binary search. (You may implement the binary search either recursively or iteratively.) Count how many steps it takes to find the target.

Sample Run

Unit5Lab15.jar