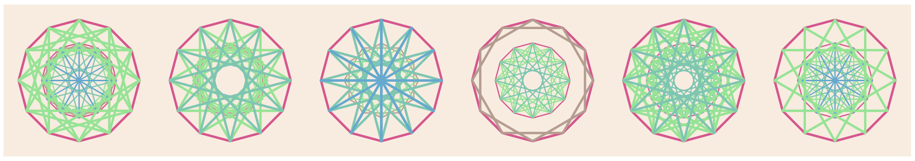
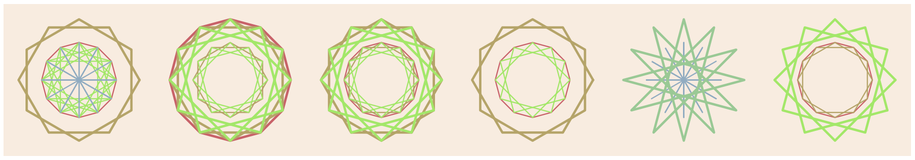
Exercises, Handouts, and Notes

CSC144 Object-Oriented Programming

15 December



Contents



1 November 15 (a.m.)

1.1 The Java Programming Language

1.1.1 Origins

- Sun Microsystems Corporation (now a part of the Oracle Corporation)
- James Gosling
- May 1995

1.1.2 Distinguishing features

- object-oriented
- a hybrid language
  - compiled and interpreted
  - Java bytecodes
  - “write once, run anywhere”
- did not become the language of the Web
- did become a very popular language
- large, standardized API
- well-suited for many kinds of applications
  - desktop
  - mobile
  - Web
  - embedded
  - enterprise

1.1.3 The future of Java

- Java continues to evolve
- computer scientists have created new languages that are compatible with Java, but better

1.1.4 Udacity course

Java Programming Basics

1.1.5 Oracle: The Java Tutorials

The Java Tutorials

1.1.6 Object-Oriented programming

- procedural languages like FORTRAN beginning in mid-1950s
- structured programming with languages like Pascal and C in 1970s
- abstract data type languages like Ada in 1980s
- object-oriented languages like C++ (1980s) and Java (1990s)
- a **package** is a collection of related classes
- a **class** is a blueprint for the creation of objects
- an **object** is a bundle of related data and methods for working with that data
- we will also use **class** to mean a bundle of related methods (without any data)
- one class can be related to another class
  - through **inheritance**: 'class A extends B' means that A is a kind of B
  - through **aggregation** (also called **composition**: a class that models a triangle might contain 3 instances of a class that models points.

1.1.7 Downloads

- Netbeans
- Java JDK

2 November 16: Exercise

```
/*
 * Statistics.java
 * CSC144 Object-Oriented Programming
 * Leon Tabak
 * 16 November 2021
 *
 * This is an example created for the purpose
 * of introducing students to the Java
 * programming language.
 *
 * TO-DO: Create your own working copy of this
 * program. Work with a partner. Replace the name
 * of the program's author with your own name.
 *
 * TO-DO: Take notes during our discussion of
 * this program.
 *   - Identify reserved words of the the Java language.
 *   - Look for rules and conventions.
 *   - Learn the distinction between rules and conventions.
 *   - We will point to some idioms.
 *   - Where could you make changes to the code without
 *      changing the program's functionality.
 *
*/

package statistics;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class Statistics {

    public static List<Integer> makeList( int size, Random rng ) {
        List<Integer> list = new ArrayList<>();

        for( int i = 0; i < size; i++ ) {
            int n = 10 + rng.nextInt(90);
            list.add( n );
        } // for

        return list;
    } // makeList( int )

    public static void printList( List<Integer> list ) {
        for( int n : list ) {
            System.out.println( n );
        } // for
    } // printList( List<Integer> )

    public static int numberOfOdds( List<Integer> list ) {
        int count = 0;

        for( int n : list ) {
            if( n % 2 == 1 ) {
                count++;
            } // if
        } // for

        return count;
    } // numberOfOdds( List<Integer> )


    // TO-DO: Write a method that returns to its
    // caller the number of even integers in a
    // list of integers.

    // TO-DO: Write a method that returns to its
    // caller the number of integers in a list
    // of integers that are divisible by 3.

    public static void main(String [] args) {
        // Create a random number generator.
        Random rng = new Random();

        // Create a list of 12 random numbers,
        // each a two digit positive integer.
        List<Integer> data = makeList( 12, rng );

        // Print the list of random numbers.
        printList( data );

        // Print the number of odd numbers in the list.
        System.out.println( "# of odd numbers = " +
            numberOfOdds(data) );
    } // main(String[])

} // Statistics
```

3 November 16

Browse on one or more of the websites that I have listed under the Resources tab and in the General Resources section of this Piazza site.

Look for...

- people with interesting backgrounds
- people with interesting responsibilities
- interesting aspects of the organization's philosophy, values, or ways of working

Share what you learn in a paragraph or two on this Piazza site.

4 November 17: Assignment for Week 0

Programming Exercise

1. Write a method that returns to its caller the value of the smallest integer in a list of integers.
2. Write a method that returns to its caller the position (that is, the index) of the smallest integer in a list of integers.
3. Write a method that returns to its caller the position of the smallest integer in that part of a list of integers that begins at a specified index.
4. Write a method that exchanges the values at two positions in a list of integers.
5. Write a method that sorts a list of integers using the selection sort algorithm.
6. Write a method whose parameters are an integer m and a list of n integers whose first m integers are in ascending order.

   $$0 \leq m \leq n$$

   The method will move the $(m+1)^{th}$ integer to the left until the first $(m+1)$ integers in the list are in ascending order.
7. Write a method whose parameters are a sorted list of m integers and a sorted list of n integers.. The integers in both lists will be in ascending order.

   The method will create and return to its caller a list that contains all of the integers in the two lists it was given, in ascending order.

Reading and Writing Exercise

Read more on the websites of the companies that you will find listed on our course's Piazza site in the General Resources section under the Resources tab.

1. Write 256 words about one of the organizations. Tell your readers something about the firm's mission, values, and/or its offices and the environment in which its employees work.

   What distinguishes this company? Do you see evidence of leadership? Innovation? Caring and support for employees?

   Whet your readers' appetites. Give them reasons to want to learn more about the organization.

   Inform and persuade.

   Post your note in the *Week-0* folder on Piazza.

2. Write 256 words about people at one of these organizations. Choose two or three or four people whose responsibilities, skills, interests, and experiences make them especially intriguing to you.

   Choose examples that will give your readers a good picture of the kind of people that work in the firm.

   Give your readers reasons to think: "These are the kinds of people with whom we should be working."

   Inform and persuade.

   Post your note in the *Week-0* folder on Piazza.

5 November 17 (a.m.)

5.1 Structure of a Java programming

- the **package** statement is the first statement in a Java source code file
- classes in a package must be in a folder with the same name as the name of the package
- by convention, names of packages begin with lower case letters
- **import** statements follow
- definition of the **class** follows that
- the name of the class matches the name of the file that contains the definition of the class
- by convention, the name of the class begins with a capital letter
- within the definition of the class, there are definitions of methods
- "method" is a synonym (or near synonym) of function, procedure, subroutine
- distinction between method and function—a method is a function defined within a class
- parts of a function...
  - **public** or **private**
  - type of value that method returns to its caller
  - (if method computes and returns no value, then the return type is **void**)
  - name of method (begins with a lower case letter)
  - list of method's parameters together with the types of those parameters
  - (the parameters are the information that the method needs to do its job)
  - the body of the method (a sequence of statements)
- statements

    - all statements end with a semicolon
    - statements allow us to do arithmetic and store the result of a calculation (an assignment)
    - we can use a statement to make a decision (**if** statements conditionally execute some other group of statements)
    - we can use a statement to repeat the execution of some group of other statements (**for** and **while** loops)
- variables
    - names begin with a lower case letters
    - names may be the concatenation of several words
    - every word after the first is capitalized ("camel case")
    - programmer must explicitly specify the type of each variable: "int n = 5;"

```java
package wednesday;

import java.util.List;
import java.util.ArrayList;

public class Wednesday {

  public static void main( String [] aardvark ) {
      System.out.println( "Hi!" );

      List<Integer> list = new ArrayList<>();

      list.add( 1 );
      list.add( 1 );
      list.add( 2 );
      list.add( 3 );
      list.add( 5 );
      list.add( 8 );

      for( int n : list ) {
          System.out.println( n );
      } // for
  } // main( String [] )

} // Wednesday
```

6 November 17 (p.m.)

```java
package selection;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class Selection {

    public static List<Integer> makeList( int size, Random rng ) {
        List<Integer> result = new ArrayList<>();

        for( int i = 0; i < size; i++ ) {
            result.add( rng.nextInt( 100 ) );
        } // for

        return result;
    } // makeList( int, Random )

    public static int posOfSmallest( List<Integer> data, int start ) {
        int bestGuessSoFar = start;

        for( int i = start + 1; i < data.size(); i++ ) {
            if( data.get(i) < data.get(bestGuessSoFar) ) {
                bestGuessSoFar = i;
            } // if
        } // for

        return bestGuessSoFar;
    } // posOfSmallest( List<Integer>, int )

    public static void exchange( List<Integer> data, int i, int j ) {
        int temp = data.get(i);
        data.set( i, data.get(j) );
        data.set( j, temp );
    } // exchange( List<Integer>, int, int )

    public static void selectionSort( List<Integer> data ) {

        for( int i = 0; i < data.size(); i++ ) {
            int j = posOfSmallest( data, i );
            exchange( data, i, j );
        } // for

    } // selectionSort( List<Integer> )

    public static void printList( List<Integer> data ) {
        for( int datum : data ) {
            System.out.println( datum );
        }  // for
    } // printList( List<Integer> )

    public static void main( String[] args ) {

        Random rng = new Random();
        List<Integer> samples = makeList( 16, rng );

        printList( samples );

        System.out.println();

        int index = posOfSmallest( samples, 8 );

        System.out.println( "index = " + index );

        System.out.println();

        exchange( samples, 3, 4 );

        printList( samples );

        selectionSort( samples );

        System.out.println();

        printList( samples );
    } // main( String [] )


} // Selection
```

7 November 18 (a.m.)

7.1 Definition of "variable"

Computer scientists use the word "variable" in a different way than do mathematicians and engineers.

For a mathematician, "variable" might connote the unknown quantity. The variable is a placeholder for a value we do not yet know. We solve an equation to find a variable's value.

For an engineer, "variable" might connote the changeable or adjustable value. We turn a knob to increase voltage, temperature, or the brightness of a light.

For a computer scientist, "variable" simply means a named location in the computer's memory. Six attributes completely define a variable:

- name
- location (also called address)
- value
- type
- scope
- lifetime

Name: Your computer's memory holds billions of cells. Your variable is one of those cells. Rather than refer to memory cell 2,319,908,435 we will refer to the cell by a name. We get to choose the name. We might, for example, call it "highestTemperatureInJuly."

Address: Our computer's operating system will find a free memory cell for us when we create a variable. It will associate that cell's numerical address with the name we give to our variable. Because we seldom need to know the numerical address of our memory cell, few programming languages let us see the numerical address.

Value: We can store many kinds of information in a computer's memory. We can store photographs, video recordings, recordings of music and speech, and letters to Mom. In the end, our software will reduce it all to numbers. Computer scientists have invented codes they use to represent images, sounds, and text as numbers. You probably know the names of some of these codes (for example, JPEG and MP4) but might not know the names of others (for example, Unicode or the IEEE Standard 754 for the representation of floating point numbers). The good news—you do not need to know anything about these codes. (Well, there might be some few circumstances in which it might help to know a little.) The value is the number that we store in our named location in the computer's memory or, in other contexts, its translation to a letter, a pixel, or a tone.

Type: The type of a variable signifies how the value is represented in memory and how we may use the variable. For example, addition makes sense with some types but not with others. Knowing that a variable's type is "32 bit signed integer" also tells us the range of values that the variable might hold. (For the curious—that type allows values from −2147483648 to +2147483647.)

Scope: The scope is that part of our program in which we may refer to our variable. If you work in an office, you put limits on who can browse through the contents of your filing cabinet. Similarly, programmers restrict the visibility of variables within a program. Some parts of the program can access the variable. Other parts cannot.

Lifetime: Our software might create some variables even before the computer begins the execution of our program. Our program might also create some variables during its execution. When a program is done with a variable, it can release that memory. Variables are born and variables die. A variable has a lifetime.

7.2 Primitive and reference types

Java has 8 primitive types:

- **int**
- **byte**
- **char**
- **short**
- **long**
- **float**
- **double**
- **boolean**

When you type "int n = 5;" you are telling the computer to allocate 32 bits of memory, give that memory the name 'n', and store the number 5 in that memory cell.

Java also has reference types.

When you type "Color background = new Color(128, 192, 248);" you are telling the computer to allocate enough memory to hold an instance of the 'Color' class and also enough memory to store the address of that block of memory that holds the description of the color. The computer stores the address of that block of memory in the memory cell named 'background'. It does not store the 'Color' object in that memory cell. The memory cell that holds the address of the 'Color' object and the memory cells that hold the 'Color' object will likely be very far apart—the address is unlikely to be adjacent to the object.

8 November 18 (p.m.)

```java
package othersorts;

import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;

public class OtherSorts {

    public static void main( String [] args ) {
        // A list whose first 4 elements are in order
        Integer [] samples = { 2, 5, 8, 14, 6, 4, 1, 11 };
        List<Integer> data = Arrays.asList( samples );

        for( int n : data ) {
            System.out.println( n );
        } // for

        // Can you write code that will transform the
        // list so that it looks like this:

        // A list whose first 5 elements are in order
        // 2, 5, 6, 8, 14, 4, 1, 11

        Integer [] a = { 2, 4, 7, 12 };
        List<Integer> aList = Arrays.asList( a );

        Integer [] b = { 1, 5, 17, 19 };
        List<Integer> bList = Arrays.asList( b );

        List<Integer> cList = new ArrayList<>();

        // Can you write code that will fill cList
        // with all of the numbers in aList and bList
        // in order?

    } // main( String [] )

} // OtherSorts
```

9 November 18: Guidelines for writing

- Put a subject and a verb in every sentence. Write complete sentences.
  - "For example, Simon and Garfunkel." is not a sentence.
  - "For example, Simon and Garfunkel performed together." is a sentence.
- Tell your reader who did what rather than what was done to whom.
  - Favor the active voice over the passive voice.
  - "The house was painted by the owner." is in the passive voice.
  - "The owner painted the house." is in the active voice.
- Prefer several short sentences to a single long sentence.
- Divide large paragraphs into 2 or 3 shorter paragraphs.
- Be careful when you cut and paste text. If not careful, you might copy more or less than you want. Your readers might then see repeated words or look for missing words. This could frustrate or confuse your readers.
- You may write in the first person: "I share some of the interests that the company's software engineers listed."
- You should express personal opinions.
  - Rather than write "I feel that I would enjoy working with these people" write "I would enjoy working with these people."

10 November 18: Understanding and comparing sorting algorithms

10.1 Measuring computational complexity

- My favorite definition of computer science (in the form of 4 questions)
  - What kinds of problems can I solve with the help of a computer? (Computability)
  - Given a problem that I can solve, how can I write the program?
  - Given two programs that solve a problem, how can I pick the best one? (Complexity)
  - Given a program that solves my problem, how can I prove that it really will solve my problem every time without error? (Correctness)

10.2 "Big-Oh"

$O(N^2)$ means the time required to sort the list is roughly proportional to the length of the list squared.

$O(N \log N)$ means the time required to sort the list is roughly proportional the length of the list times the logarithm of the length of the list.

- What is a logarithm?
  - $2^{10} = 1024$
  - $\log 2^{10} = 10$
  - $2^{16} = 65536$
  - $\log 2^{16} = 16$

10.3 Complexity of sorting algorithms

- selection sort is $O(N^2)$
- insertion sort is $O(N^2)$
- merge sort is $O(N \log N)$

11 November 19 (a.m.)

11.1 Sequential Search

- search in an unsorted list
- must examine every element in the list
- $O(N)$
- for example, to search for the minimum value
  - begin by assuming that the first value is the minimum
  - examine each other value in turn
  - update our estimate of which value is smallest as necessary

11.2 Selection Sort

- $O(N^2)$
- begin by finding the position of the smallest value in the whole list
- move that value to the front of the list (by exchanging its value with the value at position 0)
- then search through that part of the list that begins at position 1, looking for the position of the smallest value in that part of the list
- move that element to position 1
- then search through the part of the list that begins at position 2, move the smallest element in that part of the list to position 2, and so on
- at the k^{th} step of the selection sort, the first k elements are in order
- at the k^{th} step, the first k elements are the k smallest elements in the list

11.3 Insertion Sort

- $O(N^2)$
- at the k^{th} step, the first k elements of the list are in order
- at the k^{th} step, the first k elements are the first k elements that we saw in the list before starting the sorting
- at each step, move the element at position $k + 1$ into the right place among the k elements are already sorted

11.4 There are searches inside both sorts!

- in the selection sort, the search is from left to right through the unsorted part of the list
- in the insertion sort, the search is from right to left through the already sorted part of the list
- the search in the insertion sort might not have to examine every element in the sorted part of the list
- the search in the selection sort does have to examine every element in the unsorted part of the list
- the insertion sort is (on average) faster than the selection sort (but both are still $O(N^2)$!)
- insertion sort works best on lists that are already nearly in order

11.5 Merge Sort

- key insight: if I have two already sorted lists, merging them to produce one bigger sorted lists is easy
- can be done recursively
- sort the two halves of our list
- merge the two halves
- how to sort each half?—sort two quarters and merge!
- how to sort each quarter?—sort two eighths and merge!
- this is a recursive algorithm (it is also possible to write the merge sort without recursion)
- this is a divide and conquer algorithm
- this is a $O(N \log N)$ algorithm

12 November 29: A class as a blueprint

Here is a program that defines a class that models a weight, creates several instances of that class, and exercises the methods of that class.

A class that models a length (feet and inches), time (hours and minutes), fraction (numerator and denominator), vector (x and y components), or complex number (real and imaginary parts) will be very similar. In each case, the class models a number with two parts and a special rule for addition.

Get a version of this program working on your own computer. Then try writing a program that defines a class that models a length or time.

```java
package weight;

public class Weight {
    private static final int OUNCES_IN_A_POUND = 16;

    private int pounds;
    private int ounces;

    public Weight( int pounds, int ounces ) {
        this.pounds = pounds;
        this.ounces = ounces;
    } // Weight( int, int )

    public int getPounds() {
        return this.pounds;
    } // getPounds()

    public int getOunces() {
        return this.ounces;
    } // getOunces()

    public void setPounds( int pounds ) {
        this.pounds = pounds;
    } // setPounds( int )

    public void setOunces( int ounces ) {
        this.ounces = ounces;
    } // setOunces( int )

    public Weight add( Weight otherWeight ) {
        int lbs = this.pounds + otherWeight.pounds;
        int oz = this.ounces + otherWeight.ounces;
        Weight sum = new Weight( lbs + oz/OUNCES_IN_A_POUND,
                                 oz % OUNCES_IN_A_POUND );
        return sum;
    } // add( Weight )

    @Override
    public String toString() {
        return this.pounds + " lbs., " + this.ounces + " oz.";
    } // toString()

    public static void main( String[] args ) {
        Weight cherries = new Weight( 1, 10 );
        Weight grapes = new Weight( 2, 8 );
        Weight fruit = cherries.add( grapes );
        System.out.println( cherries + " + " + grapes + " = " + fruit );
    } // main( String[] )

} // Weight
```

13 November 29: Parts of a class

The definition of a class begins with a ‘package‘ statement. By convention, the name of a package begins with a lower case letter.

```
package weight;
```

A header follows. All of our classes will be ‘public‘. By convention, we will give our classes a name that begins with a capital letter:

```
public class Weight {
```

We might sometimes want to define a constant that will be shared by all instances of the class. The word ‘final‘ signifies a constant. The word ‘static‘ signifies one copy of the constant that is shared by all instances of the class.

```
    private static final int OUNCES_IN_A_POUND = 16;
```

A class will generally have instance variables. Each instance of the class has its own instance variables. By convention, we will make these instance variables ‘private‘ and give them names that begin with a lower case letter.

```
    private int pounds;
    private int ounces;
```

A class has one (or more) constructors. A class differs from a method by the fact that it has no return type (in this case, no ‘int‘ or ‘double‘ or ‘String‘ or ‘void‘ between ‘public‘ and ‘Weight‘) and its name is the same as the name of the class.

The word ‘this‘ is a way to refer to an instance of the class that we are creating. The instance will get its own name after we are completely done creating it.

In this example, the constructor has parameters named ‘pounds‘ and ‘ounces‘. The class also has instance variables named ‘pounds‘ and ‘ounces‘. The parameter is not the same as the instance variable—there are 2 variables named ‘pounds‘ in this example! One is a parameter and one is an instance variable. We can distinguish between them by prefixing the reference to the instance variable with ‘this‘.

```
    public Weight( int pounds, int ounces ) {
        this.pounds = pounds;
        this.ounces = ounces;
    } // Weight( int, int )
```

An accessor method (also called a *getter*) is a means of retrieving the value of a ‘private‘ instance variable. In this way, we give teammates who are working on other parts of our program a means of peeking inside the instance of a class.

A getter has no parameters.

The return type of a getter matches the type of an instance variable.

```
    public int getPounds() {
        return this.pounds;
    } // getPounds()

    public int getOunces() {
        return this.ounces;
    } // getOunces()
```

Sometimes we will also want to give our teammates to change the values of our instance variables. For that, we define *setters* (also called mutators).

These setters allow a programmer to give the instance variables any value at all. In other cases, you might want setters that put limits on the new values of the instance variables. For example, you might want to prevent a programmer from giving the denominator of a fraction a value of 0.

The return type of a setter is always 'void'. The type of its single parameter matches the type of an instance variable.

```
    public void setPounds( int pounds ) {
        this.pounds = pounds;
    } // setPounds( int )

    public void setOunces( int ounces ) {
        this.ounces = ounces;
    } // setOunces( int )
```

Then we might have methods that combine values or do other useful work.

```
    public Weight add( Weight otherWeight ) {
        int lbs = this.pounds + otherWeight.pounds;
        int oz = this.ounces + otherWeight.ounces;
        Weight sum = new Weight( lbs + oz/OUNCES_IN_A_POUND,
                                  oz % OUNCES_IN_A_POUND );
        return sum;
    } // add( Weight )
```

We will often want a method that gives us a printable representation of an instance of our class.

For this, we *override* the definition of a method that our class has inherited from the 'Object' class.

```
    @Override
    public String toString() {
        return this.pounds + " lbs., " + this.ounces + " oz.";
    } // toString()
```

While every program that we write will have at least one method with a 'main()' method, not every class has to have a 'main()' method.

However, even when it is not necessary to define a 'main()' method, it will often be useful to do so. For example, this 'main()' method contains code that tests my 'Weight' class.

```
    public static void main( String[] args ) {
        Weight cherries = new Weight( 1, 10 );
        Weight grapes = new Weight( 2, 8 );
        Weight fruit = cherries.add( grapes );
        System.out.println( cherries + " + " + grapes + " = " + fruit );
    } // main( String[] )
```

14 November 29: Learning how to use Git and GitHub

- Create an account on *github.com*.
- Login to your GitHub account.

- Create a Personal Access Token.
  - Select Settings.
  - Select Developer Settings.
  - Select Personal access tokens.
- Save that token in a text file on your desktop.
- Create an empty repository.
- On your own computer, open NetBeans.
  - Create a new project (a Java application).
  - Add a Java class with a main method.
  - Select Team / Git / Initialize Repository.
  - Select Team / Commit.
  - Select Team / Remote / Push

15 November 30: A class that models vectors

- a vector has a length and a direction
- the length of a vector is also called its magnitude
- we can define vectors in 2, 3, 4, or more dimensions
- vectors are very useful in computer graphics
- we will use a special kind of vector (a kind you might not have seen in your physics or linear algebra course)
- we will define a vector in 2 dimensions but our vector will have 3 components
- $\vec{v} = (x, y, 1)$
- the third coordinate allows us transform this vector through multiplication with 3×3 matrices
- a 3×3 matrix can model a rotation, a scaling, or a translation
- our Vector class will have 3 instance variables
  - x
  - y
  - h (the homogeneous coordinate—in our work, $h = 1$ always)
- our Vector class will have a constructor with 2 parameters (x and y)
- our Vector class will have getters
- our Vector class will have a ‘toString()‘ method
- our Vector class will have ‘add()‘ and ‘subtract()‘ methods (each with a single parameter whose type is ‘Vector‘)
- our Vector class will have a ‘dot()‘ method for computing the dot product of ‘this‘ vector with another vector
- our Vector class will have a ‘magnitude()‘ method

16 December 1: The start of the definition of the Vector class

```java
package com.eonsahead.designvector;

/*
 * Vocabulary that you should know by
 * the end of our course:
 *    encapsulation
 *    information hiding
 *    inheritance
 *    polymorphism
 *
 * Step 0. Define a class that has main() method, a
 * call to System.out.println() in that main() method,
 * and nothing else. Compile and run.
 * Step 1. Add instance variables to the class. Compile.
 * Step 2. Add a constructor. Compile.
 * Step 3. Add a toString() method. Add code that creates
 * an instance of the class in the main() method. Add code
 * in the main method that prints the vector.
 * Step 4. Add getters (also called accessor methods). Use NetBeans'
 * refactor / encapsulate fields command. Add code in main() method
 * that exercises the new methods.
 * Step 5. Add a method to compute a dot product of 2 vectors,
 * plus code to test that new method.
 * Step 6. Add a method to compute the magnitude of a
 * vector, plus code to test that new method.
 *
 */
public class Vector2D {

    // We might think of more than one
    // way to organize the data in our class.
    // "Information hiding" and "encapsulation"
    // mean that the programmers who use our
    // class do not need to know which of the
    // alternatives we chose.

    // We could define 3 separate instance variables.
//     private double x;
//     private double y;
//     private double h;

    // Or we could define a single instance variable
    // that is an array.
    // (Because we have chosen not to define any
    // setters, we could make this instance variable
    // final.)
    private double[] components = new double[3];

    public Vector2D(double x, double y) {
        this.components[0] = x;
        this.components[1] = y;
        this.components[2] = 1.0;
    } // Vector2D( double, double )

    public double getX() {
        return this.components[0];
    } // getX()

    public double getY() {
        return this.components[1];
    } // getY()

    public double getH() {
        return this.components[2];
    } // getH()

    // stub method
    public double dot(Vector2D other) {
        return this.getX() * other.getX() + this.getY() * other.getY();
    } // dot( Vector2D )

    public double magnitude() {
        return Math.sqrt(this.dot(this));
    } // magnitude()

    // Override annotation is helpful but optional.
    // It is a reminder to us and to the compiler that
    // we are redefining a method that this class
    // inherited from the Object class.
    // (All classes inherit from the Object class.)
    @Override
    public String toString() {
        // Java's String class provides a method for
        // formatting strings.
        // %8.4f is a formatting code.
        // It means a floating point value
        // represented with 8 digits in total,
        // including 4 digits to the right of
        // the decimal point.
        return String.format("(%8.4f,%8.4f)", this.getX(), this.getY());
    } // toString()

    public static void main(String[] args) {
        Vector2D u = new Vector2D(3, 4);
        System.out.println("u = " + u);
        System.out.println("x component of u = " + u.getX());
        System.out.println("y component of u = " + u.getY());
        System.out.println("h component of u = " + u.getH());

        Vector2D v = new Vector2D(5, 12);

        System.out.println("dot product = " + u.dot(v));

        System.out.println( "magnitude of u = " + u.magnitude());
        System.out.println( "magnitude of v = " + v.magnitude());

    } // main( String [] )

} // Vector2D
```

17 December 2

1. Inside the selection sort algorithm there is a search algorithm. Explain.

2. (a) What does f() return to its caller?

   (b) What does g() return to its caller?

```java
public static int f( List<Integer> data ) {
    int bestGuessSoFar = data.get(0);

    for( int i = 1; i < data.size(); i++ ) {
        if( data.get(i) < bestGuessSoFar ) {
            bestGuessSoFar = data.get(i);
        } // if
    } // for

    return bestGuessSoFar;
} // f( List<Integer> )

public static int g( List<Integer> data ) {
    int bestGuessSoFar = 0;

    for( int i = 1; i < data.size(); i++ ) {
        if( data.get(i) < data.get(bestGuessSoFar) ) {
            bestGuessSoFar = i;
        } // if
    } // for

    return bestGuessSoFar;
} // f( List<Integer> )
```

3. The selection sort and the insertion sort are both $O(N^2)$ algorithms. The merge sort is an $O(N \log N)$ algorithm.

N	N^2	$N \log N$
32	1024	$32 \cdot 5 = 160$
1024	$1024 \cdot 1024 = 1,048,576$	$1024 \cdot 10 = 10,240$

   In English, why might you favor the merge sort?

4. Merge sort is a divide and conquer algorithm. What does this mean?

5. The first statement in a Java source code file will be what kind of statement?

6. Which of these class headers conforms to our guidelines for good style?

```java
// (a.)
public class time {

// (b.)
public class Time {
```

7. What does **static final** signify in this statement?

```java
public static final int MINUTES_IN_AN_HOUR = 60;
```

8. Which of these two pairs of statements conforms to our guidelines for good style?

```java
// (a.)
private int hours;
private int minutes;

// (b.)
public int hours;
public int minutes;
```

9. This is a constructor. How can you tell?

```java
public Time( int hours, int minutes ) {
  this.hours = hours;
  this.minutes = minutes;
} // Time( int, int )
```

10. What kind of method is this?

```java
public int getHours() {
  return this.hours;
} // getHours()
```

11. What is the purpose of this method?

```java
@Override
public String toString() {
  return String.format( "%2d:%2d", this.hours, this.minutes );
} // toString()
```

12. This is a stub method. Why do programmers write stub methods?

```java
public Time add( Time anotherTime ) {
  return new Time( 0, 0 );
} // add( Time )
```

13. Complete the definition of this method.

```java
public Time add( Time anotherTime ) {
  int sumOfHours = this.hours + anotherTime.hours;
  int sumOfMinutes = this.minutes + anotherTime.minutes;

  // TO-DO: Assign the correct value to h.
  int h = 0;
  // TO-DO: Assign the correct value to m.
  int m = 0;

  return new Time( h, m );
} // add( Time )
```

18 December 2: How to push local files to a GitHub repository

- How to upload a NetBeans project to GitHub

19 December 2: IBM Watson: Final Jeopardy! and the Future of Watson

- IBM Watson: Final Jeopardy! and the Future of Watson

20 December 2: Turing Award winners

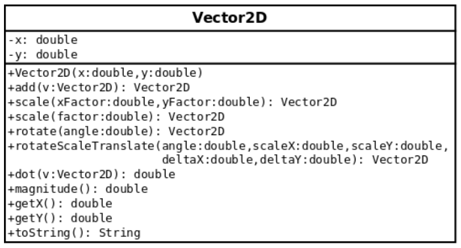
Please read for an hour on this website. Begin to familiarize yourself with some of the principal contributors to the development of computer science, some of the most important achievements in the field, and some the special areas of research in which computer scientists work.

21 December 3: Modeling a fraction

```java
// TO-DO: Add a package statement.

public class Fraction {

  // TO-DO: Add a Javadoc comment here.
  private final int numerator;
  // TO-DO: Add a Javadoc comment here.
  private final int denominator;


  // TO-DO: Add a Javadoc comment here.
  public Fraction( int numerator, int denominator ) {
      // TO-DO: Complete the definition of this constructor.
      // Represent the fraction in the simplest form:
      // for example, 2/3 instead of 4/6 or 18/27.

      // Reduce a fraction to its simplest form by
      // dividing the numerator and denominator by
      // the greatest common divisor of the two
      // numbers. Use the gcd() method (defined below)
      // to compute the greatest common divisor.
  } // Fraction( int, int )

  // Do not define getters or setters for this exercise.

  // TO-DO: Add a Javadoc comment here.
  public Fraction add( Fraction otherFraction ) {
      // This is a stub method.

      // TO-DO: Complete the definition of this method.

      // For example:  a/b + c/d = (ad + cb)/(bd)

      return new Fraction( 0, 1 );
  } // add( Fraction )

  // Do not define methods for subtraction, multiplication,
  // or division---but convince yourself that doing this
  // would not be hard.

```

```
    // TO-DO: Add a Javadoc comment here.
    public String toString() {
        // This is a stub method.

        // TO-DO: Complete the definition of this method.

        return "0/1";
    } // toString()

    // TO-DO: Add a Javadoc comment here.
    private int gcd( int a, int b ) {
        // This is a "helper" method.
        // It helps constructors or other methods within
        // the class do their jobs.
        // Because all calls to this method will
        // be within this class, the method can be
        // private.

        // Compute the greatest common divisor
        // of a and b recursively.

        // For example: gcd(12, 8) = 4

        // A recursive method has an if statement
        // and a call to itself.

        if( b == 0 ) {
            // Here is where the recursion stops.
            return a;
        } // if
        else {
            // Here is where the gcd() method
            // calls the gcd() method---this is
            // the recursive call.
            return gcd( b, a % b );
        } // else
    } // gcd( int, int )

    public static void main( String [] args ) {
        // Test the methods of the Fraction class.
        Fraction a = new Fraction( 12, 20 );
        Fraction b = new Fraction( 6, 20 );

        // TO-DO: Add code to produce the sum of a and b.

        // TO-DO: Add code to print a, b, and the sum of a and b.

    } // main( String [] )

} // Fraction
```

22 December 3: Cats Cradle program


Vector2D
-x: double
-y: double
+Vector2D(x:double,y:double)
+add(v:Vector2D): Vector2D
+scale(xFactor:double,yFactor:double): Vector2D
+scale(factor:double): Vector2D
+rotate(angle:double): Vector2D
+rotateScaleTranslate(angle:double,scaleX:double,scaleY:double, deltaX:double,deltaY:double): Vector2D
+dot(v:Vector2D): double
+magnitude(): double
+getX(): double
+getY(): double
+toString(): String

22.1 CatsCradle.java

```java
package vector2d;

import java.awt.Container;
import javax.swing.JFrame;
import javax.swing.Timer;

/**
 * Here is a final exercise.
 *
 * <p> This example features animated graphics and a class that models a kind of
 * mathematical object that has two parts (a vector in two dimensions). </p>
 *
 * @author CSC140 Foundations of Computer Science
 * @version 6 March 2013
 */
public class CatsCradle extends JFrame {

  /**
   * Choose values for these parameters that make a pleasing image—experiment!
   */
  private static final int FRAME_WIDTH = 512;
  private static final int FRAME_HEIGHT = 512;
  private static final String FRAME_TITLE = "Cat's Cradle";
  private static final int NUMBER_OF_SIDES = 12;

  /**
   * Create a window on the screen, put a panel in the
   * frame on which to draw a picture, and create a timer
   * that will send periodic signals to the panel that ask it to
   * redraw itself.
   */
  public CatsCradle() {
    this.setSize(FRAME_WIDTH, FRAME_HEIGHT);
    this.setTitle(FRAME_TITLE);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // A JFrame contains a pane to hold panels, buttons,
    // scrollbars, and anything else we want to put into
    // our JFrame.
    Container pane = this.getContentPane();
    CatsCradlePanel panel = new CatsCradlePanel(NUMBER_OF_SIDES);
    pane.add(panel);

    // The "20" here means 20 milliseconds.
    // That means a new image 50 times per second.
    // You can try something different.
    // (Smaller values will result in more images
    // per second: 10 milliseconds between images
    // equals 100 images/second.)
    Timer timer = new Timer(20, panel);
    timer.start();

    this.setVisible(true);
  } // CatsCradle( int, int )

  /**
   * The execution of this program begins here.
   *
   * @param args is a required array that could hold command line
   * arguments (but we will not use any command
   * line arguments in this example).
   */
  public static void main(String[] args) {
    CatsCradle catsCradle = new CatsCradle();
  } // main( String [] )
} // CatsCradle
```

22.2 CatsCradlePanel.java

```java
package vector2d;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Stroke;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.geom.Line2D;
import java.util.Random;
import javax.swing.JPanel;

public class CatsCradlePanel extends JPanel implements ActionListener {

  // Specify the color of the background on which the figures
  // will be drawn.
  private static final Color BG_COLOR = new Color(72, 12, 12);
  private static final Color FG_COLOR = new Color(180, 192, 224);
  // MARGIN gives some separation between the figures drawn
  // and the edge of the panel in which they are drawn.
  // The value must be at least zero and less than 0.5.
  private static final double MARGIN = 0.1;
  // Bigger values of SPEED result in a slower animation.
  // Smaller values of SPEED result in a faster animation.
  private static final double SPEED = 64.0;
  // Specify the thickness of the line segments used
  // to draw the inside and outside figures.
  private static final float OUTSIDE_LINE_THICKNESS = 4;
  private static final float INSIDE_LINE_THICKNESS = 2;
  private int numberOfSides;
  private double outerStep;
  private double innerStep;
  private double outerAngle;
  private double innerAngle;
  private double angle;
  private Color[] colors;
  private Stroke insideStroke;
  private Stroke outsideStroke;

  public CatsCradlePanel(int numberOfSides) {
    this.setBackground(BG_COLOR);
    this.setForeground(FG_COLOR);
    this.numberOfSides = numberOfSides;
    this.outerStep = -2.0 * Math.PI / (SPEED * numberOfSides);
    this.innerStep = +2.0 * Math.PI / (2.0 * SPEED * numberOfSides);
    this.outerAngle = 0.0;
    this.innerAngle = 0.0;
    this.insideStroke = new BasicStroke(INSIDE_LINE_THICKNESS,
            BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND);
    this.outsideStroke = new BasicStroke(OUTSIDE_LINE_THICKNESS,
            BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND);

    // Make a palette of random numbers.
    this.colors = new Color[numberOfSides];

    Random random = new Random();
    for (int i = 0; i < numberOfSides; i++) {
      // In the following 3 statements, instead
      // of 64 and 192 you may use any pair of
      // non-negative integers whose sum is 256
      // (or less than 256 if you wish).
      int red = 64 + random.nextInt(192);
      int green = 64 + random.nextInt(192);
      int blue = 64 + random.nextInt(192);
      this.colors[i] = new Color(red, green, blue);
    } // for
  } // CatsCradlePanel()

  @Override
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2D = (Graphics2D) g;

    // Determine the dimensions of the panel
    // in which we are drawing the figure.
    int w = this.getWidth();
    int h = this.getHeight();

    // Make 2 sets of vertices.
    // The outside vertices lie on a circle
    // whose radius is 1.0 and whose center
    // is at the origin.
    // The inside vertices lie on a circle
    // whose radius is the golden ratio and whose center
    // is also at the origin.
    Vector2D[] outside = new Vector2D[this.numberOfSides];
    Vector2D[] inside = new Vector2D[this.numberOfSides];
    double goldenRatio = 2.0 / (Math.sqrt(5.0) + 1);
    for (int i = 0; i < this.numberOfSides; i++) {
      double fraction = ((double) i) / this.numberOfSides;
      this.angle = fraction * 2.0 * Math.PI;
      double x = Math.cos(angle + outerAngle);
      double y = Math.sin(angle + outerAngle);
      outside[i] = new Vector2D(x, y);

      x = Math.cos(angle + innerAngle);
      y = Math.sin(angle + innerAngle);
      inside[i] = new Vector2D(goldenRatio * x, goldenRatio * y);
    } // for

    // Make the 2 polygons that are defined by the convex hulls of
    // of the outside and inside sets of points (respectively).
    // If there are 3 points in a set, we get an equilateral triangle.
    // If there are 4 points, we get a square.
    // If there are more points, we get a pentagon, hexagon, or n-gon.

    // No rotation in this step---we'll take care of that elsewhere.
    double rotation = 0.0;
    // This is how much bigger we have to make
    // the figures to fill the panel (and still
    // leave a margin between the figures and the
    // edge of the panel).
    double scaleX = (1.0 - 2.0 * MARGIN) * w / 2;
    double scaleY = (1.0 - 2.0 * MARGIN) * h / 2;
    // This is how far we have to move the figures
    // to put their centers at the center of the panel.
    double deltaX = w / 2;
    double deltaY = h / 2;

    g2D.setColor(FG_COLOR);
    for (int i = 0; i < outside.length; i++) {
      Vector2D u = outside[i];
      u = u.rotateScaleTranslate(rotation, scaleX, scaleY, deltaX, deltaY);

      Vector2D v = outside[(i + 1) % this.numberOfSides];
      v = v.rotateScaleTranslate(rotation, scaleX, scaleY, deltaX, deltaY);

      double x0 = u.getX();
      double y0 = u.getY();
      double x1 = v.getX();
      double y1 = v.getY();
      Line2D line = new Line2D.Double(x0, y0, x1, y1);
      g2D.setStroke(this.outsideStroke);
      g2D.draw(line);

      u = inside[i];
      u = u.rotateScaleTranslate(rotation, scaleX, scaleY, deltaX, deltaY);

      v = inside[(i + 1) % this.numberOfSides];
      v = v.rotateScaleTranslate(rotation, scaleX, scaleY, deltaX, deltaY);

      x0 = u.getX();
      y0 = u.getY();
      x1 = v.getX();
      y1 = v.getY();

      line = new Line2D.Double(x0, y0, x1, y1);
      g2D.setStroke(this.insideStroke);
      g2D.draw(line);
    } // for

    // Make the remaining edges (line segments) needed to define
    // the complete graph on the outside set of points.
    g2D.setStroke(this.outsideStroke);
    for (int i = 0; i < this.numberOfSides; i++) {
      Vector2D u = outside[i];
      u = u.rotateScaleTranslate(angle, scaleX, scaleY, deltaX, deltaY);
      double x0 = u.getX();
      double y0 = u.getY();
      for (int j = i + 1; j < this.numberOfSides; j++) {
        Vector2D v = outside[j];
        v = v.rotateScaleTranslate(rotation, scaleX, scaleY, deltaX, deltaY);
        double x1 = v.getX();
        double y1 = v.getY();

        // Make color a function of the line segment's length.
        // (The segment is longer if it connects vertices whose
        // indices differ more.)
        int index = Math.abs(i - j);
        g2D.setColor(this.colors[index]);

        Line2D line = new Line2D.Double(x0, y0, x1, y1);
        g2D.draw(line);
      } // for
    } // for

    // Make the remaining edges (line segments) needed to define
    // the complete graph on the inside set of points.
    g2D.setStroke(this.insideStroke);
    for (int i = 0; i < this.numberOfSides; i++) {
      Vector2D u = inside[i];
      u = u.rotateScaleTranslate(angle, scaleX, scaleY, deltaX, deltaY);
      double x0 = u.getX();
      double y0 = u.getY();
      for (int j = i + 1; j < this.numberOfSides; j++) {
        Vector2D v = inside[j];
        v = v.rotateScaleTranslate(rotation, scaleX, scaleY, deltaX, deltaY);
        double x1 = v.getX();
        double y1 = v.getY();

        // Make color a function of the line segment's length.
        // (The segment is longer if it connects vertices whose
        // indices differ more.)
        int index = Math.abs(i - j);
        g2D.setColor(this.colors[index]);

        Line2D line = new Line2D.Double(x0, y0, x1, y1);
        g2D.draw(line);
      } // for
    } // for
  } // paintComponent( Graphics )

  @Override
  public void actionPerformed(ActionEvent e) {
    this.outerAngle += this.outerStep;
    this.innerAngle += this.innerStep;
    this.repaint();
  } // actionPerformed( ActionEvent )
} // CatsCradlePanel
```

22.3 Vector2D.java

```
package vector2d;

/**
 * This class models a vector in two dimensions.
 *
 * <p>
 * Like a length (feet and inches), weight (pounds and ounces), time (hours and
 * minutes), fraction (numerator and denominator), a complex number (real and
 * imaginary parts), a 2D vector is a thing with two parts that can be added.
 * </p>
 *
 *
 * <p>
 * Complete the definitions of the constructor and the following methods:
 * </p>
 * <ul>
 * <li>add() </li>
 * <li>scale() </li>
 * <li>scale() </li>
 * <li>rotate() </li>
 * <li>dot() </li>
 * <li>magnitude() </li>
 * <li>getX() </li>
 * <li>getY() </li>
 * <li>toString() </li>
 * </ul>
 *
 * <img src="../../../src/vector2d/Vector2D.png" alt="UML class diagram">
 *
 * @author CSC140 Foundations of Computer Science
 * @version 6 March 2013
 */
public class Vector2D {

    private double x;
    private double y;

    /**
     * A vector can represent a point.
     *
     * @param x is the x coordinate of the point.
     * @param y is the y coordinate of the point.
     */
    public Vector2D(double x, double y) {
        this.x = x;
        this.y = y;
    } // Vector2D( double, double )

    /**
     * The addition of one vector to another produces a third vector.
     *
     * <p>
     * The components of the sum are the sums of the corresponding components of
     * the vectors we add.
     * </p>
     *
     * <p>
     * u = (ux, uy) <br>
     * v = (vx, vy) <br>
     * u + v = (ux + vx, uy + vy) <br>
     * </p>
     *
     * @param v is the vector to be added to this one.
     * @return the sum of this vector and the other vector.
     */
    public Vector2D add(Vector2D v) {

        return new Vector2D(0.0, 0.0);
    } // add (Vector2D )

    /**
     * Multiplication of the components of a vector stretches the vector.
     *
     * <p>
     * Scaling u = (u_x, u_y) by (xFactor, yFactor) produces a new vector:
     * (xFactor * ux, yFactor * uy).
     * </p>
     *
     * @param xFactor is the amount of horizontal stretching.
     * @param yFactor is the amount of vertical stretching.
     * @return
     */
    public Vector2D scale(double xFactor, double yFactor) {

        return new Vector2D( 0.0, 0.0 );
    } // scale( double )

    /**
     * We will often want to stretch a vector the same amount in the horizontal
     * and vertical directions.
     *
     * <p>
     * Scaling u = (ux, uy) by factor produces a new vector: (factor * ux,
     * factor * uy).
     * </p>
     *
     * <p>
     * If we think of a vector as an arrow, this operation produces a new vector
     * that has the same (or opposite when the scale factor is negative)
     * direction as the original but a different length.
     * </p>
     *
     * @param factor is the amount of stretching.
     * @return a stretched (or contracted) vector.
     */
    public Vector2D scale(double factor) {

        return new Vector2D( 0.0, 0.0 );
    } // scale( double )

    /**
     * If we think of a vector as an arrow rooted at the origin, then we can
     * imagine rotating it like the hand on a clock.
     *
     * <p>
     * Rotating a vector whose components are (ux, uy) by psi radians produces a
     * new vector whose components (x, y) are as follows:
     * </p>
     *
     * @param angle is the amount of rotation.
     * @return is a rotated version of this vector.
     */
    public Vector2D rotate(double angle) {
        double sine = Math.sin(angle);
        double cosine = Math.cos(angle);

        double xCoord = cosine * this.getX() - sine * this.getY();
        double yCoord = sine * this.getX() + cosine * this.getY();

        return new Vector2D( xCoord, yCoord );
    } // rotate( double )

    /**
     * We will often want to rotate, scale, and translate (move) a vector in
     * that order.
     *
     * @param angle is the amount of rotation.
     * @param scaleX is the amount of horizontal stretching.
     * @param scaleY is the amount of vertical stretching.
     * @param deltaX is the distance moved in the horizontal direction.
     * @param deltaY is the distance moved in the vertical direction.
     * @return is a rotated, scaled, and translated version of this vector.
     */
    public Vector2D rotateScaleTranslate(double angle, double scaleX,
            double scaleY, double deltaX, double deltaY) {

        Vector2D u = this.rotate(angle);
        Vector2D v = u.scale(scaleX, scaleY);

        return new Vector2D(v.x + deltaX, v.y + deltaY);
    } // rotateScaleTranslate( double, double, double, double, double )
```

```java

    /**
     * The dot product of this vector with another vector gives us a number.
     *
     * <p>
     * u = (ux, uy) <br>
     * v = (vx, vy) <br>
     * u * v = ux * vx + uy * vy <br>
     * </p>
     *
     * <p>
     * This number is the length of the one vector's projection on the other
     * (the length of its shadow).
     * </p>
     *
     * @param v is the other vector in the dot product with this vector.
     * @return is the sum of the products of the vector's corresponding
     * components.
     */
    public double dot(Vector2D v) {

        return 0.0;
    } // dot( Vector2D )

    /**
     * The magnitude of a vector is its length.
     *
     * <p>
     * u = (ux, uy) <br>
     * |u| = sqrt( ux^2 + uy^2 ) <br>
     * = sqrt( u * u ) <br>
     * </p>
     *
     * <p>
     * The magnitude is also the square root of the dot product of the vector
     * with itself. This is just the same rule we use to find the length of the
     * hypotenuse of a triangle.
     * </p>
     *
     * @return the length of the vector.
     */
    public double magnitude() {

        return Math.sqrt(this.dot(this));
    } // magnitude()

    /**
     * Here's an accessor method to allow a read-only view of the vector's x
     * component from methods in other classes.
     *
     * @return the value of the vector's x component.
     */
    public double getX() {
        return this.x;
    } // getX()

    /**
     * Here's an accessor method to allow a read-only view of the vector's y
     * component from methods in other classes.
     *
     * @return the value of the vector's y component.
     */
    public double getY() {
        return this.y;
    } // getY()

    /**
     * Here's a way to produce a printable representation of the vector.
     *
     * @return the components of the vector in parentheses and separated by a
     * comma.
     */
    @Override
    public String toString() {

        return "(" + this.getX() + ", " + this.getY() + ")";
    } // toString()

    /**
     * More than one class in a program can have a <code>main()</code> method.
     *
     * <p>
     * <code>main()</code> methods are a convenient place to put code that tests
     * the constructors and methods of a class.
     * </p>
     *
     * @param args is an array that we do not use but must include in every
     * <code>main()</code> method.
     */
    public static void main(String[] args) {
        Vector2D a = new Vector2D(3, 4);
        Vector2D b = new Vector2D(5, 12);

        System.out.println("magnitude of a " + a.magnitude());

    } // main( String [] )
} // Vector2D
```

23 December 6: Work flow

Version control software like Git and GitHub makes collaboration easier.

Here's how a project might develop:

- I create a new project on my own computer.

- I initialize (**init**) a new repository on my computer. I create source code and add it to the repository. I **commit** these changes.

- I open a Web browser. I log into my GitHub account. I create a new, empty repository on GitHub.

- Back on my own computer (and working in an IDE like NetBeans), I **push** my local repository to the remote repository on GitHub.
- I continue to write and test code on my computer. Periodically, I **commit** changes in the local repository and **push** those changes from the local repository to the remote repository.
- You discover my project on GitHub. You decide that you want to use my program, experiment with it, and maybe improve it by adding features or fixing bugs.
- You use GitHub's **fork** command to copy the repository in my GitHub account to your GitHub account. Your GitHub account now has a repository that contains a copy of my code plus a knowledge of the source of that code. (The source is, of course, my GitHub account.)
- On your own computer, you now **clone** your new repository. You find the clone command in your IDE (e.g., NetBeans). Cloning copies the code from your GitHub account to your own computer. The repository on your computer now contains the code plus a knowledge of its source—your GitHub repository.
- You edit the code on your computer. Periodically, you **commit** your changes and **push** those changes to the remote repository on your GitHub account.
- When you are ready, you might then send me a **pull request** from your GitHub account. This tells me that you have made some changes that you want me to review.
- I log into my GitHub account. I see your pull request. I inspect your code. It looks good! Now I can **pull** your contribution from your GitHub repository into my GitHub repository. Now the published code contains my original contribution plus your improvements.

24 December 6: How to describe a curve

24.0.1 Line

24.0.2 Explicit formulation

$y = m \cdot x + b$

24.0.3 Implicate formulation

$a \cdot x + b \cdot y + c = 0$

24.0.4 Parametric formulation

$x(t) = t \cdot x_1 + (1 - t) \cdot x_0$

$y(t) = t \cdot y_1 + (1 - t) \cdot y_0$

24.1 Other parametric curves

24.1.1 Circle

$$x(\phi) = x_c + r \cdot \cos\phi$$
$$y(\phi) = y_c + r \cdot \sin\phi$$

24.1.2 Ellipse

$$x(\phi) = x_c + a \cdot \cos\phi$$
$$y(\phi) = y_c + b \cdot \sin\phi$$

24.1.3 Lissajous figure

$$x(\phi) = x_c + r \cdot \cos(a \cdot \phi)$$
$$y(\phi) = y_c + r \cdot \cos(b \cdot \phi)$$

24.2 Where to look for other formulae

- Famous curves index

25 December 6: Where to learn more

25.1 GitHub tutorials

- (from GitHub) Follow this Hello World exercise to get started with GitHub.
- (from David J. Castner) GitHub & GIT Tutorial
- (from W3Schools) Git Tutorial
- Version Control with Git
- Git Tutorial

26 December 6: How to get started on paper

- Post a report on your progress each day this week.
- Share a prospectus or abstract by Tuesday, December 7.
- Share a bibliography by Wednesday, December 8. Aim for about 4 sources for this exercise. These may be (and probably will be) online sources.
- Share a rough draft of your paper by Friday, December 10.
- Begin by reading and taking notes. As we did last week, avoid copying verbatim. Where you do repeat an unusual word or phrase in your notes, annotate with a description (for example, title and page number) of your source.

27 December 6: Fractal program

27.1 FractalSet.java

```java
package com.eonsahead.fractalset;

import java.awt.Container;
import javax.swing.JFrame;

public class FractalSet extends JFrame {

    private static final int FRACTAL_WIDTH = 768;
    private static final int FRACTAL_HEIGHT = 768;
    private static final String FRACTAL_TITLE = "Fractal";

    public FractalSet() {
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(FRACTAL_WIDTH, FRACTAL_HEIGHT);
        this.setTitle(FRACTAL_TITLE);

        Container pane = this.getContentPane();

        FractalSetPanel panel = new FractalSetPanel();
        pane.add(panel);
        this.setVisible(true);
    } // FractalSet()

    public static void main(String[] args) {
        FractalSet fractalSet = new FractalSet();
    } // main( String [] )

} // FractalSet
```

27.2 FractalSetPanel.java

```java
package com.eonsahead.fractalset;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.AffineTransform;
import java.awt.image.BufferedImage;
import java.awt.image.WritableRaster;
import javax.swing.JPanel;

public class FractalSetPanel extends JPanel {

    private static final int BITMAP_WIDTH = 1024;
    private static final int BITMAP_HEIGHT = 1024;

    private BufferedImage image;

    public FractalSetPanel() {
        this.setBackground(Color.CYAN);
        int imageType = BufferedImage.TYPE_INT_RGB;
        int w = BITMAP_WIDTH;
        int h = BITMAP_HEIGHT;
        this.image = new BufferedImage(w, h, imageType);
    } // FractalSetPanel()

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2D = (Graphics2D) g;

        int w = this.getWidth();
        int h = this.getHeight();

        AffineTransform scale = new AffineTransform();
        scale.setToScale(((double) w) / BITMAP_WIDTH,
                ((double) h) / BITMAP_HEIGHT);

        WritableRaster raster = this.image.getRaster();

        int[][] palette = new int[64][3];

        Color startColor = Color.RED;
        int r0 = startColor.getRed();
        int g0 = startColor.getGreen();
        int b0 = startColor.getBlue();

        Color endColor = Color.BLUE;
        int r1 = endColor.getRed();
        int g1 = endColor.getGreen();
        int b1 = endColor.getBlue();

        for (int i = 0; i < 64; i++) {
//            double fraction = ((double) i) / 63;
//            int red = (int) ((1 - fraction) * r0 + fraction * r1);
//            int green = (int) ((1 - fraction) * g0 + fraction * g1);
//            int blue = (int) ((1 - fraction) * b0 + fraction * b1);

//            palette[i][0] = (int) (256 * Math.random()); //red;
//            palette[i][1] = (int) (256 * Math.random()); //green;
//            palette[i][2] = (int) (256 * Math.random()); //blue;
            switch (i % 4) {
                case 0:
                    palette[i][0] = 12;
                    palette[i][1] = 248;
                    palette[i][2] = 248;
                    break;
                case 1:
                    palette[i][0] = 248;
                    palette[i][1] = 248;
                    palette[i][2] = 12;
                    break;
                case 2:
                    palette[i][0] = 248;
                    palette[i][1] = 12;
                    palette[i][2] = 248;
                    break;
                case 3:
                    palette[i][0] = 248;
                    palette[i][1] = 192;
                    palette[i][2] = 248;
                    break;
                }
        } // for

        int[] blue = {0, 0, 255};
        int[] yellow = {255, 255, 0};

        double xMin = 0;
        double xMax = BITMAP_WIDTH - 1;
        double yMin = 0;
        double yMax = BITMAP_HEIGHT - 1;

        double uMin =  0.385; // 0.25;
        double uMax =  0.395; // 0.50;
        double vMin =  0.375; // 0.25;
        double vMax =  0.385; // 0.50;

        for (int row = 0; row < BITMAP_HEIGHT; row++) {
            double y = row;
            for (int column = 0; column < BITMAP_WIDTH; column++) {
                double x = column;

                double u = uMin + (uMax - uMin) * (x - xMin) / (xMax - xMin);
                double v = vMin + (vMax - vMin) * (y - yMin) / (yMax - yMin);

                Complex z = new Complex(0.0, 0.0);
                Complex c = new Complex(u, v);

                int count = 0;

                while (z.magnitudeSquared() < 4.0 && count < 64) {
                    // z = z^2 + c
                    z = z.multiply(z);
                    z = z.add(c);
                    count++;
                } // while

                if (count == 64) {
                    raster.setPixel(row, column, yellow);
                } // if
                else {
                    raster.setPixel(row, column, palette[count]);
                } // else

//                  if( row < column ) {
//                      raster.setPixel( row, column, yellow );
//                  } // if
//                  else {
//                      raster.setPixel( row, column, blue);
//                  } // else
            } // for
        } // for

        g2D.drawImage(image, scale, this);
    } // paintComponent( Graphics )
} // FractalSetPanel
```

27.3 Complex.java

```
package com.eonsahead.fractalset;


public class Complex {
    /*
     TO-DO: Define a class that models
     complex numbers.

     The class will have:
       instance variables
       a single constructor
       a method named add() for adding the complex number to
         another complex number
       a method named multiply() for multiplying the
         complex number times another complex number
       a method named magnitudeSquared() for computing
         the square of the complex number's magnitude

     Here is the mathematics that you need to know:

       Let i be the square root of -1.
       What follows is mathematical notation.
       In your program there will be no variable
       named i and no variable that holds the square
       root of -1!
       The instance variables of the Complex class
       will both hold floating point values and
       will both have the type double.

       Let u be a complex number with a real part
       equal to a and an imaginary part equal to b.
       Both a and b are floating point values.
       u = (a + bi)

       Let v be a complex number with a real part
       equal to c and an imaginary part equal to d.
       Both c and d are floating point values.
       v = (c + di)

       Then the sum of the two complex numbers is:

       u + v = (a + c) + (b + d)i

       The sum is a complex number whose real part
       is (a + c) and whose imaginary part is (b + d).

       And the product of the two complex numbers is:

       u * v = (a + bi)(c + di)
             = ac + bi * di + adi + cbi
```

```
                = (ac - bd) + (ad + cb)i

        The product is a complex number whose real part
        is (ac - bd) and whose imaginary part is (ad + cb).

        The square of the magnitude of the complex number
        u = a + bi is a * a + b * b.
    */


} // Complex
```

28 December 7:

We are going to write a Java class that models a 2×2 real matrix.

First, a little review of mathematics, then some suggestions of how to translate the mathematics into Java code.

Let **A** and **B** be 2×2 matrices:

$$\mathbf{A} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}$$
$$\mathbf{B} = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

Let **AB** be the product **C** of the two matrices:

$$\begin{aligned} \mathbf{AB} &= \mathbf{C} \\ &= \begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} \\ &= \begin{bmatrix} (a_{00} \cdot b_{00} + a_{01} \cdot b_{10}) & (a_{00} \cdot b_{01} + a_{01} \cdot b_{11}) \\ (a_{10} \cdot b_{00} + a_{11} \cdot b_{10}) & (a_{10} \cdot b_{01} + a_{11} \cdot b_{11}) \end{bmatrix} \end{aligned}$$

Element c_{ij} of the product **C** can be expressed as the dot product of a row of **A** with a column of **B**:

$$c_{ij} = a_{i0} \cdot b_{0j} + a_{i1} \cdot b_{1j}$$

Let's define a Java class that models a 2×2 matrix.

Our class will include...

- instance variable(s)
- a constructor
- a method for computing the product of the matrix with another matrix
- a method that creates a printable representation of the matrix

The header of this class will look something like this...

```java
public class Matrix2x2 {

}
```

Of course, you could choose to give the class a different name.

We could define the instance variables this way. . .

```java
  private final double m00;
  private final double m01;
  private final double m10;
  private final double m11;
```

Or you could define the instance variables this way. . .

```java
  private final double [][] m = new double[3][3];
```

Again, you may choose different names for your instance variables.

You need not make the instance variables **final** (but think about reasons for making the choice one way or another).

Let's define one constructor for the class. . .

```java
  public Matrix2x2( double m00, double m01,
      double m10, double m11 ) {

    // Assignment statements needed here to assign values
    // to the instance variable(s).

  } // Matrix2x2( double, double, double, double )
```

Let's define a method that produces a printable representation of the matrix. . .

```java
  @Override
  public String toString() {

    // TO-DO: Replace "some string" with something
    // more meaningful.

    return "some string";
  } // toString()
```

What might a printable representation of a matrix look like?

Let's say we have a matrix **M**. . .

$$\mathbf{M} = \begin{bmatrix} 11 & 13 \\ 17 & 19 \end{bmatrix}$$

You might make this string from the matrix. . .

```
  [[11, 13], [17,19]]
```

Or maybe you have a better idea?

The method that multiplies matrices will have one parameter. Its type will be Matrix2x2. It will return to its caller an instance of the Matrix2x2 class.

```
  public Matrix2x2 multiply( Matrix2x2 other ) {

    // TO-DO: Write code that computes a, b, c, and d.

    return new Matrix2x2( a, b, c, d );
  } // multiply( Matrix2x2 )
```

If you chose to use 4 floating point variables to hold the values of the array's elements, then you can compute the product using 4 assignment statements...

```
  double a = this.m[0][0] * other.m[0][0] +
    this.m[0][1] * other.m[1][0];

  double b = this.m[0][0] * other.m[0][1] +
    this.m[0][1] * other.m[1][1];

  double c = this.m[1][0] * other.m[0][0] +
    this.m[1][1] * other.m[1][0];

  double d = this.m[1][0] * other.m[0][1] +
    this.m[1][1] * other.m[1][1];
```

If you choose to use a two-dimensional array to hold the values of the array's elements, then you can compute the product using 3 nested **for** loops...

```
  double [][] product = new double[2][2];

  // this approach will work for matrices of
  // any size (not just 2 x 2 matrices)

  for( int i = 0; i < 2; i++ ) {
    for( int j = 0; j < 2; j++ ) {
      product[i][j] = 0.0;
      for( int k = 0; k < 2; k++ ) {
        product[i][j] += this.m[i][k] * other.m[k][j];
      } // for
    } // for
  } // for

  // the product array now contains the values
  // of a, b, c, and d
  double a = product[0][0];
  double b = product[0][1];
  double c = product[1][0];
  double d = product[1][1];
```

Give the class a main() method. Put code that tests the multiply() method in main().

Confirm that the method produces these results...

$$\begin{bmatrix} \cos\frac{\pi}{4} & -\sin\frac{\pi}{4} \\ \sin\frac{\pi}{4} & \cos\frac{\pi}{4} \end{bmatrix} \begin{bmatrix} \cos\frac{\pi}{4} & -\sin\frac{\pi}{4} \\ \sin\frac{\pi}{4} & \cos\frac{\pi}{4} \end{bmatrix} = \begin{bmatrix} \cos\frac{\pi}{2} & -\sin\frac{\pi}{2} \\ \sin\frac{\pi}{2} & \cos\frac{\pi}{2} \end{bmatrix} \tag{1}$$

$$\begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \tag{2}$$

$$\begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{3}$$

Equations (1) and (2) are two different ways of expressing the same equality. Equation (2) contains the numerical values of the sines and cosines in equation (1).

The two matrices on the left sides of equations (1) and (2) represent rotations by $\pi/4$ radians. The matrix on the right side represents a rotation by $\pi/2$ radians.

The equations say: a rotation by $\pi/4$ (45°) followed by another rotation by $\pi/4$ radians is the same as a single rotation by $\pi/2$ radians (90°).

Equation (3) shows the product of two scaling operations. Shrinking something to half of its original size, and then doubling the size of that smaller object has the same effect as doing nothing at all!

29 December 7: 2 meanings of "class" in Java

Class can mean a collection of **static** methods

- the programs that we wrote in the first week of the course defined this kind of a class
- those programs defined **static** methods for finding the minimum value in a list, finding the position of the minimum value, sorting the list, and so on
- we never created an instance of the class
- the class was simply a convenient holder of related functions
- another example is Java's **Math** class
- the **Math** class holds functions for computing absolute values, square roots, sines and cosines, logarithms, and so on
- (the **Math** class also contains definitions of constants: π and e)
- we never need to make an instance of the **Math** class—we do not need multiple math objects—we all live and work with the same math!
- never write a statement like Math myMath = **new** Math()—that does not make sense!
- to call a method in this kind of class, write the name of the class, a period, and the name of the method: **double** x = Math.sqrt(2.0);

Class can also mean a blueprint

- use the blueprint to create instances of the class
- "instance of class" and "object" mean the same thing
- example: our Vector2D class or our 'Weight' class
- we can create many vectors in program that draws a picture

- we can create many weights in a program adds items at the checkout counter in a supermarket
- to call a method in a class of this kind, we write the name of an instance of the class, a period, and then the name of the method

```java
Weight a = new Weight( 1, 10 );
Weight b = new Weight( 2, 8 );
Weight sum = a.add( b );
```

A complication. . . we can define classes that have **static** methods and non-static methods. We call the two kinds of methods in two different ways. We can also include both **static** and non-static variables in a class.

For example, in our weight class, we defined a **static** variable. So we could write **int** c = Weight.OUNCES_IN_A_POUND.

30 December 7: Scope and lifetime of variables

Let's suppose that we are reading a mystery novel. In chapter 4, the hero and heroine pursue the suspect to a distant city. After landing at the airport, they climb into a taxi. During the drive to their hotel, the taxi driver tells them about a recent encounter with someone who looked very much like the object of their pursuit.

Our heroes met the taxi driver in chapter 4. They never see the driver again after climbing out of the taxi. The book made no mention of the taxi driver in chapters 1–3. The book makes no mention of the taxi driver in any of the chapters that follow chapter 4.

The scope of the taxi driver is chapter 4.

Variables in a computer program also have scope. A variable might appear only in one class or one method.

The author of the mystery *could* bring the taxi driver back into the story in the final chapter of the book. There are no hard and fast rules in fiction writing. Certainly, there is no mechanism for enforcing rules. Authors of fiction can do what they want. Although this freedom allows authors to confuse their readers, the best authors exercise discipline. They make and follow their own rules. They avoid frustrating readers with unnecessary surprises.

The designers of programming languages have decided that it is better to have rules and mechanisms for enforcing rules than it is to give programmers unbounded freedom. We cannot count on every programmer working with self-imposed discipline.

How and where a programmer declares a variable (that is, how and where the programmer introduces the variable into the story) will determine the variable's scope. The compiler will prevent the programmer from referring to a variable outside of its scope.

Variables also have lifetimes. A variable might be "born" midway through the execution of a program and might "die" before the program completes its execution. Recall that a variable is a named location in the computer's memory. A variable is "born" when the system allocates a block of memory and gives that block of memory a name. It "dies" when the system releases that block of memory, freeing it for other uses.

Attempts to retrieve a value from a block of memory that no longer belongs to the program will also trigger an error, but a different kind of error than the one triggered by attempts to refer to a variable outside of its scope. The compiler can detect violations of scope (and does so before execution of the program begins) but not references to freed cells in the memory. The runtime system detects references to freed cells. It does this during the execution of the program.

The scope of a variable tells us **where** in a program we may refer to a variable.

The lifetime of a variable tells us **when** during a program's execution we may refer to a variable.

- Can you think of good reasons for limiting the scopes of variables?
- Can you think of good reasons for limiting the lifetimes of some variables?

- Would it not be nice if novels included tables that told readers where each character first enters the story? Maybe programmers have something to tell novelists about how to write stories that are easier to follow!

31 December 7: Instance variables, parameters, and local variables

In this example...

```
  \item \emph{pounds} and \emph{ounces} are instance variables
  \item \emph{lbs}, \emph{oz}, and \emph{other} are parameters
  \item \emph{totalPounds} and \emph{totalOunces} are local variables
```

The **scope** of an instance variable is the whole class. A programmer can refer to an instance variable from within any of the class' constructors or methods.

The **scope** of a local variable or a parameter is the constructor or method in which it is declared. A programmer can refer to a local variable or parameter only from within a single constructor or method.

A local variable or a parameter is "born" when the program calls the method in which the variable or parameter is declared. It "dies" when that method finishes its work and control returns to the caller. Its **lifetime** therefore is the same as the lifetime of the method.

An instance variable is "born" when the program creates an instance of the class. The program will store references to an instance of the class in one or more other variables. For example...

```java
Weight peaches = new Weight(2, 4);
```

Now *peaches* holds a reference (think "address") of an instance of the *Weight* class. When *peaches* and all other variables that hold references to this instance of the class "die", so does the instance of the class and its instance variables. A program called the "garbage collector" watches your running programs. It will notice when all references to an instance of a class have disappeared. Then it will dispose of that instance of the class.

```java
public class Weight {
  private final int pounds;
  private final int ounces;

  public Weight( int lbs, int oz ) {
    this.pounds = lbs;
    this.ounces = oz;
  } // Weight( int, int )

  public Weight add( Weight other ) {
    int totalPounds = this.pounds + other.pounds;
    int totalOunces = this.ounces + other.ounces;

    return new Weight( totalPounds + totalOunces/16,
        totalOunces % 16);
} // Weight
```

32 December 8: Review of version control

32.1 Basics

- Git is a distributed version control system
- "distributed" means every member of the team has a complete copy of the program

- members of the team can work in parallel
- members of the team can work off-line
- software can merge contributions from 2 or more members of the team
- merging is the hard, complicated part of version control
- potential for ”merge conflict”
- some merges are automatic—no human intervention required
- some merges require a person to decide which of the proposed changes to a program we want to keep

32.2 Learning how to use version control

- Step 0: one person on one computer
- Step 1: one person, a computer, and a remote repository
- Step 2: more than one programmer, all with their own computers, and a remote repository through which they share their work

32.3 Version control commands

- init
- add
- commit
- push
- clone (get a copy from a remote repository to my computer)
- fork (get a copy from a repository on someone else’s GitHub account to my GitHub account) (a GitHub command)
- pull request (a GitHub command)
- pull
- merge
- status (a way to see which changes have not yet been committed)
- log (a way to view a history of our project)

33 December 8: Mapping points

33.1 Rectangles in two coordinate systems

Programmers often define geometry in a world coordinate system that the programmers have defined for their own convenience. Maybe this world coordinate systems allows programmers to specify the locations and dimensions of very large objects in the physical world without scaling. The programmers can enter measurements made in the real world into their programs. Maybe the programmers are modelling very small objects. In that case, the units of measurement might be nanometers rather than kilometers. Maybe the right choice of a coordinate system will make the arithmetic easier. For example, calculations might be easier if the origin of the coordinate system is at the center of the figures that the computer will model and draw.

Programmers then need a way to translate between world coordinates and device coordinates (the coordinate system on which the programmer locates pixels on the computer's screen).

Let's suppose that we have a world coordinate system in which the horizontal and vertical axes are labeled x and y.

Let's suppose that we have labeled the axes in the device coordinate system u and v.

Let $(x_{min}, x_{max}, y_{min}, y_{max})$ define the boundaries of a rectangle in world coordinates.

- $x_{min} < x_{max}$
- $y_{min} < y_{max}$
- $x = x_{min}$ is the equation of a vertical line that defines the left boundary of the rectangle
- $x = x_{max}$ is the equation of a vertical line that defines the right boundary of the rectangle
- $y = y_{min}$ is the equation of a horizontal line that defines the bottom boundary of the rectangle
- $y = y_{max}$ is the equation of a horizontal line that defines the top boundary of the rectangle

Similarly, let $(u_{min}, u_{max}, v_{min}, v_{max})$ define the boundaries of another rectangle in device coordinates.

- $u_{min} < u_{max}$
- $v_{min} < v_{max}$
- $u = u_{min}$ is the equation of a vertical line that defines the left boundary of the rectangle
- $u = u_{max}$ is the equation of a vertical line that defines the right boundary of the rectangle
- $v = v_{min}$ is the equation of a horizontal line that defines the bottom boundary of the rectangle
- $v = v_{max}$ is the equation of a horizontal line that defines the top boundary of the rectangle

We want a way of establishing a correspondence between points in the two coordinate systems. For example, a point in the center of the world coordinate system will correspond to a point in the the center of the device coordinate system. A point in the lower left corner of one system will correspond to a point in the lower left corner of the other system. A point that is a third of the way from the left boundary to the right boundary of the rectangle that we defined in the world system will correspond to a point that is a third of the the way across in the device system.

Let (x, y) be a point in the world system and (u, v) be the corresponding point in the device system.

Then...

$$
\begin{aligned}
u &= u_{min} + \frac{(x - x_{min})}{(x_{max} - x_{min})} \cdot (u_{max} - u_{min}) \\
v &= v_{min} + \frac{(y - y_{min})}{(y_{max} - y_{min})} \cdot (v_{max} - v_{min})
\end{aligned}
$$

34 December 9: Data classes

A programmer might want a class only to hold data. Such a class will have instance variables, a constructor, and getters (accessor methods), but no methods that execute algorithms. Such a class is a "data class."

Picture in your mind's eye a rectangle whose edges are all parallel to one of the coordinate axes. We can describe the rectangle with just four numbers:

- the x coordinate of the rectangle's left boundary

- the x coordinate of the rectangle's right boundary
- the y coordinate of the rectangle's bottom boundary
- the y coordinate of the rectangle's top boundary

- Can you define a data class that models this kind of rectangle?
- Can you define a second class whose instance variables are two rectangles? Call these two instance variables *world* and *device*.

We will see how to use these classes in a program that draws Moiré; patterns. (Search on the Web with the word "moire" to see some of the many patterns that are possible.)

Make the class that holds the two rectangles a data class now. Later, we will add a method that, given a point in the first rectangle, finds a corresponding point in the second rectangle.

35 December 9: Rays of the sun or spokes of a wheel

Learn how to use the Point2D and Line2D classes of the java.awt.geom package.

Point2D and Line2D are *abstract classes.* This means that they specify some methods without providing definitions of those methods. Their sub-classes (Point2D.Double and Line2D.Double) do fully define how the methods are to do their jobs.

This means that you cannot write...

```
Point2D p = new Point2D( 3.0, 4.0 );
```

...but you can write...

```
Point2D p = new Point2D.Double( 3.0, 4.0 );
```

Can you write code that creates a list of line segments that together resemble the spokes of a wheel or the rays of the sun?

To make the exercise a little simpler, assume that one endpoint of each line segment is $(0.0, 0.0)$ and that the other endpoint is $(\cos\phi, \sin\phi)$, where $0.0 \leq \phi \leq 2\pi$. The angle ϕ will be different for each line segment.

36 December 10: Inheritance and polymorphism

Let's suppose that we have defined a class named 'Employee' and that within that class we have defined a method named 'getPay()':

```
public class Employee {

  // Instance variables not shown

  // Constructors not shown

  public BigDecimal getPay() {
  } // getPay()

  // Other methods not shown
} // Employee
```

We have also defined subclasses:

```java
public class SalariedEmployee  extends Employee {
  // Instance variables, constructors, methods not shown
} // SalariedEmployee

public class HourlyEmployee extends Employee {
  // Instance variables, constructors, methods not shown
} // HourlyEmployee
```

Because ‘SalariedEmployee‘ and ‘HourlyEmployee‘ inherit from ‘Employee‘, they both have a method named ‘getPay()‘.

The two classes can override the definition of the method that they inherited from their parent class. Each will override the definition in a different way, because the company computes the pay of hourly and salaried employees in different ways. However, in each case the new version of the ‘getPay()‘ method must have the same number and type of parameters and the same return type as the method in the parent class.

```java
@Override
public BigDecimal getPay() {
  // Some sequence of statements
  // that will compute the pay (not shown).
} // getPay()
```

We can make a list of employees and add employees to the list.

```java
List<Employee> employees = new ArrayList<>();
employees.add( new SalariedEmployee("Adele Goldberg") );
employees.add( new HourlyEmployee("Peter Denning"));
```

Then I want to print the pay of each employee:

```java
for( Employee e : employees ) {
  System.out.println( e.getPay() );
} // for
```

The run-time system can tell what kind of employee it is taking from the list at each step in this iteration. It will then execute the appropriate version of ‘getPay()‘. This is **polymorphism.**

Polymorphism ("poly" = many, "morphism" = forms) is a principal, defining feature of object-oriented programming. The programmer does not have to write ‘if ‘ statements to distinguish between (or among) different kinds of employees and explicitly tell the computer which version of the ‘getPay()‘ method to execute. Instead, the run-time system (the software that runs programs on the computer) can determine which kind of object it has and call the right version of the method—this happens automatically. This feature reduces the amount of code that the programmer must write, the amount of work that the programmer must do, and the number of details to which the programmer must attend.

36.1 Exercise

- How can you learn more about the ‘BigDecimal‘ class? Where will you go to learn more?
- Why should ‘getPay‘ return to its caller an instance of ‘BigDecimal‘ rather than a ‘double‘?
- Who is Adele Goldberg?
- Who is Peter Denning?

37 December 13: 3 short videos

37.1 Respond to the following questions here on Piazza.

- What parts of the advice given here might help us in our current writing exercise?
  - How to Answer Behavorial Interview Questions | Udacity Career Tip #16
- Which of the points that Mayuko makes in this next video persuaded you?
  - The Most Important Skill in Software Engineering
- What is new or surprising to you in what these writers have to say about their work?
  - Meet Technical Writers at Google

38 December 13: Writing with the English and Java languages

- something is better than nothing
- late is better than never
- start early
- work in small steps
- have something that works / have something that you can present every day!
- spelling and grammar count!
- find another set of eyes to look at your work
- read an English paper out loud (to yourself or, even better, to other people)
- do not expect to fully understand at the outset what your client wants
- whenever possible, keep up an on-going conversation with your clients
- expect the specifications to change
- expect to throw away much of what you write
- write words that you want your readers to see + notes to yourself
- write code that will execute + notes to yourself
- choose software that will help you in this work, learn how to use that software
- separate concerns: content and format (fonts, margins, indentation of paragraphs, placement of page numbers)
- separate concerns: logic/arithmetic and appearance of output
- periodically pause to refactor
  - add no new ideas, factual assertions, logic, arithmetic
  - move elements to produce a more logically ordered document
  - divide large elements (sentences and paragraphs in an English document, classes and methods in a Java program) into smaller elements
  - eliminate redundant elements
- lower quantity / higher quality is a better combination than higher quantity / lower quality
- fewer words and pages might be better!
- less functionality in a computer program might be better!

39 December 13: Programming exercise

1. Complete this program.

```java
package lastexercise;

import java.util.ArrayList;
import java.util.List;

public class LastExercise {

  public static boolean foundMatch( int value, List<Integer> list ) {
    // TO-DO: Complete the definition of this
    // stub method.

    // The completed method will return true its
    // caller if list contains value, and will
    // return false otherwise.

    return false;
  } // foundMatch( int, List<Integer> )

  public static int countMatches( int value, List<Integer> list ) {
    // TO-DO: Complete the definition of this
    // stub method.

    // The completed method will return to its
    // caller the number of times that value
    // appears in list.

    return 0;
  } // countMatches( int, List<Integer> )

  public static void main( String [] args ) {
    List<Integer> data = new ArrayList<>();

    data.add( 34 );
    data.add( 13 );
    data.add( 89 );
    data.add( 55 );
    data.add( 21 );
    data.add( 13 );

    boolean found = foundMatch( 55, data );
    System.out.println( "55 was found in the list: " + found );

    int count = countMatches( 13, data );
    System.out.printf( "Found %2d matches to 13 in the list\n",
      count );

  } // main( String [] )

} // LastExercise
```

2. You previously saw classes that model lengths and weights.

   A length (feet and inches) is a number with two parts.

A weight (pounds and ounces) is also a number with two parts.

In those previous exercises, we defined special rules (methods) for adding lengths in the first case and weights in the second case.

In this exercise, you will again create a class that models a kind of number that has two parts.

Cooks measure fluids and powders with spoons.

Let's define "spoonful" to be a number with two parts: the number of tablespoons and the number of teaspoons.

A tablespoon contains three teaspoons.

- (2 tablespoons, 1 teaspoon) plus (1 tablespoon, 2 teaspoons) equals (4 tablespoons, 0 teaspoons)

Write a program that defines a class that models a spoonful, creates two instances of the Spoonful class, computes their sum, and prints the sum.

The Spoonful class will have...

- instance variables
- a constructor
- a method to add one Spoonful to another Spoonful and return the sum to its caller
- a toString() method that returns a printable representation (a String) to its caller

40 December 14: Exercise

Create a version of this program on your own computer. Follow the hints found in the "TO-DO" comments to make an image that appeals to you.

TSquare.java

```java
package tsquare;

import java.awt.Container;
import javax.swing.JFrame;

public class TSquare extends JFrame {
  // TO-DO: Experiment with different widths,
  // heights, and titles.
  private static final int FRAME_WIDTH = 512;
  private static final int FRAME_HEIGHT = 512;
  private static final String FRAME_TITLE = "TSquare";

  public TSquare() {
    this.setSize( FRAME_WIDTH, FRAME_HEIGHT );
    this.setTitle( FRAME_TITLE );
    this.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

    Container pane = this.getContentPane();
    pane.add( new TSquarePanel() );

    this.setVisible(true);
  } // TSquare()

  public static void main( String [] args ) {
    TSquare tsquare = new TSquare();
  } // main( String [] )

} // TSquare
```

TSquarePanel.java

```java
package tsquare;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Shape;
import java.awt.geom.AffineTransform;
import java.awt.geom.Rectangle2D;
import javax.swing.JPanel;

public class TSquarePanel extends JPanel {
  // TO-DO: Experiment with different colors.
  // TO-DO: Can you produce a more appealing image
  // by using partly transparent colors?
  // Use a fourth parameter in Color's constructor
  // to specify the degree of transparency.
  private static final Color BG_COLOR = new Color( 112, 248, 196 );
  private static final Color FG_COLOR = new Color( 24, 64, 224 );
  // TO-DO: Experiment with different thickness of the lines.
  private static final float THICKNESS = 8.0F;
  // TO-DO: Experiment with different values of THRESHOLD.
  // The value of THRESHOLD determines how far recursion goes.
  private static final double THRESHOLD = 0.8;

  public TSquarePanel() {
    this.setBackground( BG_COLOR );
  } // TSquarePanel()

  @Override
  public void paintComponent( Graphics g ) {
    super.paintComponent( g );
    Graphics2D g2D = (Graphics2D) g;

    int w = this.getWidth();
    int h = this.getHeight();

    // We will define all geometry in a world
    // coordinate systems whose lower left
    // corner is at (x = -1.0, y = -1.0) and
    // whose upper right corner is at
    // (x = +1.0, y = +1.0).

    // Then will use the AffineTransform class
    // and its methods to translate and scale
    // the geometry so that it fits the panel
    //  in which we draw the picture.

    AffineTransform scale = new AffineTransform();
    scale.setToScale( w/2, h/2 );

    AffineTransform translate = new AffineTransform();
    translate.setToTranslation( 1, 1 );

    // Translation is the first operation.
    // Scaling is the second operation.

    AffineTransform transform = new AffineTransform();
    transform.concatenate( scale );
    transform.concatenate( translate );

    g2D.setColor( FG_COLOR );
    g2D.setStroke( new BasicStroke( THICKNESS ) );

    // TO-DO: You may experiment with different
    // values of size. Give it a positive value < 2.0.
    double size = 1.8;
    double xMin = -size/2;
    double yMin = -size/2;
    double xMax = +size/2;
    double yMax = +size/2;

    Rectangle2D square = new Rectangle2D.Double( xMin, yMin, size, size );
    drawSquares( square, g2D, transform );

    Shape shape = transform.createTransformedShape( square );
    g2D.draw( shape );
  } // paintComponent( Graphics )

  public void drawSquares( Rectangle2D square,
       Graphics2D g2D, AffineTransform transform ) {

    // TO-DO: Adding an integer parameter to this
    // method will make it possible to give different
    // colors to different size squares. If interested
    // in trying this, ask for more instructions.

    // Continue dividing square into smaller squares
    // only as long as the square's side exceeds a threshold.

    if( square.getBounds2D().getWidth() > THRESHOLD ) {
      // Divide square into 4 smaller squares (quadrants).

      // lower left corner of big square
      double xMin = square.getX();
      double yMin = square.getY();

      // upper right corner of big square
      double xMax = xMin + square.getWidth();
      double yMax = yMin + square.getHeight();

```

```java
      // center of big square
      double xMiddle = (xMin + xMax)/2;
      double yMiddle = (yMin + yMax)/2;

      // center of northeast quadrant
      double neX = (xMiddle + xMax)/2;
      double neY = (yMiddle + yMax)/2;

      // center of northwest quadrant
      double nwX = (xMin + xMiddle)/2;
      double nwY = (yMiddle + yMax)/2;

      // center of southwest quadrant
      double swX = (xMin + xMiddle)/2;
      double swY = (yMin + yMiddle)/2;

      // center of southeast quadrant
      double seX = (xMiddle + xMax)/2;
      double seY = (yMin + yMiddle)/2;

      // Call drawSquares() recursively on each of
      // square's 4 quadrants.

      Rectangle2D northeast = new Rectangle2D.Double(
        xMiddle, yMiddle, xMax - xMiddle, yMax - yMiddle );
      drawSquares( northeast, g2D, transform );

      Rectangle2D northwest = new Rectangle2D.Double(
        xMin, yMiddle, xMiddle - xMin, yMax - yMiddle );
      drawSquares( northwest, g2D, transform );

      Rectangle2D southwest = new Rectangle2D.Double(
        xMin, yMin, xMiddle - xMin, yMiddle - yMin );
      drawSquares( southwest, g2D, transform );

      Rectangle2D southeast = new Rectangle2D.Double(
        xMiddle, yMin, xMax - xMiddle, yMiddle - yMin );
      drawSquares( southeast, g2D, transform );

      // Create a fifth smaller square whose corners
      // are at the centers of the quadrants.

      // TO-DO: Try creating an Ellipse2D here instead
      // of a Rectangle2D. The parameters for the
      // constructor will be the same. Or you might
      // try a RoundRectangle2D.
      Rectangle2D center = new Rectangle2D.Double(
        swX, swY, neX - swX, neY - swY );
      Shape shape = transform.createTransformedShape( center );

      // TO-DO: Try replacting this next call to draw()
      // with a call to fill().
      g2D.fill( shape );
    } // if

  } // drawSquares( Rectangle2D, AffineTransform )
```

```
} // TSquarePanel
```

41 December 15: Review

1. How many multiplications do we need to computer $2^{16} = 65,536$?

$$2^{16} = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$$

$$\begin{aligned} 2^{16} &= 2^8 \cdot 2^8 \\ 2^8 &= 2^4 \cdot 2^4 \\ 2^4 &= 2^2 \cdot 2^2 \\ 2^2 &= 2 \cdot 2 \end{aligned}$$

2. How might you use what you learned from the previous question to write a method that efficiently computes a^n (where n is a non-negative integer)?
3. In what sense is Java a hybrid language?
4. If you could only have three of Java's primitive types, which would you choose?
5. What is the convention for naming Java classes?
6. In what ways does the header of a constructor differ from the header of a method?
7. What is the purpose of a toString() method?
8. How does inheritance reduce the amount of work a programmer might otherwise have to do?
9. How does polymorphism reduce the amount of work a programmer might otherwise have to do?
10. A Java programming convention calls for making the instance variables in a class **private**. How can a programmer make it possible for statements in other classes to retrieve the values of these instance variables or changes their values?

    Why not just make instance variables **public**?