Encapsulation and Reusing Code

CSE/IT 213

NMT Department of Computer Science and Engineering

"If you're afraid to change something it is clearly poorly designed."

— Martin Fowler

"If you're having trouble succeeding, fail."

— Kent Beck

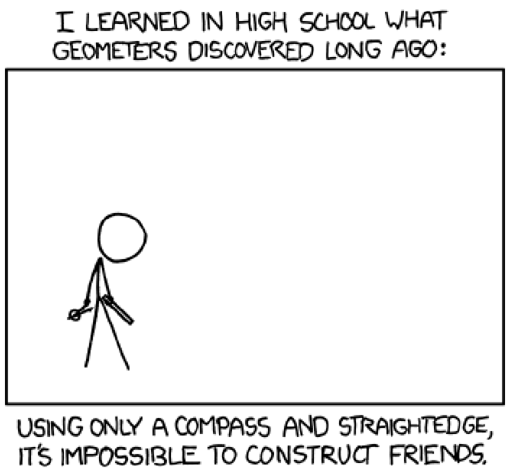

I LEARNED IN HIGH SCHOOL WHAT GEOMETERS DISCOVERED LONG AGO:
USING ONLY A COMPASS AND STRAIGHTEDGE, IT'S IMPOSSIBLE TO CONSTRUCT FRIENDS.

Figure 1: `https://xkcd.com/866/`

Introduction

In this assignment you will revisit the 2D geometry code you wrote in Homework 1, and learn how to make better use of interactions between objects. You will also see some more useful examples of how to use *encapsulation* to maintain the state of your program.

Sharing Code

So far you have been using `import` statements to include code from the Java standard library. However, it is also important to think of *every* Java class as a module that's meant to be imported and re-used in other classes. When programming, if you see that you're *repeating yourself*, it's usually an indication that you're doing something wrong. If you've already solved a problem once, you should be able to re-use that solution, rather than having to solve it again.

When two classes are in the same package, you don't need to import anything for them to be able to see each other; the neighboring classes are already in scope and ready to use. Even when a software project grows to include many different packages, you can still share with yourself code that you've already written. You just need to import the class you want from the package it belongs to:

```java
package oop.student.example;

import oop.student.project.Stuff;

class Example {
    // The class Stuff, and any of its public methods, can be used here in this class
}
```

This is very simple, but planning for code re-use has always been one of the main motivating factors behind object oriented programming.

Delegating Constructors

Another important idea meant to help us stop repeating ourselves is that of **delegating constructors**. In many cases you will have a main constructor that does all the work of setting up your object, and the other constructors essentially need to do the same thing, but starting with different inputs. When you run into this situation, it's possible for one constructor to "call" another using the keyword `this`.

For a silly example, say we've written a class to represent user accounts. And, for whatever reason, we're using this horribly misguided constructor to ensure that every password contain both numbers and capital letters:

```java
public class Account {
    private String username;
    private String password;

    public Account(String username, String password) {
        // No capital letters?
        if (!password.matches(".*[A-Z].*")) {
            password = "ASDF" + password + "XYZ";
        }

        // No numbers?
        if (!password.matches(".*\\d.*")) {
            password = password + "!1234";
        }

        this.username = username;
        this.password = password;
    }
}
```

Figure 2: A terrible password policy

If we wanted to write another constructor to create an `Account` given only a username, and give the new account the default password: `"ASDFXYZ!1234"`[1]. Since most of that work was already done in the previous example, the second constructor only needs to call the first one, like so:

```java
public Account(String username) {
    this(username, "");
}
```

Figure 3: Delegating work to the other constructor using `this()`

Delegating constructors are useful when you already have one main constructor that knows how to assign each of your class's private attributes. Usually every other constructor will be able to delegate its work to the first one, just by passing in the default settings.

If one constructor already knows how to create a circle with `x`, `y`, and `radius`; then calling `this(0, 0, 1)` is a good way for another constructor to create a unit circle centered at the origin.

Encapsulation

In object oriented programming, **encapsulation** refers to the ability of a class to **hide information** from the rest of the program. In the code you've written so far for this class, every class attribute has been marked `private`, meaning that variable cannot be seen from outside the class. However, you've also made these variables fully accessible by adding public *getters* and *setters* for each of them

[1]This is just a toy example. This is definitely *not* how you should actually manage passwords!

[2]. For all intents and purposes, that's no different from marking the attribute as `public` to begin with.

So why do it? Apart from adhering to conventions, the real usefulness of getters and setters is that they make it possible to control *how* attributes are accessed, and what happens when they are updated. Take the following code for example:

```java
public class Student {
    private int age;
    private int idNumber;
    private String firstName;
    private String lastName;

    public int getIdNumber() {
        return idNumber;
    }

    public String getUserName() {
        String username = firstName.charAt(0) + lastName;
        return username.toLowerCase();
    }

    public void setAge(int age) {
        if (this.age < age) {
            this.age = age;
        }
    }
}
```

Figure 4: More selective uses of getters and setters

A few notes on the ways the getters and setters are designed for this class:

1. First, notice that this class has the method `getIdNumber()`, but not `setIdNumber()`. Including a getter but not a setter makes `idNumber` into a *read-only* attribute — you can find out what a student's ID number is, but there's no way to change it once the object has been created. Knowing that a variable can't change can be useful for reasoning about your program; if you notice a bug having to do with that variable, then you can immediately trace the problem back to the constructor call, since that's the only place it could have been assigned.

2. Next there's `getUserName()`, which provides another read-only interface, but this time it's for an attribute that doesn't actually exist. Instead of declaring a new variable for the student's username, you can compute it on the fly every time it is requested. This is obviously much simpler than adding an extra attribute to the class.

3. Finally there's the `setAge()` method. Before allowing the `age` attribute to change, it checks to make sure that updated value will *increase* the student's age. Since this is the only way for

[2]These boilerplate getters and setters can be automatically generated in most IDEs. In IntelliJ you can go to `Code -> Generate -> Getter and Setter` and select which private attributes you want to include. This can save you a lot of tedious typing and let you get on with more important parts of your code.

that attribute to be updated, this tells you that it's impossible to accidentally make students get *younger* within your software system.

When people hear the term "information hiding," they tend to imagine that they're being asked to keep the implementation secret from other programmers. This is missing the point — if another programmer is using your code in the first place, then more often than not they will be perfectly capable of reading the source for themselves.

Rather, the point of putting access controls in a class's interface is to make it easier to reason about its behavior at run time. If you've ever found a bug in your one of your programs and thought, *"that should be impossible!"* then you should appreciate the opportunity to make scenarios like that *actually* impossible! After all, if you're making assumptions about the data underlying your program, it's probably worthwhile to take a few steps to ensure that those assumptions are always correct.

The following code shows another major use-case for using setters to manage class attributes — keeping variables *in sync* with each other:

```java
public class Timer {
    private int seconds;
    private int minutes;
    private int hours;

    public void setSeconds(int seconds) {
        this.seconds = seconds % 60;
        setMinutes(this.minutes + seconds / 60);
    }

    public void setMinutes(int minutes) {
        this.minutes = minutes % 60;
        setHours(this.hours + minutes / 60);
    }

    public void setHours(int hours) {
        this.hours = hours;
    }
}
```

Figure 5: Timer attributes are kept in sync

Here we have a timer with three private attributes, `seconds`, `minutes`, and `hours`. Since it's not possible for a clock to show more than 59 seconds, `setSeconds()` uses the mod operator to ensure the assigned value is less than 60.

However, it also goes a step beyond that by adding every 60-seconds of overflow to the `minutes` counter. This way, if you try to set the timer to 90 seconds it will update *multiple* variables to make the clock read `00:01:30`. Similarly, trying to set the timer to 150 minutes will also modify the `hours` counter so you end up with a time of: `02:30:00`.

Problems

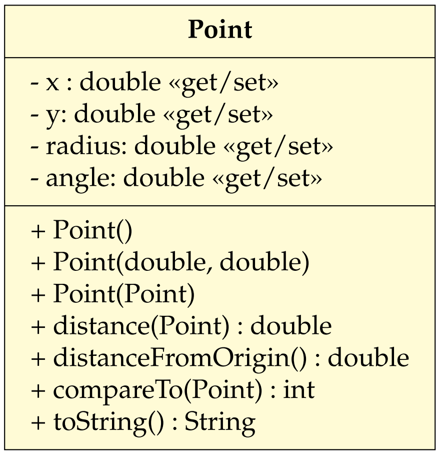
Problem 1: Geometry

In the previous assignment you wrote the classes `Point.java`, `Rectangle.java`, and `Circle.java`. For this assignment you will create modified versions of your original code, and improve upon the overall program.

Point

The first task is to modify `Point.java` so that it also supports polar coordinates. Add the attributes `radius` and `angle` to the class. The `radius` measures the point's distance from the origin, and the `angle` gives the angle (in *radians*, from $-\pi$ to π) of an arc starting on the positive x axis and ending at the point.

For example, the point $(1, 0)$ should have $\theta = 0$, and $(1, 1)$ would have $\theta = \pi/4$.



Point

- x : double «get/set»
- y: double «get/set»
- radius: double «get/set»
- angle: double «get/set»

+ Point()
+ Point(double, double)
+ Point(Point)
+ distance(Point) : double
+ distanceFromOrigin() : double
+ compareTo(Point) : int
+ toString() : String

- Write getters and setters for the private attributes. Write the setters so that the polar coordinates are updated whenever you change x or y, and vice versa.

  For reference, these formulas tell you how to convert between the two coordinate systems:

$$(r, \theta) \Rightarrow (r \cdot \cos\theta, r \cdot \sin\theta)$$
$$(x, y) \Rightarrow \left(\sqrt{x^2 + y^2}, \tan^{-1}(y/x)\right)$$

  You may run into problems trying to use `Math.atan()` to compute the arctangent. When either x or y is negative, then the direction of the angle y/x becomes ambiguous; and if $x = 0$ then dividing by zero will throw an `ArithmeticException`! Luckily, Java has another builtin method which you can use instead:

```
angle = Math.atan2(y, x);
```

  For a concrete example of how the setters should interact with each other, the following code:

```java
Point pt = new Point(1, 1);
pt.setRadius(1);
System.out.println(pt.getX());
System.out.println(pt.getY());
```

Should change the radius, but leave the existing angle ($\pi/4$) the same. But since line 2 changes the point's distance from the origin, it should also move x and y. The resulting coordinate in this case should be ($\sqrt{2}/2, \sqrt{2}/2$), so the program should output something close to this:

```
0.7071067811865476
0.7071067811865475
```

- Update the constructors you wrote in Homework 1, to make sure they also set the new attributes, `radius` and `angle`. To recap:
  1. The default constructor creates the point $(x = 0, y = 0)$, which is also $(r = 0, \theta = 0)$
  2. The second constructor takes two arguments, `x` and `y`, and creates the point (x, y), which is also $\left(\sqrt{x^2 + y^2}, \tan^{-1}(y/x)\right)$
  3. The third constructor is a copy constructor, which creates an exact duplicate of an existing `Point`
- Reimplement the `distance()` and `distanceFromOrigin()` methods from Homework 1. You will most likely not need to change your code, but you're free to do so if you'd like.
- Write the new method `compareTo()`, which will take a second `Point` and compare it to `this` one. The rules for comparing are:
  - If the points have the exact same x and y coordinates, `return 0`
  - If the other `Point` is below **or** to the left of the current point, `return 1`
  - If the other `Point` is above **and** to the right of the current point, `return -1`

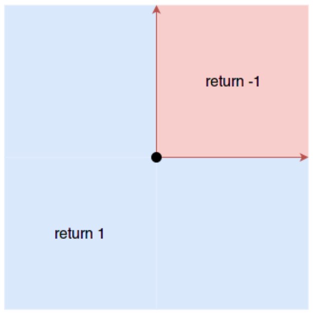


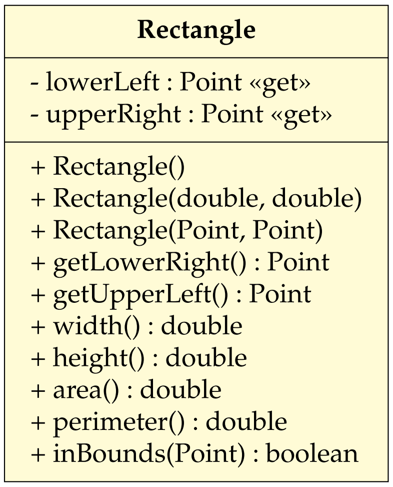
Figure 6: Visualizing the behavior of `compareTo()`

- Finally, add the method `toString()`, which will convert the point to a string displaying the Euclidean coordinates in the format `"(x, y)"`.

`toString()` is a special method in Java which tells instances of a class how to be printed with `System.out.println()`.

Rectangle

Next, you will modify `Rectangle.java` so that the boundary corners are defined by two `Point`'s instead of four `double`'s:


Rectangle
- lowerLeft : Point «get»
- upperRight : Point «get»
+ Rectangle()
+ Rectangle(double, double)
+ Rectangle(Point, Point)
+ getLowerRight() : Point
+ getUpperLeft() : Point
+ width() : double
+ height() : double
+ area() : double
+ perimeter() : double
+ inBounds(Point) : boolean

- Include getters for the lower left and upper right corner points, but *not* setters. Instead of returning the attributes directly, the getters should use `Point`'s copy constructor to return `new` copies of each point. In other words, this version of `Rectangle` should be a totally *read-only* data structure.
- Add two additional getters to the class: `getLowerRight()` and `getUpperLeft()`. You do not need to add new attributes to the class to store these points — instead you can create `new` point's on the fly using the existing corners.
- Update the constructors you wrote in Homework 1:
  1. The default constructor creates a 1×1 square whose lower left corner is on the origin
  2. The second constructor takes two `double`'s, `width` and `height`, and creates a rectangle with one corner touching the origin, and the other corner touching the point $(width, height)$.

     Handle negative inputs by delegating to the final constructor.
  3. The main constructor takes two `Point`'s, `lowerLeft` and `upperRight`, and creates **copies** of them to set the boundary corners of the rectangle. Again, you can do this using `Point`'s copy constructor.

     Compare the two points to make sure that the second is, in fact, above and to the right of the first. If the coordinates are backwards, (i.e., if the second is to the *lower left* of the first), or if they are given in the wrong orientation, (e.g., if the second is to the *upper left* of the first), then construct new points to correspond to the correct corners of the rectangle.

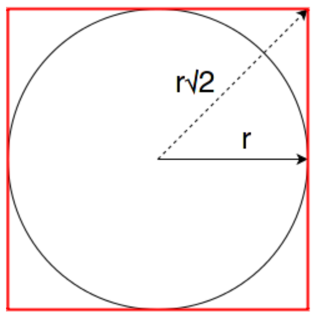
- Update the methods `width()`, `height()`, `area()`, `perimeter()`, and `inBounds()`, to work with points instead of individual coordinates.

Circle

Similarly, update `Circle.java` to use `Point`'s instead of individual coordinates:

Circle
- center : Point «get/set» - radius : double «get/set»
+ Circle() + Circle(Point, double) + diameter() : double + area() : double + perimeter() : double + inBounds(Point) : boolean + getBoundingBox() : Rectangle

- Write getters and setters for both private attributes. Just as in `Rectangle.java`, `getCenter()` should return a new **copy** of the center point, and `setCenter()` should set the center to a **copy** of the given point.
- The default constructor creates a circle with radius 1 centered at the point $(0, 0)$
- The main constructor sets the center to a **copy** of the given point, and the radius to the **absolute value** of the given number.
- Update the methods `diameter()`, `area()`, `perimeter()`, and `inBounds()`, to work with points instead of individual coordinates.
- Add the new method `getBoundingBox()` to generate a new `Rectangle` surrounding the top, bottom, left, and right margins of the circle. There are many ways to do accomplish this, but the overall goal is create the red box pictured below:



Tests

Finally, update `Tests.java` to unit test the changed methods in the program. Again, you should have *at least* one sensible unit test for each nontrivial method in this package. That is:

- At least 11 tests for `Point.java`, (including the new setter methods)
- At least 10 tests for `Rectangle.java`
- At least 6 tests for `Circle.java`

Your tests should check that each method follows the specification given in the assignment. If a method returns a wrong value, use `System.out.println()` to print an error message stating which test failed and why.

Note: the specified behavior of the `Rectangle(width, height)` constructor is different in this assignment than in Homework 1. Instead of setting negative values to 0, it should create the rectangle in the correct quadrant of the xy plane.

Finally, the `Tests` class should include a `main()` method which runs each of the tests you wrote.

You will not lose points for a test that correctly points out an error elsewhere in your code! Since you want to make sure you're turning in good work, you should strive to cover as many edge cases as you can think of.

Problem 2: Temperature

In a new package, write a program that converts between temperatures in different units, Kelvin, Celsius, and Fahrenheit. The package should contain these three classes:

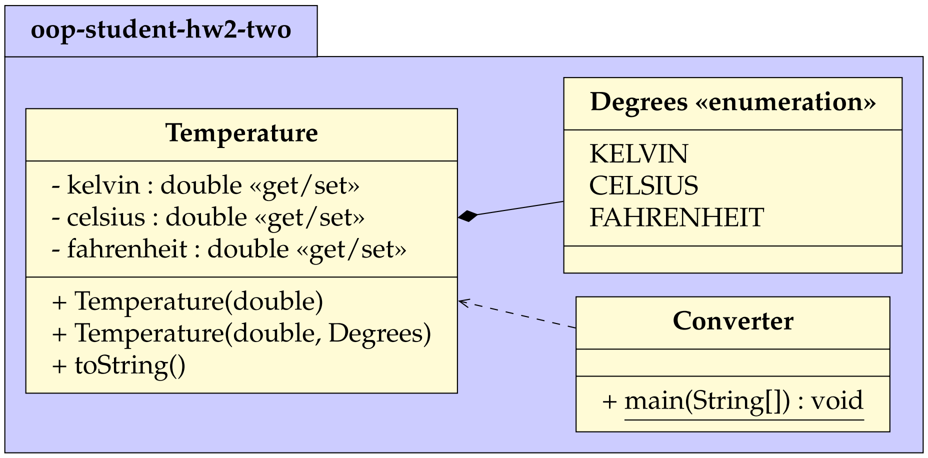


Figure 7: Package diagram — the underlining on main means it is declared `static`

Degrees

Create the file `Degrees.java` to define this simple enumeration:

```java
public enum Degrees {
    KELVIN, CELSIUS, FAHRENHEIT
}
```

Temperature

`Temperature.java` has three private attributes, a `double` for each unit of temperature.

- Write getters and setters for each private attribute. Write the setters so that updating any attribute automatically sets all three to the appropriate values. For reference, these equations tell you how to convert between Kelvin, Celsius, and Fahrenheit:

  $$K = C + 273.15$$
  $$C = \frac{5}{9} \times (F - 32)$$
  $$F = 1.8 \times C + 32$$

  Additionally, if the given value corresponds to a temperature below *absolute zero*, which is 0° Kelvin, the setters should ignore it and do nothing.

- Write one constructor which assumes the temperature is given in degrees Celsius, and set all three values accordingly. If the given temperature is below absolute zero, instead set the temperature to $-273.15^\circ C$.

- The second constructor takes the temperature and an additional argument for the unit. The unit is a member of the `Degrees` enumeration, and indicates which attribute to set directly. Set all three values to the correct temperature, converted from the given unit.

  Again, if the given temperature is below absolute zero, set all three values to the equivalent of $-273.15^\circ C$.

- Finally, add a `toString()` method that displays the temperature in the format `"13.37 C"`, with the degrees Celsius shown to two decimal places.

Converter

Finally, write a `main` class, `Converter.java`. This class contains a very simple program that prompts the user for the current temperature, then for a pair of units — either `"K"`, `"C"`, or `"F"`. If the input temperature cannot be parsed as a `double`, or either unit is invalid, print an error message and prompt again for correct input.

After processing the input, print the converted temperature then exit.

Example output:

```
Temperature Conversion Calculator
=================================

Enter the temperature> 98.6
Enter the temperature unit [K/C/F]> F
Enter the desired unit [K/C/F]> C

Current Temperature: 37.0 Degrees Celsius
```

Problem 3: Wind Chill

Background

The National Weather Service uses the concept of *wind chill temperature* to measure how cold people will feel when standing out in the wind. After many experiments, which involved putting Canadians into refrigerated wind tunnels, it's been determined that the best way to approximate the wind chill temperature is with this formula:

$$T_{wc} = 35.74 + 0.6215 \cdot T_F - 35.75 \cdot V^{0.16} + 0.4275 \cdot T_F \cdot V^{0.16} \tag{1}$$

Where T_F is the current temperature, in degrees Fahrenheit, and V is the wind speed in miles per hour.

When standing outside in extremely cold weather, you may run the risk of catching frostbite. After a certain amount of time outside, your skin eventually reaches an equilibrium with that of the cold air. If your final skin temperature drops below $-4.8°C$, the tissue will start to freeze!

Canada's Defense Research and Development team takes this matter very seriously. They suggest using the following equation to determine the equilibrium temperature that your exposed skin will drop to:

$$T_{final} = (0.1 \cdot T_C - 2.7883) \cdot \ln(V) + 0.2977 \cdot T_C + 19.874 \tag{2}$$

Temperature in this equation is given in degrees Celsius, and V is in units of kilometers per hour (km/h $= 0.621\times$ mph). This is considered accurate for outdoor wind speeds between 15 and 150 km/h, and temperatures between $-10°$ and $-45°$ Celsius.

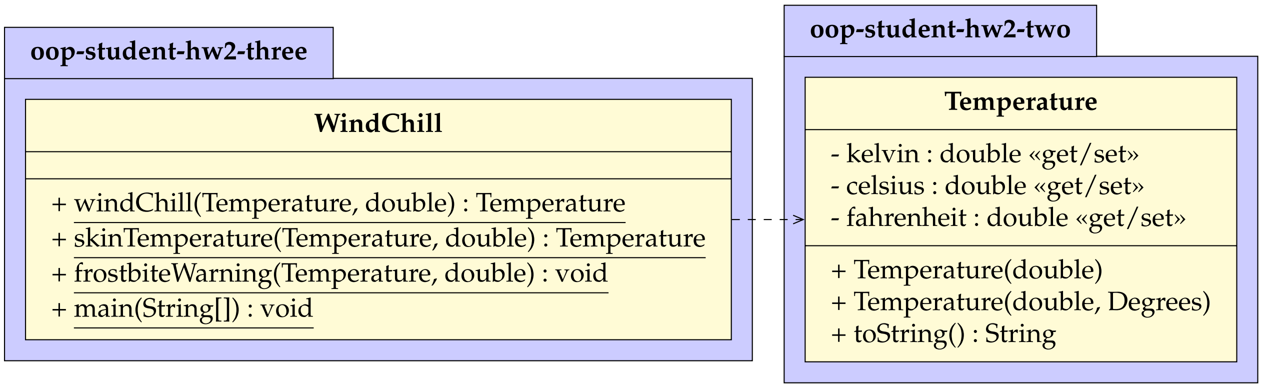
If T_{final} is less than $-4.8°C$, then staying outside in the cold will eventually give you frostbite. The danger is even more extreme if the temperature is below a certain level with respect to the wind speed:

$$T_C \leq 7.5 \cdot \ln(V) - 51.4 \tag{3}$$

In this case, the time it will take for frostbite to set in is less than 10 minutes!

Wind Chill Calculator

Using the equations detailed above, write one more simple program called `WindChill.java`. To get started, you will have to `import` the `Temperature` class from Problem 2:


oop-student-hw2-three
WindChill
+ windChill(Temperature, double) : Temperature
+ skinTemperature(Temperature, double) : Temperature
+ frostbiteWarning(Temperature, double) : void
+ main(String[]) : void
oop-student-hw2-two
Temperature
- kelvin : double «get/set»
- celsius : double «get/set»
- fahrenheit : double «get/set»
+ Temperature(double)
+ Temperature(double, Degrees)
+ toString() : String

The class will simply have three `static` methods:

1. `windChill()` takes the current temperature and the wind speed (mph), and returns the wind chill temperature (Equation (1))
2. `skinTemperature()` takes the temperature and wind speed (km/h), and returns an estimate of the final temperature one's skin will approach when exposed for the cold for too long, (Equation (2))
3. `frostbiteWarning()` takes the same arguments as `skinTemperature()` uses the information to print one of three warnings:
   - `"No Danger"`, if `skinTemperature()` returns a value greater than $-4.8°C$,
   - `"Warning"`, if the temperature is above the threshold given in Equation (3),
   - And `"Extreme Danger!"` otherwise

   If the temperature is above $-10°C$ or the wind is calmer than 15 km/h, there is definitely no danger; but if the temperature is less than $-45°C$ or the wind is faster than 150 km/h, anyone exposed to the cold is in extreme danger, (if not already frozen)!

Finally, write a `main()` program that prompts the user for the current temperature, a temperature unit (`"K"`, `"C"`, or `"F"`), and the wind speed in miles per hour, (km/h $= 0.621\times$ mph). Use the second constructor from `Temperature.java` to create a `Temperature` object with appropriate unit. Print the wind chill temperature, the temperature a person's skin will cool to, and an appropriate frostbite warning.

Example output:

```
Wind Chill/Frostbite Calculator
===============================

Enter the temperature> -11.5
Enter the temperature unit [K/C/F]> F
Enter the wind speed (mph)> 25

Wind Chill Temperature: -39.469 Degrees Fahrenheit
Final Skin Temperature: -10.482 Degrees Celsius
Extreme Danger! Get inside within 10 minutes to avoid freezing!
```

Submission

Make sure that you have Javadoc style comments for every class and method in your source code, as described in Homework 0. Document any unresolved bugs in the Javadoc comments for each of your classes. When you are satisfied that your code is complete, create a TAR file containing all of the source code for this assignment called:

```
cse213_<firstname>_<lastname>_hw2.tar.gz
```

Upload your submission to Canvas before the due date.