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

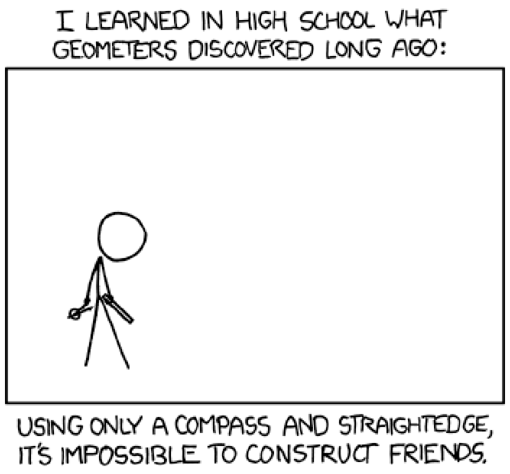

I LEARNED IN HIGH SCHOOL WHAT GEOMETERS DISCOVERED LONG AGO:
USING ONLY A COMPASS AND STRAIGHTEDGE, IT'S IMPOSSIBLE TO CONSTRUCT FRIENDS.

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 you write as a module meant to be imported and 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 in Object Oriented Programming.

Delegating Constructors

Another important concept 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 other constructors need to repeat some of that work. For cases like this, it is possible for one constructor to call another using the `this` keyword.

Say we want to make a class for user accounts, and do the bare minimum to make sure that all passwords contain numbers and capital letters. We'd have to write a bit of code in the constructor to make that happen:

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

    public User(String username, String password) {
        if (!password.matches(".*[A-Z].*")) {
            password = "ASDF" + password + "XYZ";
        }

        if (!password.matches(".*\\d.*")) {
            password = password + "!1234";
        }

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

Figure 2: A terrible password policy

Now say we wanted to add another constructor to create users from just a username. We can set a default password that will *just* the username with some characters added to the end:

```java
public User(String username) {
    this(username, username.trim());
}
```

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

Delegating constructors are often useful when one constructor has to do more than just a few assignments. It's also a good way to construct objects with default settings. If you already have a constructor that creates a circle, then just calling `this(0, 0, 1)` is a good way to create a unit circle centered at $(0, 0)$.

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, every member has been marked `private`, but has been made fully accessible through *get* and *set* methods. For all intents and purposes, that's the same as just making the attribute `public` to begin with [1]!

The real usefulness of getters and setters is that they let you control *how* attributes are accessed, and what happens when they are updated. Take the following program for example:

[1] 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.

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

    public int getIdNumber() {
        return idNumber;
    }

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

    public void resetPassword(String currentPassword, String password) {
        if (currentPassword.equals(this.password)) {
            this.password = password;
        }
    }
}
```

Figure 4: More selective uses of getters and setters

In this class `idNumber` is a *read-only* member – you can get that attribute but there's no way to set it once that object has been created. `getUserName` provides another read-only interface, but this time it's for an attribute that doesn't actually exist in the class. Instead of declaring a variable, `String username`, it computes the student's username on the fly every time it is requested.

Finally we have the `password` field, which doesn't have a getter or a setter. The only way to assign it is using the `resetPassword` method. We're using Java to enforce the policy that a student's password cannot be changed unless the user can also provide their current password [2].

The point of information hiding isn't to keep the implementation a secret from other programmers. Other programmers using your code are often perfectly capable of reading the source themselves! The point of setting up access controls is to make it easier to reason about your objects at run-time. This ultimately makes your code easier to read and debug. If you see that a student's `idNumber` is set incorrectly then you can immediately trace the problem to the constructor was call, because no other part of the program has the ability to set that value!

Getters and setters can also be used to prevent errors before they happen by short-circuiting bad inputs. One classic example would be array indexing:

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

```java
public class SafeArray {
    private int[] array;

    // Create an array of a given size, containing all 0's
    public SafeArray(int size) {
        this.array = new int[size];
    }

    // Create a copy of the given array
    public SafeArray(int[] array) {
        this.array = new int[array.length];

        for (int i = 0; i < array.length; i++) {
            this.array[i] = array[i];
        }
    }

    // Return -1 if the index is out of bounds
    public int get(int idx) {
        if (idx < 0 || idx >= array.length) {
            return -1;
        }

        return array[idx];
    }

    // Set the value only if the index is in bounds
    public void set(int idx, int value) {
        if (idx >= 0 && idx < array.length) {
            array[idx] = value;
        }
    }
}
```

Figure 5: Safe array indexing

Indexing an array out of bounds will cause a runtime error in Java. The `SafeArray` class acts just like an array of integers, but you can be sure reading or writing to a value in the array will never throw an `ArrayIndexOutOfBoundsException`. Instead the program will fail silently, which may or may not be a reasonable trade-off to make.

Another major use case for controlling access to class variables is keeping attributes in sync with each other. When different components of a data structure interact with each other, it is often useful to be able to assume that certain relationships will always hold between those components. In some cases, a data structure can end up in a broken state if two members can be set independently so that they *disagree* with each other about the meaning of the object!

Say you wanted to represent a duration of time, simultaneously using units of hours, minutes, and seconds. You could implement that using the following code:

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

    public void setSeconds(double seconds) {
        this.seconds = seconds;
        minutes = seconds / 60;
        hours = minutes / 60;
    }

    public void setMinutes(double minutes) {
        seconds = minutes * 60;
        this.minutes = minutes;
        hours = minutes / 60;
    }

    public void setHours(double hours) {
        seconds = hours * 3600;
        minutes = hours * 60;
        this.hours = hours;
    }
}
```

Figure 6: Timer attributes are kept in sync

The way the setters are written, you can know that all three variables in the class will always refer to the exact same duration. If you set the time in units of minutes, it will automatically correct the values for hours and seconds. If the user could accidentally set these independently, then it could lead to a bug later on if the program was written *assuming* they would be the same. Java lets you control what will happen when you set a variable, which makes it possible to avoid that entire category of software bug.

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 old code, and improve upon the overall program.

Point

First, rewrite `Point.java` to support polar coordinates as well as Euclidean coordinates. Add the attributes `radius` – to represent the distance from the origin, and `angle` – to represent the angle in *radians* (from $-\pi$ to π) going clockwise starting in the positive x direction. For example, the point $(1, 0)$ would 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) + distanceFromOrigin() : double + distance(Point) : double + compareTo(Point) : int + toString() : String

Write getters and setters for all four attributes, but write the setters such that changing the Euclidean coordinate variables updates the polar variables to match, and vice versa. For reference, these formulas can be used to convert between polar and Euclidean coordinates:

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

There are edge cases to consider for the second equation. If either x or y is negative, then `Math.atan` may misinterpret the value of the fraction $\frac{y}{x}$, and if $x = 0$ then you would get an `ArithmeticException` for dividing by zero! Luckily, Java has a builtin method that will handle all these cases for you. You can update the polar coordinates using:

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

For a more concrete example:

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

Should leave the existing angle ($\pi/4$) the same, but change the point's distance from the origin. The resulting coordinate in this case would be $(\sqrt{2}/2, \sqrt{2}/2)$, or:

```
(0.7071067811865476, 0.7071067811865476)
```

The constructors will be the same as in homework 1, but will also need to set the new attributes:

1. The default constructor creates the point $(0, 0)$
2. The second constructor takes two arguments, `x` and `y`, and creates the point (x, y)
3. The third constructor is a copy constructor, that creates a duplicate

`distance` and `distanceFromOrigin` also carry over from the previous homework. The first takes calculates the distance from the `Point` to another `Point` passed in as an argument. The second calculates the Point's distance from the origin, $(0, 0)$.

Implement the method `compareTo`, which will take a second Point and compare it to `this` Point. If the points have the exact same x and y coordinates, `return 0`. Otherwise, if the second point is above **and** to the right of the current point, `return -1`; and if the other point is below *or* to the left of it, `return 1`.

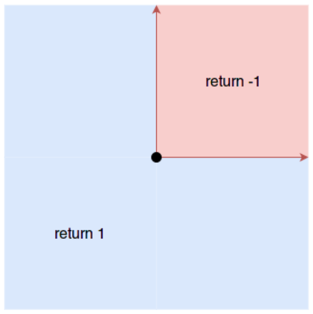


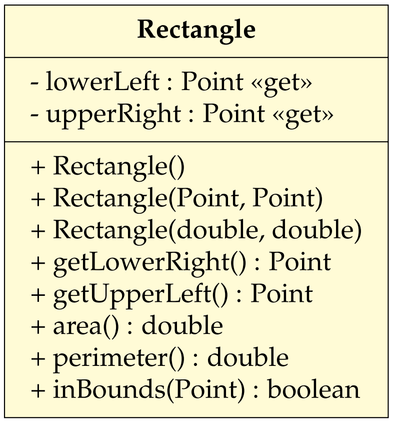
Figure 7: Visualizing the behavior of `compareTo`

Also add the method `toString`, which will convert the point to a string displaying the Euclidean coordinates in the form `"(x, y)"`, `toString` is a special method which will allow you to print instances of Point using `System.out.println`.

Rectangle

Rewrite `Rectangle.java` so that the boundary corners are defined by two `Point`s instead of four `double`s:

Write getters for the lower left and upper right corners, but *not* setters. Instead of returning the attributes directly, the getters should use `Point`'s copy constructor to return a `new` *copy* of each point. In other words, this version of `Rectangle` should be a totally *read only* data structure.


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

In addition, add two more getters to the class: `getLowerRight` and `getUpperLeft`, both of which return a new `Point`. You do not need to add new attributes for these points, you can compute them on the fly using the existing corner points. Like the other getters, these return a new copy every time they are called.

For the constructors, the boundary checking requirements are similar to the requirements in homework 1:

1. The default constructor still creates a 1×1 square whose lower left corner is on the origin.
2. The main constructor takes two Points, `lowerLeft` and `upperRight`, for the lower left and upper right corners of the rectangle, (x_1, y_1) and (x_2, y_2).

   Error check the input to ensure that (x_1, y_1) is in fact to the lower left of (x_2, y_2), rather than the other way around. If the coordinates are wrong, switch the points.

   It is also possible that neither coordinate is the lower left – you could be given a pair of *upper left* and *lower right* coordinates. In that case you can construct new points for the opposite corners, then set the points normally.

   Note: Use the `Point` copy constructor again to set the corner points within the rectangle. The two actual instances of `Point` should not be accessible outside of `Rectangle.java`!
3. The last 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`). You can delegate work to the main constructor to make sure that negative inputs are handled correctly.

The methods `area`, `perimeter`, and `inBounds`; behave exactly as in homeowrk 1. You only need to update the logic to use Points instead of individual coordinates. You do not need to include `distanceFromOrigin` in this version of the class.

Hint: The corner cases in the second constructor will be easier to detect using `Point`'s `compareTo` method!

Circle

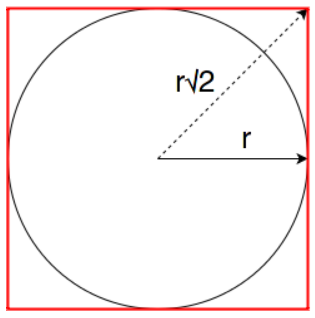
Also update `Circle.java` to use `Points` instead of individual coordinates. Write getters and setters for both attributes; just as in `Rectangle.java`, `getCenter` should return a new *copy* of the center point, instead of the point itself.

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

Make the default constructor create a circle with radius 1 centered at the origin. The main constructor will set the center to a *copy* of the given point, and the radius to the *absolute value* of the given number. The setters for both attributes should enforce the same policies – always create a copy when setting the center point, and don't set a negative radius.

The methods `area`, `perimeter`, and `inBounds`; behave exactly as in homeowrk 1. You only need to update the logic to use Points instead of individual coordinates. You do not need to include `distanceFromOrigin` in this version of the class.

Add the method `getBoundingBox` to generate `Rectangle` that surrounds 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:



for any given circle.

Problem 2: Temperature

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

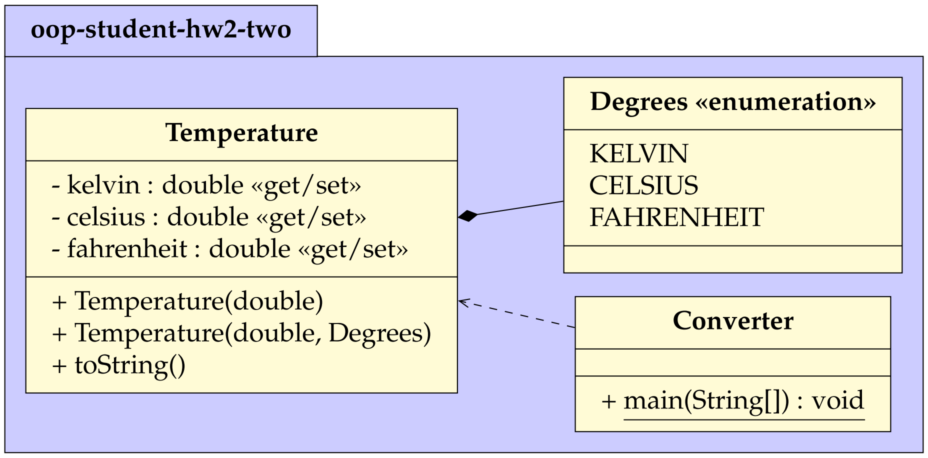


Figure 8: Package diagram – the underlining on main means it is declared `static`

`Temperature.java` has three private attributes, a `double` for each of the units of temperature. Write getters and setters for each; each setter should automatically set all three attributes to the appropriate values. For reference, these equations show you how to convert between Fahrenheit, Celsius, and Kelvin:

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

The first constructor assumes that the given temperature is in degrees Celsius, and sets all three values accordingly. The second constructor takes an additional argument for the unit, which is one of the three instances of this enumeration:

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

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

Finally, write the `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

The National Weather Service uses the concept of *wind chill* to report how cold the weather feels to you when you're outside. It can be approximated using this formula:

$$T_{wc} = 35.74 + 0.6215 \cdot T - 35.3225 \cdot v^{0.16}$$

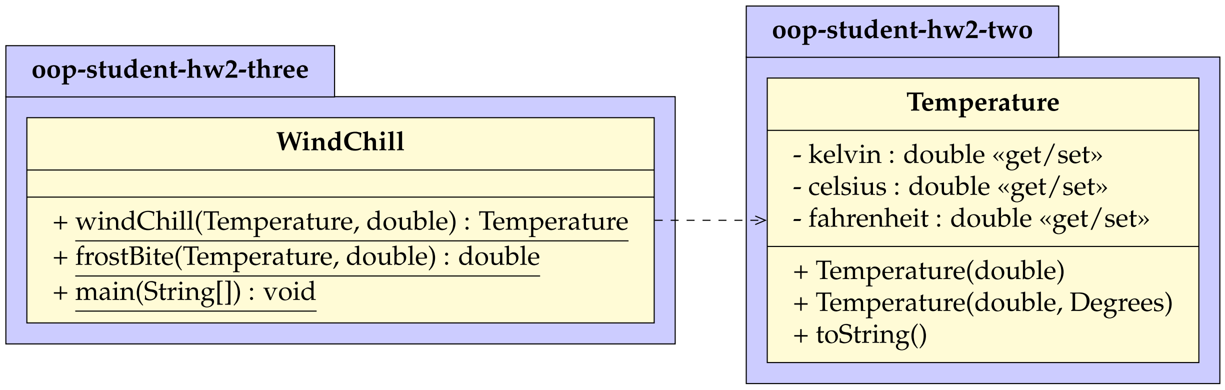
Where T is the current temperature, in degrees Fahrenheit, and v is the wind speed in miles per hour.

Wind speed effects how quickly your body loses heat when exposed to the cold. In extreme cases, high wind chill can make the difference in whether or not frostbite will develop! In case you find yourself out in a snow storm, the OFCM has developed this handy equation to estimate how long it will take before you're at risk of serious tissue damage:

$$t = (-26.133 \cdot v + 1994.6) \times (-4.8 - T)^{-1.668}$$

Here, v is the wind speed in miles per hour, and T is the temperature in degrees Celsius. t is the time in minutes it will take for frostbite to set in. This equation is more or less accurate for temperatures between $-15°C$ and $-50°C$, and with wind speeds up to 50 mph.

Use these equations to write another simple program, called `WindChill.java`, illustrated below. To get started, you will have to `import` the `Temperature` class from Problem 2.


oop-student-hw2-three
WindChill
+ windChill(Temperature, double) : Temperature
+ frostBite(Temperature, double) : double
+ 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()

The class will simply have three `static` methods. `windChill` takes the current temperature and the wind chill, and returns a new temperature taking wind chill into account.

`frostBite` takes the same arguments, and returns the time in minutes it will take for frostbite to set in. If the temperature is greater than $-15°C$ `return -1`, meaning there is no real danger of frostbite. If the temperature is below $-50°C$, or the wind speed is more than 50 mph, `return 0` to signify that the user would *already* be frostbitten.

Write a main program that prompts the user for the current temperature, unit, and wind speed. Print the wind chill temperature in degrees Fahrenheit, as well as the estimated time before frostbite sets in. In the cases when `frostBite` returns 0 or -1, print an informative message instead.

Example output:

```
Wind Chill Frostbite Clock
==========================

Enter the temperature> -11.2
Enter the temperature unit [K/C/F]> F
Enter the wind speed (mph)> 29

Wind Chill Temperature: -31.78 Degrees Fahrenheit
Time to Frostbite: 8.94 minutes
```

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.