Inheritance

CSE/IT 213

NMT Department of Computer Science and Engineering

"When there is no type hierarchy you don't have to manage the type hierarchy."

— Rob Pike

"Blessed are the young, for they shall inherit the national debt."

— Herbert Hoover

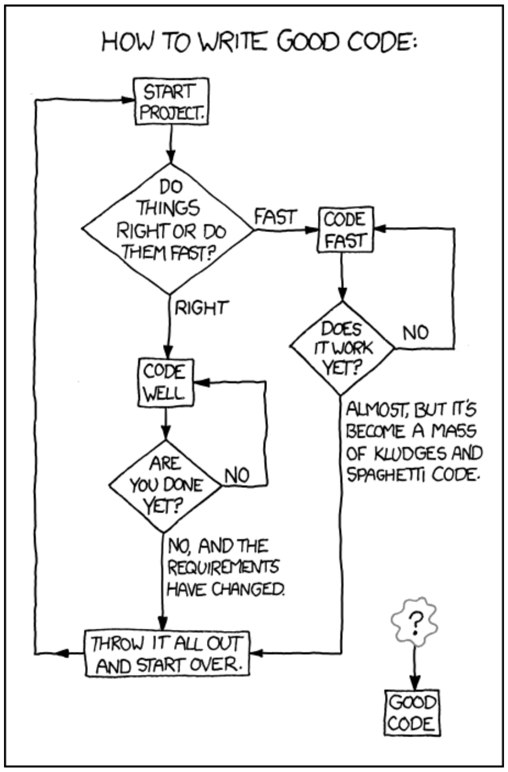

HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
RIGHT
CODE WELL.
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

Figure 1: `https://xkcd.com/844/`

Introduction

Inheritance

When used correctly, object oriented programming helps you organize your code in a way that makes it easier to write, manage, and maintain. The central concept is to add structure to the program to maximize your access to the code you've already written — *without* creating an unstructured mess of spaghetti code! Inheritance is a tool for achieving that goal, by letting you distribute your code across a tree structured collection of different classes.

When declaring a new class, you can specify a parent class for it to inherit code from. The syntax is simple:

```java
public class Specific extends General {
    ...
}
```

Figure 2: The `extends` keyword in Java

In this example, the class `General.java` has already been written. The result of extending it is that `Specific.java` will automatically have access to every non-`private` method and variable from `General.java`. For a more concrete example, say we have the class `Animal.java`:

```java
public class Animal {
    protected String species;
    private String foodSource;

    public Animal(String species, String foodSource) {
        this.species = species;
        this.foodSource = foodSource;
    }

    public String getSpecies() {
        return species;
    }

    public String getFoodSource() {
        return foodSource;
    }

    public void eat() {
        System.out.printf("The %s eats some %s\n", species, foodSource);

    }
}
```

This class represents animals using a fairly simple interface. Notice the use of the `protected` modifier, rather than `private`, for the `species` attribute. This tells Java to treat the variable as private, but that it should still be accessible within any *subclass* of `Animal`. A reasonable subclass might look like this:

```
 1  public class Bird extends Animal {
 2      private double flyingSpeed;
 3
 4      protected Bird(String name, String foodSource, double flyingSpeed) {
 5          super(name, foodSource);
 6          this.flyingSpeed = flyingSpeed;
 7      }
 8
 9      public Bird(double flyingSpeed) {
10          this("Bird", "seed", flyingSpeed);
11      }
12
13      public void fly(double distance) {
14          double time = distance / flyingSpeed;
15          System.out.printf("%s flies %.2f meters in %.2f seconds\n", species, distance,
              ↪  time);
16      }
17  }
```

You can think of `Bird` as a more specific instance of an `Animal`. All of the `public` and `protected` members of `Animal` also belong to `Bird`; so it can still call `eat()` and access the `species` attribute. However, since `foodSource` is marked `private`, you would have to use `getFoodSource()` to access that attribute within this class.

Note the use of the `super` keyword on line 5 — just like you can use `this()` to delegate work to another constructor in the same class, you can use `super()` to delegate work to a constructor of your parent class. This uses the `Animal` constructor to set the attributes `species` and `foodSource`, then sets the flying speed in the usual way.

We denote the inheritance relationship in UML using an unfilled arrow. This diagram shows how this system could be extended to include more `Animal` objects:

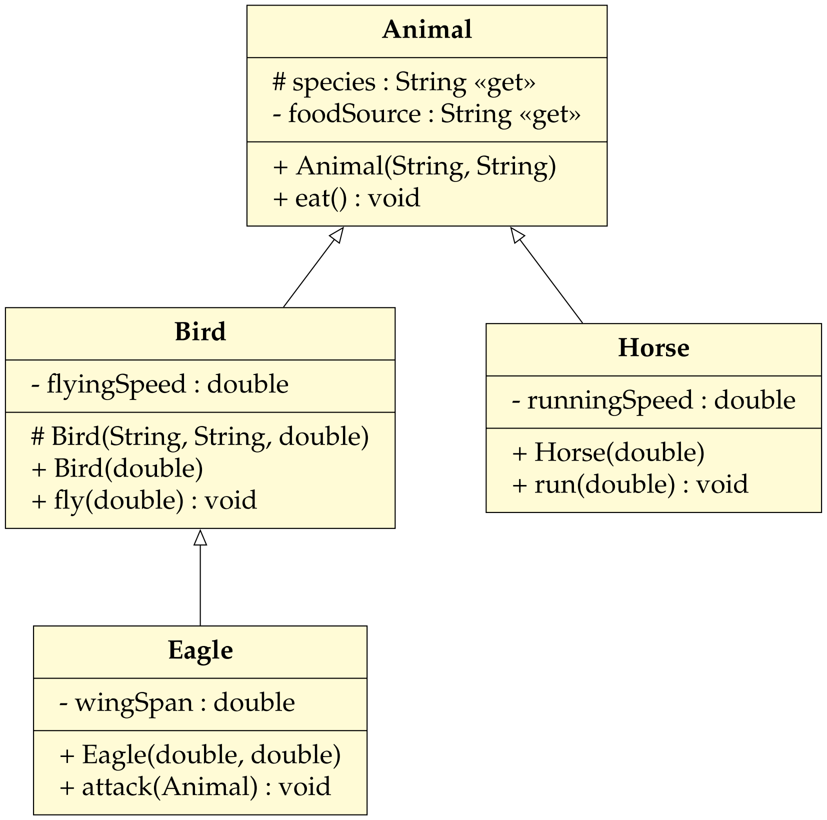


Figure 3: UML notation for class inheritance

We use the '#' to mark a member as `protected`, (along with '-' and '+' for `private` and `public` members). Any class can be inherited by another subclass, but no class can extend two different parent classes. Hence the tree structure.

Polymorphism

Inheritance doesn't just allow for a parent class to share methods with its children. It also allows you to instantiate different classes using a common type. This is called *polymorphism* — the ability of a single class to have many different implementations. A variable of type `Animal` can be instantiated with any of its subclasses, or even changed to a different `Animal`.

```java
Animal animal = new Animal("Giraffe", "high-up leaves");
animal.eat();

if (predator <= prey) {
    animal = new Bird(13.5);
} else {
    animal = new Eagle(42, 1.9);
}

System.out.println("New species: " + animal.getSpecies());
```

```
The Giraffe eats some high-up leaves
New species: Eagle
```

Figure 4: Polymorphism — an `Animal` can be instantiated by its subclasses

When you assign an `Animal` to a value of type `Bird`, you can still use all of the public methods in `Animal.java` because a bird *is an* animal. It inherited all of those methods from its parent class, so it can freely be used as an instance of that parent class. However, the compiler would not allow you to call the method `animal.fly()` in the above code, because that method is not supported by the `Animal` type. You also would **not** be able to perform the initialization backwards:

```java
Bird bird = new Animal("Shark", "humans");
```

This is because, (as you can see in the above example), not every animal *is a* bird. The *is-a* relationship only points in the direction of the parent class. Another interesting application of polymorphism is the ability to create heterogenous arrays, containing many different types of object:

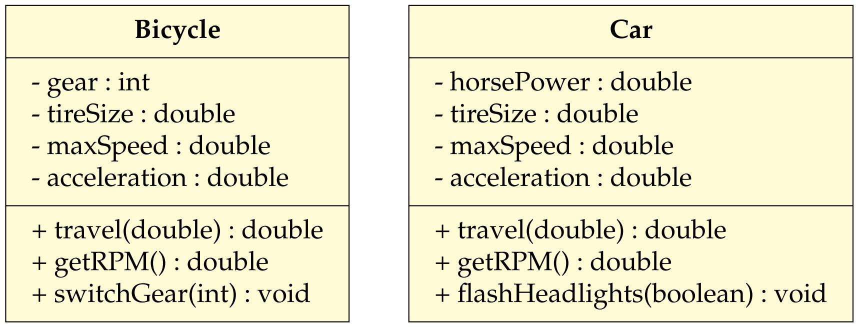
```java
Animal[] animals = {
    new Animal("Snake", "bugs"),
    new Horse(12),
    new Bird(20),
    new Eagle(38.5, 2.2)
}

for (Animal a : animals) {
    a.eat();
}
```

```
The Snake eats some bugs
The Horse eats some hay
The Bird eats some seed
The Eagle eats some helpless animals
```

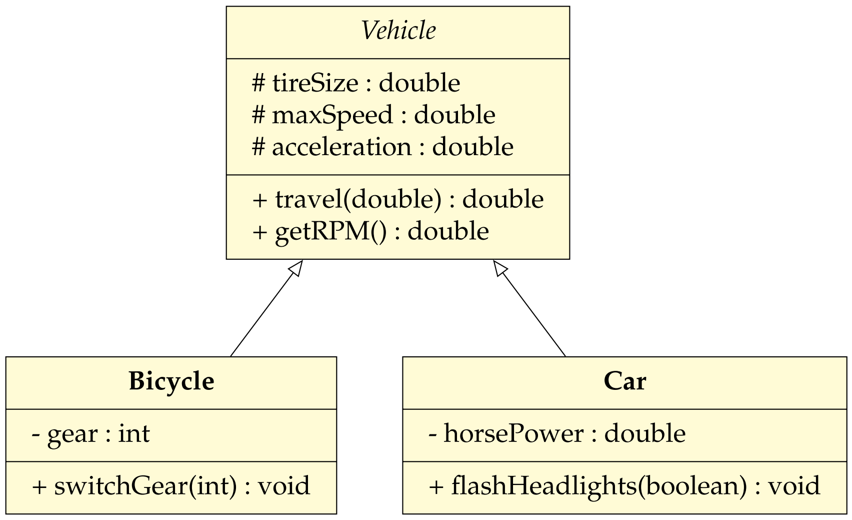
Figure 5: A heterogenous array — every element is an `Animal`

Abstract Classes

It is also possible to create an `abstract` class, meaning that the class is not meant to be instantiated. You never use an abstract class directly within your program, but they are still a useful way of creating unifying types within the class hierarchy of your program. For example, if you noticed that there was a high degree of overlap between two objects in your program:



It may be best to generalize that structure, and move some of the work out into an abstract class:



The italicized class name denotes that `Vehicle` is declared `abstract`, which has the following syntax:

```
1  public abstract class Vehicle {
2      protected double tireSize;
3      protected double maxSpeed;
4      protected double acceleration;
5
6      public double travel(double distance) {
7          ...
8      }
9
10     public abstract double getRPM();
11 }
```

The only difference between `Vehicle` and any other class is that it cannot be instantiated. That is, Java will prevent you creating a `new Vehicle(...)`. However, you can still declare variables of type `Vehicle` by instantiating them with one of its subclasses:

```
Vehicle bike = new Bicycle(...);
```

This still has all the benefits of being able to switch between `Car` and `Bicycle`, since they are compatible types. You can declare heterogenous arrays of cars and bicycles, and pass both types as arguments to a method that takes a `Vehicle`.

Also note the use of an *abstract method* on line 10. In this example, the method `getRPM()` is included in the class, even though it has no implementation. What you have in its place resembles a function prototype, like you might find in a C header file. An abstract method is like a contract between the abstract class and the subclass that extends it — it says that the subclass *must* always include code to implement the missing method! Abstract methods are useful for telling Java that it can *expect* a method to be there, even if you don't want to write it in the abstract class itself.

JUnit Testing

In the previous labs you provided unit tests using a class to ensure your implementation was correct. However, there is still some chance of error that can arise from hand rolling your own test class and it can become a hassle to write your unit test checks from scratch. To overcome this, a developer can utilize a unit testing framework such as JUnit. JUnit is a open source framework that allows developers to write and run repeatable tests. It provides features that comprise of assertions for testing expected results, test fixtures for sharing common test data and test runners for running tests.

To get started with JUnit through Intellij:

- First create a new package inside of the *src* directory and name it "tests". We need to mark our new test package as a test source root. Go to modules inside Project Structure (File -> Project Structure -> Modules). Click the *src* tab drop down arrow and highlight your tests package (it should be outside your oop package). Click on the "Mark as:" green "Tests" icon. Your package is now a test source folder

- It is recommended to add packages to your "tests" folder so you can distinguish between which tests belong to which problems you are working with. Right click on your "tests" folder and add a new package (New -> Package) and name it the same as your source code package with (tests.oop.name.hwN.exN.testing).
- Open a class you want to test such as `Eagle.java` and right click on the class name. Click "Generate..." and click "Test..." (Windows/Linux press Ctrl + Shift + T in the current class).
- A menu will prompt you if you have not created a test for the class yet. We will use JUnit4 since it is still more commonly used than the new release of JUnit5. Change the Testing library to JUnit4, you will see a warning that "JUnit4 library not not found in the module" warning. Click the "Fix" button. Once the "Download Library from Maven Repository" presents itself, enter junit:junit:4.9 and select the version of JUnit from the drop down. Make sure "Download to:" and "Transitive dependencies" is check marked to your *lib* folder and click "OK".
- Click on the "..." to the right of "Destination package:". Click a package drop down tab (that corresponds with your test class exercise i.e ex1) and click the nested testing package (ex1.testing). Leave the "Member" unchecked and click "Ok". A new testing class should be be generated inside the correct package you created earlier.
- If there are no errors, skip this step. Else, we are getting errors since we need to tell Intellij where the JUnit .jar file you downloaded is located. To fix this, go to (File -> Project Structure -> Libraries). From here, click the add icon in the inner top left region and click "Java" from the drop down. Navigate to where your project directory is located. Once there, click on the drop down to show the contents of *lib*. Click on junit-4.9.jar and click "OK". Click "OK" again to add the .jar to your module. Click "OK" to apply JUnit 4.9 to your libraries. You should now be able to write your tests in your new test class.

All JUnit tests methods are prefaced with an annotation and have a return type of void. To create a test we first preface a method with the `@Test` annotation. This tells the framework that the below method is a test method and to run this test when executed. It is good practice to name a test method with what it is testing for. This is helpful so you do not have to read the method's code to know what it is testing when you have many tests written.

JUnit supports static methods to test certain conditions using the `Assert` class. Utilizing these assertion methods, you can specify an error message, the return value you should expect and the actual value returned. An `AssertionException` is thrown if a comparison fails. For this lab, we will utilize the `assertEquals` assertion to verify your implementation is working as intended.

```java
@Test
public void testEagleWingSpanAssigned() {

    /* Eagle(double flyingSpeed, double wingSpan) */
    Eagle eagle = new Eagle(25.0, 20.0);

    double expected = 20.0;
    double actual = eagle.getWingSpan();

    /* Assert.assertEquals("Error message", expected , actual, delta) */
    Assert.assertEquals("Actual did not match expected", expected, actual, 0.1);
}
```

Figure 6: Ensuring that our constructor assigned the correct wingSpan value for our eagle object.

Note: When working doubles/floats types, we need add an additional parameter **delta** to the assertEquals method. This is due to a significant figures problem (i.e expected = 14.4 vs actual = 14.44 would throw an exception). Delta tells JUnit how much the actual return value can be off by and still be considered correct. When using integer types, the delta parameter is not needed.

To Run the test you created you can right click on the test class and click "Run". On Success the test will show a check mark on successful tests and a red "X" on failed tests. An error will show the expected value vs the actual value received and the stack trace of test:

```
java.lang.AssertionError: Actual did not match expected
Expected :20.0
Actual   :21.0
<Click to see difference>

at org.junit.Assert.failNotEquals(Assert.java:647)
at HW3.tests.EagleTest.testEagleWingSpanAssigned(EagleTest.java:20)
        ...
```

Figure 7: An example of a failed test that shows what was returned and expected.

It is good practice to make sure your expected value is correct. After, check your implementation details and debug what could be causing the error (Windows/Linux Ctrl + Shift + T will swap to the class that the test is associated with). In this case it is a small error in the constructor assigning. We check the actual parameter and debug the `eagle.getWingSpan()` method call due to the unexpected behavior:

```java
// The +1 is making the error in the code
public Eagle(double flyingSpeed, double wingSpan) {
    super(flyingSpeed);
    this.wingSpan = wingSpan + 1;
}

// Correction
public Eagle(double flyingSpeed, double wingSpan) {
    super(flyingSpeed);
    this.wingSpan = wingSpan;
}
```

Figure 8: An example of a small error that can create unexpected errors.

Rerunning the test displays a success after the modification. As you make modifications to your source code, having tests in place that have passed previous implementations can support a higher application integrity and isolate new bugs that come with new changes to your code base.

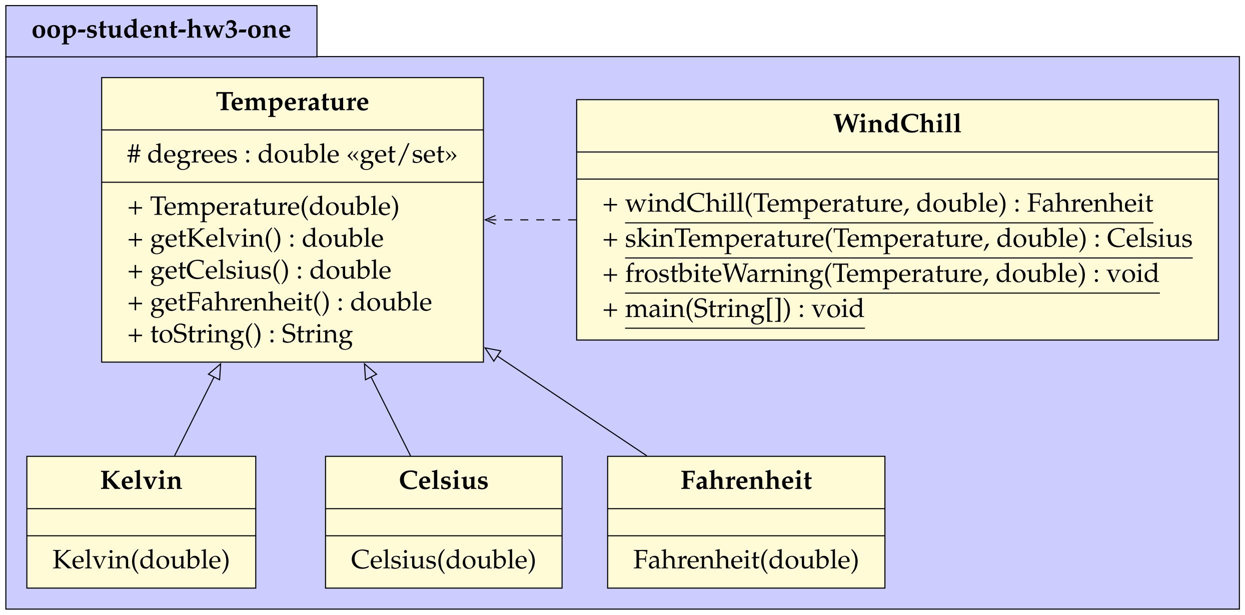
Note: It is highly recommended that all tests written in the test class do not depend on each other or other tests in order to isolate potential bugs and unexpected behaviors.

Problems

Problem 1: Wind Chill II

In the last assignment you wrote the class `Temperature.java` to represent a temperature in degrees Fahrenheit, Celsius, and Kelvin. In that assignment you used an `enum` to differentiate between the different types of temperature unit. There are many good uses of `enum`'s, but doing the work of the type system should not be considered one of them!

Rewrite the wind chill package so that it has the following layout:



Temperature

Instead of doing all the work, this version of `Temperature.java` should be a relatively bare base class. It stores the `protected` attribute `degrees`, but doesn't handle any units or conversions. Its four `get*()` methods can all return the same value, `degrees`; while `toString()` returns shows the value in the following format: `"13.37 Degrees"` — correct to two decimal places with no unit specified.

Now write three subclasses, whose `get*()` methods each perform the appropriate unit conversions separately:

- `Kelvin.java`:

$$C = K - 273.15$$
$$F = 1.8 \times K - 459.67$$

- `Celsius.java`:

$$K = C + 273.15$$
$$F = 1.8 \times C + 32$$

- `Fahrenheit.java`:

$$K = 5/9 \times (F + 459.67)$$
$$C = 5/9 \times (F - 32)$$

For each class's constructor, make sure that the specified temperature is greater than $0°K$. If it is below absolute zero, set it to the equivalent of $0°K$ in the relevant unit.

Also, each class should also override `Temperature.java`'s `toString()` method so that it appends a unit to the end of the returned text, (e.g., `"262.80 Degrees Kelvin"`)

Wind Chill

Write a modified version of `WindChill.java` to make use of the newly created subclasses of `Temperature`:

- Modify `windChill()` to return an instance of `Fahrenheit`, rather than `Temperature`. Note that the *argument* can still have the type `Temperature`, since every subclass will support the `getFahrenheit()` method.

$$F_{wc} = 35.74 + 0.6215 \cdot T_F - 35.75 \cdot (V_{mph})^{0.16} + 0.4275 \cdot T_F \cdot (V_{mph})^{0.16} \quad (1)$$

- Modify `skinTemperature()` to return an instance of `Celsius`. Again, its input argument still has the base type, Temperature.

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

- Make sure that `frostbiteWarning()` still works with the new class structure.

$$T_C \leq 7.5 \cdot \ln(V) - 51.4 \quad (3)$$

Your `main()` method should work with the exact same behavior as in Homework 2. You can instantiate a variable with the static type `Temperature` using any of the three subclass's constructors: `Kelvin()`, `Celsius()`, or `Fahrenheit()`. Decide which of these constructors to call based on the user's input, (either `"K"`, `"C"`, or `"F"`).

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!
```

Example JUnit test:

```java
public class WindChillTest {

    @Test
    public void checkWindChillTemperature() {

        Temperature temperature = new Fahrenheit(-11.5);

        double windSpeed = 25;
        double expected = -39.469;
        double actual = WindChill.windChill(temperature, windSpeed).getFahrenheit();
        double delta = 0.1;

        Assert.assertEquals("Wind Chill Temp incorrect.", expected, actual, delta);
    }
}
```

Figure 9: Test checks if your windChill() method is returnning the correct value

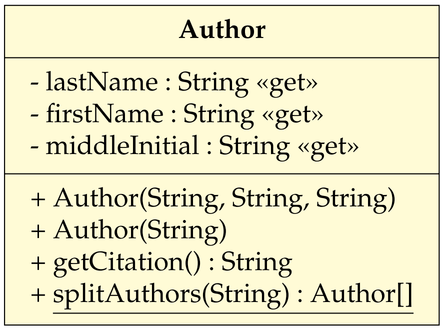
The above is an JUnit test is an example of another way of verifying your output.

Problem 2: Citations

For this problem you will write code to manage bibliographic citations.

Author

To start, implement the class `Author.java`:


Author
- lastName : String «get»
- firstName : String «get»
- middleInitial : String «get»
+ Author(String, String, String)
+ Author(String)
+ getCitation() : String
+ splitAuthors(String) : Author[]

The class has two constructors:

1. The first takes three `String`'s and uses them to set the
2. The second takes the author's full name, and needs to do a bit of string processing to separate the attributes. Examples of valid inputs to this constructor are:
   - `"Henry David Thoreau"`
   - `"Henry D Thoreau"`

- `"H. D. Thoreau"`
- `"Henry Thoreau"`

In every case, the three names are separated by only spaces; you can use `name.split(" ")` to separate the names into an array of substrings. The middle initial may not be present at all, in which case that attribute should be set to the empty string, (`""`); otherwise, the initial should be set in should come in the format: `"D."` — capitalized first letter followed by a period.

The method `getCitation()` should return a string in the format `"Thoreau, H. D."`, i.e.:

```
<lastName>, <firstInitial>. <middleInitial>.
```

The `static` method `splitAuthors()` takes a `String`, containing multiple authors' names, and returns an array of `Author` objects. The rules for processing the input are:

- If there are three or more authors then each new name will be separated by a comma, (`", "`), and the final name will be prefaced by the word `"and "`
- If there are only two authors then their names will be separated only by the string `" and "`
- If there is only one author, their name can be passed directly to the second `Author` constructor

For example:

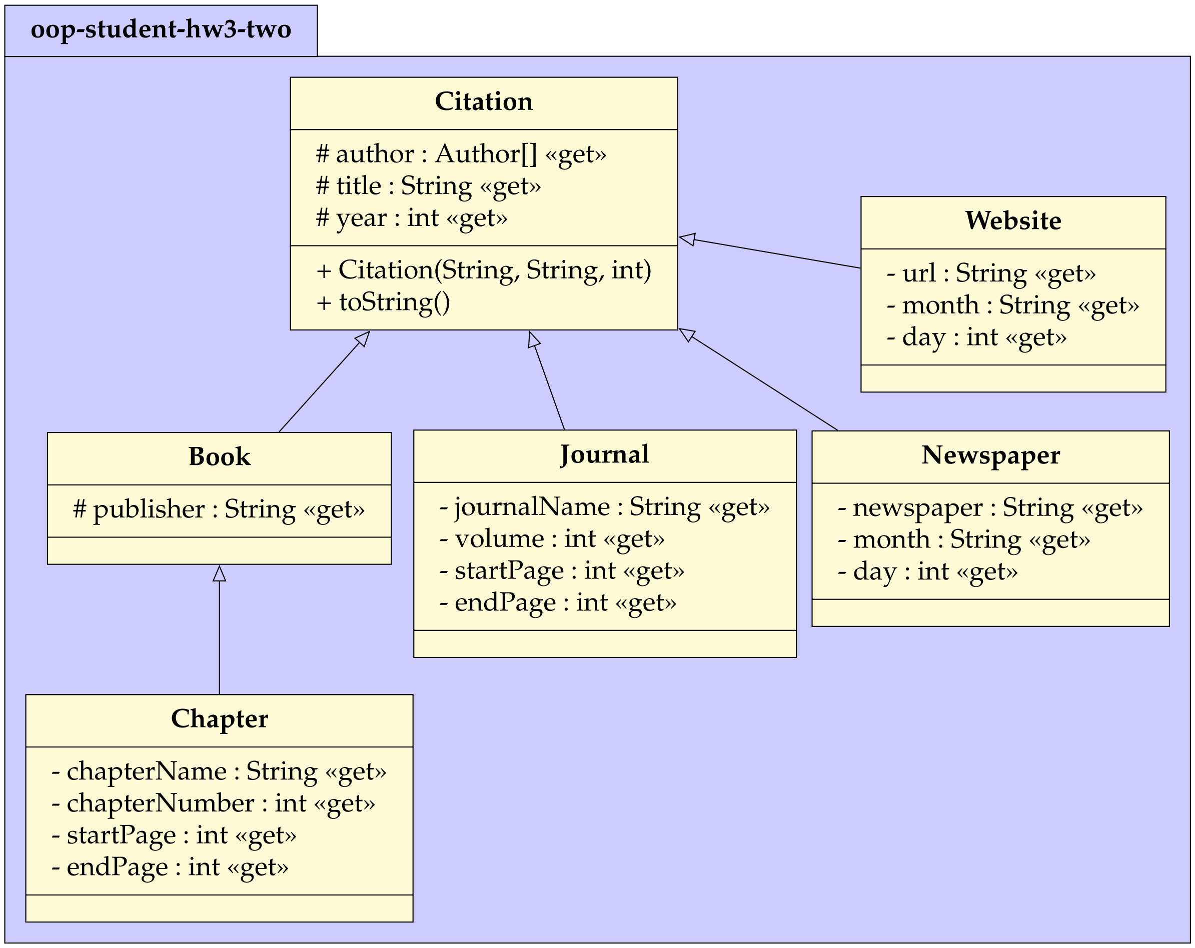
```
"Mark Twain, Henry D. Thoreau, R W Emerson, and N Hawthorne"
```

Write code to separate this string into four separate names. Use each name to construct a new `Author` object, and return an array containing each of the authors.

Hint: Check the Java documentation for information on the methods `String.contains()`, `String.split()`, `String.replaceFirst()`.

Citations

Write the class `Citation.java` to represent a reference to some work by one or more `Author`'s. Also write five additional small classes that extend `Citation` in the following way:



- The constructor for `Citation` takes a `String` containing the name or names of the author(s), a title for the work, and the year in which it was written. The first argument can be converted to an array of `Author`'s by passing it to `Author.splitAuthors()`.
- Write getters for the three private attributes. The getter method `getAuthors()` should use `array.clone()` to return a copy of the private array, rather than the array itself.
- The `toString()` method should display a citation in the format:

```
"<author 1>, ... <author N>, <title>., <year>"
```

- Write the additional subclasses of `Citation`:
  - `Book.java`,
  - `Journal.java`,
  - `Website.java`,
  - `Newspaper.java`,
  - and `Chapter.java` as a subclass of `Book`.

  Each class only needs a constructor which can set each of its attributes, and to override the `toString()` method so it shows this information in a reasonable way.

Bibliography

Finally, implement the class `Bibliography.java`, with a `main()` method that creates an array of at least five citations. Assign each element of the array to a different instance of `Citation`. Then loop over the array and print each citation to the console using `System.out.println()`.

Example output:

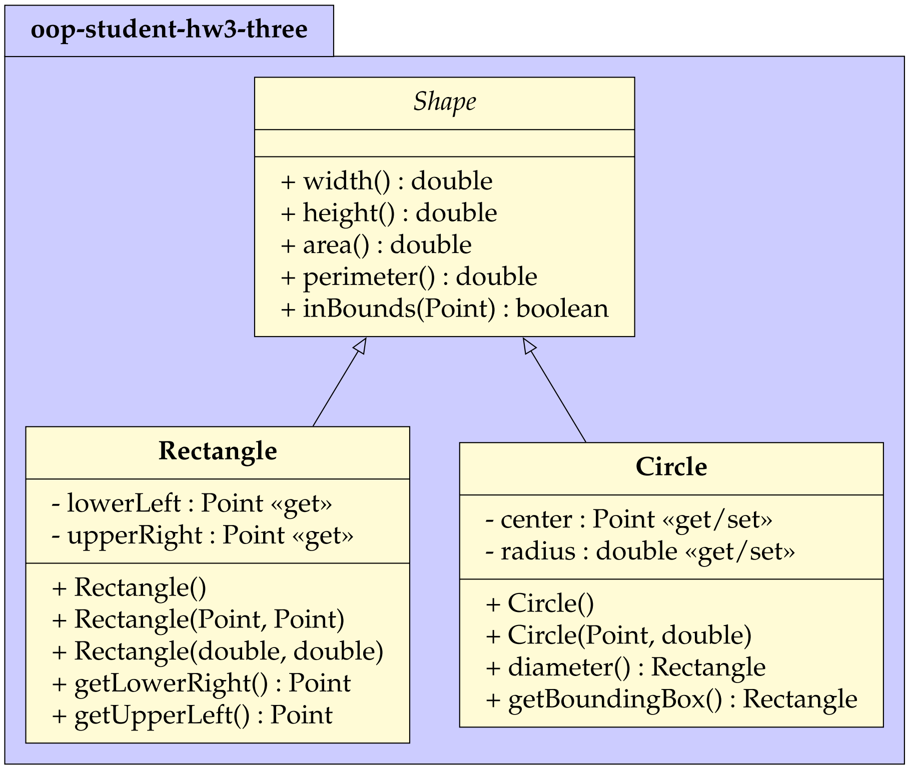
```
Thoreau, H. D., Walden., 1854
Ticknor and Fields Publishing Co.

Tolkein, J. R., The Fellowship of the Ring., 1954
Chapter 4, A Short Cut to Mushrooms (pp. 58-65)

Turing, A. M., Computing machinery and intelligence., 1950
Mind, vol. 59 (pp. 433-460)

Herbers, J., NIXON RESIGNS., 1974
New York Times, August 8

Diehl, S., Functional Programming, Abstraction, and Naming Things., 2016
http://www.stephendiehl.com/posts/abstraction.html, January 9
```

Problem 3: More Geometry

Shape

In the last two assignments you wrote and refined the classes `Point.java`, `Rectangle.java`, and `Circle.java`. For this problem, you will *organize* that code to create a class hierarchy, and extend those classes to create more complex shapes.

`Point.java` does not need to be modified, but it may be convenient for you to copy it into this new package. To start out, write an abstract class called `Shape.java`:


oop-student-hw3-three
Shape
+ width() : double
+ height() : double
+ area() : double
+ perimeter() : double
+ inBounds(Point) : boolean
Rectangle
- lowerLeft : Point «get»
- upperRight : Point «get»
+ Rectangle()
+ Rectangle(Point, Point)
+ Rectangle(double, double)
+ getLowerRight() : Point
+ getUpperLeft() : Point
Circle
- center : Point «get/set»
- radius : double «get/set»
+ Circle()
+ Circle(Point, double)
+ diameter() : Rectangle
+ getBoundingBox() : Rectangle

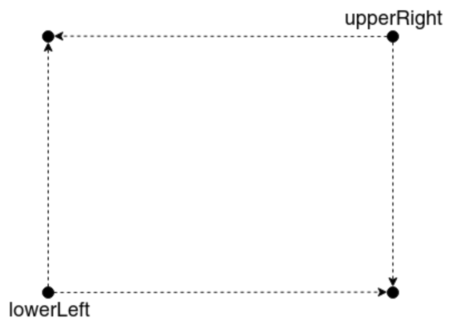
The `Shape` class's five methods can be declared `abstract`. This means you don't need to write any of the code in `Shape.java`, but you will be required to include those methods in its subclasses.

The only changes you need to make in `Rectangle.java` and `Circle.java` is to make them both `extend Shape`. `Circle.java` did not originally include a `width()` or `height()` method, but these can obviously return the same thing as `diameter()`.

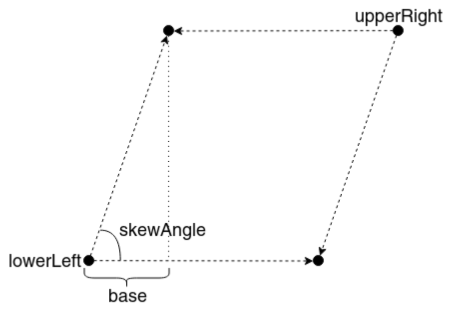
You do **not** need to change any `private` attributes to `protected` — since you can use their `get*()` methods to access those variables.

Parallelogram

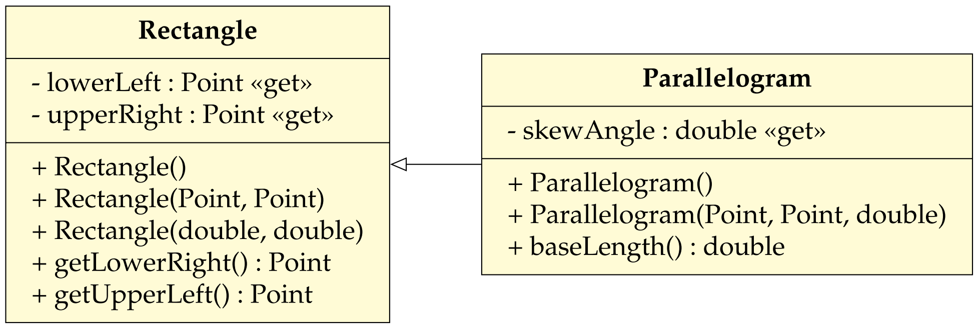
A parallelogram can be thought of as a rectangle that has been skewed sideways by a certain angle. In a rectangle, this angle is always $\pi/2$ (that is, 90°), so the upper left point always had the same x coordinate as the lower left point:



The parallelogram extends this concept by skewing the shape by some angle strictly less than π radians (180°):



With that in mind, create a subclass of `Rectangle` called `Parallelogram.java`:


Rectangle
- lowerLeft : Point «get»
- upperRight : Point «get»
+ Rectangle()
+ Rectangle(Point, Point)
+ Rectangle(double, double)
+ getLowerRight() : Point
+ getUpperLeft() : Point
Parallelogram
- skewAngle : double «get»
+ Parallelogram()
+ Parallelogram(Point, Point, double)
+ baseLength() : double

`Parallelogram` adds the private attribute `skewAngle`, which is the angle between the positive x axis and the left edge of the parallelogram.

- The default constructor should delegate to the default `Rectangle` constructor, and manually set `skewAngle` to $\pi/2$.
- The second constructor takes two points, the upper right and lower left corners, and a skew angle. Again, you can delegate to the parent constructor before setting the angle.

  The angle must be strictly less than π, and must also be greater than the angle of the diagonal of the surrounding rectangle, (i.e. `skewAngle > Math.atan2(y2 - y1, x2 - x1)`). For convenience, check that the angle is in bounds, and if not simply set it back to the default of $\pi/2$.
- Add the method `baseLength()` to return the length of the base of the parallelogram.

  The **base** — shown in the diagram above — refers to the base of the right triangle formed by cutting off one of its sides. You can calculate its length with this formula:

$$base = \frac{height}{\tan\theta}$$

  Note that when the skew angle is wider than $\pi/2$ radians, the orientation of this triangle is flipped, which will result in this value being *negative*.

- Override `getLowerRight()` and `getUpperLeft()` to return the coordinates of the skewed corner points. In general, the y coordinates of the opposing corners will be the same, but the x coordinates will be offset, either to the left or to the right, by the base length.
- Override `width()` to return the total width of the parallelogram. If $\theta \leq \pi/2$, then this is the same as the width of the surrounding rectangle; but if $\theta > \pi/2$, then the width is extended by $2\times$ the base length.
- Override both `area()` and `perimeter()`.

  The area of a parallelogram is still $width \times height$, but you can't calculate the width by subtracting the lower left x coordinate from the upper right. Instead, the length of top and bottom edges is $x_{right} - x_{left} - base$.

  Likewise, the length of the skewed vertical edges is calculated differently: $v = \frac{height}{\sin\theta}$. With this information in mind, it should be easy to calculate both the area and perimeter.
- Override `inBounds()` to check whether a given point is inside the parallelogram.

  You can easily check that the point's y coordinate is between the top and bottom edges of the parallelogram. You also need to check that the point is **below** the line of its **left** edge and **above** the line for its **right** edge, (or vice versa if $\theta > \pi/2$). A point (x, y) is within those two lines if its coordinates satisfy both of these inequalities:

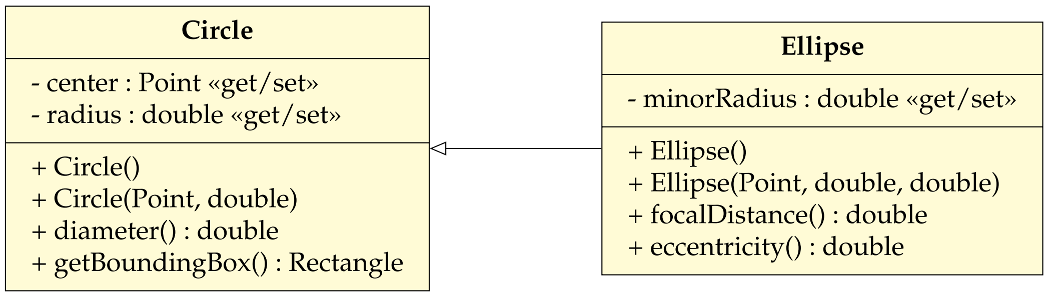
  $$y - y_{bottom} \leq \frac{height}{base}(x - x_{left}) \qquad\qquad y - y_{top} \geq \frac{height}{base}(x - x_{right})$$

  The math is almost the same in the case where $\theta > \pi/2$, but with the $\geq$ and $\leq$ symbols reversed.

Ellipse

An ellipse is a circle that has been stretched, either horizontally or vertically. As a result, an ellipse can be thought of as having two radii: the semi-major axis and the semi-minor axis, which measure half the ellipse's width and height, respectively.

Create a subclass of `Circle` called `Ellipse.java`:


Circle
- center : Point «get/set»
- radius : double «get/set»
+ Circle()
+ Circle(Point, double)
+ diameter() : double
+ getBoundingBox() : Rectangle
Ellipse
- minorRadius : double «get/set»
+ Ellipse()
+ Ellipse(Point, double, double)
+ focalDistance() : double
+ eccentricity() : double

The subclass adds a private attribute, `minorRadius`, to represent the semi-minor axis, which runs parallel to the y axis.

- The default constructor should delegate the default `Circle` constructor, and set the `minorRadius` equal to the `radius` — this creates the a unit circle.
- The second constructor takes the center `Point`, `radius`, and `minorRadius`; and uses them to set the parameters of the ellipse.
- Override `width()` and `height()` to the major and minor diameters of the ellipse, respectively.
- Override `diameter()` to return the maximum of the ellipse's width and height.
- Override `area()`, `perimeter()`, and `inBounds()` according to the following rules:
  - $A = \pi \cdot r_1 \cdot r_2$
  - $P \approx \sqrt{2}\pi\sqrt{r_1^2 + r_2^2}$
  - If the center point of the ellipse is (h, k), then the point (x, y) is inside the ellipse when:

$$\frac{(x-h)^2}{r_1^2} + \frac{(y-k)^2}{r_2^2} \leq 1$$

- Override `getBoundingBox()` to create a `Rectangle` surrounding the ellipse. The `radius` will tell you where the x coordinates belong with respect to the center, and the `minorRadius` will tell you where to put the y coordinates.
- The ellipse has two **focal points** along its longest axis. The distance between its center and each focal point is:

$$f = \sqrt{|r_1^2 - r_2^2|}$$

  Implement the method `focalDistance()` to return this value.
- The **eccentricity** of an ellipse is defined as:

$$e = \frac{f}{\max(r_1, r_2)}$$

  Implement the method `eccentricity()` to return this value.

Unit Testing

Write unit tests using JUnit for the new or overridden methods in `Parallelogram.java` and `Ellipse.java`.

You should write *at minimum* one thorough test for each new method and constructor in this package. That is:

- At least 9 tests for `Parallelogram.java`
- At least 11 tests for `Ellipse.java`

Your test cases should check that each method follows the specification given in the assignment. If a method returns a wrong value and throws and exception, check your implementation of the method/constructor being tested.

You should integrate JUnit into the previous assignment or keep the test class if you would like, so you can assure yourself that that `Point.java`, `Rectangle.java`, and `Circle.java` are still correct.

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.

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>_hw3.tar.gz
```

Upload your submission to Canvas before the due date.