Working With Objects

CSE/IT 213

NMT Department of Computer Science and Engineering

"Perhaps the greatest strength of an object-oriented approach to development is that it offers a mechanism that captures a model of the real world."

— Grady Booch (software engineer)

"After all, you cannot know the strength of your faith until it has been tested."

— David Magee (screenwriter, Life of Pi)

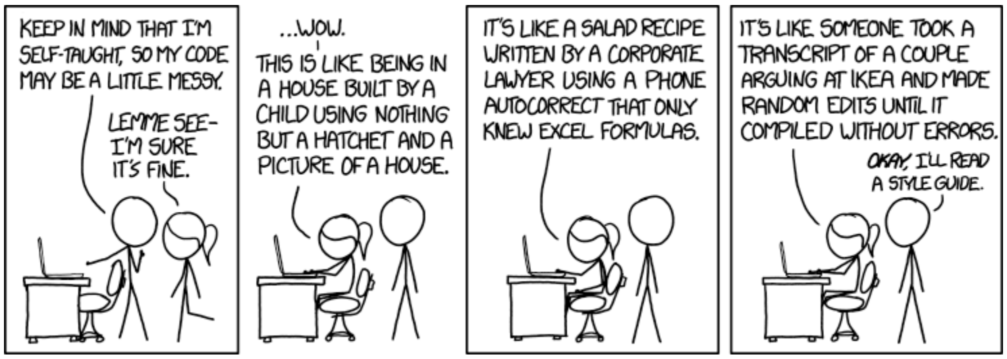
KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE– I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.

Figure 1: `https://xkcd.com/1513/`

Introduction

In this homework assignment you will use Java to define simple objects. We introduce UML as a notation for defining the *interface* of a class in Java. You will need to write code to match the interfaces shown in UML class diagrams. The assignment also touches on the topic of unit testing.

Constructors

In Homework 0 you wrote all of your code within the `main` method of each class. However, most Java classes don't usually have a `main` method; in most cases a class is only used to define a data structure whose operations will be used elsewhere in the program. A minimally simple Java program could define a structure using no code at all.

```java
public class Point {
    int x;
    int y;
}
```

Figure 2: `Point.java`

The `Point` class on its own doesn't do anything, but it can still be compiled and used by other programs. Another class within the same package could make use of `Point.java` with the following code snippet:

```java
1 Point pt = new Point();
2 pt.x = 4;
3 pt.y = 3;
4 System.out.println("Distance squared: " + (pt.x * pt.x + pt.y * pt.y));
```

Figure 3: How *not* to use objects in Java!

This program is simple to read and understand, but the style would be considered extremely ugly to a Java programmer. The problem is with lines 2 and 3, where the elements the `Point` class are modified using the dot (`.`) syntax. In object oriented programming, directly accessing the attributes of another object is generally considered a bad idea.

There is a much better way to initialize `x` and `y` *as* the `Point` is created. The solution is to use a *constructor*. Constructors are special methods that tell Java objects what to do when they are initialized using the `new` keyword. Let's add one to the `Point` class:

```java
1 public class Point {
2     int x;
3     int y;
4
5     public Point(int x, int y) {
6         this.x = x;
7         this.y = y;
8     }
9 }
```

Figure 4: `Point.java` with a constructor

The constructor is a method with the same name as the class itself, and doesn't have any return type. Notice the use of the keyword `this` – it provides you a reference to the current object. We're using it here to differentiate between the `x` passed as a parameter on line 5, and the class attribute `x` defined on line 2.

Once you've written a constructor for your class, you can initialize a variable of the type `Point` by using the `new` keyword, like so:

```java
Point pt = new Point(4, 3);
```

The arguments will be passed to the constructor, and it will assign the attributes of the new object. It is also possible to have multiple *overloaded* constructors within the same class:

```java
public class Point {
    private int x;
    private int y;

    // A default constructor, to create the point (0, 0)
    public Point() {
        this.x = 0;
        this.y = 0;
    }

    // A constructor to explicitly assign both attributes of the class
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Figure 5: `Point.java` with multiple constructors

When you use the `new` keyword to initialize an object, it will automatically figure out which constructor to call based on the types of its arguments.

Getters and Setters

As stated earlier, it is bad style to directly access the attributes of an object in Java. The private keyword, shown in the example above, allows Java to enforce that convention. When an attribute or method is marked `private`, it means that it is invisible to any class other than the one that defined it.

Unless it is explicitly stated otherwise, you should *always* assume that the attributes within a class are declared to be `private`!

This preference imposes a constraint on you as a programmer. How do you read or write to a data structure if all of its pieces are marked `private`? The idiomatic way to get around this in Java is to use *getters* and *setters*. These are short one line methods that return and assign to the private attributes of a class, respectively. We can see an example by augmenting `Point.java` one more time:

```java
public class Point {
    private int x;
    private int y;

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }
}
```

Figure 6: `Point.java` with getters and setters

The guiding principle is that all data should be immutable, unless you explicitly write an interface for accessing them. Combining the above example with the constructors in Figure 5, we see a revision of the excerpt from Figure 3:

```java
Point pt = new Point(4, 3);
int distanceSquared = pt.getX() * pt.getX() + pt.getY() * pt.getY();

System.out.println("Distance squared: " + distanceSquared);
```

Figure 7: How to use objects in Java!

Using getters and setters is a lot more verbose, but when for less trivial programs they can be a good way to maintain *encapsulation* for your class. You'll learn more about encapsulation later on in the course.

UML

The Unified Modeling Language (UML) is a graphical way to describe the layout of a program in object oriented programming. A UML class diagram shows you the *interface* provided by the class, without going into any of the implementation details. In other words, it shows what the class should look like to an outside observer, who wants to *use* your code without having to write it themselves.

UML will tell you a list of the members that a class should have, but not what the code should look like. That part is up to you! This example shows how we will specify classes using UML:

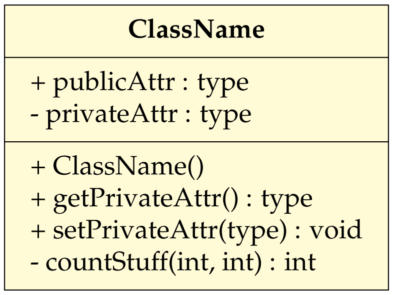
ClassName
+ publicAttr : type
- privateAttr : type
+ ClassName()
+ getPrivateAttr() : type
+ setPrivateAttr(type) : void
- countStuff(int, int) : int

Figure 8: UML example

A class diagram is just a box broken up into three sections: name, attributes, and methods. The left margin tells you the visibility of each member, a " + " means the member is `public`, and a " - " means it is `private`. The colon, : , is used to denote an element's type – the type of an attribute or the return type of a method. These are different from Java's syntax, but should be just as easy to read and understand. When you read the above class diagram, you should understand it as shorthand for this corresponding skeleton code:

```java
public class ClassName {
    public type publicAttr;
    private type privateAttr;

    public ClassName() {
        ...
    }

    public type getPrivateAttr() {
        return this.privateAttr;
    }

    public void setPrivateAttr(type pm) {
        this.privateAttr = pm;
    }

    private int countStuff(int x, int y) {
        ...
    }
}
```

Figure 9: Corresponding Java code

We will also occasionally use UML to describe the layout of a package, or the relationships between multiple classes in a package. For example, this diagram shows a package containing two classes, `Point.java` and `Rectangle.java`:

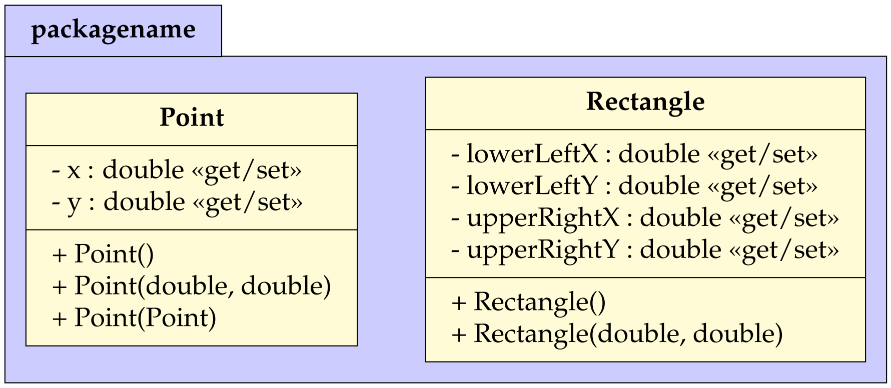
packagename
Point
- x : double «get/set»
- y : double «get/set»
+ Point()
+ Point(double, double)
+ Point(Point)
Rectangle
- lowerLeftX : double «get/set»
- lowerLeftY : double «get/set»
- upperRightX : double «get/set»
- upperRightY : double «get/set»
+ Rectangle()
+ Rectangle(double, double)

Figure 10: UML package example

We're also using "«get/set»" to indicate that those private attributes should have getters and setters. This saves us the space of adding a row for every getter and setter.

In this assignment you will have to write code to match the interface given by UML class diagrams. You should make sure you understand the meaning of the public/private modifiers (+/-) and the type annotations so you can translate the UML into Java.

Problems

Problem 1: Geometry

Point

First implement the class `Point.java`, using `double` instead of `int` as the type for the x and y coordinates. You may use the code in the Introduction for the first two constructors, and the getters and setters. In addition, implement a "copy constructor" that will initialize a new `Point` as a duplicate of another point.

Point
- x : double «get/set» - y : double «get/set»

+ Point()
+ Point(double, double)
+ Point(Point)
+ distance(Point) : double
+ distanceFromOrigin() : double

Also add two additional methods, `distance` and `distanceFromOrigin`. 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, $(0, 0)$. Both return the distance as a `double`.

Rectangle

Implement the class `Rectangle.java`, containing two pairs of coordinates – the lower left and upper right corners of the rectangle (assuming all the edges are parallel/perpendicular to the x axis). Write getters and setters for each coordinate.

Rectangle
- lowerLeftX : double «get/set» - lowerLeftY : double «get/set» - upperRightX : double «get/set» - upperRightY : double «get/set»
+ Rectangle() + Rectangle(double, double) + Rectangle(double, double, double, double) + area() : double + perimeter() : double + distanceFromOrigin() : double + inBounds(double, double) : boolean

The class should have three constructors:

1. The default constructor should create a 1×1 square whose lower left corner is on the origin.

2. The second constructor takes two doubles, width and height, and creates a rectangle with one corner touching the origin, and the other corner touching the point (width, height).
3. The last constructor takes four coordinates; x1, y1, x2, and y2; for the lower left and upper right coordinates, (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 them.

   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 correct the coordinates by swapping y_1 and y_2, then proceeding to set the corner points normally.

The area and perimeter methods should be self explanatory; they return doubles for the area and perimeter of the rectangle, respectively.

distanceFromOrigin returns the distance between the lower left corner and the origin. Both should return a double.

The final method, inBounds, takes x and y coordinates and determines whether that point is inside the rectangle, or on the perimeter. The method returns a boolean; true if the point is within the bounds of the rectangle, and false otherwise.

Circle

Implement the class Circle.java, composed of a pair of coordinates for the center point, and a radius. Write getters and setters for all three attributes. You can use this skeleton code to get started:

```java
public class Circle {
    private double centerX;
    private double centerY;
    private double radius;

    public Circle() { ... }
    public Circle(double x, double y, double radius) { ... }

    // getters and setters go here

    public double area() { ... }
    public double perimeter() { ... }
    public double distanceFromOrigin() { ... }
    public double inBounds() { ... }
}
```

Create a UML diagram describing the features of this class. You will turn in a PDF containing your class diagram in addition to your code.

The Circle class has two constructors:

1. The default constructor creates a circle centered at the origin, with a radius of `1.0`.
2. The second constructor takes three arguments; `x`, `y`, and `radius`; and uses them to set the attributes of the circle.

The `area` method returns a `double` equal to the area of the circle (πr^2), and `perimeter` returns the value of the circumference ($2\pi r$).

The method `distanceFromOrigin` calculates the distance between the center of the circle and the origin, and returns a `double`.

`inBounds` works the same as in `Rectangle.java` – it takes x and y coordinates, returns `true` if the point is inside the circle (including *on* the perimeter), and returns `false` otherwise.

Test

Finally, write the class `Test.java` to test the methods in `Point.java`, `Rectangle.java`, and `Circle.java`. Except the getters and setters, make sure you write *at least one* test for every constructor and method in the package!

Every test should run your code using some fixed input, and check to make sure the results you get back match an expected output. If a method returns the wrong value, print an error message with `System.out.println` stating which test failed and why. Here's an example you can use to unit test `Rectangle.java`:

```java
public static void testRectangleWrongPoints() {
    Rectangle rect = new Rectangle(3, 2, 1, 4);
    double x1 = rect.getLowerLeftX();
    double y1 = rect.getLowerLeftY();
    double x2 = rect.getUpperRightX();
    double y2 = rect.getUpperRightY();

    if (x1 != 1 || y1 != 2) {
        System.out.println("testRectangleWrongPoints: Wrong lower left point!");
    }

    if (x2 != 3 || y2 != 4) {
        System.out.println("testRectangleWrongPoints: Wrong upper right point!");
    }
}
```

Figure 11: Don't let this be your *only* unit test for the Rectangle constructor!

The `Test` class should include a `main` method that runs each of the tests you wrote. Use appropriate inputs, and cover for as many edge cases as you can think of!

Hint: The library `java.lang.Math` has many methods you will find useful, such as `Math.abs()` and `Math.sqrt()`. You can also use `Math.PI` to get the value of π.

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 for your `Test` class. When you are satisfied that your code is complete and correct, create a TAR file containing your `src/` directory, as well as the PDF of your UML diagram. Name your file:

```
cse213_<firstname>_<lastname>_hw1.tar.gz
```

Upload your submission to Canvas before the due date.