Object Oriented Programming – I

Review-Part 4

MODULE – IX:

The String class

String is a pre-defined class in Java and one of the most important classes.

Strings represent text as a sequence of characters.

Strings have a special means for creating instances.

While we can create instances using the new operator (just as with any class), we can also just place the desired string inside double quotes.

"Hello" ← this is a shortcut to creating a new String using the appropriate sequence of characters as input.

Note that "Hello" acts just like the new operator, it evaluates to the address of the String object storing h,e,l,l,o.

We can use the object just like any other object. We can store it

```
String s = "Hello";
```

Strings have a special operator, the "+" operator.

The result of + is a new String that concatenates the two operands together.

If one operand is not a String, an appropriate String is created (through a sequence of object creation and method calls) before the concatenation.

Ex: "Hello" + "there" → "Hellothere"

"Hello " + "there" → "Hello there"

"x = " + 3 → "x = 3"

Note: you must be careful when mixing Strings and numeric primitives with the +. When is the + meant to be a String concatenation and when is it meant to be a normal addition?

"x = " + 3 + 5 will return "x = 35" (+ is evaluated left to right) but

3 + 5 + " = y" will return "8 = y"

Some useful methods and operators:

+: creates a new String that is the result of concatentating two Strings
length(): returns the number of characters in the String
charAt(5): returns the 6th character of the String. (In Java, the first character is at index 0.)

String s = "Hello" s.length() → returns 5 s.charAt(1) → returns 'e'
"".length() → returns 0 "Hello".charAt(0) → returns 'H' "".charAt(0) → Error

"Hello".charAt("Hello".length()) → Error because it asks for the index 5 character in a string of only 5 characters.
"Hello there".charAt(5) → returns ' ' ("hello" + "there").length() → returns 10

Two important methods of Object and Overriding methods

Employee extends Object and inherits the methods of Object. There are two important methods we should override because their default behavior is not very useful.

1) String **toString()**

This method produces a String representation of the object. It is used anytime we need a String representation of the object such as when writing to the terminal or when used with the String concatenation operator + to concatenate objects to a string.

The method in Object sets the string to be the true type name of the object and it's "hash code" (basically its location in memory).

Usually, we want the string to be something more useful, so we override the method.

In our case, we want it to be the employee number followed by the employee name

```
/*change the behavior of the inherited toString */
public String toString() {
        return getNumber() + ": " + getName();
}
```

2) boolean **equals(Object o)**

This method is used to compare two objects.

➔ If you use the == operator on non-primitive values, it is comparing the addresses of the values. Thus, == only returns true if two objects are the exact same object.

For example, "Hi" == "Hi" only returns true if the two Strings are really the same string stored at the same location in memory.

If you create a String s that contains "Hi" but is stored at a different location in memory,

then s == "Hi" will return false.

The **equals** method is provided to examine the contents of the object to determine if they are structurally equal. However, the **equals** method will not compare the contents of objects by default. The equals method in Object just does an == test**. Instead, we must override the method.** For example, the String class overrides the equals method to compare the individual characters of the string. Thus "Hi".equals("Hi") always returns true.

IMPORTANT: To override any method, we must exactly match the method parameter signature. Many Java textbooks and on-line references do not give the correct way to write an equals. If you seach online, you will often find the suggestion to use

```
public boolean equals(Employee e) {
```

but this does not match the parameter signature of the method in Object, and so it is overloading, and not overriding!
The problem with overloading is that we can change the version of the method we call by typecasting:

Employee e1 = new Employee(10, "Mekayla");
Employee e2 = new Employee(10, "Mekayla");
e1.equals(e2) and e1.equals((Object)e2) will call different versions of the method! One for input Employee, and one for input Object.

This is bad because we don't want the behavior to change just because the current type of the object changes. (In fact, the first would return true, and the second will return false!)
Because overriding is such an important idea in Java and because it is easy to make a mistake and accidentally overload instead of override, the Java compiler provides an "**annotation**" we can add to the code. An annotation is not part of the Java language, but it is a directive to the compiler.
This particular annotation is **@Override**, and if we place it before a method that is overriding an inherited method, the compiler, upon seeing the annotation, will verify that we really are overriding. If we are not, it will give a compiler error instead of allowing the overload.
A correct equals method that says this Employee is equal to the input if the input is an Employee with the same employee number and the same name:

```
@Override
/* Change the behavior of the inherited equals method. Two employee instances are the same if they have the same number and name */
public boolean equals(Object o) {
if (o instanceof Employee) {
        Employee e = (Employee)o;
        return this.getNumber() == e.getNumber() && this.getName().equals(e.getName());
}else
        return false;
}
```

instanceof: returns true if object that is the left operand can be typecast as the type that is the right operand. (I.e. is the object's true type equal or "below" the given type in the hierarchy.)
If we do not use **instanceof**, the method will generate a **TypeCastException** when typecasting a non-Employee to Employee.

This equals method has overridden the inherited equals method. And now there is only one version of equals available to Employee, and no matter how Employee is typecast, it will always use this version of the method.

Fields / Variables:

There are 4 kinds of variables, and the kind you create depends on where you create it:

1) class (static) fields: the variable is stored in the class (one copy that all the objects share)

- class fields exist for the duration of the program.
- they are given default initial values of 0, 0.0, false, or null, depending on their type
- null = "not a valid address"

2) instance (non-static) fields: a separate variable stored in each instance of the class. Every instance has its own version.
- an instance field exists as long as the containing instance exists
- they are given default initial values of 0, 0.0, false, or null

3) method parameter: the variable(s) that store an input to the method. It exists only in the body of the method.
- the initial value is set by the method call input

4) local variables: a variable declared inside a method. It exists from the moment created until the end of the compound statement it is declared it.
- they are not given an initial value, and they must be assigned a value as their first use

RULES FOR O-O CODING:

1. Create private fields
2. Create public getter/setter methods so that classes that extends this class can change behavior if they want.
3. Everywhere in our code that uses a value, we use the getter/setter methods instead of the fields.
4. Except in the constructor where, if we want a field to be initialized, we use the field directly.

O-O TYPE RULES:

1. Every instance is many types at the same time.
2. The "true" type is what the instance is at creation (from: new XXX() → the true type is XXX).
3. The "current" type is what the instance is typecast as at this particular part of the code.
4. An instance may only call methods and fields that exist for the "current" type.
5. However, the version of the method used is the version of the "true" type. (This applies only to instance methods! Fields and static methods are still used from the current type)
Today we will see why those rules greatly simplifies our coding:

Example: Let's create a method in Employee that compares employee's by how much money they make. We created the method **earnsMoreThan** that compares the salaries of this employee to another employee.o Suppose we have this:

```
public boolean earnsMoreThan(Employee e) {
    return this.getSalary() > e.getSalary();
}
```

There is a problem: whoever created Employee assumed that all employee's have salaries. This is not the case, but we often have situations in Java where the person creating a type does not think of every situation and makes incorrect assumptions on how it is used. Because the **earnsMoreThan** method is following proper O-O coding (using the getter methods instead of the fields), it is easy for other classes to adjust so that their classes properly work.

We will have every employee type define for itself what "salary" means.
For example, an hourly employee can decide that a salary is hours worked * rate per hour.

A sales employee can decide salary is number of sales * commission, and so forth.
HourlyEmployee will "define" how salary works by overriding the salary method:

```
public double getSalary() {
  return this.getHoursWorked() * this.getHourlyRate();
}
```

Because the true type version of an instance method is always used, the hourly employee will always report its salary as the product of its hourly rate and hours worked.

- it does not matter if we typecast the hourly employee
- it does not matter if we try to set the salary of the hourly employee It just works!

BAD IDEA: Use the fields directly:

```
public boolean earnsMoreThan(Employee e) {
   return salary > e.salary;
}
```

If we did this, it would take more work to get hourly employee to work with the method. Since fields are not inherited and cannot be overridden, we are going to have to make sure hourly employee calls the inherited setSalary method everytime that its hourlyrate is changed or its number of hours worked is changed.
Not only is that going to mean changes in at least two different places, it means that if Employee changes how earnsMoreThan is computed, we will have to change HourlyEmployee!

The moral: everytime you choose to not follow the O-O guidelines, you make it more challenging for other coders to extend and use your class. So, if you want to violate the O-O guidelines, you should think carefully to make sure you have a good reason to do so.

MODULE – X:

The main method.
The main method is a special method of Java. Every stand-alone program must have a main method. It is what the Java Virtual Machine calls to start your program.

Each class may have one (and only one) main method. The form is

```
public static void main(String[] args) {
```

If you have a program that contains a class with a main method, you can run your program from the command line of your computer using the Java runtime environment (JRE):

```
java DesiredClass
```

the above can be followed by 0 or more input values that will be passed to your main method through the String array parameter. (Depending on your system, you may also have to set the CLASSPATH environment property to indicate the folder where the Java files are.)

DesiredClass must have a main method in it.
See the MyFirstProgram class we wrote in the lecture. For example, try

```
java MyFirstProgram this is a test of the program
```

We can also create a standalone program by placing all the classes of the program in a jar file. (We will demonstrate this when we start building larger programs later in the second half of the course.)(A jar file is just a collection of Java .class files, and when creating the jar file, we specify which class contains the main method that should be called when launching the program.)

You can then run the jar file with

```
java -jar jarfile
```

Most operating systems will let you run a jar file by just clicking on it.

Finally, what should you place into the main method to run your program?

The main method is a static method.

The interactions pane is a static context (i.e. it acts like the body of a static method).

So, whatever you would have typed in the interactions pane to launch the program, place into the main method.

Creating a Class Hierarchy

Suppose we want to create a program that does high school geometry? We first need to identify the types of things that must be modelled in the program: lines, points, area, polygons, perimeter, triangles, squares, circles, circumference, etc..

How do we organize this into classes?

1. is-a rule: If A is-a B then we make A and B classes where A extends B.

Examples: Square is-a Rectangle.

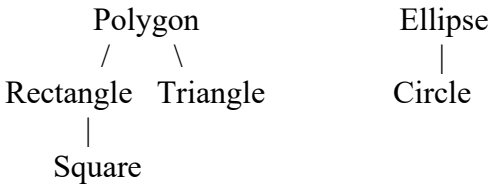
Square is-a Polygon

Triangle is-a Polygon

Rectangle is-a Shape

Circle is-a Ellipse

So, the is-a relation suggests the following hierarchy:



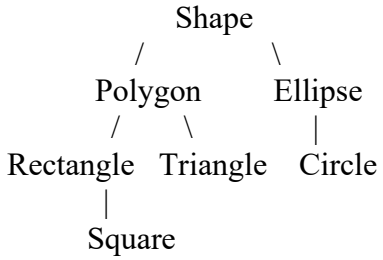
2. has-a rule: If A has-a B, then make B a method/methods in A.

Examples: Square has-a area

Polygon has-a numAngles

Circle has-a area

To use inheritance well, we want to move each attribute as high up the hierarchy as we can. Since all the above classes have an area, this suggests that we should make a common parent at the top of the hierarchy so we can define area there, and if the other classes need to change how area is defined, they can "override" the area method.



numAngles does not belong in Shape because ellipse and circle does not have sides. Instead, the highest we will move that attribute is into Polygon. We will place length and width into Rectangle as well as into Triangle. It would be nice to move them up to a common ancestor as well, but Polygon may not be the right choice. What does length and width mean for an arbitrary polygon?

Abstract Classes:

The only reason we created the Shape class was to have a common ancestor for things that all shapes have like area and perimeter. We do not want to be able to create an instance of "Shape" without specifying what type of Shape. So we can make Shape be an abstract class.

An abstract class is a class that cannot be instantiated. It is used to enforce common behavior of all its child classes.

In this case, we want to be able to say that polygons, circles, rectangles, triangles, and squares are all "shapes", and all shapes have area. To get the last behavior, we make area a method of the abstract class. However, we do not know how to compute the area without knowing what type of shape it is, so we make area an "abstract method".

An abstract method is a method with no body (i.e. it is a method stub). All classes inherit the abstract method, but unless the class is abstract, it must override the abstract method to give it a body. (Otherwise, we could call the method without defining what it does!)

```
public abstract class Shape {
  public abstract double area();
  public abstract double peremeter();
}
```

An abstract class can contain -everything- that a normal class can (including constructors!), plus it can contain abstract methods. So, I can add normal methods too.

```
public abstract class Shape {
```

```
  public abstract double area();
  public abstract double peremeter();

  public boolean isLargerThan(Shape s) {
    return this.area() > s.area();
  }
}
```

Notice that by defining abstract area inside abstract Shape, then we can use isLargerThan to compare all the types that are under Shape in the class hierarchy.

Polygon should also be an abstract class because we do not know how to define the area of an arbitrary polynomial. Polygon class will be an abstract class with a constructor. The constructor will set the number of sides to the polygon.

```
public abstract class Polygon extends Shape {
        private int numAngles;
        public Polygon(int numAngles) {
                this.numAngles = numAngles;
        }
         public int getNumAngles() {
                return numAngles;
         }
}
```

Notice that Polygon does not need to override the area method. It inherits the abstract method, and because Polygon is abstract, it is allowed to have abstract methods.

Now, we will create the Rectangle class. Note that Rectangle is required to have a constructor because the Polygon DOES NOT have a constructor that takes no input. Because Rectangle is not abstract, we must override any abstract methods Rectangle inherits. In this case, it inherits area. To compute the area, we need the length and width, and thus we should have a constructor that sets those values.

```
public class Rectangle extends Polygon {
  private double length;
  private double width;

  public Rectangle(double length, double width) {
    super(4);
    this.length = length;
    this.width = width;
  }

  /* include getter and setter methods for length and width */
```

```
  @Override
  public double area() {
    return getLength() * getWidth();
  }

  @Override
  public double getPermeter() {
    return 2 * (getLength() + getWidth());
  }

}
```

Note: why did we use "this.length = length" instead of "setLength()" in the constructor? Because "setLength()", or rather "this.setLength()" may not call the setLength method in Rectangle.

It will call the setLength method of the true type. As a result, the length field might not get set. To make sure the length and width fields are always set, we need to put the assignment statements in the constructor and not call methods that could be overridden.

Note that we are using the getter methods inside the body for area. Doing so will make our work for Square (and any other class that extends Rectangle) a lot simpler. Square will not have to change how area is computed because Square will simply override the getter and setter methods to make sure that a square has the same width as length.

```
public class Square extends Rectangle {
  public Square(double size) {
    super(size, size);
  }

  /* need to override setLength and setWidth to make sure that this object is always has equal width and length (i.e. it is always a square) */
}
```

There are two ways to do this:

Version 1: Override the setWidth and setWidth methods so that both the width and length are always set the same

```
  @Override
  public void setWidth(double width) {
    super.setWidth(width);
    super.setLength(width);
  }

  @Override
  public void setLength(double length) {
    this.setWidth(length);
  }
```

➔ Note the use of "super" above in super.getWidth(width); super is a special word that means to use the method (or field) of the parent class. This is the only way we can have a class call a method that was overridden. If we did not use "super", then the true type's version of setWidth would be called.

Version 2: We will use only the Rectangle's length to determine both the length and width of Square

```
@Override
public void setWidth(double width) {
  this.setLength(width);
}

@Override
public double getWidth() {
  return this.getLength();
}
```

(In this version, we can change the constructor to be: super(size, 0); because we are using only the length of Rectangle for the square and not the width)

Which method is better? Neither! They both work the same. However, note that if we had Rectangle's area method use "length * width" instead of "getLength() * getWidth()", then Version 2 would not work!

That does not mean Version 1 is better because there are other things we could have done in Rectangle using fields instead of methods that could have broken Version 1 as well.

SUMMARY: IMPORTANT RULES FOR PROPER OBJECT-ORIENTED PROGRAMMING IN JAVA

1. Form a hierarchy using the "is-a" and "has-a" rules.

2. "is-a":

2a. If type B "is-a" type A, then we make B extend A. You must be sure that everywhere type A is used type B can also be used. If that is not true, then B is not also type A.

For example, we can use Square anywhere Rectangle is expected, and it makes sense. Likewise, we can use Square and Rectangle anywhere that expects a Shape, and that makes sense. It would not make sense to use Triangle anywhere that expects a Rectangle.

2b. If B extends A, then B should add features to A or restrict features of A. B should not block or remove features of A.

Square does not block the ability to change the width or length of Rectangle, but it restricts the change so that both dimensions change together.

3. "has-a":

3a. Use "has-a" to decide on the methods for the class, and push common methods as high up the hierarchy as possible.

3b. Use private fields to store the data and public or protected getter/setter methods as needed.

3c. Do not save the same information multiple times. Remember that all fields of the class and parent classes exist in the object. Use the getter/setter methods to access the values of the parent class.

4. Outside of the constructors, always use the getter/setter methods instead of the fields when accessing values of an object.

This will allow any classes that extend your class to easily override methods to specify necessary behavior.

(If you do not want public getter/setter methods, then at least use protected ones.)

5. Do not use getter/setter methods inside the constructor. The purpose of the constructor is to initialize the object correctly, and if you use methods, a class that extends your class could override the

methods and accidentally break how your class works by preventing proper initialization of the instance.

==========

Multiple Inheritance

Some object-oriented languages such as C++ allow a class to have more than one parent. For example, class C can extend both classes A and B. This means C will inherit methods from both A and B. What if both A and B have a method m(), and inside class C we call method m()? Whose method is called, A's or B's?

This becomes really tricky if A and B both extend a super class X. Suppose X has method m() and A and B both override m().

Now consider: X x = new C();

x has current type X so x.m() is a legal method call.

But the true type C determines which version of m is called. So which version is it, A's or B's?

In Java, each class can only extend one other class. Thus there is no ambiguity. We can get something like multiple inheritance by allowing a class to implement more than one interface.

Why Do We Need Multiple Inheritance?

Consider the Shape hierarchy from the last lecture.

What if we want to add a RegularPolygon to the hierarchy?

A RegularPolgon is-a Polygon

A Square is-a RegularPolygon

A Rectangle is not a RegularPolygon

A Hexagon is a RegularPolygon

There is no way to correctly place RegularPolygon in the tree hierarchy for the classes and have all the "is-a" relations correct.

Instead, let use create RegularPolygon as an interface. Now, we can have classes like Octagon and Hexagon extend Polygon, and have Octagon, Hexagon and Square "implement" RegularPolygon.

MODULE – XI:

Java Interfaces

A Java interface is a non-primitive type like a class, but it cannot contain instance methods, fields, or constructors.

Specifically, an interface can contain:

- static final fields
- static public nested types
- static methods
- abstract public instance methods (method stubs)
- starting in Java 9, private methods will be allowed

The main purpose is to contain public abstract methods.

To create an interface:

```
public interface MyInterface {
        void methodStub1(int x, int y);
        int methodStub2();
}
```

(since all methods stubs in an interface have to be public and abstract, we can drop the "**public abstract**" part)

An interface can extend 0 or more interfaces. (You place "extends ..." just like you do with a class, and for multiple interfaces, you separate them with commas.)

To use an interface in a class:

a class can implement 0 or more interfaces.

```
public class MyClass implements MyInterface {
        // here MyClass inherits the abstract methods methodStub1 and methodStub2
       // because a class cannot have abstract methods, we must override these method stubs with
methods with bodies
}
```

Here is another example:

```
public class Square extends Rectangle implements RegularPolygon {
  ....
}
```

If you will implement more than one interface, separate the interface names with commas. Implementing an interface is -exactly- like extending a class. So, a class that implements an interface inherits all the methods (or in this case method stubs) of the interface.

A class (that is not abstract) cannot contain method stubs. So, the class must override each of the method stubs from the interface.

Note how this simplifies the multiple inheritance problem above. A class may inherit a method stub from more than one parent, but it can only inherit an instance method that contains a body from its class parent.

NOTE: Students tend to get mixed up on "extends" vs. "implements". They mean exactly the same thing. Java uses "**implements**" for the specific situation where we are indicating that a class as an interface as a supertype. All other situations where we are indicating a supertype, we use "**extends**".

Interface Example
We created the RegularPolygon interface

```
public interface RegularPolygon {
```

The first thing we added is a method to compute the area of a regular polygon. There are two ways we could do that. Since interfaces allow static methods, we did so:

```
    public static double areaOfRegularPolygon(RegularPolygon p) {
      ... a formula using p.getNumberAngles() and p.getSideLength()
    }
```

We could also make a non-static abstract method, but give it a default method body:

```
    public default double getArea() {
       ... a formula using this.getNumberAngles() and this.getSideLength()
    }
```

Note one benefit of interfaces: we can create a method that only works on RegularPolygons by specifying that as an input. Since interfaces are a non-primitive type, we can make the current type of an object be a RegularPolygon as long as the true type implements the RegularPolygon interface.

Next, note that we need the instance methods getNumberAngles and getSideLength. How do we make sure that RegularPolygon has them? We add them as abstract methods.

Since all instance methods of an interface MUST be public and abstract, the Java style is to drop those two modifiers (but you can keep them if you want).

```
public interface RegularPolygon {
    int getNumberAngles();
    double getSideLength();

    public static double areaOfRegularPolygon(RegularPolygon p) {
```

```
        ... a formula to compute the area using getNumberAngles and getSideLength ....
      }

      public default double getArea() {
        ... a formula using this.getNumberAngles() and this.getSideLength()
      }
    }
```

Now, anything that is a RegularPolygon will inherit the getNumberAngles and getSideLength method stubs and (if not an abstract class), must override them to make them normal methods.

Notice one trick I used. Polygon already has getNumberAngles as a normal method. So, anything that implements RegularPolygon will already get a normal getNumberAngles method inherited from Polygon and so will not have to override the method stub.
The classes that implement RegularPolygon will still have to override the getSideLength method stub.

```
    public class Hexagon extends Polygon implements RegularPolygon {

      private double sideLength;

      public Hexagon(double sideLength) {
        super(6);
        this.sideLength = sideLength;
      }

      public double getSideLength() {  ← this method is required by RegularPolygon so I decided to name my getter/setters appropriately
        return sideLength;
      }

      public void setSideLength(double sideLength) {
        this.sideLength = sideLength;
      }

      public double perimeter() {       ← this method is required by Shape
        return getNumberAngles() * getSideLength();
      }

      public double area() {          ← this method is require by Shape, but we have a static method in the interface for the area
        return RegularPolygon.areaOfRegularPolygon(this);    ← remember that "this" is a variable storing the object this method is acting on
      -OR-
        return this.getArea();
      }
    }
```