Abstract Classes

In this lecture we will discuss:

- Review of Inheritance
- Review of Polymorphism
- Review of Interfaces
- Introduction of Abstract classes

Your future in CS

I used to include this on my slides, but since these slides have changed - going to just leave it up here for every notebook. I get a lot of questions about more programming courses, the concentrations, and minors in computer science. Here is a brief reminder.

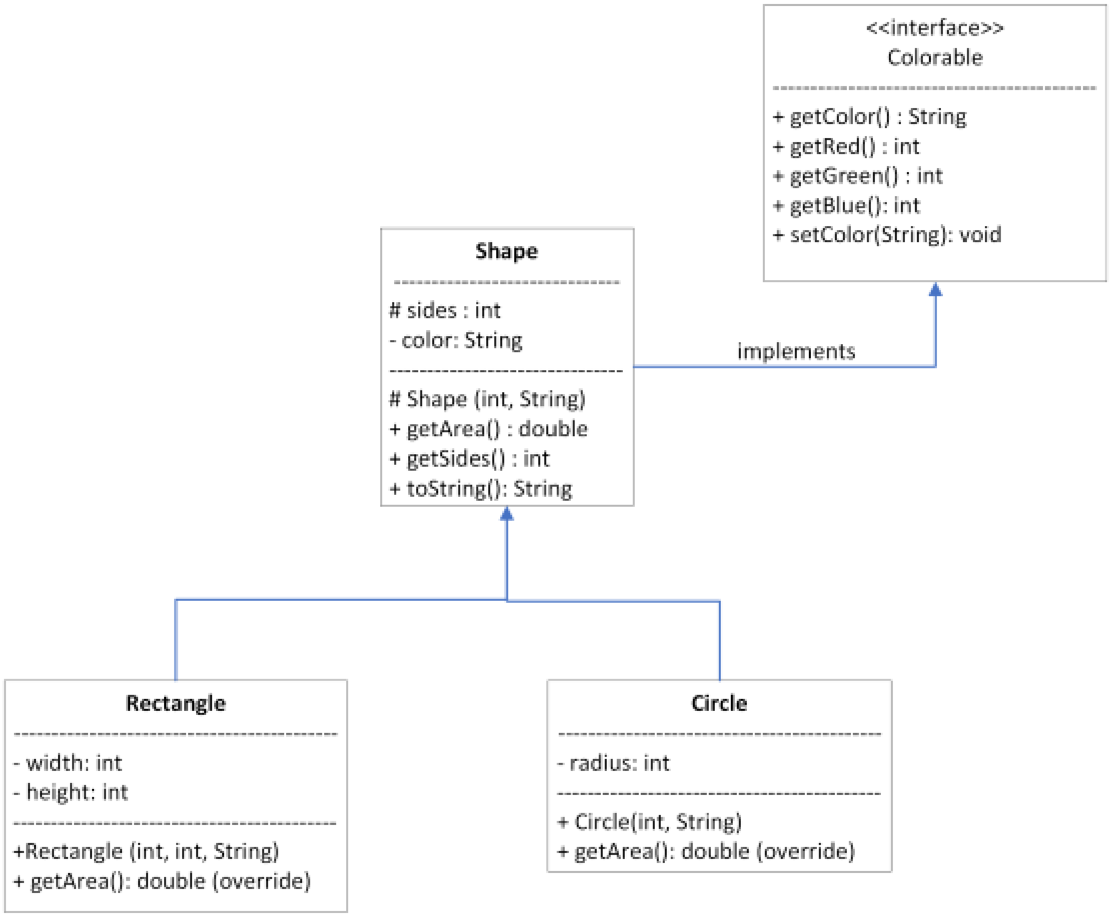
CS 165 – Next Course In Sequence, also consider CS 220 (math and stats especially)

- CO Jobs Report 2021 – 77% of *all* new jobs in Colorado require programming
- 60% of all STEM jobs requires *advanced* (200-300 level)
- 31% of all Bachelor of Arts degree titled jobs also required coding skills
- 2016 Report found on average jobs that require coding skills paid $22,000 more

- Concentrations in CS:
  - Computer science has a number of concentrations.
    - General concentration is the most flexible, and even allows students to double major or minor pretty easily.
    - Software Engineering
    - Computing Systems
    - Human Centered Computing
    - Networks and Security
    - Artificial Intelligence
    - Computer Science Education.
  - Minors:
    - Minor in Computer Science - choose your own adventure minor
    - Minor in Machine Learning - popular with stats/math, and engineering
    - Minor in Bioinformatics - Biology + Computer Science

Inheritance and Polymorphism

Recall, in the polymorphism lecture last week:



We implemented this structure.

> Discussion
>
> Take a moment to discuss the example. Define the following elements:
>
> - Interface
> - Methods that are overwritten
> - Superclasses
> - Subclasses
> - (looking at your implementation) Key word to call the superclass constructor that was used?
> - What methods are being used in toString()?
> - Does getArea() really make sense for Shape?
>   - what does that really mean?
>
>   Also use this time to get caught up on the code from last week if you never finished it at your table!

In [1]:

```
public interface Colorable {
    public String getColor();
    public int getRed();
    public int getGreen();
    public int getBlue();
    public void setColor(String color);
}
```

In [8]:

```
public class Shape implements Colorable {
    protected int sides;
    private String color;
    public Shape(int sides, String color) {
        this.sides = sides;
        setColor(color);
    }
    public double getArea() { return 0;}
    public String getColor() { return color;}
    public int getSides() {return sides;} // added from the UML
    public void setColor(String color) {this.color = color;}
    public int getRed() {return Integer.parseInt(color.substring(0, color.indexOf(",")
    public int getGreen() {return Integer.parseInt(color.substring(color.indexOf(",")+
                                                   color.lastIndexOf(",")));}
    public int getBlue() { return Integer.parseInt(color.substring(color.lastIndexOf("
    public String toString() {
        return String.format("Sides: %d, Area: %.2f", sides, getArea());
    }
}
```

In [3]:

```
public class Rectangle extends Shape {
    private int width;
    private int height;
    public Rectangle(int width, int height, String color) {
        super(4, color);
        this.width = width;
        this.height = height;
    }
    public double getArea() {
        return width*height;
    }
}
```

In [4]:

```
public class Circle extends Shape {
    private int radius;
    public Circle(int radius, String color) {
        super(1, color);
        this.radius = radius;
    }
    public double getArea() {
        return Math.PI * (radius * radius);
    }
    public int getDiameter() { return radius * 2;} // this was added at the end of the
}
```

Definition Review:

- Inheritance:
    - Creates an *is-a* relationship between classes
        - Used to keep your code DRY
        - Allows fully implemented 'more generalized' classes as the super classes
            - specialized subclasses as the subclasses

        - uses the key word **extends**
            - can only extend / inherited from one immediate parent (but can have 'chain' of parents)
    - Example:
        - A circle gains the properties of shape including and implemented methods

In [10]:

```
Circle crcl = new Circle(10, "234,255,123");

System.out.println("The color is " + crcl.getColor());
```

```
The color is 234,255,123
```

Interfaces

- Interfaces
    - Define what needs to be implemented
    - But they provide no actual implementation
        - Can't hold state
        - Can't have private methods or variables
        - There is something called a default method or static in an interface - we don't explore those in this class. (reference)
    - Think of them as a recipe that must be followed
    - uses the key word **implements**
        - can implement more than one Interface

Polymorphism

- Allows the subclass to be declared as the super
    - actually a subclass can 'substitute' in for the super
- Extremely useful for things like Arrays and ArrayLists
- Useful on overall class design

In [11]:

```
Shape[] shapes = new Shape[3]; // fixed size
shapes[0] = crcl;
shapes[1] = new Rectangle(23, 5, "123,125,255");

System.out.println(Arrays.toString(shapes));
```

```
[Sides: 1, Area: 314.16, Sides: 4, Area: 115.00, null]
```

Abstract Classes

- Going back to a discussion question
    - Does it make sense for Shape to have `.getArea()` ?
    - Not really?
    - *but* getArea() is used in shape!
- Do we ever really initialize a shape by itself?

- Not really, as the idea isn't very concrete for what we are doing.
- Abstract classes to the rescue!
  - Allows for most methods to be implemented
  - Allows for some methods to be only a definition but not implemented
    - forces inheriting classes to implement them before they will compile!

In [18]:

```java
public abstract class ProcessData {
    protected final List<Integer> data = new ArrayList<>();

    public ProcessData(String filename) {
        loadDataFromFile(filename); // notice I am calling a method that isn't impleme
    }

    public int getSum() {
        int sum = 0;
        for(Integer val : data) {
            sum += val;
        }
        return sum;
    }

    abstract protected void loadDataFromFile(String filename);  // no implementation,

}
```

In [32]:

```java
public class ProcessCsvData extends ProcessData {

    public ProcessCsvData(String filename) {
        super(filename);
    }

    protected void loadDataFromFile(String filename) {
        FileInputStream in;
        try {
           in = new FileInputStream(filename);
        }catch(FileNotFoundException ex) {
            System.err.println("File not found! " + ex.getMessage());
            return; // leave the method early
        }
        Scanner scn = new Scanner(in);
        scn.useDelimiter(",");
        while(scn.hasNext()) {
           if(scn.hasNextInt()) data.add(scn.nextInt());
           else scn.next();
        }
    }
}
```

In [33]:

```java
public class ProcessTxtData extends ProcessData {

    public ProcessTxtData(String filename) {
        super(filename);
    }

    protected void loadDataFromFile(String filename) {
        FileInputStream in;
        try {
           in = new FileInputStream(filename);
        }catch(FileNotFoundException ex) {
            System.err.println("File not found! " + ex.getMessage());
            return; // leave the method early
        }
        Scanner scn = new Scanner(in);
        while(scn.hasNextLine()) {
            String line = scn.nextLine().trim();
            data.add(Integer.parseInt(line));
        }
    }
}
```

In [35]:

```
ProcessData data_one = new ProcessCsvData("data/output.csv");
ProcessData data_two = new ProcessTxtData("data/output.txt");

System.out.println(data_one.getSum());
System.out.println(data_two.getSum());
```

```
45
45
```

Abstract class discussion

- The superclass can call a method implemented in the subclass. (this is major!)
- You will not need to design thinking about this for a bit, but very powerful

In class activity

- Take the `Shape` class and make it abstract
- Make `getArea()` abstract
- Run the current code (shouldn't change much)
- Add an additional class called Triangle.java
  - Implement the needed constructor and method
  - as a reminder, triangle area is: $\frac{(base*height)}{2}$
- Compile between different stages to see what happens if you try compiling without implemented .getArea()

In [37]:

```
public abstract class Shape implements Colorable {
    protected int sides;
    private String color;
    protected Shape(int sides, String color) {
        this.sides = sides;
        setColor(color);
    }
    abstract public double getArea();
    public String getColor() { return color;}
    public int getSides() {return sides;} // added from the UML
    public void setColor(String color) {this.color = color;}
    public int getRed() {return Integer.parseInt(color.substring(0, color.indexOf(",")
    public int getGreen() {return Integer.parseInt(color.substring(color.indexOf(",")+
                                                            color.lastIndexOf(",")));}
    public int getBlue() { return Integer.parseInt(color.substring(color.lastIndexOf('
    public String toString() {
        return String.format("Sides: %d, Area: %.2f", sides, getArea());
    }
}
```

In [42]:

```
public class Triangle extends Shape {
    int base;
    int height;

    public Triangle(int base, int height, String color) {
        super(3, color);
        this.base = base;
        this.height = height;
    }

}
```

```
|   public class Triangle extends Shape {
|       int base;
|       int height;
|
|       public Triangle(int base, int height, String color) {
|           super(3, color);
|           this.base = base;
|           this.height = height;
|       }
|
|   }
Triangle is not abstract and does not override abstract method getArea() in Shape
```

In [45]:

```
public class Triangle extends Shape {
    int base;
    int height;

    public Triangle(int base, int height, String color) {
        super(3, color);
        this.base = base;
        this.height = height;
    }

    public double getArea() { return (base * height) / 2.0;}

}
```

In [46]:

```
List<Shape> shapes = new ArrayList<>();

shapes.add(new Circle(10, "233,234,223"));
shapes.add(new Circle(12, "203,134,133"));
shapes.add(new Rectangle(10, 20, "123,253,292"));
shapes.add(new Rectangle(15, 5, "123,253,292"));
shapes.add(new Triangle(10, 20, "193,153,202"));
shapes.add(new Triangle(15, 5, "123,53,12"));

for(Shape s : shapes) {
    System.out.println(s);
}
```

```
Sides: 1, Area: 314.16
Sides: 1, Area: 452.39
Sides: 4, Area: 200.00
Sides: 4, Area: 75.00
Sides: 3, Area: 100.00
Sides: 3, Area: 37.50
```

Overview

- You now have three different ways to look at objects:
    - class - everything is fully implemented
    - interface - nothing is implemented, but provides definitions of what to implement
    - abstract class - some things are implemented (most actually), but provides definitions of things it needs implemented to work.