Polymorphism

- Topics for this lecture
    - Review of inheritance
    - Polymorphism

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

Four Pillars of Object Oriented Programming

- Abstraction
    - You have practiced this already by building 'general' classes/types
    - You can reuse these types in a variety of different ways
- Encapsulation
    - You have practiced this by

        - Making classes wrap/focus on a single idea
        - Controlled entry points into that class (public / private control)
- Inheritance
    - The ability to create an **is a** relationship between classes
    - Classes inherit from other classes, gaining their properties
- Polymorphism
    - Intrinsically connected to Inheritance
    - Allows subclasses to fill the role of super class
        - Barbara Liskov - type hierarchies
        - Liskov Substitution Principle (named after her and her ideas, she didn't name it)

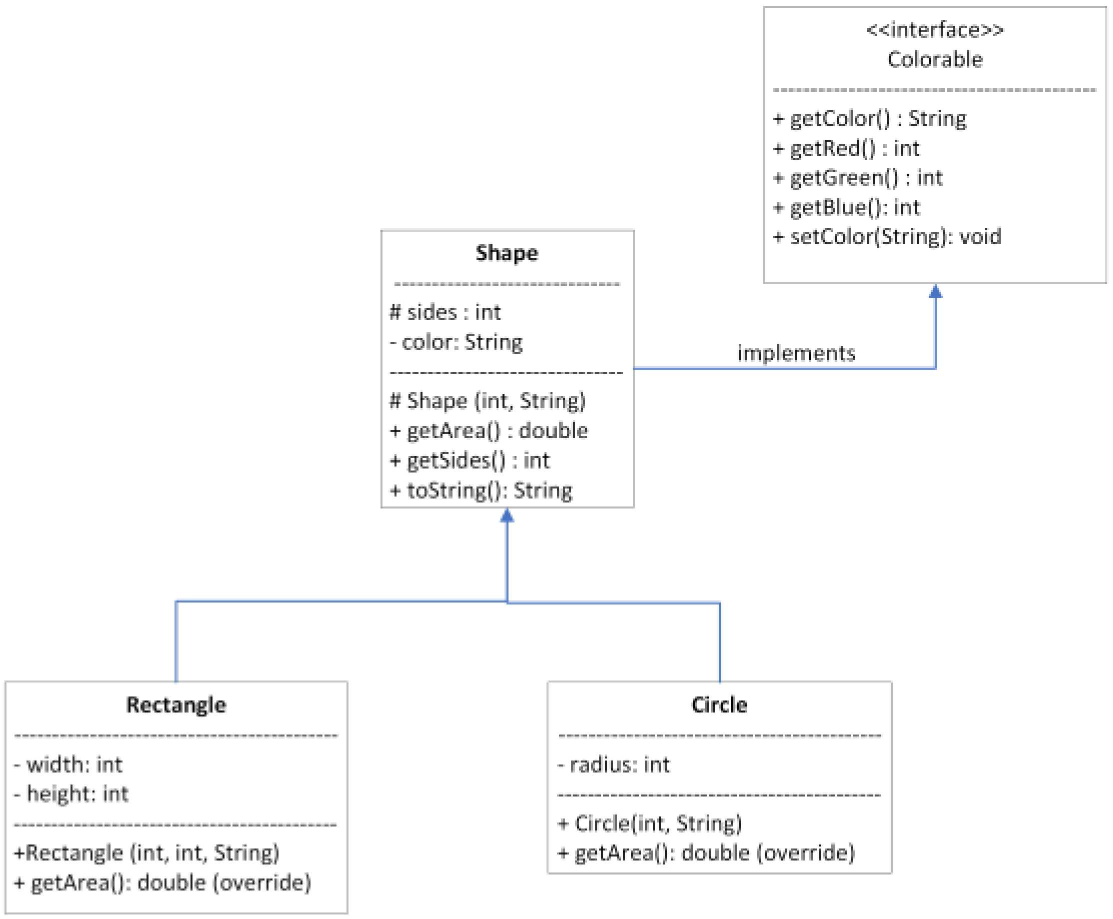
These are nice technical interview questions, but not something we talk about daily. Understanding them helps you understand the "why" on how java is designed (every class can be an object, and one class per file embodies encapsulation!)

Inheritance Review

- Keep your programs DRY
- Super/Parent classes
- Subclass/Child Class extends parent
- Can only extend one immediate parent
- Subclass Classes
    - gain public, protected, package-protected methods and variables of parent
- is-a relationship

In Class Practice

Based on the following UML scheme, write the classes that match them!



- A couple points to consider:
  - toString() - do whatever makes sense. Maybe the number of sides and area?
  - Recall that interfaces are single lines of code that define the 'contract' of what to write
    - start there!
  - Build up
- UML Reminder
  - # means protected
  - - means public
  - - means private
  - the return type is after the colon :
  - So this line in the interface

    ```
    + getColor(): String
    ```

  - becomes

    ```
    public String getColor();
    ```

- Color is stored as a string of red,green,blue.
  - You can assume it is correct / matches that format
  - getRed(): int, would grab the substring of red (before the first comma)
    - and then use Integer.parseInt(val) to return the int value of red
- Only need one programmer for the table, but this can take time to write.
  - I provided two of the four classes in zybooks under the in class lab for the week

In [2]:
```
public interface Colorable {
    public String getColor();
    public int getRed();
    public int getGreen();
    public int getBlue();
    public void setColor(String color);
}
```

In [10]:
```
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 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 [11]:
```
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 [13]:
```
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);
    }
}
```

Practice continued

- Now create three objects.
- Two rectangles, and one Circle.
    - Store all three into a single ArrayList...

  - Wait! is that possible?

Polymorphism

- Strengthening the **is-a** relationship
- We can 'cast' subclasses to be the superclass
- Examples:

In [19]:
```
Shape circle = new Circle(5, "255,255,255");
Shape rec1 = new Rectangle(10, 15, "128,255,124");
Shape rec2 = new Rectangle(30, 4, "64,23,34");

System.out.println(circle);
System.out.println(rec1);
System.out.println(rec2);
```

```
Sides: 1, Area: 78.54
Sides: 4, Area: 150.00
Sides: 4, Area: 120.00
```

In [20]:
```
List<Shape> shapes = new ArrayList<>(); // notice List!, not ArrayList
shapes.add(circle);
shapes.add(rec1);
shapes.add(rec2);

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

```
[Sides: 1, Area: 78.54, Sides: 4, Area: 150.00, Sides: 4, Area: 120.00]
```

> List?
> List is an interface used for multiple types of lists in java. Best practice is to cast ArrayList as List unless you need *specific* features of ArrayList, that other lists don't share.

What if the child has methods not in the parent?

In [24]:
```
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;}
}

List<Shape> shapes = new ArrayList<>();
shapes.add(new Circle(5, "255,255,255"));
shapes.add(new Rectangle(10, 15, "128,255,124"));
shapes.add(new Rectangle(30, 4, "64,23,34"));

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

```
Sides: 1, Area: 78.54
Sides: 4, Area: 150.00
Sides: 4, Area: 120.00
```

In [25]:
```
int diameter = shapes.get(0).getDiameter();
```

```
|   int diameter = shapes.get(0).getDiameter();
cannot find symbol
  symbol:   method getDiameter()
```

Why?

The compiler is looking for a Shape, it only becomes a Circle when we need to, so it doesn't see the method .getDiameter(). To fix this - **casting** to the rescue!

Casting

- we have been using casting to convert between primitive types

  ```
  int val = 67;
    char letter = (char)val;
  ```

- However casting to between objects helps:
  - Tell the compiler you know what you are doing
  - Forces the compiler to treat an object like another object
  - ONLY valid if going between super and subclasses.

In [27]:
```
Circle tmp = (Circle)shapes.get(0);
System.out.println(tmp.getDiameter());

// could also do it in one line, but can be messy
System.out.println(((Circle)shapes.get(0)).getDiameter());
```

```
10
10
```

instanceOf

A reserved word / command to determine if an object is an "instance of" a class.

For example:

In [29]:
```
boolean isCircle = shapes.get(0) instanceof Circle;
boolean isRectangle = shapes.get(0) instanceof Rectangle;

System.out.println(isCircle);
System.out.println(isRectangle);
```

```
true
false
```

Practice

- Take the array list of circles and rectangles
- Loop through the items and only print Rectangles
- Further practice
    - Loop through the classes, and only print items whose red values are greater than 64

In [31]:
```
for(Shape s : shapes) {
    if(s instanceof Rectangle) System.out.println(s);
}

System.out.println("Color blind safe objects...");
for(Shape s : shapes) {
    if(s.getRed() > 64) System.out.println(s);
}
```

```
Sides: 4, Area: 150.00
Sides: 4, Area: 120.00
Color blind safe objects...
Sides: 1, Area: 78.54
Sides: 4, Area: 150.00
```

Overview

- Polymorphism
    - Essential to good programming
    - Adds a lot of power to how you store objects
    - Also adds flexibility in your parameters
    - For example:
        - What if the only method I require is .getRed()?
        - I can set my parameter to be Colorable!
        - This allows any object that meets that interface to use the method
        - Reduced the number of variations I care to write about
- With all that said:
    - This takes practice, and you won't be expected to 'design from scratch' at this stage.