SWEN 601 Software Construction

Abstract Classes, & Interfaces

Next Two Weeks

WEEK 05	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #8		Quiz #9		
LECTURE			Interfaces & Abstract Classes		Unit Testing & Incremental Development		
HOMEWORK	Hwk 8 Due (**11:30pm**)		*Hwk 9 Assigned*		*Hwk 10 Assigned*	Hwk 9 Due (**11:30pm**)	

WEEK 04	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #10		Quiz #10		
LECTURE			Recursion & Binary Search		Sorts & Complexity		
HOMEWORK	Hwk 10 Due (**11:30pm**)		*Hwk 11 Assigned*		*Hwk 12 Assigned*		

Activity: Getting Started

We are trying something new!

1. Begin by accepting the Homework 9 GitHub Classroom invitation. *You should notice that the project already contains some code.*
2. Create a package `session11` package. This is where you will write your solutions to today's activities.
3. Create a `homework09` package. This is where you will implement your solution to the homework.

When you submit your homework, you will include your activities. You may earn up to a 10% bonus on the homework if you have completed all of the activities.

Do not submit code that ***does not compile***. Comment it out if necessary.

A Quick Recap of our Shapes

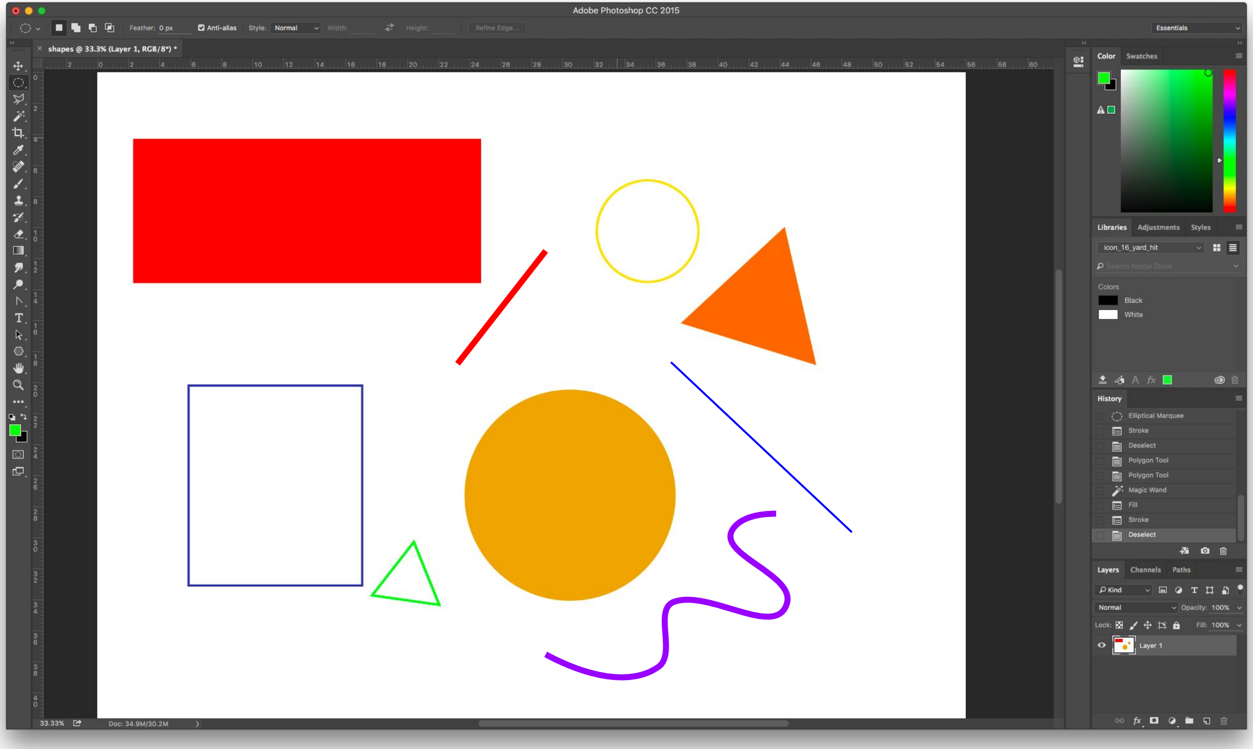
A Simple Drawing Program

Many of you have probably used a drawing program before.

Something like Photoshop, Illustrator, GIMP, Paint.net, etc.

Later this semester we will write a simple drawing program that allows users to draw simple shapes and lines.

Let’s think about the different shapes you might need to implement as classes...



Adobe Photoshop CC 2015

shapes @ 33.3% (Layer 1, RGB/8*) *

Feather: 0 px

☑ Anti-alias

Style: Normal

Normal

Opacity: 100%

Fill: 100%

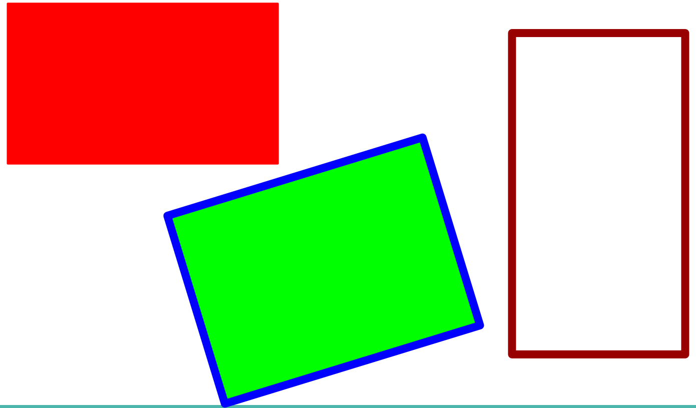
33.33%

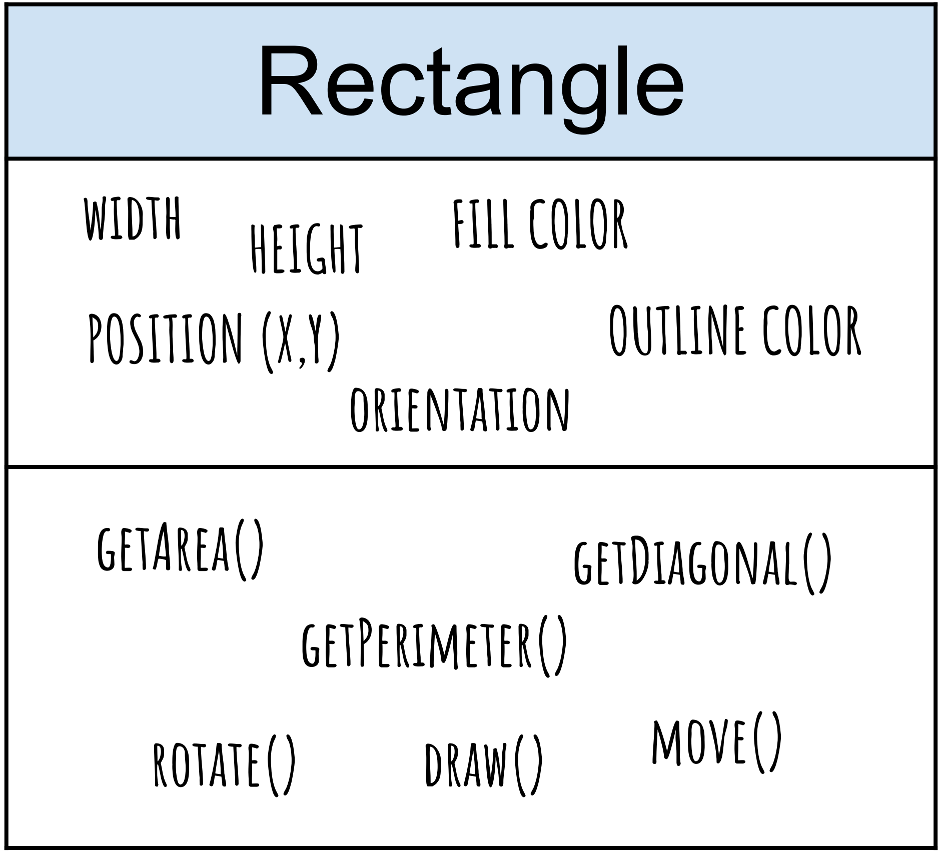
Doc: 34.9M/30.2M

A Rectangle

Thinking in terms of drawing shapes on a digital canvas, what state and behavior might a Rectangle have?

Keep in mind that there may be several different styles of rectangle that the user might want to draw.





Rectangle

WIDTH

HEIGHT

FILL COLOR

POSITION (X,Y)

OUTLINE COLOR

ORIENTATION

GETAREA()

GETDIAGONAL()

GETPERIMETER()

ROTATE()

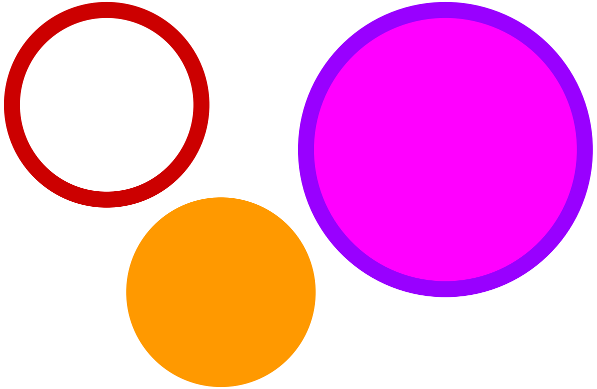
DRAW()

MOVE()

A Circle

Thinking in terms of drawing shapes on a digital canvas, what state and behavior might a Circle have?

Keep in mind that there may be several different styles of circle that the user might want to draw.



Circle

RADIUS

FILL COLOR

OUTLINE COLOR

POSITION (X,Y)

GETAREA()

GETDIAMETER()

GETPERIMETER()

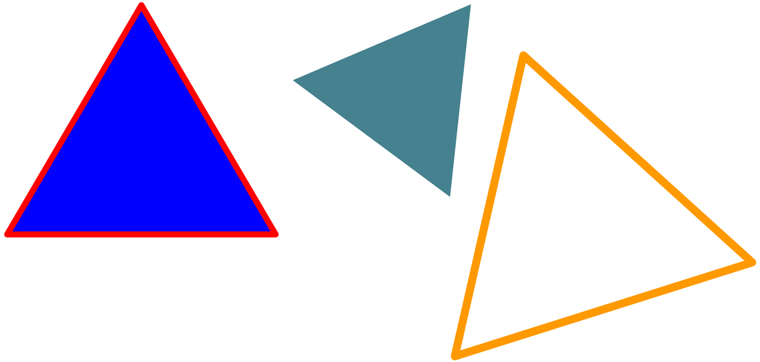
MOVE()

DRAW()

A Triangle

Thinking in terms of drawing shapes on a digital canvas, what state and behavior might a triangle have?

Keep in mind that there may be several different styles of triangle that the user might want to draw.



Triangle (Equilateral)

SIDE LENGTH FILL COLOR
OUTLINE COLOR
POSITION (X,Y)
ORIENTATION

GETAREA() GETHEIGHT()
DRAW()
GETPERIMETER()
ROTATE() MOVE()

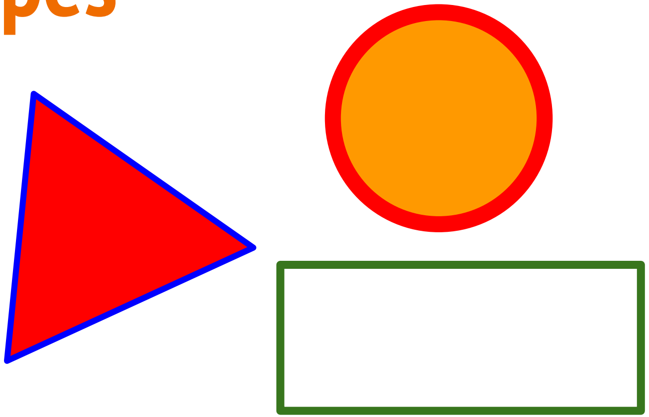
Shapes

You should recall that...

...they share a lot of the same state and behavior.

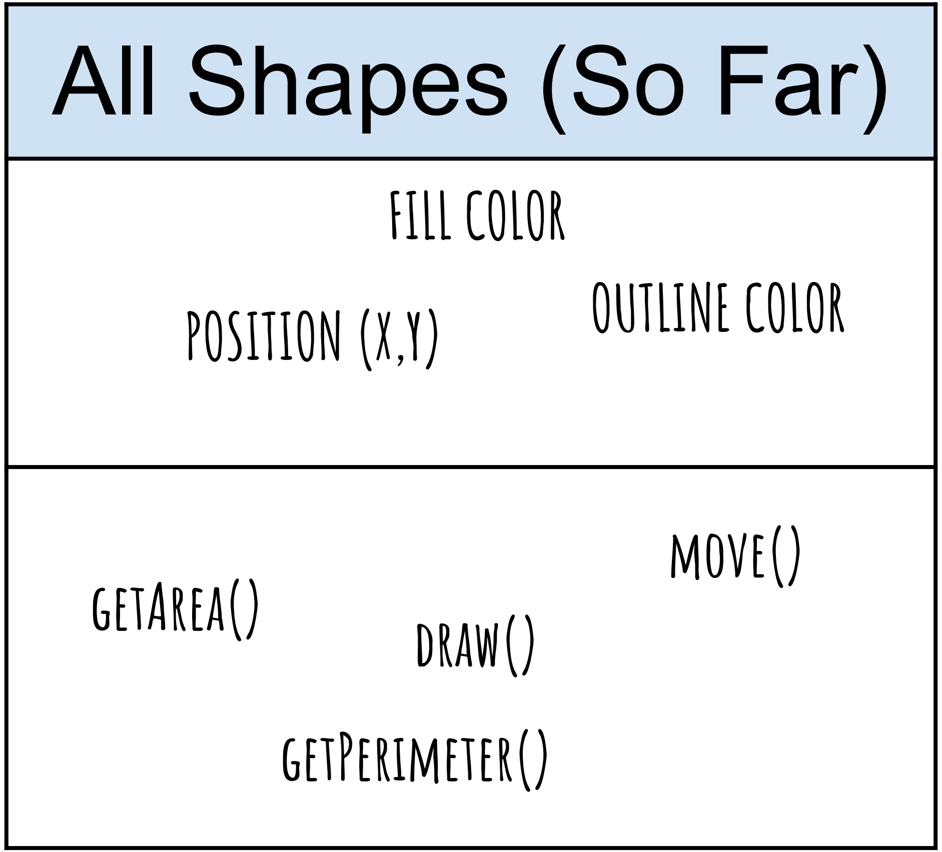
Rectangle	Circle	Triangle
double width, height Position pos // top left String fillColor String outlineColor double orientation	double radius Position // center String fillColor String outlineColor	double sideLength Position // corner String fillColor String outlineColor double orientation
double getDiagonal() double getArea() double getPerimeter() void draw() void move(int x, int y) void rotate(double angle)	double getDiameter() double getArea() double getPerimeter() void draw() void move(int x, int y)	double getHeight() double getArea() double getPerimeter() void draw() void move(int x, int y) void rotate(double angle)

Shapes



It's clear that all shapes have a significant number of members (fields and methods) in common.

Wouldn't it be useful if we could put this code in *one place* and reuse it in *multiple classes*?



All Shapes (So Far)

FILL COLOR

POSITION (X,Y)

OUTLINE COLOR

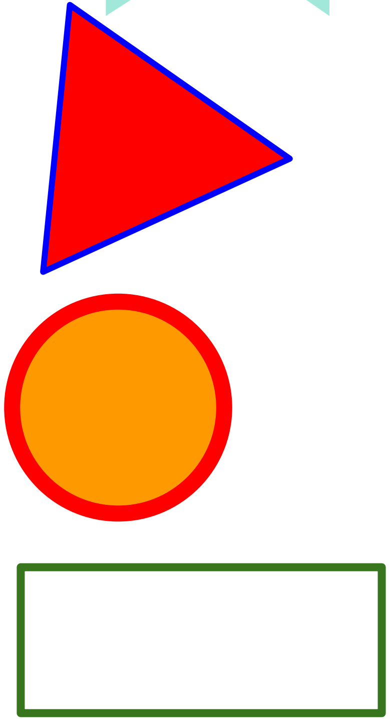
MOVE()

GETAREA()

DRAW()

GETPERIMETER()

Activity: Shape Classes



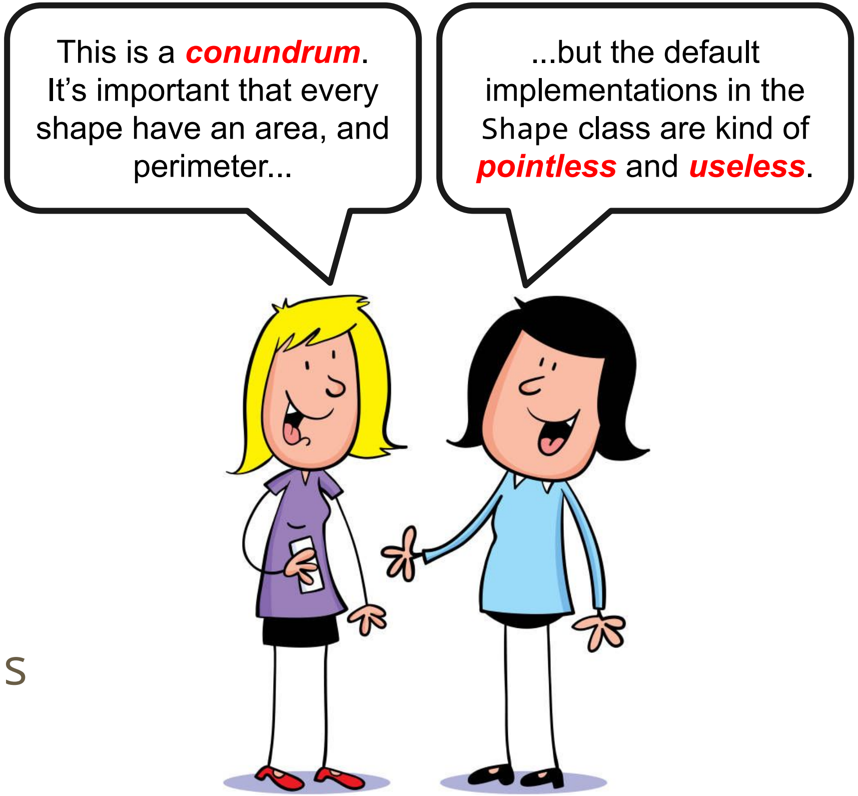
We will be modifying the code that you wrote last time throughout today's lecture.

1. Take a moment to examine the code in the `session09` package. You should note that it is similar to the code that you wrote including the `Position`, `Shape`, `Rectangle`, `Circle`, `Triangle`, and `ShapeMover` classes.
2. Copy the classes and paste them into your activities package for this session.
3. When you are finished, commit and push it to your repository.

Now, on to the new Stuff...

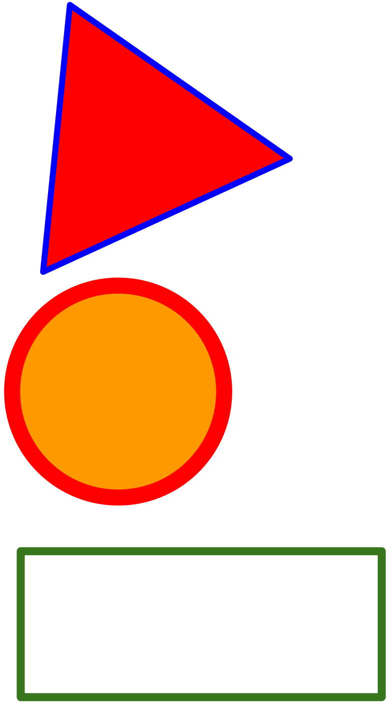
The Shape Conundrum

- Let's talk about the `Shape` class.
- It provides lots of *useful* stuff…
  - Common **state** like position, fill color, and line color.
  - Common **behavior** like move and draw.
  - It is the parent class of Triangle, Rectangle, and Circle, and so we can leverage polymorphism.
- But some of the stuff that it provides is *not so useful*.
  - The `getArea()` method always returns 0.
  - The `getPerimeter()` method does, too.


This is a conundrum. It's important that every shape have an area, and perimeter...
...but the default implementations in the Shape class are kind of pointless and useless.

So what happens if we remove them from the `Shape` class?

Activity: Removing Pointless Methods



1. Remove the “pointless” methods from the Shape class by ***commenting them out***.
   a. This include the `area()` and `perimeter()` methods.
2. Take a moment to examine the effects that this has on the rest of the code.
   a. What happened to the `Rectangle`, `Triangle`, and `Circle` classes? How can this be fixed?
   b. What happened to the `ShapeMover` class? Why?

Pointless Methods

- On the one hand, we want to be able to use polymorphism whenever possible.
  - We can't write code that assumes that all shapes have `getArea()`, and `getPerimeter()` methods if the `Shape` class doesn't define those methods.
- But on the other hand, we don't want pointless methods that don't do anything useful.
- Furthermore, do we want to create instances of the `Shape` class? Does that make any sense at all?

Remember, the ***reference type*** is the type used in a variable declaration. It determines the **state** and **behavior** that can be accessed using the reference.

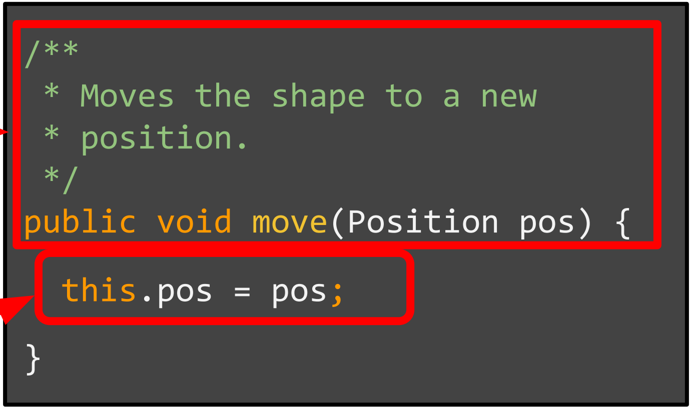
If the `Shape` class does not **define** methods for area, and perimeter, then those methods can't be called on a `Shape` reference.

At the same time, it doesn't make sense to **implement** those methods on the `Shape` class, because the implementations are not useful at all.

Furthermore, it doesn't make sense to instantiate a "`Shape`." What does that even mean? What shape is `Shape`?

Definition vs. Implementation

- A method **definition** includes the following:
  - A name.
  - A return type.
  - A parameter list.
  - A description of the behavior of the method (for humans).
- A method **implementation** is the code inside the body of the method that implements the described behavior.
- Sometimes, we would like to **define** a method without **implementing** it.
  - That is to say write a method declaration, but don't include a pointless body.
- We can do this using the `abstract` modifier.



```java
/**
 * Moves the shape to a new
 * position.
 */
public void move(Position pos) {
  this.pos = pos;
}
```

For example, we want all `Shapes` to **define** `area` and `perimeter` methods so that we can use them in classes like `ShapeMover`...

...but it doesn't make any sense to try and **implement** those methods in the Shape class.

Abstract Methods and Classes

- An `abstract` method is one that includes a **definition** but no **implementation**.
  - It includes a signature but no body.
  - A semicolon (`;`) is used instead of the curly braces (`{}`).
- A method without a body must be declared `abstract`.
- A class may also be declared `abstract`.
  - An `abstract` class may include ***zero or more*** abstract methods.
  - An `abstract` class ***cannot*** be instantiated. Why?
- A class that is not `abstract` cannot contain abstract methods. Why?

An `abstract` class must include the `abstract` modifier in its declaration...

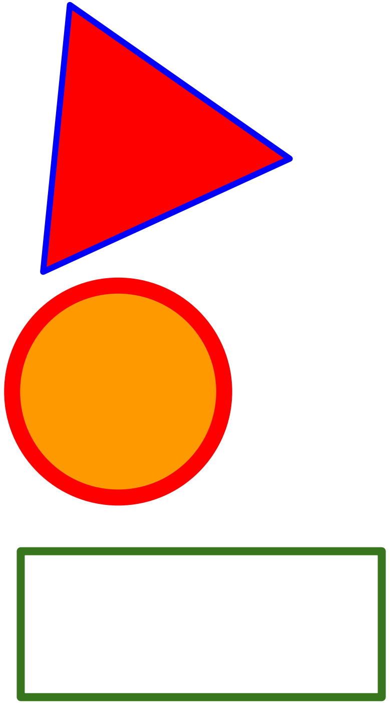
```
public abstract class Parent {
  public String toString() {
    return “Parent”;
  }

  public abstract int age();
}
```

An `abstract` method must also include the `abstract` modifier in its declaration...

An `abstract` class may mix zero or more `abstract` and concrete methods together.

Activity: Abstract Methods



1. In the last activity, you commented out the `area()` and `perimeter()` methods in the `Shape` class. Uncomment them.
2. Modify both methods so that they are `abstract`.
   a. Add the `abstract` modifier.
   b. Remove the method bodies (replace them with a semicolon (`;`)).
3. A normal class cannot contain `abstract` methods, so you will now need to modify the `Shape` class so that it is `abstract`.
4. There is still a compilation problem in your code.
   a. Why?
   b. What is the solution to this problem?

Abstract vs. Concrete

- A ***concrete*** class is one that is not declared to be `abstract` and does not contain any `abstract` methods.
  - Remember, an `abstract` class does not need to contain any `abstract` methods!

	Concrete	Abstract
Concrete Methods	✓	✓
Abstract Methods	✗	✓
Can be Instantiated	✓	✗
Fields	✓	✓
Constructors	✓	✓
Can be Extended	✓	✓
Must Implement Inherited `abstract` Methods	✓	✗

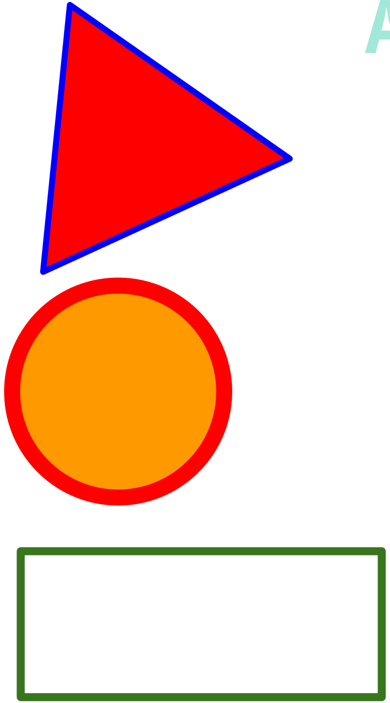
This is the current source of the compilation problem in the `ShapeMover` class...

Because it is now `abstract`, the `Shape` class can no longer be instantiated.

And that's OK! It doesn't make sense to instantiate a generic `Shape` anyway.

A concrete class ***must*** implement any inherited `abstract` methods!

Activity: Fix ShapeMover



1. Fix the `ShapeMover` class so that it no longer tries to instantiate the Shape class.
   a. It doesn't make sense to instantiate the class anyway. What shape is a Shape?

Why Can’t an Abstract Class be Instantiated?

- As we know by now, a class that is declared to be `abstract` cannot be instantiated. But why?
- An abstract class may contain ***zero or more*** `abstract` methods.
- What happens if such a class is instantiated, and you try to call one of those methods?
  - This is exactly what the `ShapeMover` class was doing! It tried to call the `abstract` `area()` and `perimeter()` methods on a Shape that was passed in as an argument!
- There is no implementation code in an `abstract` method. So what does Java do when the method is called?
  - What value is returned from the `area()` or `perimeter()` method if there is no implementation in the body?

An `abstract` method does not contain a body, and so it cannot be invoked.

Any `abstract` class may contain `abstract` methods. Even if it does not, an `abstract` method may be added in the future.

Because of this, `abstract` classes cannot be instantiated, because if they were, trying to invoke `abstract` methods would break the code!

Interfaces

- What if you wanted to create a class that included ***only*** abstract methods?
    - No concrete methods.
    - No state.
- You could simply declare the class abstract and add all of the abstract methods.
- Or, you could write an interface. An interface defines ***only*** abstract methods and may not include any fields.
    - Well, static fields are OK.
- Interfaces are useful if you can **define** behavior but not **implement** it.
- An interface is declared using the interface keyword rather than class.

An interface is declared using the interface keyword (rather than class).

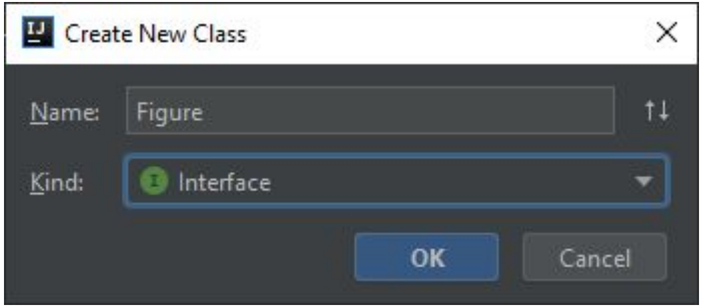
```
public interface Person {
 public abstract String getName();
 int age();
}
```

An interface may contain any number of methods that are both public and abstract.

In fact, ***all*** of the non-static methods in an interface ***must*** be both public and abstract, and so those modifiers can be omitted.

Activity: A Figure Interface

1. Add a new interface to your project: `Figure`.



Create New Class

Name: Figure

Kind: Interface

2. Add the following methods to the interface:
   a. `Position getPos()`
   b. `move(Position)`
   c. `String getFillColor()`
   d. `String getLineColor()`
   e. `double area()`
   f. `double perimeter()`

Implementing an Interface

- A Java class `implements` an `interface` using the `implements` keyword rather than `extends`.
  - In the event that a class ***both*** `extends` another class ***and*** `implements` an `interface`, the `interface` should come ***last***.
- A **concrete** class that `implements` an interface ***must*** provide an implementation of ***all*** of the methods in the `interface`.
  - An **abstract** class does not need to.
- In Java, a class may only `extend` one other class, but may implement an unlimited number of interfaces (separated by commas).

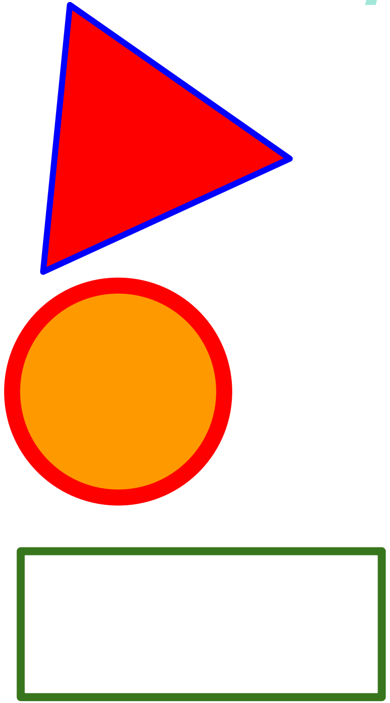
A class may extend up to ***one*** other class, but implement ***many*** interfaces.

```java
public class Parent extends Mammal
  implements Person, Animal {
  // body of class
}
```

The interface names are separated by commas.

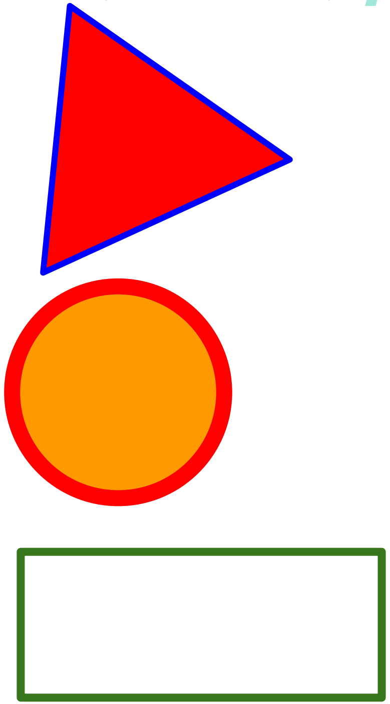
If the class is not abstract, it ***must*** implement every method in any interface that it implements.

Activity: Implementing Figure



1. Update your `Shape` class to implement `Figure`.
2. The `area()` and `perimeter()` methods are now ***redundant***.
   a. They are defined in ***both*** the `Figure` interface ***and*** the `Shape` class.
   b. There is no need to define them in both places. Why?
   c. Delete them from the `Shape` class. What effect did this have on your code?

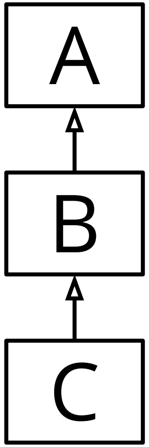
Activity: Using Figure



An interface can be used like any other type. That means it can be used as a parameter or field.

1. Update your `ShapeMover` class to use the `Figure` interface instead of the `Shape` class.

Inheritance is Transitive



Inheritance is ***transitive***. This means that if class B `extends` A, and C `extends` B, then C inherits all of the accessible state and behavior from ***both*** A and B.

This also applies to interfaces.

```
public abstract class Parent
  extends Mammal
  implements Person, Animal {
  // implementations optional
}
```

If a class `extends` an `abstract` class ***or*** `implements` an `interface` (or both), it ***must*** implement the `abstract` methods defined by its parent ***unless*** it is also declared to be `abstract`.

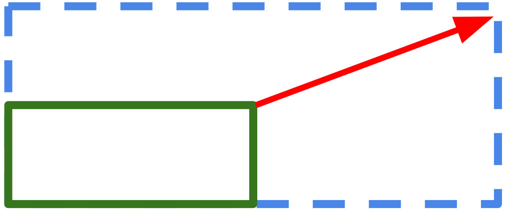
An `abstract` class may implement any number of the `abstract` methods from its parent classes or interfaces (including ***zero*** of them).

```
public class Child
  extends Parent {
  // implementations required
}
```

However, because of the transitive nature of inheritance, this means that the responsibility for implementing those methods will be passed on to any child classes.

Unless of course the child class ***also*** is declared to be `abstract`.

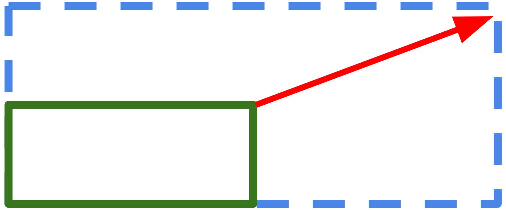
Activity: Scaling I



Add a new method to scale the size of a `Figure`.

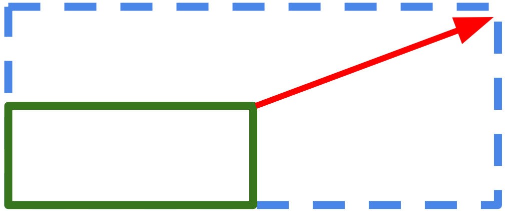
1. Begin by adding a `scale(double factor)` method to the Figure interface.
2. What happens to your `Rectangle`, `Circle`, and `Triangle` classes? Why?

Activity: Scaling II



1. In your `ShapeMover` class, write a new method to scale a figure.
    a. It should take a `Figure` and a `factor` as parameters.
    b. Print the `Figure` before and after it is scaled.
2. Update your `main` method to scale each of your figures after moving it.

Activity: Scaling III



Your code still doesn’t compile!

1. You will need to implement the scale method in each of your figures.
   a. `Rectangle`
   b. `Circle`
   c. `Triangle`
2. Once you have done so, run the `main` method in your `ShapeMover` class.

Concrete vs. Abstract vs. Interfaces

	Concrete	Abstract	Interfaces
Concrete Methods	✓	✓	✗
Abstract Methods	✗	✓	✓
Can be Instantiated	✓	✗	✗
Fields	✓	✓	✗
Constructors	✓	✓	✗
Can be Extended	✓	✓	✓
Must Implement Inherited `abstract` Methods	✓	✗	✗
Multiple Inheritance	✗	✗	✓