SWEN-601
Software Construction

Inheritance & Polymorphism

Today's Quiz Password: OneObject

Activity: Accept the GitHub Classroom Assignment

You will be asked to accept a new assignment at the start of nearly every class.

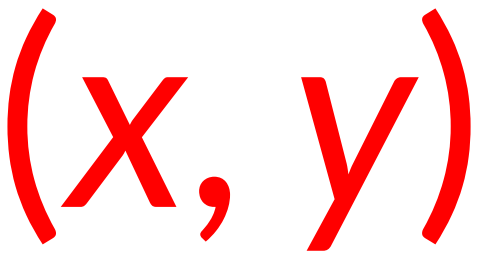
You should get used to accepting the assignment and starting your new project right after you finish your quiz each day.

Your instructor has provided a GitHub classroom invitation. You should be able to find it under "Homework" on MyCourses.

1. Click the GitHub classroom invitation.
2. Assuming that you have already linked your GitHub account with your name in the class roster, you should be prompted to accept the assignment. Do so.
3. Once the repository is created, copy the URL.
4. Clone the repository to your local file system. As before, the repository will be empty.
5. Create a new IntelliJ Project inside the repository, and push it to GitHub.
6. Create a package named "`activities`" in your `src` folder.
7. You are now ready to begin today's activities!

Object Oriented Programming

- **Classes** - Templates that define the state and behavior of a class of thing.
- **Objects** - Instances of a class. Each gets its own copy of the state and behavior.
- **Fields** - State (variables) that belongs to an object.
- **Methods** - Behavior (functions) that belongs to an object.
- **Encapsulation** - An object keeps related state and behavior together in one package and uses data privacy to protect and control access to it.
- **Constructors** - Special methods that create and return a new object.
- **Identity** - Every object has a unique identity (i.e. its address). Shallow equality (==) only considers identity.
- **Equality** - The concept that two distinct objects may be equal. Deep equality (`equals(Object)`) considers the state of two different objects.



Activity: A Position Class

Create a new class to represent an x,y coordinate.

- Name the class `Position`.
- Include accessors and mutators for both x and y values.
- Include `toString` and `equals` methods.

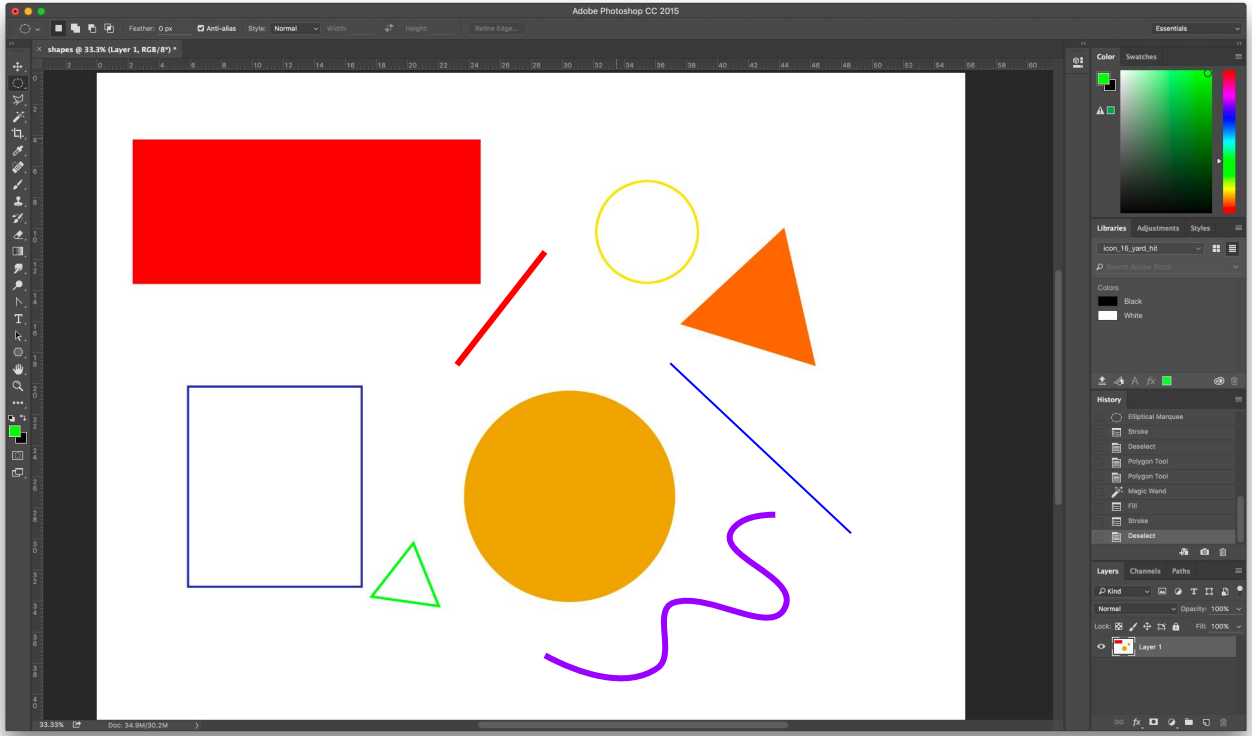
A Simple Drawing Program

Many of you have probably used a drawing program before.

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

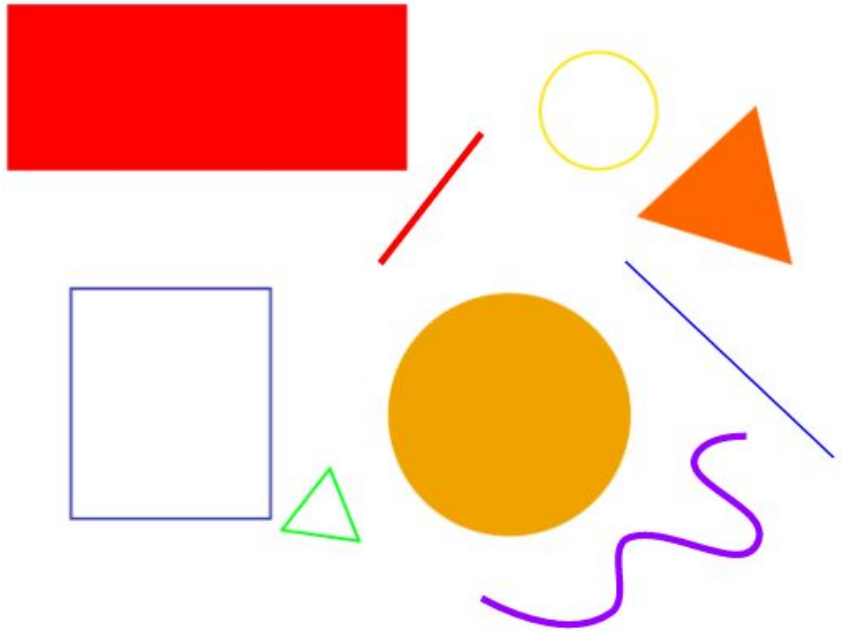
Assume that we are writing a simple drawing program that allows users to draw simple shapes and lines.

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



Shapes

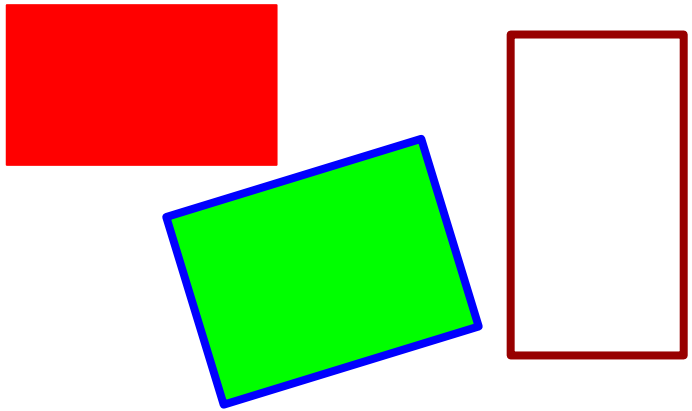
- Let's assume that your program should support a few basic shapes.
  - Rectangles
  - Circles
  - Triangles
  - Straight Lines
  - "Free Hand" Lines.
- Let's brainstorm the state and behavior that some of these shapes might have.
  - Remember that you'll need to consider things like position, size, color, etc.



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

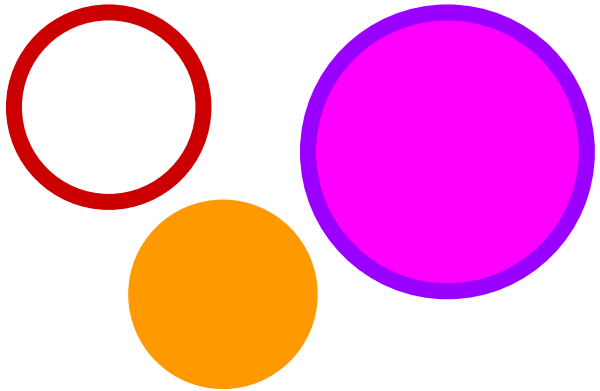
WIDTH
HEIGHT
FILL COLOR
POSITION (X,Y)
OUTLINE COLOR
ORIENTATION

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.

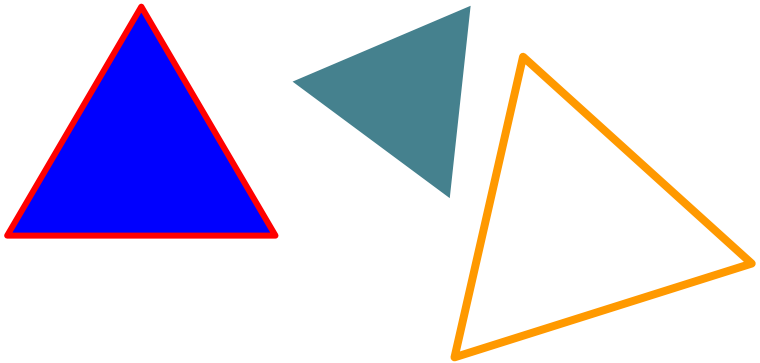


Circle
RADIUS FILL COLOR OUTLINE COLOR POSITION (X,Y)
GETAREA() GETDIAMETER() GETPERIMETER() MOVE() DRAW()

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()
move()
rotate()

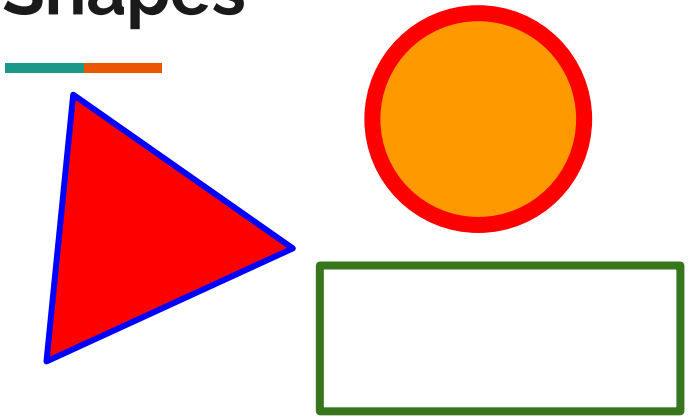
Shapes

You have probably already noticed this, but it's more obvious when all three shapes are side-by-side...

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

Rectangle	Circle	Triangle
double width, height int x // top left corner int y String fillColor String outlineColor double orientation	double radius int x // center int y String fillColor String outlineColor	double sideLength int x // corner int y 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)

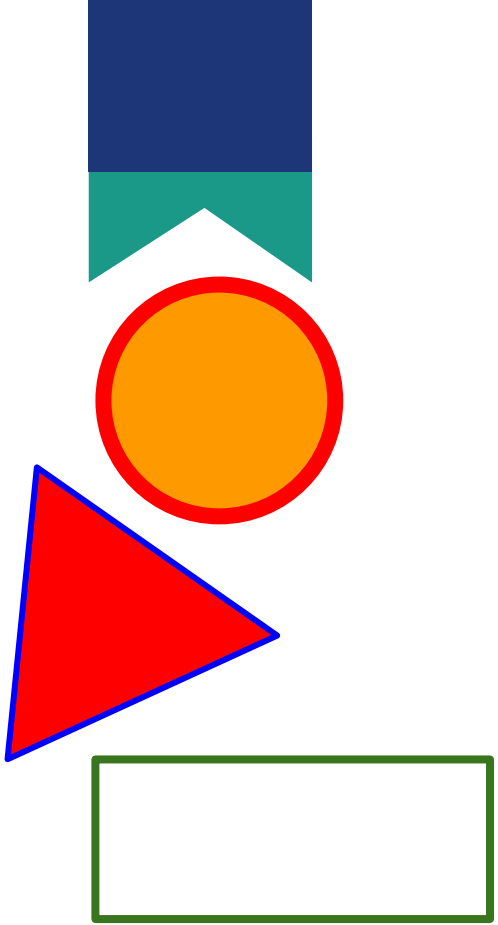
OUTLINE COLOR

MOVE()

GETAREA()

DRAW()

GETPERIMETER()



Activity: A Shape Class

Create a class for a generic *shape*.

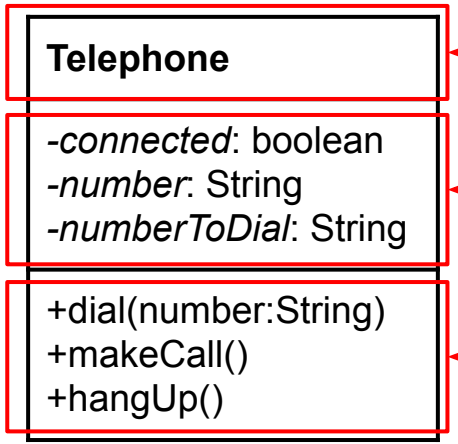
- Make sure to include the *state* that all of our shapes have in common:
  - A position
  - Fill Color (String)
  - Line Color (String)
- An *initializing constructor*.
- And include all of the *behavior* that all of our shapes have in common:
  - Area - `return 0` for now
  - Perimeter - `return 0` for now
  - Move - change X,Y position

A Little UML

Classes can be drawn using a special *modeling language* called the *Unified Modeling Language* (UML).

It can be *very* useful to sketch out a UML class diagram before you begin coding.

Each UML class diagram has a few different parts...



The class **name**.

The **state** defined by the class (fields).

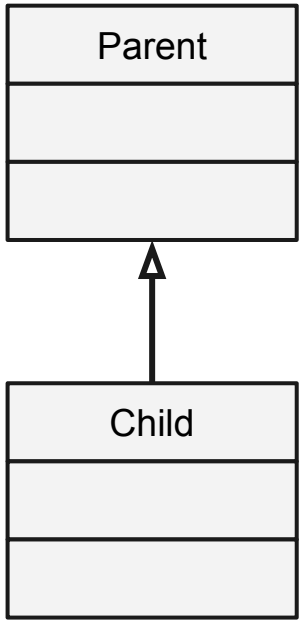
The minus (-) indicates that the fields are `private`.

The **behavior** defined by the class (methods).

The plus (+) indicates that the methods are `public`.

Reuse

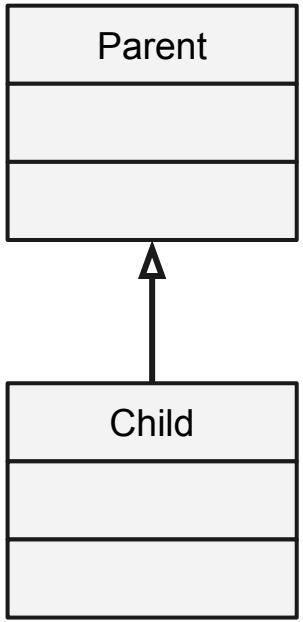
- One of the major benefits of *object oriented programming* is ***reuse***.
  - Write code once, and use it in many places.
- Reuse is an alternative to “copy and paste” coding, which duplicates the same code wherever it is needed.
  - There are many drawbacks to duplicating code, not the least of which is duplicating bugs and increasing maintenance costs.
- We have already seen how this is possible by encapsulating state and behavior inside of a class.
  - Rather than cutting and pasting the code, the class is instantiated wherever the state and behavior is needed.
- But what if multiple classes need to share similar state and behavior?
- Object oriented programming provides a core feature to handle this: ***inheritance***.



Inheritance is shown in UML with an arrow pointing from the ***child*** to the ***parent***.

“Is A” Relationships - Inheritance

- A ***child class*** inherits the *accessible* state and behavior of its ***parent class***.
  - `private` members are not accessible to the child class.
  - Child classes *can* access `protected` members in the parent.
  - Constructors are ***not*** inherited. The child class must define its own.
- But what does “*inherit*” mean?
- Inheritance establishes an “**is a**” relationship between two classes.
  - The child ***is an*** instance of the parent.
- This means that an instance of the child class can be treated as though it is an instance of the parent class.
  - This also means that an instance of the *child* can be used anywhere a *parent* is needed/expected. This is **extremely** powerful and important.



Inheritance

All `Animals` are multicellular organisms.

All `Chordates` are `Animals` with spinal cords.

All `Mammals` are `Chordates` with hair and warm blood.

The *children* have all of the *general* characteristics of the *parent* but add something more *specialized*.

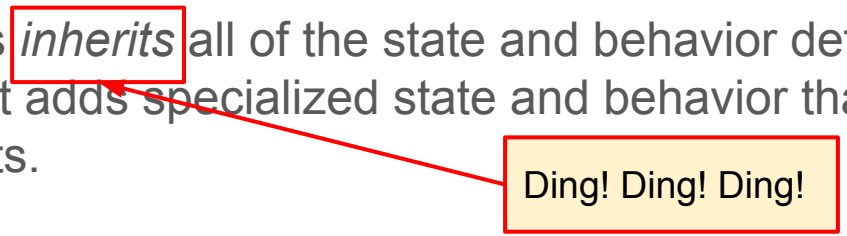
All `MotorVehicles` use fuel.

All `WheeledVehicles` are `Vehicles` with wheels.

All `Cars` are `WheeledVehicles` with doors and windows.

Inheritance

- Inheritance allows a software developer to derive a *specialized* class from a more *generalized* class that already exists.
- The existing class is the *generalization* and is called the *parent class*, *superclass*, or *base class*
  - The superclass defines state and behavior that is common to all objects of that type. **Not any** state and behavior that *some* objects have but *others* do not.
- The derived class is the *specialization* and is called the *child class* or *subclass*.
  - The child class *inherits* all of the state and behavior defined by the superclass, but adds specialized state and behavior that is not shared by other objects.



Inheritance

- A programmer can create a child class by:
  - Adding new state
  - Adding new behaviors
  - Modifying existing behaviors (**overriding**)
- Software reuse is a fundamental benefit of inheritance
  - Common state and behavior is defined in a parent class.
  - Child classes inherit this behavior. It doesn't need to be copied and pasted to multiple classes.
- By using existing software components to create new ones, we capitalize on all of the effort that went into the design, implementation, and testing of the existing software.
  - Most importantly, because the code is in one place (the parent) changes, fixes, improvements only need to be made in one place.

We have done this already with the `toString` and `equals` methods. Now you will override some of your own methods.

Subclassing

- ***Subclassing*** refers to creating a child class that inherits the members of a parent class.
  - In Java this is done using the `extends` keyword.

```
public class Mammal extends Chordate {
  // ...
}
```

- The new class is referred to as the ***child class*** or the ***subclass***.
  - It inherits the members (state and behavior) of the parent class, and therefore does not need to reimplement it.
- Constructors are different.
  - Constructors are ***not*** inherited. The child class must define its own.
  - If the parent class defines one or more constructors, the child **must** call one of them from its own constructor using the `extends` keyword.

Constructor Sequence

- If the parent class defines at least one constructor with parameters, the child class *must* invoke one of the constructors.
  - This is done using the `super` keyword.

```
public class Parent {
  private int x;
  public Parent(int x) {
    this.x = x;
  }
}
```

Given this `Parent` class definition, which declares a constructor...

```
public class Child extends Parent {
  private String name;
  public Child(int x, String name) {
    super(x);
    this.name = name;
  }
}
```

The `Child` class *must* invoke the parent constructor using super, *and* it *must* be the *first line* of the Child's constructor.

Constructor Sequence

- If the parent class does not define a constructor, or defines a *parameterless* constructor, that constructor is invoked transparently if the child doesn't call another constructor with `super`.

```
public class Parent {
  private int x;
  public Parent() {
    this.x = 10;
  }
}
```

```
public class Child extends Parent {
  private String name;
  public Child(String name) {
    this.name = name;
  }
}
```

Given this `Parent` class definition, which declares a parameterless constructor...

If the `Child` class doesn't explicitly invoke some other constructor using `super`, the parameterless constructor will be invoked automatically.

Constructor Sequence

```java
public class Parent {
  private int x;
  public Parent() {
    x = 10;
    System.out.println("Parent()");
  }

  public Parent(int x) {
    this.x = x;
    System.out.println("Parent(" +
      x + ")");
  }
}
```

```java
public class Child extends Parent{
  private String name;
  public Child(String name) {
    this.name = name;
    System.out.println("Child(" +
      name + ")");
  }

  public Child(int x, String name) {
    super(x);
    this.name = name;
    System.out.println("Child(" +
      x + "," + name + ")");
  }

}
```

Q: What is the output if the `Child(25,"Kimmi")` constructor is called?

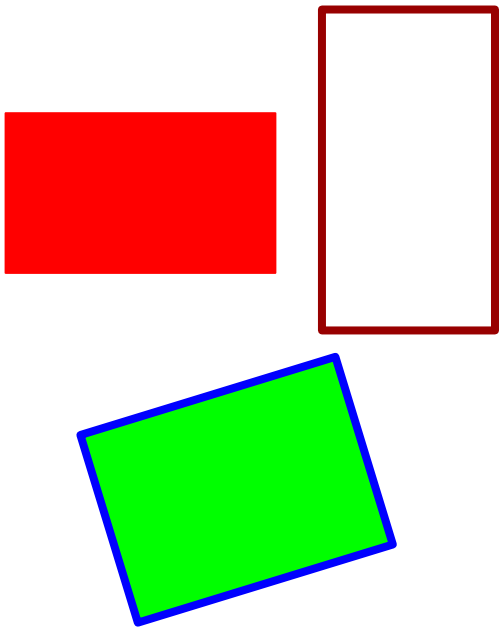
Access Modifiers (Refresher)

Visibility	`public`	`protected`	*package private*	`private`
Objects of the Same Class	✔	✔	✔	✔
Other Classes in the Same Package	✔	✔	✔	
Child Classes*	✔	✔		
The Entire Program	✔			

super

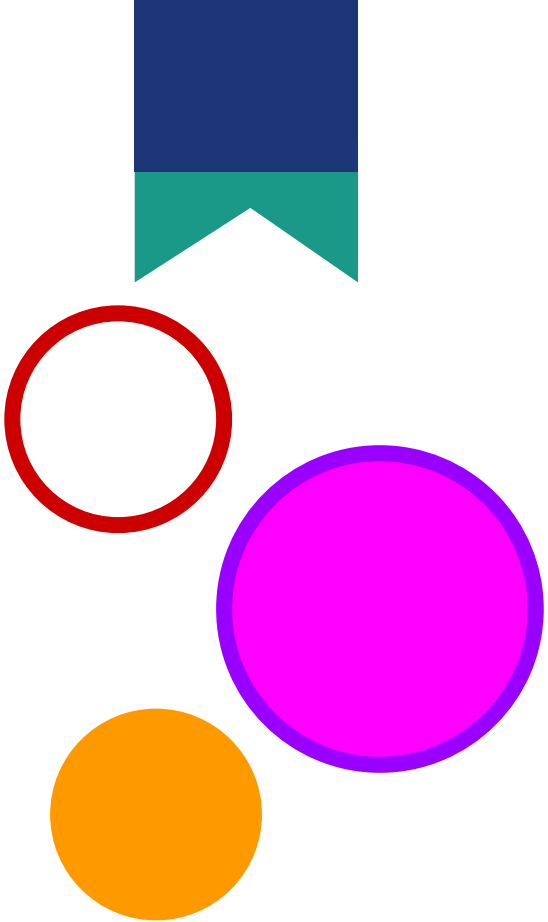
When used with a	super has this effect
constructor	A constructor in a child class may invoke a constructor on its parent class using `super(arguments)`. If the superclass does not define a parameterless constructor, the child class *must* call one of the defined constructors in this way. This must be the *first* statement in the child constructor. If the superclass *does* define a parameterless constructor it will be transparently invoked by the child constructor if `super()` is not used.
field	Disambiguates between a local variable and a field in the superclass *or* a field in the child class with the same name in the event that the child class has hidden a field in the superclass. For example, if the superclass defines the field: `protected int x = 4;` and the child class has defined a field `private double x = 2.5;` then `super.x` can be used to refer to the `int` in the superclass. Hiding fields is considered bad practice and should be avoided.
method	A child class may *override* a method in the parent class by defining a method with the same signature (including name, parameters, and return type) as a method in the parent class. `super` may be used from the child class to call the superclass's version of the method.

Activity: A Rectangle Class



Create a class for *rectangles*.

- A rectangle *is a* shape, so start by subclassing the `Shape` class that we already made.
  - Use `extends`.
- Make sure to include the state that is unique to rectangles:
  - Width
  - Height
- An initializing constructor.
  - You will need to use `super`!
- And the following behavior:
  - Accessors and mutators for width and height
  - Area - ***override*** the parent implementation
  - Perimeter - ***override*** the parent implementation
  - `toString` and `equals` methods.



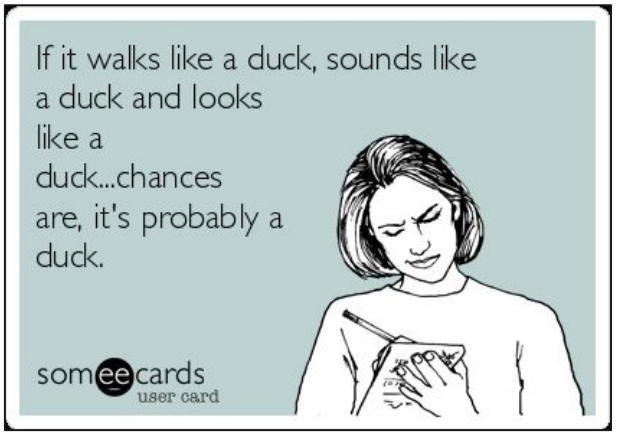
Activity: A Circle Class

Create a class for *circles*.

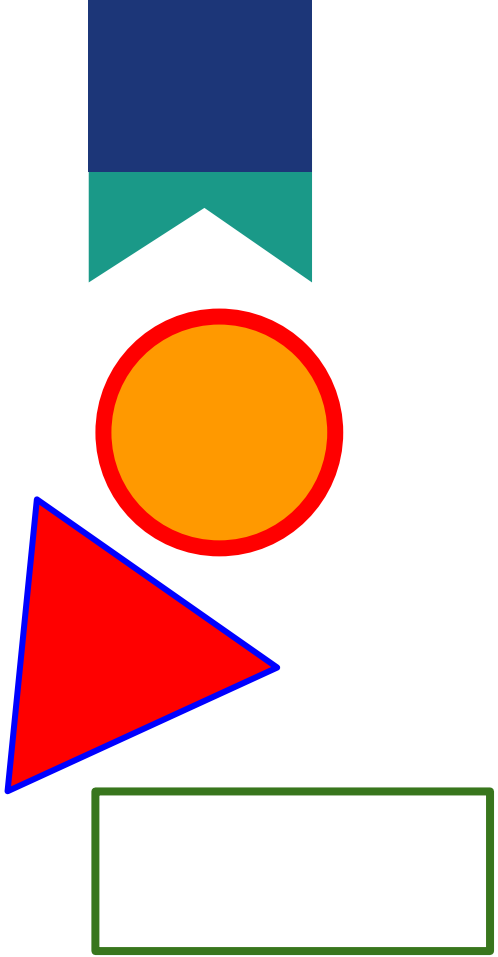
- A circle *is a* shape, so start by subclassing the `Shape` class that we already made.
  - Use `extends`.
- Make sure to include the state that is unique to rectangles:
  - Radius
- An initializing constructor.
  - You will need to use `super`!
- And the following behavior:
  - An accessor and mutator for radius
  - Area - ***override*** the parent implementation
  - Perimeter - ***override*** the parent implementation
  - `toString` and `equals` methods.

Polymorphism: Part 1

- The Shape class defines specific state and behavior.
  - A location (x, y).
  - A method to change the location (`move(x, y)`).
- Any code that is written using a Shape reference *only* has access to the state and behavior defined by the Shape class.
  - This means that the code can *only* use location and the move method.
- And remember, the child classes (`Rectangle`, `Circle`) *inherit* all of the state and behavior defined by Shape.
  - This means that any code that uses the state and behavior defined by Shape will find that state and behavior in the child classes as well.
- ***Polymorphism*** means that a child class can be used as though it is an instance of its parent class.



A class that `extends` Shape ***is a*** Shape, and can be used anywhere a Shape is expected.



Activity: ShapeMover

Create a new class called `ShapeMover`.

- Write a static method that, given any `Shape` and `Position`, moves the shape to the new position.
  - Print the `Shape` before and after it is moved.
- Add a `main` to test your method with several different kinds of shapes (e.g. shapes, rectangles, circles).

Reference Type vs. Actual Type

```
Shape shape = new Circle();
```

The **reference** type is the type used in the variable declaration and/or on the left side of an assignment statement.

The **actual** type is the type created on the right side of the assignment statement.

Remember: **Only** the state and behavior defined by this type may be used.

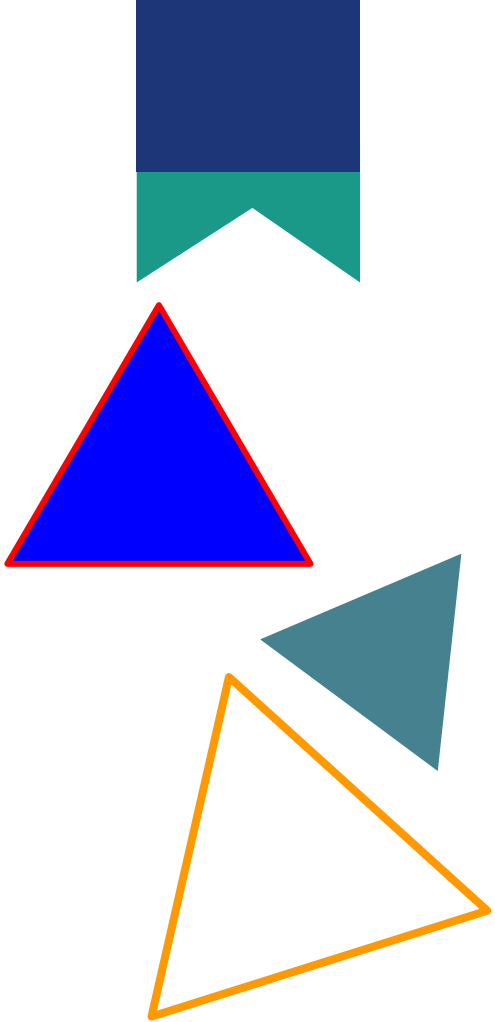
The **actual** type is used to determine which method is invoked at runtime *and may be changed* with another assignment statement.

Polymorphism: Part 2

- There are two parts to polymorphism.
- The first is that a child class can be used anywhere that it's parent class is expected.
  - e.g. as a parameter to a method.
- The second is that what ***appears*** to be one method at compile time may be a different method at runtime.
- Consider the following code:

```
public static void printArea(Shape shape) {
  System.out.println(shape.getArea());
}
```

- Can you say for sure ***which*** area method will be called when this code is executed?
  - No! It depends on what kind of shape is passed into the method!

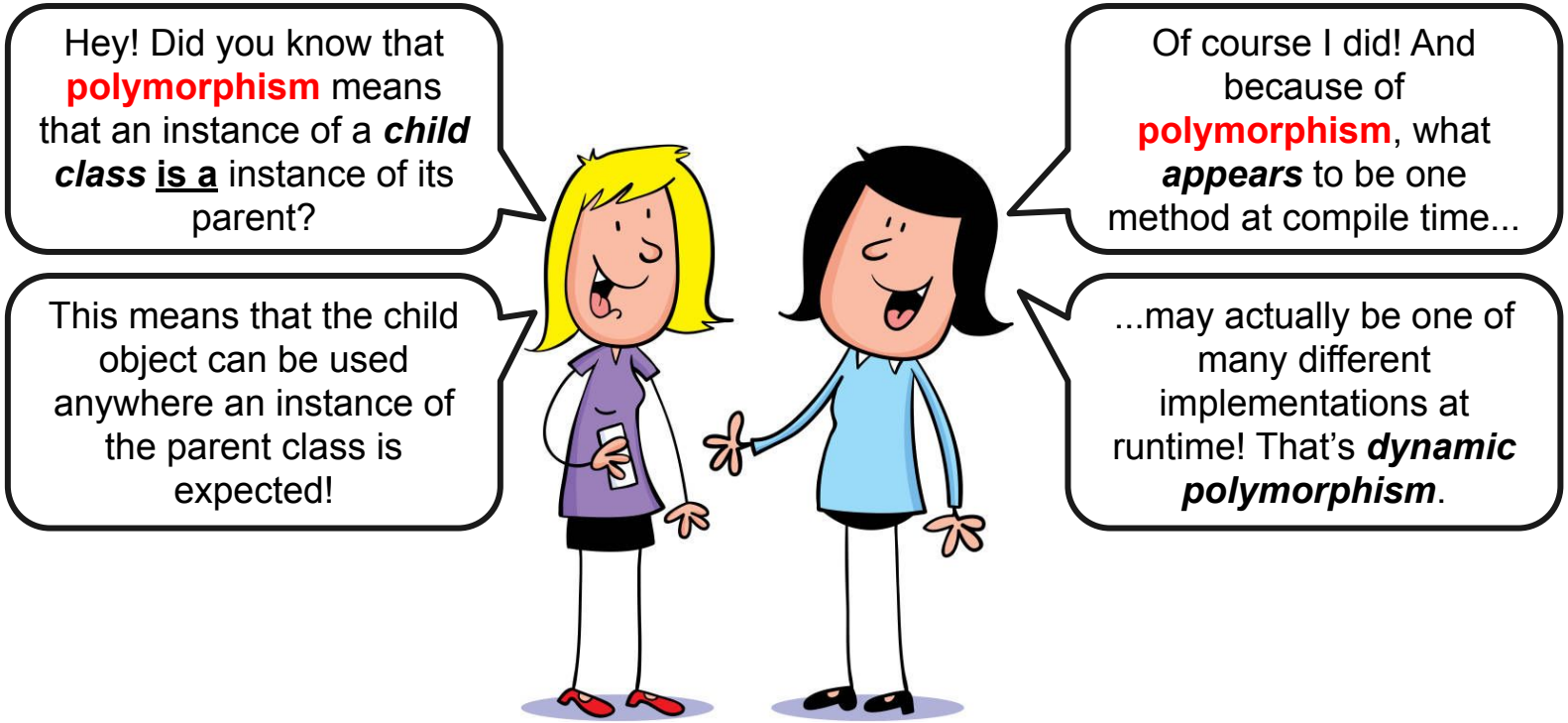


Activity: A Triangle Class

Create a class for *equilateral triangles*.

- A triangle *is a* shape, so start by subclassing the `Shape` class that we already made.
  - Use `extends`.
- Be sure to include state that is unique to a triangle:
  - Side Length
- An initializing constructor.
  - You will need to use `super`!
- And the following behavior:
  - An accessor and mutator for side length
  - Area - ***override*** the parent implementation
  - Perimeter - ***override*** the parent implementation
  - `toString` and `equals` methods.
- Update your `ShapeMover` to try it out with a triangle or two!

Polymorphism



java.lang.Object

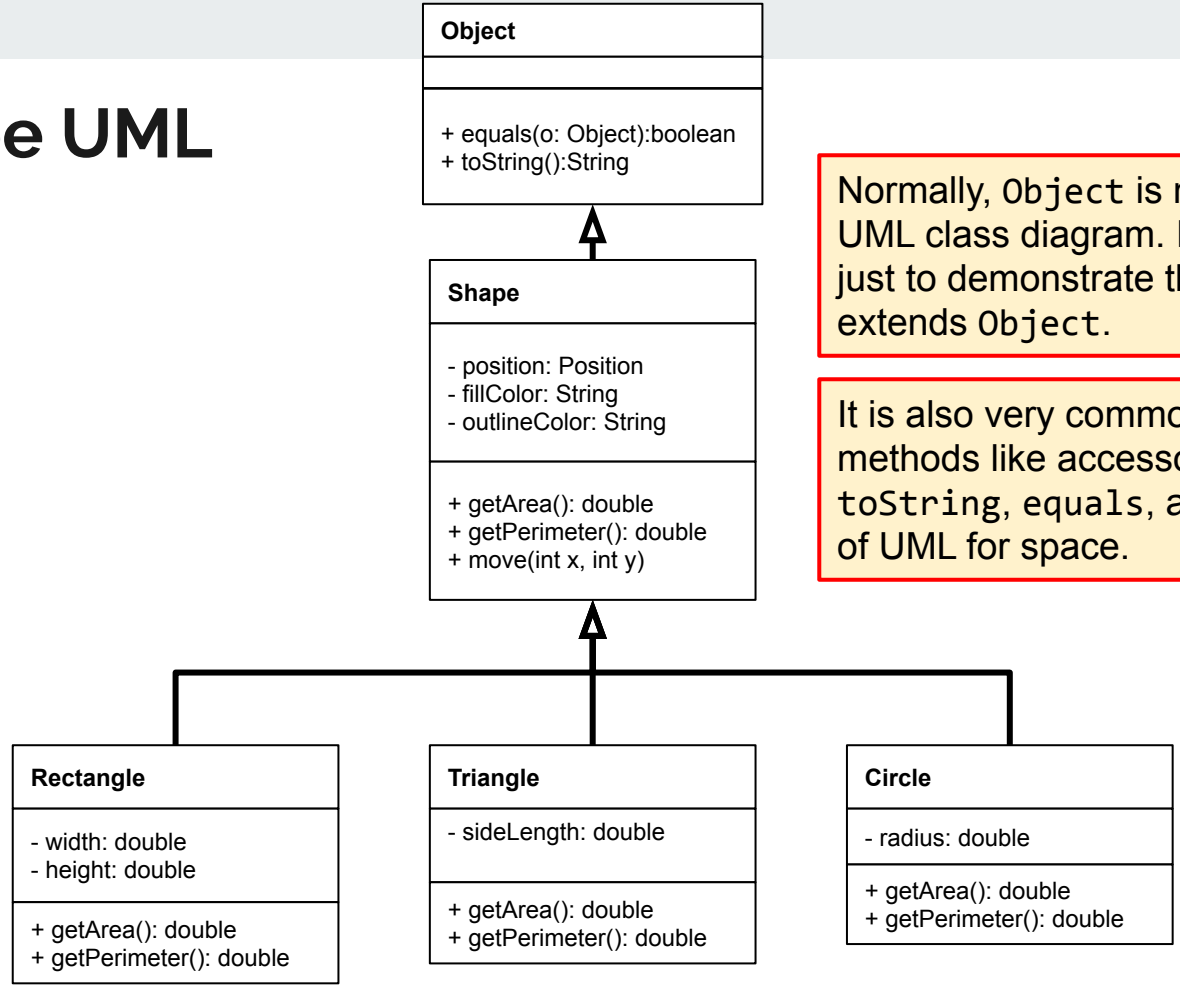
- The `java.lang.Object` class is the parent of all other classes in Java.
  - If a class does not explicitly extend some other class, it implicitly extends the `java.lang.Object` class.
- The `java.lang.Object` class provides the default implementations of several important methods including:
  - `equals(Object)` - be default compares two objects using shallow equality.
  - `toString()` - by default returns a not-very-useful string.
- `Object` is at the root of the class hierarchy for all objects.

There are many features of the Java language that will work with ***any*** `Object`.

`System.out.println` and `equals(Object)` are just two examples.

And in Java, ***every*** class **is a** `Object`.

Shape UML



Normally, `Object` is not included in every UML class diagram. It is included here just to demonstrate that `Shape` implicitly extends `Object`.

It is also very common to leave common methods like accessors, mutators, `toString`, `equals`, and constructors out of UML for space.