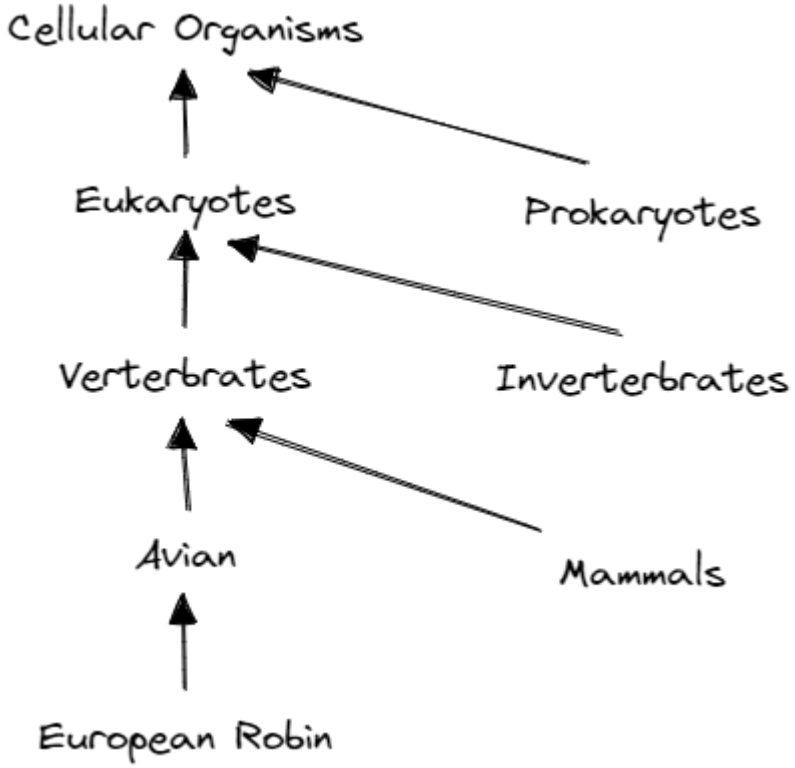
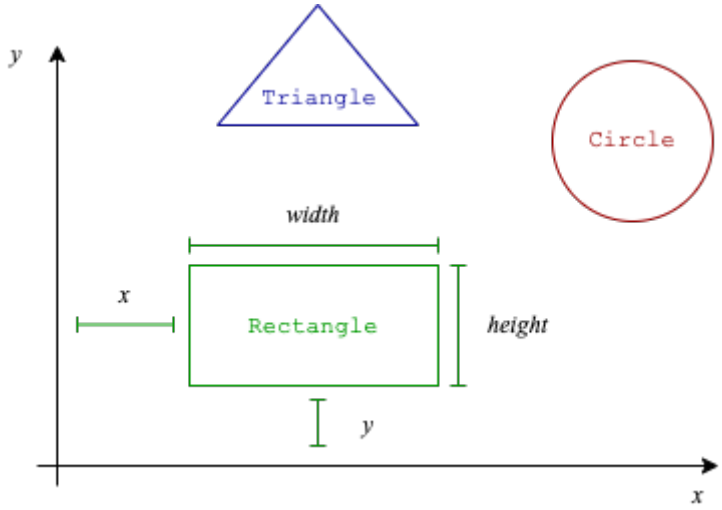
Object Oriented Programming - Day 1

Module: Interfaces



Shapes



Rectangles

```
public class Rectangle {
    private double x;
    private double y;
    private double width;
    private double height;

    public Rectangle(double x, double y, double width, double height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

    public double getX() { return x; }
    public void setX(double x) { this.x = x; }
    public double getY() { return y; }
    public void setY(double y) { this.y = y; }
    public double getWidth() { return width; }
    public void setWidth(double width) { this.width = width; }
    public double getHeight() { return height; }
    public void setHeight(double height) { this.height = height; }
}
```

Circles

```
public class Circle {
    private double x;
    private double y;
    private double radius;

    public Circle(double x, double y, double radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    public double getX() { return x; }
    public void setX(double x) { this.x = x; }
    public double getY() { return y; }
    public void setY(double y) { this.y = y; }
    public double getRadius() { return radius; }
    public void setRadius(double radius) { this.radius = radius; }
}
```

Triangles

```
public class Triangle {
    private double x;
    private double y;
    private double base;
    private double height;

    public Triangle(double x, double y, double base, double height) {
        this.x = x;
        this.y = y;
        this.base = base;
        this.height = height;
    }

    public double getX() { return x; }
    public void setX(double x) { this.x = x; }
    public double getY() { return y; }
    public void setY(double y) { this.y = y; }
    public double getBase() { return base; }
    public void setBase(double base) { this.base = base; }
    public double getHeight() { return height; }
    public void setHeight(double height) { this.height = height; }
}
```

Moving shapes

```
public class Mover {

    public void moveToRight(Rectangle rectangle) {
        double currentX = rectangle.getX();
        double updatedX = currentX + 1;
        rectangle.setX(updatedX);
    }

    public void moveToRight(Triangle triangle) {
        double currentX = triangle.getX();
        double updatedX = currentX + 1;
        triangle.setX(updatedX);
    }

    public void moveToRight(Circle circle) {
        double currentX = circle.getX();
        double updatedX = currentX + 1;
        circle.setX(updatedX);
    }
}
```

Interfaces

```
public interface Positioned {

    public double getX();
    public void setX(double x);
    public double getY();
    public void setY(double y);
}
```

Interfaces as types

```
jshell> Positioned positioned = new Positioned();
|  Error:
|  Positioned is abstract; cannot be instantiated
|  Positioned positioned = new Positioned();
|                          ^--------------^
```

Rectangles as positioned shapes

```
public class Rectangle implements Positioned {

    // Like before...
}
```

Other shapes

```
public class Circle implements Positioned {

    // Like before...
}

public class Triangle implements Positioned {

    // Like before...
}
```

Interfaces and instances

```
jshell> Positioned positioned = new Rectangle(0.0, 0.0, 12.0, 12.0);
positioned ==> Rectangle@7cc355be
```

Moving with interfaces

We can move any shape that has a position (is positioned):

```
public class Mover {

    public void moveToRight(Positioned positioned) {
        double currentX = positioned.getX();
        double updatedX = currentX + 1;
        positioned.setX(updatedX);
    }
}
```

Grouping with interfaces

```
Positioned[] thingsWithPosition = {
    new Circle(4, 1, 8),
    new Triangle(5, 9, 1, 1),
    new Rectangle(0, 0, 12, 2)
};
```

Exercises

- Write a loop that moves each shape in `thingsWithPosition` twenty-four units to the right.
- Create another method in `Mover` that moves shapes an arbitrary number of units to the right. Modify the code in the previous exercise to use that.
- Create a method in `Mover` that moves shapes to the left. Can you make sure that objects never move off screen (negative `x`)?

More interfaces

```
interface Sized {

    public double getWidth();
    public double getHeight();
}
```

Rectangles as shapes with size

```
public class Rectangle implements Positioned, Sized {

    // Like before...
}
```

Triangles as shapes with size

```
public class Triangle implements Positioned, Sized {

    // Like before...
}
```

```
|  Error:
|  Triangle is not abstract and does not override abstract method getWidth() in Sized
|  public class Triangle implements Positioned, Sized {
|  ^-------------------------------------------------...
```

Making it work

```
public class Triangle implements Positioned, Sized {

    // Like before...

    // This method was only added to satisfy `Sized`:
    public double getWidth() {
        return base;
    }
}
```

Now it all works!

@Override

```
public class Triangle implements Positioned, Sized {

    // Like before...

    @Override
    public double getWidth() {
        return base;
    }
}
```

Exercises

- Can we make `setWidth` and `setHeight` part of the interface `Sized`. Why or why not?
- Suppose we make a typing error. Let's say we type `getWitdh()` instead of `getWidth()` (which happens to me all the time!). Compare the error message you get with and without the `@Override` on top of your method. Which error message do you prefer?
- Make `Circle` implement `Sized` too. The width and height of a circle are twice the radius (also called the diameter).
- Introduce another interface called `HasArea` with a single method `double getArea()`. Make each shape calculate the appropriate area.
- Can we calculate circumferences too?

Module: Inheritance

Interfaces recap

We learned about interfaces about contracts that

- force classes like `Rectangle` to provide a set of methods
- allow classes like `Mover` to accept any class that implements that interface (follows the contract)

Remember `Rectangle`?

```
public class Rectangle implements Positioned, Sized {
    private double x;
    private double y;
    private double width;
    private double height;

    public Rectangle(double x, double y, double width, double height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

    @Override public double getX() { return x; }
    @Override public void setX(double x) { this.x = x; }
    @Override public double getY() { return y; }
    @Override public void setY(double y) { this.y = y; }
    @Override public double getWidth() { return width; }
    @Override public double getHeight() { return height; }
}
```

Remember `Circle`?

```
public class Circle implements Positioned, Sized {
    private double x;
    private double y;
    private double radius;

    public Circle(double x, double y, double radius) {
        this.x = x;
        this.y = y
        this.radius = radius;
    }

    @Override public double getX() { return x; }
    @Override public void setX(double x) { this.x = x; }
    @Override public double getY() { return y; }
    @Override public void setY(double y) { this.y = y; }
    public double getRadius() { return radius; }
    public void setRadius(double radius) { this.radius = radius; }

    private double getDiameter() {
        return this.radius * 2;
    }

    @Override public double getWidth() { return getDiameter(); }
    @Override public double getHeight() { return getDiameter(); }

}
```

Remember `Triangle`?

```
public class Triangle implements Positioned, Sized {
    private double x;
    private double y;
    private double base;
    private double height;

    public Triangle(double x, double y, double base, double height) {
        this.x = x;
        this.y = y;
        this.base = base;
        this.height = height;
    }

    @Override public double getX() { return x; }
    @Override public void setX(double x) { this.x = x; }
    @Override public double getY() { return y; }
    @Override public void setY(double y) { this.y = y; }
    public double getBase() { return base; }
    public double getHeight() { return height; }

    @Override public double getHeight() { return getBase(); }
}
```

Share behaviour between subclasses

What we can do, is create a `Shape` class that gathers all of the common parts.

```
public class Shape {
    private double x;
    private double y;

    public Shape(double x, double y) {
        this.x = x;
        this.y = y
    }

    public double getX() { return x; }
    public void setX(double x) { this.x = x; }
    public double getY() { return y; }
    public void setY(double y) { this.y = y; }
}
```

`Rectangle` as `Shape`

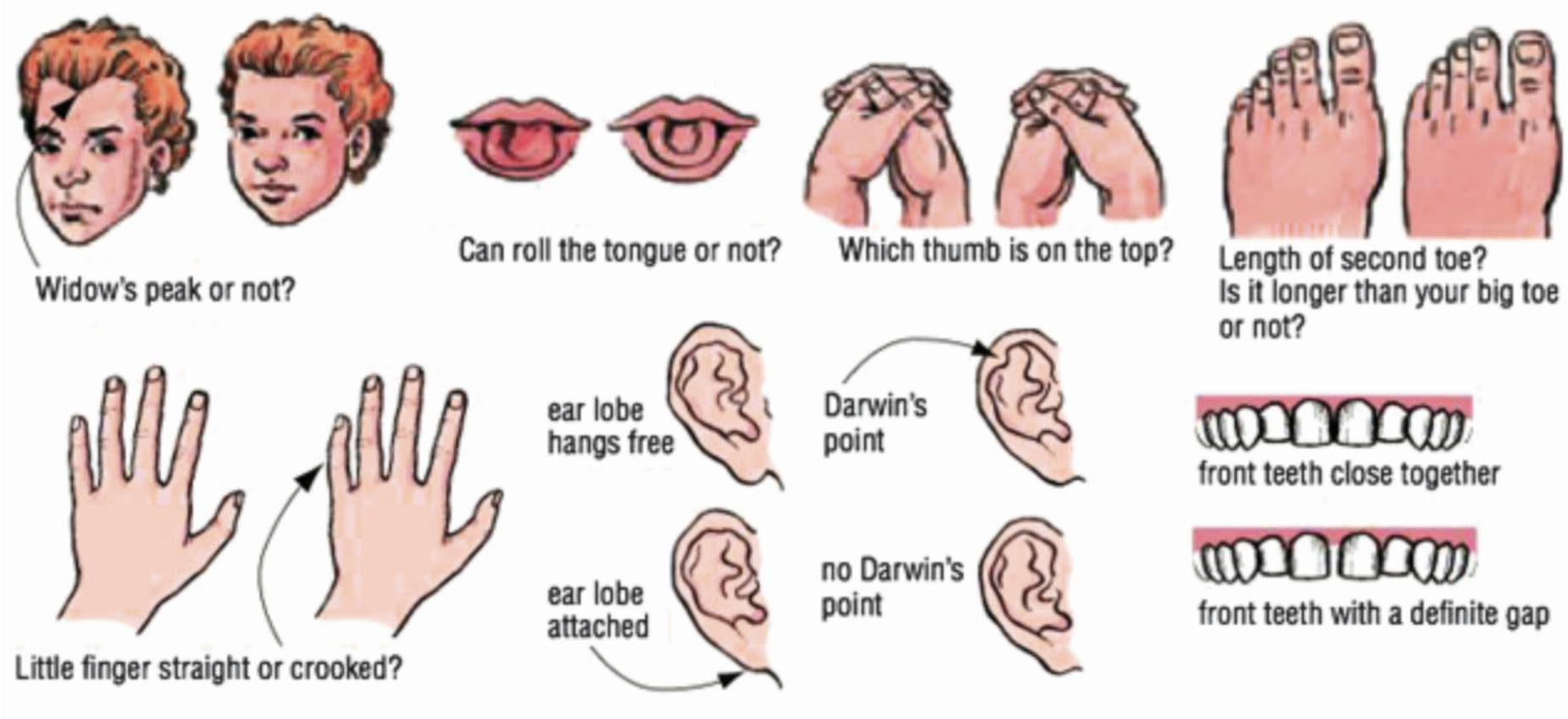
Now, we can use the `extends` keyword to reuse all of that stuff that we just put in the Shape class:

```
public class Rectangle extends Shape implements Positioned, Sized {
    private double width;
    private double height;

    public Rectangle(double x, double y, double width, double height) {
        super(x, y);
        this.width = width;
        this.height = height;
    }

    @Override public double getWidth() { return width; }
    @Override public void setWidth(double width) { this.width = width; }
    @Override public double getHeight() { return height; }
    @Override public void setHeight(double height) { this.height = height; }
}
```

Inheritance

```
Rectangle rectangle = new Rectangle(0.0, 0.0, 12.0, 12.0);
double horizontalPosition = rectangle.getX();
```

Nature


Widow's peak or not?
Can roll the tongue or not?
Which thumb is on the top?
Length of second toe?
Is it longer than your big toe
or not?
Little finger straight or crooked?
ear lobe
hangs free
ear lobe
attached
Darwin's
point
no Darwin's
point
front teeth close together
front teeth with a definite gap

Parent and child

```
class Parent {

    public void rollTongue() {
        this.tongue.loop();
    }

    public void becomeBaldAtTwentyFive() {
        if (age >= 25) {
            this.hair.fallOut();
        }
    }

    public boolean isAthletic() {
        // Not at all!
        return false;
    }
}

class Child extends Parent {

}
```

What can a `Child` do?

```
Child werner = new Child();
werner.rollTongue();
werner.becomaBaldAtTwentyFive();
```

Child-specific behavior

```
class Child extends Parent {

    public void listenToRapMusic() {
        System.out.println("Will the real Slim Shady please stand up?");
    }
}
```

where

```
Child werner = new Child();
werner.rollTongue();
werner.becomaBaldAtTwentyFive();
werner.listenToRapMusic();
```

`Child` is really an *extension* of `Parent`!

@Override

```
class Child extends Parent {

    public void listenToRapMusic() {
        System.out.println("Will the real Slim Shady please stand up?");
    }

    @Override
    public boolean isAthletic() {
        // Very!
        return true;
    }
}
```

Exercises

- Make `Triangle` and `Circle` extend `Shape` too.

Exercises

- Create an interface called `Animal` with a `void`-method `makeSound()`
- Create a class called `LandAnimal` with a `void`-method `walk()` that prints `"Walking…"`
- Create a class `Cow` that extends `LandAnimal` and implements `Animal` by printing a typical cow-sound when `makeSound()` is called. Create an instance of it. What methods does it support?
- Create a class `Fish` that implements `Animal`. Can it extend `LandAnimal`? Should it?

Difference between classes and interfaces

There is a bit of overlap between `Shape` and `Positioned`:

```
public interface Positioned {

    public double getX();
    public void setX(double x);
    public double getY();
    public void setY(double y);
}

class Shape {
    private double x;
    private double y;

    public Shape(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() { return x; }
    public void setX(double x) { this.x = x; }
    public double getY() { return y; }
    public void setY(double y) { this.y = y; }
}
```

Subtyping

```
// A `Rectangle` can do everything
// a `Shape` can do (and more!)
Shape shape = new Rectangle(0.0, 0.0, 12.0, 6.0);
```

Mover

```
public class Mover {

    public void moveToRight(Shape shape) {
        double currentX = shape.getX();
        double updatedX = currentX + 1;
        shape.setX(updatedX);
    }
}
```

Extending and implementing

```
interface Customer {
    void pay(double amount);
}

class Child extends Parent
    implements Customer, Employee, Partner, SoccerCoach {

        @Override
        void pay(double amount) {
            this.creditCard.pay(amount);
        }

}
```

What to use (part I)

Use a class if you can provide an implementation

Use a class if you want to share properties (data)

Otherwise, **use an interface**

(**Use an interface** if you don’t have a choice)

Extending multiple classes

Where did my mother get her ability to roll her tongue?

```
public class GrandParent {

    public void rollTongue() {
        this.tongue.loop();
    }

}

class Parent {

    public void becomeBaldAtTwentyFive() {
        if (age >= 25) {
            this.hair.fallOut();
        }
    }

    public boolean isAthletic() {
        // Not at all!
        return false;
    }
}

class Child extends Parent {

    public void listenToRapMusic() {
        System.out.println("Will the real Slim Shady please stand up?");
    }

    @Override
```

```
    public boolean isAthletic() {
        // Very!
        return true;
    }
}
```

Squares

```
public class Square extends Rectangle {

    public Square(double x, double y, double edge) {
        super(x, y, edge, edge);
    }

    // Be careful of `setWidth` and `setHeight`!
}
```

Immutable squares and rectangles don’t have this problem!

Abstract classes

```
public abstract class Shape {
    private double x;
    private double y;

    public Shape(double x, double y) {
        this.x = x;
        this.y = y
    }

    public double getX() { return x; }
    public void setX(double x) { this.x = x; }
    public double getY() { return y; }
    public void setY(double y) { this.y = y; }

    public abstract double getWidth();
    public abstract double getHeight();
}
```

No instances

```
jshell> Shape shape = new Shape(0.0, 0.0);
|  Error:
|  Shape is abstract; cannot be instantiated
|  Shape shape = new Shape(0.0, 0.0);
|                ^-----------------^
```

Compare this to interfaces

```
jshell> Positioned positioned = new Positioned();
|  Error:
|  Positioned is abstract; cannot be instantiated
|  Positioned positioned = new Positioned();
|                          ^--------------^
```

Why abstract classes

Mix of both interfaces and classes.

Best of both worlds?

- Can’t create an instance of an abstract class
- Only possible to extend *one* base class (abstract or not).

Concrete interfaces

Confession!

```
public interface Positioned {

    double getX();
    void setX(double x);

    double getY();
    void setY(double y);

    default void moveUp(double howMuch) {
        double currentY = getY();
        double updatedY = currentY + howMuch;
        setY(updatedY);
    }

    default void moveRight(double howMuch) {
        double currentX = getX();
        double updatedX = currentX + howMuch;
        setX(updatedX);
    }
}
```

Why `default` methods

Mix of both interfaces and classes.

Best of both worlds?

- Can only implement using other methods (no data)

What to use (part II)

Use an interface if you can provide an implementation using provided methods

Use a class if you want to share properties (data)

Otherwise, **use an interface**

(**Use an interface** if you don't have a choice)

Exercises

Given

- `Shape` from before
- `Rectangle`, `Triangle` and `Circle` from before

```
public class Colored {
    private final String color;

    public Colored(String color) {
        this.color = color;
    }

    public String getColor() {
        return this.color;
    }
```

```
}
```

How can we give each shape a color?