Inheritance and Composition

Last Time: OOP

- Group related data and functions together
- Don't worry about how something happens, just worry about what happens.

One of the manifestations of these ideas is Object Oriented Program.
Create compound structures consisting of multiple *members*.

Data members referred to as *fields*.

Function members referred to as *methods*.

Classes and Objects

Classes form the blueprint for data and functions.

- Name: Car
- Fields:
  - make
  - model
  - color
  - speed
  - fuel
- methods:
  - accelerate(rate)
  - brake(rate)

Objects are concrete realizations of a class.

- Name: car_7
- Fields:
  - make = "Honda"
  - model = "Civic"
  - color = PURPLE
  - speed = 0
  - fuel = 10.0

How to Create Classes + Objects

```
class Spot {
  float x, y, radius;

  void display() {
    ellipse(x, y, radius, radius);
  }

  Spot(){
    x = 50.0;
    y = 50.0;
    radius = 30.0;
  }

  Spot(float x, float y, float r) {
    this.x = x;
    this.y = y;
    this.radius = r;
  }
}
```

```
Spot sp1, sp2;

void setup(){
  sp1 = new Spot();
  sp2 = new Spot(75, 80, 15);
}

void draw(){
  sp1.display();
  sp2.display();
}
```

Code Review

Questions

Is it possible for global variables to be visible to a class without placing them in a class variable?

What order are files run in? Is the setup function called after all global scripts are run?

Remember: there are broadly two types of statements:

- statements that make something available to other code (variable declarations, functions, classes)
- statements that do something (calls to background, variable assignments, etc.)

All statements that "do something" need to be in `setup()` or `draw()`. So the order of statements doesn't matter, since the only statements that do computation are in setup and draw, and we know what order those are going to be called in!

Order of Execution

- Evaluate all the statements that don't do anything (just make variables, functions, and classes available)
- Run `setup()`
- Run `draw()` in a loop

So for a well-formed program, order doesn't matter!

In practice, order seems to be main file first, then alphabetical by name, but there is no guarantee of this.

This is the left-to-right order in the Processing IDE

Will we mostly be implementing in object-oriented styles moving forward?

For projects: absolutely.

For hands-on: mixed.

Do we need to implement OOP in Project 1?

No. You can do so if you want.

Is it only possible to do `$THING` using OOP?

Not at all! Object-oriented programming is just an organizational technique. Everything you do using it can be done without it.

Is there a way to delete objects in Java so that we don't have the cost of calling `new`?

No. In fact, because Java handles this for you, the cost of eventually deleting the object *is* part of calling `new`.

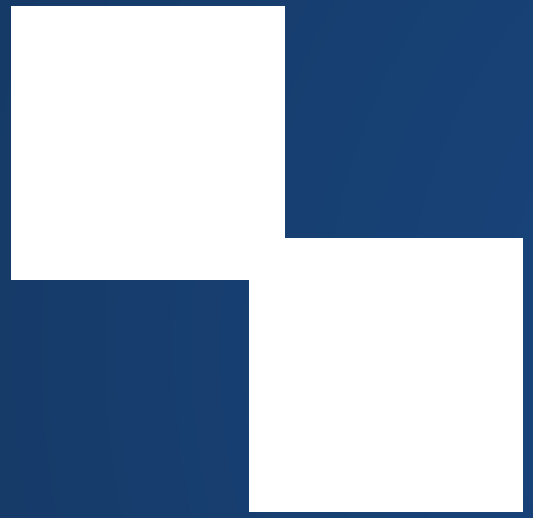
Why can we use draw/setup as methods in our classes?

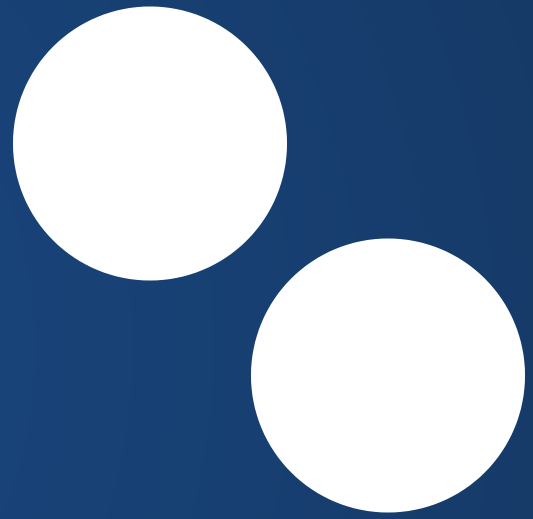
```
1  class CantStopMe {
2    void setup() { }
3    void draw() { }
4  }
5
6  class CodeRebellion {
7    void setup() { }
8    void draw() { }
9  }
10
11 class ItsAFreeCountry {
12   void setup() { }
13   void draw() { }
14 }
15
16 // Can only have one of these
17 void setup(){ }
18
19 // And one of these
20 void draw() { }
```

`draw()` and `setup()` are only special functions at the top level of Processing (i.e. not in a class). If you're putting them as methods in a class, you can have as many as you want.

That being said, the Processing interface tends to treat `draw` and `setup` as special names, so if you're more comfortable using other names, you can do that.

Is there a way to check object collisions other than checking if the x and y values of the two objects are equal?





In fact, that's not enough! You need to check if the x and y values are within certain ranges of each other, but you **also** need to know about the shapes of each object!

When designing classes, is simpler better?

When programming **in general**, simpler is better.
The trick is being able to assess when some
complexity now will pay off later.

Do you know any really screwed up OOP designs?

...wait and see ;)

Relating Classes to Each Other

The ability to bundle related data and functions together is nice, but can be obtained with other tools as well. In Python, we could use dicts, sets, and modules to bundle code/variables together.

The true power of OOP lies in the ability to specify relationships between classes (and thus between objects).

Composition

Suppose we are trying to design a `Bike` class. What pieces might we need for this class?

- Frame
- Wheels
- Brakes
- Drivetrain
- Handlebars

In English, we would say that "a bike *has a* frame". This is the has-a relation, or composition.

Animating a Bicycle

To animate:

- Bike must move
- Wheels must rotate

Additional rule: wheels on front and back must be the same model.

```
1  class Bicycle{
2    Frame frame;
3    Wheel frontW;
4    Wheel backW;
5    float x, y;
6    float wheelDist;
7
8    Bicycle(Frame f, Wheel w){
9      // Initialize bike
10   }
11
12   void displayBike(){
13     // ?
14   }
15
16   void moveBike(float dx){
17     // ?
18   }
19 }
```

```
1  void displayBike() {
2    frame.drawAt(x, y);
3    frontW.drawAt(x+wheelDist/2, y);
4    backW.drawAt(x-wheelDist/2, y);
5  }
```

```
1  void moveBike(float dx) {
2    this.x += dx;
3
4    // Why can we compute one rotation for both?
5    float wheelRotation = dx / (Math.PI * frontW.diameter);
6
7    frontW.rotate(wheelRotation);
8    backW.rotate(wheelRotation);
9  }
```

Note: we don't have to do 100% of the work ourselves!
Delegate work to components.

Why Use Composition?

Allow us to break complex problems into simpler ones.

Don't need to know how to draw the entire bike: just have to know where to draw the wheels and let the wheels draw themselves!

We can ignore how certain things are implemented.

How is `Wheel::rotate()` implemented? Don't know. Don't care!

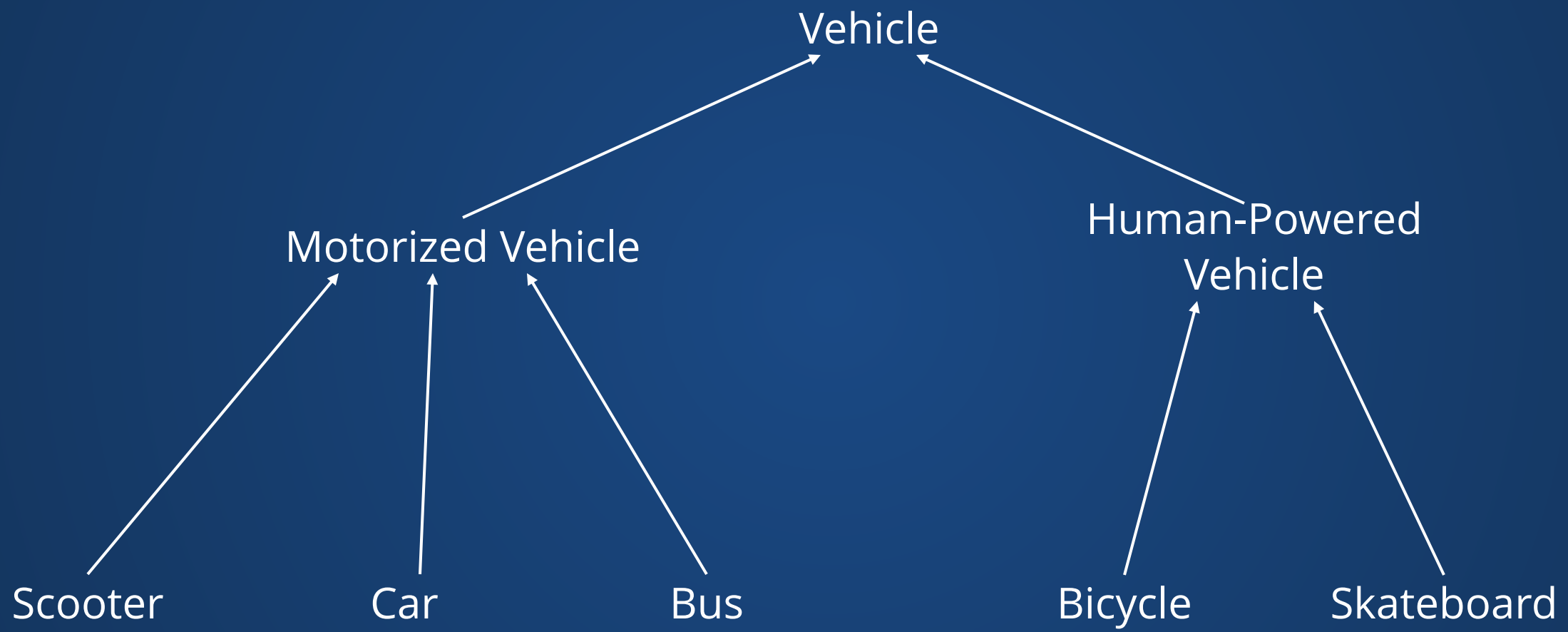
Inheritance

Suppose we are trying to model multiple forms of transport vehicles. What classes do we have?

- Bike
- Car
- Truck
- Scooter
- Skateboard

Has-a isn't really appropriate to describe the relationship between these!

Hierarchical Description

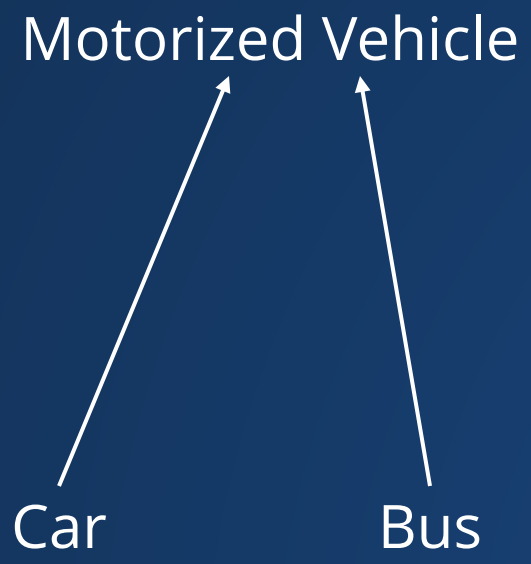


What might we call these arrows?

- A class can *inherit* from another class.
  - The class that inherits is known as a *subclass*, *derived class*, or *child class*.
  - The class that is inherited from is know as a *superclass*, *base class*, or *parent class*.
- A subclass *extends* a superclass. It contains all the fields and methods of the superclass, but can also be extended with more.

An example of an is-a relationship.

A car does not contain a motorized vehicle, it *is a* motorized vehicle. It can do everything a motorized vehicle can and more.



- Use `extends` keyword to declare that a class is a subclass of another.
- Subclass declares members that are not in the superclass
- Still has access to superclass members!

```
1  class MotorizedVehicle {
2    float fuel, speed;
3    void addFuel(float amount){ /* */ }
4    void accelerate(float rate){ /* */ }
5  }
6
7  class Car extends MotorizedVehicle {
8    color color;
9  }
10
11 class Bus extends MotorizedVehicle {
12   int numSeats;
13 }
```

```
1  Car myCar = new Car();
2  myCar.color = color(10, 20, 30);
3  myCar.addFuel(10.0);
4  myCar.accelerate(20.0);
```

Hmmmm.....

Constructors in Inheritance

When writing a constructor, construct the parent class by calling super().

```
1  class MotorizedVehicle {
2    float fuel, speed;
3    void addFuel(float amount){ /* */ }
4    void accelerate(float rate){ /* */ }
5
6    MotorizedVehicle(float fuel){
7      if (fuel < 0){
8        // ERROR: Fuel cannot be negative!
9        this.fuel = 0.0;
10     } else {
11       this.fuel = fuel;
12     }
13     this.speed = 0.0;
14   }
15 }
```

```
1  class Car extends MotorizedVehicle {
2    color color;
3
4    Car(color c, float fuel){
5      super(fuel);
6      this.color = c;
7    }
8  }
```

Subclass Relations + Overriding

Suppose we have a MotorVehicle object. What can we do with it?

```
1  class MotorizedVehicle {
2    float fuel, speed;
3    void addFuel(float amount){ /* */ }
4    void accelerate(float rate){ /* */ }
5  }
6
7  class Car extends MotorizedVehicle {
8    color color;
9  }
10
11 class Bus extends MotorizedVehicle {
12   int numSeats;
13 }
```

```
1 void doAThing(MotorVehicle v){
2   // What can we do to v?
3 }
```

What if `v` is a `Car` or a `Bus`?

Since a `Car` *is a* `MotorizedVehicle`, anywhere that we expect a `MotorizedVehicle`, we can use a `Car` instead!

```
1 void doAThing(MotorVehicle v){
2   // Check fuel
3   if (fuel == 0){
4     v.addFuel(10);
5   }
6   v.accelerate(5);
7 }
```

```
1 MotorVehicle v1 = new MotorVehicle();
2 Car c1 = new Car();
3 Bus b1 = new Bus();
4
5 doAThing(v1);
6 doAThing(c1);
7 doAThing(b1);
8
9 MotorVehicle v1 = new Car(); // Tricky
```

Note that the opposite is not true!

```
1 void doThing2(Car c){ /* ... */ }
2
3 MotorVehicle v1 = new MotorVehicle();
4 doThing2(v1); // ERROR
```

What if we implement a method in the derived class that has the same name/types as one in the base?

```
1 class Foo {
2   ...
3   void printHello() {
4    print("Hi I'm Foo");
5   }
6 }
```

```
1 class Bar extends Foo {
2   ...
3   void printHello() {
4    print("Hi I'm Bar");
5   }
6 }
```

```
1 Foo f = new Foo();
2 Bar b = new Bar();
3 f.printHello();
4 b.printHello();
```

The method in the child class *overrides* the one in the parent!

Example of Overrides

Motor Vehicles consume fuel to accelerate.

```
1  class MotorizedVehicle {
2    float fuel, speed;
3    void addFuel(float amount){
4      fuel += amount;
5    }
6    void accelerate(float rate){
7      this.speed += rate;
8      this.fuel -= 0.01 * rate;
9    }
10 }
```

Buses need to consume more fuel the more seats they have.

```
1 class Bus extends MotorizedVehicle {
2   int seats;
3   void accelerate(float rate){
4     this.speed += rate;
5     this.fuel -= seats * 0.01 * rate;
6   }
7 }
```

Do we know what class v actually is?

```
1  void goGoGo(MotorVehicle v){
2    v.accelerate(100);
3  }
```

Do we need to worry about acceleration being incorrect (i.e. we need to do something to v to make it "work"?)

Warning

Don't Mix The Relations!

People sometimes get confused about the difference between *is-a* and *has-a* and model their problem with OOP incorrectly. This leads to what has sometimes been called "certifiably insane" designs.

Real life examples from people who really should have known better:

> *CarWithBumperSticker is a subclass of Car*

> *A circle inherits from a point because I need to show off inheritance for this textbook*

> *Person should multiple-inherit* from Head, Body, Arm, and Leg*

> *Car inherits from ParkingGarage because parking garages contain cars*

* Multiple Inheritance is a feature that does not exist in Java or Processing

Example: ColorSpot

You may want to pull out your laptops and follow along, since some of this will be on the hands-on.

Hands-On: ColorSpot Class

1. Get the `Spot` subclass `ColorSpot` working in your code.
2. Create another `Spot` subclass called `TwoSpots`. (Dropdown > New Tab). `TwoSpots` displays two ellipses around a center point.
3. Write code that lets you move a `TwoSpots` object to a specific location. Do this without modifying the `TwoSpots` class directly.

Index Cards!

1. Your name and EID.

2. One thing that you learned from class today. You are allowed to say "nothing" if you didn't learn anything.

3. One question you have about something covered in class today. You *may not* respond "nothing".

4. (Optional) Any other comments/questions/thoughts about today's class.