Another Example: Association (1/6)

- Here we have the class `CS15Professor`
- We want `CS15Professor` to know about his Head TAs—he didn't create them or vice versa, hence no containment – they are peer objects
- And we also want Head TAs to know about `CS15Professor`
- Let's set up associations!

```
public class CS15Professor {

    // declare instance variables here
    // and here…
    // and here…
    // and here!

    public CS15Professor(/* parameters */) {


        // initialize instance variables!
        // …
        // …
        // …
    }

    /* additional methods elided */
}
```

Another Example: Association (2/6)

- The `CS15Professor` needs to know about 5 Head TAs, all of whom will be instances of the class `HeadTA`
- Once he knows about them, he can call methods of the class `HeadTA` on them: `remindHeadTA`, `setUpLecture`, etc.
- Take a minute and try to fill in this class

```
public class CS15Professor {

    // declare instance variables here
    // and here…
    // and here…
    // and here!

    public CS15Professor(/* parameters */) {


        // initialize instance variables!
        // …
        // …
        // …
    }

    /* additional methods elided */
}
```

Another Example: Association (3/6)

- Here's our solution!
- Remember, you can choose your own names for the instance variables and parameters
- The `CS15Professor` can now send a message to one of his `HeadTA`s like this:

```
_hta2.setUpLecture();
```

```
public class CS15Professor {

    private HeadTA _hta1;
    private HeadTA _hta2;
    private HeadTA _hta3;
    private HeadTA _hta4;
    private HeadTA _hta5;

    public CS15Professor(HeadTA firstTA,
        HeadTA secondTA, HeadTA thirdTA
        HeadTA fourthTA, HeadTA fifthTA) {

        _hta1 = firstTA;
        _hta2 = secondTA;
        _hta3 = thirdTA;
        _hta4 = fourthTA;
        _hta5 = fifthTA;
    }

    /* additional methods elided */

}
```

Another Example: Association (4/6)

- We've got the `CS15Professor` class down
- Now let's create a professor and head TAs from a class that contains all of them: `CS15App`
- Try and fill in this class!
  - you can assume that the `HeadTA` class takes no parameters in its constructor

```
public class CS15App {

    // declare CS15Professor instance var
    // declare five HeadTA instance vars
    // …
    // …
    // …

    public CS15App() {
        // instantiate the professor!
        // …
        // …
        // instantiate the five HeadTAs
    }
}
```

Another Example: Association (5/6)

- We declare `_andy`, `_julie`, `_angel`, `_noah`, `_taylor` and `_lucy` as instance variables
- In the constructor, we instantiate them
- Since the constructor of `CS15Professor` takes in 5 `HeadTA`s, we pass in `_julie`, `_angel`, `_noah`, `_taylor` and `_lucy`

```
public class CS15App {

    private CS15Professor _andy;
    private HeadTA _julie;
    private HeadTA _angel;
    private HeadTA _noah;
    private HeadTA _taylor;
    private HeadTA _lucy;

    public CS15App() {
        _julie = new HeadTA();
        _angel = new HeadTA();
        _noah = new HeadTA();
        _taylor = new HeadTA();
        _lucy = new HeadTA();
        _andy = new CS15Professor(_julie,
              _angel, _noah, _taylor,
             _lucy);
    }
}
```

Another Example: Association (6/6)

```
public class CS15Professor {

    private HeadTA _hta1;
    private HeadTA _hta2;
    private HeadTA _hta3;
    private HeadTA _hta4;
    private HeadTA _hta5;

    public CS15Professor(HeadTA firstTA,
         HeadTA secondTA, HeadTA thirdTA
         HeadTA fourthTA, HeadTA fifthTA) {

        _hta1 = firstTA;
        _hta2 = secondTA;
        _hta3 = thirdTA;
        _hta4 = fourthTA;
        _hta5 = fifthTA;
    }

    /* additional methods elided */
}
```

```
public class CS15App {

    private CS15Professor _andy;
    private HeadTA _julie;
    private HeadTA _angel;
    private HeadTA _noah;
    private HeadTA _taylor;
    private HeadTA _lucy;

    public CS15App() {
        _julie = new HeadTA();
        _angel = new HeadTA();
        _noah = new HeadTA();
        _taylor = new HeadTA();
        _lucy = new HeadTA();
        _andy = new CS15Professor(_julie,
              _angel, _noah, _taylor,
             _lucy);
    }
}
```

More Associations (1/5)

- Now the `CS15Professor` can call on the `HeadTA`s but can the `HeadTA`s call on the `CS15Professor` too?
- NO: Need to set up another association
- Can we just do the same thing and pass `_andy` as a parameter into each `HeadTA`s constructor?

```
public class CS15App {

  private CS15Professor _andy;
  private HeadTA _julie;
  private HeadTA _angel;
  private HeadTA _noah;
  private HeadTA _taylor;
  private HeadTA _lucy;

  public CS15App() {
    _julie = new HeadTA();
    _angel = new HeadTA();
    _noah = new HeadTA();
    _taylor = new HeadTA();
    _lucy = new HeadTA();
    _andy = new CS15Professor(_julie,
              _angel, _noah, _taylor,
              _lucy);
  }
}
```

Code from previous slide

More Associations (2/5)

- When we instantiate `_julie`, `_angel`, `_noah`, `_taylor`, and `_lucy`, we would like to use a modified `HeadTA` constructor that takes an argument, `_andy`
- But `_andy` hasn't been instantiated yet (will get a `NullPointerException`)! And we can't initialize `_andy` first because the `HeadTA`s haven't been created yet…
- How to break this deadlock?

```
public class CS15App {

  private CS15Professor _andy;
  private HeadTA _julie;
  private HeadTA _angel;
  private HeadTA _noah;
  private HeadTA _taylor;
  private HeadTA _lucy;

  public CS15App() {
    _julie = new HeadTA();
    _angel = new HeadTA();
    _noah = new HeadTA();
    _taylor = new HeadTA();
    _lucy = new HeadTA();
    _andy = new CS15Professor(_julie,
              _angel, _noah, _taylor,
              _lucy);
  }
}
```

Code from previous slide

More Associations (3/5)

- Instantiate `_julie`, `_angel`, `_noah`, `_taylor`, and `_lucy` **before** we instantiate `_andy`
- Use a new method (mutator), `setProf`, and pass `_andy` to each `HeadTA`

```
public class CS15App {

   private CS15Professor _andy;
   private HeadTA _julie;
   private HeadTA _angel;
   private HeadTA _noah;
   private HeadTA _taylor;
   private HeadTA _lucy;

   public CS15App() {
     _julie = new HeadTA();
     _angel = new HeadTA();
     _noah = new HeadTA();
     _taylor = new HeadTA();
     _lucy = new HeadTA();
     _andy = new CS15Professor(_julie,
                 _angel, _noah, _taylor,
           _lucy);

     _julie.setProf(_andy);
     _angel.setProf(_andy);
     _noah.setProf(_andy);
     _taylor.setProf(_andy);
     _lucy.setProf(_andy);
   }
}
```

More Associations (4/5)

```
public class HeadTA {

    private CS15Professor _professor;

    public HeadTA() {

        //Other code elided

    }

    public void setProf(CS15Professor prof)
    {
        _professor = prof;
    }
}
```

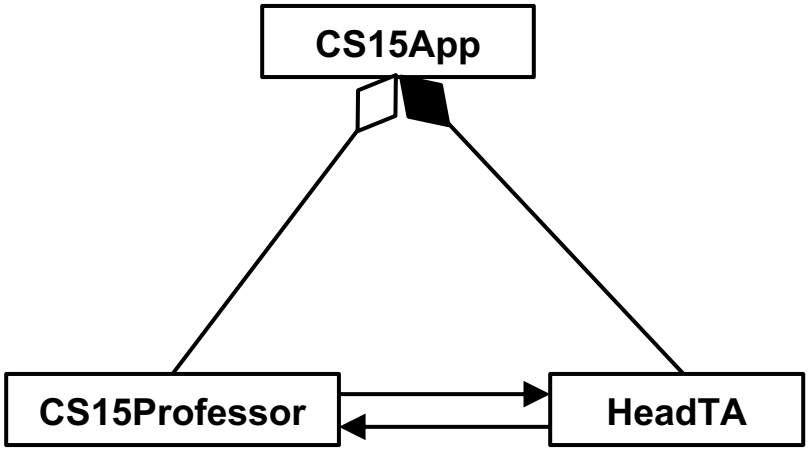
- Now each HeadTA will know about _andy!

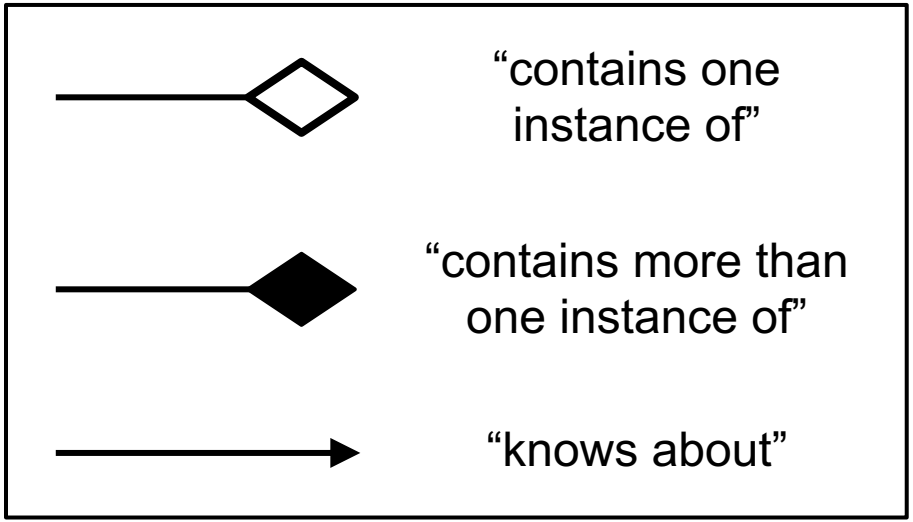
```
public class CS15App {

    private CS15Professor _andy;
    private HeadTA _julie;
    private HeadTA _angel;
    private HeadTA _noah;
    private HeadTA _taylor;
    private HeadTA _lucy;

    public CS15App() {
      _julie = new HeadTA();
      _angel = new HeadTA();
      _noah = new HeadTA();
      _taylor = new HeadTA();
      _lucy = new HeadTA();
      _andy = new CS15Professor(_julie,
                  _angel, _noah, _taylor,
            _lucy);

      _julie.setProf(_andy);
      _angel.setProf(_andy);
      _noah.setProf(_andy);
      _taylor.setProf(_andy);
      _lucy.setProf(_andy);
    }
}
```

More Associations (5/5)

- But what happens if `setProf` is never called?
- Will the Head TAs be able to call methods on the `CS15Professor`?
- No! We would get a `NullPointerException`!
- So this is not a completely satisfactory solution, but we will learn more tools soon that will allow us to develop a more complete solution

Visualizing Containment and Association





Summary

Important concepts:

- Using **local variables**, whose scope is limited to a method
- Using **instance variables**, which store the properties of instances of a class for use by multiple methods—use them only for that purpose
- A variable that “goes out of scope” is **garbage collected**
  - for a local variable when the method ends
  - for an instance when the last reference to it is deleted
- **Containment**: when one object is a component of another so the container can therefore send the component it created messages
- **Association**: when one object knows about another object that is not one of its components—has to be set up explicitly

Lecture 5

Interfaces and Polymorphism

Outline

- Transportation Example
- Intro to Interfaces
- Implementing Interfaces
- Polymorphism

Recall: Declaring vs. Defining Methods

- What's the difference between declaring and defining a method?
  - method declaration has the scope (`public`), return type (`void`), method name and parameters (`makeSounds()`)
  - method definition is the body of the method – the actual implementation (the code that actually makes the sounds)

```
public class Dog {
    //constructor elided

    public void makeSounds() {
        this.bark();
        this.whine();
        this.bark();
    }
    public void bark() {
        //code elided
    }
    public void whine() {
        //code elided
    }
}
```

Using What You Know

- Imagine this program:
  - Lucy and Angel are racing from their dorms to CIT
    - whoever gets there first, wins!
    - catch: they don't get to choose their method of transportation
- Design a program that
  - assigns mode of transportation to each racer
  - starts the race
- For now, assume transportation options are `Car` and `Bike`

Goal 1: Assign transportation to each racer

- Need transportation classes
  - app needs to give one to each racer
- Let's use `Car` and `Bike` classes
- Both classes will need to describe how the transportation moves
  - `Car` needs `drive` method
  - `Bike` needs `pedal` method

Coding the project (1/4)

- Let's build transportation classes

```
public class Car {

    public Car() {//constructor
        //code elided
    }
    public void drive() {
        //code elided
    }
    //more methods elided
}
```

```
public class Bike {

    public Bike() {//constructor
        //code elided
    }
    public void pedal() {
        //code elided
    }
    //more methods elided
}
```

Goal 1: Assign transportation to each racer

- Need racer classes that will tell Lucy and Angel to use their type of transportation
  - `CarRacer`
  - `BikeRacer`
- What methods will we need? What capabilities should each `-Racer` class have?
- `CarRacer` needs to know when to use the car
  - write `useCar()` method
- `BikeRacer` needs to know when to use the bike
  - write `useBike()` method

Coding the project (2/4)

- Let's build the racer classes

```
public class CarRacer {
    private Car _car;

    public CarRacer() {
        _car = new Car();
    }

    public void useCar(){
        _car.drive();
    }
    //more methods elided
}
```

```
public class BikeRacer {
    private Bike _bike;

    public BikeRacer() {
        _bike = new Bike();
    }

    public void useBike(){
        _bike.pedal();
    }
    //more methods elided
}
```

Goal 2: Tell racers to start the race

- `Race` class contains `Racers`
  - `App` contains `Race`
- `Race` class will have `startRace()` method
  - `startRace()` tells each racer to use their transportation
- `startRace()` gets called in App

```
startRace:
    Tell _angel to useCar
    Tell _lucy to useBike
```

Coding the project (3/4)

- Given our CarRacer class, let's build the Race class

```
public class CarRacer {
    private Car _car;

    public CarRacer() {
        _car = new Car();
    }

    public void useCar(){
        _car.drive();
    }
    //more methods elided
}

//BikeRacer class elided
```

```
public class Race {
    private CarRacer _angel;
    private BikeRacer _lucy;

    public Race() {
        _angel = new CarRacer();
        _lucy = new BikeRacer();
    }

    public void startRace() {
        _angel.useCar();
        _lucy.useBike();
    }
}
```

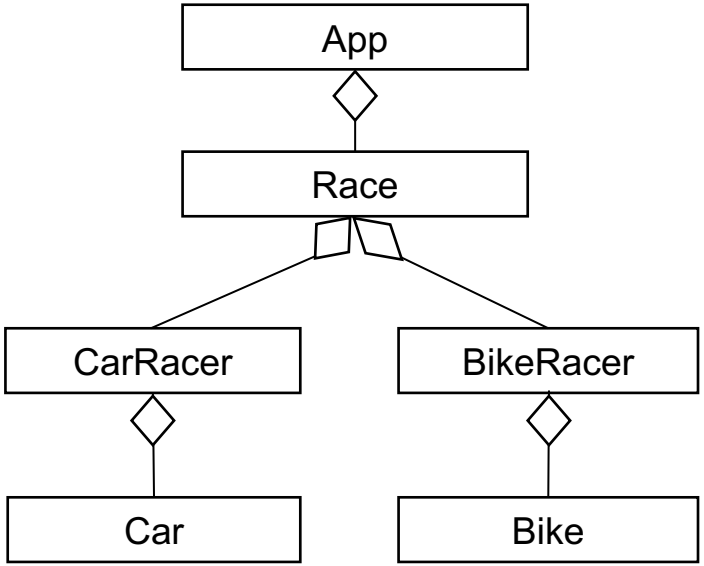
Coding the project (4/4)

```
public class App {

    public static void main (String[] args) {
        Race cs15Race = new Race();
        cs15Race.startRace();
        launch(args); //magic for now
    }

}
```

- Now build the App class
- Program starts with main()
- main() calls startRace() on cs15Race

```
//from the Race class

public void startRace() {
    _angel.useCar();
    _lucy.useBike();
}
```

What does our design look like?



How would this program run?

- An instance of `App` gets initialized by `main`
- `App`'s constructor initializes an instance of `Race`
- `Race`'s constructor initializes `_angel`, a `CarRacer` and `_lucy`, a `BikeRacer`
  - `CarRacer`'s constructor initializes `_car`, a `Car`
  - `BikeRacer`'s constructor initializes `_bike`, a `Bike`
- `App` calls `cs15Race.startRace()`
- `cs15Race` calls `_angel.useCar()` and `_lucy.useBike()`
- `_angel` calls `_car.drive()`
- `_lucy` calls `_bike.pedal()`

Can we do better?

Things to think about

- Do we need two different `Racer` classes?
  - we want multiple instances of `Racers` that use different modes of transportation
    - both classes are very similar, they just use their own mode of transportation (`useCar` and `useBike`)
    - do we need 2 different classes that serve essentially the same purpose?
  - but how can we simplify?

Solution 1: Create one Racer class with multiple useX methods!

- Create one Racer class
  - define different methods for each type of transportation
- _angel is instance of Racer and elsewhere we have:

```
Car angelsCar = new Car();
_angel.useCar(angelsCar);
```

  - Car's drive() method will be invoked
- But any given instance of Racer will need a new method to accommodate every kind of transportation!

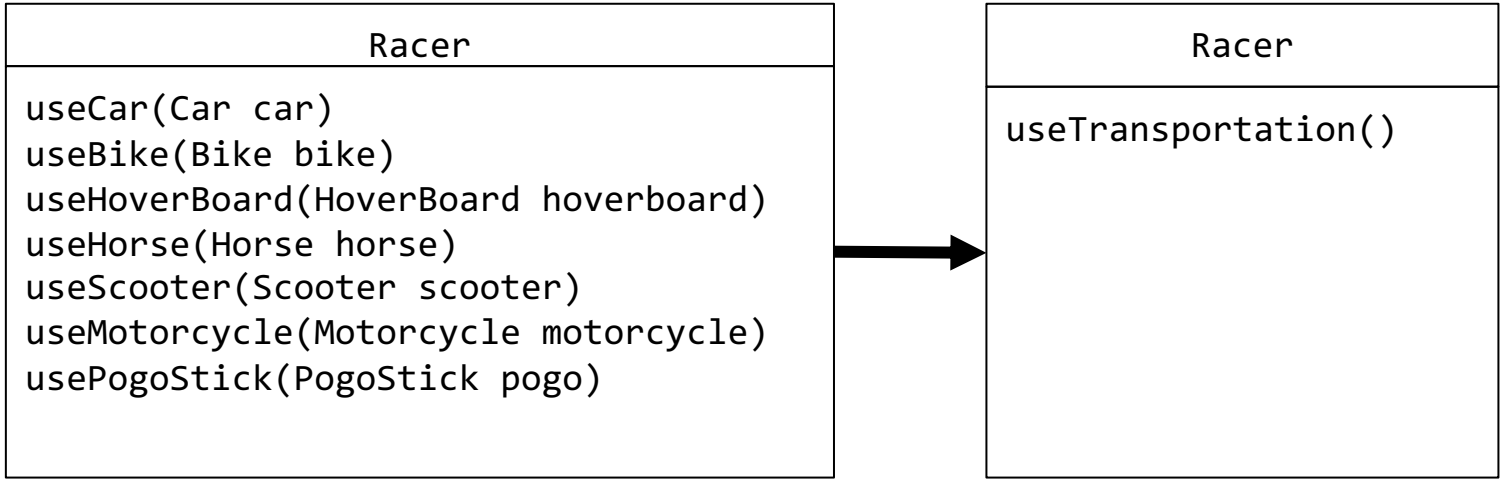
```
public class Racer {
    public Racer(){
        //constructor
    }

    public void useCar(Car myCar){
        myCar.drive();
    }

    public void useBike(Bike myBike){
        myBike.pedal();
    }
}
```

Solution 1 Drawbacks

- Now imagine all the CS15 TAs join the race and there are 10 different modes of transportation
- Writing these similar `useX()` methods is a lot of work for you, as the developer, and it is an inefficient coding style

```
public class Racer {

    public Racer() {
        //constructor
    }
    public void useCar(Car myCar){//code elided}
    public void useBike(Bike myBike){//code elided}
    public void useHoverboard(Hoverboard myHb){//code elided}
    public void useHorse(Horse myHorse){//code elided}
    public void useScooter(Scooter myScooter){//code elided}
    public void useMotorcycle(Motorcycle myMc) {//code elided}
    public void usePogoStick(PogoStick myPogo){//code elided}
    // And more…
}
```

Is there another solution?



- Can we go from left to right?

Interfaces and Polymorphism

- In order to simplify code, we need to learn
  - Interfaces
  - Polymorphism
  - we'll see how this new code works shortly:

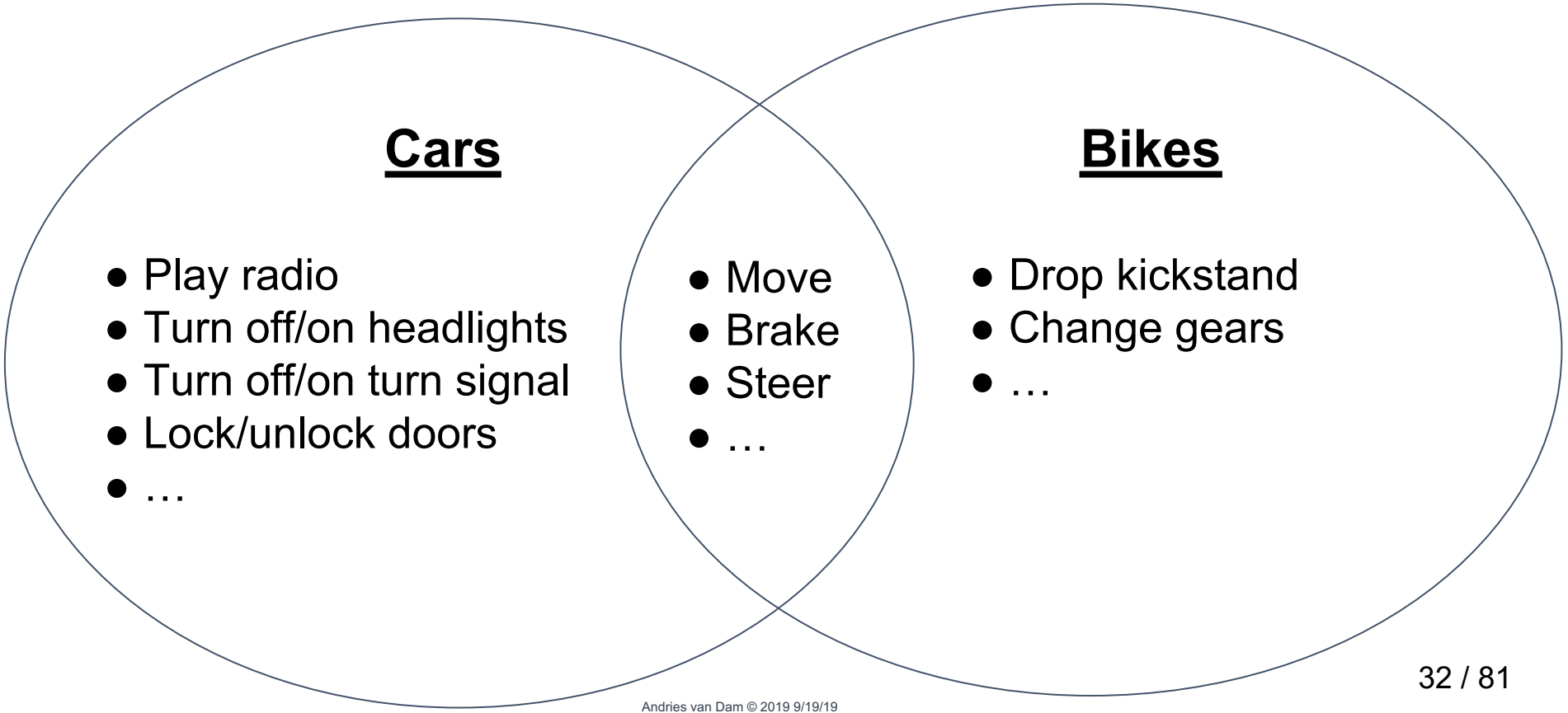
```
public class Car implements Transporter {

    public Car() {
        //code elided
    }
    public void drive(){
        //code elided
    }
    @Override
    public void move(){
        this.drive();
    }
    //more methods elided
}
```

```
public class Racer {

    //previous code elided
    public void useTransportation(
          Transporter transport) {
        transport.move();
    }

}
```

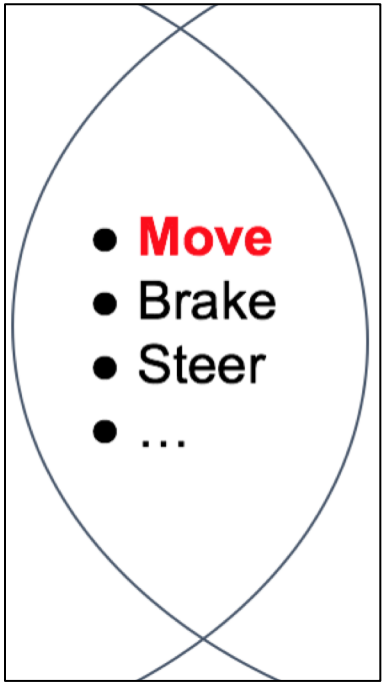
Interfaces: Spot the Similarities

- What do cars and bikes have in common?
- What do cars and bikes not have in common?

Cars vs. Bikes



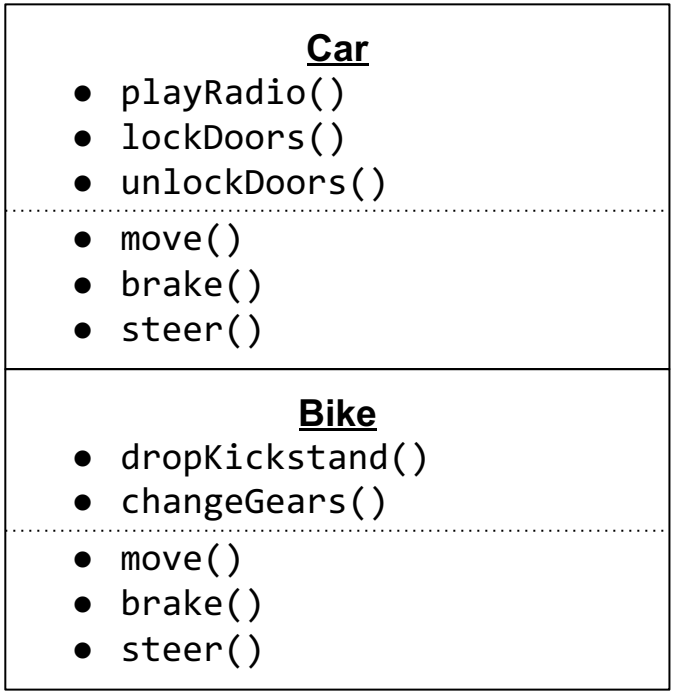
Digging deeper into the similarities



- How similar are they when they move?
  - do they move in same way?
- Not very similar
  - cars drive
  - bikes pedal
- Both can move, but in different ways

Can we model this in code?

- Many real-world objects have several broad similarities
  - cars and bikes can move
  - cars and laptops can play radio
  - phones and Teslas can be charged
- Take `Car` and `Bike` class
  - how can their similar functionalities get enumerated in one place?
  - how can their broad relationship get portrayed through code?



Introducing Interfaces

- Interfaces group similar capabilities/function of different classes together
- Model “acts-as” relationship
- `Car`s and `Bike`s could implement a `Transporter` interface
  - they can transport people from one place to another
  - they “act as” transporters
    - objects that can move
    - have shared functionality, such as moving, braking, turning etc.
  - for this lecture, interfaces are green and classes that implement them pink

Introducing Interfaces

- Interfaces are contracts that classes agree to
- If classes choose to implement given interface, it must define all methods declared in interface
  - if classes don't implement one of interface's methods, the compiler raises errors
    - later we'll discuss strong motivations for this contract enforcement
- Interfaces don't define their methods – classes that implement them do
  - interfaces only care about the fact that the methods get defined - not how – *implementation-agnostic*
- Models similarities while ensuring consistency
  - what does this mean?

Models Similarities while Ensuring Consistency (1/2)

Let’s break that down into two parts

1) Model Similarities

2) Ensure Consistency

Models Similarities While Ensuring Consistency (2/2)

- How does this help our program?
- We know `Car`s and `Bike`s both need to move
  - i.e., should both have some `move()` method
  - let compiler know that too!
- Let's make the `Transporter` interface!
  - what methods should the `Transporter` interface declare?
    - `move()`
    - only using a `move()` for simplicity, but `brake()`, etc., would also be useful
  - compiler doesn't care how method is defined, just that it has been defined
  - general tip: methods that interface declares should model functionality **all** implementing classes share

Declaring an Interface (1/4)

What does this look like?

```
public interface Transporter {

    public void move();

}
```

- Declare it as interface rather than class

Declaring an Interface (2/4)

What does this look like?

```
public interface Transporter {

    public void move();

}
```

- Declare methods - the contract
- In this case, only one method required: `move()`
- All classes that sign contract (implement this interface) must define actual implementation of any declared methods

Declaring an Interface (3/4)

What does this look like?

```
public interface Transporter {

    public void move();

}
```

- Interfaces are only contracts, not classes that can be instantiated
- Interfaces can only declare methods - not define them
- Notice: method declaration end with semicolons, not curly braces!

Declaring an Interface (4/4)

What does this look like?

```
public interface Transporter {

    public void move();

}
```

- That's all there is to it!
- Interfaces, just like classes, have their own `.java` file. This file would be `Transporter.java`

Implementing an Interface (1/6)

```
public class Car implements Transporter {

    public Car() {
        // constructor
    }

    public void drive() {
        // code for driving the car
    }

}
```

- Let's modify `Car` to implement `Transporter`
  - declare that `Car` "acts-as" `Transporter`
- Add `implements Transporter` to class declaration
- Promises compiler that `Car` will define all methods in `Transporter` interface
  - i.e., `move()`

Implementing an Interface (2/6)

```
public class Car implements Transporter {

    public Car() {
        // constructor
    }

    public void drive() {
        // code for driving the car
    }

}
```

"Error: Car does not override method move() in Transporter" *

- Will this code compile?
  - nope :(
- Never implemented `move()` -- `drive()` doesn't suffice. Compiler will complain accordingly

*Note: the full error message is "`Car is not abstract and does not override abstract method move() in Transporter`." We'll get more into the meaning of abstract in a later lecture.

Implementing an Interface (3/6)

```
public class Car implements Transporter {

    public Car() {
        // constructor
    }

    public void drive() {
        //code for driving car
    }

    @Override
    public void move() {
        this.drive();
    }

}
```

- Next: honor contract by defining a `move()` method
- Method ***signature*** (name and number/type of arguments) must match how it's declared in interface

Implementing an Interface (4/6)

What does `@Override` mean?

```
public class Car implements Transporter {

    public Car() {
        // constructor
    }

    public void drive() {
        //code for driving car
    }

    @Override
    public void move() {
        this.drive();
    }

}
```

- Include `@Override` right above the method signature
- `@Override` is an annotation – a signal to the compiler (and to anyone reading your code)
  - allows compiler to enforce that interface actually has method declared
  - more explanation of `@Override` in next lecture
- Annotations, like comments, have no effect on how code behaves at runtime

Implementing an Interface (5/6)

```
public class Car implements Transporter {

    //previous code elided

    public void drive() {
        //code for driving car
    }

    @Override
    public void move() {
        this.drive();
        this.brake();
        this.drive();
    }
    public void brake() { //code elided}
}
```

- Defining interface method is like defining any other method
- Definition can be as complex or as simple as it needs to be
- Ex.: Let's modify `Car`'s move method to include braking
- What will instance of `Car` do if `move()` gets called on it?

Implementing an Interface (6/6)

- As with signing multiple contracts, classes can implement multiple interfaces
  - “I signed my rent agreement, so I'm a renter, but I also signed my employment contract, so I'm an employee. I'm the same person.”
  - what if I wanted `Car` to change color as well?
  - create a `Colorable` interface
  - add that interface to `Car`’s class declaration
- Class implementing interfaces must define every single method from each interface

```
public interface Colorable {

    public void setColor(Color c);
    public Color getColor();

}

public class Car implements Transporter, Colorable {

    public Car(){ //body elided }
    //@Override annotation elided
    public void drive(){ //body elided }
    public void move(){ //body elided }
    public void setColor(Color c){ //body elided }
    public Color getColor(){ //body elided  }
}
```

Modeling Similarities While Ensuring Consistency

- Interfaces are formal contracts and ensure consistency
  - compiler will check to ensure all methods declared in interface are defined
- Can trust that any object from class that implements `Transporter` can `move()`
- Will know how 2 classes are related if both implement `Transporter`

TopHat Question

Which statement of this program is incorrect?

```
A. public interface Colorable {
    B. public Color getColor() {
            return Color.WHITE;
    }
}
C. public class Rectangle implements Colorable {
        //constructor elided
        D. @Override
        public Color getColor() {
            E. return Color.PURPLE;
        }
}
```

TopHat Question

Given the following interface:

```
public interface Clickable {
    public void click();
}
```

Which of the following would work as an implementation of the `Clickable` interface? (don't worry about what `changeXPosition` does)

A.
```
@Override
public double click() {
    return this.changeXPosition(100.0);
}
```

B.
```
@Override
public void click(double xPosition) {
    this.changeXPosition(xPosition);
}
```

C.
```
@Override
public void clickIt() {
    this.changeXPosition(100.0);
}
```

D.
```
@Override
public void click() {
    this.changeXPosition(100.0);
}
```

Back to the CIT Race

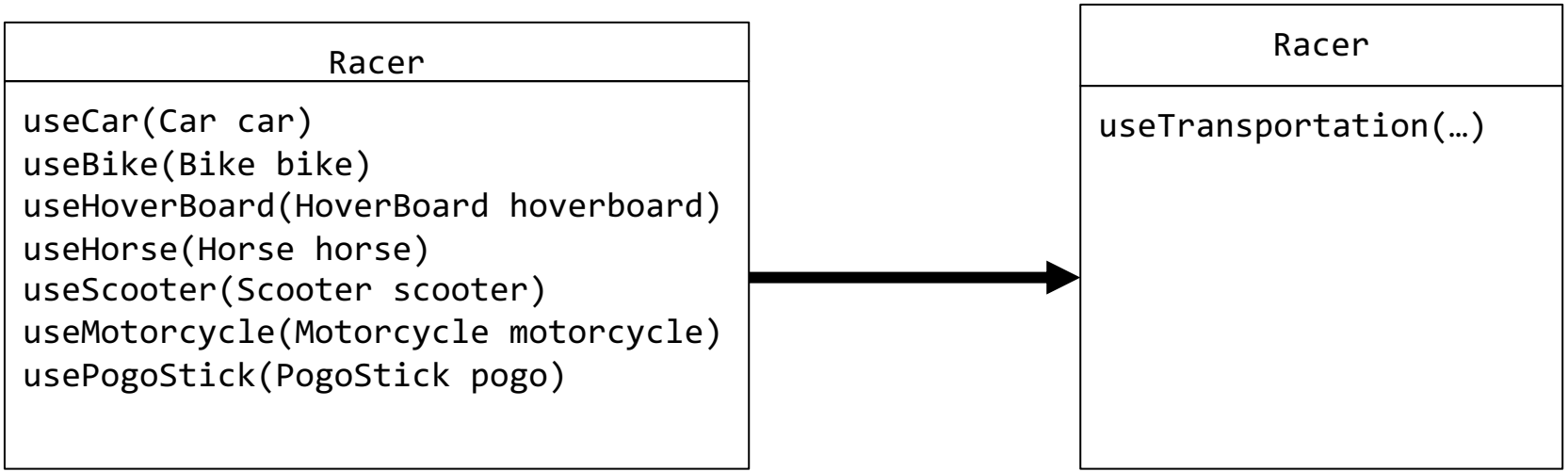
- Let's make transportation classes use an interface

```
public class Car implements Transporter {

    public Car() {
        //code elided
    }
    public void drive() {
        //code elided
    }
    @Override
    public void move() {
        this.drive();
    }
    //more methods elided
}
```

```
public class Bike implements Transporter {

    public Bike() {
        //code elided
    }
    public void pedal() {
        //code elided
    }
    @Override
    public void move() {
        this.pedal();
    }

    //more methods elided
}
```

Leveraging Interfaces

- Given that there's a guarantee that anything that implements `Transporter` knows how to `move`, how can it be leveraged to create single `useTransportation(…)` method?



Introducing Polymorphism

- Poly = many, morph = forms
- A way of coding generically
  - way of referencing many related objects as one generic type
    - cars and bikes can both `move()` → refer to them as `Transporter` objects
    - phones and Teslas can both `getCharged()` → refer to them as `Chargeable` objects, i.e., objects that implement `Chargeable` interface
    - cars and boomboxes can both `playRadio()` → refer to them as `RadioPlayer` objects
- How do we write one generic `useTransportation(…)` method?

What would this look like in code?

```
public class Racer {

    //previous code elided
    public void useTransportation(Transporter transportation) {
        transportation.move();
    }

}
```


This is polymorphism! `transportation` object passed in could be instance of `Car`, `Bike`, etc., i.e., any class that **implements** the interface

Let’s break this down

There are two parts to implementing polymorphism:

1. Actual vs. Declared Type
2. Method resolution

```
public class Racer {

    //previous code elided
    public void useTransportation(Transporter transportation) {
        transportation.move();
    }

}
```

Actual vs. Declared Type (1/2)

- Consider following piece of code:

```
Transporter angelsCar = new Car();
```

- We say "`angelsCar` is of type `Transporter`", but we instantiate a new `Car`...is that legal?
  - doesn't Java do "strict type checking"? (type on LHS = type on RHS)
  - how can instances of `Car` get stored in `Transporter` variable?

Actual vs. Declared Type (2/2)

- Can treat `Car`/`Bike` objects as `Transporter` objects
- `Car` is the actual type
  - Java compiler will look in this class for the definition of any method called on `transportation`
- `Transporter` is the declared type
  - compiler will limit any caller so it can only call methods on instances that are declared as `Transporter` objects AND are defined in that interface
- If `Car` defines `playRadio()` method, is this correct? `transportation.playRadio()`

```
Transporter transportation = new Car();
transportation.playRadio();
```

Nope. The `playRadio()` method is not declared in `Transporter` interface, therefore compiler does not recognize it as a valid method call

Determining the Declared Type

- What methods must `Car` and `Bike` have in common?
  - `move()`
- How do we know that?
  - they implement `Transporter`
    - guarantees that they have `move()` method, plus whatever else is appropriate to that class
- Think of `Transporter` like the "lowest common denominator"
  - it's what all transportation classes will have in common

```
class Bike implements Transporter {
    void move();
    void dropKickstand();
    //etc.
}
```

```
class Car implements Transporter {
    void move();
    void playRadio();
    //etc.
}
```

Is this legal?

```
Transporter lucysBike = new Bike();
```

```
Transporter lucysCar = new Car();
```

```
Transporter lucysRadio = new Radio();
```

`Radio` wouldn't implement `Transporter`. Since `Radio` cannot "act as" a `Transporter`, you cannot treat it as a `Transporter`

Motivations for Polymorphism

- Many different kinds of transportation but only care about their shared capability
  - i.e., how they move
- Polymorphism let programmers sacrifice specificity for generality
  - treat any number of classes as their lowest common denominator
  - limited to methods declared in that denominator
    - can only use methods declared in `Transporter`
- For this program, that sacrifice is ok!
  - `Racer` doesn't care if an instance of `Car` can `playRadio()` or if an instance of `Bike` can `dropKickstand()`
  - only method `Racer` wants to call is `move()`

Polymorphism in Parameters

- What are implications of this method declaration?

```
public void useTransportation(Transporter transportation) {
    //code elided
}
```

- `useTransportation` will accept any object that implements `Transporter`
- We say that `Transporter` is the (declared) type of the parameter
- We can pass in an instance of any class that implements the `Transporter` interface
- `useTransportation` can only call methods declared in `Transporter`

Is this legal?

```
Transporter lucysBike = new Bike();
_lucy.useTransportation(lucysBike);

Car lucysCar = new Car();
_lucy.useTransportation(lucysCar);

Radio lucysRadio = new Radio();
_lucy.useTransportation(lucysRadio);
```

Even though `lucysCar` is declared as a `Car`, the compiler can still verify that it implements `Transporter`

A `Radio` wouldn't implement `Transporter`. Therefore, `useTransportation()` cannot treat it like a `Transporter` object

Why move()? (1/2)

- Why call move()?
- What move() method gets executed?

```
public class Racer {

    //previous code elided
    public void useTransportation(Transporter transportation) {
        transportation.move();
    }

}
```

Why `move()`? (2/2)

- Only have access to `Transporter` object
  - cannot call `transportation.drive()` or `transportation.pedal()`
    - that's okay, because all that's needed is `move()`
  - limited to the methods declared in `Transporter`

Method Resolution: Which `move()` is executed?

- Consider this line of code in `Race` class:

```
_lucy.useTransportation(new Bike());
```

- Remember what `useTransportation` method looked like

```
public void useTransportation(Transporter transportation) {
    transportation.move();
}
```

What is “actual type” of transportation in

```
_lucy.useTransportation(new Bike());
```
?

Method Resolution (1/4)

```
public class Race {

    private Racer_lucy;
    //previous code elided

    public void startRace() {
        _lucy.useTransportation(new Bike());
    }
}
```

```
public class Racer {
    //previous code elided

    public void useTransportation(Transporter
    transportation) {
        transportation.move();
    }

}
```

- `Bike` is actual type
  - `_lucy` was handed an instance of `Bike`
    - `new Bike()` is argument
- `Transporter` is declared type
  - `_lucy` as `Racer` treats `Bike` object as `Transporter` object
- So… what happens in `transportation.move()`?
  - What `move()` method gets used?

Method Resolution (2/4)

```
public class Race {
    //previous code elided
    public void startRace() {
        _lucy.useTransportation(new Bike());
    }
}
```

```
public class Racer {
    //previous code elided
    public void useTransportation(Transporter
    transportation) {
        transportation.move();
    }
}
```

```
public class Bike implements Transporter {
    //previous code elided
    public void move() {
        this.pedal();
    }
}
```

- `_lucy` is a `Racer`
- `Bike`'s `move()` method gets used
- Why?
  - `Bike` is the actual type
    - compiler will execute methods defined in `Bike` class
  - `Transporter` is the declared type
    - compiler limits methods that can be called to those declared in `Transporter` interface

Method Resolution (3/4)

- What if `_lucy` received an instance of `Car`?
  - What `move()` method would get called then?
    - `Car`'s!

```
public class Race {

    //previous code elided

    public void startRace() {
        _lucy.useTransportation(new Car());
    }
}
```

Method Resolution (4/4)

- This method resolution is example of dynamic binding, which is when actual method implementation used is not determined until runtime
  - contrast with static binding, in which method gets resolved at compile time
- `move()` method is bound dynamically – the compiler does not know which `move()` method to use until program runs
  - same "`transport.move()`" line of code could be executed indefinite number of times with different method resolution each time

TopHat Question

Given the following class:

```
public class Laptop implements Typeable, Clickable { //two interfaces
   public void type() {
     // code elided
   }
   public void click() {
       //code elided
}
```

Given that `Typeable` has declared the `type()` method and `Clickable` has declared the `click()` method, which of the following calls is valid?

A.
```
Typeable macBook= new Typeable();
macBook.type();
```

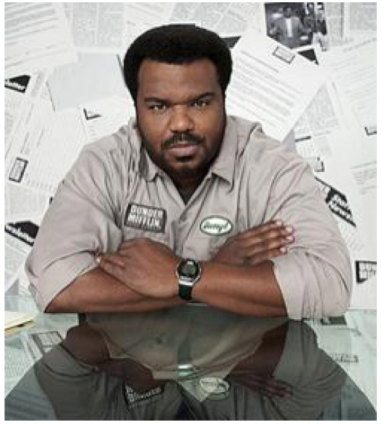
B.
```
Clickable macBook = new Clickable();
macBook.type();
```

C.
```
Typeable macBook= new Laptop();
macBook.click();
```

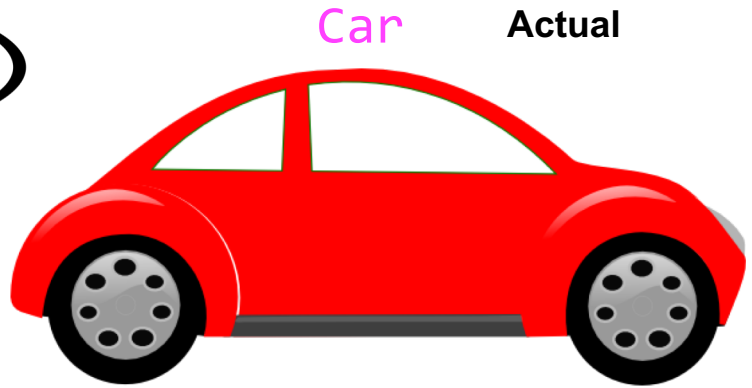
D.
```
Clickable macBook = new Laptop();
macBook.click();
```

Why does calling methods on polymorphic objects work? (1/2)

- Declared type and actual type work together
  - declared type keeps things generic
    - can reference a lot of objects using one generic type
  - actual type ensures specificity
    - when defining implementing class, methods can get implemented in any way



Declared



Actual

Why does calling methods on polymorphic objects work? (2/2)

- Declared type and actual type work together
  - declared type keeps things generic
    - can reference a lot of objects using one generic type
  - actual type ensures specificity
    - when defining implementing class, methods can get implemented in any way

This is my `Transporter` object!

Declared

`Bike` **Actual**

When to use polymorphism?

- Using only functionality declared in interface or specialized functionality from implementing class?
  - if only using functionality from the interface → polymorphism!
  - if need specialized methods from implementing class, don't use polymorphism
- If defining `goOnScenicDrive()`…
  - want to put `topDown()` on `Convertible`, but not every `Car` can put top down
    - don't use polymorphism, every `Car` can't `goOnScenicDrive()` i.e., can't code generically

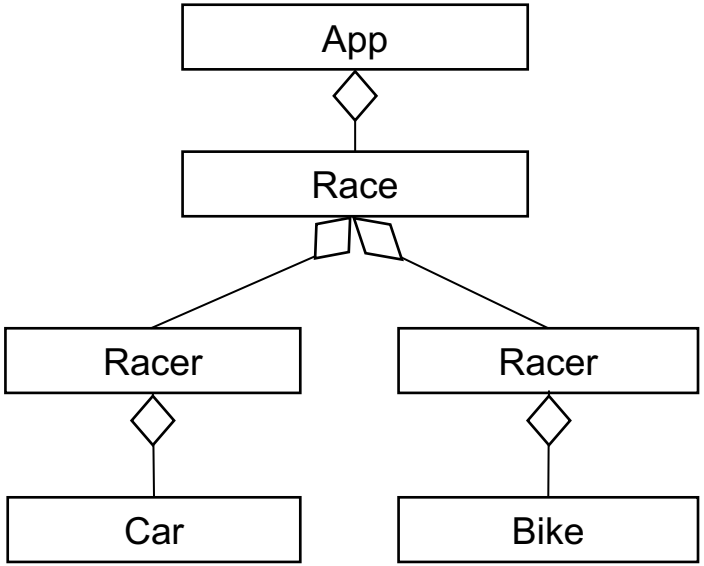
Why use interfaces?

- Contractual enforcement
  - will guarantee that class has certain capabilities
    - `Car` implements `Transporter`, therefore it must know how to `move()`
- Polymorphism
  - can have implementation-agnostic classes and methods
    - know that these capability exists, don't care how they're implemented
    - allows for more generic programming
      - `useTransportation` can take in any `Transporter` object
      - can easily extend this program to use any form of transportation, with minimal changes to existing code
    - an extremely powerful tool for extensible programming

Why is this important?

- With 2 modes of transportation!
- Old Design:
  - need more classes → more specialized methods (`useRollerblades()`, `useBike()`, etc)
- New Design:
  - as long as the new classes implement `Transporter`, `Racer` doesn't care what transportation it has been given
  - don't need to change `Racer`!
    - less work for you!
    - just add more transportation classes that implement `Transporter`
    - "need to know" principle, aka "separation of concerns"

What does our new design look like?



How would this program run?

- An instance of `App` gets initialized by `main`
- `App`'s constructor initializes an instance of `Race`
- `Race`'s constructor initializes `_angel`, a `Racer` and `_lucy`, a `Racer`
- `App` calls `cs15Race.startRace()`
- `cs15Race` calls:
  - `_angel.useTransportation(new Car())`,
  - `_lucy.useTransportation(new Bike())`
- `useTransportation(new Car())` initializes a `Car` and calls `Car`'s `move()` method which calls `this.drive()`
- `useTransportation(new Bike())` initializes a `Bike` and calls `Bike`'s `move()` method which calls `this.pedal()`

The Program

```
public class App {
    public App() {
        Race r = new Race();
        r.startRace();
    }
}
```

```
public class Race {
    private Racer _angel, _lucy;

    public Race(){
        _angel = new Racer();
        _lucy = new Racer();
    }
    public void startRace() {
        _angel.useTransportation(new Car());
        _lucy.useTransportation(new Bike());
    }
}
```

```
public interface Transporter {
    public void move();
}
```

```
public class Racer {
   public Racer() {}

   public void useTransportation(Transporter transport){
       transport.move();
   }
}
```

```
public class Car implements Transporter {
   public Car() {}
   public void drive() {
      //code elided
   }
   public void move() { //missing @Override
       this.drive();
   }
}
```

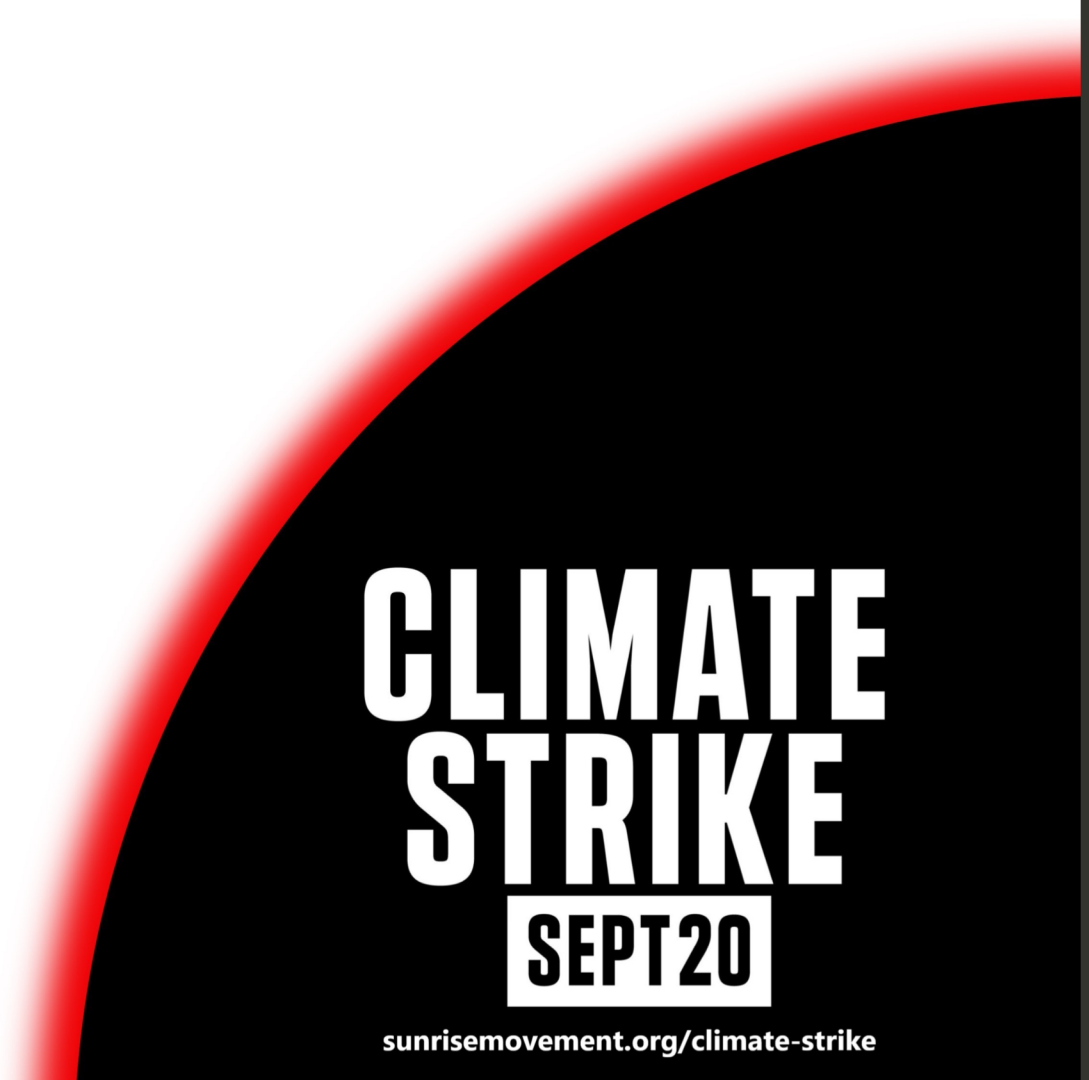
```
public class Bike implements Transporter {
   public Bike() {}
   public void pedal() {
       //code elided
   }
   public void move() { //missing @Override
       this.pedal();
   }
}
```

In Summary

- Interfaces are contracts, can’t be instantiated
  - force classes that implement them to define specified methods
- Polymorphism allows for generic code
  - treats multiple classes as their “generic type” while still allowing specific method implementations to be executed
- Polymorphism + Interfaces
  - generic coding
- Why is it helpful?
  - want you to be the laziest (but cleanest) programmer you can be

Announcements

- AndyBot due today at 11:59pm
- Litebrite will be released on Saturday 9/21
  - Early hand-in: 9/24
  - On-time hand-in: 9/26
  - Late hand-in: 9/28
- TA Hours schedule on the course website
  - go there for questions related to any code related issues
- Conceptual Hours schedule on the course website
  - go there for questions related to any lecture or general material
- Review the TA Hours missive for more information
- Email section TAs before the first section of the week for swaps



bit.ly/ri-strike