ATA

- ATA is an **AI-powered teaching assistant** designed for CS15, available 24/7 as a first line of defense
  - this means that whenever you have a problem, ATA will always be available to you!
  - it does **not** mean that if ATA is unable to help you with your problem, you should give up – we have many other resources available to get you the help that you need.
- ATA is experimental. Sometimes it can be frustrating, or incorrect — but give it a chance!
- Ask **any** course-related questions you have! Please don't intentionally manipulate ATA or ask it unrelated questions
  - we look at your conversations to HELP you – understand the common confusions in course content (like how we look at Ed)
- Your usage and feedback will help tune both CS15 and ATA
- ATA is a supplement for Ed and Office hours, not a replacement.

Lecture 6

Interfaces and Polymorphism

Outline

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

Review: Association

- Association allows us to create a “knows about” relationships between different classes
- In association, one instance of a class knows about an instance of another *peer class* and can call methods on it
- Association is a consequences of delegating responsibilities to other classes
  - they are design choices, not Java constructs, and require no new syntax

Outline

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

Using What You Know

- Problem Statement:
  - Chloe and Karim are racing from their dorms to the 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 {

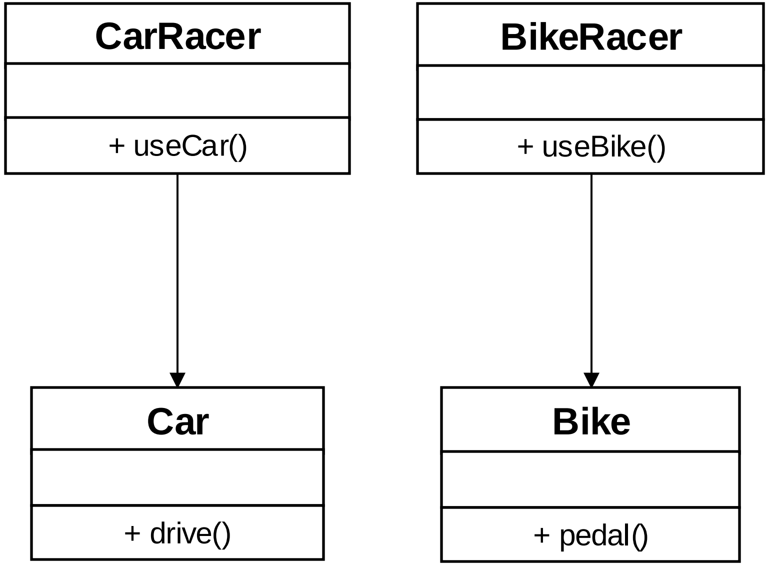
    public Car() { // constructor
        // code elided
    }
    public void drive() {
        // code elided
    }
    // more methods elided
}
```

```
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 Chloe and Karim to use their type of transportation
  - `CarRacer`
  - `BikeRacer`
- What methods will we need? What capabilities should each `-Racer` class have?
- `CarRacer` needs to know how to use the car
  - write `useCar()` method: uses `drive()`, shields caller from knowing what all `useCar()` might need to do
- `BikeRacer` needs to know how to use the bike
  - write `useBike()` method: uses `pedal()`, shields caller from knowing what all `useBike()` might need to do



Coding the project (2/4)

- Let's build the racer classes

```
public class CarRacer {
    private Car car;

    public CarRacer() {
        this.car = new Car();
    }

    public void useCar() {
        this.car.drive();
        // other methods as needed
    }
    // more methods elided
}
```

```
public class BikeRacer {
    private Bike bike;

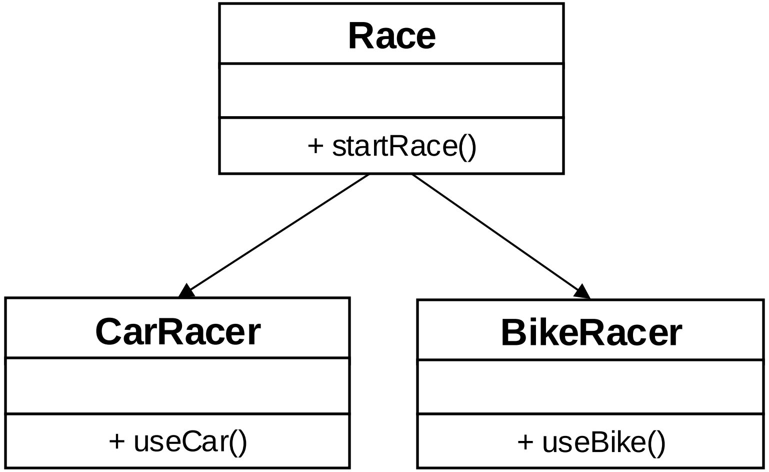
    public BikeRacer() {
        this.bike = new Bike();
    }

    public void useBike() {
        this.bike.pedal();
        // other methods as needed
    }
    // more methods elided
}
```

Goal 2: Tell racers to start the race

- Race class is composed of two Racers
  - App instantiates Race
  - Race is the top-level logic class
- Race class will have startRace() method
  - startRace() tells each Racer to use their transportation
- startRace() gets called in App

startRace:
  Tell this.chloe to useCar
  Tell this.karim 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() {
        this.car = new Car();
    }

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

// BikeRacer class elided
```

Old code

```
public class Race {
    private CarRacer chloe;
    private BikeRacer karim;

    public Race() {
        this.chloe = new CarRacer();
        this.karim = new BikeRacer();
    }

    public void startRace() {
        this.chloe.useCar();
        this.karim.useBike();
    }
}
```

But how does a Race get created and how does startRace() get called?

Coding the project (4/4)

```
public class App {

    public static void main(String[] args) {
        Race cs15Race = new Race();
        cs15Race.startRace();
    }

}
```

```
// from the Race class on slide 11

public void startRace() {
    this.chloe.useCar();
    this.karim.useBike();
}
```

- Now build the App class
- Program starts with `main()`
- `main()` calls `startRace()` on `cs15Race`
  - Could call `startRace()` in Race's constructor, however flow of control is more clear starting race in App class

The Program

```
public class App {
    public static void main(String[] args) {
        Race cs15Race = new Race();
        cs15Race.startRace();
    }
}
```

```
public class Race {
    private CarRacer chloe;
    private BikeRacer karim;

    public Race() {
        this.chloe = new CarRacer();
        this.karim = new BikeRacer();
    }

    public void startRace() {
        this.chloe.useCar();
        this.karim.useBike();
    }
}
```

```
public class CarRacer {
    private Car car;

    public CarRacer() {
        this.car = new Car();
    }

    public void useCar() {
        this.car.drive();
    }

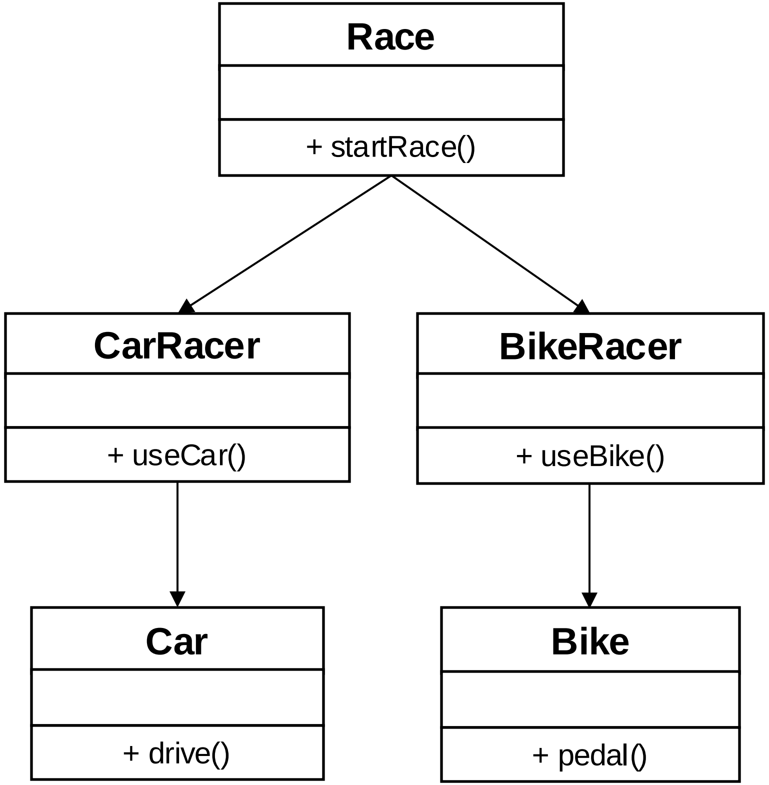
}
```

```
public class BikeRacer {
    private Bike bike;

    public BikeRacer() {
        this.bike = new Bike();
    }

    public void useBike() {
        this.bike.pedal();
    }
}
```

What does our design look like?



- Java initializes an instance of `App`, calling `main`
  - `App` omitted from class diagram by convention
- `main` initializes an instance of `Race`
- `Race`'s constructor initializes `chloe`, a `CarRacer` and `karim`, a `BikeRacer`
  - `CarRacer`'s constructor initializes `car`, a `Car`
  - `BikeRacer`'s constructor initializes `bike`, a `Bike`

Flow of control (1/2)

```
public class App {

    public static void main(String[] args) {
        Race cs15Race = new Race();
        cs15Race.startRace();
    }
}
```

```
public class Race {
    private CarRacer chloe;
    private BikeRacer karim;

    public Race() {
        this.chloe = new CarRacer();
        this.karim = new BikeRacer();
    }

    public void startRace() {
        this.chloe.useCar();
        this.karim.useBike();
    }
}
```

```
public class CarRacer {
    // constructor elided, creates car

    public void useCar() {
        this.car.drive();
    }
}
```

```
public class BikeRacer {
    // constructor elided, creates bike

    public void useBike() {
        this.bike.pedal();
    }
}
```

- `main` initializes an instance of `Race`
- `Race`'s constructor initializes `chloe`, a `CarRacer` and `karim`, a `BikeRacer`
    - `CarRacer`'s constructor initializes `car`, a `Car`
    - `BikeRacer`'s constructor initializes `bike`, a `Bike`

Flow of control (2/2)

```
public class App {

    public static void main(String[] args) {
        Race cs15Race = new Race();
        cs15Race.startRace();
    }
}
```

```
public class Race {
    // constructor elided; creates chloe and karim

    public void startRace() {
        this.chloe.useCar();
        this.karim.useBike();
    }
}
```

```
public class CarRacer {
    // constructor elided, creates car

    public void useCar() {
        this.car.drive();
    }
}
```

```
public class BikeRacer {
    // constructor elided, creates bike

    public void useBike() {
        this.bike.pedal();
    }
}
```

- After `Race` constructs `chloe` and `karim`, `App` calls `cs15Race.startRace()`
- `chloe` calls `useCar()` and `karim` calls `useBike()`
- `useCar()` calls `this.car.drive()`
- `useBike()` calls `this.bike.pedal()`

Can we do better?

Things to think about

- Do we need two different `Racer` classes?
  - we want multiple instances of `Racer`s 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?
  - how can we simplify?

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

- Create one Racer class
  - define different use methods for each type of transportation
- chloe would be an instance of Racer and in startRace we would call:

```
this.chloe.useCar(new Car());
```

  - Car's drive() method will be invoked
- Good: only one Racer class
- But: Racer has to aggregate a use…() 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();
    }
}
```

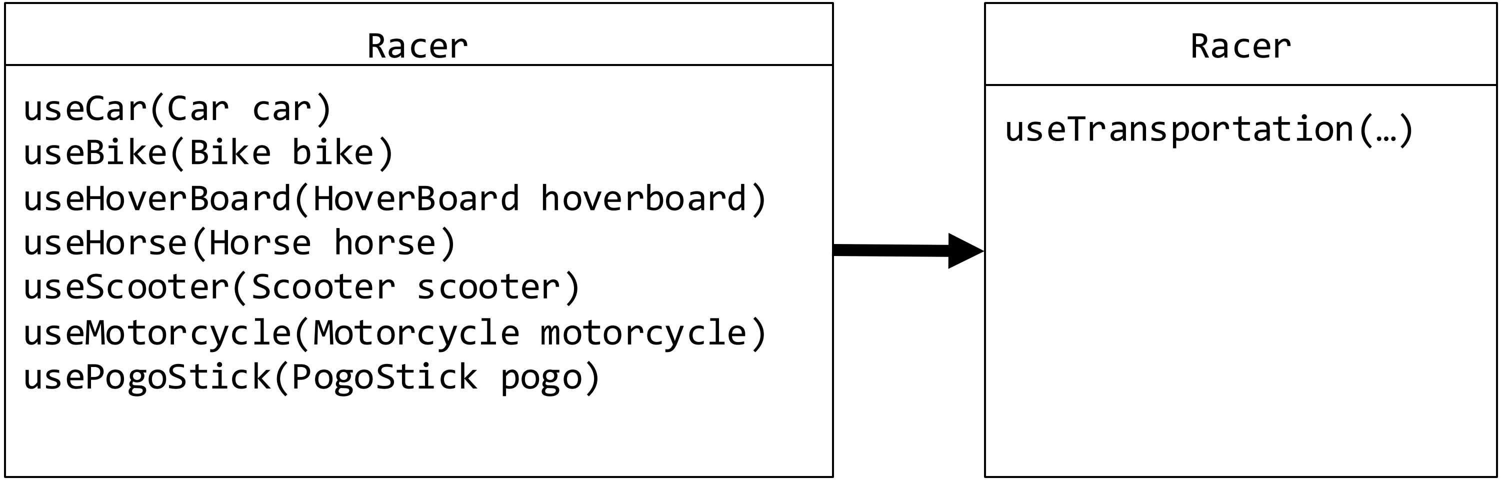
Solution 1 Drawbacks

- 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?

Outline

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

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 Racer {

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

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

```
public class Car implements Transporter {

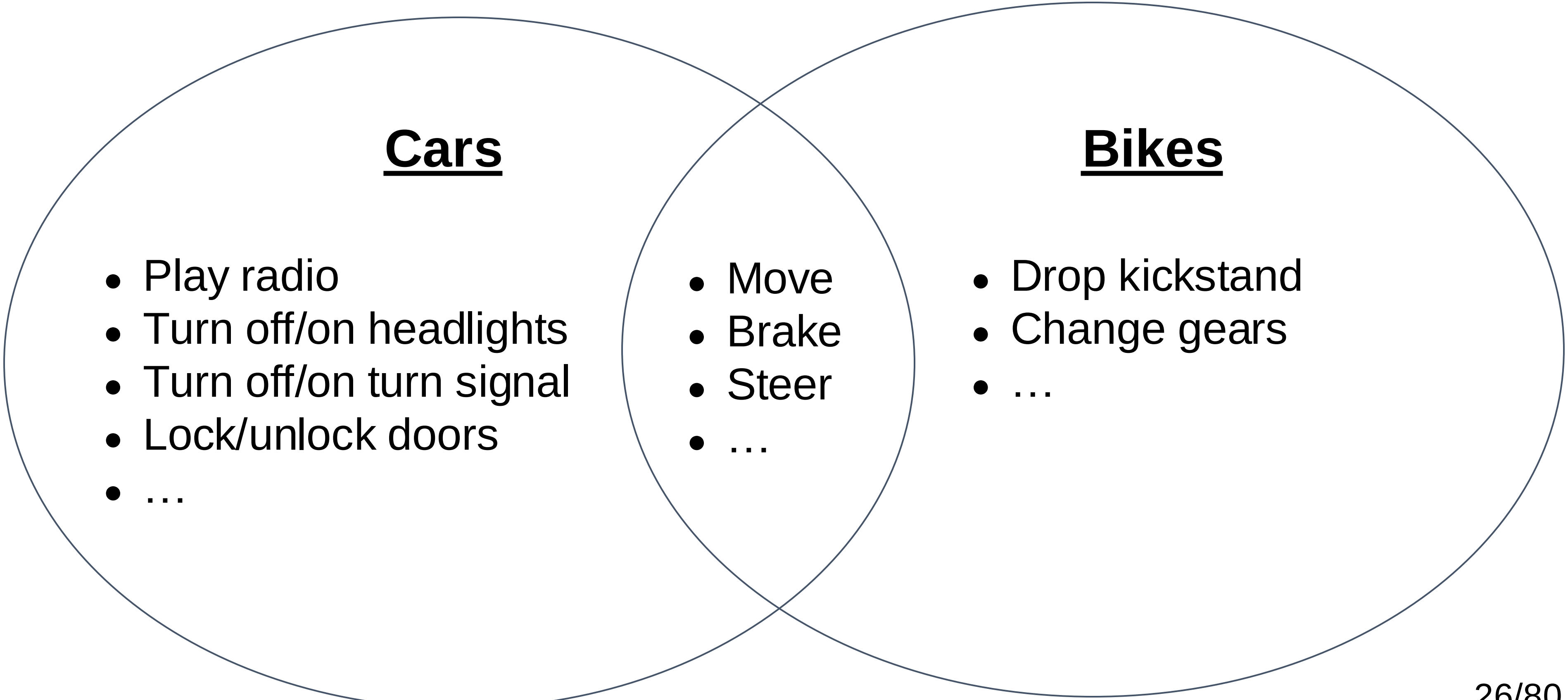
    public Car() {
        // code elided
    }
    public void drive(){
        // code elided
    }

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

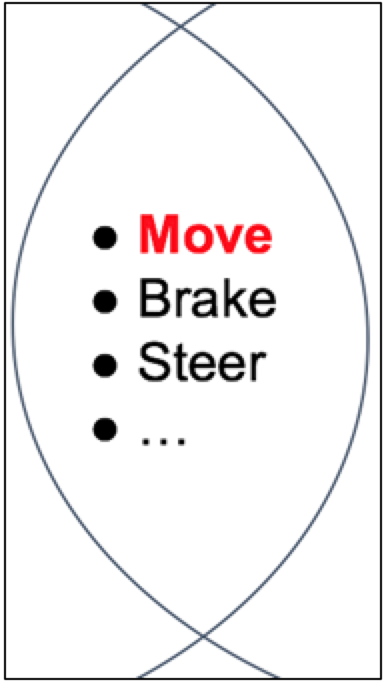
Interfaces: Spot the Similarities

- 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
- We prefer the more general move to the previous `useCar()`, `useBike()`

Can we model this in code?

- Many real-world objects have several broad functional similarities
  - cars and bikes can move
  - cars and laptops can play radio
  - phones and Teslas can be charged
- Take `Car` and `Bike` classes
  - how can their similar functionalities get enumerated in one place?
  - how can their broad relationship get modeled through code?
- Note: cars and bikes serve a similar purpose while phones and Teslas don't – we only care that they share *some similar functionality* (but potentially quite different implementations)

Car

- `move()`
- `brake()`
- `steer()`

- `playRadio()`
- `lockDoors()`
- `unlockDoors()`

Bike

- `move()`
- `brake()`
- `steer()`

- `dropKickstand()`
- `changeGears()`

Abbreviated class diagram

Introducing Interfaces (1/2)

- `Interface` groups declarations of similar capabilities of different classes together

- Looks like a totally stripped-down class declaration, with just method declarations:

- ```
  public interface Transporter {
      public void move();
      // other common methods (brake, steer…)
  }
  ```

- `Car`s and `Bike`s can "implement" a `Transporter` interface
  - they can transport people from one place to another
  - they "act as" transporters
    - can move (and brake, steer…)
  - for this lecture, interfaces are green and classes that implement them are pink

Car

- `move()`
- `brake()`
- `steer()`

- `playRadio()`
- `lockDoors()`
- `unlockDoors()`

Bike

- `move()`
- `brake()`
- `steer()`

- `dropKickstand()`
- `changeGears()`
  ```

Introducing Interfaces (2/2)

- Interfaces are contracts that classes agree to
- If a class chooses to implement a given interface, it must define all methods declared in interface
    - if a class doesn't implement one of interface's methods, the compiler "raises errors"
        - later we'll discuss strong motivations for this "contract enforcement"
- Interfaces only declare, don't define their methods – classes that implement the interfaces provide definitions/implementations
    - interfaces only care every class that implements the interface must define the methods declared in the interface – not how they are defined
- Interfaces model similarities while ensuring consistency
    - what does this mean?

Models Similarities while Ensuring Consistency (1/3)

Let’s break that down into two parts:

1) Model Similarities

2) Ensure Consistency

Models Similarities while Ensuring Consistency (2/3)

- 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!
- Make the `Transporter` interface
  - what methods should the `Transporter` interface declare? Similarities!
    - `move()` (plus `brake()`, `steer()`...)
  - compiler ensures consistency--doesn't care how method is defined, just that it has been defined
  - general tip: methods that interface declares should model functionality that **all** implementing classes share

Declaring an Interface (1/3)

```
public interface Transporter {

    public void move();
    //other methods

}
```

- Declare it as interface rather than class
- Declare methods – the contract
- In this case, we show only one required method: `move()`
- All classes that sign contract (implement this interface) must define actual implementation of any declared methods

Declaring an Interface (2/3)

```
public interface Transporter {

    public void move();
    //other methods

}
```

- 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 – no code!

Declaring an Interface (3/3)

```
public interface Transporter {

    public void move();
    //other methods

}
```

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

Outline

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

Implementing an Interface (1/6)

```
public class Car implements
Transporter {

    public Car() {
        // constructor
    }

    public void drive() {
        // code for driving 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 declared 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 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 parameters) and return type *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 simple or complex 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 be able 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 for each method
    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 (3/3)

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

TopHat Question

Join Code: 316062

Can you instantiate an interface as you can a class?

A. Yes

B. No

TopHat Question

Join Code: 316062

Can an interface define code for its methods?

A. Yes

B. No

TopHat Question

Join Code: 316062

Which color-coded segment of this program is **incorrect**?

```
A. public interface Colorable {
        public Color getColor() {
B.          return Color.PINK;
        }
   }

C. public class Rectangle implements Colorable {
        // constructor elided
D.      @Override
        public Color getColor() {
E.          return Color.RED;
        }
   }
```

TopHat Question

Join Code: 316062

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

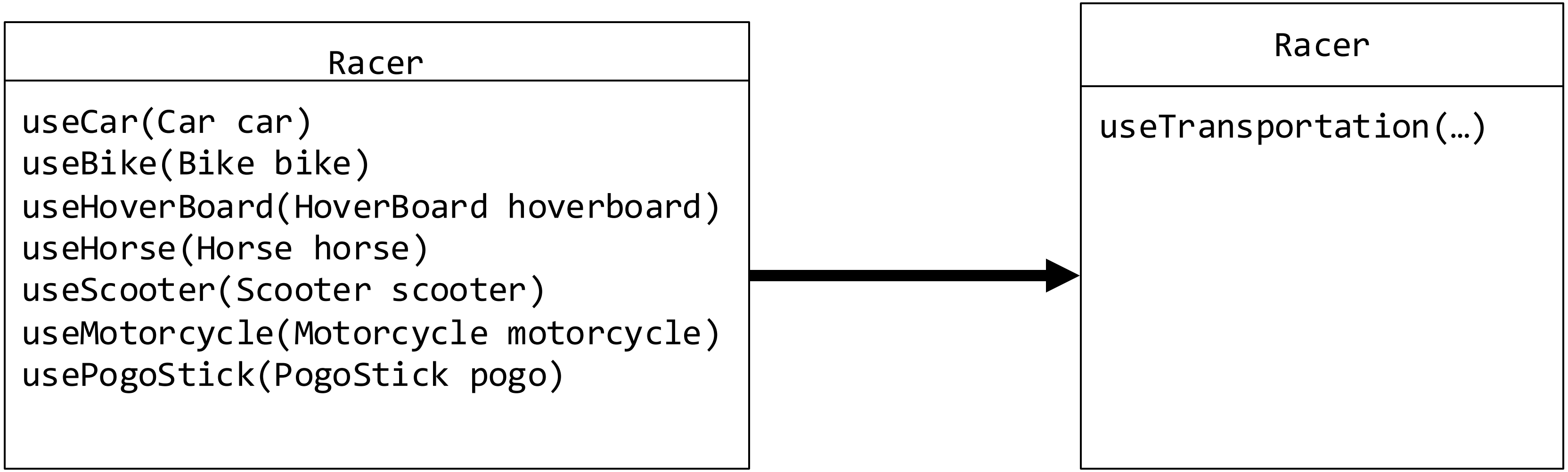
- Let's make transportation classes use an interface

```java
public class Car implements Transporter {

    public Car() {
        // code elided
    }
    public void drive() {
        // code elided
    }

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

    // more methods elided
}
```

```java
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?



Outline

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

Introducing Polymorphism

- Poly = many, morph = forms
- A way of coding generically
  - way of referencing multiple classes sharing abstract functionality as acting as one generic type
    - cars and bikes can both `move()` → refer to them as classes of type `Transporter`
    - phones and Teslas can both `getCharged()` → refer to them as class of type `Chargeable`, i.e., classes that implement `Chargeable` interface
    - cars and boomboxes can both `playRadio()` → refer to them as class of type `RadioPlayer`
- 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` instance passed in could be instance of `Car`, `Bike`, etc., i.e., of 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

what’s the `actual vs. declared type` of any transportation instance passed in?

```
public class Racer {

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

}
```

which `move()` is executed?

Actual vs. Declared Type (1/2)

- We first show polymorphic assignment (typically not useful by itself) and then polymorphic parameter passing

- Consider following polymorphic assignment statement:

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

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

Actual vs. Declared Type (2/2)

- Can treat `Car`/`Bike` instances as instances of type `Transporter`
- `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 instances of type `Transporter` AND the methods are declared in that interface

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

- If `Car` defines `playRadio()` method, is this correct?
`transportation.playRadio()`

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

Is this legal?

```
Transporter karimsBike = new Bike();
```
✓

```
Transporter chloesCar = new Car();
```
✓

```
Transporter chloesRadio = new Radio();
```
✗

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

Only Declared Type's Methods Can be Used

- What methods must `Car` and `Bike` have in common?
  - `move()`
- How do we know that?
  - they implement `Transporter`
    - guarantees that they have `move()`, plus whatever else is appropriate to that class
- Think of `Transporter` like the "lowest common denominator"
  - it's what all classes of type `Transporter` will have in common
  - only `move()` may be called if an instance is passed as the declared interface type

```
class Bike implements Transporter{

    public void move();
    public void dropKickstand();
    // etc.
}
```

```
class Car implements Transporter{

    public void move();
    public void playRadio();
    // etc.
}
```

Motivations for Polymorphism

- Many different kinds of transportation but only care about their shared capability
  - i.e., how they move
- Polymorphism lets 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 class 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?

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

```
Transporter karimsBike = new Bike();
this.karim.useTransportation(karimsBike);
```

✓

```
Car chloesCar = new Car();
this.chloe.useTransportation(chloesCar);
```

✓

```
Radio chloesRadio = new Radio();
this.chloe.useTransportation(chloesRadio);
```

✗

Even though `chloesCar` is declared as a `Car`, not a `Transporter`, the compiler can still verify that `Car` implements `Transporter`

A `Radio` wouldn't implement `Transporter`. Therefore, `useTransportation()` cannot treat it as a type of `Transporter`

Let's look at move() (1/2)

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

```
public class Racer {

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

}
```

- Since the only method declared in Transporter is move(), all we will *ever* ask objects of type Transporter to do is move()

Let's look at `move()` (2/2)

- Only have access to instance of type `Transporter`
  - 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 section of code in Race class:

```
Transporter karimsBike = new Bike();
this.karim.useTransportation(karimsBike);
```

- Remember what useTransportation() method looks like:

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

What is "actual type" of transportation in this.karim.useTransportation(karimsBike); ?

Method Resolution (1/4)

```
public class Race {

    private Racer karim;
    // previous code elided

    public void startRace() {
        Transporter karimsBike = new Bike();
        this.karim.useTransportation(karimsBike);
    }
}
```

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

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

- `Bike` is actual type
  - `karim` was passed an instance of `Bike` as the argument
- `Transporter` is declared type
  - `Bike` instance is treated as type of `Transporter`
- So… what happens in `transportation.move()`?
  - What `move()` method gets used?

Method Resolution (2/4)

```
public class Race {
    // previous code elided
    public void startRace() {
        Transporter karimsBike = new Bike();
        this.karim.useTransportation(karimsBike);

    }
}
```

```
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();
    }
}
```

- `karim` is a `Racer`
- `Bike`'s `move()` method gets used
- Why?
  - `Bike` is the actual type of this `Transporter`
    - 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)

```
public class Race {
    // previous code elided
    public void startRace() {
        Transporter karimsCar = new Car();
        this.karim.useTransportation(karimsCar);

    }
}
```

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

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

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

Method Resolution (4/4)

- `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
  - this method resolution is an example of dynamic binding, which directly contrasts the normal static binding, in which method gets resolved at compile time

TopHat Question

Join Code: 316062

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 polymorphism work when calling methods?

- Declared type and actual type work together
  - declared type keeps things generic
    - can reference many classes using one generic type
  - actual type ensures specificity
    - when calling declared type's method on an instance, the *actual* code that is called is the code defined in the *actual* type's class (dynamic binding)

Bender interface declares the bend() method for all Benders

Declared

Actual

Katara, an instance of WaterBender, defines bend() to water bend

When to use polymorphism?

- Do you use only functionality declared in interface OR do you need 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, not every `Car` can `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 capabilities exist, don't care how they're implemented
    - allows for more generic programming
      - `useTransportation()` can take in any instance of type `Transporter`
      - can easily extend this program to use any form of transportation, with minimal changes to existing code
    - a tool for extensible programming
      - how?

Why is this important?

- Using more than 2 methods of transportation?
- Old Design:
  - need more classes → more specialized methods (`useCar()`, `useBike()`, `useRollerblades()`, 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"

The Program

```java
public class App {
    public static void main(String[] args) {
        Race cs15Race = new Race();
        cs15Race.startRace();
    }
}
```

```java
public class Race {
    private Racer chloe, karim;

    public Race() {
        this.chloe = new Racer();
        this.karim = new Racer();
    }

    public void startRace() {
        Transporter chloesCar = new Car();
        this.chloe.useTransportation(chloesCar);
        Transporter karimsBike = new Bike();
        this.karim.useTransportation(karimsBike);
    }
}
```

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

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

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

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

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

Flow of Control (1/2)

```
public class App {
    public static void main(String[] args) {
        Race cs15Race = new Race();
        cs15Race.startRace();
    }
}
```

```
public class Race {
    private Racer chloe, karim;

    public Race() {
        this.chloe = new Racer();
        this.karim = new Racer();
    }

    public void startRace() {
        Transporter chloesCar = new Car();
        this.chloe.useTransportation(chloesCar);
        Transporter karimsBike = new Bike();
        this.karim.useTransportation(karimsBike);
    }
}
```

How would this program run?

- Program begins with `main` method of `App` class
- `main` method initializes `cs15Race`, an instance of `Race`
- `Race`'s constructor initializes `chloe`, a `Racer`, and `karim`, a `Racer`
- `main` method calls `cs15Race.startRace()`
- `startRace()` calls:

```
Transporter chloesCar = new Car();
this.chloe.useTransportation(chloesCar);
Transporter karimsBike = new Bike();
this.karim.useTransportation(karimsBike);
```

Flow of Control (2/2)

- `useTransportation(chloesCar)` calls `Car`'s `move()` method which calls `this.drive()`
- `useTransportation(karimsBike)` calls `Bike`'s `move()` method which calls `this.pedal()`

```
    public void startRace() {
        Transporter chloesCar = new Car();
        this.chloe.useTransportation(chloesCar);
        Transporter karimsBike = new Bike();
        this.karim.useTransportation(karimsBike);
    }
}
```

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

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

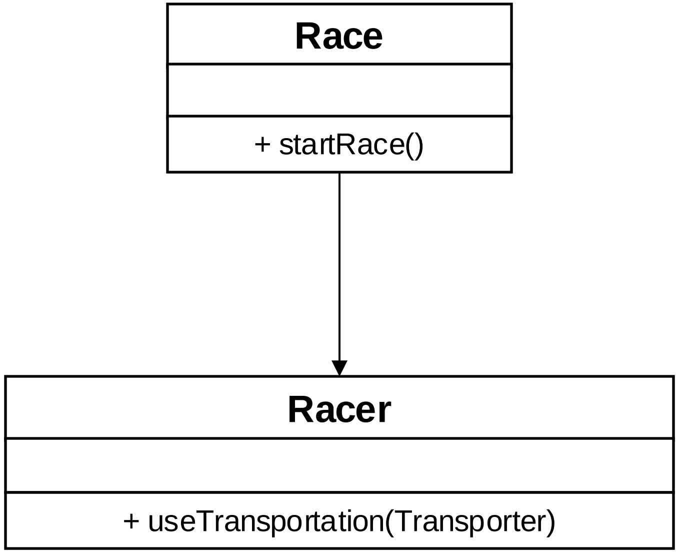
karimsBike

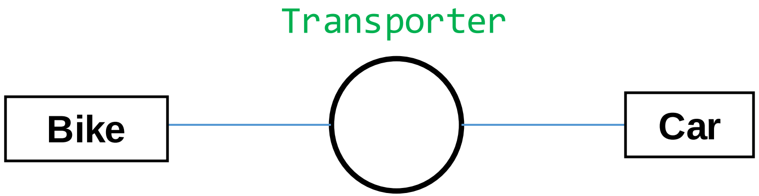
```
public interface Transporter {
    public void move();
}
```

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

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

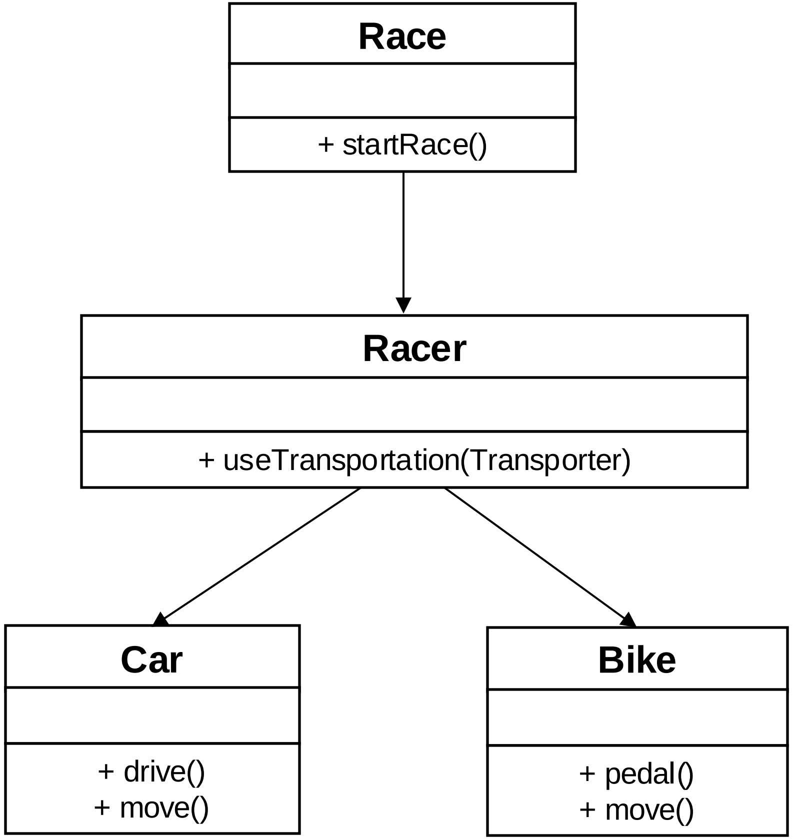
What does our new design look like? (1/2)





- main method instantiates cs15Race, an instance of Race
    - App omitted from class diagram
- Race's constructor initializes two Racers – karim and chloe
    - Race is composed of two Racers
- startRace() instantiates chloesCar and karimsBike
    - these are local variables, and do not exist on the class diagram
- In interface diagram, can represent relationship between our vehicles and Transporter

What does our new design look like? (2/2)



- In a larger version of this program, we may want each `Racer` to send more messages to their `Transporter`
  - we could store an instance variable of declared type `Transporter`
- Now, `Car` and `Bike` are peer objects of the `Racer` class

Modified Program

```java
public class App {
    public static void main(String[] args) {
        Race cs15Race = new Race();
        cs15Race.startRace();
    }
}
```

```java
public class Race {
    private Racer chloe, karim;

    public Race() {
        Transporter chloesCar = new Car();
        this.chloe = new Racer(chloesCar);
        Transporter karimsBike = new Bike();
        this.karim = new Racer(karimsBike);
    }
    public void startRace() {
        this.chloe.useTransportation();
        this.karim.useTransportation();
    }
}
```

```java
public interface Transporter {
    public void move();
    // other methods of Transporters elided
}
```

```java
public class Racer {

    private Transporter transporter;

    public Racer(Transporter myTransporter) {
        this.transporter = myTransporter;
}

    public void useTransportation() {
        this.transporter.move();
    }

    public void returnVehicle() {
        // code elided - will call a method on
           transporter here
    }

}
```

```java
public class Car implements Transporter {
    // omitted, same as before
}
```

```java
public class Bike implements Transporter {
    // omitted, same as before
}
```

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?
  - you want to be the laziest (but cleanest) programmer you can be

Announcements

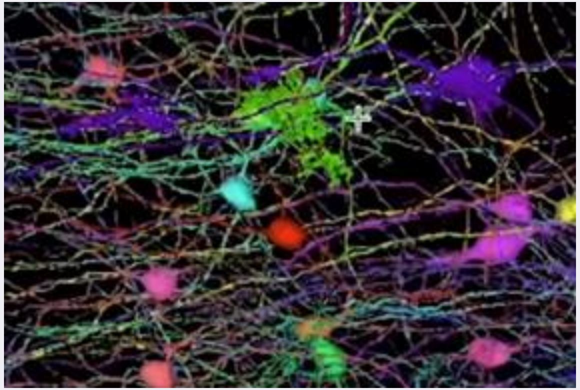
- TicTacToe released today (9/24)
  - Early hand-in: 9/26
  - On-time hand in: 9/28
  - Late hand-in: 9/30
- Class Relationships Section
  - Mini Assignment due before section
  - Fill out the form linked at the bottom of handout for credit
- CS15 Mentorship
  - Officially begun!
- T-Shirt Contest!!!!!
  - Designs due **next Tuesday before Lecture**!! (looking at you RISD students :D)

AI II: Intro to Neural Networks

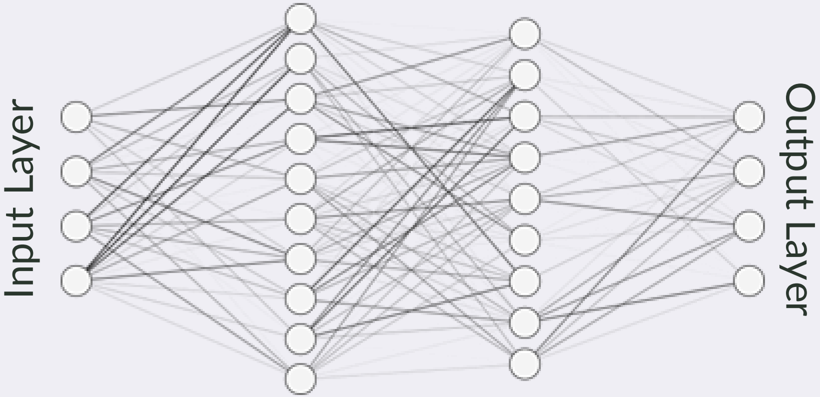
Topics in Socially Responsible Computing

What is a Neural Network?

- A type of computer system inspired by the human brain
- Made up of layers of interconnected nodes called neurons
- Successive layers allow for recognition of more complex features
- Learn and make decisions by recognizing patterns in data



Source: Facebook/Cleo Abram



Source: Wavefrontshaping.net

Image Classification

- Neural networks are great for image recognition…
- *Example: handwritten digit classification!*
- Can our neural network identify which number is represented in the images of these handwritten digits…?

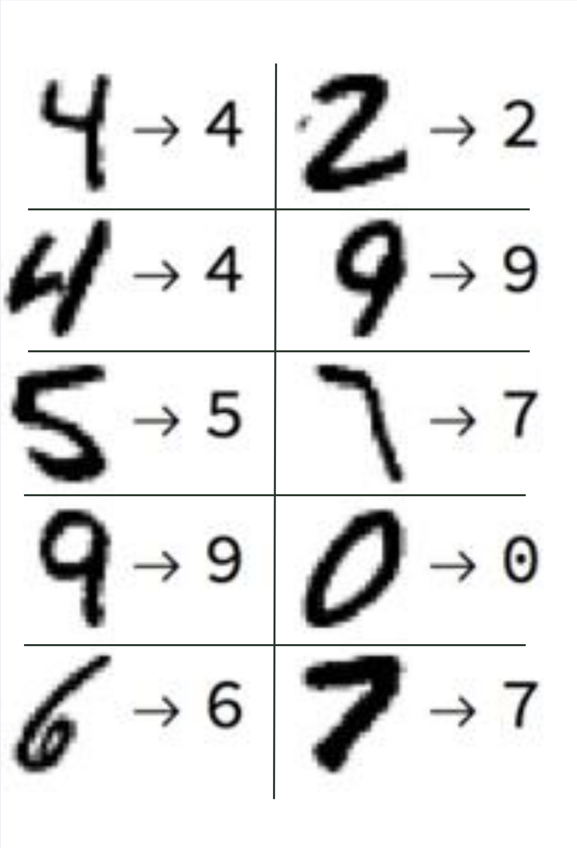
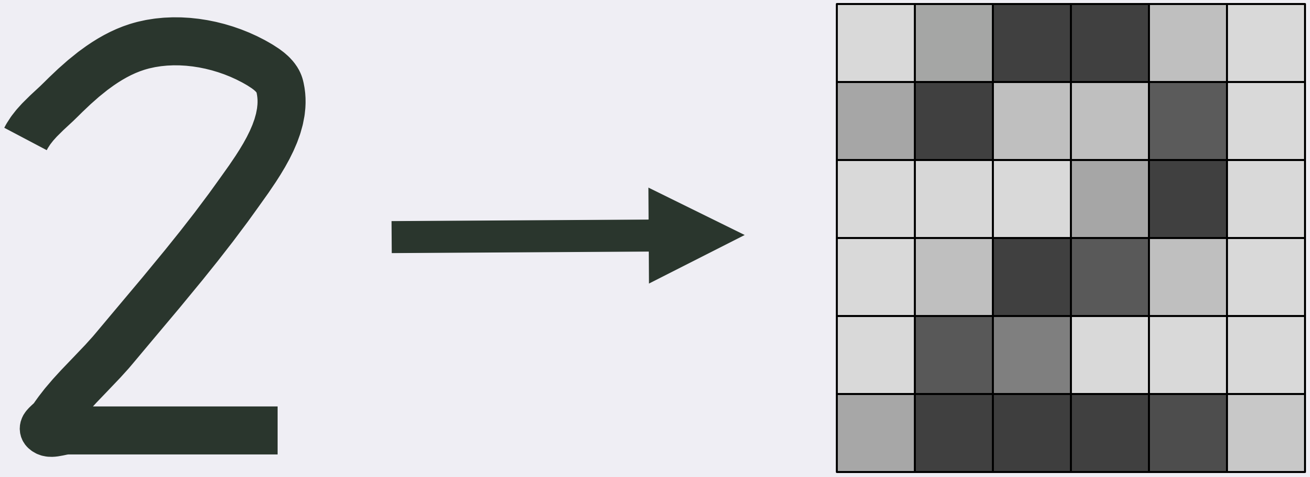


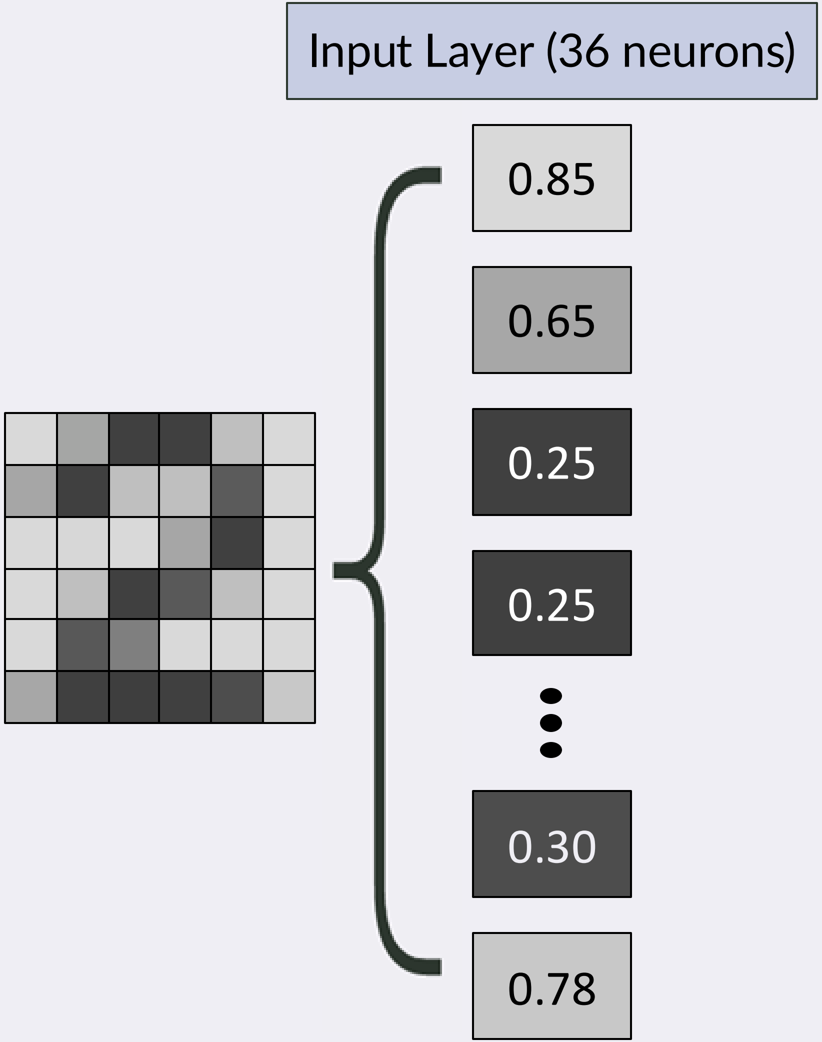
Image Source: Wolfram

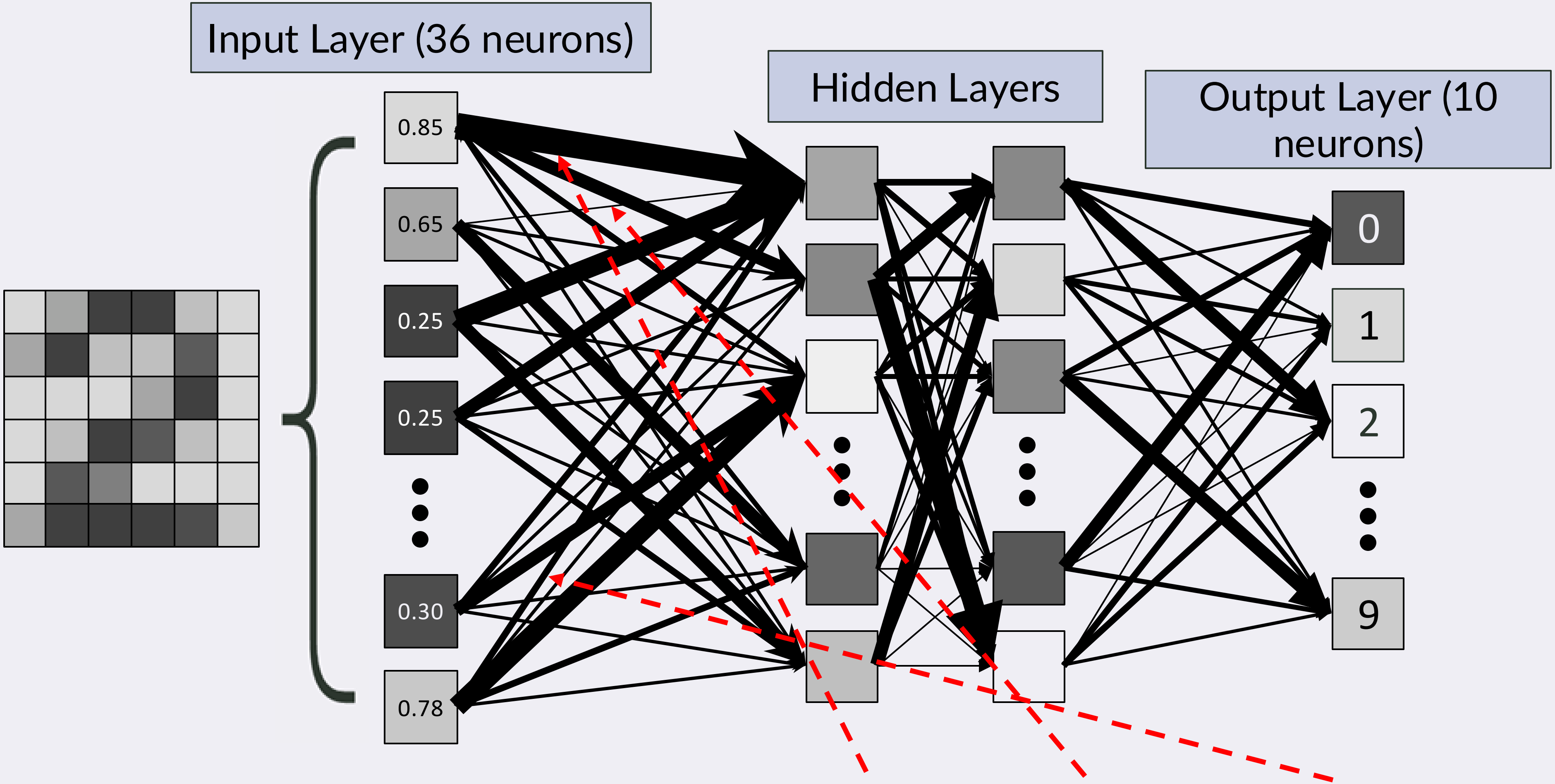
How do we represent an image?



What is a neuron?

- A **neuron** holds a number in a neural network, representing information like pixel values or processed data
- The **input layer** has neurons that store pixel grayscale values (0 = black, 1 = white, with shades of gray in between)
- In this example, the input layer has **36 neurons** (for a 6x6 grid)





activation value = φ ($w_0 i_0 + w_1 i_1 + \ldots + w_{35} i_{35}$) = φ ((0.90)(0.85) + (0.1)(0.65) + … (0.45)(0.78))

How does the network learn? (supervised learning)

Training process:

1. Each piece of training data is labeled with the correct output
2. Input data processed through the

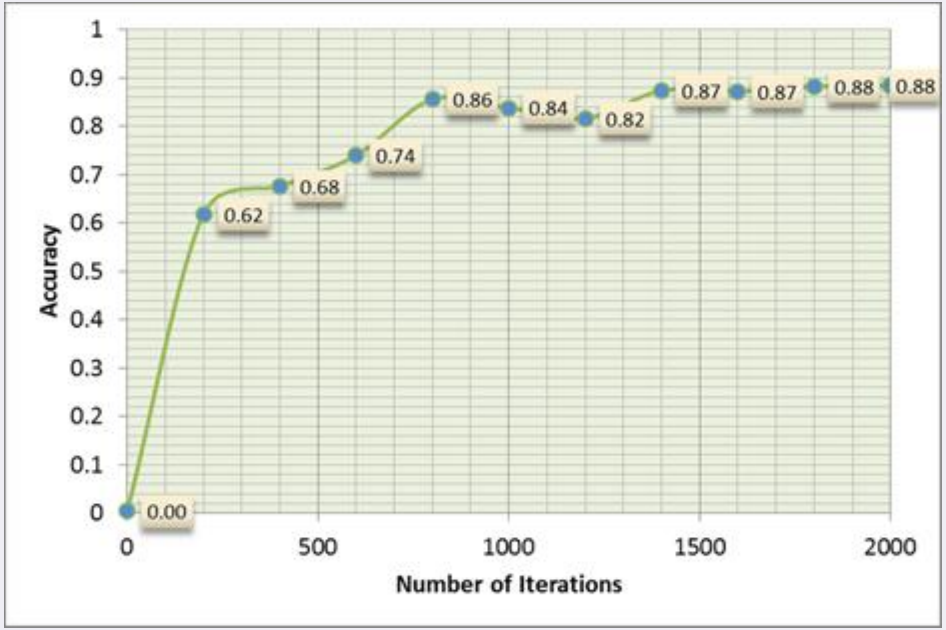


Image Source: Abhishek Maity

For more in-depth information on how neural networks work...

Video series that goes in-depth on how neural networks work: https://www.3blue1brown.com/topics/neural-networks

Article that explains neural networks and details the history of deep learning: https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414