Health & Wellness

Come chat if you...

- want help getting accommodations or support
- want to find resources on staying well @ Brown CS
- are feeling overwhelmed (or want to avoid overwhelm!)
- have feedback on accessibility @ Brown CS
- just want to say hi!

Calendar: [https://browncs-health-and-wellness.github.io/](https://browncs-health-and-wellness.github.io/)
(sign up on SignMeUp – CSHEALTH queue!)

Email: wellness.advocates@lists.cs.brown.edu

Shira’s Hours: Mon 10-11am, Weds 5-6pm, Thurs 4-5pm (right after lecture!)
Lecture 5
Interfaces and Polymorphism
Outline

- Transportation Example
- Intro to Interfaces
- Implementing Interfaces
- Polymorphism
Review: Declaring vs. Defining Methods

- like a dictionary entry: term, followed by definition (the “body”)

- method **declaration** has the scope (**public**), return type (**void**), method name (**makeSounds**), and parameters

- method **definition** is the body of the method – the actual implementation (the code that actually makes the sounds)

```
public class Dog {
    // class declaration
    // properties elided
    // constructor elided

    public void makeSounds() {
        this.bark();
        this.whine();
        this.bark();
    }

    public void bark() {
        // code elided
    }

    public void whine() {
        // code elided
    }
}
```

**Insurrection:**

an act or instance of rising in revolt, rebellion, or resistance against civil authority or an established government
Using What You Know

● Imagine this program:
  ○ Marina and Anna 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

```java
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 Marina and Anna 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

- **BikeRacer** needs to know how to use the bike
  - write `useBike()` method
Coding the project (2/4)

- Let's build the racer classes

```java
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 Racer
  - App contains Race

- Race class will have startRace() method
  - startRace() tells each Racer to use their transportation

- startRace() gets called in App

```kotlin
startRace:
  Tell _anna to useCar
  Tell _marina to useBike
```
Coding the project (3/4)

- Given our **CarRacer** class, let’s build the **Race** class

```java
public class Race {
    private CarRacer _anna;
    private BikeRacer _marina;

    public Race() {
        _anna = new CarRacer();
        _marina = new BikeRacer();
    }

    public void startRace() {
        _anna.useCar();
        _marina.useBike();
    }
}
```

**Old code**

```java
public class CarRacer {
    private Car _car;

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

    public void useCar() {
        _car.drive();
    }

    // more methods elided
}
```

// BikeRacer class elided
Coding the project (4/4)

public class App {

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

}

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

//from the Race class on slide 11

public void startRace() {
    _anna.useCar();
    _marina.useBike();
}
What does our design look like?

How would this program run?

- Java initializes an instance of `App`, calling `main`
- `main` initializes an instance of `Race`
- `Race`’s constructor initializes `_anna`, a `CarRacer` and `_marina`, a `BikeRacer`
  - `CarRacer`’s constructor initializes `_car`, a `Car`
  - `BikeRacer`’s constructor initializes `_bike`, a `Bike`
Full flow of control

- Java initializes an instance of `App`, calling `main`
- `main` initializes an instance of `Race`
- `Race`'s constructor initializes `_anna`, a `CarRacer` and `_marina`, a `BikeRacer`
  - `CarRacer`'s constructor initializes `_car`, a `Car`
  - `BikeRacer`'s constructor initializes `_bike`, a `Bike`

- `App` calls `cs15Race.startRace()`
- `cs15Race` calls `_anna.useCar()` and `_marina.useBike()`
- `_anna` calls `_car.drive()`
- `_marina` calls `_bike.pedal()`
Can we do better?
Things to think about

● Do we need two different Racer classes?
  o we want multiple instances of Racer\textregistered 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?
  o 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
- `_anna` is instance of `Racer` and elsewhere we have:
  - `_anna.useCar(new Car());`
  - Car’s `drive()` method will be invoked
- But `Racer` will need a new method to accommodate every kind of transportation!

```java
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

```java
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?

```java
Racer/useCar(Car car)
useBike(Bike bike)
useHoverBoard(HoverBoard hoverboard)
useHorse(Horse horse)
useScooter(Scooter scooter)
useMotorcycle(Motorcycle motorcycle)
usePogoStick(PogoStick pogo)
```

```java
Racer/useTransportation(…)
```
Interfaces and Polymorphism

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

```java
public class Car implements Transporter {
    public Car() {
        //code elided
    }
    public void drive() {
        this.drive();
    }
    //more methods elided
}
public class Racer {
    //previous code elided
    public void useTransportation(Transporter transport) {
        transport.move();
    }
}
public interface Transporter {
    public void move();
}
```

● In order to simplify code, we need to learn:

- Interfaces
- Polymorphism
- we’ll see how this new code works shortly:
Interfaces: Spot the Similarities

- What do cars and bikes have in common?
- What do cars and bikes *not* have in common?
Cars vs. Bikes

**Cars**
- Play radio
- Turn off/on headlights
- Turn off/on turn signal
- Lock/unlock doors
- ...

**Bikes**
- Move
- Brake
- Steer
- ...
- Drop kickstand
- Change gears
- ...
- ...
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** classes
  - how can their similar functionalities get enumerated in one place?
  - how can their broad relationship get portrayed through code?

<table>
<thead>
<tr>
<th>Car</th>
<th>Bike</th>
</tr>
</thead>
<tbody>
<tr>
<td>move()</td>
<td>move()</td>
</tr>
<tr>
<td>brake()</td>
<td>brake()</td>
</tr>
<tr>
<td>steer()</td>
<td>steer()</td>
</tr>
<tr>
<td>playRadio()</td>
<td>dropKickstand()</td>
</tr>
<tr>
<td>lockDoors()</td>
<td>changeGears()</td>
</tr>
<tr>
<td>unlockDoors()</td>
<td></td>
</tr>
</tbody>
</table>
Introducing Interfaces (1/2)

- **Interfaces** group declarations of similar capabilities of different classes together

- Model “acts-as” relationship

- **Cars and Bikes** could implement a **Transporter** interface
  - they can transport people from one place to another
  - they “act as” transporters
    - can move
    - have other shared functionality, such as moving, braking, turning etc.
  - for this lecture, interfaces are **green** and classes that implement them are **pink**

<table>
<thead>
<tr>
<th>Car</th>
<th>Bike</th>
</tr>
</thead>
<tbody>
<tr>
<td>- move()</td>
<td>- move()</td>
</tr>
<tr>
<td>- brake()</td>
<td>- brake()</td>
</tr>
<tr>
<td>- steer()</td>
<td>- steer()</td>
</tr>
<tr>
<td>- playRadio()</td>
<td>- dropKickstand()</td>
</tr>
<tr>
<td>- lockDoors()</td>
<td>- changeGears()</td>
</tr>
<tr>
<td>- unlockDoors()</td>
<td></td>
</tr>
</tbody>
</table>
Introducing Interfaces (2/2)

- 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 only declare, don’t define their methods – classes that implement the interfaces provide definitions/implementations
  - 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 Cars and Bikes 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 (1/3)

What does this look like?

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

- Declare it as `interface` rather than class
- 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 (2/3)

What does this look like?

```java
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 (3/3)

What does this look like?

```java
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)

```java
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) *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
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
  o “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.”
  o what if I wanted Car to change color as well?
  o create a Colorable interface
  o add that interface to Car’s class declaration

● Class implementing interfaces must define every single method from each interface

```java
public interface Colorable {
    public void setColor(Color c);
    public Color getColor();
}
```

```java
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 instance of class that implements `Transporter` can `move()`

- Will know how 2 classes are related if both implement `Transporter`
Lecture Question

Which statement of this program is incorrect?

A. public interface Colorable {
   public Color getColor() {
   
   }
   return Color.WHITE;
   }

B. public class Rectangle implements Colorable {
   //constructor elided
   
   }

C. public class Rectangle implements Colorable {
   //constructor elided
   
   }

D. @Override
   public Color getColor() {
   return Color.PURPLE;
   }

E. 
   
   }
Lecture Question

Given the following interface:

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

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

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

D. ```java
   @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
}

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

Let's make transportation classes use an interface
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?

```
Racer
useCar(Car car)
useBike(Bike bike)
useHoverBoard(HoverBoard hoverboard)
useHorse(Horse horse)
useScooter(Scooter scooter)
useMotorcycle(Motorcycle motorcycle)
usePogoStick(PogoStick pogo)
```

```
Racer
useTransportation(...)
```
Introducing Polymorphism

- Poly = many, morph = forms
- A way of coding generically
  - way of referencing many related classes 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?
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

```java
public class Racer {
    //previous code elided
    public void useTransportation(Transporter transportation) {
        transportation.move();
    }
}
```
Actual vs. Declared Type (1/2)

- Consider following polymorphic assignment statement:

  \[
  \text{Transporter } \text{annasCar} = \text{new } \text{Car}();
  \]

- We say “\text{annasCar} is of type \text{Transporter},” but we instantiate a new \text{Car} and assign it to \text{annasCar}... is that legal?
  - doesn’t Java do “strict type checking”? (type on LHS = type on RHS)
  - how can instances of \text{Car} get stored in variable of type \text{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
  - 
  ```java
  Transporter transportation = new Car();
  transportation.playRadio();
  ```
- **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 are defined in that interface
  - 
  ```java
  Nope. The playRadio() method is not declared in Transporter interface, therefore compiler does not recognize it as a valid method call
  ```
- If **Car** defines **playRadio()** method, is this correct?
  - transportation.playRadio()
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 classes of type Transporter will have in common

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

```java
class Car implements Transporter {
    void move();
    void playRadio();
    //etc.
}
```
Is this legal?

Transporter marinasBike = new Bike(); ✔

Transporter marinasCar = new Car(); ✔

Transporter marinasRadio = new Radio(); ❌

Radio wouldn’t implement Transporter. Since Radio cannot “act as” type Transporter, you cannot treat it as an type Transporter
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?

```java
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?

Transporter marinasBike = new Bike();
_marina.useTransportation(marinasBike);

Car marinasCar = new Car();
_marina.useTransportation(marinasCar);

Radio marinasRadio = new Radio();
_marina.useTransportation(marinasRadio);

A Radio wouldn’t implement Transporter. Therefore, useTransportation() cannot treat it like as of type Transporter.

Even though marinasCar is declared as a Car, the compiler can still verify that it implements Transporter.
Why move()? (1/2)

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

```java
public class Racer {
    //previous code elided
    public void useTransportation(Transporter transportation) {
        transportation.move();
    }
}
```
Why 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 line of code in `Race` class:
  
  ```java
  _marina.useTransportation(new Bike());
  ```

- Remember what `useTransportation` method looked like:
  
  ```java
  public void useTransportation(Transporter transportation) {
    transportation.move();
  }
  ```

  What is “actual type” of `transportation` in
  
  ```java
  _marina.useTransportation(new Bike());
  ```
Method Resolution (1/4)

public class Racer {
    private Racer _marina;
    //previous code elided
    public void useTransportation() {
        _marina.useTransportation(new Bike());
    }
}

public class Race {
    private Racer _marina;
    //previous code elided
    public void startRace() {
        _marina.useTransportation(new Bike());
    }
}

- **Bike is actual type**
  - _marina was handed a new Bike() instance as argument

- **Transporter is declared type**
  - Bike instance is treated as of type Transporter

- So... what happens in transportation.move()?
  - What move() method gets used?
Method Resolution (2/4)

- _marina 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

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

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

```java
public class Race {
    //previous code elided
    public void startRace() {
        _marina.useTransportation(new Bike());
    }
}
```
Method Resolution (3/4)

- What if \texttt{marina} received an instance of \texttt{Car}?
  - What \texttt{move()} method would get called then?
    - \texttt{Car}'s!

```java
public class Race {
    //previous code elided

    public void startRace() {
        _marina.useTransportation(new Car());
    }
}
```
Method Resolution (4/4)

- This method resolution is an example of **dynamic binding**, which is when the actual method implementation used is not determined until runtime.
  - Contrast with **static binding**, in which method gets resolved at compile time.
- The `move()` method is bound dynamically – the compiler does not know which `move()` method to use until the program runs.
  - Same “`transport.move()`” line of code could be executed indefinitely number of times with different method resolution each time.
Given the following class:

```java
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? (1/2)

- **Declared type** and **actual type** work together
  - declared type keeps things generic
    - can reference a lot of classes using one generic type
  - actual type ensures specificity
    - when defining implementing class, methods can get defined without restriction

This is my instance of type `Transporter`!
Why does polymorphism work when calling methods? (2/2)

- **Declared type and actual type** work together
  - **declared type** keeps things generic
    - can reference a lot of classes using one generic type
  - **actual type** ensures specificity
    - when defining implementing class, methods can get defined without restriction

This is my instance of type **Transporter**!
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, 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
    - an extremely powerful tool for extensible programming
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”
What does our new design look like?

How would this program run?

- An instance of App gets initialized by `main`
- App’s constructor initializes `cs15Race`, an instance of Race
- Race’s constructor initializes `_anna`, a Racer and `_marina`, a Racer
- App calls `cs15Race.startRace()`
- `cs15Race` calls:
  - `_anna.useTransportation(new Car())`
  - `_marina.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

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

public class Race {
    private Racer _anna, _marina;

    public Race(){
        _anna = new Racer();
        _marina = new Racer();
    }

    public void startRace() {
        _anna.useTransportation(new Car());
        _marina.useTransportation(new Bike());
    }
}

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();
    }
}

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

● AndyBot due today (2/4) at 11:59pm
● Leap Frog will be released tomorrow (2/5)
  ○ Early hand-in: 2/8
  ○ On-time hand-in: 2/10
  ○ Late hand-in: 2/12
● If you haven’t been checked off for Lab0 email HTAs ASAP
● Conceptual Hours schedule on the course website
  ○ go there for questions related to lecture, Java concepts, or project concepts
● Review the TA Hours missive for more information
● Email section TAs before the first section of the week for swaps (before Tuesday evening)
IT in the News

ft. Socially Responsible Computing!
January 28, 2021: Facebook’s independent oversight board (its “Supreme Court”, formed in 2020) issues first post reviews
  ○ board designed to handle controversial cases, remove some responsibility from Zuckerberg et al.
    ■ comprised of law experts, former politicians, human rights lawyers, Nobel Peace prize winner…

Surprisingly, board overwhelmingly reinstates posts taken down by content moderators
  ○ posts include hate speech (and possible agitations to violence), Nazi quotations, COVID misinformation
  ○ but also nudity (as part of breast cancer awareness ad)

How does content moderation work?
  ○ recall: Section 230 of the CDA – platforms cannot be held legally liable for content posted or hosted
Aside: Facebook’s Content Moderation Process (2/3)

1. User or AI flags post for violating community standards or Terms of Service

2. Content moderator* reviews post based on FB’s Community Guidelines
   *15,000 FB Content Moderators worldwide – underpaid, overworked

3. If post taken down, poster may appeal decision to moderators

4. If post still not reinstated after 2nd review, poster may appeal decision to Oversight Board

5. Oversight Board decides whether to take on case

6. If Board takes on case, deliberations in private

7. Board issues recommendation

8. ????

Big question: Now what?!
Tech, Speech, and Moderation (3/3)

- Existing content moderation system deeply flawed, needs reform
  - at FB, moderators see 5,000+ pieces of content per day, much of it severely traumatizing
  - problem extends way beyond Facebook
- Given the Board’s decisions, how will this change moderation at Facebook and elsewhere?
  - Board’s recommendations are not binding, though Facebook must respond within 30 days
    - will Facebook follow recommendations?
  - Facebook’s VP for Global Affairs hopes other platforms and even governments will buy in
- Is Oversight Board an improvement on current system?
- Who should decide what is and isn’t said online?
  - governments? independent boards? private companies? ...
  - can self-regulation really work?
Bigger Picture: IT in the News Topics of Focus

- Areas include many subtopics
  - will discuss subtopics with varying levels of detail
- Complement lab/section activities
- Goals
  - give high-level overview of state of computing
  - expose you to real issues in industry
  - give you interesting questions to think about!

Social media & democracy

Social (Ir)responsibility in Tech Industry

Broader Impacts & Responsibilities of Technology

AI/ML, Algorithms, and Decision-Making