Information Session
Thursday, September 22\textsuperscript{nd}
5:00pm
Barus & Holley 161

shpe.brown@gmail.com
www.fb.com/shpebrown
CS15 & beyond
Gregory Chatzinoff ’15
Software Engineer, Chrome iOS
About Me

At Brown:

CS A.B.

Research Assistant in Andy’s group

HTA’d/TA’d CS15 (played Joffrey in Game of Thrones skit)

Involved with student government

Since graduation:

Software Engineer on Chrome iOS team in Mountain View, CA

UI improvements, stability/memory management experiments, back end work, data analysis, etc.
CS15 is a transformative course
Life of a Software Engineer in Industry

Large codebase with many contributors

  Code lasts a long time

Design Docs

Code Reviews
Wendy Ginsberg ’15
Associate Product Manager, Chrome Web Platform
Who am I?

Studied Computer Science (Class of ‘15)

- Started late, from Engineering
- Bachelor of Arts
- Took classes in 13 depts
- Musical Forum & Brown Debating Union

Worked as a TA

- CS15: 2013, 2014
- CS8, CS1951A, HIST97
I’m not a regular professor

I’m a cool professor.

CS15 kicked my butt & changed my life
“What if I don’t (think I) want to be a Software Engineer…?”
I’m not a Software Engineer!

Associate Product Manager at Google

PM on the Polymer team. OOP on Web!

Create vision, manage development, oversee product launch, repeat.

“Captain of the [Pirate] Ship”

Internships

Web/Mobile Developer at DreamIt Ventures

Software Engineer at Redfin
Wendy Ginsberg
Associate Product Manager, APM
wmginsberg@google.com
(include “CS15” in subject)
Wendy Ginsberg
Associate Product Manager, APM
wmginsberg@google.com
(include “CS15” in subject)
Interfaces and Polymorphism
Recall: Declaring vs. Defining Methods

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

```java
public class Dog {
    //constructor elided

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

public class School{
    private Teacher _teacher;
    public School() {
        _teacher = new Teacher(this);
        this.assignTeacher();
    }
    public void assignTeacher() {
        //code elided
    }
}

public class Teacher{
    private School _school;
    public Teacher(School school) {
        _school = school;
    }
    //other code elided
}

- Does School contain Teacher?
  - Yes! School instantiated Teacher, therefore School contains Teacher - Teacher is a component of School
- Can School send messages to Teacher?
- Does Teacher contain School?
  - No! Teacher knows about School that created it, but does not contain it
Using What You Know

● Imagine this program:
  o Sophia and Dan 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
  o assigns mode of transportation to each racer
  o starts the race

● For now, assume transportation options are Car and Bike
Goal 1: Assign transportation to each racer

- Need transportation classes (something to give to racers)
- 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

- Let’s build transportation classes

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

```java
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 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 (cont.)

- 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
}
```

```java
public class BikeRacer {
    private Bike _bike;

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

    public void useBike() {
        _bike.pedal();
    }

    //more methods elided
}
```
Goal 2: Tell the 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

```java
startRace:
  Tell _dan to useCar
  Tell _sophia to useBike
```
Coding the project (cont.)

- Let’s build the Race class

```java
public class Race {
    private CarRacer _dan;
    private BikeRacer _sophia;

    public Race() {
        _dan = new CarRacer();
        _sophia = new BikeRacer();
    }

    public void startRace() {
        _dan.useCar();
        _sophia.useBike();
    }
}
```
public class App {

    public App() {
        Race cs15Race = new Race();
        cs15Race.startRace();
    }

    public static void main (String[] args) {
        new App();
    }
}
What does our design look like?

How would this program run?

- An instance of `App` gets initialized
- `App`’s constructor initializes an instance of `Race`
- `Race`’s constructor initializes `_dan` (`CarRacer`) and `_sophia` (`BikeRacer`
  - `CarRacer`’s constructor initializes a `_car` (`Car`)
  - `BikeRacer`’s constructor initializes a `_bike`
- `App` calls `race.startRace()`
- `race` calls `_dan.useCar()` and `_sophia.useBike()`
- `_dan` calls `_car.drive()`
- `_sophia` calls `_bike.pedal()`
Can we do better?
Things to think about

- Do we need two different `Racer` classes?
  - Want multiple instances of `Racers` that use different modes of transportation
  - But how?
Solution 1: Create one Racer class with methods!

- Create one **Racer** class
  - define different methods for each type of transportation
- _dan_ is instance of **Racer** and elsewhere we have:
  ```java
  Car dansCar = new Car();
  _dan_.useCar(dansCar);
  ```
  - Car’s **drive**() method will be invoked
- But any given instance of **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 are a lot of work for you, the developer, and 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?

<table>
<thead>
<tr>
<th>Racer</th>
</tr>
</thead>
<tbody>
<tr>
<td>useCar(Car car)</td>
</tr>
<tr>
<td>useBike(Bike bike)</td>
</tr>
<tr>
<td>useHoverBoard(HoverBoard hoverboard)</td>
</tr>
<tr>
<td>useHorse(Horse horse)</td>
</tr>
<tr>
<td>useScooter(Scooter scooter)</td>
</tr>
<tr>
<td>useMotorcycle(Motorcycle motorcycle)</td>
</tr>
<tr>
<td>usePogoStick(PogoStick pogo)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Racer</th>
</tr>
</thead>
<tbody>
<tr>
<td>useTransportation()</td>
</tr>
</tbody>
</table>

Andries van Dam © 2016 9/22/16
Interfaces and Polymorphism

- In order to simplify code, need to learn
  - Interfaces
  - Polymorphism

```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 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

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

Bikes
- Move
- 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** class
  - 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>playRadio()</td>
<td>dropKickstand()</td>
</tr>
<tr>
<td>lockDoors()</td>
<td>changeGears()</td>
</tr>
<tr>
<td>unlockDoors()</td>
<td>pedal()</td>
</tr>
<tr>
<td>drive()</td>
<td></td>
</tr>
</tbody>
</table>
Introducing Interfaces

- **Interfaces** group similar capabilities/function 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
  - “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 error
    - later we’ll discuss strong motivations for this contract enforcement
- Interfaces don’t define their methods - implementing classes 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?
Let’s break that down

1) Models Similarities

2) Ensures Consistency
Models Similarities While Ensuring Consistency

- How does this help our program?
- We know **Cars** and **Bikes** both need to move
  - i.e., should all 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’s been defined
  - general tip: methods that interface declares should model functionality all implementing classes share
Interface Example (1/4)

What does this look like?

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

- That’s it!
- Interfaces, just like classes, have their own `.java` file. This file would be `Transporter.java`
Interface Example (2/4)

What does this look like?

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

- Declare it as `interface` rather than class
```
Interface Example (3/4)

What does this look like?

```java
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
Interface Example (4/4)

What does this look like?

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

- Interfaces can only declare methods - not define them
- Notice: method declaration end with **semicolons**, not curly braces!
Clicker Questions

Which line of this program is incorrect?

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

C. public class Rectangle implements Colorable {
   //constructor elided
   D. @Override
   public Color getColor() {
      E. return Color.PURPLE;
   }
}
Let's modify **Car**

```
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()**
- Will this code compile?
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?
  o nope :(
- Never implemented move() and 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)

Next: honor contract by defining a `move()` method

Method *signature* (name and number/type of arguments) must match how its declared in interface

```java
public class Car implements Transporter {

    public Car() {
        // constructor
    }

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

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

- Implementing class must define every single method in each of its every interfaces

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

public class Car implements Transporter, Colorable{
    public Car(){ //body 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 \texttt{Transporter} can \texttt{move()}
- Will know how 2 classes are related if both implement \texttt{Transporter}
Clicker 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() {
       this.changeXPosition(100.0);
   }
```  

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 double click() {
       return this.changeXPosition(100.0);
   }
```
Back to the CIT Race

- Let’s make transportation classes use an interface

```java
public class Bike implements Transporter {

    public Bike() {
        //code elided
    }

    public void pedal() {
        //code elided
    }

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

    //more methods elided
}
```

```java
public class Car implements Transporter {

    public Car() {
        //code elided
    }

    public void drive() {
        //code elided
    }

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

    //more methods elided
}
```
Leveraging Interfaces

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

<table>
<thead>
<tr>
<th>Racer</th>
</tr>
</thead>
<tbody>
<tr>
<td>useCar(Car car)</td>
</tr>
<tr>
<td>useBike(Bike bike)</td>
</tr>
<tr>
<td>useHoverBoard(HoverBoard hoverboard)</td>
</tr>
<tr>
<td>useHorse(Horse horse)</td>
</tr>
<tr>
<td>useScooter(Scooter scooter)</td>
</tr>
<tr>
<td>useMotorcycle(Motorcycle motorcycle)</td>
</tr>
<tr>
<td>usePogoStick(PogoStick pogo)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Racer</th>
</tr>
</thead>
<tbody>
<tr>
<td>useTransportation()</td>
</tr>
</tbody>
</table>
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 \texttt{move()} \rightarrow refer to them as \texttt{Transporter} objects
    - phones and Teslas can both \texttt{getCharged()} \rightarrow refer to them as \texttt{Chargeable} objects, i.e., objects that implement \texttt{Chargeable} interface
    - cars and boomboxes can both \texttt{playRadio()} \rightarrow refer to them as \texttt{RadioPlayer} objects
- How do we write one generic \texttt{useTransportation()} method?
What would this look like in code?

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

This is polymorphism! The `transportation` object passed in could be instance of `Car`, `Bike`, etc., i.e., any class that implements the interface.
Let’s break this down.

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

1. Actual vs. Declared Type
2. Method resolution
Actual vs. Declared Type (1/2)

- Consider following piece of code:

  ```java
  Transporter dansCar = 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` object as `Transporter` objects
- **Car** is the actual type
  - Java will look in this class for the definition of the method
- **Transporter** is declared type
  - Java will limit caller so it can only call methods on instances that are declared as `Transporter` objects
- If **Car** defines `playRadio()` method. Is `transportation.playRadio()` correct?

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

Nope. The `playRadio()` method is not declared in `Transporter` interface, therefore Java does not recognize it as viable method call.
Determining the Declared Type

- What methods do Car and Bike have in common?
  - `move()`

- How do we know that?
  - They implement `Transporter` guarantees that they have `move()` method

- Think of `Transporter` like the “lowest common denominator”
  - It’s what all transportation classes will have in common

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

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

`Transporter sophiasBike = new Bike();` ✓

`Transporter sophiasCar = new Car();` ✓

`Transporter sophiasRadio = new Radio();` ✗

Radio wouldn’t implement `Transporter`. Since Radio cannot “act as” a `Transporter`, you cannot treat it as `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 instance of `Car` can `playRadio()` or if 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 object that implements `Transporter`
● `useTransportation` can only call methods declared in `Transporter`
Is this legal?

```java
Transporter sophiasBike = new Bike();
_sophia.useTransportation(sophiasBike);
```

Even though
```
sophiasCar is
declared as a Car,
the compiler can still
verify that it implements
Transporter.
```
```
Car sophiasCar = new Car();
_sophia.useTransportation(sophiasCar);
```

```
Radio sophiasRadio = new Radio();
_sophia.useTransportation(sophiasRadio);
```

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?

```java
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 \texttt{move()} is executed?

- Consider this line of code in \texttt{Race} class:
  
  \_sophia.useTransportation(\texttt{new Bike()});

- Remember what \texttt{useTransportation} method looked like
  
  \begin{verbatim}
  public void useTransportation(Transporter transportation) {
    transportation.move();
  }
  \end{verbatim}

What is “actual type” of \texttt{transportation} in this method invocation?
Method Resolution (1/4)

public class Race {
    private Racer_sophia;
    //previous code elided

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

public class Racer {
    //previous code elided

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

● Bike is actual type
  ○ Racer was handed instance of Bike
    ▪ new Bike() is argument

● Transporter is declared type
  ○ 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() {
        _sophia.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();
    }
}

● _Sophia is a Racer
● Bike’s move() method gets used
● Why?
  o Bike is actual type
    ▪ Java will execute methods defined in Bike class
  o Transporter is declared type
    ▪ Java limits methods that can be called to those declared in Transporter interface
Method Resolution (3/4)

- What if \_sophia received 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() {
        _sophia.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 – Java 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
Clicker Question

Given the following class:

```java
public class Laptop implements Typeable, Clickable {
    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. Typable macBook = new Laptop();
   macBook.click();

D. Clickable macBook = new Laptop();
   macBook.click();
Why does that 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, the methods can get implemented in any way

![Diagram](image)
Why does that 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!
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
  o will guarantee that class has certain capabilities
    ▪ Car implements Transporter, therefore it must know how to move()

● Polymorphism
  o 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:
  o need more classes → more specialized methods
    (useRollerblades(), useBike(), etc)
● New Design:
  o as long as the new classes implement Transporter, Racer
even if the new classes implement different transportation
    doesn’t care what transportation it has been given
  o don’t need to change Racer!
    ▪ less work for you!
    ▪ just add more transportation classes that implement Transporter
public class Race {
    private Racer _dan, _sophia;

    public Race() {
        _dan = new Racer();
        _sophia = new Racer();
    }

    public void startRace() {
        _dan.useTransportation(new Car());
        _sophia.useTransportation(new Bike());
    }
}

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

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() {
        this.drive();
    }
}

public class Bike implements Transporter {
    public Bike() {}

    public void pedal() {
        // code elided
    }

    public void move() {
        this.pedal();
    }
}
In Summary

- Interfaces are contracts
  - force classes to define certain methods
- Polymorphism allows for extremely 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

- HW2 and LiteBrite are released today
- HW2 is due on Sunday (9/25) 2PM
  - Remember: homeworks only have one handin time
- LiteBrite early handin is Tuesday 09/27, on-time handin is Thursday 9/29, late handin is Saturday 10/1
  - LiteBrite help slides will be released on Friday
- Review sessions are held every Thursday from 7:30-9pm and Sunday from 12:00-1:30pm in MacMillan 115
- **START EARLY, START TODAY, START YESTERDAY!!!!!**
- **PLEASE FILL OUT THE INITIAL SURVEY**