Lecture 6
Interfaces and Polymorphism

TopHat Survey Question
How much use are you making of recorded lectures?
A. Most of the time
B. Occasionally
C. Hardly ever

Outline
- Transportation Example
- Intro to Interfaces
- Implementing Interfaces
- Polymorphism
Review: Containment and Association

- Containment and association are two key ways of establishing relationships between instances of a class
- In containment, one class creates an instance of another (its component) and can call methods on it
- In association, one instance of a class knows about an instance of another class (that is not its component) and can call methods on it
- Containment and association are side-effects of delegating responsibilities to other classes
  - they are modeling/design patterns, not Java constructs and require no new syntax

Outline

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

Using What You Know

- Imagine this program:
  - Daniel and Lila 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 Daniel and Lila 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() {
        this.car = new Car();
    }
    public void useCar() {
        this.car.drive();
    }
    //more methods elided
}
```

```java
public class BikeRacer {
    private Bike bike;
    public BikeRacer() {
        this.bike = new Bike();
    }
    public void useBike() {
        this.bike.pedal();
    }
    //more methods elided
}
```

Goal 2: Tell racers to start the race

- Race class contains Racers
  - App contains Race
  - Tell this.daniel to useCar
  - Tell this.lila to useBike
- Race class will have `startRace()` method
  - `startRace()` tells each Racer to use their transportation
  - `startRace()` gets called in App

Coding the project (3/4)

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

```java
public class Race {
    private CarRacer daniel;
    private BikeRacer lila;
    public Race() {
        this.daniel = new CarRacer();
        this.lila = new BikeRacer();
    }
    public void startRace() {
        this.daniel.useCar();
        this.lila.useBike();
    }
    // Old code
    public void startRace() {
        this.daniel.useCar();
        this.lila.useBike();
    }
    //more methods elided
}
```
Coding the project (4/4)

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

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

The Program

```java
public class Race {
    private CarRacer daniel;  
    private BikeRacer lila;

    public Race() {
        daniel = new CarRacer();
        lila = new BikeRacer();
    }

    public void startRace() {
        daniel.useCar();
        lila.useBike();
    }
}
```

Flow of control (1/2)

How would this program run?
- Java initializes an instance of `App`, calling `main`
- `main` initializes an instance of `Race`
- `Race`'s constructor initializes `daniel`, a `CarRacer` and `lila`, a `BikeRacer`
- `CarRacer`'s constructor initializes `car`, a `Car`
- `BikeRacer`'s constructor initializes `bike`, a `Bike`
Flow of control (2/2)

- With all instances constructed, App calls cs15Race.startRace()
- cs15Race calls this.daniel.useCar() and this.lila.useBike()
- daniel calls this.car.drive()
- lila 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 that use different modes of transportation (useCar and useBike)
    - both classes are very similar, they just use their own mode of transportation
    - 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
  - Use different use methods for each type of transportation
- Daniel would be an instance of Racer and in startRace we would call:
  - this.daniel.useCar(new Car());
- 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();
    }
    // And more…
}
```

Solution 1 Drawbacks

- Now imagine all the CS15 TAs join the race and there are 10 different modes of transportation
- Writing these similar use 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 hoverboard) {
        //code elided
    }
    public void useHorse(Horse horse) {
        //code elided
    }
    public void useScooter(Scooter scooter) {
        //code elided
    }
    public void useMotorcycle(Motorcycle motorcycle) {
        //code elided
    }
    public void usePogoStick(PogoStick pogo) {
        //code elided
    }
    // And more…
}
```

Is there another solution?

- Can we go from left to right?

```java
Racer
    useTransportation()
        useHoverboard(hoverboard)
            useMotorcycle(motorcycle)
                useScooter(scooter)
                    useBike(bike)
                        useCar(car)
```

```java
Racer
    useTransportation()
        usePogoStick(pogo)
            useMotorcycle(motorcycle)
                useScooter(scooter)
                    useBike(bike)
                        useCar(car)
```
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:

```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(Tanker transport) {
        transport.move();
    }
}
```

```java
public interface Transporter {
    public void 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

- Play radio
- Turn on/off headlights
- Turn on/off turn signal
- Lock/unlock doors
- ...  
- 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 modeled through code?
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
  - have other shared functionality, such as braking, steering, etc.
- for this lecture, interfaces are green and classes that implement them are pink

<table>
<thead>
<tr>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td>move()</td>
</tr>
<tr>
<td>brake()</td>
</tr>
<tr>
<td>lockbrake()</td>
</tr>
<tr>
<td>unlockbrake()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bike</th>
</tr>
</thead>
<tbody>
<tr>
<td>move()</td>
</tr>
<tr>
<td>brake()</td>
</tr>
<tr>
<td>lockbrake()</td>
</tr>
<tr>
<td>unlockbrake()</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 {
    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 {
    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?

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

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

Outline

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

Implementing an Interface (1/6)

```java
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)

```java
public class Car implements Transporter {
    public Car() {
        // constructor
    }
    public void drive() {
        // code for driving
        // the car
    }
}
```

- Will this code compile? Nope!
- Never implemented `move()` method.
  - Compiler will complain accordingly.
  - Error message: "Car does not override method move() in Transporter."
  - 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.
- Include `@Override` right above the method signature.
- `@Override` is an annotation — a signal to the compiler (and to anyone reading your code) that 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 (4/6)

What does `@Override` mean?

```java
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) that 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)

```java
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 to be able to change color as well?
  - create a Colorable interface and add this interface to the 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
TopHat Question

Can you instantiate an interface as you can a class?
A. Yes
B. No

TopHat Question

Can an interface define code for its methods?
A. Yes
B. No

TopHat Question

Which statement of this program is incorrect?
A. public interface Colorable {
   Color getColor() {
      return Color.WHITE;
   }
}
B. public class Rectangle implements Colorable {
   @Override
   public Color getColor() {
      return Color.PURPLE;
   }
}
TopHat Question
Given the following interface:

```java
public interface Clickable {
    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 clickIt() {
       this.changeXPosition(100.0);
   }
```  
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 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?

```
Racer
  useCar(car)
  useBike(bike)
  useHoverBoard(hoverboard)
  useHorse(horse)
  useScooter(scooter)
  useMotorcycle(motorcycle)
  usePogoStick(pogo)
```

Racer
  `useTransportation(…)`
Outline

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

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?

What would this look like in code?

```java
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();
    }
}
```

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

```
52
```

Actual vs. Declared Type (1/2)

- Consider following polymorphic assignment statement:
  ```java
  Transporter danielsCar = new Car();
  ```

- We say “danielsCar” is of type `Transporter”, but we instantiate a new `Car` and assign it to `danielsCar”... 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 are defined in that interface
- If `Car` defines `playRadio()` method, is this correct?
  ```java
  transportation.playRadio();
  ```

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

```
53
```
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 Car implements Transporter {
    void move();
    void playRadio();
    //etc.
}
```

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

Is this legal?

```java
Transporter lilasBike = new Bike(); ✓
Transporter danielsCar = new Car(); ✓
Transporter danielsRadio = new Radio(); ✗
```

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

\[
\text{public void useTransportation(Transporter transportation) \{ \\
\quad //code elided \\
\}}
\]

- 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?

\[
\text{public void useTransportation(Transporter transportation) \{ \\
\quad //code elided \\
\}}
\]

- Transporter lilasBike = new Bike();
- this.lila.useTransportation(lilasBike);
- Car danielsCar = new Car();
- this.daniel.useTransportation(danielsCar);
- Radio danielsRadio = new Radio();
- this.daniel.useTransportation(danielsRadio);

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

Even though danielsCar 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?

\[
\text{public class Racer \{ \\
\quad //previous code elided \\
\quad public void useTransportation(Transporter transportation) \{ \\
\quad \quad transportation.move(); \\
\quad \} \\
\quad \\
\}}
\]
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
  this.lila.useTransportation(new Bike());
  ```
- Remember what `useTransportation` method looks like
  ```java
  public void useTransportation(Transporter transportation) {
    transportation.move();
  }
  ```
  What is "actual type" of `transportation` in `this.lila.useTransportation(new Bike());`?

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

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

Method Resolution (1/4)

- Bike is actual type
  - lila 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()`?
  - `move()` method gets used?

```
public class Race {
  private Racer lila;
  //previous code elided
  public void useTransportation(Transporter transportation) {
    transportation.move();
  }
}
```
Method Resolution (2/4)

- lila 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 Race {
    // previous code elided
    public void startRace() {
        this.lila.useTransportation(new Bike());
    }
}
```

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

Method Resolution (3/4)

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

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

```java
public class Race {
    // previous code elided
    public void startRace() {
        this.lila.useTransportation(new Car());
    }
}
```

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

Method Resolution (4/4)

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

C. `Clickable macBook = new Typeable();
   macBook.type();`

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 of the actual code called
    - 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?

- 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
    - 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 `daniel`, a `Racer`, and `lila`, a `Racer`
- App calls `cs15Race.startRace()`
- `cs15Race` calls:
  - `this.daniel.useTransportation(new Car())`
  - `this.lila.useTransportation(new Bike())`
- `useTransportation` method which calls `this.move()` method which calls `this.drive()`

The Program

```java
public class Race {
    private Racer daniel, lila;
    public Race() {
        this.daniel = new Racer();
        this.lila = new Racer();
    }
    public void startRace() {
        this.daniel.useTransportation(new Car());
        this.lila.useTransportation(new Bike());
    }
}
```

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

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

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

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

```java
public interface Transporter {
    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

- TicTacToe released today (9/28)
  - Early hand-in: 10/1
  - On-time hand-in: 10/3
  - Late hand-in: 10/5
- Class Relationships Section
  - Mini Assignment due before section
  - Email answers to your section TA
- CS15 Mentorship
  - Officially begun!
  - Identity/Interest-based form out

---

Topics in Social Responsible Computing

How Big Tech Does Ethics

What do “ethics” mean for Big Tech?

- Rise of internal ethics boards for large tech corporations—primarily surrounding AI
  - In the past 5 years, IBM, Microsoft, Google, Salesforce, Facebook, Apple, Amazon all create “ethics boards” or “ethics groups”
    - Range of issues: bias, uneven/adverse impacts, diversity
- “Ethics washing” and other common critiques
  - Cannot undo fundamental culture: meritocracy, technosolutionism, prioritizing profit over all else
  - Committees often at the mercy of the company
    - Who selects board, how committed are they
    - Facebook Oversight Board (FBOB), supposedly independent
Limits of Big Tech Ethics: Google (1/3)

- **Dr. Timnit Gebru**: Trailblazing AI researcher, led Google Ethical AI (research team), founder of Black in AI, advocate for diversity & inclusion in computing.
- December 2020: Google censors research paper by Dr. Gebru et al. on dangers of language technology.
  - “On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?” explores the risks of language models (limits of the data set used to train data).
  - Google claimed the paper “ignored too much relevant research.”
- Dr. Gebru refuses to revise or rescind → Google fires her over email!

Limits of Big Tech Ethics: Google (2/3)

- **Dr. Margaret Mitchell**: co-founder & lead of Google Ethical AI team.
  - Advocated for better D&I at Google.
  - Publicly supported Dr. Gebru after firing.
- Feb. 2021: Google fired her after locking her out of professional accounts for 5 weeks.
  - Allegedly for downloading files.
  - Decision delivered as internal investigation of incident surrounding Dr. Gebru’s departure. Speculation that these incidents are related.

Limits of Big Tech Ethics: Google (3/3)

**Why does this matter?**

- Google benefited from Dr. Gebru’s & Dr. Mitchell’s presence & work...
  - academic contributions: both are high-profile researchers in AI & ethics.
  - diversity.
- ...but were they too willing to fire them?
  - Sends message of Google violating academic freedom & stifling critique of company.
- Note: we don’t take sides, Google had reasons that are unknown to the public.
Other examples of ‘Big Tech Ethics’

- 2018: Facebook formed Society and AI Lab (SAIL)
  - Focus on bias in AI/ML algorithms
- Is this the most important issue for responsible AI @ Facebook?
  - Criticized for not addressing more pressing issues of misinformation and hate speech

Facebook AI and Misinformation (Again)

- Facebook engagement-first algorithms actively amplify misinformation according to in-house studies
- Initially, SAIL proposed ML-based techniques against polarization
  - Little buy-in from executives
  - Company accused of prioritizing growth above all else
  - Insufficient incentive to reduce extreme content

Facebook graphic visualizing “natural engagement patterns”

Facebook AI and Misinformation (Again)

- FB SAIL focuses on bias in AI (not misinformation)
  - doesn’t threaten growth of platform (no decrease in “engagement” metrics)
  - may also help Facebook avoid proposed regulation (e.g., Algorithmic Accountability Act of 2019 – died in Congress in 2019)
  - gives outward impression of care about AI ethics: ethics-washing
- Caveat: Facebook does have some AI systems & teams combating hate speech & misinformation
  - but ML systems, by design, cannot catch new types of hate speech
  - no dedicated team investigating how engagement algorithms fuel misinformation
More reading that may be of interest!

- Datasociety—Owning Ethics
- NPR—Dr. Timnit Gebru Profile
- The Verge—The Problem with AI Ethics
- Dr. Timnit Gebru et al.—On the Dangers of Stochastic Parrots: Can Language Models Be too Big
- Reuters—Money, mimicry and mind control: Big Tech slams ethics brakes on AI
- Aspen Institute—Deplatforming Trump: The Facebook Oversight Board Decision