Lecture 6
Interfaces and Polymorphism

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 consequences of delegating responsibilities to other classes
  - they are design choices, not Java constructs and require no new syntax
Outline

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

Using What You Know

- Imagine this program:
  - Lexi and Anastasio 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 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 Bike {
    public Bike() {
        // constructor
    }
    public void pedal() {
        // code elided
    }
    // more methods elided
}
public class Car {
    public Car() {
        // constructor
    }
    public void drive() {
        // code elided
    }
    // more methods elided
}
```

Goal 1: Assign transportation to each racer

- Need racer classes that will tell Lexi and Anastasio to use their type of transportation
  - `CarRacer`
  - `BikeRacer`

  What methods will we need? What capabilities should each `racer` class have?

  - `CarRacer` needs to know how to use the car
    - write `useCar()` method, uses `drive()`, shields caller from knowing what all `car` might need to do
  - `BikeRacer` needs to know how to use the bike
    - write `useBike()` method, uses `pedal()`, shields caller from knowing what all `bike` might need to do

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();
        // other methods as needed
    }
    // more methods elided
}
public class BikeRacer {
    private Bike bike;
    public BikeRacer() {
        this.bike = new Bike();
    }
    public void useBike() {
        this.bike.pedal();
        // other methods as needed
    }
    // more methods elided
}
Goal 2: Tell racers to start the race

- Race class contains Racers
  - App contains Race
- 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 lexi;
    private BikeRacer anastasio;
    public Race() {
        this.lexi = new CarRacer();
        this.anastasio = new BikeRacer();
    }
    public void startRace() {
        this.lexi.useCar();
        this.anastasio.useBike();
    }
}
```

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();
    }
}
```

//from the Race class on slide 11
The Program

```java
public class Race {
    private CarRacer lexi;
    private BikeRacer anastasio;
    public Race() {
        this.lexi = new CarRacer();
        this.anastasio = new BikeRacer();
    }
    public void startRace() {
        this.lexi.useCar();
        this.anastasio.useBike();
    }
}

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

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 lexi, a CarRacer and anastasio, a BikeRacer
- CarRacer's constructor initializes car, a Car
- BikeRacer's constructor initializes bike, a Bike

Flow of control (2/2)

- After Race constructs lexi and anastasio, App calls cs15Race.startRace()
- lexi calls useCar() and anastasio calls useBike()
- useCar() calls this.car.drive()
- useBike() calls this.bike.pedal()
Can we do better?

Things to think about

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

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

- Create one Racer class
- define different use methods for each type of transportation
  - lexi would be an instance of Racer and in startRace we would call:
    - this.lexi.useCar(new Car());
    - useCar() method will be invoked
- Good: only one Racer class
- But: Racer has to aggregate a useX() 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 car) {
        //code elided
    }
    public void useBike(Bike bike) {
        //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?

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

```java
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

- Play radio
- Lock/unlock doors
- "...

- Move
- 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
- We prefer the more general move to the previous useCar, useBike

### Can we model this in code?

Many real-world objects have several broad functional similarities:
- cars and bikes can move
- phones and Teslas can be charged
- Take car and bike classes
  - One can drive both functionalities get enumerated in one place?
  - Some can have broad interfaces get enumerated
- Note: cars and bikes serve a similar purpose while phones and Teslas don’t — we only care that they share some similar functionality (but potentially quite different implementations)

<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)

- Interface groups declarations of similar capabilities of different classes together
- Looks like a totally stripped-down class declaration, with just method declarations:

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

- Cars and Bikes can “implement” a Transporter Interface
  - they can transport people from one place to another
  - they move, transport
  - for this lecture, interfaces are given and classes that implement them are used
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 they are defined.
- Models similarities while ensuring consistency.
  - What does this mean?

Models Similarities while Ensuring Consistency (1/3)

Let’s break that down into two parts:

1) Model Similarities
2) Ensure Consistency

Models Similarities while Ensuring Consistency (2/3)

- How does this help our program?
- We know *Cars* and *Bikes* both need to move.
  - i.e., should both have some `move()` method.
  - Let compiler know that too!
- Make the *Transporter* interface.
  - What methods should the *Transporter* interface declare? Similar!
  - `move()` (plus brake, steer...)
  - Compiler ensures consistency — doesn’t care how method is defined, just that it has been defined.
  - General tip: methods that interface declares should model functionality 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, we show only one required method: `move`
- All classes that sign contract (implement this interface) must implement any declared methods

What does this look like?

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

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 and with semicolons, not curly braces!

What does this look like?

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

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`
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
    }

    // Error: Car does not override method move() in Transporter
}
```

- Will this code compile?
  - No, it will fail
- 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 learn more about meaning of abstract in later lectures.*
Implementing an Interface (3/6)

public class Car implements Transporter {
    public Car() { }
    public void drive() { //code for driving car
        @Override
        public void move() { this.drive(); }
    }
}

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

Implementing an Interface (4/6)

public class Car implements Transporter {
    public Car() { }
    public void drive() { //code for driving car
        @Override
        public void move() { this.drive(); this.brake(); this.drive(); }
    }
    public void brake() { //code elided }
}

- 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
  - inclusion of @Override in source file
- Annotations, like comments, have no effect on how code behaves at runtime

What does @Override mean?

Implementing an Interface (5/6)

public class Car implements Transporter {
    public Car() { }
    public void drive() { //code for driving car
        @Override
        public void move() { this.drive(); this.move(); this.drive(); this.brake(); }
        public void brake() { //code elided }
    }
}

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

- Defining interface method is like defining any other method
- Definition can be as simple or complex as it needs to be
- Ex.: Let's modify Car's move method to include braking
- What will instance of Car do if move() gets called on it?
Implementing an Interface (6/6)

- As with signing multiple contracts, classes can implement multiple interfaces.
- "I signed my rent agreement, so I’m a renter, but I also signed my employment contract, so I’m an employee. What if I wanted to change roles as well?"
- Classes implementing interfaces must define every single method from each interface.

```java
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 }
}
```

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

Modeling Similarities While Ensuring Consistency (3/3)

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

TopHat Question

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

TopHat Question

Given the following interface:
public interface Clickable {
   public void click();
}
Which of the following would work as an implementation of the Clickable interface? (Don’t worry about what changeXPosition does)
A. @Override
   public void click(double xPosition) {
      this.changeXPosition(xPosition);
   }
B. @Override
   public void clickIt() {
      this.changeXPosition(100.0);
   }
C. @Override
   public double click() {
      return this.changeXPosition(100.0);
   }
D. @Override
   public void click() {
      this.changeXPosition(100.0);
   }
Back to the CIT Race

- Let’s make transportation classes use an interface

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

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

Leveraging Interfaces

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

```
useTransportation()
```

Outline

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

- Poly = many, morph = forms
- A way of coding generically
  - way of referencing multiple classes sharing abstract functionality as acting as one generic type
  - cars and bikes can both move() — refer to them as classes of type Transporter
  - phones and Teslas can both getCharged() — refer to them as classes of type Chargeable, i.e., classes that implement Chargeable’s 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?
Actual vs. Declared Type (1/2)

- We first show polymorphic assignment (typically not useful by itself) and then polymorphic parameter passing.
- Consider following polymorphic assignment statement:
  ```java
  Transporter lexisCar = new Car();
  ```
- We say "lexisCar" is of type `Transporter`, but we instantiate a new `Car` and assign it to `lexisCar`. 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();
  ```

Is this legal?

- `Transporter anastasiosBike = new Bike();` ✓
- `Transporter lexisCar = new Car();` ✓
- `Transporter lexisRadio = new Radio();` ✗

Radio wouldn't implement `Transporter`. Since `Radio` cannot "act as" type `Transporter`, you cannot treat it as of type `Transporter`. 
### Only Declared Type’s Methods Can be Used

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

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

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

### Motivations for Polymorphism

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

```java
public void useTransportation(Transporter transportation) {
    // code elided
    Transportation montezuma = new Transportation()
    this.montezuma.useTransportation(montezuma);
    Car lexusCar = new Car();
    this.lexusCar.useTransportation(lexusCar);
    Radio lexisRadio = new Radio();
    this.lexis.useTransportation(lexisRadio);
}
```

- Even though lexusCar is declared as a Car, the compiler can still verify that it implements Transporter
- A Racer that extends Transportation cannot treat a Type of Transportation

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

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

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

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

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

- Only have access to instance of type `Transporter`
  - cannot call `transportation.drive()` or `transportation.pedal()`
  - that's okay, because all that's needed is `move()`
  - limited to the methods declared in `Transporter`
Method Resolution: Which `move()` is executed?

- Consider this line of code in `Race` class:
  ```java
this.anastasio.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.anastasio.useTransportation(new Bike());`?

Method Resolution (1/4)

- `Bike` is the actual type of `transportation` as `Racer` was handed a `new Bike()` instance as argument
- `Transporter` is the declared type
- This instance is treated as type of `Transporter`
- So… what happens in `transportation.move()`?
- Which `move()` method gets used?

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

Method Resolution (2/4)

- `anastasio` is a `Racer`
- `Bike` is the actual type of this `Transporter`
- `Bike` can only call methods of `Bike` interface
- `Transporter` is the declared type
- `useTransportation` method gets used
- Why?
  - `Bike` is the actual type of this `Transporter`
  - can only call methods of `Bike` interface
  - `Transporter` is the declared type
  - `useTransportation` method gets used
Method Resolution (3/4)

- What if anastasio received an instance of Car?
  - What move() method would get called then?

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

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

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

Method Resolution (4/4)

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

TopHat Question

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.type();`
C. `Clickable macBook = new Laptop(); macBook.click();`
D. `Clickable macBook = new Clickable(); macBook.type();`
Why does polymorphism work when calling methods?

- Declared type and actual type work together
  - Declared type keeps things generic
    - Can reference many classes using one generic type
  - Actual type ensures specificity
    - When calling declared type's method on an instance, the actual code that is called is the code defined in the actual type's class (dynamic binding)
  
- Every district does their job!
- District 12's job is specifically mining coal

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
        - Can have implementation-agnostic classes and methods
          - Use interface Transporter, can take in any instance of type Transporter
          - Can easily extend this program to use any form of transportation, with minimal changes to existing code
          - A tool for extensible programming
            - How?
Why is this important?

- Using more than 2 methods of transportation?
  - 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?

- `App`'s constructor initializes `lexi`, an instance of `Racer`
- `App`'s constructor initializes `anastasio`, a `Racer`
- `App` calls `cs15Race.startRace()`
- `cs15Race` calls:
  - `this.lexi.useTransportation(new Car())`
  - `this.anastasio.useTransportation(new Bike())`
- `useTransportation(new Car())` initializes a `Car` and calls `Car`'s `move()` method which calls `this.drive()`
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 (9/26)
  - Early hand-in: 9/26
  - On-time hand-in: 9/30
  - Late hand-in: 10/2
- Class Relationships Section
  - Mini Assignment due before section
  - Email answers to your section TA
- CS15 Mentorship
  - Officially begun!
- T-Shirt Contests!!!!
  - Designs due Thursday before Lecture!! (looking at you RISD students :D)
From Stochastic Parrot to Coherent Language

Coined by American Linguist Emily Bender in her paper: On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?

How to Train Your Dragon LLM

• As discussed in last lecture we need to feed our model data to train the weights
• In our first example this data was images of numbers
• To train a LLM we require, instead, massive amounts of textual data

Word Embeddings

• An embedding is a sequence of numbers that represents each token, and each token has a unique sequence
• It can be difficult to visualize word embeddings since they exist in a high-dimensional space
• Consider the following example:
  • How do we visualize the following 2D example?
  • How can we understand the relationships among these words?
  • How can we map these relationships so well that it mimics the natural language relationships and semantics of the words?
Word Embeddings (Continued)

Following Training: King - Man + Woman = Queen

Garbage in, Garbage Out (GIGO)

- Flawed inputs create flawed outputs
  - "Why did the machine give me the right answer if the inputs were wrong?" - Passages from the Life of a Philosopher (1864)
  - Since ChatGPT is trained on false or incorrect statements, it will (confidently) produce flawed outputs
  - This is why ChatGPT appears to hallucinate sometimes...
  - ChatGPT may also appear politically biased

Fine Tuning

- To reduce incorrect and biased outputs and to fix the model's poorly trained weights, the model is fine-tuned after initial training
- Sama uses gig workers in developing economies to create training datasets for Silicon Valley clients
- Sama workers, for example, manually labeled toxic responses for ChatGPT to build a mechanism for filtering them out
- Other clients include Google, Meta, and Microsoft
- Fine-tuning can also include training the model to perform better at certain tasks or conform to a certain writing style!
Further Courses @ Brown post CS15 & CS200

Many courses in the Artificial Intelligence/Machine Learning pathway go in depth and have you implement what we discussed over the last lectures!

- CS1410 – Artificial Intelligence
- CS1420 – Machine Learning
- CS1430 – Computer Vision
- CS1440 – Computational Linguistics
- CS1470 – Deep Learning
- CS1496 – Data Science

Introducing GPTA!

- GPTA is CS15’s very own “virtual TA” Chatbot
- Instead of using ChatGPT or other chatbots for questions, you can ask GPTA!
- GPTA is a great resource for those quick questions and misunderstandings you have about concepts and syntax
- Access will be granted in your section this week
  - If you had section this morning, you will be granted access shortly after lecture :)

www.cs15gpta.com

Usage Guidelines

- You CAN ask: conceptual questions, for code examples explaining concepts
- You CANNOT ask: debugging questions, for project code
  - Specific examples of these are on the CS15 Ethics Usage Doc
- You’ll see these guidelines every time you sign in to GPTA
- We have a user guide and usage guidelines on the CS15 Policy and the CS15 Ethics Usage Doc
Terms and Conditions

• To make sure that this tool is not being abused, we will be logging all questions and responses.
  • we will be reviewing these responses to make sure no disallowed questions (e.g., 'debug my code', 'generate project code' questions)

• Before you can start using GPTA, you must fill out our Terms and Conditions form.
  • acknowledges you understand GPTA’s role in our course, how you must use it, and that we will be monitoring questions asked.

DISCLAIMERS

• This is a big experiment!
  • caution advised – issues are expected early on
  • feedback form linked on the GPTA website

• Like all GenAI, GPTA will occasionally produce inaccurate and irrelevant information – not a replacement for real TA help.
  • jut like with ChatGPT, sometimes issues with generated code
  • Explanations are based on general info in the wild, not specific CS15 ways we teach OOP
  • may be differences in terminology and concept explanations, as well as style

• Anticipating some server load issues.
  • You’re guinea pigs, based on our testing we found it useful but your mileage may vary

  • bear with us as we figure this out together!