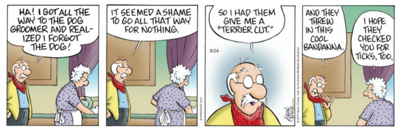
Lecture 6

Interfaces and Polymorphism



1

Outline

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

2

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

3

Outline

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

4/92

4

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`

5/92

5

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

6/92

6

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
}
```

```java
public class Bike {

    public Bike() { //constructor
        //code elided
    }
    public void pedal() {
        //code elided
    }
    //more methods elided
}
```

 7/92

7

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

 8/92

8

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
}
```

```java
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
}
```

 9/92

9

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

startRace:
  Tell this.lexi to useCar
  Tell this.anastasio to useBike

10

Coding the project (3/4)

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

```
public class CarRacer {
    private Car car;

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

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

//BikeRacer class elided
```

Old code

```
public class Race {
    private CarRacer lexi;
    private BikeRacer anastasio;

    public Race() {
        this.lexi = new CarRacer();
        this.anastasio = new BikeRacer();
    }

    public void startRace() {
        this.lexi.useCar();
        this.anastasio.useBike();
    }
}
```


11

Coding the project (4/4)

```
public class App {

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

}
```

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

public void startRace() {
    this.lexi.useCar();
    this.anastasio.useBike();
}
```

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

12

The Program

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

```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();
    }
}
```

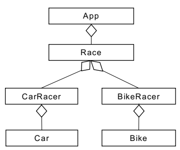
```java
public class CarRacer {
    private Car car;

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

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

}
```

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

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

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


13/92

13

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

14/92

14

Flow of control (2/2)

```java
public class App {

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

```java
public class Race {
    // constructor elided; creates lexi and anastasio

    public void startRace() {
        this.lexi.useCar();
        this.anastasio.useBike();
    }
}
```

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

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

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

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

- 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()

15/92

15

Can we do better?

16

Things to think about

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

17

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());`
  - `Car`'s `drive()` method will be invoked
- Good: only one `Racer` class
- But: `Racer` has to aggregate a `use…()` method to accommodate every kind of transportation!

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

    public void useCar(Car myCar){
        myCar.drive();
    }

    public void useBike(Bike myBike){
        myBike.pedal();
    }
}
```


18

Solution 1 Drawbacks

- Now imagine all the CS15 TAs join the race and there are 10 different modes of transportation
- Writing these similar `useX()` methods is a lot of work for you, as the developer, and it is an inefficient coding style

```
public class Racer {

    public Racer() {
        //constructor
    }
    public void useCar(Car myCar){//code elided}
    public void useBike(Bike myBike){//code elided}
    public void useHoverboard(Hoverboard myHb){//code elided}
    public void useHorse(Horse myHorse){//code elided}
    public void useScooter(Scooter myScooter){//code elided}
    public void useMotorcycle(Motorcycle myMc) {//code elided}
    public void usePogoStick(PogoStick myPogo){//code elided}
    // And more…
}
```


19

Is there another solution?

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

→

Racer
useTransportation(…)

- Can we go from left to right?

20

Outline

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

21

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 Racer {

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

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

```java
public class Car implements Transporter {

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

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


22

Interfaces: Spot the Similarities

- What do cars and bikes have in common?
- What do cars and bikes *not* have in common?

23

Cars vs. Bikes

<u>Cars</u>

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

Cars and Bikes

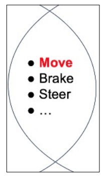
- Move
- Brake
- Steer
- …

<u>Bikes</u>

- Drop kickstand
- Change gears
- …

24

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

25/92

25

Can we model this in code?

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

Car

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

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

Bike

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

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

26/92

26

Introducing Interfaces (1/2)

- `Interface` groups declarations of similar capabilities of different classes together
- Looks like a totally stripped-down class declaration, with just method declarations:
-
```
public interface Transporter {
    public void move();
    //other common methods (brake, steer…)
 }
```
- `Car`s and `Bike`s can "implement" a `Transporter` interfac
  - they can transport people from one place to another
  - they "act as" transporters
    - can move (and brake, steer…)
  - for this lecture, interfaces are green and classes that implement them are pink

Car

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

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

Bike

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

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

27/92

27

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?

28/92

28

Models Similarities while Ensuring Consistency (1/3)

Let's break that down into two parts:

1) Model Similarities

2) Ensure Consistency

29/92

29

Models Similarities while Ensuring Consistency (2/3)

- How does this help our program?
- We know `Car`s and `Bike`s both need to move
  - i.e., should both have some `move()` method
  - let compiler know that too!
- Make the `Transporter` interface
  - what methods should the `Transporter` interface declare? Similarities!
    - `move()` (plus `brake`, `steer`...)
  - compiler ensures consistency--doesn't care how method is defined, just that it has been defined
  - general tip: methods that interface declares should model functionality **all** implementing classes share

30/92

30

Declaring an Interface (1/3)

What does this look like?

```
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 define actual implementation of any declared methods

31/92

31

Declaring an Interface (2/3)

What does this look like?

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

- Interfaces are only contracts, not classes that can be instantiated
- Interfaces can only declare methods – not define them
- Notice: method declaration end with semicolons, not curly braces!

32/92

32

Declaring an Interface (3/3)

What does this look like?

```
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`

33/92

33

Outline

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

34

Implementing an Interface (1/6)

```
public class Car implements
Transporter {

    public Car() {
        // constructor
    }

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

}
```

- Let's modify `Car` to implement `Transporter`
  - declare that `Car` "acts-as" `Transporter`
- Add `implements Transporter` to class declaration
- Promises compiler that `Car` will define all methods in `Transporter` interface
  - i.e., `move()`

35

Implementing an Interface (2/6)

```
public class Car implements
Transporter {

    public Car() {
        // constructor
    }

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

}
```

"Error: Car does not override method move() in Transporter" *

- Will this code compile?
  - nope :(
- Never implemented `move()` – `drive()` doesn't suffice. Compiler will complain accordingly

*Note: the full error message is "`Car is not abstract and does not override abstract method move() in Transporter`." We'll get more into the meaning of abstract in a later lecture.

36

Implementing an Interface (3/6)

```
public class Car implements
Transporter {

    public Car() {
        // constructor
    }

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

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

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

37

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

38

Implementing an Interface (5/6)

```
public class Car implements Transporter {

    //previous code elided

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

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

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

39

Implementing an Interface (6/6)

- As with signing multiple contracts, classes can implement multiple interfaces
  - "I signed my rent agreement, so I'm a renter, but I also signed my employment contract, so I'm an employee. I'm the same person."
  - what if I wanted `Car` to be able to change color as well?
  - create a `Colorable` interface
  - add that interface to `Car`'s class declaration
- Class implementing interfaces must define every single method from each interface

```
public interface Colorable {

    public void setColor(Color c);
    public Color getColor();

}
```

```
public class Car implements Transporter, Colorable
{

    public Car(){ //body elided }
    //@Override annotation elided
    public void drive(){ //body elided }
    public void move(){ //body elided }
    public void setColor(Color c){ //body elided }
    public Color getColor(){ //body elided }
}
```


40

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`

41

TopHat Question

Can you instantiate an interface as you can a class?

A. Yes

B. No

42

TopHat Question

Can an interface define code for its methods?

A. Yes

B. No

43

TopHat Question

Which statement 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;
       }
   }
```


44

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 double click() {
    return this.changeXPosition(100.0);
}
```

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

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

D.
```
@Override
public void click() {
    this.changeXPosition(100.0);
}
```


45

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
}
```


46

Leveraging Interfaces

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

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

→

Racer
useTransportation(…)

47

Outline

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

48

Introducing Polymorphism

- Poly = many, morph = forms
- A way of coding generically
  - way of referencing multiple classes sharing abstract functionality as acting as one generic type
    - cars and bikes can both `move()` → refer to them as classes of type `Transporter`
    - phones and Teslas can both `getCharged()` → refer to them as class of type `Chargeable`, i.e., classes that implement `Chargeable` interface
    - cars and boomboxes can both `playRadio()` → refer to them as class of type `RadioPlayer`
- How do we write one generic `useTransportation(…)` method?

49

What would this look like in code?

```
public class Racer {

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

}
```

*This is polymorphism! `transportation` instance passed in could be instance of `Car`, `Bike`, etc., i.e., of any class that **implements** the interface*

50

Let's break this down

There are two parts to implementing polymorphism:

1. Actual vs. Declared Type
2. Method resolution

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

```
public class Racer {

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

}
```

which `move()` is executed?

51

Actual vs. Declared Type (1/2)

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

```
Transporter 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`?

52

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?
  `transportation.playRadio()`

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

~~`transportation.playRadio();`~~

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

53

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`

54

Only Declared Type's Methods Can be Used

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

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

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


55

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()`

56

Polymorphism in Parameters

- What are implications of this method declaration?

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

- `useTransportation` will accept any class that implements `Transporter`
- we say that `Transporter` is the (declared) type of the parameter
- we can pass in an instance of any class that implements the `Transporter` interface
- `useTransportation` can only call methods declared in `Transporter`

57

Is this legal?

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

```
Transporter anastasiosBike = new Bike();
this.anastasio.useTransportation(anastasiosBike);

Car lexisCar = new Car();
this.lexi.useTransportation(lexisCar);

Radio lexisRadio = new Radio();
this.lexi.useTransportation(lexisRadio);
```

Even though `lexisCar` is declared as a `Car`, the compiler can still verify that it implements `Transporter`

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

58

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()`

59

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`

60

Method Resolution: Which move() is executed?

- Consider this line of code in Race class:

```
this.anastasio.useTransportation(new Bike());
```

- Remember what useTransportation method looks like:

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

What is "actual type" of transportation in
this.anastasio.useTransportation(new Bike()); ?

61

Method Resolution (1/4)

```
public class Race {

    private Racer anastasio;
    //previous code elided

    public void startRace() {
        this.anastasio.useTransportation(new Bike());
    }
}
```

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

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

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

62

Method Resolution (2/4)

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

- anastasio is a Racer
- Bike's move() method gets used
- Why?
  - Bike is the actual type of this Transporter
    - compiler will execute methods defined in Bike class
  - Transporter is the declared type
    - compiler limits methods that can be called to those declared in Transporter interface

63

Method Resolution (3/4)

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

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

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

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

64/92

64

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

65/92

65

TopHat Question

Given the following class:

```
public class Laptop implements Typeable, Clickable { //two interfaces
    public void type() {
      // code elided
    }
    public void click() {
        //code elided
}
```

Given that `Typeable` has declared the `type()` method and `Clickable` has declared the `click()` method, which of the following calls is valid?

A.
```
Typeable macBook = new Typeable();
macBook.type();
```

B.
```
Clickable macBook = new Clickable();
macBook.type();
```

C.
```
Typeable macBook = new Laptop();
macBook.click();
```

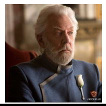
D.
```
Clickable macBook = new Laptop();
macBook.click();
```


66/92

66

Why does polymorphism work when calling methods?

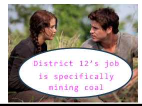
- Declared type and actual type work together
  - declared type keeps things generic
    - can reference many classes using one generic type
  - actual type ensures specificity
    - when calling declared type's method on an instance, the *actual* code that is called is the code defined in the *actual*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



Every district do their job!

Declared

Actual



When to use polymorphism?

- Do you use only functionality declared in interface OR do you need specialized functionality from implementing class?
  - if only using functionality from the interface → polymorphism!
  - if need specialized methods from implementing class, don't use polymorphism
- If defining `goOnScenicDrive()`…
  - want to put `topDown()` on `Convertible`, but not every `Car` can put top down
    - don't use polymorphism, not every `Car` can `goOnScenicDrive()` i.e., can't code generically

Why use interfaces?

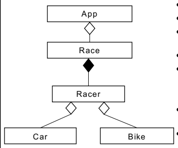
- Contractual enforcement
  - will guarantee that class has certain capabilities
    - `Car` implements `Transporter`, therefore it must know how to `move()`
- Polymorphism
  - can have implementation-agnostic classes and methods
    - know that these capabilities exist, don't care how they're implemented
    - allows for more generic programming
      - `useTransportation` can take in any instance of type `Transporter`
      - can easily extend this program to use any form of transportation, with minimal changes to existing code
    - a tool for extensible programming
      - How?

Why is this important?

- Using more than 2 methods of transportation?
- Old Design:
  - need more classes → more specialized methods (`useCar()`, `useBike()`, `useRollerblades()`, etc.)
- New Design:
  - as long as the new classes implement `Transporter`, `Racer` doesn't care what transportation it has been given
  - don't need to change `Racer`!
    - less work for you!
    - just add more transportation classes that implement `Transporter`
    - "need to know" principle, aka "separation of concerns"

70

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 `lexi`, a `Racer` and `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()`
- `useTransportation(new Bike())` initializes a `Bike` and calls `Bike`'s `move()` method which calls `this.pedal()`

71

The Program

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

```java
public class Race {
    private Racer lexi, anastasio;

    public Race() {
        this.lexi = new Racer();
        this.anastasio = new Racer();
    }

    public void startRace() {
        this.lexi.useTransportation(new Car());
        this.anastasio.useTransportation(new Bike());
    }
}
```

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

```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() { // @Override elided
       this.drive();
   }
}
```

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


72

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

73/92

73

Announcements

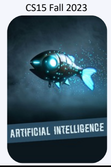
- TicTacToe released today (9/26)
  - Early hand-in: 9/28
  - On-time hand in: 9/30
  - Late hand-in: 10/2
- Class Relationships Section
  - Mini Assignment due before section
  - Email answers to your section TA
- CS15 Mentorship
  - Officially begun!
- T-Shirt Contest!!!!!
  - Designs due **Thursday before Lecture**!! (looking at you RISD students :D)

74/92

74

Topics in Socially Responsible Computing

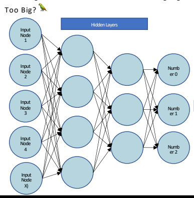
CS15 Fall 2023

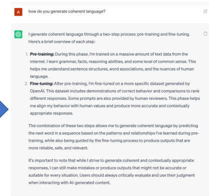


75

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 do you generate coherent language?

I generate coherent language through a two-step process: pre-training and fine-tuning. Here's a brief overview of each step:

1. **Pre-training:** During this phase, I'm trained on a massive amount of text data from the internet. I learn grammar, facts, reasoning abilities, and some level of common sense. This helps me understand sentence structures, word associations, and the nuances of human language.
2. **Fine-tuning:** After pre-training, I'm fine-tuned on a more specific dataset generated by OpenAI. This dataset includes demonstrations of correct behavior and comparisons to rank different responses. Some prompts are also provided by human reviewers. This phase helps me align my behavior with human values and produce more accurate and contextually appropriate responses.

The combination of these two steps allows me to generate coherent language by predicting the next word in a sequence based on the patterns and relationships I've learned during pre-training, while also being guided by the fine-tuning process to produce outputs that are more reliable, safe, and relevant.

It's important to note that while I strive to generate coherent and contextually appropriate responses, I can still make mistakes or produce outputs that might not be accurate or suitable for every situation. Users should always critically evaluate and use their judgment when interacting with AI-generated content.

76

How to Train Your ~~Dragon~~ LLM

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

0

Dataset	Quantity (tokens) (1)	Weight in training mix
Common Crawl (filtered)	[illegible]10 billion	60%
WebText2	19 billi[illegible]	22%
Books1	12 billio[illegible]	8%
Books2	55 [illegible]llion	8%
Wikipedia	3 [illegible]llion	3%

1

Source: (Brown et al. 2020)

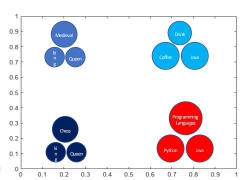
2

(1) Tokens ≅

77

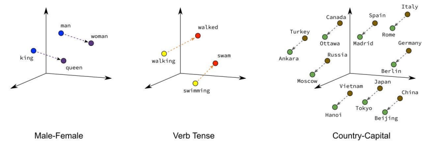
Word Embeddings

- 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 (100+ dimensions)
- Consider instead the following 2D example:
  - king and queen are semantically similar
    - But are we talking about Medieval History or Chess?
  - Likewise, Java could refer to a programming language or a drink
- Chat GPT maps these relationships so well that it seems to always know the exact context in which words are used



78

Word Embeddings (Continued)

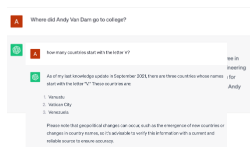


Following Training: King – Man + Woman = Queen

79

Garbage in, Garbage Out (GIGO)

- Flawed inputs creates flawed outputs
- "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" - Passages from the Life of a Philosopher (1864!!)
- Since Chat GPT is trained on false or incorrect statements, it will (confidently) produce flawed outputs
  - This is why Chat GPT appears to hallucinate sometimes…
- Chat GPT may also appear politically biased
  - Biased training material -> biased outputs



80

Fine Tuning

- To reduce incorrect and biased outputs and tailor the model towards specific tasks, 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 Chat-GPT 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!

Algorithmic Pre-Training (takes months!) — GPT with 1 Trillion Parameters

Human Fine Tuning (less-time but takes human Input) — GPT with 1 Trillion Fine Tuned Parameters

81

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
- CS1460 – Computational Linguistics
- CS1470 – Deep Learning
- CS1951A – Data Science

82

Introducing GPTA!

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

83

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 GenAI Usage Doc
- You'll see these guidelines every time you sign in to GPTA
- We have a user guide and usage guidelines on the Collab Policy and the GenAI Usage Doc

84

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 (ie, '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

85

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
  - Just 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!

86