

CSE201: Monsoon 2020, CSE Section  
Advanced Programming

# **Lecture 05: Interfaces and Polymorphism**

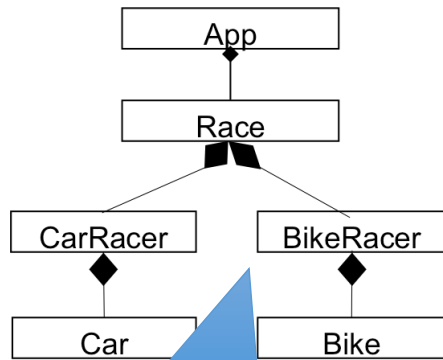
Vivek Kumar

Computer Science and Engineering

IIT Delhi

[vivekk@iiitd.ac.in](mailto:vivekk@iiitd.ac.in)

# Last Lecture



Do we need two different Racer classes??

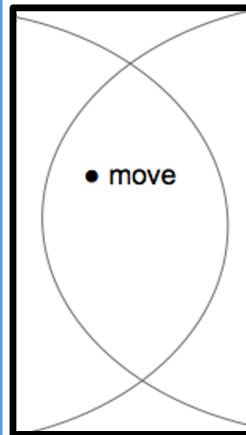
- Imagine this program:
  - Sophia and Dan are racing from their home to city center
    - whoever gets there first, wins!
    - catch: they don't get to choose their method of transportation
- Design a program that
  - assigns mode of transportation to each racer
  - starts the race
- For now, assume transportation options are **Car** and **Bike**

I am skipping this slide as you already have the recording of previous lecture

How about one Racer class with different methods?

```
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...
}
```

Any similarities?



## Interfaces in Java

- Group similar capabilities/function of different classes together
- Interfaces can only declare methods - not define them
- Interfaces are contracts that classes agree to
- If classes choose to **implement** given interface, it must define all methods declared in interface
  - if classes don't implement one of interface's methods, the compiler raises error

Declaring an Interface

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

Implementing an Interface

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

**@Override** is an annotation – a signal to the compiler (and to anyone reading your code)

# This Lecture

- Interfaces and Polymorphism

Slide acknowledgements: CS15, Brown University

# Back to the Race

- Let's make transportation classes use an interface

```
public class Car implements Transporter{

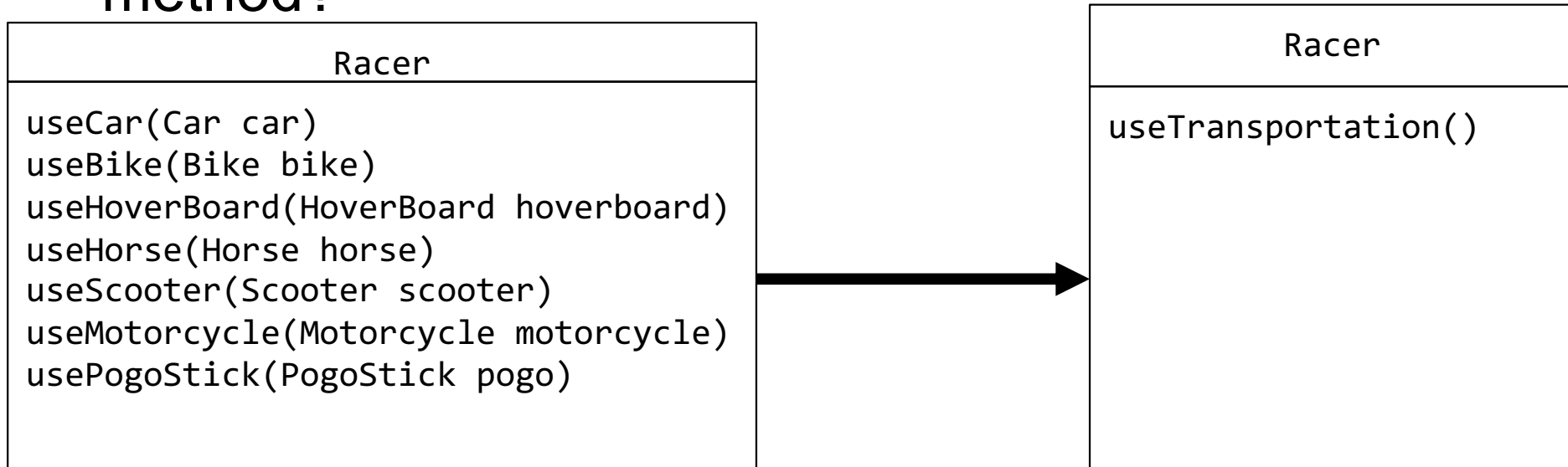
    public Car() {
        //code elided
    }
    public void drive(){
        //code elided
    }
    @Override
    public void move() {
        this.drive();
    }
    //more methods elided
}
```

```
public class Bike implements Transporter{

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

# Leveraging Interfaces

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




# Introducing Polymorphism

- Poly = many, morph = forms
- A way of coding **generically**
  - way of referencing many related objects as one generic type
    - cars and bikes can both `move()` → refer to them as `Transporter` objects
    - phones and camera can both `getCharged()` → refer to them as `Chargeable` objects, i.e., objects that implement `Chargeable` interface
    - cars and mobile phones can both `playRadio()` → refer to them as `RadioPlayer` objects
- How do we write one generic `useTransportation()` method?

# What would this look like in code?

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



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

# Let's break this down.

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

1. Actual vs. Declared Type
2. Method resolution



# Actual vs. Declared Type (1/2)

- Consider following piece of code:

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

- ...is that legal?
  - doesn't Java do strict type checking? (type on LHS = type on RHS)
  - how can instances of `Car` get stored in `Transporter` variable?

# Actual vs. Declared Type (2/2)

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

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

Nope. The **playRadio()** method is not declared in **Transporter** interface, therefore Java does not recognize it as viable method call

# Determining the Declared Type

- What methods do **Car** and **Bike** have in common?
  - `move()`
- How do we know that?
  - they implement **Transporter**
    - guarantees that they have `move()` method
- Think of **Transporter** like the “lowest common denominator”
  - it's what all transportation classes will have in common

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

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

# Is this legal?

`Transporter sophiasBike = new Bike();` ✓

`Transporter sophiasCar = new Car();` ✓

`Transporter sophiasRadio = new Radio();` ✗

`Radio` wouldn't implement `Transporter`. Since `Radio` cannot “act as” a `Transporter`, you cannot treat it as `Transporter`.

# Motivations for Polymorphism

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

# Polymorphism in Parameters

- What are implications of this method declaration?

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

- `useTransportation` will accept any object that implements `Transporter`
- `useTransportation` can only call methods declared in `Transporter`

# Is this legal?

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



```
Car sophiasCar = new Car();  
_sophia.useTransportation(sophiasCar);
```



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



Even though `sophiasCar` 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 like a `Transporter` object.

# Why move()? (1/2)

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

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



# Why move()? (2/2)

- Only have access to **Transporter** object
  - cannot call `transportation.drive()` or `transportation.pedal()`
    - that's okay, because all that's needed is `move()`
  - limited to the methods declared in **Transporter**

# Method Resolution: Which move() is executed?

- Consider this line of code in **Race** class:

```
_sophia.useTransportation(new Bike());
```

- Remember what **useTransportation** method looked like

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

What is “actual type” of **transportation** in this method invocation?

# Method Resolution (1/4)

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

---

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

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

# Method Resolution (2/4)

```
public class Race {  
    //previous code elided  
    public void startRace() {  
        _sophia.useTransportation(new Bike());  
    }  
}
```

---

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

---

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

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

# Method Resolution (3/4)

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

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

# Method Resolution (4/4)

- This method resolution is example of **dynamic binding**, which is when actual method implementation used is not determined until runtime
  - contrast with **static binding**, in which method gets resolved at compile time
- **move()** method is bound dynamically – Java does not know which **move()** method to use until program runs
  - same “**transport.move()**” line of code could be executed indefinite number of times with different method resolution each time

# Clicker Question

Given the following class:

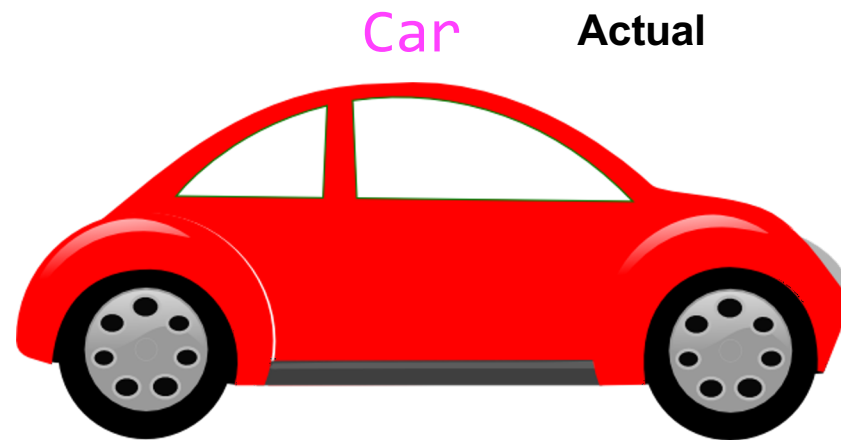
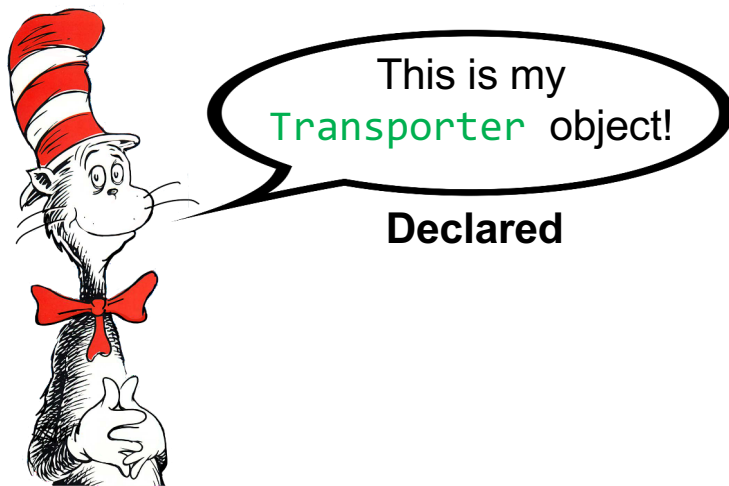
```
public class Laptop implements Typeable, Clickable {  
    public void type() {  
        // code elided  
    }  
    public void click() {  
        //code elided  
    }  
}
```

Given that typeable has declared the type method and clickable has declared the click method, which of the following calls is/are **valid**?

- A.   Typeable macBook= new Typeable();  
      macBook.type();
- B.   Clickable macBook = new Clickable();  
      macBook.type();
- C.   Typeable macBook= new Laptop();  
      macBook.click();
- D.   Clickable macBook = new Laptop();  
      macBook.click();

# Why does that work? (1/2)

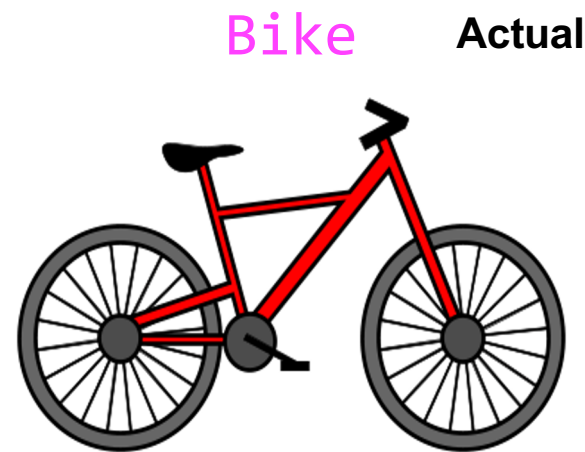
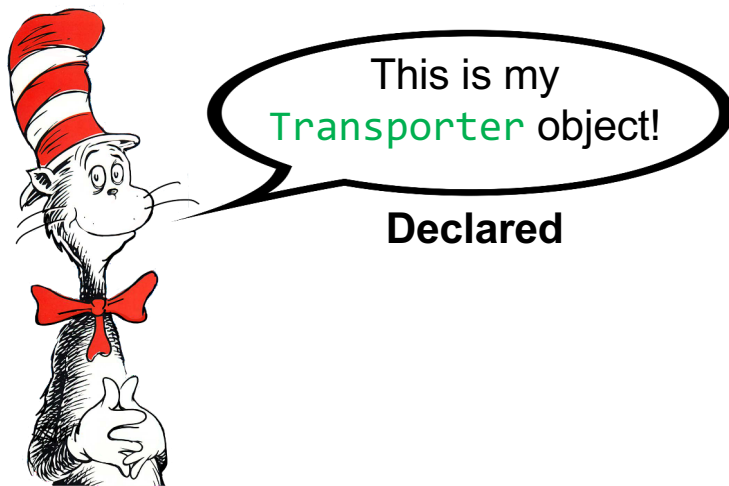
- **Declared type** and **actual type** work together
  - **declared type** keeps things generic
    - can reference a lot of objects using one generic type
  - **actual type** ensures specificity
    - when defining implementing class, the methods can get implemented in any way





# Why does that work? (2/2)

- **Declared type** and **actual type** work together
  - **declared type** keeps things generic
    - can reference a lot of objects using one generic type
  - **actual type** ensures specificity
    - when defining implementing class, the methods can get implemented in any way



# When to use polymorphism?

- Using only functionality declared in interface or specialized functionality from implementing class?
  - if only using functionality from the interface → polymorphism!
  - if need specialized methods from implementing class, don't use polymorphism

# 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 capability exists, don't care how they're implemented
  - allows for more generic programming
    - `useTransportation` can take in any `Transporter` object
    - can easily extend this program to use any form of transportation, with minimal changes to existing code
  - an extremely powerful tool for extensible programming

# Why is this important?

- With 2 modes of transportation!
- Old Design:
  - need more classes → more specialized methods (`useRollerblades()`, `useBike()`, 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`

# The Program

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

---

```
public class Race {  
    private Racer _dan, _sophia;  
  
    public Race(){  
        _dan = new Racer();  
        _sophia = new Racer();  
    }  
    public void startRace() {  
        _dan.useTransportation(new Car());  
        _sophia.useTransportation(new Bike());  
    }  
}
```

---

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

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

---

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

---

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

# In Summary

- Interfaces are contracts
  - force classes to define certain methods
- Polymorphism allows for extremely generic code
  - treats multiple classes as their “generic type” while still allowing specific method implementations to be executed
- Polymorphism + Interfaces
  - generic coding
- Why is it helpful?
  - want you to be the laziest (but cleanest) programmer you can be

# Next Lecture

- Inheritance and polymorphism
- Assignment-1 to be announced today evening
  - All assignments (including this) are individual implementations (no group assignments)
  - We are very strict about plagiarism. IIITD plagiarism policy will be strictly followed
  - Deadline Sunday midnight
  - Make sure to upload your solution before the deadline. We will not evaluate any late submissions (zero marks for late submission)