

CSE201: Monsoon 2022
Advanced Programming

Lecture 04: Interfaces in Java

Raghava Mutharaju (Section-A)

CSE, IIIT-Delhi

raghava.mutharaju@iiitd.ac.in

Last Lecture

● Class relationships

- When writing a program, need to keep in mind “big picture”—how are different classes related to each other?

○ Association

- Class A and class B are **associated** if A “**knows about**” B, but B is not a component of A
- Class A holds a class level reference to class B

○ Composition

- Class A **contains** object of class B
- A **instantiate** B
- The death relationship
 - B is garbage collected when A gets garbage collected

○ Dependency

- Neither class A or class B “knows about” each other, nor one of them is a “component” of the other. However, if A requests a service from B then A is said to be dependent on B

```
class Cart {  
    private double price;  
    public void addProduct(Product P){  
        price+=P.getPrice();  
    }  
}
```

```
class Project {  
    private String name;  
    public boolean status() { ... }  
    .....  
}  
// Contractor's project keep changing  
class Contractor {  
    private Project currentProject;  
    public Contractor(Project proj) {  
        this.currentProject = proj;  
    }  
    public void setProject(Project proj){  
        this.currentProject = proj;  
    }  
}
```

```
class Project {  
    private String name;  
    public boolean status() { ... }  
    .....  
}  
// A manager is fixed for a project  
class Manager {  
    private Project project;  
    public Manager() {  
        this.project = new Project("ABC");  
    }  
    public boolean projectCompleted() {  
        return project.status();  
    }  
}
```

This Lecture

- Interfaces in Java
 - Declaring
 - Defining

Slide acknowledgements: CS15, Brown University

Recall: Declaring vs. Defining Methods

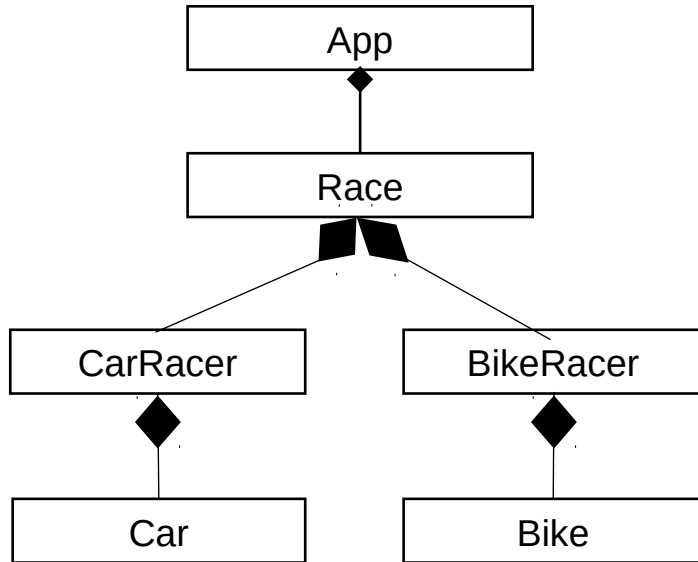
- What's the difference between **declaring** and **defining** a method?
 - method **declaration** is the scope (**public**), return type (**void**), name and parameters (**makeSounds()**)
 - method **definition** is the body of the method – the actual implementation (the code that actually makes the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

```
public class Dog {  
    //constructor elided  
  
    public void makeSounds() {  
        this.bark();  
        this.whine();  
        this.bark();  
    }  
    public void bark() {  
        //code elided  
    }  
    public void whine() {  
        //code elided  
    }  
}
```

Using What You Know

- 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

What does our design look like?



- 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**

Goal 1: Assign transportation to each racer

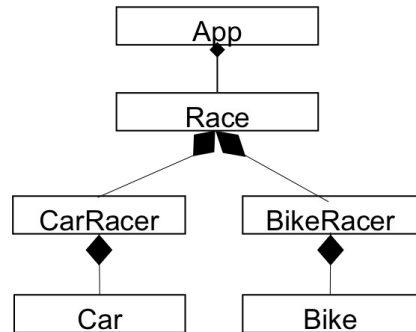
- Need transportation classes (something to give to racers)
- Let's use **Car** and **Bike** classes
- Both classes will need to describe how the transportation moves
 - **Car** needs **drive** method
 - **Bike** needs **pedal** method

Coding the project (1/4)

● Let's build transportation classes

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

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



Goal 1: Assign transportation to each racer

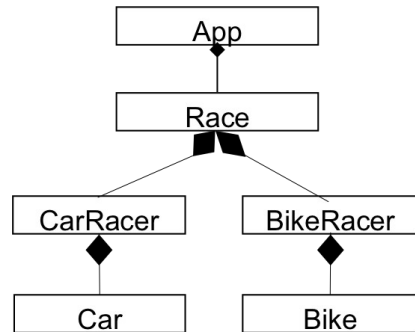
- Need racer classes that will use their type of transportation
 - `CarRacer`
 - `BikeRacer`
- What methods will we need? What capabilities should each `Racer` class have?
- `CarRacer` needs to know when to use the car
 - write `useCar()` method
- `BikeRacer` needs to know when to use the bike
 - write `useBike()` method

Coding the project (2/4)

● Let's build the racer classes

```
public class CarRacer {  
    private Car _car;  
  
    public CarRacer() {  
        _car = new Car();  
    }  
  
    public void useCar(){  
        _car.drive();  
    }  
    //more methods elided  
}
```

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



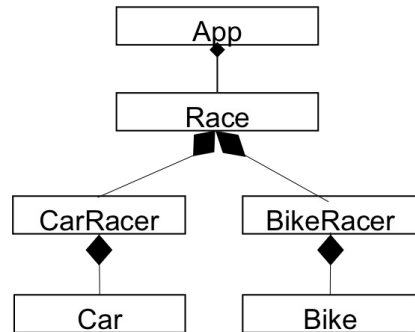
Goal 2: Tell the racers to start the race

- Race class contains Racers
 - App contains Race
- Race class will have `startRace()` method
 - `startRace()` tells each racer to use their transportation
- `startRace()` gets called in App

```
startRace:  
  Tell _dan to useCar  
  Tell _sophia to useBike
```

Coding the project (3/4)

- Let's build the Race class



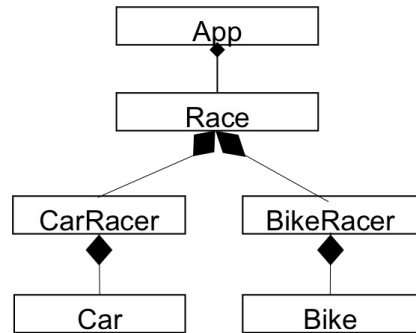
```
public class Race {
    private CarRacer _dan;
    private BikeRacer _sophia;

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

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

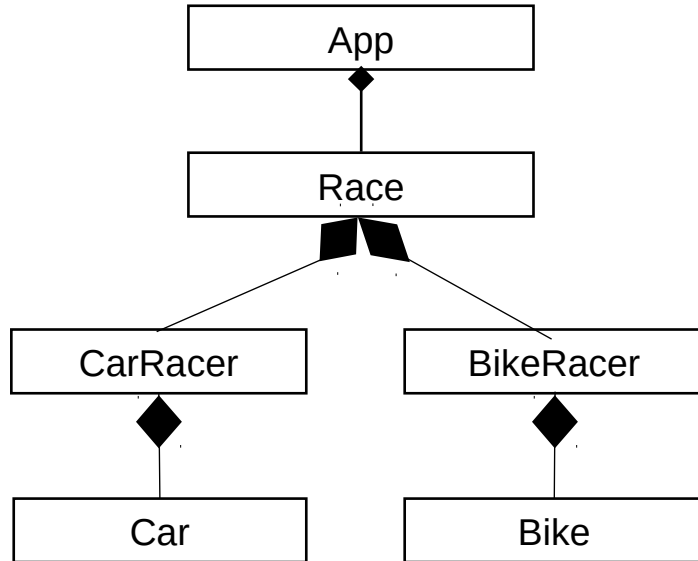
Coding the project (4/4)

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



- Now build the App class
- Now race to the city center!

Recap: What does our design look like?



How would this program run?

- An instance of **App** gets initialized
- **App**'s constructor initializes an instance of **Race**
- **Race**'s constructor initializes **_dan** (**CarRacer**) and **_sophia** (**BikeRacer**)
 - **CarRacer**'s constructor initializes a **_car** (**Car**)
 - **BikeRacer**'s constructor initializes a **_bike**
- App calls **race.startRace()**
- **race** calls **_dan.useCar()** and **_sophia.useBike()**
- **_dan** calls **_car.drive()**
- **_sophia** calls **_bike.pedal()**

Can we do better?

Things to think about

- Do we need two different **Racer** classes?
 - Want multiple instances of **Racers** that use different modes of transportation
 - But how?

Solution 1: Create one Racer class with methods!

- Create one **Racer** class
 - define different methods for each type of transportation
- dan is instance of **Racer** and elsewhere we have:

```
Car dansCar = new Car();  
_dan.useCar(dansCar);
```

- **Car**'s **drive()** method will be invoked
- But any given instance of **Racer** will need a new method to accommodate every kind of transportation!

```
public class Racer {  
    public Racer(){  
        //constructor  
    }  
  
    public void useCar(Car myCar){  
        myCar.drive();  
    }  
  
    public void useBike(Bike myBike){  
        myBike.pedal();  
    }  
}
```

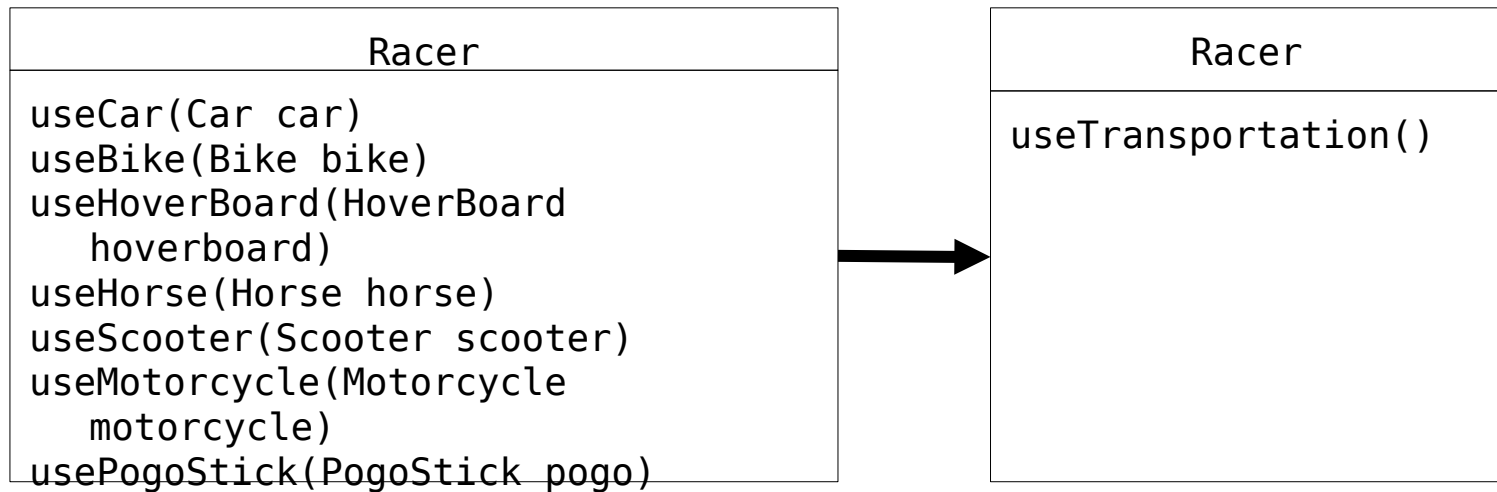
Question: What is the relationship between Racer+Car and Racer+Bike?

Solution 1 Drawbacks

- Now imagine 10 people join the race and so there are 10 different modes of transportation
- Writing these similar `useType()` methods are a lot of work for you, the developer, and 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...  
}
```

Is there another solution?



- Can we go from left to right?

Interfaces: Spot the Similarities

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



Cars vs. Bikes

Cars

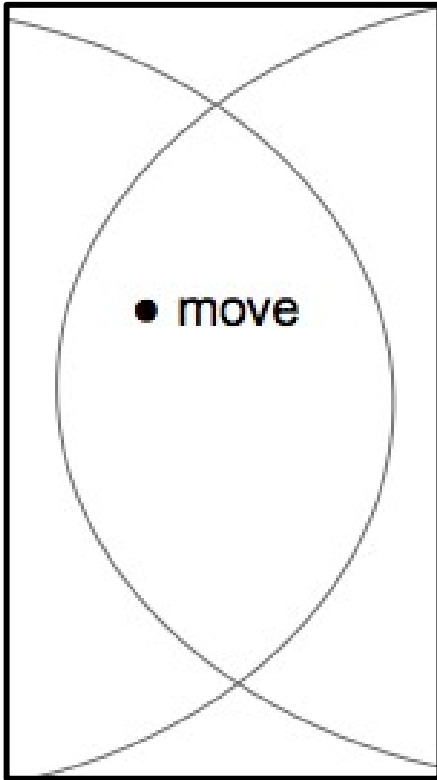
- Play radio
- Turn off/on headlights
- Turn off/on turn signal
- Lock/unlock doors

● Move

Bikes

- Drop kickstand
- Change gears

Digging deeper into the similarities



- How similar are they when they move?
 - do they move in same way?
- Not very similar
 - cars drive
 - bikes pedal
- Both can move, but in different ways

Can we model this in code?

- Many real-world objects have several broad similarities
 - cars and bikes can move
 - cars and laptops can play radio
- Take **Car** and **Bike** class
 - how can their similar functionalities get enumerated in one place?
 - how can their broad relationship get portrayed through code?

Car

- playRadio()
- lockDoors()
- unlockDoors()
- drive()

Bike

- dropKickstand()
- changeGears()
- pedal()

Introducing Interfaces

- **Interfaces** group similar capabilities/function of different classes together
- Model “acts-as” relationship
- **Cars** and **Bikes** could implement a **Transporter** interface
 - they can transport people from one place to another
 - “act as” transporters
 - objects that can move
 - have shared functionality, such as moving, braking, turning etc.
 - for this lecture, interfaces are **green** and classes that implement them **pink**

Introducing Interfaces

- Interfaces are contracts that classes agree to
- If classes choose to **implement** given interface, it must define all methods declared in interface
 - if classes don't implement one of interface's methods, the compiler raises error
 - later we'll discuss strong motivations for this contract enforcement
- Interfaces don't define their methods - implementing classes do
 - Interfaces **only** care about the fact that the methods get defined - not **how** – *implementation-agnostic*
- **Models similarities while ensuring consistency**
 - **What does this mean?**

Let's break that down

1) Models Similarities

2) Ensures Consistency

Models Similarities While Ensuring Consistency

- How does this help our program?
- We know **Cars** and **Bikes** both need to move
 - i.e., should all have some `move()` method
 - let compiler know that too!
- Let's make the **Transporter** interface!
 - what methods should the **Transporter** interface declare?
 - `move()`
 - only using a `move()` for simplicity, but `brake()`, etc. would also be useful
 - compiler doesn't care how method is defined, just that it's been defined
 - general tip: methods that interface declares **should model functionality all implementing classes share**

Declaring an Interface (1/4)

What does this look like?

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

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

Declaring an Interface (2/4)

What does this look like?

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

- Declare it as **interface** rather than class

Declaring an Interface (3/4)

What does this look like?

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

- Declare methods - the contract
- In this case, only one method required: `move()`
- All classes that sign contract (implement this interface) **must define actual implementation** of any declared methods

Declaring an Interface (4/4)

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!

Questions

Which line(s) of this program is incorrect?

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

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


Implementing an Interface (1/6)

Let's modify **Car**

```
public class Car implements
Transporter {

    public Car() {
        // constructor
    }

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

}
```

- Let's modify **Car** to implement **Transporter**
 - declare that **Car** “acts-as” **Transporter**
- Add **implements** **Transporter** to class declaration
- Promises compiler that **Car** will define all methods in **Transporter** interface
 - i.e., **move()**
- Will this code compile?

Implementing an Interface (2/6)

```
public class Car implements
Transporter {

    public Car() {
        // constructor
    }

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

}
```

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

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

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

Implementing an Interface (3/6)

```
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 arguments) **must match** how its declared in interface

Implementing an Interface (4/6)

What does **@Override** mean?

```
public class Car implements
Transporter {

    public Car() {
        // constructor
    }

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

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

- Include **@Override** right above the method signature
- **@Override** is an annotation – a signal to the compiler (and to anyone reading your code)
 - allows compiler to enforce that interface actually has method declared
 - more explanation of **@Override** in next lecture
- Annotations, like comments, have **no effect on how code behaves** at runtime

Implementing an Interface (5/6)

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

```
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  
}
```

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

Implementing an Interface (6/6)

- As with signing multiple contracts, classes can implement multiple interfaces
 - “I signed my rent agreement, so I'm a renter, but I also signed my employment contract, so I'm an employee. I'm the same person.”
 - what if I wanted **Car** to change color as well?
 - create a **Colorable** interface
 - add that interface to **Car**'s class declaration
- Implementing class must define **every single method** in each of its interfaces

```
public interface Colorable {  
  
    public void setColor(Color c);  
    public Color getColor();  
  
}  
  
public class Car implements Transporter, Colorable{  
  
    public Car(){ //body elided }  
    public void drive(){ //body elided }  
    public void move(){ //body elided }  
    public void setColor(Color c){ //body elided }  
    public Color getColor(){ //body elided }  
  
}
```

Summary

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

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.

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

C.

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

B.

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

D.

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


Next Lecture

- Interface and polymorphism