CSE201: Monsoon 2020, CSE Section
Advanced Programming

Lecture 06: Inheritance and Polymorphism

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

- Polymorphism in Java
  - A way of coding generically
    - way of referencing many related objects as one generic type

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

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

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

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

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

This Lecture

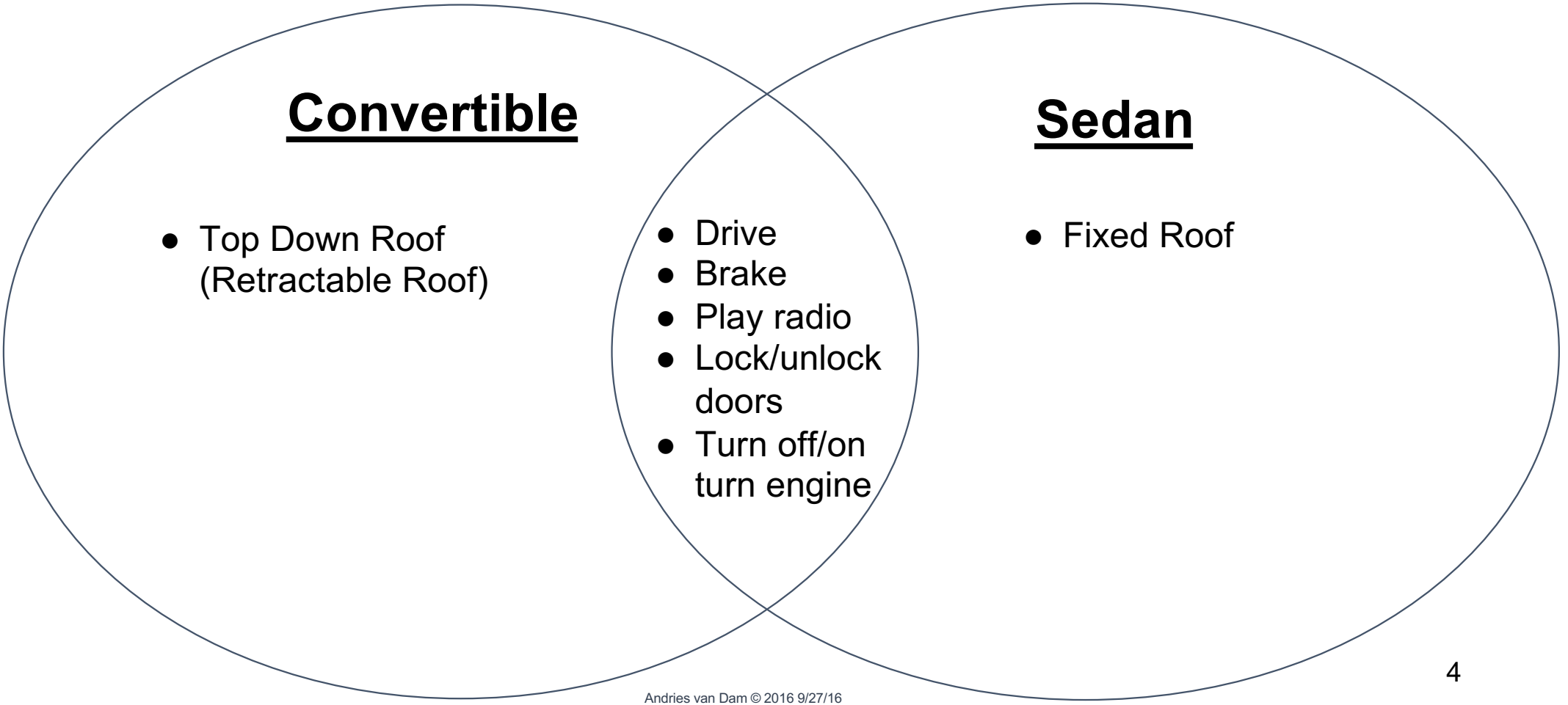
- Inheritance and Polymorphism

Slide acknowledgements: CS15, Brown University

Spot the Similarities

- What are the similarities between a convertible and a sedan?
- What are the differences?

Convertibles vs. Sedans



Can we model this in code?

- In some cases, objects can be very closely related to each other
  - Convertibles and sedans drive the same way
  - Flip phones and smartphones call the same way
- Imagine we have an `Convertible` and a `Sedan` class
  - Can we enumerate their similarities in one place?
  - How do we portray their relationship through code?

Convertible

- `putTopDown()`
- `turnOnEngine()`
- `turnOffEngine()`
- `drive()`

Sedan

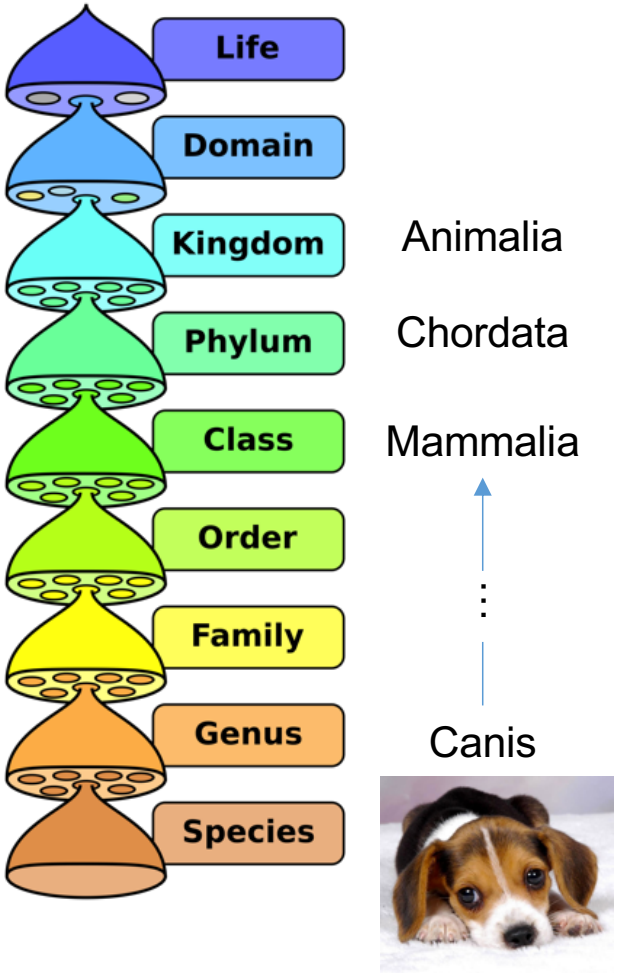
- `parkInCompactSpace()`
- `turnOnEngine()`
- `turnOffEngine()`
- `drive()`

Can we use Interfaces?

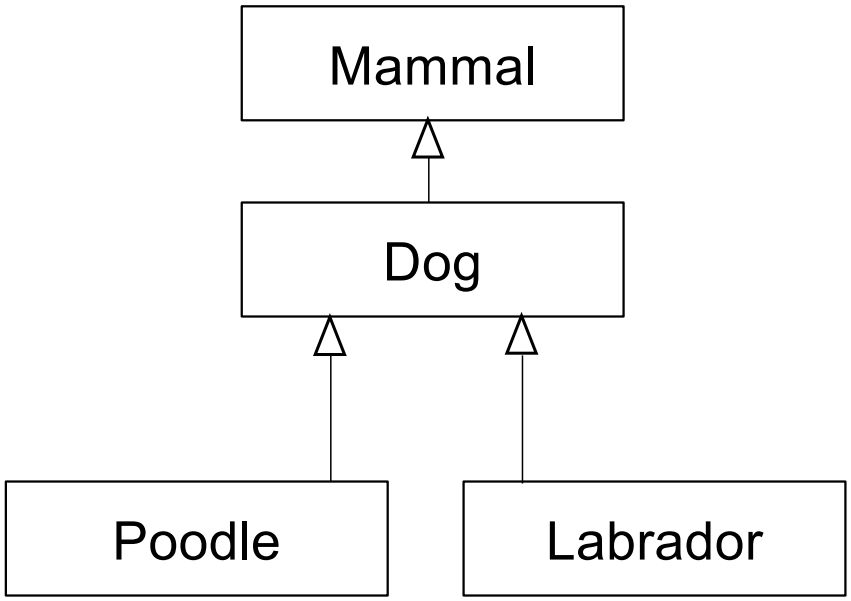
- We could build an interface to model their similarities
  - Build a `Car` interface with the following methods:
    - `turnOnEngine()`
    - `turnOffEngine()`
    - `drive()`
    - etc.
- Remember: interfaces only declare methods
  - Each class will need to implement the method in its own way
  - Thinking ahead: a lot of these method implementations would be the same across classes
    - `Convertible` and `Sedan` would have the same definition for `drive()`
      - `startEngine`, `shiftToDrive`, etc
- **Is there a better way where we can reuse the code?**

Inheritance

- In OOP, inheritance is a way of modeling very similar classes
- **Inheritance** models an **“is-a”** relationship
  - A sedan “is a” car
  - A dog “is a” mammal
- Remember: **Interfaces** model an **“acts-as”** relationship
- You’ve probably seen inheritance before!
  - Taxonomy from biology class

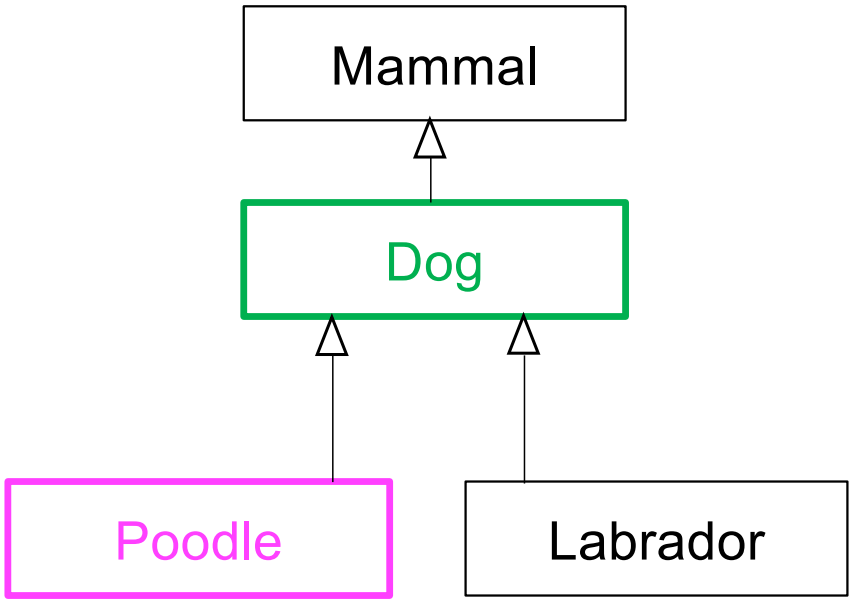


Modeling Inheritance (1/2)



- This is an inheritance diagram
  - Each box represents a class
- A Poodle “is-a” Dog, a Dog “is-a” Mammal
  - Transitively, a Poodle is a Mammal
- “Inherits from” = “is-a”
  - Poodle inherits from Dog
  - Dog inherits from Mammal
- This relationship is not bidirectional
  - A Poodle is a Dog, but not every Dog is a Poodle (could be a Labrador, a German Shepard, etc)

Modeling Inheritance (2/2)



- **Superclass/parent/base**: A class that is inherited from
- **Subclass/child/derived**: A class that inherits from another
- "A Poodle is a Dog"
  - Poodle is the subclass
  - Dog is the superclass
- A class can be both a superclass and a subclass
  - Ex. Dog
- In Java you can only inherit from one superclass (no multiple inheritance)
  - Other languages, like C++, allow for multiple inheritance

Motivations for Inheritance

- A subclass inherits all of its parent's **public** and **protected** capabilities
  - If `Car` defines `drive()`, `Convertible` inherits `drive()` from `Car` and drives the same way. This holds true for all of `Convertible`'s subclasses as well
- Inheritance and Interfaces both legislate class's behavior, although in very different ways
  - Interfaces allow the compiler to enforce method implementation
    - An implementing class will have all capabilities outlined in an interface
  - Inheritance assures the compiler that all **subclasses** of a **superclass** will have the superclass's public capabilities without having to respecify code – methods are inherited
    - A `Convertible` knows how to drive and drives the same way as `Car` because of inherited code
- Benefit of inheritance
  - Code reuse
    - If `drive()` is defined in `Car`, `Convertible` doesn't need to redefine it! Code is inherited
    - Only need to implement what is different, i.e. what makes `Convertible` special

Superclasses vs Subclasses

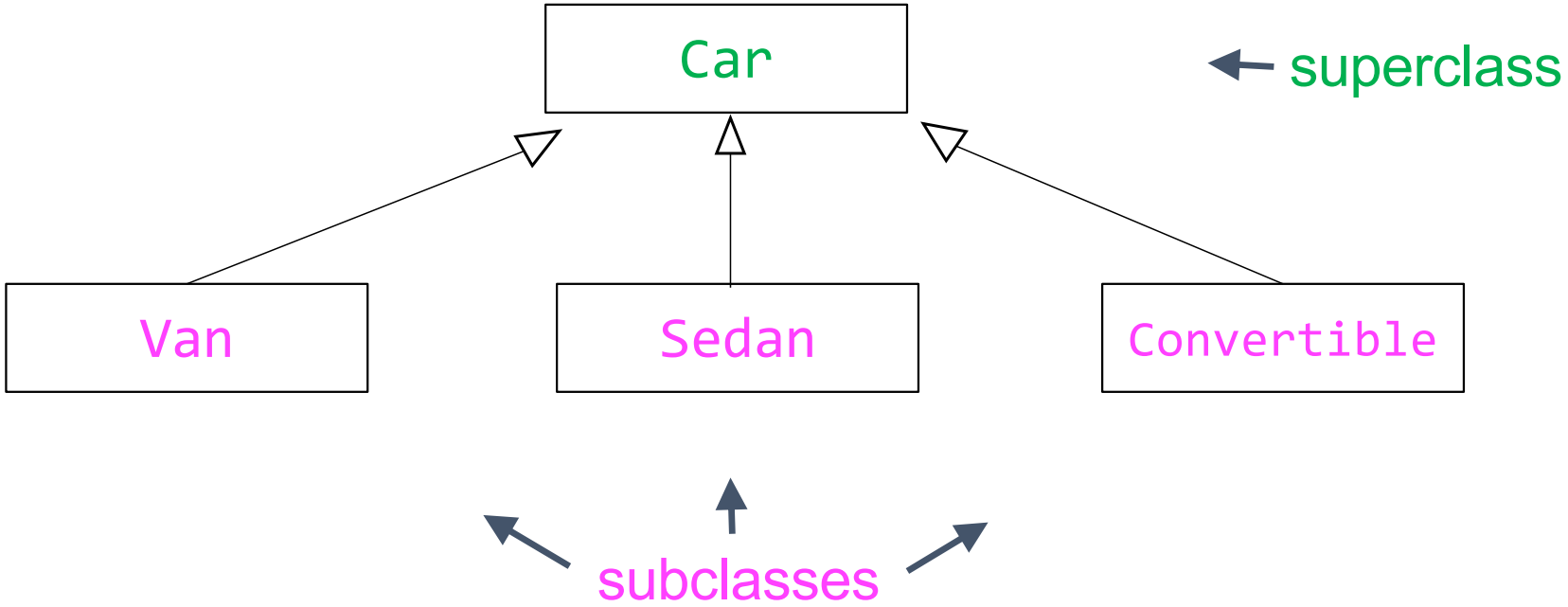
- A superclass factors out commonalities among its subclasses
  - describes everything that all subclasses have in common
  - `Dog` defines things common to all `Dog`s
- A subclass differentiates/specializes its superclass by:
  - **adding new methods:**
    - the subclass should define specialized methods. All `Animal`s cannot swim, but `Fish` can
  - **overriding inherited methods:** (more on this after few slides!)
    - a `Bear` class might override its inherited sleep method so that it hibernates rather than sleeping as most other `Animals` do
  - **defining "abstract" methods:** (next lecture!)
    - the superclass declares but does not define

Let's examine inheritance further

1. Model inheritance relationship
2. Adding new methods
3. Overriding methods

Modeling Inheritance

- Let's model a `Van`, a `Sedan`, and a `Convertible` class with inheritance!



Step 1: Define the superclass

- Defining `Car` is just like defining any other class

```
public class Car {
    private Engine _engine;
    //other variables elided
    public Car(){
        _engine = new Engine();
    }
    public void turnOnEngine() {
        _engine.start();
    }
    public void turnOffEngine() {
        _engine.shutOff();
    }
    public void cleanEngine() {
        _engine.steamClean();
    }
    public void drive() {
        //code elided
    }
    //more methods elided
}
```

Step 2: Define a subclass

- Notice the `extends` keyword
  - `extends` means “is a subclass of” or “inheriting from”
  - `extends` lets the compiler know that `Convertible` is inheriting from `Car`
  - Whenever you create a class that inherits from a superclass, must include “`extends <superclass name>`” in class declaration

```
public class Convertible extends Car {
    //code elided for now
}
```

Model Inheritance

- You can create any number of subclasses
  - `Sedan`, `Van`, `Convertible`, `SUV`...could all extend from `Car`
  - These classes will inherit public capabilities from `Car`
- Each subclass can only inherit from one superclass
  - `Convertible` cannot extend `Car`, `FourWheeledTransportation`, and `GasFueledTransportation`
  - Contrast with interfaces: you can implement as many interfaces as you want

Let's examine inheritance further

1. Model inheritance relationship
2. Adding new methods
3. Overriding methods

Adding new methods (1/2)

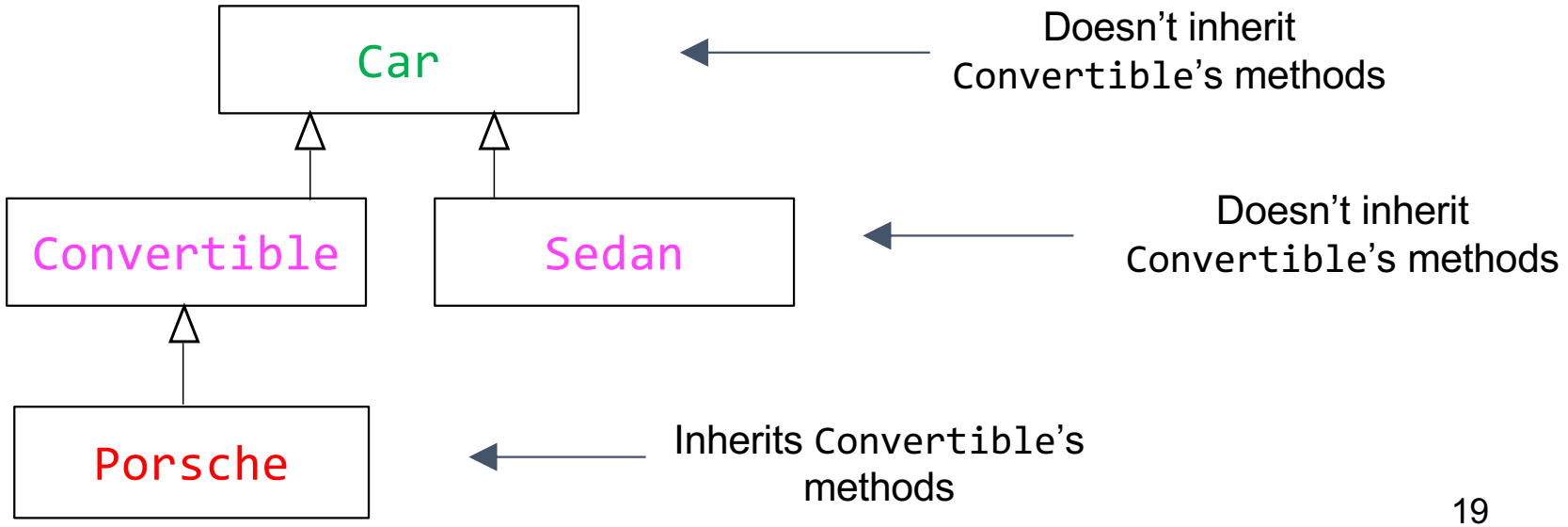
- Let's make a `Sedan` class that inherits from `Car`
- Let's make Convertible class that inherits from Car
- Can `Sedan` use `putTopDown()`?
  - Nope. That method is defined in `Convertible`, so only `Convertible` and `Convertible`'s subclasses can use it

```
public class Sedan extends Car {

    public Sedan (){

    }
    //other methods elided

}
```

```
public class Convertible extends Car {

    public Convertible(){

    }

    public void putTopDown(){
        //code elided
    }
}
```

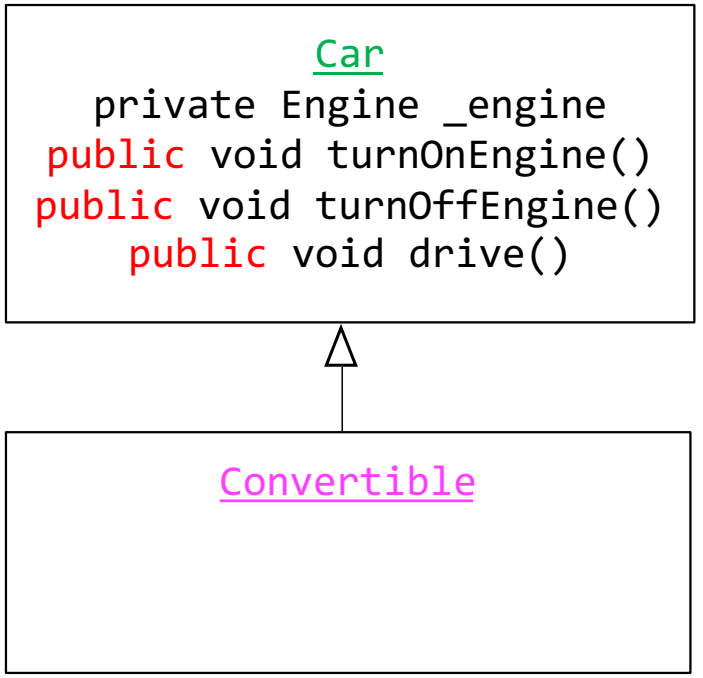
Adding new methods (2/2)

- You can add specialized functionality to a subclass by defining methods
- These methods can only be inherited if a class extends this subclass



What can subclasses access? (1/2)

- Remember: a subclass inherits any **public or protected** methods and variables from its superclass. Subclass cannot access any **private** field/method from superclass
- Before adding any code to `Convertible` class, what does `Convertible` already know how to do?
  - It can do anything a `Car` can do!
    - `turnOnEngine()`
    - `turnOffEngine()`
    - `drive()`



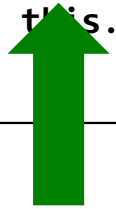
Note that we don't list the parent's public methods again here – they are implicitly inherited!

What can subclasses access? (2/2)

```
public class Car {
    private Engine _engine;
    //other variables elided
    public Car(){
        _engine = new Engine();
    }
    public void turnOnEngine() {
        _engine.start();
    }
    public void turnOffEngine() {
        _engine.shutOff();
    }
    public void drive() {
        //code elided
    }
    protected void cleanEngine() { ... }
}
```

```
public class Convertible extends Car {
    //constructor elided
    public void cleanCar() {
        _engine.steamClean();
    }
}
```

```
public class Convertible extends Car
{
    //constructor elided
    public void cleanCar() {
        this.cleanEngine();
    }
}
```



This makes use of *the parent's* inherited `cleanEngine` method, hence our use of **`this`**

- Will `Convertible` have access to _engine?
- Subclasses **cannot directly inherit private** variables / methods from parent
  - But you can can use methods defined in your parent, which have access to the variable

Question

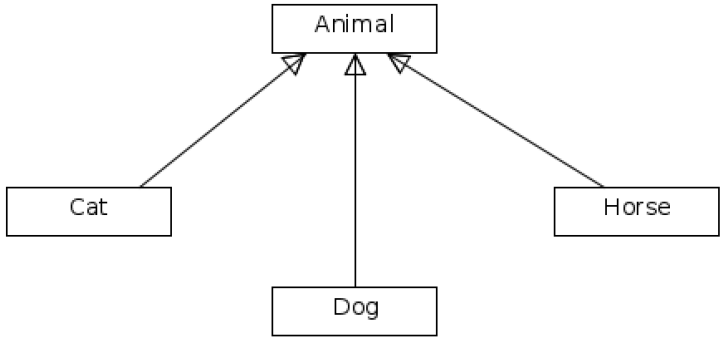
Which of the following is a superclass/parent of the rest?

A. Lions
B. Tigers
C. Cats
D. Leopards

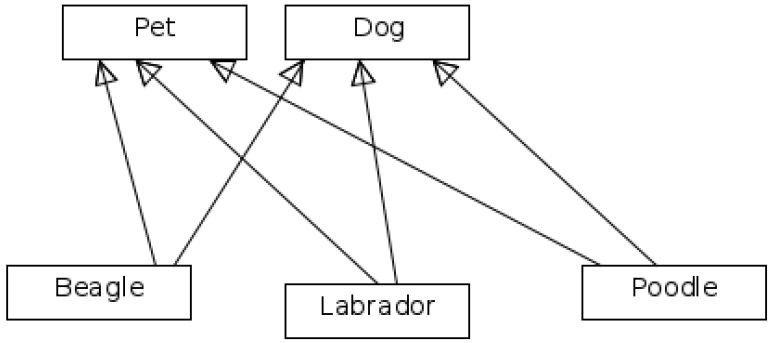
Question

All of the following are appropriate ways to model superclasses and subclasses EXCEPT:

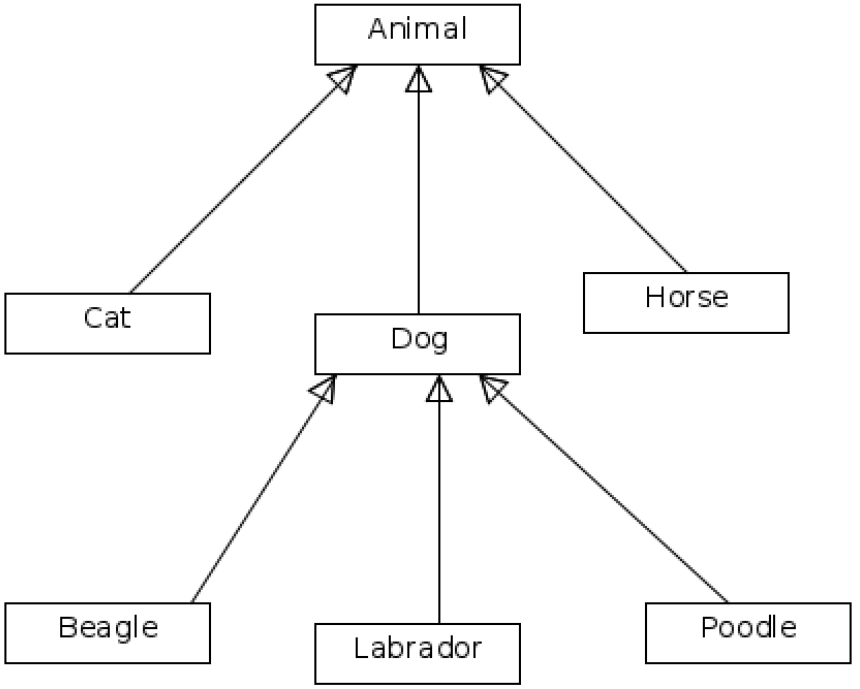
A.



B.



C.



Let's examine inheritance further

1. Model inheritance relationship
2. Adding new methods
3. Overriding methods

Overriding methods (1/3)

- A `Convertible` may decide `Car`'s `drive()` method just doesn't cut it
  - A `Convertible` drives much faster than a regular car
- Can **override** a parent class's method and redefine it

```
public class Car {

    private Engine _engine;
    //other variables elided

    public Car() {
        _engine = new Engine();
    }
    public void drive() {
        this.goFortyMPH();
    }
    public void goFortyMPH() {
        //code elided
    }
    //more methods elided
}
```

Overriding methods (2/3)

- `@Override` is an annotation--signals to compiler (and to anyone reading your code) that you're overriding a method of the superclass
  - We include `@Override` right before we declare method we mean to override

```
public class Convertible extends Car {

    public Convertible() {

    }

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

    public void goSixtyMPH(){
        //code elided
    }
}
```

Overriding methods (3/3)

- Here's where we re-declare method we want to override
  - Be careful – method signature must match that of the superclass's method exactly else Java will create a new additional method instead of overriding !
- `drive()` is the ***method signature***, indicating that name of method is drive and it takes in no parameters
  - When a `Convertible` is told to drive, it will execute this code instead of the code in its superclass's drive method

```
public class Convertible extends Car {

    public Convertible() {

    }

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

    public void goSixtyMPH(){
        //code elided
    }
}
```

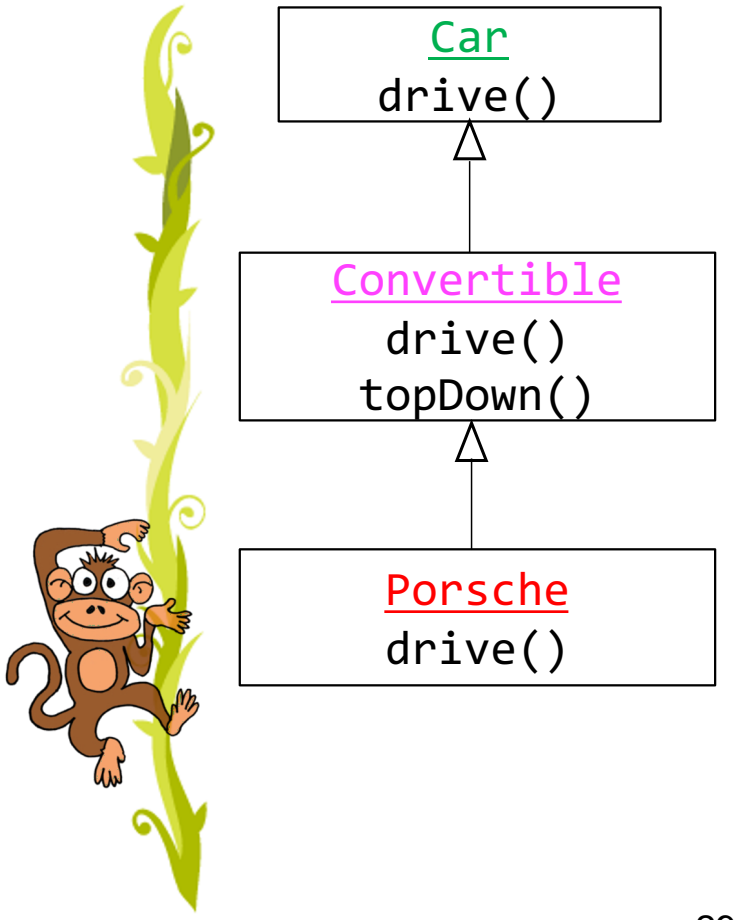
Partially overriding methods

- Keyword `super` used to invoke original inherited method from parent: in this case, drive as implemented in parent `Car`
- While you can use `super` to call other methods in the parent class, it's strongly discouraged
  - Use the **`this`** keyword instead
  - *Except* when you are calling the parent's method within the child's method of the same name
    - This is **partial overriding**
    - What would happen if we said `this.drive()` instead of `super.drive()`?

```
public class Sedan extends Car {

    public Sedan () {
          //code elided
    }

    @Override
    public void drive(){
        this.turnOnEngine();
        super.drive(); // super == parent
                                    class
        this.addPinToMap();
        super.drive();
        super.drive();
        this.addPinToMap();
    }

}
```

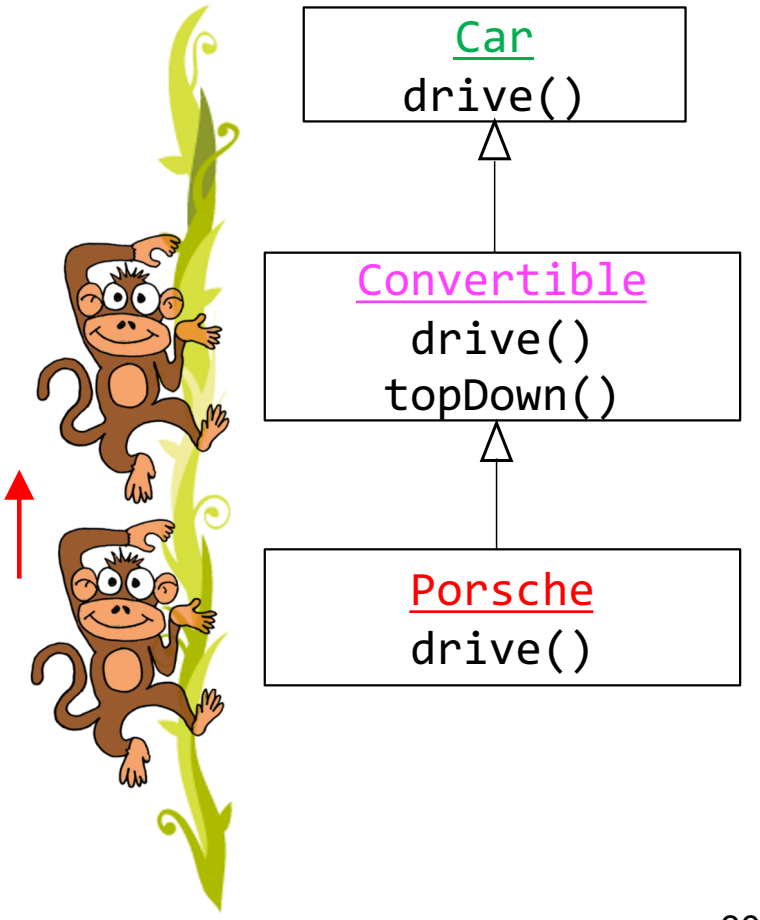
Method Resolution (1/2)

- When we call `drive()` on some instance of `Porsche`, how does Java know which version of the method to call?
- Essentially, Java "walks up the class inheritance tree" from subclass to superclass until it either:
  - finds the method, and calls it
  - doesn't find the method, and generates a compile-time error. You can't send a message for which there is no method!



Method Resolution (2/2)

- When we call `drive()` on a `Porsche`, Java executes the `drive()` method defined in `Porsche`
- When we call `topDown()` on a `Porsche`, Java executes the `topDown()` method defined in `Convertible`



Inheritance and Polymorphism (1/3)

- Let's borrow the Racer class from the example we discussed in lecture on interfaces
- However, we change the parameter type in method `useTransportation()` from `Transporter` to `Car`
- What would happen?
  - **We can only pass in `Car` and subclasses of `Car`**

```
public class Racer {

    //previous code elided

    public void useTransportation(Car myCar) {
        //code elided
    }

}
```

Inheritance and Polymorphism (2/3)

- Let's define `useTransportation()`
- What method should we call on `myCar`?
  - Every `Car` knows how to drive, which means we can guarantee that every subclass of `Car` also knows how to drive

```
public class Racer {

    //previous code elided

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

}
```

Is this legal?

```
Car convertible = new Convertible();
_sophia.useTransportation(convertible);
```

```
Car sedan = new Sedan();
_sophia.useTransportation(sedan);
```

```
Car bike = new Bike();
_sophia.useTransportation(bike);
```

`Bike` is not a subclass of `Car`, so you cannot treat an instance of `Bike` as a `Car`.

Inheritance and Polymorphism (3/3)

- That's all we needed to do!
- Our inheritance structure looks really similar to our interfaces structure
  - Therefore, we only need to change 2 lines in `Racer` in order to use any of our new cars!
  - But remember: what's happening behind the curtain is very different: method resolution "climbs up the hierarchy" for inheritance
- Polymorphism is an incredibly powerful tool
  - Allows for generic programming
  - Treat multiple classes as their generic type while still allowing specific method implementations to be executed
- Polymorphism+Inheritance is strong generic coding

Question

In the following code, the `Elephant` subclass extends the `Animal` superclass, both of which contain and define an `eat()` method:

```
Animal horton = new Elephant();
horton.eat();
```

Whose eat method is being called?

A. Animal
B. Elephant
C. Sedan
D. None of the above

Next Lecture (Tomorrow)

- Inheritance and polymorphism (continued)
- Immutable classes
- Abstract classes