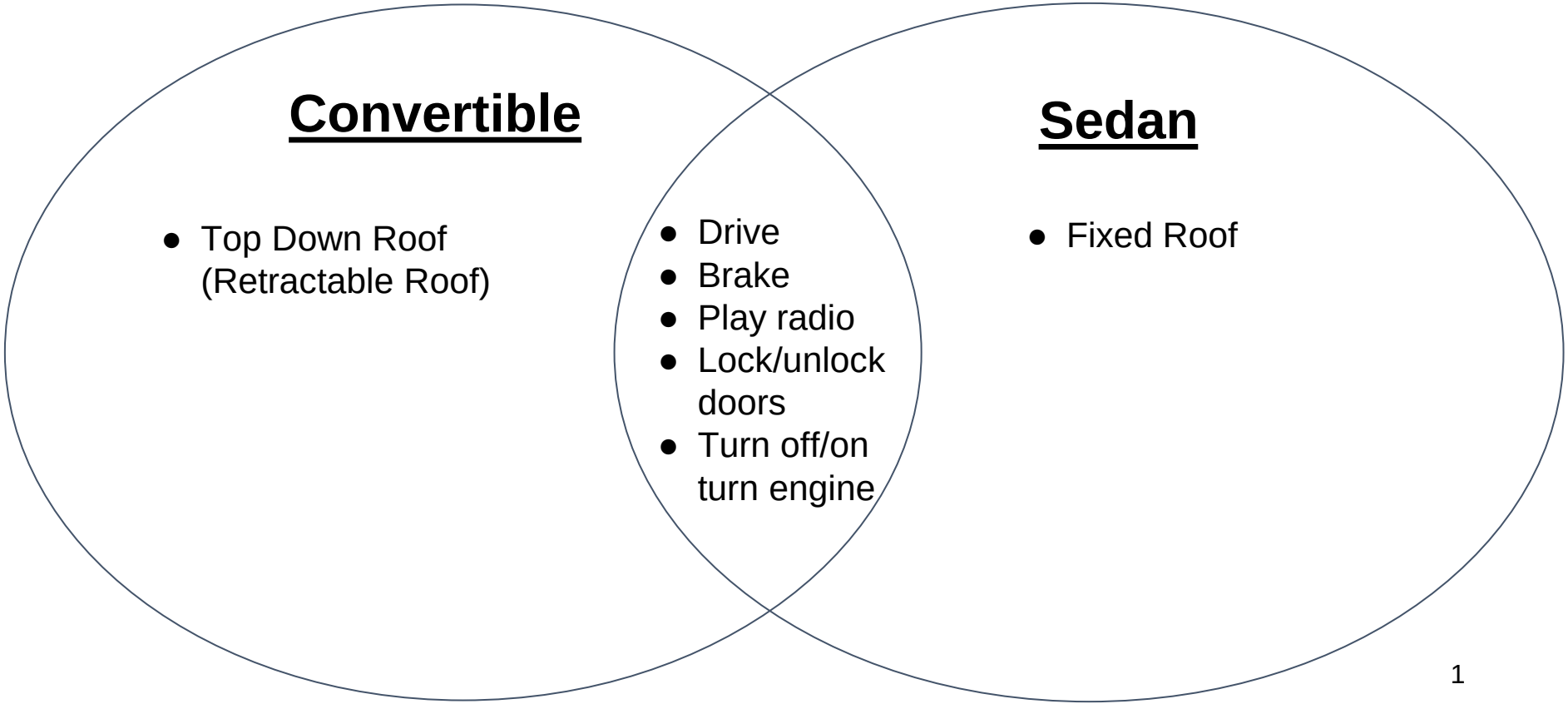
CSE201: Monsoon 2024
Advanced Programming

# Lecture 06: Inheritance - Abstract Class and Immutable Class

Dr. Arun Balaji Buduru
Head, Center of Technology in Policing
Founding Head, Usable Security Group (USG)
Associate Professor, Dept. of CSE | HCD
IIIT-Delhi, India

# 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**

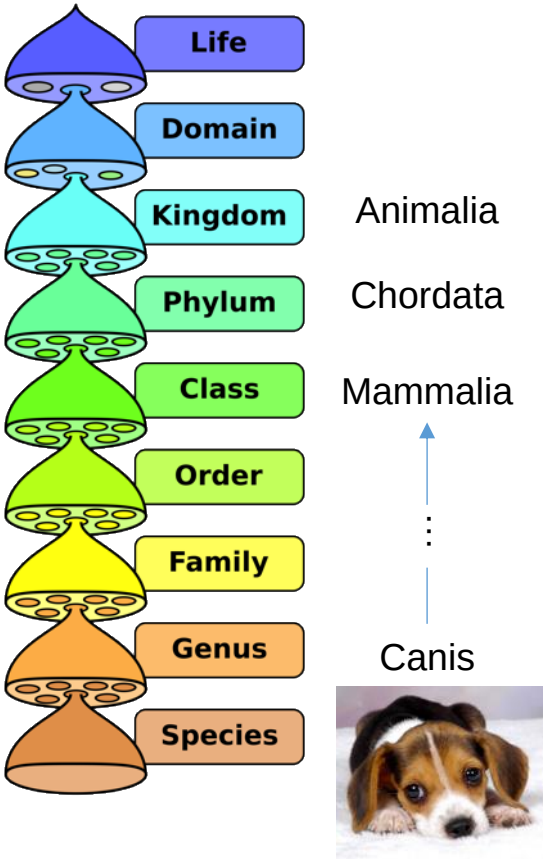
- `parkInCompactSpace()`
- `turnOnEngine()`
- `turnOffEngine()`
- `drive()`

# 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?**

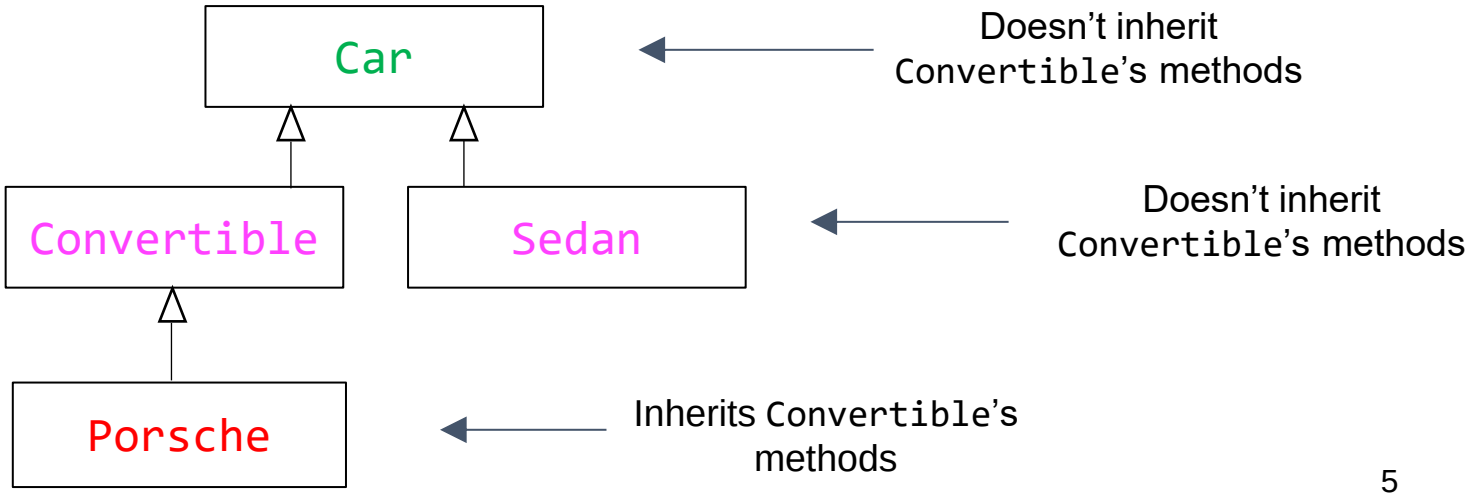
# Inheritance

- 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



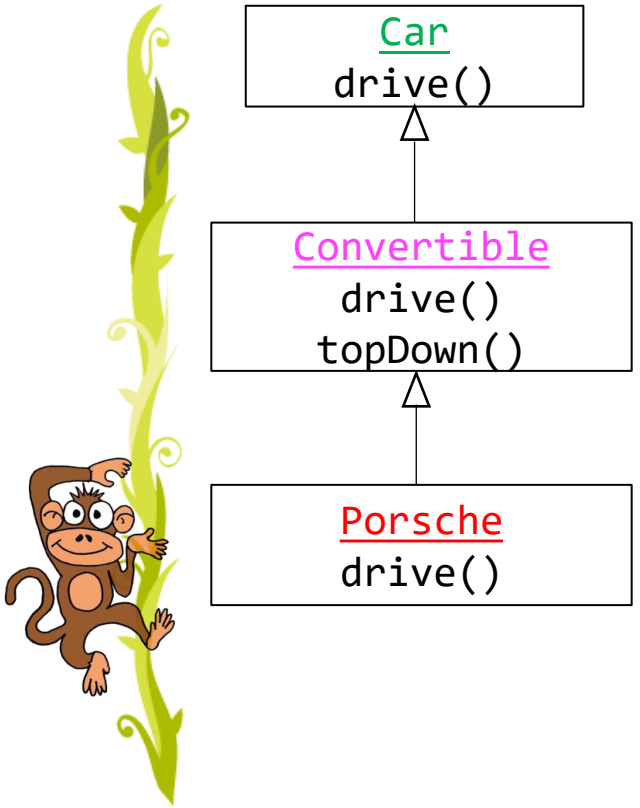
# Adding new methods

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



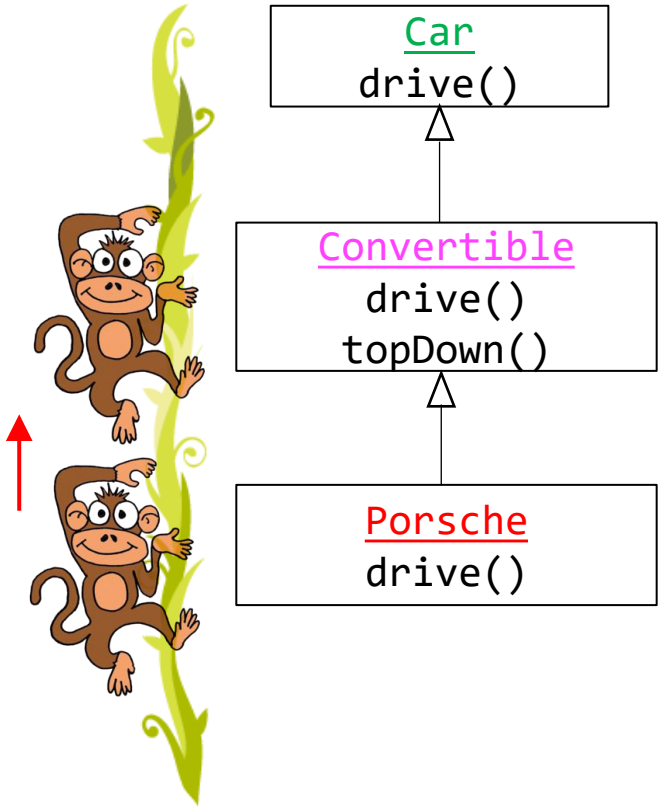
# Method Resolution (1/2)

- 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`



# Indirectly Accessing `private` Instance Variables in Superclass by defining Accessors and Mutators

```
public class Car  {

    private Radio _myRadio;

    public Car() {
        _myRadio = new Radio();
    }

    protected Radio getRadio(){
        return _myRadio;
    }
    protected void setRadio(Radio radio){
        _myRadio = radio;
    }

}
```

- Remember from earlier that private variables are not directly inherited by subclasses
- If `Car` does want its subclasses to be able to access and change the value of `_myRadio`, it can define protected accessor and mutator methods
  - Will non-subclasses be able to access `getRadio()` and `setRadio()` ?
- Very carefully consider these design decisions in your own programs – which properties will need to be accessible to other classes?

# Calling Accessors/Mutators From Subclass

- `Convertible` can get a reference to `_radio` by calling `this.getRadio()`
  - Subclasses automatically inherit these public accessor and mutator methods
- Note that using “double dot” we’ve chained two methods together
  - First, `getRadio` is called, and returns the radio
  - Next, `setFavorite` is called on that radio

```
public class Convertible extends Car {
   public Convertible() {
   }

   public void setRadioPresets(){
      this.getRadio().setFavorite(1, 95.5);
      this.getRadio().setFavorite(2, 92.3);
   }
}
```

# Let’s step through some code

- Somewhere in our code, a `Convertible` is instantiated

```
//somewhere in the program
Convertible convertible = new Convertible();
convertible.setRadioPresets();
```

- The next line of code calls `setRadioPresets()`
- Let’s step into `setRadioPresets()`

# Let's step through some code

- When someone calls `setRadioPresets()`; first line is `this.getRadio()`
- `getRadio()` returns `_myRadio`
- What is the value of `_myRadio` at this point in the code?
  - Has it been initialized?
  - Nope, assuming that the structure of class Car is exactly as shown on right side (i.e. without any constructor), we'll run into a `NullPointerException` here :(

```
public class Convertible extends Car {
   public Convertible() { //code elided
   }

   public void setRadioPresets() {
      this.getRadio().setFavorite(1, 95.5);
      this.getRadio().setFavorite(2, 92.3);
   }
}
```

---

```
public class Car {

   private Radio _myRadio;

   public Radio getRadio() {
      return _myRadio;
   }
}
```

# Making Sure Superclass's Instance Variables are Initialized

- `Convertible` may declare its own instance variables, which it initializes in its constructor
- `Car`'s instance variables are initialized in the `Car` constructor
- When we instantiate `Convertible`, how can we make sure `Car`'s instance variables are initialized too?
  - Case-1: Car has a default constructor that instantiate all its fields
  - Case-2: Car has a parameterized constructor for initializing all its fields

# `super()`: Invoking Superclass's Default Constructor (Case 1)

- Let's assume that `Car`'s instance variables (like `_radio`) are initialized in `Car`'s default constructor
- Whenever we instantiates `Convertible`, default constructor of `Car` is called automatically
- To **explicitly** invoke `Car`'s default constructor, we can call `super()` inside the constructor of `Convertible`
  - **Can only make this call once**, and it must be the very first line in the subclass's constructor

```
public class Convertible extends Car {

   private ConvertibleTop _top;

   public Convertible() {
      super();
      _top = new ConvertibleTop();
      this.setRadioPresets();
   }

   public void setRadioPresets(){
      this.getRadio().setFavorite(1, 95.5);
      this.getRadio().setFavorite(2, 92.3);
   }
}
```

# super(): Invoking Superclass's Parameterized Constructor (Case 2)

```
public class Car {

    private Racer _driver;
    public Car(Racer driver) {
        _driver = driver;
    }
    ......
}
```

```
public class Convertible extends Car {

    private ConvertibleTop _top;

    public Convertible(Racer driver) {
        super(driver);
        _top = new ConvertibleTop();
    }

    ......
}
```

- What if the superclass's constructor takes in a parameter?
  - We've modified `Car`'s constructor to take in a `Racer` as a parameter
  - How do we invoke this constructor correctly from the subclass?
- In this case, need the `Convertible`'s constructor to also take in a `Racer`
- The `Racer` is then passed as an argument to `super()` – now `Racer`'s constructor will initialize `_driver` to the instance of `Racer` that was passed to the `Convertible`

# What if we don't call `super()`?

- What if we forget to call `super()`?
- If you don't explicitly call `super()` first thing in your constructor, Java automatically calls it for you, passing in no arguments
- But if superclass's constructor requires a parameter, you'll get an error!
- In this case, we get a **compiler error** saying that there is no constructor "`public Car()`", since it was declared with a parameter

```
public class Convertible extends Car {

    private ConvertibleTop _top;

    public Convertible(Racer driver) {
        //oops forgot to call super()
        _top = new ConvertibleTop();
    }

    ......
}
```

# How to Load Passengers?

- What if we wanted to seat all of the passengers in the car?
- Sedan, Convertible, and Van all have different numbers of seats
  - They will all have different implementations of the same method

# Solution-1: Using Constructor Parameters

```
public class Convertible extends Car {
     private Passenger _p1;
     public Convertible(Racer driver, Passenger p1) {
          super(driver);
          _p1 = p1;
     }
     //code with passengers elided
}
```

```
public class Sedan extends Car {
     private Passenger _p1, _p2, _p3, _p4;
     public Sedan(Racer driver, Passenger p1,
          Passenger p2, Passenger p3, Passenger p4) {
          super(driver);
          _p1 = p1;
          _p2 = p2;
          _p3 = p4;
     }
    //code with passengers elided
}
```

- Notice how we only need to pass `driver` to `super()`
- We can add additional parameters in the constructor that only the subclasses will use
- Note that super() has to be the first statement inside the constructor.

# Any drawbacks in Previous Approach?

- How about creating an interface Passengers with a method loadPassenger?
  - Which class should implement that?
    - Superclass (Car) or Subclasses (Convertible, Sedan, and Van) ?
  - Issues
    - Creating an extra interface (possibly a new file)
    - Each subclass should have the declaration in the following form:
      - public class Sedan extends Car implements Passengers { …. }

# `abstract` Methods and Classes

- We declare a method `abstract` in a superclass when the subclasses can't really re-use any implementation the superclass might provide
- In this case, we know that all `Car`s should `loadPassengers`, but each subclass will `loadPassengers` very differently
- `abstract` method is declared in superclass, but not defined – up to subclasses farther down hierarchy to provide their own implementations

# Solution-2: Using abstract Methods and Classes

- Here, we've modified Car to make it an abstract class: a class with preferably an abstract method
  - You can avoid abstract method and just mark class as abstract if you don't wish to allow object creation of this class
- We declare both Car and its loadPassengers method abstract: if one of a class's methods is abstract, the class itself must also be declared abstract
- An abstract method is only declared by the superclass, not implemented – use semicolon after declaration instead of curly braces

```
public abstract class Car {

    private Racer _driver;

    public Car(Racer driver) {
        _driver = driver;
    }

    public abstract void loadPassengers();

}
```

# Solution-2: Using abstract Methods and Classes

```
public class Convertible extends Car{
    @Override
    public void loadPassengers(){
       Passenger p1 = new Passenger();
       p1.sit();
    }
}
```

```
public class Sedan extends Car{
    @Override
    public void loadPassengers(){
        Passenger p1 = new Passenger();
        p1.sit();
        ......
        Passenger p3 = new Passenger();
        p3.sit();
    }
}
```

```
public class Van extends Car{
    @Override
    public void loadPassengers(){
       Passenger p1 = new Passenger();
       p1.sit();
       .....
       .....
       Passenger p6 = new Passenger();
       p6.sit();
    }
}
```

- All concrete subclasses of Car override by providing a concrete implementation for Car's abstract loadPassengers() method
- As usual, method signature must match the one that Car declared

# `abstract` Methods and Classes

- `abstract` classes cannot be instantiated!
  - This makes sense – shouldn't be able to just instantiate a generic `Car`, since it has no code to `loadPassengers()`
  - Instead, provide implementation of `loadPassengers()` in concrete subclass, and instantiate subclass
- Subclass at any level in inheritance hierarchy can make `abstract` method concrete by providing implementation
- Even though an `abstract` class can't be instantiated, its constructor must still be invoked via `super()` by a subclass
  - because only the superclass knows about (and therefore only it can initialize) its own instance variables

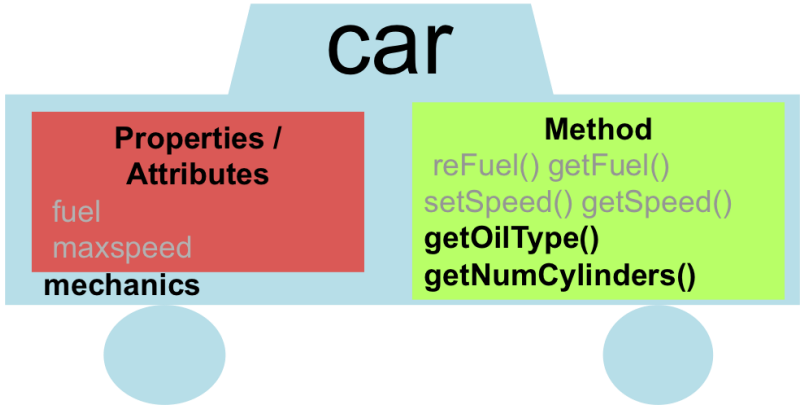
# So.. What's the difference?

- You might be wondering: what's the difference between `abstract` classes and interfaces?
- `abstract` Classes:
  - Can define instance variables
  - Can define a mix of concrete and `abstract` methods
  - You can only inherit from one class
- Interfaces:
  - Cannot define any instance variables/concrete methods
  - You can implement multiple interfaces

# What if the Cars are Getting Modified?


No modifications should ever be allowed !!

# Immutable Classes (1/5)

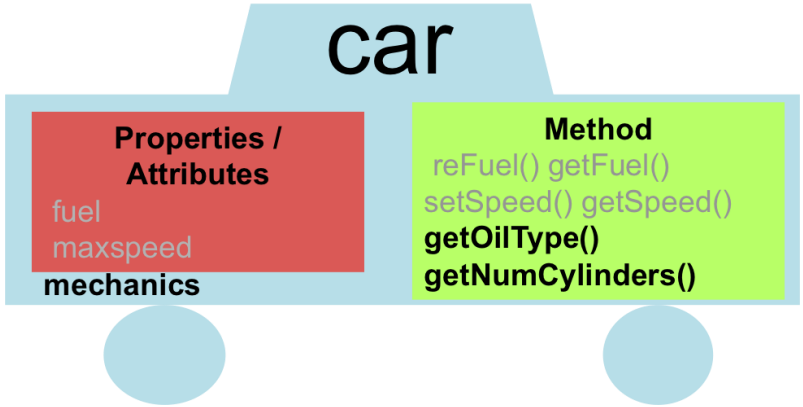



1. Don't provide any methods that modify the object's state.
2. Make all fields `private`. (ensure encapsulation)
3. Make all fields `final`.

```
public class Mechanics {
    private final String oilType;
    private final int numCylinders;
    public Mechanics(String oil, int cylinders)
    public String getOilType();
    public int getNumCylinders();
}
```

# Immutable Classes (2/5)

1. Don't provide any methods that modify the object's state.
2. Make all fields `private`. (ensure encapsulation)
3. Make all fields `final`.

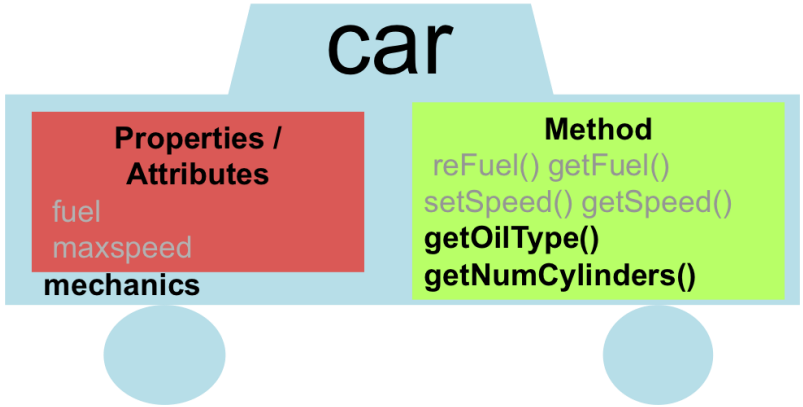



```
public class Mechanics {
   public final Tire tire;
   .......

}

// The user can easily do this:
mechanics.tire.setSize(20);
```

```
public class Tire {
    private int size;
    public int getSize();
    public void setSize(int);
}
```

# Immutable Classes (3/5)

1. Don't provide any methods that modify the object's state.
2. Make all fields `private`. (ensure encapsulation)
3. Make all fields `final`.

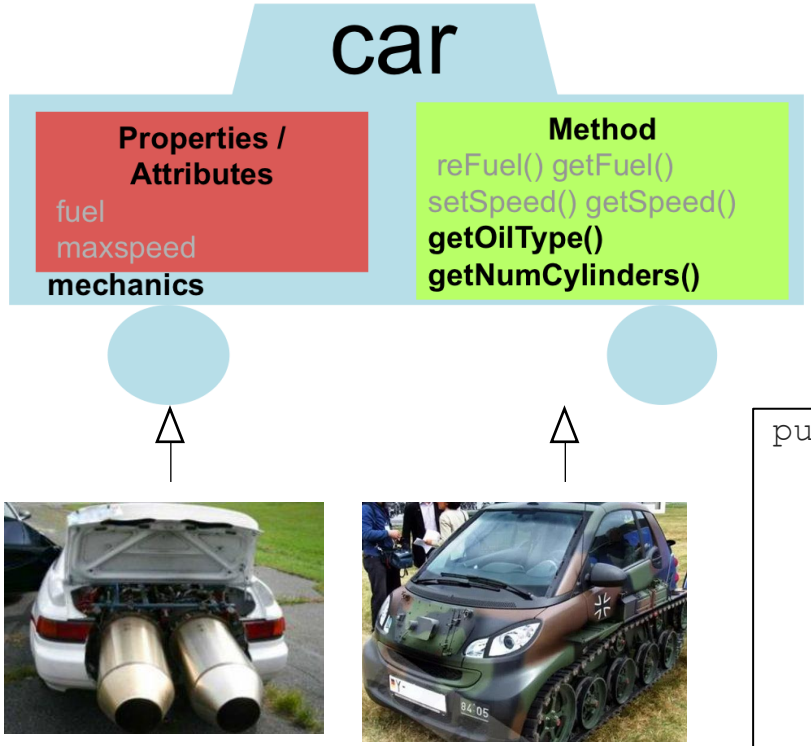



- Setting a reference variable final means that it can never be reassigned to refer to a different object.
  - You can't set that reference to refer to another object later ( = ).
  - It does not mean that the object's state can never change!

```
public class Mechanics {
    private final Tire tire;
    ........
    public Tire getTire(){return tire;}
}

// The user can easily do this:
mechanics.getTire().setSize(20);
```

```
public class Tire {
    private int size;
    public int getSize();
    public void setSize(int);
}
```

# Immutable Classes (4/5)

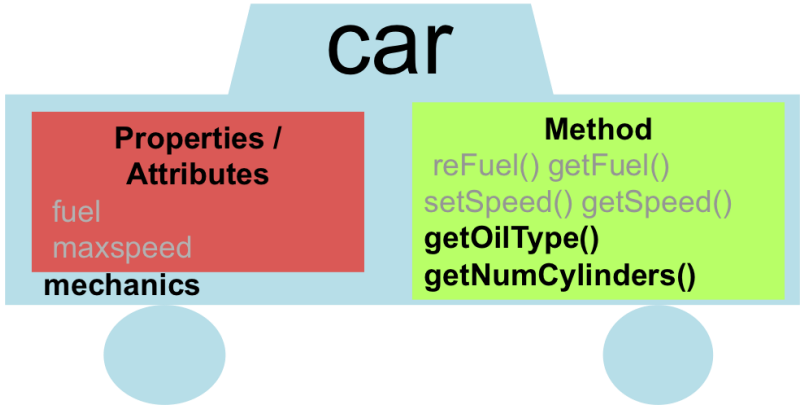



```
public class Mechanics {
    private final String oilType;
    private final int numCylinders;

    public Mechanics(String oil, int cylinders)
    public String getOilType();
    public int getNumCylinders();
}
```

```
public class ModifiedMechanics extends Mechanics {

  ......
  @Override
  public String getOilType(){
      return "Rocket Fuel";
  }
  @Override
  public int getNumCylinders(){return 18;}//Bugatti
}
```

How to fix these?

# Immutable Classes (5/5)




```
public class final Mechanics {
    private final String oilType;
    private final int numCylinders;

    public Mechanics(String oil, int cylinders)
    public String getOilType();
    public int getNumCylinders();
}
```

Mechanics cannot be extended as it is declared as final

```
public class ModifiedMechanics extends Mechanics {

  ......
  @Override
  public String getOilType(){
      return "Rocket Fuel";
  }
  @Override
  public int getNumCylinders(){return 18;}//Bugatti
}
```

# Summary: Making a Class Immutable

1. Don't provide any methods that modify the object's state.
2. Make all fields `private`. (ensure encapsulation)
3. Make all fields `final`.
4. Ensure exclusive access to any mutable object fields.
   - Don't let a client get a reference to a field that is a mutable object (don't allow any mutable representation exposure.)
5. Ensure that the class cannot be *extended*.