CSE201: Monsoon 2020, Section-A
Advanced Programming

Lecture 08: Abstract Class and Immutable Class

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

This Lecture

- Abstract class and abstract methods
- Immutable classes

Slide acknowledgements: CS15, Brown University

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?

- Car or Sedan or Convertible need not **know about** the passenger
  - Racer and passenger would always be changing
  - They are not going to be used anywhere in the class other than loadPassenger method
- 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

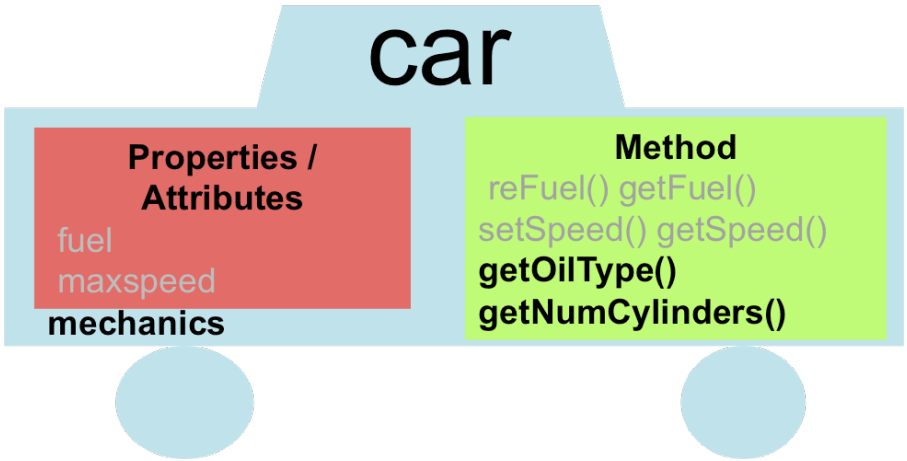
So.. What's the difference?

- 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 (except default method)
  - You can implement multiple interfaces

What if the Cars are Getting Modified?

No modifications should ever be allowed !!

Immutable Classes (1/5)



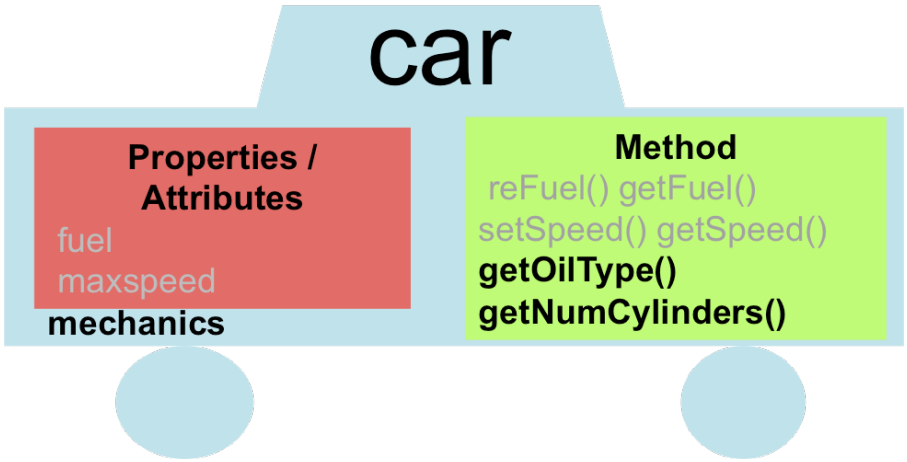
1. Don't provide any methods that modify the object's state.
2. Make all fields `private`. (ensure encapsulation)
3. Make all fields `final`.

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

Question

- Immutable classes have their fields marked as *final*. Then, why can’t we make those fields as *public* and let clients access them without any getter methods ?

Immutable Classes (2/5)



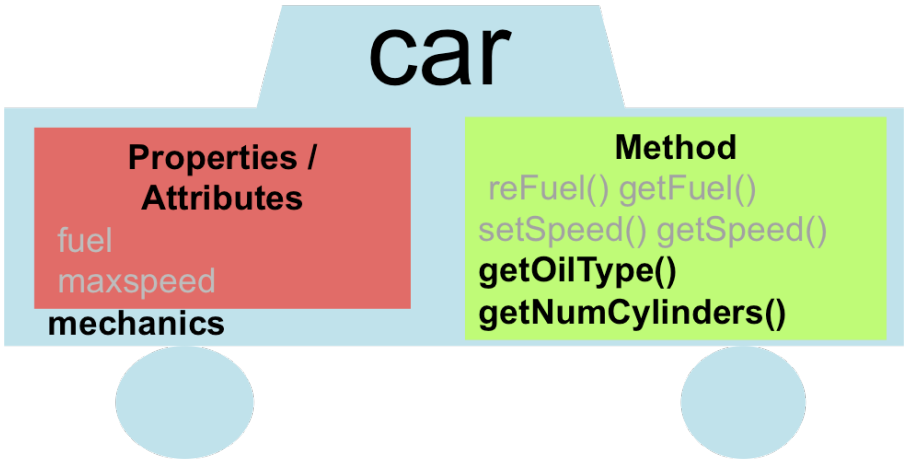
1. Don't provide any methods that modify the object's state.
2. Make all fields `private`. (ensure encapsulation)
3. Make all fields `final`.

```
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`.

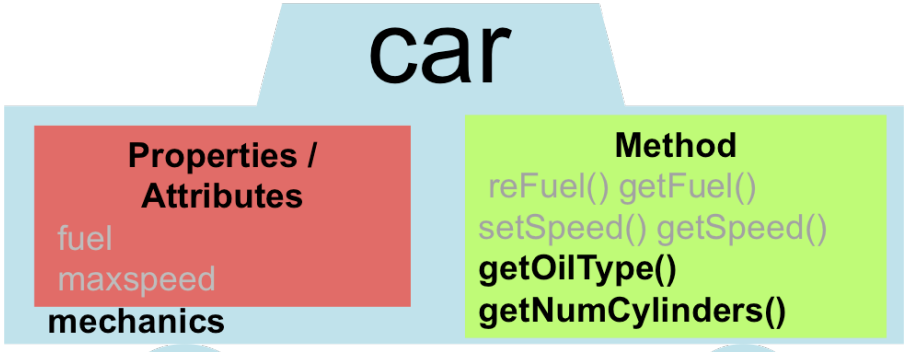


- Setting a reference variable final means that it can never be reassigned to refer to a different object.
  - You can't set that reference to refer to another object later (=).
  - It does not mean that the object's state can never change!

```
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)

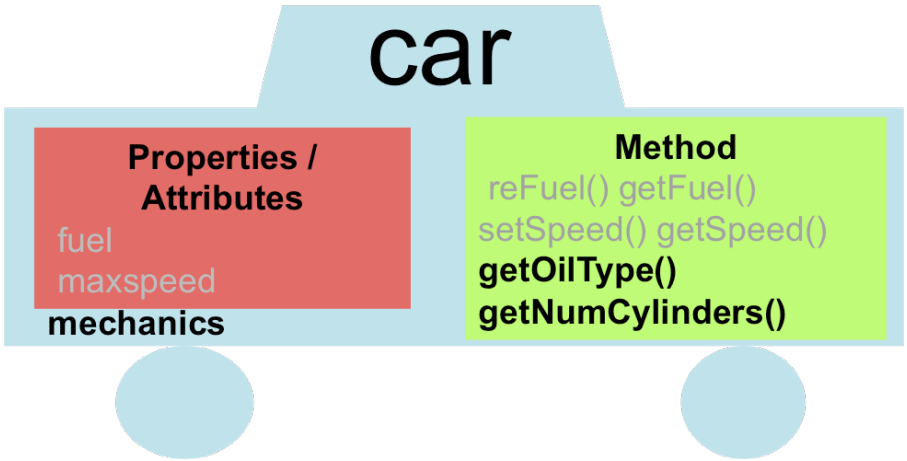


```
public class Mechanics {
    private final String oilType;
    private final int numCylinders;

    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*.

Next Lecture

- Class Object
- Assignment-2
  - What: Syllabus: Interfaces, Inheritance, and Polymorphism
  - When: Friday 18th
- Quiz-2
  - What: Syllabus: Inheritance and Polymorphism, Abstract Class, Immutable Class, and Class Object
  - When: Lab slot on Friday 25th from 4.15pm-4.35pm