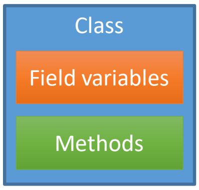
Object oriented design

Designing with objects

As we have seen, a class can consist of one or more of the following:

- Field variables
- Methods (including constructors)



Designing with objects

Lets design a class structure representing a car

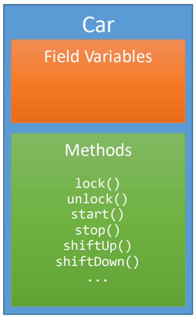
The requirements are:

- A car can contain x number of doors.
- A car can contain x number of tires.

How could we implement this?

Designing with objects

First, we set up our car and add methods specific to the car.



Car

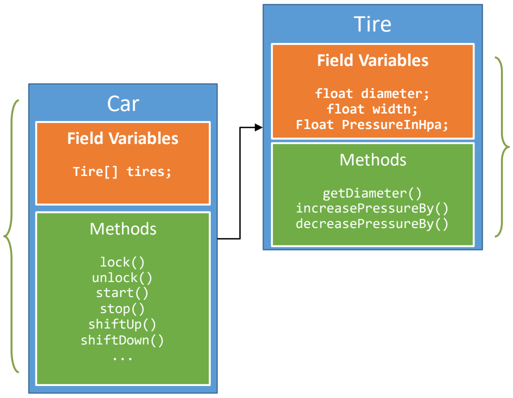
Field Variables

Methods

```
lock()
unlock()
start()
stop()
shiftUp()
shiftDown()
...
```

Designing with objects

Next, we set up the tires. This is in a separate class, and the Car contains 1 or more Tires.



Add an array of Tire objects as a field.

Car

Field Variables

```
Tire[] tires;
```

Methods

```
lock()
unlock()
start()
stop()
shiftUp()
shiftDown()
...
```

Tire

Field Variables

```
float diameter;
float width;
Float PressureInHpa;
```

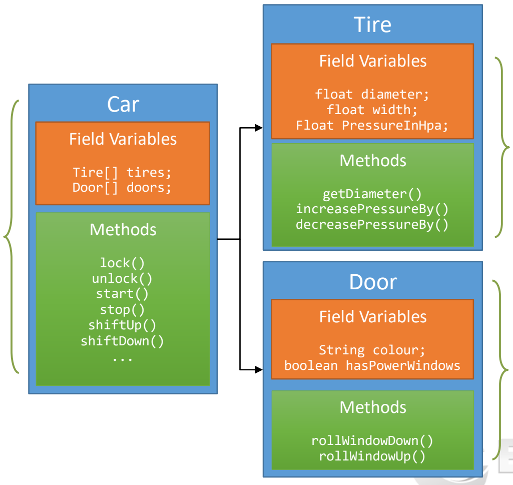
Methods

```
getDiameter()
increasePressureBy()
decreasePressureBy()
```

The responsibility to control tire pressure is handled by this class, not by the **Car** class.

Designing with objects

Lastly, we set up a Door class, and add one or more doors to the Car class.



Add an array of Door objects as a field.

Car

Field Variables

```
Tire[] tires;
Door[] doors;
```

Methods

```
lock()
unlock()
start()
stop()
shiftUp()
shiftDown()
...
```

Tire

Field Variables

```
float diameter;
float width;
Float PressureInHpa;
```

Methods

```
getDiameter()
increasePressureBy()
decreasePressureBy()
```

The responsibility to control tire pressure is handled by this class, not by the **Car** class.

Door

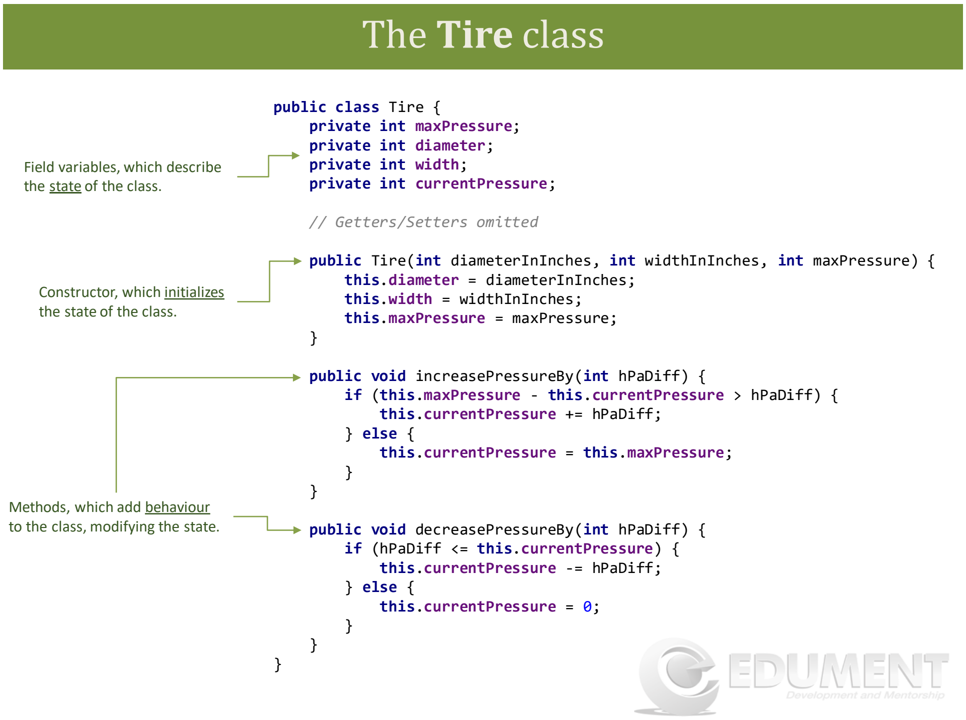
Field Variables

```
String colour;
boolean hasPowerWindows
```

Methods

```
rollWindowDown()
rollWindowUp()
```

The responsibility to steer windows is delegated to the **Door** class.



The **Tire** class

Field variables, which describe the state of the class.

Constructor, which initializes the state of the class.

Methods, which add behaviour to the class, modifying the state.

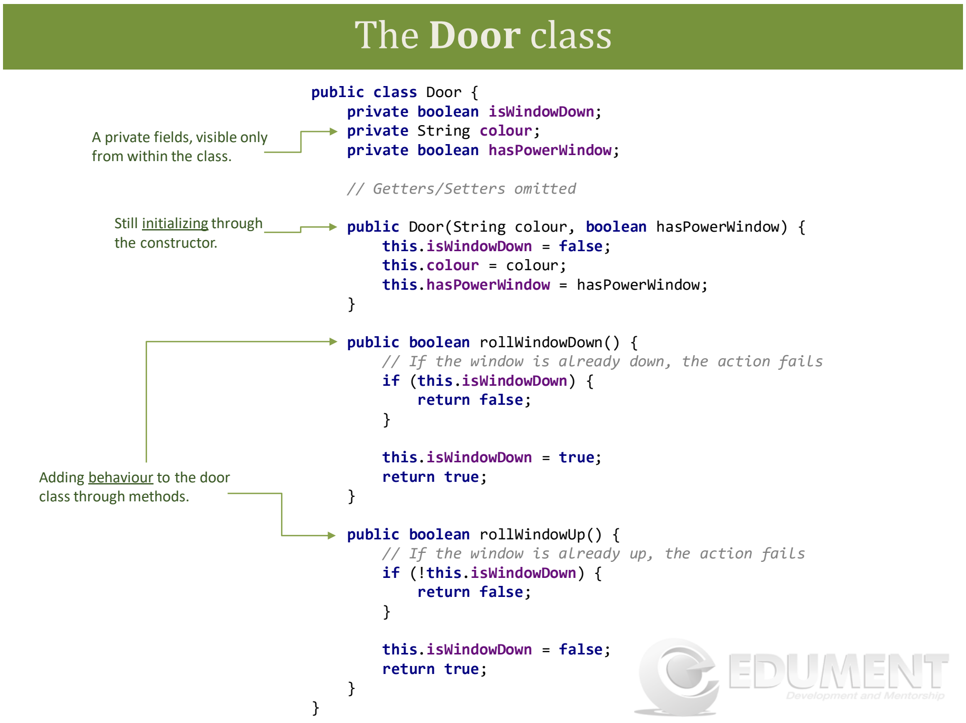
```java
public class Tire {
    private int maxPressure;
    private int diameter;
    private int width;
    private int currentPressure;

    // Getters/Setters omitted

    public Tire(int diameterInInches, int widthInInches, int maxPressure) {
        this.diameter = diameterInInches;
        this.width = widthInInches;
        this.maxPressure = maxPressure;
    }

    public void increasePressureBy(int hPaDiff) {
        if (this.maxPressure - this.currentPressure > hPaDiff) {
            this.currentPressure += hPaDiff;
        } else {
            this.currentPressure = this.maxPressure;
        }
    }

    public void decreasePressureBy(int hPaDiff) {
        if (hPaDiff <= this.currentPressure) {
            this.currentPressure -= hPaDiff;
        } else {
            this.currentPressure = 0;
        }
    }
}
```



The **Door** class

A private fields, visible only from within the class.

Still initializing through the constructor.

Adding behaviour to the door class through methods.

```java
public class Door {
    private boolean isWindowDown;
    private String colour;
    private boolean hasPowerWindow;

    // Getters/Setters omitted

    public Door(String colour, boolean hasPowerWindow) {
        this.isWindowDown = false;
        this.colour = colour;
        this.hasPowerWindow = hasPowerWindow;
    }

    public boolean rollWindowDown() {
        // If the window is already down, the action fails
        if (this.isWindowDown) {
            return false;
        }

        this.isWindowDown = true;
        return true;
    }

    public boolean rollWindowUp() {
        // If the window is already up, the action fails
        if (!this.isWindowDown) {
            return false;
        }

        this.isWindowDown = false;
        return true;
    }
}
```

The **Car** class

Custom types as fields.

Constructor, initializing the state. Part of the state consists of types that we've created ourselves.

```java
public class Car {
    private Tire[] wheels;
    private Door[] doors;
    private String colour;
    private String carMake;

    // Getters/Setters omitted

    public Car (int tireCount, int doorCount,
                String carMake, String colour) {
        this.carMake = carMake;
        this.colour = colour;
        this.wheels = new Tire[tireCount];
        this.doors = new Door[doorCount];
    }
}
```

Composition

This is a relatively simple example which we could easily continue to build upon.

For instance, we could add methods to the **Car** class, such as **StartCar** or field variables such as **TotalWeight**.

Object relations

Relations

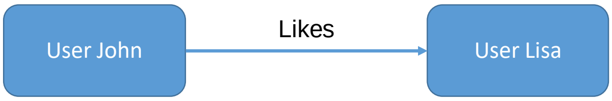
A relationship defines the connection between objects

We have three types of relations:

- Association
- Aggregation
- Composition

Association

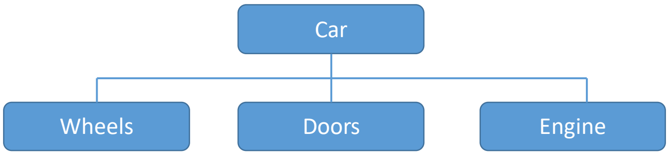
Represents a relationship between two or more objects where all objects have their own lifecycle and there is no owner.



Aggregation

Aggregation is a specialized form of Association where all object have their own lifecycle but **there is ownership**.

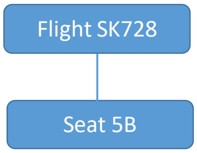
In the example below the wheels, doors and engine have their own lifecycle, they can exist before the car is made and also after the car is scrapped.



Composition

Composition is a specialized strong form of Aggregation. In this relationship child objects **can't exist without Parent object**.

In the example below, the seats that can be reserved for a given flight have no logical meaning outside that specific flight.



Responsibility between classes should be clear

The **Car** class will expect a certain behaviour from the classes it’s composed of.

It doesn’t care about **HOW** these classes perform their tasks, just that they do.

The implementation specifics are unknown to the ”wrapping” class. It’s outside of the **responsibility** of the **Car** class.

We’re still separating concerns!

Exercise 15

Let's do exercise 15