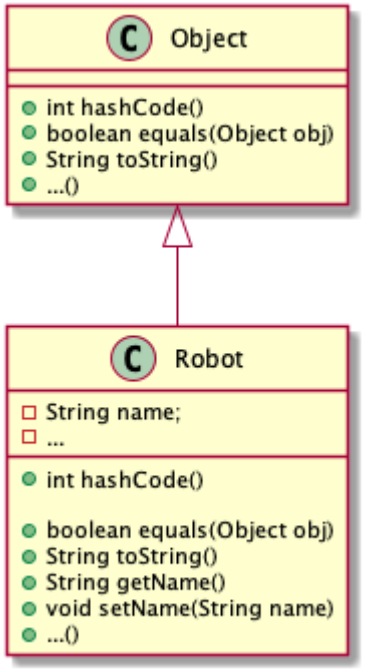
Java Fundamentals Course 2 - Day 2

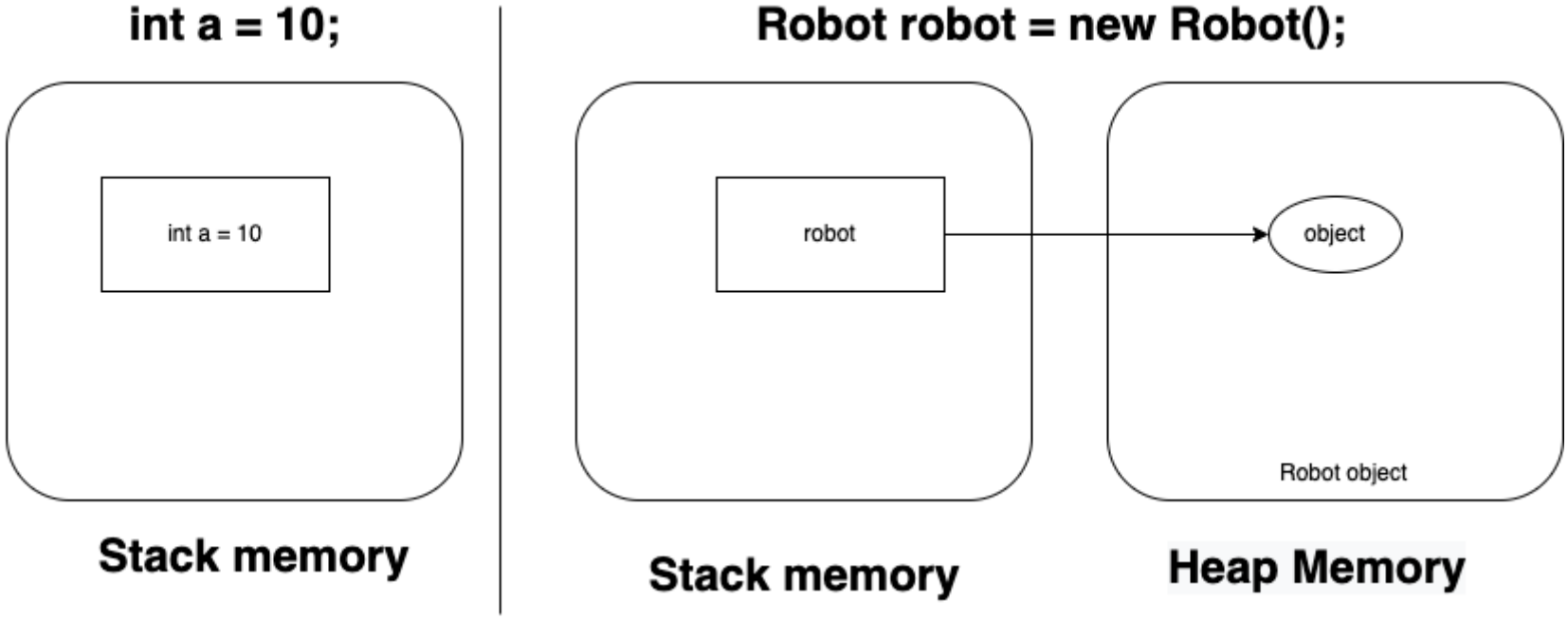
Looking Back: Day 10: Recap day JF2-1

- Java Object: equals, hashcode, toString?
- Java Memory Layout?
- Interfaces in Java?
- Abstract Classes?
- Warmup exercise!

Java Object: equals, hashcode, toString



Recap: Java Memory Layout?

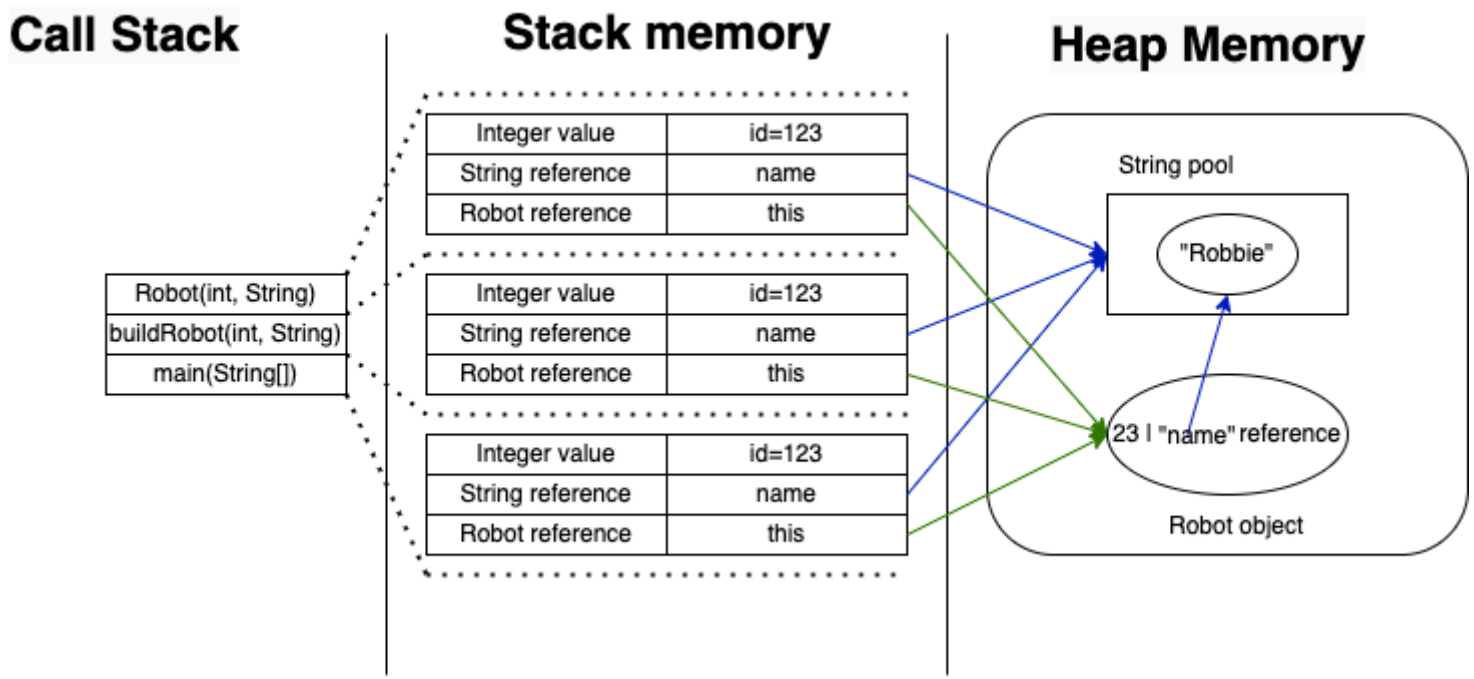


Java **Heap Space** is used throughout the application, but **Stack** is only used for the method — or methods — currently running.

The **Heap Space** contains all objects are created, but **Stack** contains any reference to those objects.

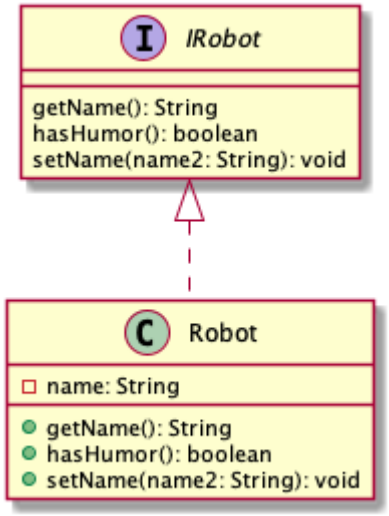
Java Memory Layout: Another example

```
public class Main {
    private static Robot buildRobot(int id, String name) {
        return new Robot(id, name);
    }

    public static void main(String[] args) {
        int id = 23;
        String name = "robbie";
        Robot robot = null;
        robot = buildRobot(id, name);
    }
}

class Robot {
    public Robot(int id, String name) {
        this.id = id;
        this.name = name;
    }

    private int id;

    private String name;
}
```



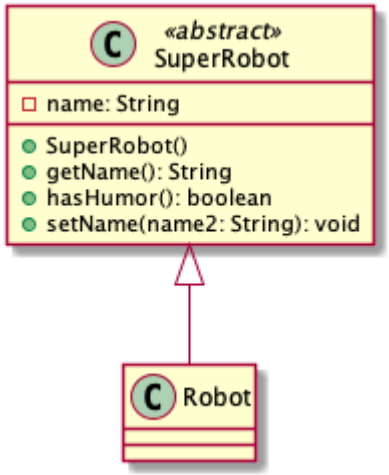
Garbage collection?

Recap: Interfaces in Java?



Interfaces are used to achieve abstraction. Designed to support dynamic method resolution at run time It helps you to achieve loose coupling. Allows you to separate the definition of a method from the inheritance hierarchy

Recap: Abstract Classes



Abstract classes offer default functionality for the subclasses. Provides a template for future specific classes Helps you to define a common interface for its subclasses Abstract class allows code reusability.

Recap: Exercise extending Classes

*Exercise *

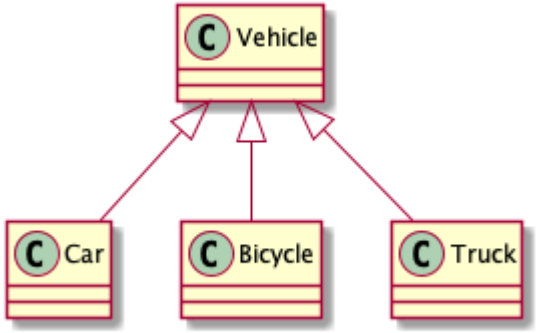
A client wants to open a **Vehicle** store and wants sell **Cars**, **Bicycles** and **Trucks**.

The client want to give each object a name and a description.

Hints

- Create a class Vehicle.
- Create 3 subclasses 'Car', 'Bicycle' and 'Truck' which extends 'Vehicle' class.
- You can give these classes 2 fields, name and description.
- You can give these classes getters and can and a constructor with arguments for fields name and description.

```
class Vehicle {}
class Car extends Vehicle {}
class Bicycle extends Vehicle {}
class Truck extends Vehicle {}
```



Todays modules

- Enumerations
- Java Records
- Lombok
- Sealed Classes

Module: Enumerations

What are enumerations for?

What problem do they solve?

Java enums are a powerful construct. They allow grouping of constants - where each constant is a singleton object. Each constant can optionally declare a body, which can be used to override the behavior of the base enum declaration.

Using constants without enum

```
public class PizzaStatus{
    public static final String ORDERED = "ORDERED";
    public static final String READY = "READY";
    public static final String DELIVERED = "DELIVERED";
}
```

Using constants with enum

```
public enum PizzaStatus {
    ORDERED,
    READY,
    DELIVERED;
}
```

How do you use enumerations in Java?

Using if-else

Here we have a **PizzaStatus** enum and a **Pizza** class. **Pizza** 's field status uses **PizzaStatus**.

```
enum PizzaStatus {
    ORDERED, READY, DELIVERED;
}

class Pizza {
    private PizzaStatus status;


    public PizzaStatus getStatus() {
        return status;
    }

    public void setStatus(PizzaStatus status) {
        this.status = status;
    }

    public boolean isDeliverable() {
        if (getStatus() == PizzaStatus.READY) {
            return true;
        }
        return false;
    }
}
```

if-else continued…

Here we have a **Main** class to run the application. The main method creates an instance of Pizza and sets it's status field. We can use **if-else** to test the status.

```
public class Main {

    public static void main(String[] args) {
        Pizza pizza = new Pizza();
        pizza.setStatus(PizzaStatus.ORDERED);
        System.out.println("isDeliverable:" + pizza.getStatus());
        System.out.println("getStatus:" + pizza.isDeliverable());

        pizza.setStatus(PizzaStatus.READY);
        System.out.println("getStatus:" + pizza.getStatus());
        System.out.println("isDeliverable:" + pizza.isDeliverable());

        if (pizza.getStatus() == PizzaStatus.ORDERED) {
            System.out.println("pizza is ordered");
        } else if (pizza.getStatus() == PizzaStatus.READY) {
            System.out.println("pizza is ready");

        } else if (pizza.getStatus() == PizzaStatus.DELIVERED) {
            System.out.println("pizza is delivered");

        } else
            throw new IllegalArgumentException("Unexpected value: " + pizza.getStatus());
    }
}
```

Using switch

Here we have a **Main** class to run the application. The main method creates an instance of the same Pizza class and sets it's status field. We can use **switch** to test the status.

```
public class Main {

    public static void main(String[] args) {
        Pizza pizza = new Pizza();
        pizza.setStatus(PizzaStatus.ORDERED);
        System.out.println("isDeliverable:" + pizza.getStatus());
        System.out.println("getStatus:" + pizza.isDeliverable());

        //...

        switch (pizza.getStatus()) {
        case ORDERED: {
            System.out.println("pizza is ordered");
            break;
        }
        case READY: {
            System.out.println("pizza is ready");
            break;
        }
        case DELIVERED: {
            System.out.println("pizza is delivered");
            break;
        }
        default:
            throw new IllegalArgumentException("Unexpected value: " + pizza.getStatus());
        }
    }
}
```

How do you make enumerations?

Exercise

A client wants to update the **Vehicle** application and add a field vehicleStatus to keep strack of a status. And the field status can only have a value 'ordered', 'ready' or 'delivered'.

When the client uses the application, 3 vehicles should already be created. One task with status 'ordered', one task with status 'ready' and one with status 'delivered'. And only the vehicles with status 'ordered' and 'ready' should be displayed to know if new vehicles should added.

Bonus Exercise

Override toString method in enum VehicleStatus.

When field status has value 'ORDERED', print 'This vehicle has status ordered'. When field status has value 'READY', print 'This vehicle has status ready'. When field status has value 'DELIVERED', print 'This vehicle has status delivered'.

You can print the status in the main method in Main class.

Module: Java Records

Many times we write classes to simply hold data, such as database results, query results, or information from a service.

To accomplish this, we create data classes with the following:

1. **private, (final) fields** for each piece of data
2. **getter** (and setters) for each field
3. **public constructor** with a corresponding argument for each field
4. **equals** method that returns true for objects of the same class when all fields match
5. **hashCode** method that returns the same value when all fields match
6. **toString** method that includes the name of the class and the name of each field and its corresponding value

Record classes are a new kind of class in the Java language. **Record** classes help to model plain data aggregates with less ceremony than normal classes.

The declaration of a **record** class primarily consists of a declaration of its state; the **record** class then commits to an API that matches that state. This means that **record** classes give up a freedom that classes usually enjoy — the ability to decouple a class's API from its internal representation — but, in return, record class declarations become significantly more concise.

What are Java records?

What problem do they solve? (immutability, strong guarantees)

It is a common complaint that "Java is too verbose" or has "too much ceremony". Some of the worst offenders are classes that are nothing more than **immutable data carriers** for a handful of values. Properly writing such a data-carrier class involves a lot of **low-value**, **repetitive**, **error-prone** code: constructors, accessors, equals, hashCode, toString, etc.

The Java programming language has been enhanced with **records**, which are classes that act as transparent carriers for immutable data.

- immutable object: unchangeable, an object whose state **cannot** be modified after it is created
- mutable object: changeable, an object whose state **can** be modified after it is created

Released: 16

Traditional class

With lots of **low-value**, **repetitive**, **error-prone** code.

```
class Point {
    private final int x;
    private final int y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    int x() { return x; }
    int y() { return y; }

    public boolean equals(Object o) {
        if (!(o instanceof Point)) return false;
        Point other = (Point) o;
        return other.x == x && other.y == y;
    }

    public int hashCode() {
        return Objects.hash(x, y);
    }

    public String toString() {
        return String.format("Point[x=%d, y=%d]", x, y);
    }
}
```

Observations

There are two problems with it:

1. There is a lot of boilerplate code. Creating 'setters' for the fields we cause even more **repetitive** code.
2. We obscure the purpose of our class – to represent a Point with a x, and y.

We want to avoid repetitive code so can be more productive. We can use **record** classes or use Lombok **annotations**.

How do we use Java records?

```
record MessageRecord(String from, String to, String body) { }
```

Compiler generates boilerplate code

Using record:

```
record MessageRecord(String from, String to, String body) { }
```

Generates:

```
final class MessageRecord extends Record {
    private final String from;
    private final String to;
    private final String body;

    public final String toString() {
        //...
    }

    public final int hashCode() {
        //...
    }

    public final boolean equals(Object object) {
        //...
    }

    MessageRecord(String from, String to, String body) {
        this.from = from;
        this.to = to;
        this.body = body;
    }

    public String from() {
```

```
        return this.from;
    }

    public String to() {
        return this.to;
    }

    public String body() {
        return this.body;
    }
}
```

Constructor (& validation)

A **constructor** may be declared explicitly with a list of formal parameters which match the record header, as shown above.

This, so we can focus on **validating and normalizing parameters** without the tedious work of assigning parameters to fields.

```
record MessageRecord(String from, String to, String body) {
    public MessageRecord {
        Objects.requireNonNull(from, "'from' can not be null");
        Objects.requireNonNull(to, "'to' can not be null");
    }

    //or
    /*
    public MessageRecord {
        if (from == null) {
            throw new NullPointerException("'from' can not be null");
        }

        if (to == null) {
            throw new NullPointerException("'to' can not be null");
        }
    }
    */

    public MessageRecord(String from, String to) {
        this(from, to, "");
    }
}
```

constructor continued…

```
public class MessageMainRecord {
    public static void main(String[] args) {
        MessageRecord messageRecord = new MessageRecord("Alice", "Bob");
        System.out.println(messageRecord);

        try {
            MessageRecord messageRecordNPE = new MessageRecord(null, null);

        } catch (NullPointerException nullPointerException) {
            System.err.println(nullPointerException);
        }

    }
}
```

How do you make Java records?

Exercise

A client has heard of the usage of 'record' and it’s advantages. The client wants you to refactor the existing Task application.

Bonus exercise

After some thoughts and using the application (we don’t have any interface for inpint yet):

The client wants you to make the field 'name' mandatory. And when creating a Task and have a message displayed when the field is null.

Bonus exercise

After some thoughts and using the application (we don’t have any interface for inpint yet):

The client wants you to make it optional to set the field 'description', because filling the field for every task takes too much time.

And when a Task is created without a description, a default value of '' should be set.

Records vs. Lombok

Using project Lombok we can also generate a lot of boilerplate code: equals, hashCode, and toString, constructor, getters and setters.

And a lot more like builders and loggers.

Same as **record** example. Compiler plugin **Lombok** generates code for us. Not using **record** but **annotations** '@Getter', '@AllArgsConstructor' and '@NonNull' and more.

How do can we Lombok?

```
@ToString
@EqualsAndHashCode
@AllArgsConstructor
@Getter
public class MessageData {
    @NonNull
    private String from;
    @NonNull
    private String to;
    private String body;

    public MessageData(String from, String to) {
        this(from, to, "");
    }
}
```

We can also use annotations @Data to replace @ToString, @EqualsAndHashCode and @Getter. We can also use annotations @Builder and loggers like @Slf4j.

How do you use Lombok?

Exercise

A client has heard of the usage of 'Lombok' and it’s advantages. The client wants you to refactor the existing Tasks application by replacing 'record' with Lombok annotations.

Notes

If needed install IDE, see plugin https://projectlombok.org/

If needed add dependency to project: * maven: https://mvnrepository.com/artifact/org.projectlombok/lombok/ * or gradle, see same page

```
<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.22</version>
    <scope>provided</scope>
</dependency>
```

Module: Sealed classes

A **sealed** class or interface can be extended or implemented only by those classes and interfaces permitted to do so.

A class is sealed by applying the sealed modifier to its declaration. Then, after any extends and implements clauses, the permits clause specifies the classes that are permitted to extend the sealed class.

Released in 17

What are Java sealed classes?

What problem do they solve?

The object-oriented data model of inheritance hierarchies of classes and interfaces has proven to be highly effective in modeling the real-world data processed by modern applications.

This expressiveness is an important aspect of the Java language.

Java developers are familiar with the idea of restricting the set of subclasses because it often crops up in API design. The language provides limited tools in this area: Either make a **class final**, so it has zero subclasses, or make **the class or its constructor package-private**, so it can only have subclasses in the same package. An example of a package-private superclass appears in the JDK:

```
package java.lang;

abstract class AbstractStringBuilder { ... }
public final class StringBuffer  extends AbstractStringBuilder { ... }
public final class StringBuilder extends AbstractStringBuilder { ... }
```

So sealed classes and interfaces were introduced to restrict which other classes or interfaces may extend or implement them.

It should be possible for a superclass to be widely accessible (since it represents an important abstraction for users) but not widely extensible (since its subclasses should be restricted to those known to the author).

How can we use a sealed class with permits

Sealed can be used with **normal classes**, **interfaces**, **abstract classes** and **records**

Remember the extended classes need to have one of the following modifiers:

- Sealed: the class can be extended by the defined classes using permits
- Final: the class cannot be extended
- Non-sealed: any class can extend this class

Using sealed in normal Classes

```
sealed class Vehicle permits Car, Truck, Bicycle{}
final class Car extends Vehicle {}
final class Bicycle extends Vehicle {}
final class Truck extends Vehicle {}
```

Using sealed in abstract Classes

```
sealed abstract class Vehicle permits Car, Truck, Bicycle{}
final class Car extends Vehicle {}
final class Bicycle extends Vehicle {}
final class Truck extends Vehicle {}
```

Using sealed in record Classes

```
sealed record Vehicle permits Car, Truck, Bicycle{}
final class Car implements Vehicle {}
final class Bicycle implements Vehicle {}
final class Truck implements Vehicle {}
```

Using sealed in Interfaces

```
sealed interface Vehicle permits Car, Truck, Bicycle{}
final class Car implements Vehicle {}
final class Bicycle implements Vehicle {}
final class Truck implements Vehicle {}
```

Now lets create some Java 17 sealed classes

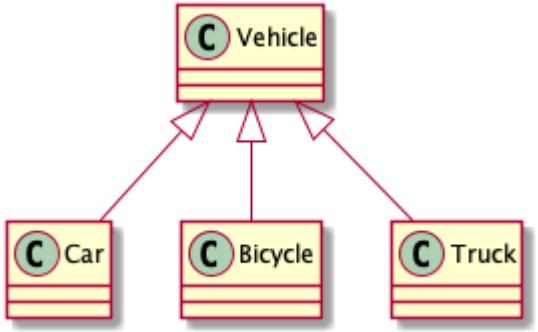
Exercise (part 1)

A client wants to open a **Vehicle** store and wants sell Cars, Bicycles and Trucks from his catalog.

Hints

- Create a class Vehicle.
- Create 3 subclasses 'Car', 'Bicycle' and 'Truck' which extends 'Vehicle' class.

```
class Vehicle {}
class Car extends Vehicle {}
class Bicycle extends Vehicle {}
class Truck extends Vehicle {}
```



Exercise (part 2)

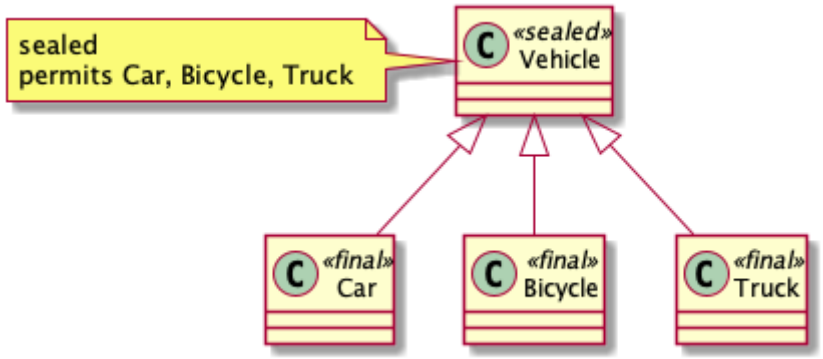
The client wants to limit his catalog to selling only the 3 types of Vehicles: Cars, Trucks an Bicycles.

No other Vehicle types may be created in the future.

Hints

- Continue using previous exercise.
- Make class Vehicle 'sealed' and still 'permit' Car, Truck, Bicycle to extend.
- You will need to make Car, Truck, Bicycle final so they can not be extended.

```
sealed class Vehicle permits Car, Truck, Bicycle{}
final class Car extends Vehicle {}
final class Bicycle extends Vehicle {}
final class Truck extends Vehicle {}
```



Using non-sealed class

Now lets make it possible to extend 'Car'.

Exercise (part 3)

The client wants to create more Trucks and Cars in his catalog because they are popular.

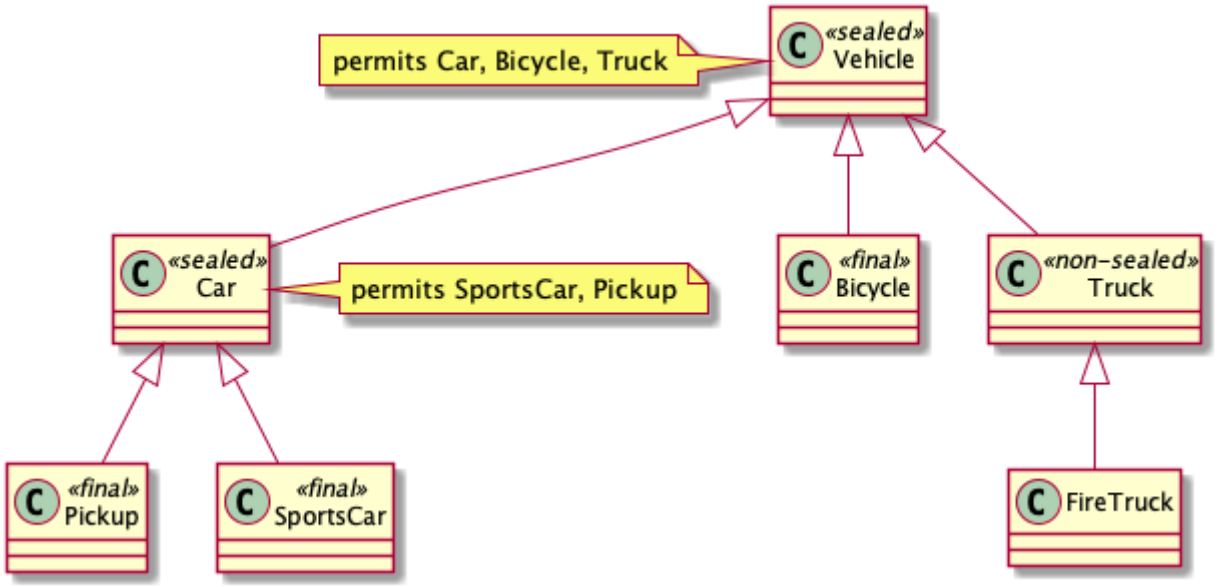
Currently only one Truck type should be added to the catalog. Its a FireTruck.

The client also want to sell more types of Cars. The Cars to sell are SportsCars and Pickups.

Hints

- Continue using previous exercise.
- Create subclass 'FireTruck' which extends non-sealed 'Truck' class.
- Create subclasses 'SportsCar', 'Pickup' which extends sealed 'Car' class.

```
sealed class Vehicle permits Truck, Bicycle, Car {}
final class Bicycle extends Vehicle {}
sealed class Car extends Vehicle permits SportsCar, Pickup{}
final class SportsCar extends Car {}
final class Pickup extends Car {}
non-sealed class Truck extends Vehicle {}
class FireTruck extends Truck {}
```


permits Car, Bicycle, Truck
«sealed»
Vehicle
«sealed»
Car
permits SportsCar, Pickup
«final»
Bicycle
«non-sealed»
Truck
«final»
Pickup
«final»
SportsCar
FireTruck

Bonus Exercise (part 4)

Now lets use some interfaces!

We don’t really want to create objects from Vehicle, Car and Truck.

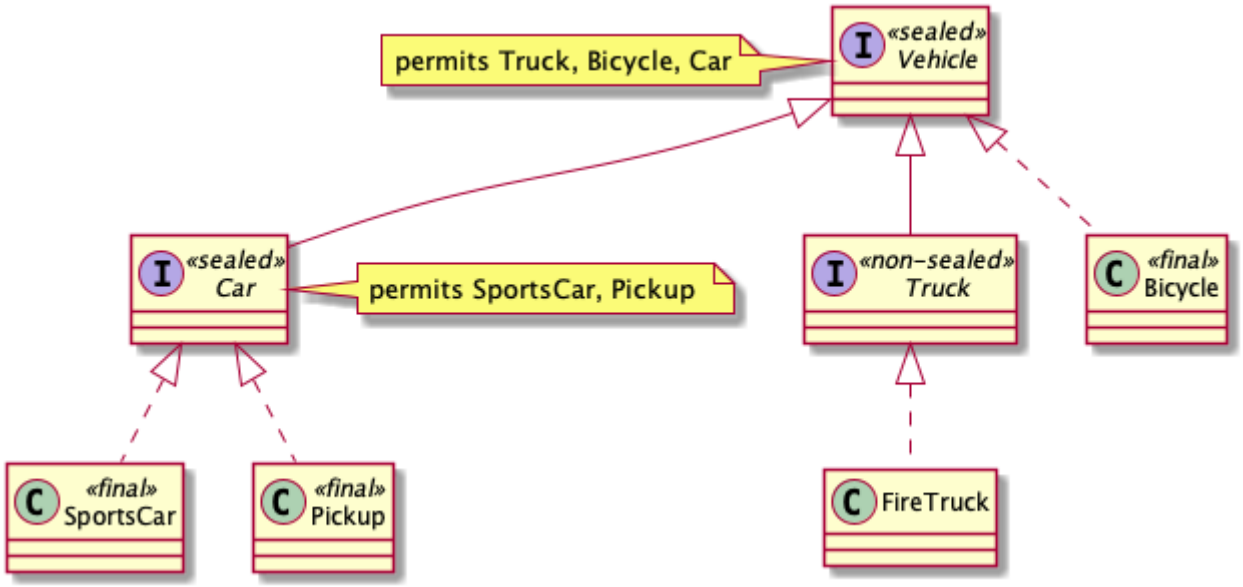
We want to define polymorphism in a declarative way. Vehicle, Car and Truck may have their own behaviours. A Vehicle may have have a driver, thus getDriver behaviour, and all classes which extend from Vehicle should implement getDriver(). A Car may have have passengers, thus getPassengers behaviour. A Truck may have have load, thus getLoad behaviour.

The client is not sure yet which behaviours may be added, but he likes to use interfaces.

Hints

- Continue using previous exercise.
- Update Vehicle class and make it an interface and permit Truck, Bicycle and Car to extend.
- Update class Car and make it an interface and permit SportsCar, Pickup to extend.

```
sealed interface Vehicle permits Truck, Bicycle, Car {}
sealed interface Car extends Vehicle permits SportsCar, Pickup {}
final class SportsCar implements Car {}
final class Pickup implements Car {}
final class Bicycle implements Vehicle { }
non-sealed interface Truck extends Vehicle{}
class FireTruck implements Truck {}
```


permits Truck, Bicycle, Car
«sealed»
Vehicle
«sealed»
Car
permits SportsCar, Pickup
«non-sealed»
Truck
«final»
Bicycle
«final»
SportsCar
«final»
Pickup
FireTruck