(7) Java Abstractions: Interfaces – Part 3

Nico Ludwig (@ersatzteilchen)

TOC

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

Interfaces can change

- When we introduced interfaces we underscored, that callers and callees are implemented at different times,
  - And we used interfaces as promises to API-calling code, that if it implements interfaces, it'll to work with the API now and in future!
  - => With interfaces we decoupled callers and callees in an effective way.
- But sometimes, an interface must be extended afterwards, then the callers won't understand the callees or vice versa.
- The topic we will discuss on the next slides is "what to do, when interfaces are getting incompatible".
- The good news is, that Java provides some language features, that help us to cope with "interface-development".

Interfaces are getting incompatible – Part 1

- The next step at *Garage* is offering even more types of gas, esp. gas of different octane rating (RON).
- Instead of adding new interfaces for new fuel types, it was decided to modify the interface *GasAcceptor*:

```
// <GasAcceptor.java>
public interface GasAcceptor {
        void fillGas(double liters);
        boolean supportsOctaneRating(double RON);
}
```

```
// <Garage.java>
public class Garage { // (other members omitted)
        public void refuel(GasAcceptor gasAcceptor, double liters, double RON) {
                if (gasAcceptor.supportsOctaneRating(RON)) {
                        gasAcceptor.fillGas(liters);
                }
        }
}
```

  - A *GasAcceptor* can now be checked, whether it supports fuel of certain octane rating via *GasAcceptor.supportsOctaneRating()*.
  - A new overload of *Garage.refuel()* was added, which allows to specify the RON of the gas to be refueled as as extra argument.
  - The idea is, that *GasAcceptor.fillGas() can* only be called, if *GasAcceptor.supportsOctaneRating()* returns true.

- But, there is a problem! All types, which implemented *GasAcceptor* in past do no longer implement *GasAcceptor* fully.

```
// <GasCar.java>
public class GasCar extends Car implements GasAcceptor { // (other members omitted)
        @Override
        public void fillGas(double liters) { /* pass */ }
}
```

  - It doesn't compile: "*GasCar* is not abstract and does not override abstract method *supportsOctaneRating(double)* in *GasAcceptor*"

- RON is for "Research Octane Number" to check how knock proof a *GasAcceptor*, i.e. an otto-engine is.

Interfaces are getting incompatible – Part 2 – Solution: abstract Classes

- The compiler already gave us a hint what to do in the case of *GasCar*: we could make *GasCar* an abstract class:

```
// <GasCar.java>
public abstract class GasCar extends Car implements GasAcceptor { // (other members omitted)
        @Override
        public void fillGas(double liters) { /* pass */ }
}
```

- But this doesn't solve the basic problems we have introduced by modifying *GasAcceptor*:
  - (1) Concretely we have to inherit another UDT from the abstract *GasCar* in order to implement it fully and create instances of it.
  - (2) But, seriously, we cannot make all implementors of *GasAcceptor* abstract!
    - Because meanwhile other developers have also created UDTs implementing *GasAcceptor*.
    - This is a very severe problem: we've wrecked the code of others, which trusted the integrity of our interface *GasAcceptor*.

- We hurt an important principle: we hurt the open close principle (OCP).
  - OCP says, that UDTs should be open for extension, but closed for modification.
  - We hurt OCP, because we modified *GasAcceptor* rendering all implementors unusable!

Interfaces are getting incompatible – Part 3 – Solution: default Methods

- The idea of default methods follows this thought:
  - "Do the added methods make sense for every implementor? – If not, we could provide reasonable default implementations."
  - E.g. *GasAcceptor.supportsOctaneRating()* may not make sense for a lot of *GasAcceptor*s, which already exist (e.g. *GasJerryCan*).

- In our case *supportsOctaneRating()* does not make sense for a lot of *GasAcceptor*s, which already exist.
  - Therefor we make *GasAcceptor.supportsOctaneRating()* a default method to not force others to implement it:

```
// <GasAcceptor.java>
public interface GasAcceptor {
        void fillGas(double liters);
        default boolean supportsOctaneRating(double RON) {
                return true;
        }
}
```

- After the default method was implemented, *GasCar* is again a valid implementor of *GasAcceptor*:
  - *GasCar* compiles successfully and this code is valid:

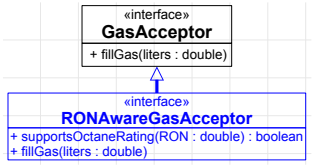
```
// We can again create GasCar instances:
GasAcceptor gasAcceptor = new GasCar();
gasAcceptor.fillGas(32.6);
```

  - And we can call *GasAcceptor.supportsOctaneRating()*, because *GasCar* got the default implementation for free:

```
boolean supportsSuperGas = gasAcceptor.supportsOctaneRating(95.0);
// supportsSuperGas = true
```

Interfaces are getting incompatible – Part 4 – Solution: inherit new Interfaces

- A better solution in this case, is strictly following the OCP: we just create a new interface, inheriting from *GasAcceptor*:

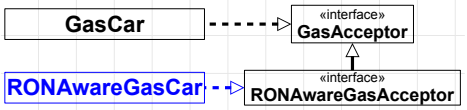



```
// <RONAwareGasAcceptor.java>
public interface RONAwareGasAcceptor extends GasAcceptor {
        boolean supportsOctaneRating(double RON);
}
```

... and add a new overload of Garage.refuel()

```
// <Garage.java>
public class Garage { // (other members omitted)
        public void refuel(RONAwareGasAcceptor RONAwareGasAcceptor, double liters, double RON) {
                // pass
        }
}
```

- This solves our problems completely, because we haven't modified *GasAcceptor*, but extended it.
  - All implementors of *GasAcceptor* are still working.
  - New UDTs can then select to implement *RONAwareGasAcceptor* or just *GasAcceptor*.
    - Let's create the UDT *RONAwareGasCar*:




```
RONAwareGasAcceptor acceptsSuperGas = new RONAwareGasCar(95);
joesGarage.refuel(acceptsSuperGas, 45.0, 95.0);
```

```
// <RONAwareGasCar.java>
public class RONAwareGasCar extends Car implements RONAwareGasAcceptor {
        private double liters;
        private final double minimalAcceptedOctaneRating;

        public RONAwareGasCar(double minimalAcceptedOctaneRating) {
                this.minimalAcceptedOctaneRating = minimalAcceptedOctaneRating;
        }
        @Override
        public void fillGas(double liters) {
                this.liters = liters;
        }
        @Override
        public boolean supportsOctaneRating(double RON) {
                return RON >= minimalAcceptedOctaneRating;
        }
}
```

Some Words on Type Naming

- Esp. on the last slide the way type names "develop" in oo architectures is showing up clearly.
- The idea is to reflect specific aspects of types in their name, by joining words:
  - E.g. *GasAcceptor* -> ***RONAware****GasAcceptor* and *Car* -> ***Gas****Car* -> ***RONAware****GasCar*
  - Java's convention to use PascalCase and camelCase underscores the will to follow this idea.
- Usually chaining word together for new type names is a good idea, but it become difficult to handle.
  - When names are getting too long due to chaining, something could be wrong with the naming or the architecture.
  - However, this naming style is used for *Exception* types in the JDK all over, so for *Exception*s it seems to be fine…
- The chaining naming style for types is mainly influenced from Smalltalk's conventions and traditions.
  - See Kent Beck's book "Smalltalk best practice patterns" from 1996.

Interfaces are getting incompatible – overall Summary

- The simple rule is: public interfaces should never be modified! This rule directly follows the OCP.
  - A modified interface generally breaks all implementors of the original interface!
  - We assume, that modification means added methods. Removed methods or modified signatures are not acceptable at all!

- Java provides three solutions to handle situations, when we as developers think an interface should be modified:
  - (1) Make all present implementors abstract classes.
    - Solution (1) seems not to be a feasible solution, because all implementors must change, even those not programmed by us.
  - (2) Create a new interface, which inherits the old one. This should be your preferred solution, because:
    - This solution is directly portable to many other oo languages.
  - (3) Make the added methods default methods. This is no preferred solution, because:
    - This solution is not directly portable to many other platforms/languages. (Gradually some languages get equipped with mixins or extension methods.)
    - The added methods are not clearly separated. – The interface was still modified, but we make "default assumptions" about the modifications.
    - (One could say, that default methods seem like a "dirty hack". default methods are sometimes called defender methods!)
  - => The solution (2), new inherited interfaces, seems the simpler solution. So: rather avoid solution (3), default methods.

- Guideline to minimize the need to modify interfaces: keep them as small as possible! Smaller interfaces are more stable!
  - This principle is called the interface segregation principle (ISP).

Overriding default Methods

- default methods can be overridden in a UDT like all interface-methods.

```
// <RONAwareGasCar.java>
public class RONAwareGasCar extends Car implements GasAcceptor {
        private double liters;
        private final double minimalAcceptedOctaneRating;

        public RONAwareGasCar(double minimalAcceptedOctaneRating) {
                this.minimalAcceptedOctaneRating = minimalAcceptedOctaneRating;
        }

        @Override
        public void fillGas(double liters) {
                this.liters = liters;
        }

        @Override
        public boolean supportsOctaneRating(double RON) {
                return RON >= minimalAcceptedOctaneRating;
        }
}
```

  - This allows another explanation of default methods: they make overriding/implementing interface-methods optional.

Direct Call of default Methods

- Additionally, we can also call the original default method of the interface from within an implementing UDT!
  - We can even call the original default method from within the *@Override* of that default method!
  - E.g. we *@Override GasAcceptor.supportsOctaneRating()* to add a log message, but "delegate" to the original default method:

```
// <RONAwareGasCar.java>
public class RONAwareGasCar extends Car implements GasAcceptor {
        private double liters;
        private final double minimalAcceptedOctaneRating;

        public RONAwareGasCar(double minimalAcceptedOctaneRating) {
                this.minimalAcceptedOctaneRating = minimalAcceptedOctaneRating;
        }

        @Override
        public void fillGas(double liters) {
                this.liters = liters;
        }

        @Override
        public boolean supportsOctaneRating(double RON) {
                System.out.printf("Can be refueled with gas of %.2fl octane rating.%n", RON);
                return GasAcceptor.super.supportsOctaneRating(RON);
        }
}
```

```
// <GasAcceptor.java>
public interface GasAcceptor {
        void fillGas(double liters);
        default boolean supportsOctaneRating(double RON) {
                return true;
        }
}
```

call delegated to default method

- The expression *GasAcceptor.super.supportsOctaneRating(RON)* does the trick and calls the default method directly:

```
GasAcceptor acceptsSuperGas = new RONAwareGasCar(95);
boolean isNormalGasAcceptable = acceptsSuperGas.supportsOctaneRating(81);
// isNormalGasAcceptable = true
// >Can be refueled with gas of 81.00l octane rating.
```

Summary on default Methods

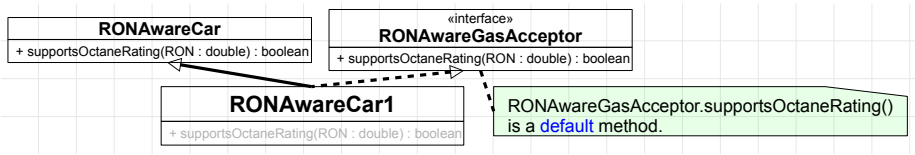
- Implementation side of default methods:
  - The code of default methods has no special constraints.
  - Esp. the code of default methods is allowed to call other methods of the defining interface.
    - This feature is crucial for Java's *Streams* on the JDK interface *Iterable*. It allows to implement a template method design pattern.
  - Methods inherited from *Object*, like *Object.toString()* or *Object.equals()* can not be "defaulted".

- Usage side of default methods:
  - default methods can not be called directly of its defining interface from "outside".
    - Sure, because we can not have instances of an interface!
    - Instead, the interface in question must be implement from another UDT and on instances of that UDT default methods can be called.
  - But default methods can be directly called from the implementing UDT of the default method's interface.
    - Syntax for the direct call: *<interfaceName>.super.<defaultMethodName>()*
  - default methods can be overridden like any other method derived from an interface.
  - Even if a default method is overridden, the default method of the interface can still be directly called from an implementing UDT.

- Other names for the "concept" of default methods:
  - default methods enable interface extensibility w/o breaking implementors, so they're sometimes called virtual extension methods.
  - default methods shield established implementors from modification in interfaces, so they're also called defender methods.

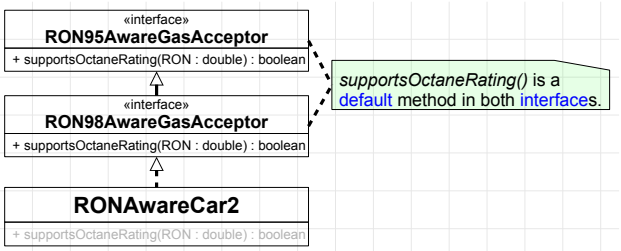
Interfaces – Multiple Inheritance and default Methods – Inherited Method or default Method?

- In a past lecture we've already discussed, that interfaces allow some kind of MI in Java.
  - We have to discuss this topic once again after the introduction of default methods.



- (1) The class *RONAwareCar1* extends another class *RONAwareCar* with the method *supportsOctaneRating(*double*)* and implements the interface *RONAwareGasAcceptor* with the default method *supportsOctaneRating(*double*)*.
  - Which *supportsOctaneRating(*double*)* is effective in *RonAwareCar1*?
  - The implementation of an extended class always "wins": *RONAwareCar.supportsOctaneRating(*double*)* is effective in *RONAwareCar1*.

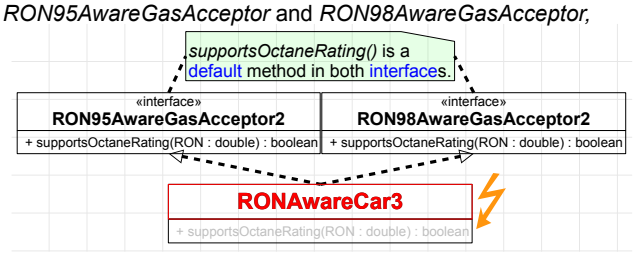
Interfaces – Multiple Inheritance and default Methods – Which default Method in a Hierarchy?



- (2) The class *RONAwareCar2* implements the interface *RON98AwareGasAcceptor*, which inherits *RON95AwareGasAcceptor*. Both interfaces have a default method *supportsOctaneRating(*double*)*.
  - Which *supportsOctaneRating(*double*)* is effective in *RonAwareCar2*?
  - The default method of the interface on the hierarchal layer nearest to the class always "wins": *RON98AwareGasAcceptor.supportsOctaneRating(*double*)* is effective in *RonAwareCar2*.

Interfaces – Multiple Inheritance and default Methods – Which default Method from multiple Interfaces?

- (3) The class *RONAwareCar3* implements the interfaces *RON95AwareGasAcceptor* and *RON98AwareGasAcceptor,* which both have a default implementation of *supportOctaneRating(*double*)*.
  - Which *supportsOctaneRating(*double*)* is effective in *RonAwareCar3*?
  - Neither is effective! This will result in a compile time error!
  - => Instead we have to implement *RonAwareCar3.supportsOctaneRating(*double*)* explicitly!



- In case (3) we have to explicitly *@Override supportsOctaneRating(*double*)* in *RONAwareCar3*! There are two alternatives:
  - (a) Just *@Override* it with any code.

```
// <RONAwareCar3a.java>
public class RONAwareCar3a implements RON95AwareGasAcceptor2, RON98AwareGasAcceptor2 {
        @Override
        public boolean supportsOctaneRating(double RON) {
            return RON <= 100;
        }
}
```

  - (b) *@Override* it and call the default methods of the interfaces.

```
// <RONAwareCar3b.java>
public class RONAwareCar3b implements RON95AwareGasAcceptor2, RON98AwareGasAcceptor2 {
        @Override
        public boolean supportsOctaneRating(double RON) {
            return RON95AwareGasAcceptor2.super.supportsOctaneRating(RON);
        }
}
```

Static Methods in Interfaces

- With the introduction of default methods, we get yet another aspect of reusability. Consider this code:

```
public interface GasAcceptor { // (members hidden)
        default boolean supportsOctaneRating(double RON) {
                return 0 <= RON;
        }
}
```

  - We extend the default method *GasAcceptor.supportsOctaneRating()*, so that it does at least a validity check for its argument.

- We could imagine other developers also being interested to check for RON validity, but the check is cast into stone …
  - To make the predicate 0 <= RON available for other developers we extract it into a static method in the interface *GasAcceptor*:

```
public interface GasAcceptor {
        static boolean isValidOctaneRating(double RON) {
                return 0 <= RON;
        }
        default boolean supportsOctaneRating(double RON) {
                return isValidOctaneRating(RON);
        }
}
```

  - Now we can easily use this predicate independent of instances implementing *GasAcceptor*:

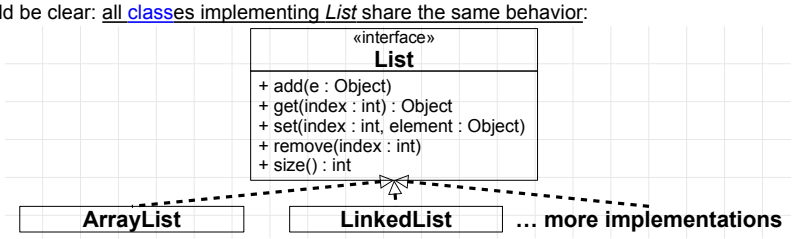
```
double RON = inputScanner.nextDouble();
if (GasAcceptor.isValidOctaneRating(RON)) {
        System.out.println("All right, you've entered a valid RON!");
}
```

- Hint: static methods are implicitly public in interfaces, since Java 9 those can also be private to support information hiding.

- static interface methods were added to support programming default methods, esp. to DRY. Esp. the functional programming tools, that came with Java 8 offer many useful static interface methods like *Function.identity()* or *Comparator.reversedOrder()*.

Introducing the DIP: The List Interface – Part 1

- As we noticed in several places, interfaces are a very pervasive idiom in Java.
- We will now revisit the class *ArrayList*. *ArrayList* implements the interface *List*.
  - The idea should be clear: all classes implementing *List* share the same behavior:



  - *ArrayList* implements *List* using an array.
  - *LinkedList* implements *List* using a tree of connected nodes forming a doubly linked list.
- What we have gained is, that due to the substitution principle, we can use *ArrayList* or *LinkedList*, whenever *List* is used:

```
List busses = new ArrayList();
// Add elements:
busses.add(new Bus(150.0));
busses.add(new Bus(230.2));
// Get an element:
Bus firstBus = (Bus) busses.get(0);
```

```
List busses = new LinkedList();
// Add elements:
busses.add(new Bus(150.0));
busses.add(new Bus(230.2));
// Get an element:
Bus firstBus = (Bus) busses.get(0);
```

  - For the rest of the code it doesn't matter, if *busses* is backed by an *ArrayList* or *LinkedList*, as long as it implements *List*.

Introducing the DIP: The List Interface – Part 2

- Because *List* is a very top-level interface and we typically just use *List*'s methods instead of special *ArrayList* methods, we can apply the dependency inversion principle (DIP), which is one of the SOLID principles.

```
// DIP at return types
List cars = getCarsInStock();
Car firstCar = (Car) cars.get(0)

public List getCarsInStock() {
        LinkedList busses = new LinkedList();
        busses.add(new Bus(150.0));
        busses.add(new Bus(230.2));
        return busses:
}
```

```
// DIP at parameter types
ArrayList gasCars = new ArrayList();
gasCars.add(new GasCar(100));
gasCars.add(new GasCar(105));
fillEmUp(gasCars, 30);

public void fillEmUp(List gasCars, double maxLiters) {
        for (Object item : gasCars) {
            ((GasCar)item).fillGas(maxLiters);
        }
}
```

- DIP: let types only depend on abstract types (interfaces), never on concrete types (implementations) to reduce coupling.
  - The code calling *getCarsInStock()* only depends on *List*, not on *LinkedList*, which is actually returned.
  - The code of *fillEmUp()* only depends on *List*, not on *ArrayList*, which is actually passed.
  - We have effectively decoupled callers of *getCarsInStock()* and the code in *fillEmUp()* from any concrete *List*-implementation!

- The substitution principle on abstract classes as well as on interfaces allow implementation of the DIP in Java

Guidelines:
(1) Use the most abstract type, which is sufficient for the task in parameter types, return types and ideally also in field types.
(2) Use the most abstract type, which is sufficient for the static/reference types in the code of methods.

Marker Interfaces

- interfaces can be used to mark a class for dynamic type checks, so that special features can be checked.
  - This usage of interfaces is weird, because marker interfaces are often just empty without super interfaces.

- Consider, we want to mark *Car* types, which have winter facilities such as a park heating, seat heater and so forth.
  - The idea is to handle such *Car*s specially: *Car*s with winter facilities could be parked outside, others under the car port:

```
// <WinterFacility.java>
public interface WinterFacility {
        // empty
}
```

```
// <ArcticCar.java>
public class ArcticCar extends Car implements WinterFacility {
        // empty
}
```

```
// <Concierge.java>
public class Concierge {
        public void park(Car car) {
                if (car instanceof WinterFacility) {
                        parkOutside(car);
                } else {
                        parkAtCarPort(car);
                }
        }
}
```

- Marker interfaces are an exception to the rule, that interfaces are used as static type, here we check for its dynamic type!

- There is a problem with marker interfaces: sub classes of a class implementing this interface are implicitly also marked.
  - I.e. we cannot "unmark" sub classes.
  - To solve this problem Java 5 introduced another concept to mark classes in isolation: annotations.

- In some places the JDK uses marker interfaces since older versions of Java: *Serializable*, *Cloneable*

- *ArrayList* implements the marker interface *RandomAccess*.

Decorators and the optional Feature Pattern – Part 1

- There can be situations, in which an interface must be implemented formally, but not all methods can be implemented.
- Java officially supports a way out with the so called optional implementation/feature pattern.
- Example: The JDK provides a type of *List*, which can only be instantiated with content, but not be modified afterwards.
  - This is done by creating an object of a class implementing *List*, e.g. *ArrayList* and passing this *List* to *Collections.unmodifiableList()*.

  - *Collections.unmodifiableList()* returns a new object implementing *List*, which cannot be modified (elements added or removed).
    - "Unmodifiyable" means, that the size of the returned *List* cannot be changed afterwards.
  - Notice, that *Collections.unmodifiableList()* accepts and returns the very abstract type *List*: this perfectly fits the DIP!
  - *java.util.Collections* is the companion class for Java's collections like *List*s.
- The object returned from *Collections.unmodifiableList()* implements *List*, but some *List*-methods just don't work.
  - I.e. those *List*-methods, which modify the *List* throw an *UnsupportedOperationException* when called:

```
List names = new ArrayList();
names.add("Joe");

List fixedNames = Collections.unmodifiableList(names);
// This won't work:
fixedNames.add("Beau"); // Invalid! Throws java.lang.UnsupportedOperationException
```

Decorators and the optional Feature Pattern – Part 2

- How does *Collections.unmodifiableList()* work?
  - It wraps a passed *List* into an object of another class implementing *List*, but strictly throwing *UnsupportedOperationException* on some methods.
  - And those "some methods" are exactly those methods, which allow modification of a *List*. The implementation looks like this:

```
public class Collections {  // (members hidden)
        static class UnmodifiableList implements List {  // (members hidden)
                private final List list;
                UnmodifiableList(List list) {
                        this.list = list;
                }
                @Override
                public Object get(int index) {
                        return list.get(index);
                }
                @Override
                public void add(int index, Object element) {
                        throw new UnsupportedOperationException();
                }
        }
}
```

  - As can be seen the idea is that non-modifying methods like *get()* are just getting relayed to the wrapped list and others throw.

- With this wrapper type *UnmodifiableList*, *Collections.unmodifiableList()*'s implementation is trivial:

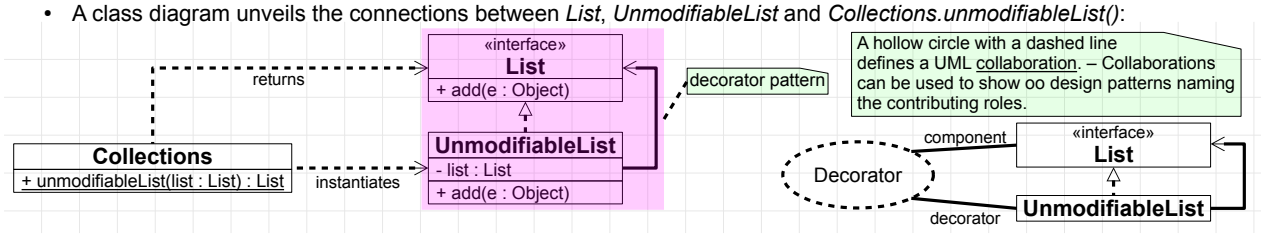
```
public class Collections {  // (members hidden)
        public static List unmodifiableList(List list) {
                return new UnmodifiableList(list);
        }
}
```


  - Notice, that *UnmodifiableList* has package access in *java.util*. It makes no sense to be public, because it is never exposed!

Decorators and the optional Feature Pattern – Part 3

- An *UnmodifiableList* wraps a *List* and limits access to this *List* in a special manner: this is Java's optional feature pattern.
  - This pattern says, that a sub type may not *@Override* all methods functionally, but throw *UnsupportedOperationException* instead.
  - The idea is, that objects with limited functionality keep their interface to stay backward compatible obeying DIP/substitution principle.

- A class diagram unveils the connections between *List*, *UnmodifiableList* and *Collections.unmodifiableList()*:



  - We show *UnmodifiableList* as (public) top-level class: the UML offers no notation for nested non-public classes (2023).

- The connection between *List* and *UnmodifiableList*: *UnmodifiableList* implements *List* and it has a *List*.
  - This type constellation is an oo design pattern: here the wrapper design pattern, also called decorator design pattern.
  - Decorator is used in many places in the JDK, e.g. the collections framework and Java's i/o-libraries.
  - Decorator is simple to understand and use and it allows powerful designs obeying the SOLID* principles.
    - *It should be said, that the optional feature pattern doesn't follow the LSP because it disables features of the super interface.

- Warning: A UML class diagram showing a collaboration, which looks like decorator pattern need not to be a decorator pattern! – Mind that (multiple) interfaces could be implemented by a certain class for different reasons!

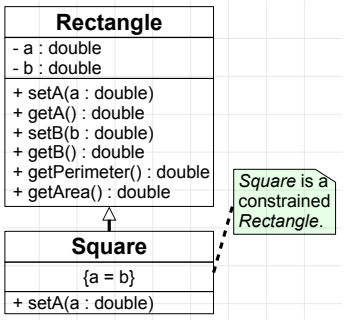
The Truth about Object Orientation!

- When we introduced oo, we said, that oo has the core idea of simulating the reality.
- This is correct to a certain degree, but "oo simulating the reality" is basically a marketing term.
- (1) When we program "more serious" code/solve a customer's problems we rather utilize oo tools, than simulate realities.
  - An example is implementing *Comparable*, were we utilize polymorphism, but this not a "reality concept".
  - The idea of *Comparable* is more the abstraction of a behavior.
  - To be fair, one of the first oo languages was Simula, it was named this way for a reason!
- (2) A benefit said of oop is, that code is more maintainable.
  - This can be correct, but only if the maintainer knows the design principles behind the code to be maintained.
  - This often fails, because one thinks objects are self-contained and there is no design, but objects need context/other dependencies.
  - E.g. our *Car*-example can be extended, but the idea of, e.g., the *hasStarter()*-pattern or the many interfaces must be understood.
- (3) Sometimes inheritance is added into a type hierarchy to reuse code or save space (DRY OOP trap after Jeff Ward).
  - (3a) Either sub classing is applied to reuse super class code,
  - (3b) or if a method/code is used in more than one sub class -> move it to the super class.

The Truth about Object Orientation – the dreaded Square-Rectangle Example – Part 1

- A well known example on how oo-design can be suspicious is the "dreaded" square and rectangle example.
  - Assume the classes *Square* and *Rectangle* representing squares and rectangles of the reality: How should they be modeled?

- Idea 1: a square is a special rectangle. The specialty: all it's sides have same length. ***Square* shall sub class *Rectangle*.**

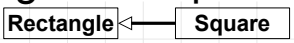



```
// <Rectangle.java>
public class Rectangle {
        private double a;
        private double b;

        public void setA(double a) {
            this.a = a;
        }
        public void setB(double b) {
            this.b = b;
        }
        public double getB() {
            return b;
        }
        public double getA() {
            return a;
        }
        public double getPerimeter() {
            return 2 * a + 2 * b;
        }
        public double getArea() {
            return a * b;
        }
}
```

```
// <Square.java>
public class Square extends Rectangle {
        @Override
        public void setA(double a) {
            // force the constraint {a = b}:
            super.setA(a);
            super.setB(a);
        }
}
```

The Truth about Object Orientation – the dreaded Square-Rectangle Example – Part 2



- That a square is a special rectangle reflects the reality, but substitutability can lead to surprises:
  - This one is clear:

```
Rectangle myRoom = new Rectangle();
myRoom.setA(4);
myRoom.setB(5.6);
System.out.println("My room - a: "+myRoom.getA()+", b: "+myRoom.getB());
// >My room - a: 4.0, b: 5.6
```

  - This one is also clear:
    - Setting *a* also sets *b*.

```
Rectangle myOtherRoom = new Square();
myOtherRoom.setA(5);
System.out.println("My other room - a: "+myOtherRoom.getA()+", b: "+myOtherRoom.getB());
// >My other room - a: 5.0, b: 5.0
```

  - Assume the method *fitWidth()*, which sets the side *b* (the width) to a fixed value of 2:

```
public static void fitWidth(Rectangle room) {
        room.setB(2);
}
```

  - No problem with a *Rectangle*:

```
fitWidth(myRoom);
System.out.println("My fitted room - a: "+myRoom.getA()+", b: "+myRoom.getB());
// >My fitted room - a: 4.0, b: 2.0
```

  - But setting *b* on a *Square* hurt the constraint {a = b}:

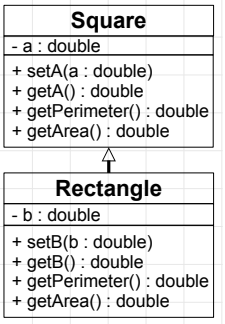
```
fitWidth(myOtherRoom);
System.out.println("My fitted other room - a: "+myOtherRoom.getA()+", b: "+myOtherRoom.getB());
// >My fitted other room - a: 5.0, b: 2.0
```

  - => We have a *Square*, which is actually a *Rectangle* with {a <> b}!
- I.e. *Square* is not a real substitute for *Rectangle*, because *a* and *b* are connected in *Square* via a constraint ({a = b}).
  - The problem: *Square* requires constraints of the super class, but misses to force them: e.g. we didn't *@Override setB()*!

- Such a design can quickly introduce bugs! Consider, that all methods of the sub class must adhere to the constraint! – If something was not overridden in the sub class to handle the constraint correctly, it might fail!

The Truth about Object Orientation – the dreaded Square-Rectangle Example – Part 3

- Idea 2: a rectangle has "a side more" than a square. Let's to borrow *Square*'s side *a*. ***Rectangle* shall sub class *Square*.**



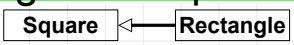

```
// <Square.java>
public class Square {
        private double a;

        public void setA(double a) {
            this.a = a;
        }
        public double getA() {
            return a;
        }
        public double getPerimeter() {
            return 4 * a;
        }
        public double getArea() {
            return a * a;
        }
}
```

```
// <Rectangle.java>
public class Rectangle extends Square {
        private double b;

        public void setB(double b) {
            this.b = b;
        }
        public double getB() {
            return b;
        }
        @Override
        public double getPerimeter() {
            return 2 * super.getA() + 2 * b;
        }
        @Override
        public double getArea() {
            return super.getA() * b;
        }
}
```

The Truth about Object Orientation – the dreaded Square-Rectangle Example – Part 4



- That *Rectangle* inherits from *Square* was done for economic reasons, but it doesn't reflect the reality.
- Consider:

```
Rectangle aRectangle = new Rectangle();
aRectangle.setA(4);
aRectangle.setB(5.6);
```

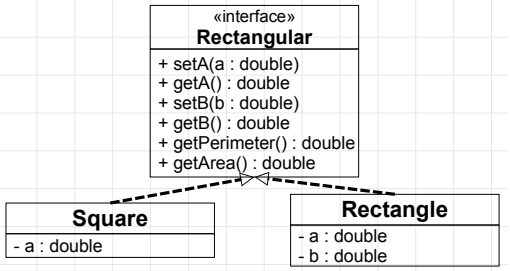
  - But the substitutability of a *Square* by a *Rectangle* is really strange, but it compiles of course:

```
Square aSquare = aRectangle;
```

- In oo terms we say: we cannot formulate a good discriminator of the sub class *Rectangle*.
  - A discriminator defines, in which aspects sub classes differ from the super class.
  - (To be frank, it is also be difficult to find a discriminator, when *Square* is derived from *Rectangle*…)
- So, this design is more economic, but not understandable for people as simulation of the reality.
- All right, is there a "more correct" "solution" for the rectangle-square design?

The Truth about Object Orientation – the dreaded Square-Rectangle Example – Part 5

- There is no "correct" "solution"! There can be more-or less suitable solutions depending on the problem, we have to solve.
- A completely other approach is to assume, that rectangle and square have no relation at all!
- We can instead assume, that rectangle and square share a lot of common behavior we put into the interface *Rectangular*:




```
// <Rectangular.java>
public interface Rectangular {
        void setA(double a);
        double getA();
        void setB(double b);
        double getB();
        default double getPerimeter() {
            return 2 * getA() + 2 * getB();
        }
        default double getArea() {
            return getA() * getB();
        }
}
```

  - Mind, that we didn't do this to put common code into a common super type just to reuse code, but to reflect common behavior!
  - *Rectangular* is no abstract class, because we want to let sub classes decide how they design sides as fields.
  - Nevertheless, we could make *Rectangular.getPerimeter()* and *Rectangular.getArea()* default methods.
  - One could argue, that *Square* must implement superfluous methods now … but this is a matter of perspective.

29

- Mind the shift of focus: since *Square* and *Rectangle* now have a common interface, we can treat *Square* and *Rectangle* the same way, when we base our algorithms only on this common interface. → DIP: only depend on abstract types and what could be more abstract than an interface...

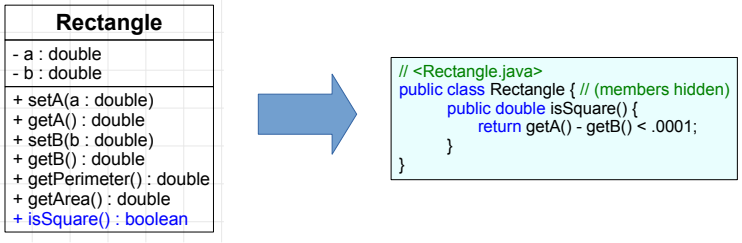
The Truth about Object Orientation – the dreaded Square-Rectangle Example – Part 6

- Thinking of rectangle and square being represented by different types can be misleading!

- When we assume, that, e.g., a *Rectangle* is modifiable as object it could become a square at run time!

```
Rectangle myRoom = new Rectangle();
myRoom.setA(4);
myRoom.setB(5.6);
// myRoom is now rather a square:
myRoom.setB(4);
```

  - With this assumption we could put the rectangle- and square-concept into a single class and introduce the predicate *isSquare()*.




```
// <Rectangle.java>
public class Rectangle { // (members hidden)
        public double isSquare() {
                return getA() - getB() < .0001;
        }
}
```

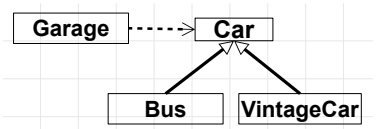
```
Rectangle myRoom = new Rectangle();
myRoom.setA(6);
myRoom.setB(6);
boolean squareRoom = myRoom.isSquare();
// squareRoom = true
```

The Liskov Substitution Principle

- The Liskov Substitution Principle (LSP) tells us: a type can be substituted by another type, if it offers the same behavior.
  - Barbara Liskov is a computer scientist and mathematician, who formulated this rule dedicatedly.

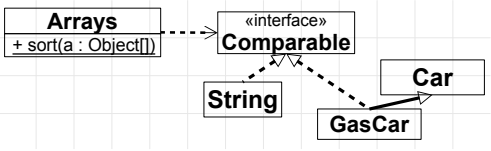
- In Java, we easily consider the LSP this with sub typing, more exactly sub classing:

```
// <Garage.java>
public class Garage { // (other members omitted)
        public void refuel(Car car, double liters) {
                car.fillIn(liters);
        }
}
```



  - An instance of any sub class of *Car* can be used as argument for *Garage.refuel()*, because any sub class of *Car* is a substitute!
  - The sub class relation guarantees, that substitute types offer compatible behavior.
  - This guarantee also includes a guarantee of backward-compatibility: any sub class of *Car* is compatible to *Car*.

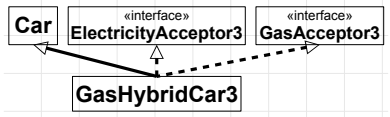
- But sometimes, we want to exploit the LSP/backward-compatibility also for unrelated types.
  - Consider *Arrays.sort()*, that accepts an *Object[]*. The only restriction on the *Objects* in the array: their dynamic type must be *Comparable*.
  - The concept *Comparable* does not demand a relation, but just the bare common behavior, which satisfies the LSP perfectly.
  - The LSP for unrelated types can be considered with interfaces.



Polymorphism revisited with Interfaces – Part 1

- (Oo) Polymorphism is the ability to call methods of a dynamic type through a static type.
  - "Through a static type" means through a reference of static type.
  - The difference between static and dynamic type is to be strengthened here.
  - The static type must be a polymorphic type, i.e. the type of a super class or interface in Java.
  - The actual object "behind" the reference must be an instance of a class specializing/concertizing the static type, the dynamic type.

- An object can have multiple dynamic types, consider:




```
Car myCar = new GasHybridCar3();
// Some dynamic type checks:
boolean allDynamicTypes = myCar instanceof Car && myCar instanceof Object
                          && myCar instanceof GasHybridCar3 && myCar instanceof ElectricityAcceptor3
                          && myCar instanceof GasAcceptor3;
// allDynamicTypes = true
```

- So, super classes and interfaces are mainly used as static types, esp. because there can be no instances.
  - Notable exception: when super classes or interfaces are used in dynamic type checks, as seen in the code above.

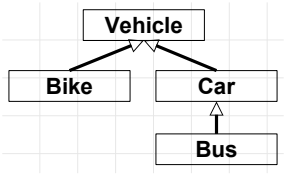
Polymorphism revisited with Interfaces – Part 2

- The main point in oo is using polymorphism to express substitutability, not inheritance to reuse code or safe space!
  - The truth about abstract classes: abstract classes have basically only be added to Java to support polymorphism.
- The idea of polymorphism is to allow building growing and modifiable program structures/architectures.
  - Oo languages provide tools, which supports this idea, we just need to know, how to use those tools.
  - Java's polymorphism only works on references of polymorphic types! interfaces and super classes are such types.
- interfaces should be applied to isolate relatively fixed parts from varying parts of the architecture.
  - The way fixed and varying parts are arranged often shows a certain pattern, which shows in many other places.
  - We call those (oo) design patterns. There exist predefined descriptions of well known design patterns.
  - One can start designing programs by applying design patterns or by recognizing design patterns during programming.
- An example of an isolating interface is *Comparable*: it shields the comparison algorithm from the working algorithm.
  - It is often said "an extra layer of indirection" is required for polymorphism to work, this is exactly *Comparable*'s function!

Type Navigation – Part 1

- Using interfaces instead of super classes for abstraction also has visible but not so obvious effects on navigability.

- Navigability means the ability to switch from one dynamic type to another dynamic type of an object.
  - We can understand this as "switching from one type to another type".
  - Navigation between types in Java means assigning or casting an object's reference to a compatible type.
  - Consider following hierarchy:



  - The allowed navigations between those types in Java is pretty simple to get:

```
Car car = new Car();
Bus bus = (Bus) car; // Could be fine… checked at run time
```

  - The class-hierarchy also forbids some navigations, e.g. we cannot cast a *Car* to a *Bike*: those classes/types are unrelated!

```
Bike bike1 = (Bike) car; // Won't compile!
```

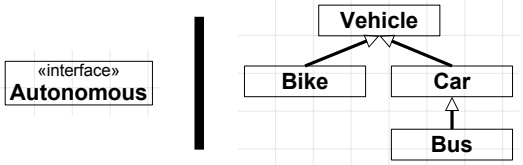
  - Using cast "contact lenses", navigating via *Vehicle*, we could fool the compiler here:

```
Bike bike2 = (Bike) (Vehicle)car; // Well… will be checked and fail at run time with a ClassCastException
```

    - But we can't fool Java's typesafe runtime, the cast will fail at run time with a *ClassCastException!*

Type Navigation – Part 2

- The broader navigability aspect we have with interfaces compared to classes is quite simple to unleash now.
  - Consider following hierarchy: the interface *Autonomous* is not implemented by any class on the right side, there is no relation:



- Surprisingly, the Java compiler allows almost all conversions from any class type to any interface type.
  - Java doesn't rely on any relation, instead it assumes, that a reference could have some dynamic type implementing the interface.
  - All following navigations are ok for the Java compiler:

```
Autonomous autonomous = (Autonomous) car; // OK!
Bus bus = getBus();
Autonomous autonomous2 = (Autonomous) bus; // OK!
Bike bike = getBike();
Autonomous autonomous3 = (Autonomous) bike; // OK!
```

    - The compiler assumes, that a *Car*, *Bus* or *Bike* reference could refer to sub classes implementing *Autonomous*, which are yet unknown at compile time.
    - But those sub classes could be known at run time.

- There is one situation, in which this freedom in interface-navigation is not given: if a class is final, it cannot have any sub classes and interface-navigation is stopped by the compiler:

```
FinalBus bus = getFinalBus();
Autonomous autonomous4 = (Autonomous) finalBus(); // Won't compile.
```

- There is another situation when there is only limited freedom in interface-navigation, namely when sealed interfaces are used, which were introduced with Java 15. A sealed interface actually knows all its implementing types, it allows exhaustive pattern matching in Java 15+.

Thank you!