(5) Java Abstractions – Part 1

Nico Ludwig (@ersatzteilchen)

TOC

- Cited Literature:
  - Just Java, Peter van der Linden
  - Thinking in Java, Bruce Eckel

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!

A new Challenge: Refueling Cars in Garages

- We've already discussed *joesGarage* (of type *Garage*). Meanwhile *joesGarage* offers another service: refueling!

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

```
// <Car.java>
public class Car {
        private double tankedLiters;

        public void fillIn(double liters) {
                this.tankedLiters = liters;
                System.out.printf("Filled with %.2f l%n", liters);
        }
}
```

  - *Garage* and *Car* have been modified to support refueling, so that every *Car* can be refueled in *Garages*!
    - The new method *Garage.refuel(Car)* represents the functionality of the fuel dispenser in *Garage*.
    - The new method *Car.fillIn(double)* is self-explanatory. Every sub-type of *Car* now and in future inherits/will inherit *Car.fillIn()*:

```
// Garage.refuel() calls Car.fillIn(double):
joesStation.refuel(seatLeon, 20.0);
// Filled with 20.00l.
```

```
// Garage.refuel() calls (the inherited) Car.fillIn(double):
joesStation.refuel(fordTinLizzy, 5.6);
// Filled with 5.60l.
```

- So far so good! But we missed some facts about refueling in the real world:
  - (1) Not each *Car* can be refueled with the same fuel-type!
    - E.g. *Car*s can have gas or diesel internal combustion engines.
  - (2) Certain *Car*s cannot be refueled in *Garages*, just think about *SolarCar*!
  - (3) There are items, which are no *Car*s, but could be refueled, e.g. *JerryCan*s.

"Refueling" adds another vector of change (we have new *Car* types appear): different ways to refuel different things! It shows: the vector of change is often not what we think it is! And there can also be many vectors of change in an architecture. It also depends on how customers use our stuff, which is hard to predict.

Refueling Cars in Garages – Solution 1: multiple super Classes

- In oo design such a situation could be handled by adding an additional layer of indirection.
  - We will introduce some new abstract classes, each per fuel type and call them acceptors, e.g. *GasAcceptor* and *DieselAcceptor*
  - The idea is, that each class, which accepts the respective fuel type must inherit the matching abstract class.
  - Then we rewrite *Garage*, so that it can refuel only fuel acceptors, which are supported.

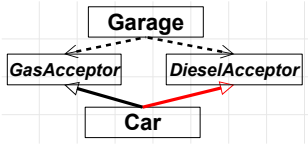
```
// <Garage.java>
public class Garage { // (other members omitted)
        public void refuel(GasAcceptor gasAcceptor, double liters) {
            gasAcceptor.fillGas(liters);
        }

        public void refuel(DieselAcceptor dieselAcceptor, double liters) {
            dieselAcceptor.fillDiesel(liters);
        }
}
```

```
// <GasAcceptor.java>
public abstract class GasAcceptor {
        private double currentGasLiters;

        public void fillGas(double liters) {
            this.currentGasLiters = liters;
        }
}
```

```
// <DieselAcceptor.java>
public abstract class DieselAcceptor {
        private double currentDieselLiters;

        public void fillDiesel(double liters) {
            this.currentDieselLiters = liters;
        }
}
```

- Now we can use *GasAcceptor* and *DieselAcceptor* as super classes for *Car*:

```
// <Car.java>
public class Car extends GasAcceptor, DieselAcceptor { // (other members omitted)
}
```



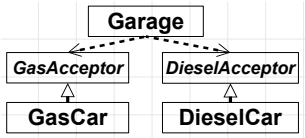
- But this solution won't work!
  - (1) A class in Java can not directly inherit from more than one super class! *Car* can't inherit from *GasAcceptor* and *DieselAcceptor*.
  - (2) In reality most *Car*s do need to either refuel gas or diesel, but not both.

Refueling Cars in Garages – Solution 2: parallel Class Hierarchies

- Another way to apply the additional layer of indirection is the creation of *GasAcceptor*- and *DieselAcceptor*-hierarchies.

```
// <GasCar.java>
public class GasCar extends GasAcceptor { // (other members omitted)
}
```

```
// <DieselCar.java>
public class DieselCar extends DieselAcceptor { // (other members omitted)
}
```



  - In the end we have completely removed the UDT *Car*, but invented two parallel hierarchies of *GasCars* and *DieselCars*.

- But with this solution we introduce other problems:
  - We would have to care for two hierarchies in future! E.g. we had to add the two types *GasBus* and *DieselBus*!
  - We would have to add more hierarchies for alternative types of fuel! E.g. *ElectricityAcceptor -> ElectricityCar -> ElectricityBus*.
  - After these changes in the *Car*-hierarchy, we would have to change *Garage*:
    - We have to remove *tryToStartCar(Car)*, because the UDT doesn't exist any longer.
    - On the other hand we have to add *tryToStartCar(GasCar)* and *tryToStartCar(DieselCar)*.
    - … and we would have to add more and more *tryToStartCar()*-methods for other types of fuel of the growing parallel hierarchy! E.g. *tryToStartCar(ElectricityCar)*.

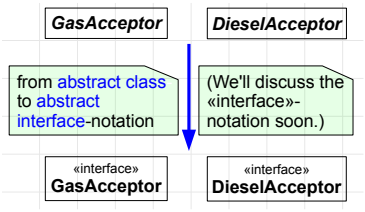
```
public class Garage { // (other members omitted)
        public void tryToStartCar(GasCar car) {
            // pass
        }
        public void tryToStartCar(DieselCar car) {
            // pass
        }
}
```

- The good news is, there exists another appropriate solution in Java: abstract interfaces.

Refueling Cars in Garages – Solution 3: Java's abstract Interfaces

- Actually, we are pretty much close to a suitable solution with an additional layer of indirection. Our findings:
  - Java does not support classes, which have more than one direct super class.
  - And this just means, that we cannot use multiple classes as additional layer of indirection to solve our problem.

- Finally, here we have our Java-way to solve such sort of problems with so called abstract interfaces.
  - (1) We strip all fields and all method implementations from *GasAcceptor* and *DieselAcceptor*.
  - (2) Finally we replace the keyword "abstract class" with "abstract interface".




```
// <GasAcceptor.java>
public abstract class GasAcceptor {
        private double currentGasLiters;

        public void fillGas(double liters) {
            this.currentGasLiters = liters;
        }
}
```

```
// <DieselAcceptor.java>
public abstract class DieselAcceptor {
        private double currentDieselLiters;

        public void fillDiesel(double liters) {
            this.currentDieselLiters = liters;
        }
}
```

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

```
// <DieselAcceptor.java>
public abstract interface DieselAcceptor {
        public abstract void fillDiesel(double liters);
}
```

- abstract interfaces strip a UDT down to, well, to its bare interface, i.e. to bare method declarations w/o implementations or fields.

Abstract Classes vs abstract Interfaces

```
// <GasAcceptor.java>
public abstract class GasAcceptor {
        private double currentGasLiters;

        public void fillGas(double liters) {
            this.currentGasLiters = liters;
        }
}
```



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

- abstract classes and abstract interfaces are both UDTs!
  - Neither abstract classes nor abstract interfaces can be instantiated with new: they are too abstract to have instances!
  - abstract interfaces cannot have fields, only a behavior remains: "*GasAcceptor* is something, which provides *fillGas(double)*"
  - abstract classes can have fields and fully implemented methods among abstract methods!

- Design: abstract classes allow to build a hierarchy of related types.
  - The idea of an abstract class is basically a class, that left things away, which must be completed somewhere else.
  - On the other hand: the abstract class tells us exactly, which stuff/methods must be completed!

- Design: abstract interfaces allow unrelated types to be used the same way, because they have the same behavior.
  - abstract interfaces strip an abstract class down to its bare interface, so everything must be completed somewhere else.
  - On the other hand: the abstract interface tells us exactly, which stuff/methods must be completed, namely all of them!

Abstract Interfaces – Motivation – Part 1

- What have we done?

- Some types do not only fulfill the super type's behavior, but other, maybe more general behaviors.
  - *GasCar* is a *Car* but it is also a "thing", that is consuming gas and should behave so.
  - So, we could introduce a new super class "*GasAcceptor*" to add this behavior, because *GasCar* accepts gas.
    - This class "*GasAcceptor*" would then, e.g., only have a field holding the *liters* of fuel in the tank.
  - But in Java we cannot let *GasCar* inherit from more than one super type! – What can we do instead?

- abstract interfaces to the rescue!

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

  - *GasAcceptor*'s methods to access/manipulate its fields must be left abstract, each *GasAcceptor* type differs...
  - The type *GasAcceptor* does rather define a behavior, than a concrete type.
  - Any type that behaves as *GasAcceptor* substitutes exactly like sub classes do, which enables polymorphism.

Abstract Interfaces – Motivation – Part 2

- abstract interfaces are UDTs, which only declare a behavior but have no functionality.
  - An abstract interface encapsulates no substantial data, so it cannot have instance-fields and it consumes no memory at run time.
  - I.e. abstract interfaces are even more abstract than abstract classes.

- Instead we've to create classes, which implement the abstract interfaces completely to have concrete classes, e.g. *GasCar*:

```
// <GasCar.java>
public class GasCar extends Car implements GasAcceptor, NextInterface, NextInterface2, ... {
        private double liters;

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

In the class definition, the super type (extends *Car*) must be written before the list of interfaces (implements *GasAcceptor*).

  - We specify abstract interfaces to be implemented at the top of a classes definition with the keyword implements instead of extends.
  - A class can implement multiple abstract interfaces, so a comma separated list of abstract interface names can be specified.
    - The written order of the specified abstract interfaces has no special meaning in Java.
  - The important thing: a class can inherit from a single super class and implement multiple abstract interfaces at the same time!
  - Generally, a class needs to override, or rather implement all methods of the abstract interfaces specified in the abstract interface list.

- Now *GasCar* is compatible to the abstract interface *GasAcceptor* and we can code this:

```
GasAcceptor gasAcceptor = new GasCar();
```

Abstract Interfaces – simple UML Notation

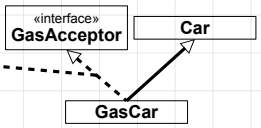
- The UML notation of abstract interfaces follows that of classes, but we've to add the stereotype "«interface»" to the name compartment.

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

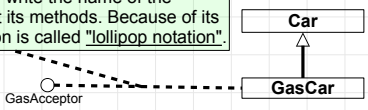
The "pseudo"-stereotype «interface» denotes a classifier to represent an abstract interface. Although abstract, abstract interfaces' names are usually not set in italics.

- A class implementing an abstract interface, can also be notated using UML:

To show that a class implements an abstract interface, we draw a dashed line with a hollow arrow tip pointing to the abstract interface.



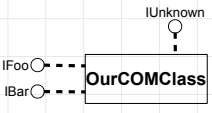
Alternatively, there exists a short notation. We can add a dashed line having a ball at its end to the classifier. On this end we write the name of the abstract interface without its methods. Because of its resemblance, this notation is called "lollipop notation".



  - The lollipop notation is useful, if we show a class implementing a lot of abstract interfaces!
  - Either notation shows basically the same important information: Another line fanning out from a box <-> another dependency.

Good to know
UML stereotypes are UML's way to define structural elements in a design. Virtually, stereotypes extend the UML, but «interface» doesn't extend the UML, but it rather extends our architecture, therefor «interface» is no real stereotype, but marks a special classifier.
We use french quotes, « and », so called guillements, to mark stereotype names. Interestingly, if we use those quotes in a flipped manner, i.e. » and then «, they are called chevrons.

The lollipop notation is prominent in COM (Component Object Model), where all classes must implement *IUnknown*, whose "lollipop" is written above the class box and all other interfaces' lollipops are written left from the class box.



Abstract Interfaces – How do they influence the UDT Garage?

- abstract interfaces are crucial in Java, in upcoming lectures, we discuss abstract interfaces in depth with many examples.

- Looking back to *Garage*, after the introduction of the abstract interfaces nothing needs to be changed:

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

```
// <DieselAcceptor.java>
public abstract interface DieselAcceptor {
        public abstract void fillDiesel(double liters);
}
```

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

        public void refuel(DieselAcceptor dieselAcceptor, double liters) {
            dieselAcceptor.fillDiesel(liters);
        }
}
```

  - The UDTs *GasAcceptor* and *DieselAcceptor* are still used as parameter types!
  - But outside of *Garage*, *GasAcceptor* and *DieselAcceptor* are abstract interfaces, no longer classes!
  - As can be seen abstract interfaces can basically be used like any other UDT.

- The important point is that *Garage* only calls methods defined in the UDTs *GasAcceptor* and *DieselAcceptor*.
  - Whether *GasAcceptor* and *DieselAcceptor* are (abstract) classes or abstract interfaces does not matter to *Garage* at this point.

- => In *Garage* it was not noticed, that *GasAcceptor* and *DieselAcceptor* are abstract interfaces now!

Abstract Interfaces – Separation of Usage and Implementation – Part 1

- *Garage* uses the abstract interfaces *GasAcceptor* and *DieselAcceptor*:
  - *Garage* knows nothing about their implementations like *GasCar*!

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

        public void refuel(DieselAcceptor dieselAcceptor, double liters) {
                dieselAcceptor.fillDiesel(liters);
        }
}
```

- The abstract interfaces just declare the callable public methods:

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

```
// <DieselAcceptor.java>
public abstract interface DieselAcceptor {
    public abstract void fillDiesel(double liters);
}
```

- Implementations of *GasAcceptor* and *DieselAcceptor*:
  - *GasCar* doesn't know about its usage in *Garage*.

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

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

```
// <DieselCar.java>
public class DieselCar extends Car
                implements DieselAcceptor {
        private double liters;

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

- Java's abstract interfaces separate the usage of an object from its implementation.
  - E.g. *GasAcceptor* lies between *Garage* and *GasCar* like an electric isolator between conductors.
  - The abstract interface *GasAcceptor* makes the UDTs *Garage* and *GasCar* independent form each other as a layer of indirection.

Abstract Interfaces – Separation of Usage and Implementation – Part 2

- In a UML class diagram, we notate the class *Garage* being dependent on *GasAcceptor* and *DieselAcceptor*.

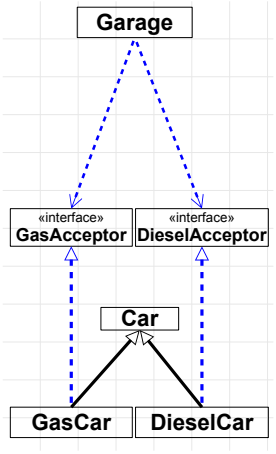
```
// <Garage.java>
public class Garage { // (other members omitted)
        public void refuel(GasAcceptor gasAcceptor, double liters) {
              gasAcceptor.fillGas(liters);
        }
        public void refuel(DieselAcceptor dieselAcceptor, double liters) {
              dieselAcceptor.fillDiesel(liters);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

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

```
// <DieselAcceptor.java>
public abstract interface DieselAcceptor {
        public abstract void fillDiesel(double liters);
}
```

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

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

```
// <DieselCar.java>
public class DieselCar extends Car implements DieselAcceptor {
        private double liters;

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



- *Garage* is only dependent on the abstract interfaces directly, not on the implementing classes *GasCar* or *DieselCar*!
  - There is no line pointing from *Garage* to either concrete class (*GasCar* or *DieselCar*).

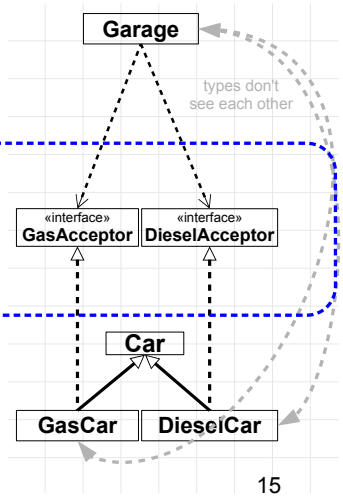
Abstract Interfaces – Separation of Usage and Implementation – Part 3

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

        public void refuel(DieselAcceptor dieselAcceptor, double liters) {
            dieselAcceptor.fillDiesel(liters);
        }
}
```

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

```java
// <DieselAcceptor.java>
public abstract interface DieselAcceptor {
    public abstract void fillDiesel(double liters);
}
```

abstract interfaces:
< reduce type visibility >
< layer of indirection >
< loose coupling >
< type isolation >
< like a contract >
< like a pivot >

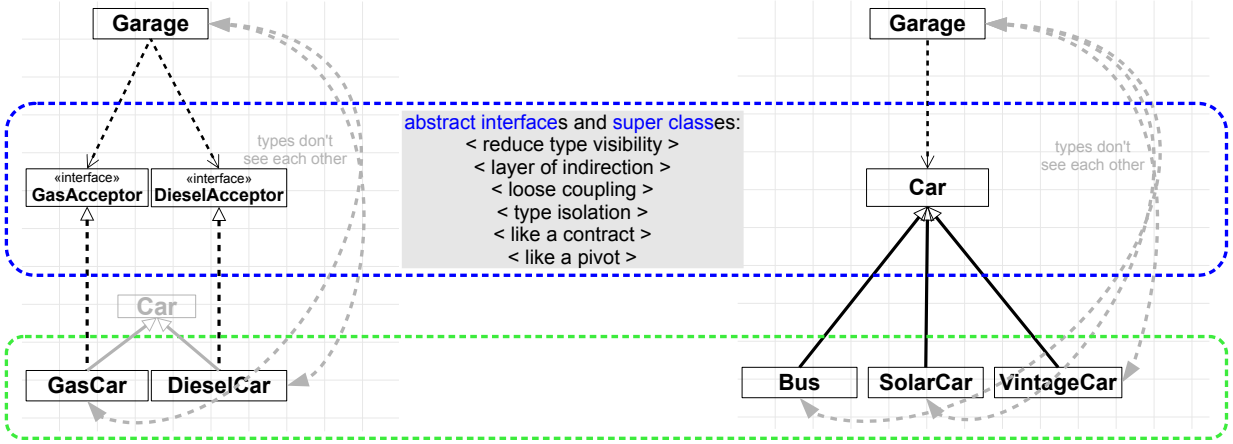



```java
// <GasCar.java>
public class GasCar extends Car
            implements GasAcceptor {
        private double liters;

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

```java
// <DieselCar.java>
public class DieselCar extends Car
            implements DieselAcceptor {
        private double liters;

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

Abstract Interfaces – Separation of Usage and Implementation – Part 4

- abstract interfaces and super classes share many features, which can be used in type hierarchies to add layers of indirection.



  - The layer of indirection (blue) shields the consumer type (*Garage*) from the concrete implementations (green).

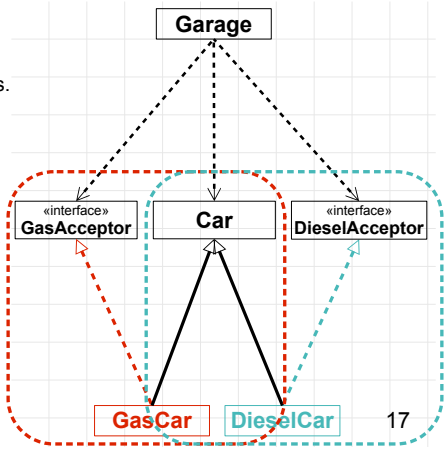
- Layers of indirection is the common idea behind super classes and abstract interfaces. – But what's the difference?

Abstract Interfaces – Separation of Usage and Implementation – Part 5

- So, what kind of problem do abstract interfaces actually solve?
  - The problem with classes is, that we can only extend a single class per layer!
  - But with abstract interfaces, we can implement multiple behaviors per layer!

Fundamental concept of abstract interfaces:
abstract interfaces help decoupling, because they allow to define exactly, how objects are coupled as a set of exactly defined methods.

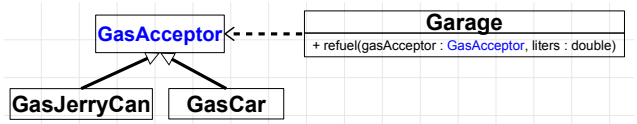
- E.g. *GasCar* can be handled in *Garage* in two ways:
  - (1) Via its "*Car*-nature", it can call *Car*-methods on *GasCar*s.
  - (2) Via its "*GasAcceptor*-nature", it can call *GasAcceptor*-methods on *GasCar*s.
- Because *GasCar* presents multiple abstract interfaces (that of *Car* and *GasAcceptor*) *GasCar* can be used via both types as layer of indirection.
- Because we can implement multiple abstract interfaces on the same layer of a type, we can use multiple indirections on that type!
- To show how useful this is, we'll implement another abstract interface in *GasCar*, which gives *GasCar* an additional behavior.
- We'll implement the JDK abstract interface *Comparable* in *GasCar*!



Abstract Interfaces – Separation of Usage and Implementation – Part 6

- Remember, when we discussed coupling and coherence, esp. generalization-specialization adds tight coupling.
  - This is due to the fact, that generalization-specialization generally adds substitutability as a type-feature.
  - And it can be difficult to exchange the super class *A* of class *B*, if other code relies on the fact that *B*s can substitute *A*s.

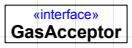
- Assume *GasAcceptor* as class instead of as abstract interface and *Garage.refuel()* having a *GasAcceptor* as 1st parameter:



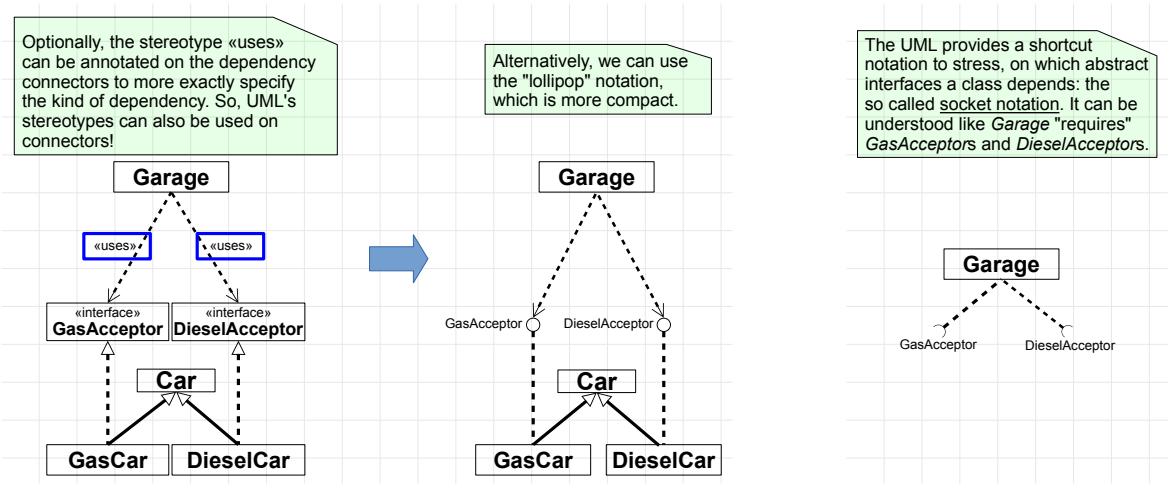

```
// <Garage.java>
public class Garage { // (other members omitted)
        public void refuel(GasAcceptor gasAcceptor, double liters) {
            gasAcceptor.fillGas(liters);
        }
}
```

  - The problem is coupling: we cannot change the super class of *GasJerryCan* and *GasCar* to something different as *GasAcceptor*!
  - If we did, *GasJerryCan*s and *GasCar*s could no longer be passed to *Garage.refuel()*: they would be no *GasAcceptor*s anymore!
  - => We broke substitutability/compatibility of *GasJerryCan* and *GasCar* with *Garage*, because of their tight coupling via class *GasAcceptor*.

- If *GasAcceptor* is an abstract interface, *GasJerryCan* and *GasCar* can decide which class to extend and extra abstract interface to implement.
  - Coupling *GasJerryCan* and *GasCar* to *Garage* via the abstract interface *GasAcceptor* is not tight.
  - => *GasJerryCan* and *GasCar* are not tightly/solely bound to *GasAcceptor* for substitution.
  - abstract interfaces can add multiple substitutability/compatibility to types, which can lessen coupling.



Abstract Interfaces – reduced UML Notations



Implementing Comparable – Part 1

- Let's discuss the practical programming problem of "sorting" to make use of one of Java's predefined abstract interfaces.

- Let's assume, we extended *GasCar*, so that it has a new field *fuelCapacity* to store its fuel capacity in liters:

```
// <GasCar.java>
public class GasCar extends Car { // (other members omitted)
        private final double fuelCapacity;

        public GasCar(double fuelCapacity) {
        this.fuelCapacity = fuelCapacity;
        }
}
```

  - And we have a *GasCar[]*, that should be sorted after the fuel capacity of the contained *GasCars*:

```
GasCar[] gasCars= { new GasCar(23.0), new GasCar(35.6), new GasCar(19.7), new GasCar(33.5) };
```

  - We already know, that Java provides the useful method *java.util.Arrays.sort()* to sort arrays, why not using it?

```
Arrays.sort(cars);
```

  - Because it doesn't work:

```
java.lang.ClassCastException: GasCar cannot be cast to java.lang.Comparable
```

- So, what have we done wrong?

Implementing Comparable – Part 2

- The logical problem is: How should *java.util.Arrays.sort()* perform the sorting?
  - It doesn't know, after which criterion we want to sort!

- Principally, sorting algorithms compare the items to be sorted and shift their positions to come to the desired order.
  - And here we can find what we have missed: We have to tell *java.util.Arrays.sort()* how to compare *GasCars*!

- The *Exception* we get already told us, what *java.util.Arrays.sort()* expects: objects to be sorted must extend *Comparable*:

```
java.lang.ClassCastException: GasCar cannot be cast to java.lang.Comparable
```

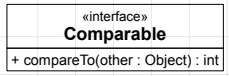
  - But, this won't work! *GasCar* cannot extend *Car* and *Comparable*!

- Fortunately, the JDK designers foresaw this problem and made *Comparable* an abstract interface! This makes sense:
  - (1) A (any) class should at least have the possibility to be sorted, no matter which other classes are already in the hierarchy!
  - (2) Being *Comparable* is more a behavior than a functionality we want to extend or reuse.
    - We'll also see, that *Comparable* is a simple abstract interface having only one method.
  - (3) The name is well chosen: if an object "is *Comparable*" it can be sorted!
    - Mind, how the name reflects an ability of a type and its objects and also a contract or behavior, that is expected by a caller like j*ava.util.Arrays.sort()*.

Implementing Comparable – Part 3

- Without further ado, here, we have *GasCar* implementing *Comparable* to compare *GasCar*s' fuel capacity:

```
// <GasCar.java>
public class GasCar extends Car implements Comparable { // (other members omitted)
        private final double fuelCapacity;

        public GasCar(double fuelCapacity) {
                this.fuelCapacity = fuelCapacity;
        }
        @Override
        public int compareTo(Object otherCar) {
                double fuelCapacityOfOtherCar = ((GasCar) otherCar).fuelCapacity;
                if (this.fuelCapacity < fuelCapacityOfOtherCar) {
                        return -1;
                } else if (this.fuelCapacity > fuelCapacityOfOtherCar) {
                        return 1;
                } else {
                        return 0;
                }
        }
}
```



- As can be seen, *Comparable* is a simple abstract interface. We only have to implement one method: *compareTo()*.
  - The implementation of *compareTo()* looks strange, but the idea behind is relatively simple.
  - *compareTo()* can be implemented simpler, and also more correct, but now we want to dissect how it works principally.
  - However, sorting *GasCar[]* will now work:

```
GasCar[] cars = { new GasCar(23.0), new GasCar(35.6), new GasCar(19.7), new GasCar(33.5) };
Arrays.sort(cars);
// gasCars = { GasCar(19.7), GasCar(23.0), GasCar(33.5), GasCar(35.6) }
```

Comparable.compareTo()'s Parameter

```
@Override
public int compareTo(Object otherCar) {
        double fuelCapacityOfOtherCar = ((GasCar) otherCar).fuelCapacity;
        if (this.fuelCapacity < fuelCapacityOfOtherCar) {
        return -1;
        } else if (this.fuelCapacity > fuelCapacityOfOtherCar) {
                return 1;
        } else {
                return 0;
        }
}
```

```
// Somewhere in the JDK, the interface Comparable is defined:
public abstract interface Comparable { // (declaration simplified)
        public int compareTo(Object other);
}
```

- The only parameter of *compareTo()* is of type *Object*.
  - This is true, because *Comparable.compareTo()* is defined to accept an *Object*-type parameter.
  - If an *Object*-type parameter is used in the method to override, it must also be of type *Object* in overrides, i.e. it must be invariant!
  - Mind, that the same invariance-rule is valid for overriding *Object.equals()*,
- However, that parameter represents the other object against we want to compare this.
- Because the parameter is of type *Object*, we have to make a down cast to the expected type, here *GasCar*.
  - Contrary to *equals()*, we do not make any type checks. i.e. *GasCar.compareTo()* could throw a *ClassCastException*.

Comparable.compareTo()'s Return Value – Part 1

```
@Override
public int compareTo(Object otherCar) {
	double fuelCapacityOfOtherCar = ((GasCar) otherCar).fuelCapacity;
	if (this.fuelCapacity < fuelCapacityOfOtherCar) {
	return -1;
	} else if (this.fuelCapacity > fuelCapacityOfOtherCar) {
		return 1;
	} else {
		return 0;
	}
}
```

- *compareTo()* returns an int, not a boolean! Why?
  - The idea of *compareTo()* is to determine the relative order of objects to each other. – This is not a "yes/no-question"!
  - "Relative order" just defines, if an object is logically less than other, greater than other, or equal to other.

- The relative order is determined by *compareTo()* like this:
  - If this is less than other, the return value must be negative, if this is greater than other the return value must be positive (not 0).
  - If neither this is less than or greater than other, it must return 0!

```
GasCar gasCar1 = new GasCar(35.6), gasCar2 = new GasCar(23.0);
int comparisonResult = gasCar1.compareTo(gasCar2);
System.out.println("gasCar1 is greater than gasCar2: " + (comparisonResult > 0));
// >gasCar1 is greater than gasCar2: true
```


- Another term that describes the idea behind *Comparable.compareTo()* is "three-way-comparison".

Comparable.compareTo()'s Return Value – Part 2

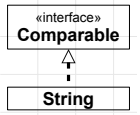
- So, *compareTo()* gives us another way to define equivalence of objects beyond *equals()*.
  - Comparing objects using *compareTo()*, is called relative-order-based-comparison.
  - Comparing objects by their equality, i.e. using *equals()*, is called equality-comparison.
  - If of two objects one is neither less than nor greater than the other the objects are said to be equivalent (Not equal!).

- In a past lecture we already used *compareTo()* on *String*s!

```
String fstName = "Alberta";
String sndName = "Caroline";
int result = fstName.compareTo(sndName));
// result < 0
```

  - Among other interfaces the class *String* implements *Comparable*, thus it provides an *@Override* of *Comparable.compareTo()*:

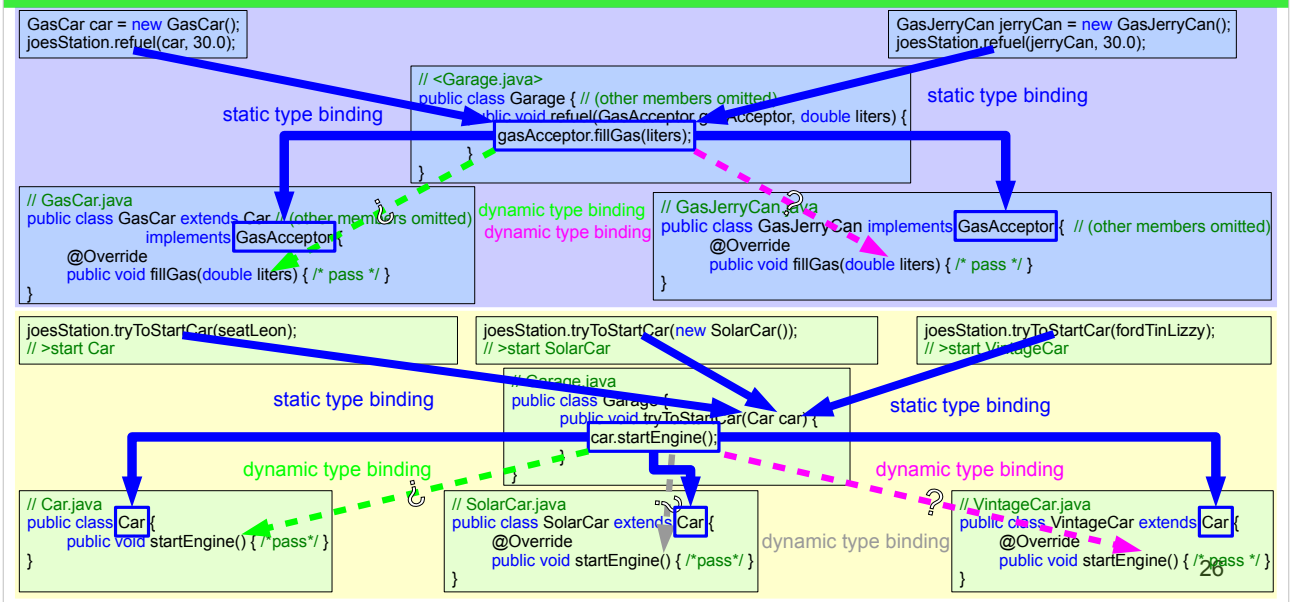
```
// Simplified implementation of String
public final class String implements Comparable {
        @Override
        public int compareTo(Object other) {
                // pass
        }
        // pass
}
```



  - Because *String* implements *Comparable*, we can use *java.lang.util.Arrays.sort()* to sort a *String[]*:
    - *Arrays.sort()* can work with *String*s, because *String* understands the *Comparable* "protocol".

```
String[] names = { "Frida", "Marge", "Carina", "Klaus", "Jennifer" };
Arrays.sort(names);
// names = {"Carina", "Frida", "Jennifer", "Klaus", "Marge"}
```

Abstract Interfaces enable Polymorphism!



Abstract Interfaces – Definitions

- Definition side of abstract interfaces:
  - abstract interfaces are ordinary UDTs in the Java world, they are just slightly more abstract than classes.
  - public interfaces are coded in one java-file per interface. Here we follow the same rules as for other UDTs like classes or enums.
  - The Java compiler produces class-files from java-files with an interface definition (i.e. there exist no "interface-files")!
  - The names of interfaces follow the same rule as for other UDTs: PascalCase. – Don't use prefixes for interface names (like "I")!
  - All method declarations in interfaces are implicitly abstract and public and cannot be declared with other access specifiers.
  - interfaces don't encapsulate details and can not have instance fields, their only job is to expose an "interface to the public".
  - Therefore we can reduce the syntax: the explicit specification of the public (for methods) and abstract keywords can be left away:

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

Good to know
We can also have a (static) *main()*-method in an interface!

- Specialties of interface definitions:
  - (interfaces cannot contain instance-fields!)
  - But interfaces can contain implicitly public final static fields, other access specifiers are not allowed.
    - This static final fields must be initialized with compile time constants!
  - interfaces can contain fully implemented public or private static methods.
  - interfaces can define default methods. – We'll discuss those in a future lecture.
  - interfaces can be static nested (and local with Java 15) and can have any access-specifier as static nested classes.

```
// <MyInterface.java>
public interface MyInterface {
        // constant static field:
        public static final double PI = 3.14;
        // static method:
        public static void doit() {
                System.out.println("static method in interface");
        }
}
```

27

- A good way to understand interfaces is to view them as concepts. Then the implementations of interfaces are just the implementations of the concepts.
- Within an interface, we can define static nested classes, i.e. enums, classes and yet more interfaces.
- interfaces can also be nested, but only as static nested UDTs (they are implicitly nested, so no need for the static keyword), because a this-context of the outer class wouldn't make sense for interfaces.

Interface Implementation and Usage

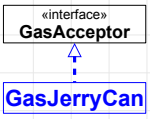
- Implementation side of interfaces (i.e. classes, which implement interfaces):
  - The methods of an interface to be implemented in a class need to be public!
  - A class, which doesn't *@Override* all methods of the interfaces to be implemented has to be declared abstract!
  - Also enums can implement interfaces.
- Usage side of interfaces:
  - Like with abstract classes we cannot create objects from interfaces. interfaces are too abstract to be instantiated.
  - classes can implement one or more interfaces! But classes cannot extend/inherit from more than one class!
- Other names for the concept of Java interfaces:
  - *GasAcceptor* describes only a behavior, therefor, it is a behavioral type.
  - Contract
  - Protocol

- All enums implicitly inherit the class *Enum*, which itself implements *Comparable*. – Therefor all enum values can be compared with the method *Comparable.compareTo()*!

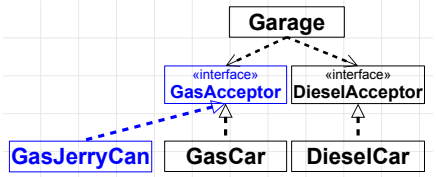
Interfaces make Code Future-proof – Part 1

- An important aspect of interfaces is, that UDTs using interfaces will work w/ all UDTs implementing that interface in future.
  - Assume the new UDT *GasJerryCan*:

```
// <GasJerryCan.java>
public class GasJerryCan implements GasAcceptor {
        private double liters;

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



  - With *GasJerryCan* we have another implementation of *GasAcceptor* besides *GasCar*.



Mind, how the interface *GasAcceptor* allows us to use *GasJerryCan* the same way as *GasCar*: although they're a conceptionally unrelated types, they offer common behavior via *GasAcceptor* to any object, that can deal with *GasAcceptor*s now and in future.

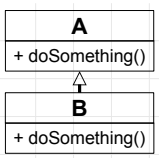
- *Garage* can also fill *GasJerryCan*, because it can fill all UDTs implementing *GasAcceptor*.
  - Once again the other perspective: *GasAcceptor* perfectly shields *Garage* from any concrete implementations.

- => Because *Garage* uses *GasAcceptor* it is future-proof. It is able to handle all implementors of *GasAcceptor* in future.

Interfaces make Code Future-proof – Part 2 – the LSP

- OO Polymorphism with generalization-specialization and interfaces enable the Liskov Substitution Principle (LSP).

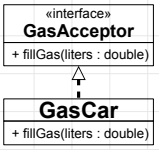
- LSP tells us, that a type can be exchanged by another type, as long as it offers the same interface.




```
A instance = new B();
instance.doSomething(); // Calls B.doSomething()!
```

  - Substitution: We can do this with sub classing: wherever an *A* is accepted the sub class *B* can be set.
  - The LSP can also be understood as inheritance establishing class-relations, which guarantee backward-compatibility implicitly.
  - => Wherever an *A* is accepted any sub class of *A* can be set, now and in future.

- Java additionally offers the LSP via its interface idiom:




```
GasAcceptor gasAcceptor = new GasCar();
gasAcceptor.fillGas(23.5); // Calls GasCar.fillGas()!
```

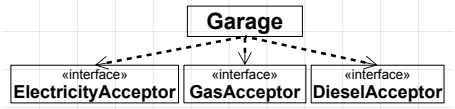
  - Substitution: Wherever a *GasAcceptor* is accepted, an object of the implementing class *GasCar* can be set.
  - So, the LSP can also be understood as interfaces guarantee backward-compatibility of unrelated types implicitly.
  - => Wherever a *GasAcceptor* is accepted any class implementing *GasAcceptor* can be set, now and in future.

Interfaces make Code Future-proof – Part 3 – deferring Implementations

- Code, which uses an API is usually not programmed the same time the API is programmed.
  - The oo paradigm addresses this fact with interfaces.
  - An interface is a promise to API-calling code, that if it implements the interface, it'll be able to work with the API now and in future!
  - In opposite, an interface is also a promise to the API, that API-calling code will adhere to the interface.
  - => The interface could be defined and used, long before the API was finished.
  - => With interfaces we decouple callers and callees in an effective way!

- The term interface can just be replaced by the word protocol or contract or also specification.

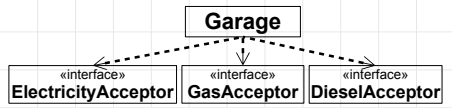
- Assume *Garage*, which knows and fulfills the interfaces *ElectricityAcceptor*, *GasAcceptor* and *DieselAcceptor*.



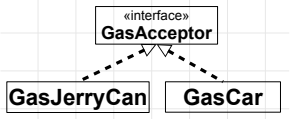
  - Mind, that *Garage* is dependent on those interfaces!

Interfaces make Code Future-proof – Part 4 – a Layer of Indirection

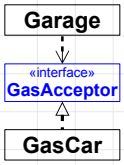
- Another way to understand interfaces is to view them as concepts.
  - *Garage* knows the concepts *ElectricityAcceptor*, *GasAcceptor* and *DieselAcceptor* and can deal with them:



- On the other hand, we have classes which implement concepts: *GasGar* and *GasJerryCan* implement *GasAcceptor*:



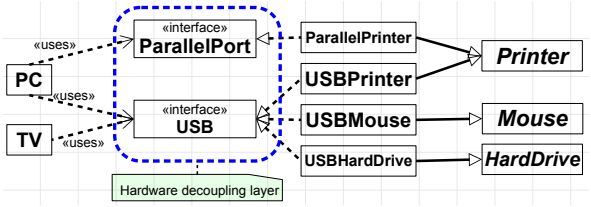
- interfaces add a layer of indirection: *Garage* is only dependent on *GasAcceptor*, *GasCar* is only dependent on *GasAcceptor*.
  - There is no dependency between *Garage* and *GasCar*! *Garage* can deal with any *GasAcceptor*s, also *GasJerryCan*s.



Excursus: We also have decoupling on Hardware Level

- In the beginning of computing, computers with specific hardware and often bundled OS were strict on choosing peripherals.
  - E.g. a C64 disk drive cannot "just" be connected to a IBM PC in 1982.
  - Manufactures wanted their customers only to buy peripherals they produced themselves.

- Today hardware consortia standardize how peripherals connect electrically and on the protocol level.
  - On the hardware side, manufacturers of computers and peripheral devices comply to the electrical specification.
  - On the software side, manufacturers must provide drivers as implementation of the software protocol.



- This yields a lot of benefits, because interfaces allow to decouple development of interface-implementors and -users.
  - The interfaces could be defined before all the thinkable peripherals were known. – E.g. USB-keys came relatively late.
  - Present devices, e.g. *USBHardDrive* can be connected to any USB-savvy "host", not only PC's but also TVs.
  - A *USBMouse* could connect to TV electrically, but not on a functional level, this must be done by (software) drivers.

Implementation and Usage of abstract Types

- As with abstract classes, we can use interfaces in five ways basically.
- (1) As interface implemented by a class
  - (1a) As super interface of another interface

```
public class OtherGasAcceptor implements GasAcceptor {
        // pass
}
```

- (2) On the left side of an assignment

```
GasAcceptor gasAcceptor = new OtherGasAcceptor();
joesStation.refuel(gasAcceptor, 20);
```

- (3) As parameter type

```
public void refuel(GasAcceptor gasAcceptor, double liters) {
        // pass
}
```

- (4) As return type

```
public GasAcceptor nextAtFuelDispenserQueue() {
        // pass
}
```

- (5) For dynamic type checks and casts

```
if (objectToBeTanked instanceof GasAcceptor)
        GasAcceptor gasAcceptor = (GasAcceptor) objectToBeTanked;
}
```

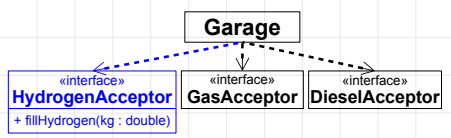
- Conclusions:
  - interfaces are mainly used as static types, esp. because there can be no instances.
  - The "type-character" of an interface, i.e. the contract, is crucial, not the instance.

- There are also cases, in which an interface is to be used as dynamic type in a cast operation.

Interfaces – Modification Scenario: Offering the Fuel-Type Hydrogen in Garages

- This requires two changes in our current abstraction of *Garage*s:
  - adding the new interface *HydrogenAcceptor* and
  - adding the new method *Garage.refuel(HydrogenAcceptor, double)*

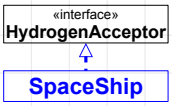
```
// <HydrogenAcceptor.java>
public interface HydrogenAcceptor {
        void fillHydrogen(double kg);
}
```




```
// <Garage.java>
public class Garage { // (other members omitted)
        public void refuel(HydrogenAcceptor hydrogenAcceptor, double kg) {
                hydrogenAcceptor.fillHydrogen(kg);
        }
}
```

- Having that changes in place somewhen in future anybody may implement *HydrogenAcceptor* and pass instances to *Garage*s.
  - Assume, that *SpaceShip*s are replenished with hydrogen, let's go with it!
  - And now we can tank the Space Shuttle at *joesStation*! ;)

```
SpaceShip spaceShuttle = new SpaceShip();
joesStation.refuel(spaceShuttle, 106_261);
```

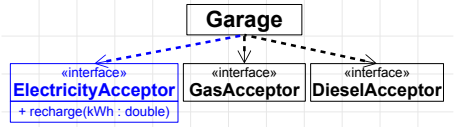


- It's important to understand, that *joesStation* was able to refuel *HydrogenAcceptor*s before the UDT *SpaceShip* existed!

Interfaces – Modification Scenario: Hybrid Electric Car accept two Fuel-Types – Part 1

- This requires two changes in our current abstraction of *Garages*:
  - adding the new interface *ElectricityAcceptor* and
  - adding the new method *Garage.refuel(ElectricityAcceptor, double)*

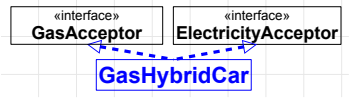
```
// <ElectricityAcceptor.java>
public interface ElectricityAcceptor {
        void recharge(double kWh);
}
```




```
// <Garage.java>
public class Garage { // (other members omitted)
        public void refuel(GasAcceptor gasAcceptor, double liters) {
            gasAcceptor.fillGas(liters);
        }
        public void refuel(ElectricityAcceptor electricityAcceptor, double kWh) {
            electricityAcceptor.recharge(kWh);
        }
}
```

- With that changes in place, we can code *GasHybridCar*, which implements both, *GasAcceptor* and *ElectricityAcceptor*.
  - Now we are exploiting the ability to implement more than one interface in one class.
  - Therefor we just pass a comma separated list of interfaces to be implemented.

```
// <GasHybridCar.java>
public class GasHybridCar extends Car implements GasAcceptor, ElectricityAcceptor {
        private double liters;
        private Battery battery;
        @Override
        public void fillGas(double liters) {
                this.liters = liters;
        }
        @Override
        public void recharge(double kWh) {
                this.battery.recharge(kWh);
        }
}
```

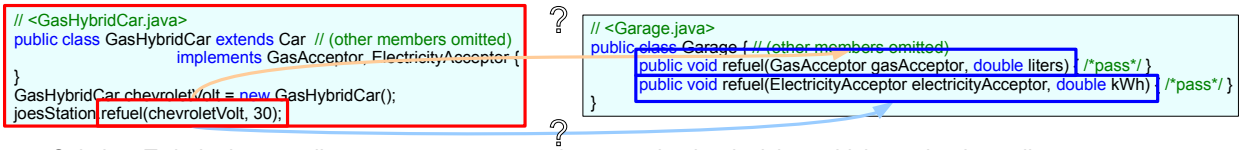


Interfaces – Modification Scenario: Hybrid Electric Car accept two Fuel-Types – Part 2

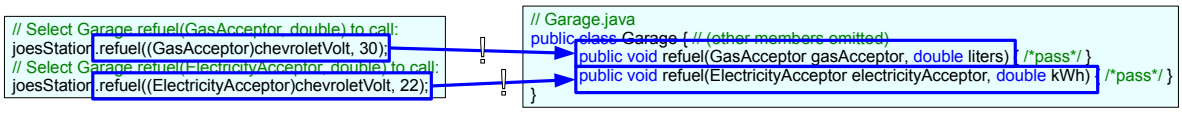
- But it doesn't work, passing an object of type *GasHybridCar* to *Garage.refuel()* produces a compiler error:

```
GasHybridCar chevroletVolt = new GasHybridCar();
joesStation.refuel(chevroletVolt, 30);
// Invalid! Ambiguous method call. Both
// refuel refuel(ElectricityAcceptor, double) in Garage and
// refuel refuel(GasAcceptor, double) in Garage and
```

- The compiler cannot decide which method to call:
  - (1) *GasHybridCar*s like *chevroletVolt* implement *ElectricityAcceptor* and *GasAcceptor*.
  - (2) *Garage.refuel()* has two overloads, one dealing with an *ElectricityAcceptor* another one dealing with a *GasAcceptor*.
  - => Both *Garage.refuel()* overloads are candidates for the compiler, because *chevroletVolt*'s type implements both interfaces!



- Solution: To help the compiler, we as programmers have to take the decision, which overload to call.
  - Here we have to up-cast *chevroletVolt* to a specific (interface) type, e.g. to *GasAcceptor* to select the desired overload.



Interfaces – Interface can inherit from other Interfaces – Part 1

- After the years go by, more and more customers fill their *GasHybridCar*s at *joesGarage*.
  - So it was decided to set for those cars special fuel dispensers in place.
  - From a programing standpoint it is required to add a method to *Garage*, to refill *GasHybridCar*s' tanks and batteries.

```
// <Garage.java>
public class Garage { // (other members omitted)
        public void refuel(GasHybridCar gasHybridCar, double kWh, double liters) {
                gasHybridCar.recharge(kWh);
                gasHybridCar.fillGas(liters);
        }
}
```

- But this solution lets another issue reappear: We can only refill *GasHybridCar*s.
  - Let's assume *GasHybridBus* as new UDT, which should be refueled in *Garage*s.
  - If a *GasHybridBus* wants to refuel, it needs to be used with *Garage.refuel(GasAcceptor)* and *Garage.refuel(ElectricityAcceptor)* respectively and individually.

```
GasHybridBus gasHybridBus = new GasHybridBus();
joesGarage.refuel((GasAcceptor)gasHybridBus, 100);
joesGarage.refuel((ElectricityAcceptor)gasHybridBus, 45.7);
```

  - All other "Hybrid"-UDTs also need to use two calls to refuel their two fuel-types!

```
// <GasHybridBus.java>
public class GasHybridBus extends Bus
        implements GasAcceptor, ElectricAcceptor {
        private double liters;
        private Battery battery;

        @Override
        public void fillGas(double liters) {
                this.liters = liters;
        }
        @Override
        public void recharge(double kWh) {
                this.battery.recharge(kWh);
        }
}
```

- But we can solve this problem by creating another interface!

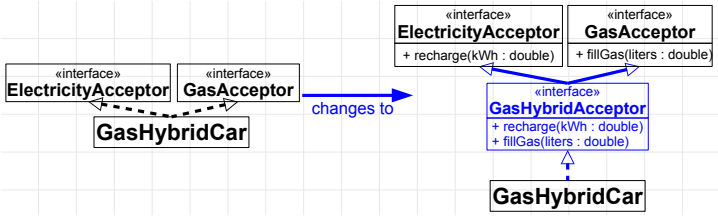
Interfaces – Interface can inherit from other Interfaces – Part 2

- The idea is to create yet another interface, that combines the interfaces *ElectricityAcceptor* and *GasAcceptor*.
  - This new interface combines the behaviors of *ElectricityAcceptor*s and *GasAcceptor*s.

- How do we combine interfaces or combine behavior? – This can be done via interface inheritance!
  - To implement this idea we create a new interface *HybridGasAcceptor*, which inherits *ElectricityAcceptor* and *GasAcceptor*:

```
// <GasHybridAcceptor.java>
public interface GasHybridAcceptor extends ElectricityAcceptor, GasAcceptor {
}
```

  - Exactly! An interface can inherit from another interface! An interface can even inherit from multiple interfaces in Java!
    - An interface can not inherit from a class or enum! And remember, that classes can not inherit multiple classes!
    - The written order of the interfaces in the extends list has no special meaning to Java.
  - Then we implement the new interface *HybridGasAcceptor* in *GasHybridCar*. Our hierarchy changes like this:




```
// <GasHybridCar.java>
public class GasHybridCar extends Car  // (other members omitted)
                          implements GasHybridAcceptor {
    // The overridden methods need not to be changed!
    @Override
    public void fillGas(double liters) { /* pass */ }
    @Override
    public void recharge(double kWh) { /* pass */ }
}
```


- We added *GasHybridCar* as additional layer of indirection to solve our problem! Also *GasHybridBus*ses can be handled now!

Interfaces – Interface can inherit from other Interfaces – Part 3

- Now its time to add support for the new fuel dispensers in *Garage*, therefor we add *Garage.refuel(GasHybridAcceptor)*.

```
// <Garage.java>
public class Garage { // (other members omitted)
        public void refuel(GasHybridAcceptor gasHybridAcceptor, double kWh, double liters) {
            gasHybridAcceptor.recharge(kWh);
            gasHybridAcceptor.fillGas(liters);
        }
}
```

- Combined refueling of *GasHybridAcceptor* is significantly simpler now!
  - Only a single call to *Garage* and no longer multiple calls to *Garage*: refueling the *chevroletVolt* changes like this:

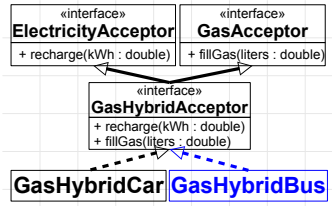
```
// Refuel chevroletVolt with multiple calls:
// Select Garage.refuel(GasAcceptor, double) to call:
joesStation.refuel((GasAcceptor)chevroletVolt, 30);
// Select Garage.refuel(ElectricityAcceptor, double) to call:
joesStation.refuel((ElectricityAcceptor)chevroletVolt, 22);
```

→

```
// Refuel a GasHybridAcceptor directly with a single call:
joesStation.refuel(chevroletVolt, 22, 30);
```

  - If the hypothetical *GasHybridBus* implements *GasHybridAcceptor* as well, it can also benefit from the new fuel dispenser:

```
// Refuel a GasHybridBus directly with a single call:
GasHybridBus gasHybridBus = new GasHybridBus();
joesGarage.refuel(gasHybridBus, 45.7, 100);
```

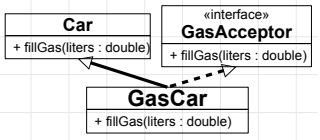


  - Once again an additional layer of indirection (the interface *GasHybridAcceptor*) made our architecture more future-proof! – All objects implementing *GasHybridAcceptor* can be refueled now!

Interfaces – Multiple Interface-Inheritance – Part 1

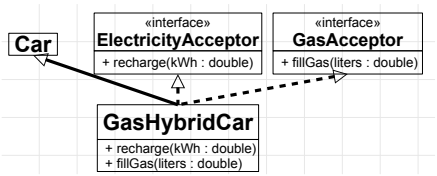
- Basically, interfaces allow a kind of multiple inheritance (MI), resulting in questionable situations. Let's discuss those.

- (1) The class *GasCar* extends another class *Car* with the method *fillGas(*double*)* and implements the interface *GasAcceptor* also with a method *fillGas(*double*)*:
  - This is no problem! Either *GasCar* inherits *Car.fillGas(*double*)* or *@Overrides fillGas(*double*)*.
  - If *fillGas(*double*)* is overridden in *GasCar*, it will also implement *GasAcceptor.fillGas(*double*)*.




```
// <GasCar.java>
public class GasCar extends Car implements GasAcceptor {
	@Override
	public void fillGas(double liters) {
		System.out.println("Implements GasAcceptor.fillGas(double), also overrides Car.fillGas(double)");
	}
}
```

- (2) The class *GasHybridCar* implements the interfaces *ElectricityAcceptor* and *GasAcceptor*, which offer different methods (i.e. disjunct methods) to be implemented:

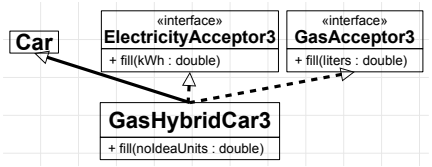


  - In this case *GasHybridCar* has to implement the methods *recharge(*double*)* and *fillGas(*double*)* to implement both interfaces.
    - Then *GasHybridCar* is a concrete class! If *GasHybridCar* wouldn't implement both interfaces, it was an abstract class.

Interfaces – Multiple Interface-Inheritance – Part 2

- (3) Consider the new interfaces *ElectricityAcceptor3* and *GasAcceptor3*, that both require *fill(*double*)* to be implemented:

```
// <GasHybridCar3.java>
public class GasHybridCar3 extends Car implements GasAcceptor3, ElectricityAcceptor3 {
	@Override
	public void fill(double noIdeaUnits) {
		System.out.println("Implements GasAcceptor3.fill(double) and ElectricityAcceptor3.fill(double)");
	}
}
```

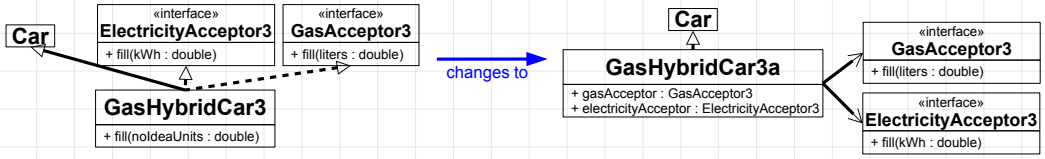


  - Here, the methods *fill(*double*)* to be implemented from *ElectricityAcceptor3* and *GasAcceptor3* have the same name and signature.
  - *GasHybridCar3*'s single *@Override* of *fill(*double*)* overrides *ElectricityAcceptor3.fill(double)* and *GasAcceptor3.fill(double)*!
  - In Java, we have no means to implement a method explicitly for either interface. – It can be "kind of solved" by delegation.

- Let's shortly discuss delegation as a way to have explicit interface implementation.

Interfaces – Multiple Interface-Inheritance – Part 3

- (3a) As mentioned, we can somehow handle situation (3) with delegation for explicit interface implementation:



  - Here we delegate the implementation of the interfaces to the fields *gasAcceptor* and *electricityAcceptor*.
    - Those fields are instances of private nested classes implementing *GasAcceptor3* and *ElectricityAcceptor3* respectively.

```
// <GasHybridCar3a.java>
public class GasHybridCar3a extends Car {
	private double liters;
	private double kWh;
	private class GasAcceptor3Impl implements GasAcceptor3 {
		@Override public void fill(double liters) {
			GasHybridCar3a.this.liters = liters; // Sets liters of outer class' instance.
		}
	};
	private class ElectricityAcceptor3Impl implements ElectricityAcceptor3 {
		@Override public void fill(double kWh) {
			GasHybridCar3a.this.kWh = kWh; // Sets kWh of outer class' instance.
		}
	};
	public final GasAcceptor3 gasAcceptor = new GasAcceptor3Impl();
	public final ElectricityAcceptor3 electricityAcceptor = new ElectricityAcceptor3Impl();
}
```

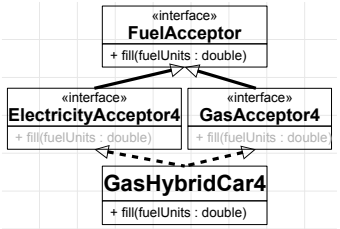
```
GasHybridCar3a chevroletVolt = new GasHybridCar3a();

// Call GasAcceptor3 delegate:
chevroletVolt.gasAcceptor.fill(30);

// Call ElectricityAcceptor3 delegate:
chevroletVolt.electricityAcceptor.fill(22);
```


  - Here the usage of public fields makes a lot of sense, because now, this delegation aspect is no implementation detail!

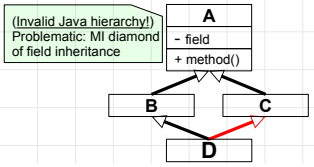
- Mind that the implementing classes *GasAcceptor3Impl* and *ElectricityAcceptor3Impl* are private nested classes in *GasHybridCar3a*, i.e. no other entity than *GasHybridCar3a*, knows the real implementors of those interfaces.

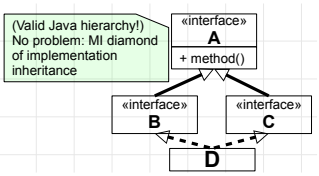
Interfaces – Multiple Interface-Inheritance – Part 4

- (4) Here the interfaces *ElectricityAcceptor4* and *GasAcceptor4* inherit from a common super interface *FuelAcceptor*:
  - This time, *ElectricityAcceptor4* and *GasAcceptor4*, both inherit *FuelAcceptor.fill(*double*)* from the super interface type rather than declaring that method themselves.
  - Then, as in case (3), the methods to be overridden from *ElectricityAcceptor4* and *GasAcceptor4* in *GasHybridCar4* have the same name and signature.
  - We can still neither explicitly implement *FuelAcceptor* nor *ElectricityAcceptor4* nor *GasAcceptor4* in Java.



- What we see here is the "deadly diamond of death" of MI. – What?

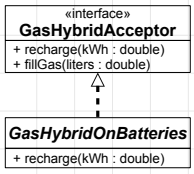




  - MI is per se no problem! But it can become a problem, when the inherited types get a common super type, a diamond hierarchy.
  - In languages allowing MI, this can be problem, because we don't know how fields of *A* are inherited down to *D* via *B* and *C*!
  - But this is no problem in Java: since Java does not allow multiple inheritance of data, but only of functionality all is fine.
    - The question concerning fields is never relevant in Java, because interfaces, of which we implement multiple, cannot have fields!

Partial Implementation of Interfaces

- Sometimes, we can only implement some methods of an interface, and still leave some implementations open.
- E.g., we can assume that many *GasHybridAccceptor*s are going to have a battery, of which recharging is trivial.
  - It means we can at least implement *GasHybridAccceptor* half way, namely *GasHybridAccceptor.recharge()* recharging the battery.
- We can do this in Java, but in this case the class implementing only parts of an interface becomes an abstract class:




```
// <GasHybridAcceptor.java>
public interface GasHybridAcceptor {
        void recharge(double kWh);
        void fillGas(double liters);
}
```

```
// <GasHybridOnBatteries.java>
public abstract class GasHybridOnBatteries implements GasHybridAcceptor {
        private Battery battery;

        @Override
        public void recharge(double kWh) {
                battery.recharge(kWh);
        }
}
```

  - Mind, that the initial wording makes sense: a class, that leaves details away is an abstract class.
- So, a concrete *GasHybridOnBatteries* sub class must only *@Override GasHybridOnBatteries.fillGas()*!

Interface Iterable and for each

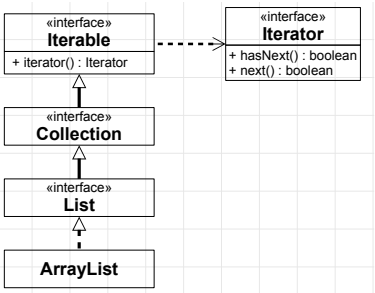
- interfaces are a crucial concept in Java, some Java idioms are based on the implementation of certain interfaces.

- E.g. why does *ArrayList* work with for each?

```
ArrayList<String> words = new ArrayList<String>();
words.add("Helga"); words.add("Olivia"); words.add("Trish");
for (String word : words) {
        System.out.println(word);
}
```

  - Each class, that implements *Iterable* (read: fulfills the *Iterable* protocol) can be used in for each!
  - The class hierarchy of *ArrayList* shows, that it implements *Iterable*.
  - The protocol: When for each is executed it calls *Iterable.iterator()* on the object to be iterated to get an *Iterator*, then *Iterator.hasNext()* and *Iterator.next()* are called to progress through the items of the object:

```
Iterator<String> wordsIterator = words.iterator();
while (wordsIterator.hasNext()) {
        System.out.println(wordsIterator.next());
}
```



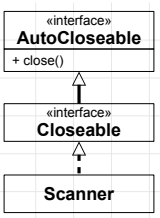
Interface AutoCloseable and try-with-resource

- In past examples, we used *Scanner*-objects to deal with better console input.
  - However, there is one crux: we had to call *Scanner.close()* to close the *Scanner*, when we are done with it:

```
Scanner inputScanner = new Scanner(System.in);
int[] numbers = new int[50];

for (int i = 0; i < numbers.length; ++i) {
        numbers[i] = inputScanner.nextInt();
}
inputScanner.close();
```

- Java provides a better and saver way to deal with objects like *Scanner*, esp. to close resources, when we're done with them.

```
try (Scanner inputScanner = new Scanner(System.in)) {
        int[] numbers = new int[50];

        for (int i = 0; i < numbers.length; ++i) {
                numbers[i] = inputScanner.nextInt();
        }
} // Calls inputScanner.close() automatically at the end of the block.
```



  - Each class, that implements *AutoCloseable* can be used in Java's try-with-resource control structure!
  - The class hierarchy of *Scanner* shows, that it implements *AutoCloseable*.
  - The protocol: On the object declared in the try-clause, the method *AutoCloseable.close()* is called automatically, when control flow leaves the belonging to block!
  - *Scanner.close()* is the implementation of *AutoCloseable.close()*, so *Scanner.close()* is automatically called!
  - => The service for us: try-with-resources does not forget to call *close()* on the *Scanner*!

- The class *Closeable* (added in Java 5) was present in Java before *AutoCloseable* (added in Java 7). *Closeable* resides in the package *java.io* and *AutoCloseable* in the package *java.lang*, which underscores *AutoCloseable*'s idiomatic meaning. *Closeable.close()* declares throws *java.io.IOException*, *AutoCloseable.close()* declares throws *java.lang.Exception*.

Implementing AutoCloseable in our own Classes

- Up to here, we could have used abstract classes instead of interfaces!
  - I.e. *Iterable* and *AutoCloseable* could have been abstract classes! So, where is the point?

- Let's assume, that we want to change *GasCar*, so that it offers a method, which passes the *GasCar* to the scrapyard.
  - => This method ends the lifetime of a *GasCar*! We call the method *GasCar.toScrap()*, it passes the *GasCar* to the *ScrapYard*.

```
// <GasCar.java>
public class GasCar extends Car implements GasAcceptor {
        public void toScrap() { // (members hidden)
                ScrapYard.accept(this);
        }
}
```

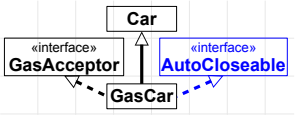
```
GasCar theCar = new GasCar();
joesStation.refuel(theCar, 30.0);
theCar.startEngine();
theCar.drive();
theCar.toScrap();
```

Good to know
AutoClosable.close() must be implemented to be callable for multiple times without harm. We say the implementation of *AutoClosable.close()* must be idempotent.

- A *GasCar* is a resource, which can be discarded, after we finished with it: We can easily make it work with try-with-resource.
  - We have just *GasCar* implement *AutoCloseable* besides the interface *GasAcceptor*:

```
// <GasCar.java>
public class GasCar extends Car implements GasAcceptor, AutoCloseable {
        public void toScrap() {  // (members hidden)
                ScrapYard.accept(this);
        }
        @Override
        public void close() {
                toScrap(); // just delegate!
        }
}
```

```
try (GasCar theCar = new GasCar()) {
        joesStation.refuel(theCar, 30.0);
        theCar.startEngine();
        theCar.drive();
} // Calls theCar.close() automatically
  // at the end of the block.
```



  - Mind, this cannot be done with abstract classes! We cannot inherit multiple classes, the interface is enough to fulfill the contract!

Thank you!