(3) Java Abstractions: Inheritance – Part 1

Nico Ludwig (@ersatzteilchen)

TOC

- (3) Java Abstractions: Inheritance – Part 1
  - Aggregation: Whole – Part Associations
  - Inheritance: Generalization – Specialization Associations
  - The Substitution Principle
  - Problem of Pasta-object-oriented Architecture
  - Polymorphism
  - Method Overriding
  - Static and dynamic Type
  - Array-Covariance

- 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!

Defining the UDT Car with Classes

- Definition of the UDT *Car*:
  - It has two fields:
    - *theEngine*
    - *spareTyre*
    - <u>*Car* has an *Engine* and a SpareTyre.</u>
  - And three methods:
    - *startEngine()*
    - *setSpareTyre()*
    - *getSpareTyre()*

```java
// <Car.java>
public class Car {
	private Engine theEngine;
	private Tyre spareTyre;

	public void startEngine() {
		System.out.println("start Car");
		theEngine.start();
	}
	public void setSpareTyre(Tyre spareTyre) {
		this.spareTyre = spareTyre;
	}
	public Tyre getSpareTyre() {
		return this.spareTyre;
	}
}
```

- The definition of this UDT shouldn't contain any surprises.
  - Therefor we're going to use it for the following discussion.

```java
// Creation and usage of a Car instance:
Car fordFocus = new Car();
fordFocus.setEngine(new Engine());
fordFocus.setSpareTyre(new Tyre());
fordFocus.startEngine();
```

Remember the Concepts of Object Orientation

- **Abstracted types** require two concepts:
  - (1) Abstraction by combining data and methods into a UDT to define a concept.
  - (2) Encapsulation to protect data from unwanted access and modification, it influences cohesion:
    - The *day*-part of a *Date* instance should not be modifiable from "outside".
  - => abstracted types are functionally complete, but do only expose a part of this functionality to the public.

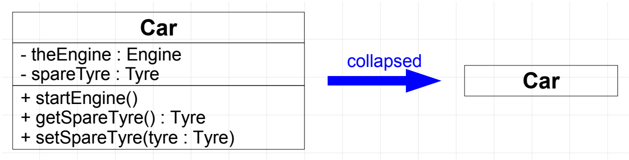
- **Object orientation (oo)** is not only combining behavior and data! – Its aim is simulation of reality in a computer program!
  - To simulate reality, oo requires two more concepts, which influence coupling:
  - (3) The whole – part (aggregation or composition) association:
    - We say "A car object has an engine object.".
  - (4) The specialization – generalization association:
    - We say "three cars drive in front of me", rather than there "drives a van, a bus and a sedan in front of me". We can generalize, as, e.g., a van is a car.

- "Object-orientation" is only the umbrella term for these four concepts.
  - Oo languages provide idioms that allow expressing these concepts.

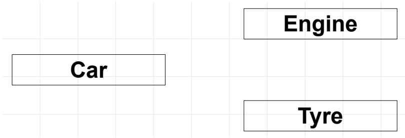
- In this lecture we're going to understand the **association** concepts **(3) whole – part** and **(4) specialization – generalization**.

The UML's Bird's Eye View of Classes

- We have already discussed UML class diagrams.
  - Up to now, we used classifiers like a microscope to show the details of a class. Now we enter the next abstraction level.
  - "abstract" means, that we put away all details of a class. UML allows to collapse a classifier to only the name compartment:



- For our further discussion of oo topics, we can use the collapsed view of class diagrams to get a high level view of *Car*:



  - Now, we can concentrate on the connections between the class *Car*, *Engine* and *Tyre*, and ignore their details.
  - Mind, that we haven't defined *Engine* and *Tyre*! – We have only assumed their existence as classifiers. – This is the power of UML!

- But, what kind of connections do we have between *Car*, *Engine* and *Tyre*?

We have already seen Whole-Part Associations!

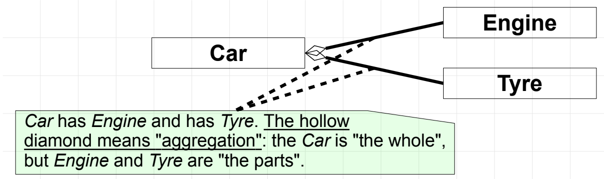
- Basically, we have already discussed whole-part associations! – We called the resulting UDTs record types!
- A very neutral, or, well, abstract view of the UDT *Date*: a type, which has a *day*, a *month* and a *year*.

```java
// <Date.java>
public class Date { // (members hidden)
	public int day;
	public int month;
	public int year;
}
```

  - (We leave the fields *day*, *month* and *year* with public access for now.)
- Technically, Java allows to implement whole-part associations with fields.
- In formal words: Java's idiom to implement whole-part associations are classes ("wholes"), which have fields ("parts").
- Now, we'll discuss the UDT *Car* we have introduced just recently.

UML Notation of Car with Whole-Part Associations

- UML class diagrams can also be used to design whole-part associations (i.e. aggregated types).
  - This class diagram underscores *Car*'s dependencies with connectors and classifiers:



  - So, this is the connection between *Car*, *Engine* and *Tyre*: *Car* aggregates *Engine* and aggregates *Tyre*.

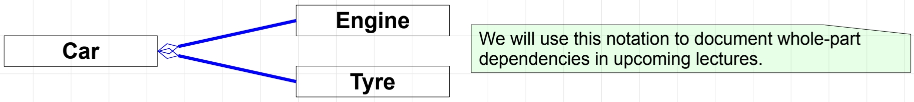
- A key point of UML is, that the model can be understood by non-programmers (e.g. customers) generally.
  - Another key point derived from the first key point: the UML is independent from a specific (oo) programming language.
  - But this also means, that some of the terms in UML are different from the terms we know from Java or the reality.

- What we see here and in upcoming lectures are more class diagrams with multiple boxes.
  - Lines connecting classifiers ("connectors"), so called associations, show the dependencies between classes.
  - This is time to remember the concept of coupling!
    - We don't know how Engine and Tyre are designed, but we know that Car and Engine and Tyre are coupled.

- The key point, that non-programmers should understand what a UML diagram expresses, is far away from reality in practice. When the content of a UML diagram shows a lot of details, non-programmers must understand more details of the UML syntax and "grammar", which is not what non-programers, esp. customers, do not want to be bothered with. => Sometimes, UML is too academic.

UML: Whole-Part Associations vs Aggregation vs Composition

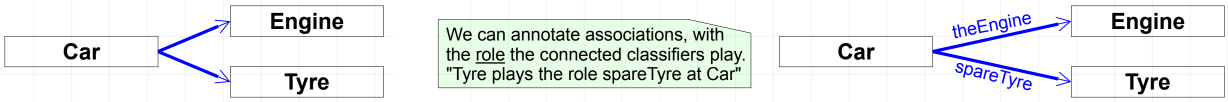
- The notation of an aggregation (connector with hollow diamond) underscores a common whole-part association in UML.



- UML allows expressing a tighter whole-part association, when we notate a composition with a connector with a filled diamond:

  - The association *House-Room* is different from the one between *Car-Engine*, because a *Room* cannot exist without a *House*.
    - => This legitimates usage of a composition instead of only an aggregation.
  - The aggregation between *House* and *Room* does also carry a multiplicity: "one *House* can have one or more *Rooms*".

- If we don't know exactly, what association classifiers have, we can use a "general" association (simply an "association"):

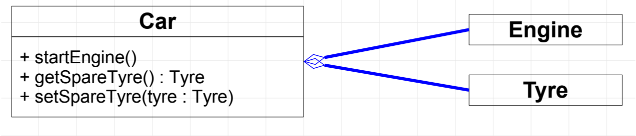


  - Associations are notated as simple lines connecting classifiers. We can add arrow tips to show the direction of an association.
    - "Direction" means: "*Car* knows the classifiers *Engine* and *Tyre*, but not (necessarily) vice versa".
    - The direction of an association is called navigatibility in the UML. It defines how types are coupled, *Car* is coupled to *Tyre*, not vice versa!

- Multiplicity is sometimes also called cardinality (esp. by database aficionados).
- Multiplicity always expressed a "potential" multiplicity, i.e. 1:2 could also mean 1:1 in a real case, but nothing above 1:2 in numbers.

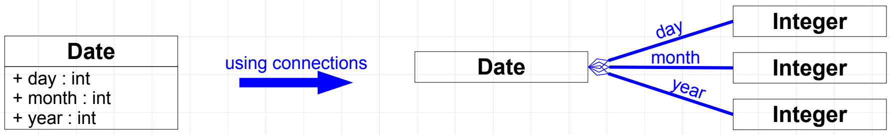
UML Notation of Car with Whole-Part Associations

- An interesting aspect: we can combine "connector view" and "compartment view", and we can collapse any compartment:



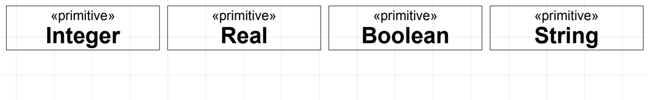
- The UML is so abstract, that we can even abstract primitive types as classifiers instead of using compartments:

```
// <Date.java>
public class Date {  // (members hidden)
        public int day;
        public int month;
        public int year;
}
```



  - The UML predefines several classifiers to represent primitive types:

A stereotype like «primitive» just specifies the meaning of a UML element more precisely. Stereotype names are written in guillemets above the classifier name.



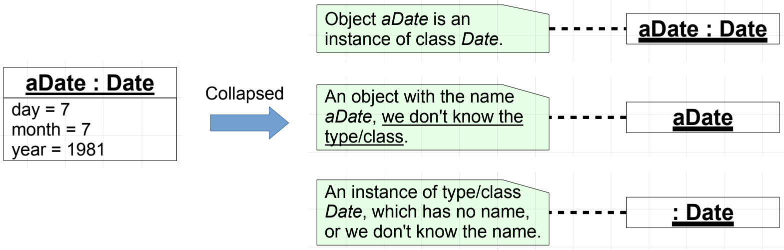
  - The UML adds the stereotype "«primitive»" to the classifier compartment.

Excursus – Objects in UML Class Diagrams (collapsed View)

- Let's once again discuss an object of type *Date* (*aDate*):

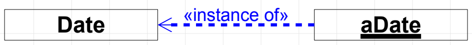
```
Date birthday = new Date();
birthday.day = 7;
birthday.month = 7;
birthday.year = 1981;
```

- We've already discussed, how we represent objects, i.e. instances of classifiers with underscored classifier names.



  - As can be seen, we can collapse the field compartment, which hides the fields' values as well.

- Additionally, the UML allows representing instances as a connector between classifier and object.
  - This connection is just annotated with the so called stereotype: «instance of»:



Looking back… – Part 1

- Oo as a paradigm sits on the shoulders of other paradigms which were based on core believes of their time.
  - Core believe of imperative and structured programming: only use structured programming features → maths in algorithms
  - (Core believe of functional programming: only use functions and avoid side effects → maths in formulas)
  - Core believe of procedural programming: use procedures and records and DRY → building things like an engineer
  - Core believe of oo programming: program to simulate the reality → think about programs as real things
    - Oo was a design pattern in procedural programing, when people used records, belonging-to procedures/functions and handles.
  - Over the time, core believes changed: the degree of abstraction from the technical execution of software is constantly growing.

- Learning oo programming is not a big deal, but learning oo-design is a constant activity.
  - Technically, the idea to implement own UDTs is the manifestation of oo in Java, one just need to understand "enough Java".
  - From a design-perspective, ideally, oo-programmers must understand the domain/problem/environment and define its abstraction.
  - After the "abstraction is clear", "it just needs to be put into UDTs/ Java classes".
  - Oo starts in our minds, oo-languages only support putting these abstractions into effect … more or less.

- For oo-beginners the separation of design time and run time is also a challenge, this separation doesn't exist in reality.
  - At design time we have to deal with encapsulation, concepts, declaration and abstracted types (i.e. classes).
  - At run time we have basically only objects.

Looking back… – Part 2

- For all oo-programmers answering questions about the right abstraction is challenging.
  - Maybe oo-beginners, esp. when they used other paradigms, query the oo-paradigm as such, for experts there's no alternative anymore.

- Where does abstraction start? Which concepts of the reality are concrete enough to justify UDTs?
  - An "idea" is also an object, it isn't as concrete as a car but logically it's concrete: it has attributes (fields) and an interface (methods).
  - On the next lectures, we'll see classes that have no counterpart in reality, but we need them to support our design.
  - When we design the class *Car*, this abstraction is based on our perception of *Car*, what is enough for the problem in question.
  - The class *Car* is not just a module! An object is not just a collection of data and functions!
  - When and how to implement and use UDTs is matter of experience, patience, creativity and luck.

- In order to understand how mighty oo can be, we have to discuss more complex topics: generalization – specialization.

Generalization – Specialization

- Generalization – specialization is a very different association and more abstract than the whole – part association.
  - Let's start by thinking about how we can simulate and abstract the reality with generalization – specialization.

- Let's reconsider the *Car* example. – How can a *Car* be involved in generalization-specialization associations?
  - Now we're going to introduce *Bus*. *Bus* is a UDT, which represents the concept of a, yes, of a bus.

```
// <Bus.java>
public class Bus {  // UDTs Engine and Tyre elided.
	private static final int COUNT_SEATBENCHES = 42;
	private int nOccupiedSeatBenches;
	private Engine theEngine;
	private Tyre spareTyre;

	public void startEngine() {
		System.out.println("start Bus");
		theEngine.start();
	}
	public void setSpareTyre(Tyre spareTyre) {
		this.spareTyre = spareTyre;
	}
	public Tyre getSpareTyre() {
		return this.spareTyre;
	}
	public boolean newPassengerCanEnter() {
		return COUNT_SEATBENCHES <= nOccupiedSeatBenches;
	}
}
```

Examining another Kind of Association between Types – Part 1

- When we look closer at the definition of *Bus*, we'll notice, that there is an intersection between the UDTs *Car* and *Bus*:

```java
// <Car.java>
public class Car { // UDTs Engine and Tyre elided.
	private Engine theEngine;
	private Tyre spareTyre;

	public void startEngine() {
		System.out.println("start Car");
		theEngine.start();
	}
	public void setSpareTyre(Tyre spareTyre) {
		this.spareTyre = spareTyre;
	}
	public Tyre getSpareTyre() {
		return this.spareTyre;
	}
}
```

```java
// <Bus.java>
public class Bus {  // UDTs Engine and Tyre elided.
	private static final int COUNT_SEATBENCHES = 42;
	private int nOccupiedSeatBenches;
	private Engine theEngine;
	private Tyre spareTyre;

	public void startEngine() {
		System.out.println("start Bus");
		theEngine.start();
	}
	public void setSpareTyre(Tyre spareTyre) {
		this.spareTyre = spareTyre;
	}
	public Tyre getSpareTyre() {
		return this.spareTyre;
	}
	public int getOccupiedSeatBenches() {
		return nOccupiedSeatBenches;
	}
	public boolean newPassengerCanEnter() {
		return COUNT_SEATBENCHES <= nOccupiedSeatBenches;
	}
}
```

  - It looks like there is a *Car* "in" *Bus*? – What? But it can't be a whole-part association, it doesn't reflect the reality!
  - Or, huh, do they have something different in common?
  - Come on, we should avoid having duplicated code, mind the DRY principle! – What can we do here?

Examining another Kind of Association between Types – Part 2

```java
// <Car.java>
public class Car { // UDTs Engine and Tyre elided.
	private Engine theEngine;
	private Tyre spareTyre;

	public void startEngine() {
		System.out.println("start Car");
		theEngine.start();
	}
	public void setSpareTyre(Tyre spareTyre) {
		this.spareTyre = spareTyre;
	}
	public Tyre getSpareTyre() {
		return this.spareTyre;
	}
}
```

```java
// <Bus.java>
public class Bus {  // UDTs Engine and Tyre elided.
	private static final int COUNT_SEATBENCHES = 42;
	private int nOccupiedSeatBenches;
	private Engine theEngine;
	private Tyre spareTyre;

	public void startEngine() {
		System.out.println("start Bus");
		theEngine.start();
	}
	public void setSpareTyre(Tyre spareTyre) {
		this.spareTyre = spareTyre;
	}
	public Tyre getSpareTyre() {
		return this.spareTyre;
	}
	public int getOccupiedSeatBenches() {
		return nOccupiedSeatBenches;
	}
	public boolean newPassengerCanEnter() {
		return COUNT_SEATBENCHES <= nOccupiedSeatBenches;
	}
}
```

- What we can extract from this similarity between *Car* and *Bus*: A *Bus* is a kind of *Car*.
- In oo terms this often shortened to a *Bus* is a *Car*, or a *Bus* is a special *Car*. => This matches the reality ... right!
- Now we have a generalization – specialization association between *Car* and *Bus*: a *Bus* is a special *Car*.
  - So a *Bus* doesn't contain a *Car* and a *Car* is not a part of a *Bus*, instead a *Bus* is a *Car*!

Expressing Generalization – Specialization in Java – initial Example

- We modify the class definition of the derived type by using the keyword extends and naming the super type.

```java
// <Car.java>
public class Car { // UDTs Engine and Tyre elided.
	private Engine theEngine;
	private Tyre spareTyre;

	public void startEngine() {
		System.out.println("start Car");
		theEngine.start();
	}
	public void setSpareTyre(Tyre spareTyre) {
		this.spareTyre = spareTyre;
	}
	public Tyre getSpareTyre() {
		return this.spareTyre;
	}
}
```

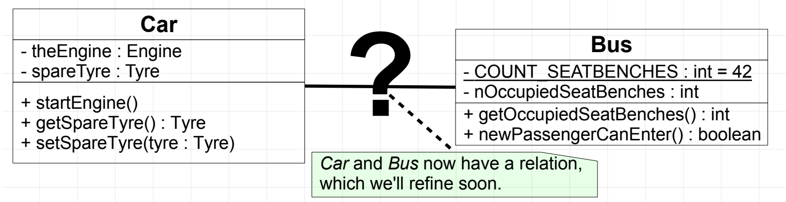
```java
// <Bus.java>
// Bus inherits from Car:
public class Bus extends Car {
	private static final int COUNT_SEATBENCHES = 42;
	private int nOccupiedSeatBenches;

	public int getOccupiedSeatBenches() {
		return nOccupiedSeatBenches;
	}
	public boolean newPassengerCanEnter() {
		return COUNT_SEATBENCHES <= nOccupiedSeatBenches;
	}
}
```

- In Bus.java the extends keyword expresses, that the UDT *Bus*, is inherited from *Car*.
- In other words this inheritance expresses following specialization: a *Bus* is a *Car*.
- The word "extends" fits: *Bus* extends *Car* with *COUNT_SEATBENCHES*, *nOccupiedSeatBenches* and *newPassengerCanEnter()*.
- *Bus* extends *Car* rather than it copies code from *Car*! They have a dependency, that cannot expressed as a whole-part relationship!

- In formal words: Java's idiom for generalization-specialization associations is inheritance using the extends keyword.

Expressing Generalization – Specialization in Java – inherited Methods – Part 1



- So let's instantiate and use a *Bus*:

```
Bus bus = new Bus();
```

  - Of course we can call the methods, which we defined in *Bus*. E.g. *newPassengerCanEnter()*:

```
boolean hasVacantSeats = bus.newPassengerCanEnter(); // Accessing a method of Bus.
```

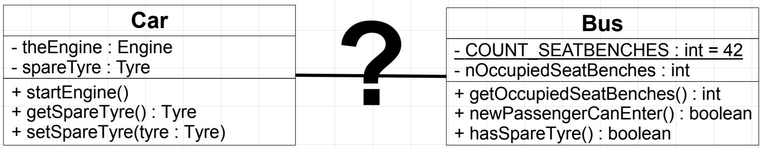
  - The new aspect is, that we can also call *Car* methods, like *getSpareTyre()* on this *Bus* object!
    - Because a *Bus* is a *Car*, in Java terms *Bus* extends *Car*, we can call *Car* methods on a *Bus* object:

```
Tyre spareType = bus.getSpareTyre(); // Accessing a method of Bus' super type Car.
```

    - A sub class inherits behavior from its super class (and intended behavior from the interfaces it implements, which we'll discuss soon).

- What we see here, is that *Bus* inherits the public methods from *Car*!
  - *Car*'s public methods can be called on a *Bus* instance.
  - The ctors of the super type are not inherited to the subtype!
    - This statement is technically true, but dctors are handled in a special way. – We'll discuss this is a future lecture.

Expressing Generalization – Specialization in Java – inherited Methods – Part 2



- An inherited class like *Bus* has full access to all public members of the super class.
  - It means, we can write a method *Bus.hasSpareTyre() calling Car.getSpareTyre()* like so:

```
// <Bus.java>
public class Bus extends Car {  // (members missing)
	public boolean hasSpareTyre() {
		return getSpareTyre() != null; // accesses Car.getSpareType()
	}
}
```

  - Then we can call the new method:

```
Bus bus = new Bus();
boolean hasSpareTypre = bus.hasSpareTyre();
```

- Bottom line: <u>we can call inherited methods within a sub class</u>!

Expressing Generalization – Specialization in Java – inherited Fields – Part 1

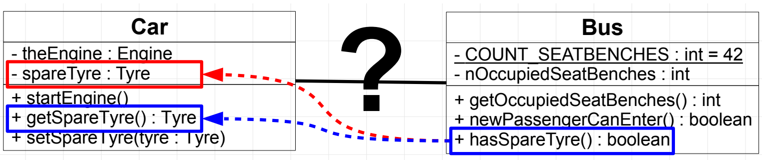
- *Bus* inherits *Car*, which also means, that *Bus* inherits substantial data from *Car*, esp. its fields.
  - But we can not access private fields of a super class! E.g. this implementation of *Bus.hasSpareTyre()* won't compile:

Notice:
All fields and methods of the super class are implicitly added, but possibly not directly accessible, because they were defined with private access.

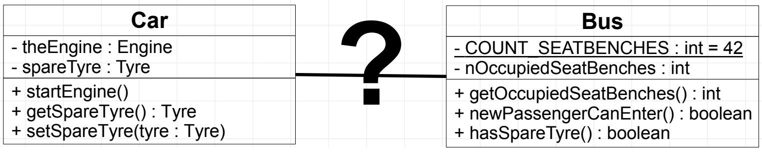
```
// <Bus.java>
public class Bus extends Car { // (members missing) UDT Tyre elided.
	public boolean hasSpareTyre() {
		return spareTyre != null;
	}
}
```

- The *spareTyre* field is private in the super class and unaccessible from any sub classes.
  - Instead we have to call the public method *Car.getSpareTyre()* in *Bus* to implement *Bus.hasSpareTyre()*:

```
// <Bus.java>
public class Bus extends Car { // (members missing) UDT Tyre elided.
	public boolean hasSpareTyre() {
		return getSpareTyre() != null; // Fine!
	}
}
```



Expressing Generalization – Specialization in Java – inherited Fields – Part 2



- In sub classes like *Bus*, private fields of super classes, e.g. *Car.spareTyre*, cannot be directly accessed!
  - => private fields of super classes are a substantial, but not directly accessible part of sub classes.
  - => Read: privately encapsulated data is also encapsulated from sub class access!

- In sub classes like *Bus*, we can call public methods inherited from super classes, e.g. *Car.getSpareTyre()*!
  - This allows to indirectly access private fields via public methods of the super class.
  - This aspect lives the Uniform Access Principle!

Expressing Generalization – Specialization in Java – inherited static Fields – Part 1

- Next, let's assume, we also extend *Bus* creating a new class *LargeBus*, which has basically more seat benches:

```java
// <Bus.java>
public class Bus extends Car {
	public static final int COUNT_SEATBENCHES = 42;
	private int nOccupiedSeatBenches;

	public int getOccupiedSeatBenches() {
		return nOccupiedSeatBenches;
	}
	public boolean newPassengerCanEnter() {
		return COUNT_SEATBENCHES <= nOccupiedSeatBenches;
	}
}
```

```java
// <LargeBus.java>
public class LargeBus extends Bus {
	public void printFillState() {
		System.out.printf("%d seat benches of %d are occupied.%n"
			, getOccupiedSeatBenches()
			, COUNT_SEATBENCHES);
	}
}
```

  - Technically, there is no problem: an inheriting class also inherits the static fields (incl. final fields) and methods from the super class.
  - => We can access *Bus.COUNT_SEATBENCHES* from *LargeBus.printFillState()*, because it is an inherited public (static) field.

- We can also reach *Bus.COUNT_SEATBENCHES* from the class name *LargeBus*:

```java
System.out.println(LargeBus.countSeatBenches); // OK!
// >42
```

Expressing Generalization – Specialization in Java – inherited static Fields – Part 2

- However, to set *COUNT_SEATBENCHES* to a different value for *LargeBus*, we have to
  - make *COUNT_SEATBENCHES* a non-final static field *countSeatBenches* in *Bus*
  - and set *countSeatBenches* to a different value in *LargeBus*' dctor:

```
// <Bus.java>
public class Bus extends Car {
	public static int countSeatBenches = 42;
}
```

```
// <LargeBus.java>
public class LargeBus extends Bus {
	public LargeBus() {
		countSeatBenches = 54;
	}
	public void printFillState() {
		System.out.printf("%d seat benches of %d are occupied.%n"
			, getOccupiedSeatBenches()
			, countSeatBenches);
	}
}
```

- *But* this solution does not work as intended:

```
System.out.println(Bus.countSeatBenches);
// >42
LargeBus largeBus = new LargeBus();
System.out.println(LargeBus.countSeatBenches);
// >54 Clear, as intended!
System.out.println(Bus.countSeatBenches);
// >54 Oups!
```

  - The problem is that the change of *Bus*' static field we did in *LargeBus* actually appears like a change of a global variable.
  - It doesn't matter from where *countSeatBenches* was modified, it has an effect on every location using it.
  - => static non-private fields should be avoided!

Expressing Generalization – Specialization in Java – inherited static Fields – Part 3

- An alternative would be just creating an additional final static field in *LargeBus* that carries the other value:

```
// <Bus.java>
public class Bus extends Car {
	public static final int COUNT_SEATBENCHES = 42;
}
```

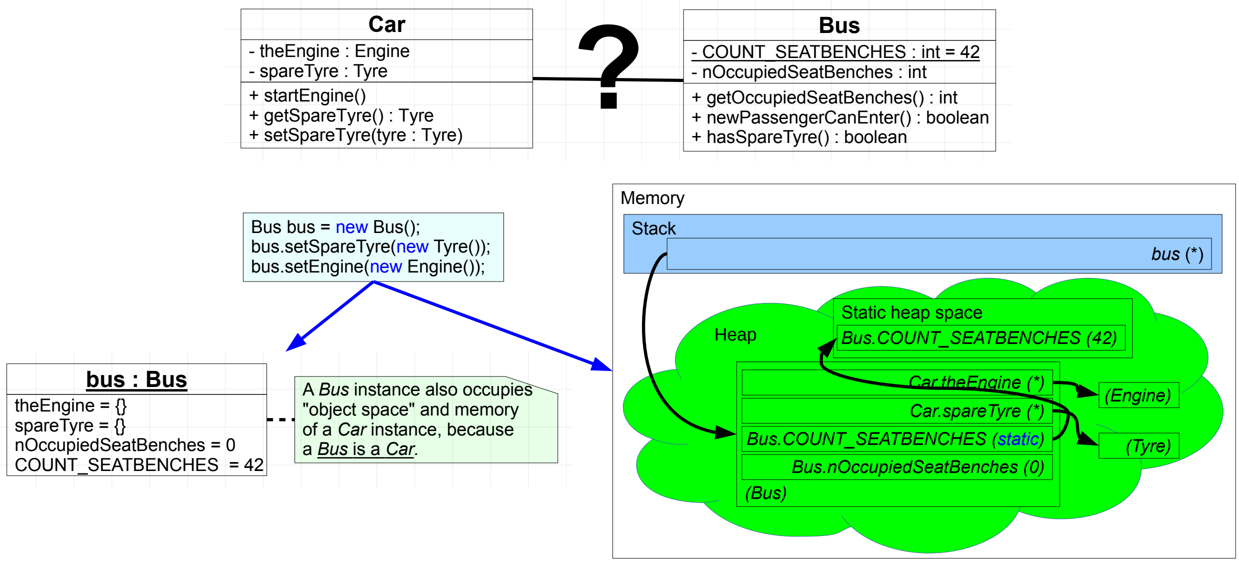
```
// <LargeBus.java>
public class LargeBus extends Bus {
	public static final int COUNT_SEATBENCHES = 54;
	public void printFillState() {
		System.out.printf("%d seat benches of %d are occupied.%n"
			, getOccupiedSeatBenches()
			, COUNT_SEATBENCHES);
	}
}
```

```
System.out.println(Bus.COUNT_SEATBENCHES);
// >42
LargeBus largeBus = new LargeBus();
System.out.println(LargeBus.COUNT_SEATBENCHES);
// >54 Clear, as intended!
System.out.println(Bus.COUNT_SEATBENCHES);
// >42 Ok!
```

- As can be seen, the name of the field in *LargeBus* can be the same as the one in *Bus*, this leads to variable hiding.
  - In *LargeBus* we can access the hidden variable by prefixing it with the name of the super class:

```
// <LargeBus.java>
public class LargeBus extends Bus {
	public static final int COUNT_SEATBENCHES = Bus.COUNT_SEATBENCHES + 8;
}
```

Expressing Generalization – Specialization in Java – in Memory



Expressing Generalization – Specialization in Java – Theory

- We just saw, how inheritance is expressed in Java, but we also have to discuss some theory...

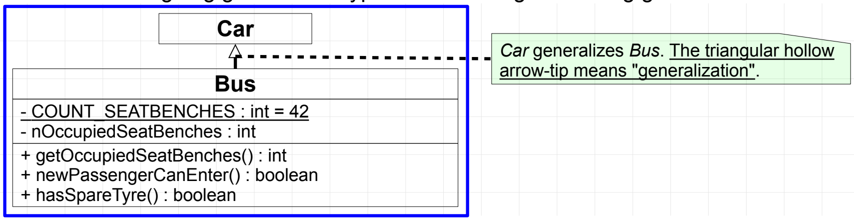
- Existing UDTs can be used as base types for new types.
  - Base types are usually called super types in Java.
  - More special types, like *Bus*, inherit from more general types, like *Car*. This is called inheritance.
    - Instead of "inherits from" we'll occasionally use the phrases "derives from", "subclasses" or "extends".
  - A super type's data ((private) fields) is inherited to an derived type. *Bus* inherits *spareTyre* and *theEngine* from *Car*.
  - A super type's behavior (methods) is inherited to a derived type. *Bus* inherits *startEngine()*, *getSpareTyre()*/*setSpareTyre()* from *Car*.
    - A new type inherits the public interface from its super types transitively. – We'll discuss this later!
    - The inheriting type can access the public interface of its super type but not its private members.
  - The new type can add more members to the inherited ones, so the keyword "extends" was a wise choice!

- In Java, a new type can only inherit directly from one type! -> This is called single inheritance.
  - Java does, however, allow a type to implement multiple interfaces. This concept enables features similar to multiple inheritance.

- Generalization-specialization also defines yet another way of coupling, different from whole-part.

UML Notation of the Car-Bus Generalization Relationship

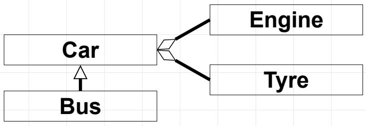
- The UML does also allow designing generalized types in class diagrams using generalization-relationships:



  - UML calls this a generalization, but when we read the relationship upside down, we can also tell, that *Bus* specializes *Car*.
  - In oo/Java programming terms we interpret from this diagram, that *Bus* inherits/extends *Car*.

- In oo, aggregation- and generalization-relationships create a type hierarchy and amplify coupling!
  - This can be nicely shown in UML class diagrams:

Good to know:
In UML generalization relationships are often written having the more general classifier box above the more special one, but this not required.

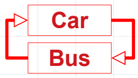


  - We're going to extend this hierarchy during the following discussion.

Good to know:
Java doesn't allow cyclic inheritance dependencies.

```
// <Car.java>
public class Car extends Bus {
	// pass
}
```

```
// <Bus.java>
public class Bus extends Car {
	// pass
}
```



Inheritance defines Substitutability – Part 1

- We discussed that, e.g., a *Bus* is a *Car* and we programmed this association with Java's inheritance.
- Another way to understand this association: inheritance defines substitutability!
  - Wherever the UDT *Car* is awaited, any UDT inherited from *Car* can be used, e.g. *Bus*. – Because a *Bus* is a *Car*!
  - Substitutability makes sense: as *Bus*es just extend *Car*s, all features (esp. methods) a *Car* has must be contained in a *Bus*!
- Let's understand this better with a programmatic example of substitution.

```
Bus bus = new Bus();
```

  - Wherever a *Car* is awaited an object of type *Bus* can be used! A *Bus* is a *Car*.

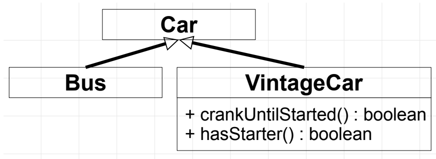
```
Car anotherCar1 = bus; // Aha! We can assign a Bus to a Car object. A Bus is a substitute for a Car.
```

  - *anotherCar1* accepts objects of type *Car* to be assigned and *Bus*-objects are acceptable as well, because *Bus*es are *Car*s!
  - A *Bus* is *Car*, but "can do more".

Inheritance defines Substitutability – Part 2

- To further improve our understanding of substitutability we'll introduce another subtype of *Car*: *VintageCar*!

```
// <VintageCar.java>
public class VintageCar extends Car { // (members hidden)
	public boolean crankUntilStarted() {
		// pass
	}
	public boolean hasStarter() {
		// pass
	}
}
```



- A *VintageCar* object can also be assigned to a *Car*, because a *VintageCar* is a *Car*!

```
VintageCar vintageCar = new VintageCar();
Car anotherCar2 = vintageCar;	// We can assign a VintageCar to a Car object. A VintageCar is a
				// substitute for a Car.
```

- But a *VintageCar* is no *Bus*! So this assignment will not work:

```
Bus anotherBus = vintageCar;	// Invalid at compile time! A VintageCar is no Bus. VintageCar and Bus are
				// siblings in the inheritance hierarchy, they don't "know" each other!
```

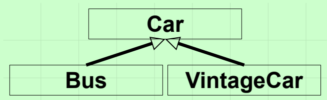
  - The UML diagram clearly shows, that *VintageCar* and *Bus* are siblings and they have no association (line) with each other.
  - Siblings in type hierarchies have common ancestors, but do not know each other!

- Because generation-specialization also adds substitutability as a type-feature, it is a firmer coupling than whole-part.
  - It can be very difficult to change *Bus*' super class from *Car* to *Van*, if other code relies on the fact that *Bus*es can substitute *Car*s.

A mathematical View: Inheritance also defines Subsets

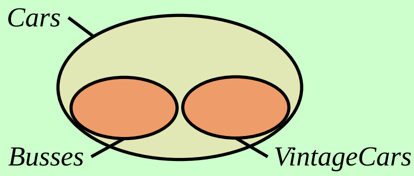
- From a mathematical perspective, inheritance just defines subsets: all *Bus*ses are *Car*s, thus *Bus*ses is a subset of *Car*s.

OO/UML/Java Perspective




```
Car.class.isAssignableFrom(Bus.class) && Car.class.isAssignableFrom(VintageCar.class)
```

Mathematical/Venn Diagram Perspective



$Busses \subset Cars \wedge VintageCars \subset Cars$

- The mathematical subset-relation between *Car* and *Bus* also underscores the substitutability between *Car* and *Bus*:

```
Car aCar = new Bus();                // We can assign a Bus to a Car object. A Bus is a substitute for a Car.
Car anotherCar = new VintageCar();  // We can assign a VintageCar to a Car object. A VintageCar is a substitute for a Car.
```

Static and dynamic Type of an Object – A Tale of multiple Types – Part 1

- The concept of objects is getting a bit more complex after the introduction of inheritance.

- The complexity arises, because an object can now have multiple types! Let's understand why this is the case.
  - Let's start with a reference of type *Car*, which does not refer to any object. Instead it is a null reference:

```
Car car = null;
```

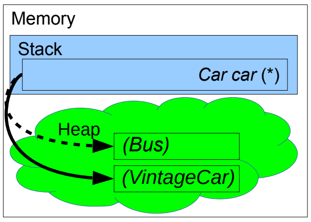
  - As we just learned, we can assign a *Bus* object to *car*. Because a *Bus* is a *Car*, substitutability is given there:

```
car = new Bus();
```

  - => *car* does indeed have two types now! (1) *Car* is the type of the reference and (2) the referenc**ed** type is *Bus*.
  - Now we assign a *VintageCar* object to *car*. Sure, this works, because a *VintageCar* is a *Car*, and substitutability is given as well:

```
car = new VintageCar();
```

  - *Car* is still the reference type, but now *VintageCar* is the referenc**ed** type.

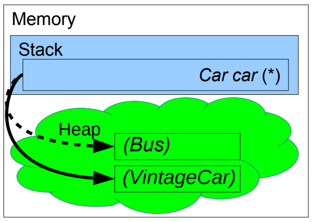


Static and dynamic Type of an Object – A Tale of multiple Types – Part 2

```java
// Static type of car: Car; null reference
Car car = null;
```

```java
// Static type of car: still Car; dynamic type of car: Bus
car = new Bus();
```

```java
// Static type of car: still Car; dynamic type of car: VintageCar
car = new VintageCar();
```



- Compile time: the reference type of *car* is always the same, i.e. *Car*. Therefor we call it the static type of *car*.
  - Notice, that during compile time the static type of an object is relevant.

Good to know:
The term "static type", is a general oo term and has nothing to do with Java's static keyword!

- Run time: the referenced type of *car* was *Bus* and then *VintageCar*. Because it varies, we call those dynamic types of *car*.
  - Notice, that during run time, objects on the heap are relevant, thus the dynamic type of an object is relevant.

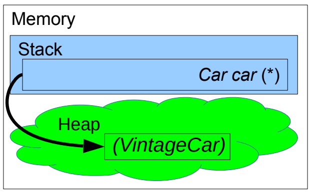
- => The key point: a reference of static type can refer to objects of different dynamic type at run time.
  - Objects of varying dynamic type can be referenced by an object of a static type.

- Dynamic types are sometimes called "run time types".
- There are languages in which variables have no static type at all, e.g. Smalltalk and JavaScript.

Static and dynamic Type of an Object – Downcasting – Part 1

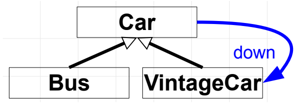
- So, an object can have a static and a dynamic type.
  - The type of the reference variable of the object is the static type of the object. -> The static type is the type of the variable.
  - The type of the referenced object, i.e. the "data behind the variable in the heap" is the dynamic type of the object.

```java
Car car = new VintageCar();
car.setEngine(new Engine());
```



- Java allows to "squeeze" the object of dynamic type out of the statically typed reference variable.
  - This "squeeze" operation is the simple cast-operation we already know!

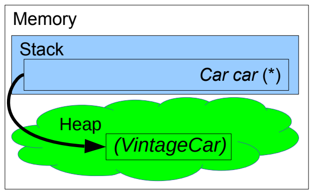
```java
VintageCar vintageCarBehind = (VintageCar)car; // Cast "the VintageCar out of car".
vintageCarBehind.crankUntilStarted();
```



- This kind of casting is called "downcasting", because we cast down the type hierarchy (see the UML diagram).
  - We cast the *VintageCar* out of *car* and initialize *vintageCarBehind* of static type *VintageCar* with the result.
  - Metaphor: we put contact lenses on *car* to have a look at the dynamic type.
  - Via the reference *vintageCarBehind* can access the methods of the dynamic type "behind" *car*.

Static and dynamic Type of an Object – Downcasting – Part 2

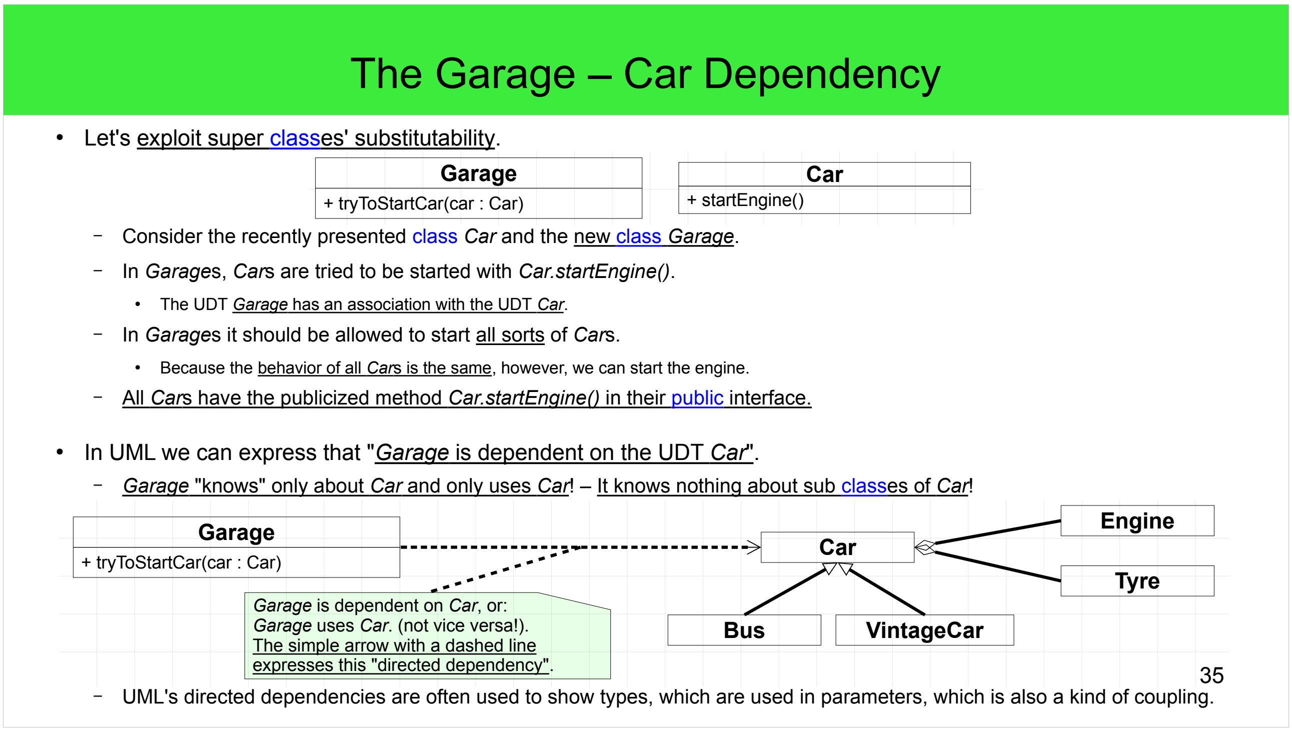
```
Car car = new VintageCar();
```



- However, if we <u>try to cast the wrong dynamic type out of the reference</u>, it will result in an <u>error</u>!

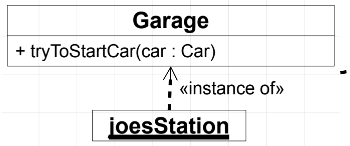
```
Bus busBehind = (Bus)car ; // Invalid at run time! Cast "a Bus out of car".
```

  - <u>This code will compile</u>, because *car* (static type *Car*) <u>could potentially refer to an object of type *Bus*</u> (dynamic type).
  - <u>Downcasts are type checked at run time.</u> Sometimes they are called <u>"dynamic casts"</u>.
  - But this code will fail at run time with a <u>*ClassCastException*</u>. We try to cast down to *Bus*, but *car* still refers to a *VintageCar*!
  - <u>Metaphor: we put on the wrong contact lenses on and hit the wall.</u>



- The term "association": Aggregation and inheritance are nothing but special (and also tighter) associations.
- OO seems to be perfectly good to build GUIs, because the visual representation of GUIs can be directly simulated with oo UDTs and oo paradigms (e.g. "has"- and "is a"-associations of forms and controls). – In opposite to procedural or functional programming.

Using Cars in Garages




```
// <Garage.java>
public class Garage {
	public void tryToStartCar(Car car) {
		car.startEngine();
	}
}
```

- Now let's inspect the class *Garage* and its method *Garage.tryToStartCar()*:
  - We've just learned that the more special UDT *Bus* can substitute *Car*, so this is allowed:

```
Garage joesStation = new Garage();
Car seatLeon = new Car();
Bus mercedesIntegro = new Bus();
joesStation.tryToStartCar(seatLeon); // Calls Car.startEngine().
// >start Car
joesStation.tryToStartCar(mercedesIntegro); // Also calls Car.startEngine(): it was inherited from Car.
// >start Car
```

  - We'll use the *Garage* instance *joesStation* in upcoming examples.

- The substitutability allows us to pass more special types to *Garage.tryToStartCar()*.
  - Calling *startEngine()* on the passed *car* always calls *Car.startEngine()*, because *Car.startEngine()* is inherited by all subtypes!
  - Well, this "rigidness" can be a problem! Let's continue our analysis!

- This example shows how the "substitution principle" allows that the types of a method's arguments (dynamic types) need not to be exactly the types of the declared parameters (static types).

The Problem of special Behavior

- To make the mentioned rigidness-problem visible we'll look at the implementation of the UDT *VintageCar* once again:

```java
// <VintageCar.java>
public class VintageCar extends Car { // (other members omitted)
	public boolean crankUntilStarted() {
		// pass
	}
	public boolean hasStarter() {
		// pass
	}
}
```

  - Some *VintageCar*s can only be started with a crank, because they have no starter!

- Ok, then we're going to give our *fordTinLizzie* to *joesStation*:

```java
VintageCar fordTinLizzie = new VintageCar();
fordTinLizzie.setEngine(new Engine());
joesStation.tryToStartCar(fordTinLizzie); // Oops! Calls the inherited Car.startEngine() and this is not enough!
// This is no compiler error, but a logical error: the fordTinLizzie can't be started in tryToStartCar()!
```

  - Nobody in *joesStation* knows how to start our *fordTinLizzie*!
  - They can't just turn the key or hold a button and keep going!
  - The passed *Car* is a *VintageCar* (dynamic type) – this is not respected in *Garage.tryToStartCar()*.
  - Maybe *VintageCar* is so special that we have to enhance *Garage.tryToStartCar()*?

Type Flags to the Rescue

- We can solve the problem by the introduction of a ***CarType* flag field**:

```java
// <CarType.java>
// An enum representing Car types:
public enum CarType {
        CAR_TYPE,
        BUS_TYPE,
        VINTAGE_CAR_TYPE
}
```

```java
// Add a CarType field to the UDT Car:
public class Car { // (members hidden)
        private CarType carType; // The type flag.

        public CarType getCarType() {
                return carType;
        }
        public void setCarType(CarType carType) {
                this.carType = carType;
        }
}
```

- Let's apply the *CarType* flag on our *Car* types:

```java
// Create a Car object and flag it as being of "CarType.CAR_TYPE".
Car seatLeon = new Car();
seatLeon.setCarType(CarType.CAR_TYPE);
// Create a VintageCar object and flag it as being of "CarType.VINTAGE_CAR_TYPE".
Car fordTinLizzie = new VintageCar();
fordTinLizzie.setCarType(CarType.VINTAGE_CAR_TYPE);
```

  - How will this help us? => With this flag we can identify the dynamic type of a *Car*!

Type Flags in Action

- Now we have to modify *Garage.tryToStartCar()* to interpret the *CarType* flag:

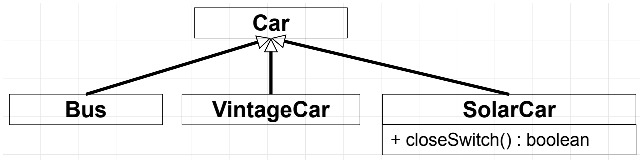
```
public class Garage {
	public void tryToStartCar(Car car) {
		// If car's CarType flag is CarType.VINTAGE_CAR_TYPE start it in a special way:
		if (CarType.VINTAGE_CAR_TYPE == car.getCarType()) {
			// Cast the dynamic type "out of" car:
			VintageCar vintageCar = (VintageCar)car;
			// ...to access VintageCar's interface:
			if (!vintageCar.hasStarter()) {
				vintageCar.crankUntilStarted();
			} else {
				vintageCar.startEngine();
			}
		// Else use the default start procedure for other cars:
		} else {
			car.startEngine(); // Start other cars just by calling Car.startEngine().
		}
	}
}
```

- Yes, with this implementation of *Garage.tryToStartCar()* we can start *fordTinLizzie*:

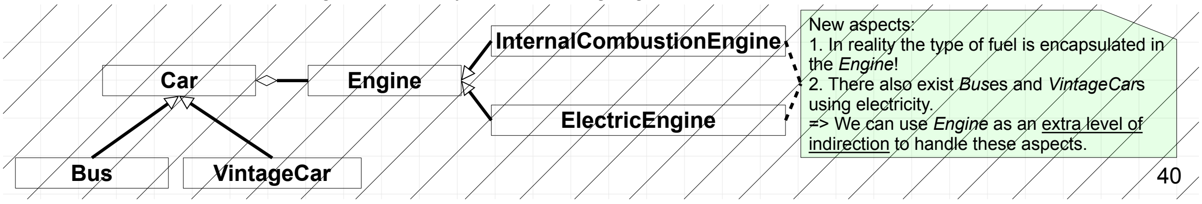
```
VintageCar fordTinLizzie = new VintageCar();
fordTinLizzie.setEngine(new Engine());
fordTinLizzie.setCarType(CarType.VINTAGE_CAR_TYPE);
joesStation.tryToStartCar(fordTinLizzie);	// Ok! Garage.tryToStartCar() will pick the correct start-algorithm
						// depending on the CarType flag!
```

Other special Types need Handling

- After a while the clientele at *joesStation* grows:
  - More and more clients bring their solar cars to *joesStation*.
  - But solar cars can't be directly started in *Garages*! We assume that a special battery switch needs to be closed before starting!
  - As an oo programmer we start by encapsulating the solar car concept into a new UDT:



- Btw.: Virtually our type hierarchy is becoming dubious!
  - In fact we needed to redesign the hierarchy, but we're not going to do this in this here!



Type Flags are becoming nasty...

- After modifying the enum *CarType* we can modify *Garage.tryToStartCar()* accordingly:

```
// The CarType flag SOLAR_CAR_TYPE
// needs to be added:
public enum CarType {
	CAR_TYPE,
	BUS_TYPE,
	VINTAGE_CAR_TYPE,
	SOLAR_CAR_TYPE
}
```

```
// <Garage.java>
public class Garage {
	public void tryToStartCar(Car car) {
		if (CarType.VINTAGE_CAR_TYPE == car.getCarType()) {
			/* pass */
		} else if (CarType.SOLAR_CAR_TYPE == car.getCarType()) {
			SolarCar solarCar = (SolarCar)car;
			solarCar.closeSwitch();
			solarCar.startEngine();
		} else {
			car.startEngine();
		}
	}
}
```

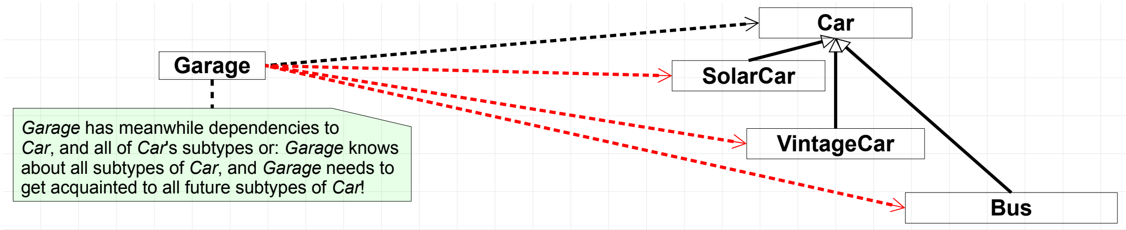
```
SolarCar solarCar = new SolarCar();
fordTinLizzie.setCarType(CarType.SOLAR_CAR_TYPE);
joesStation.tryToStartCar(solarCar); // Ok! Garage.tryToStartCar() will pick the correct start-algorithm!
```

- It works, but should we really add more *CarType* flags for upcoming *Car* types?
  - Let's remember that we also have to add more else ifs for upcoming *Car* types in *Garage.tryToStartCar()*!
  - It's uncomfortable and unprofessional doing the same all over and knowing about that!
  - This approach is also very error prone!

- Hm... what is the basic problem we've encountered just now?

A "Pasta-object-oriented" Hierarchy

- We changed the dependency "*Garage – Car*" in a negative way!



  - *Garage* needs to know each subtype of *Car* now and in future!
    - We can see this, because *Garage.tryToStartCar()* inspects the dynamic type of the *Car* parameter.
    - We needed to interpret flags, because we had to deal with different interfaces of the dynamic type. (Then we also have a dependency to the enum!)
    - We have to make down casts to the dynamic type respectively!
    - The bad consequence: *Garage.tryToStartCar()* must be modified when new *Car* types emerge!
  - We can also spot a "bad smell" in the class diagram!
    - There are dependencies to the super type *Car* and to all of its (currently known) subtypes.
- It lead us to spaghetti programming the oo-way! How can we improve that?

- Often too many arrows pointing from a single type to multiple other types is a bad smell. – It can be nicely spotted in a UML class diagram.
- What we've just seen on using enum type flags is called the "typedef-enum-antipattern".

Overriding Methods – Our first Try – Part 1

- The idea is to not let *Garage.tryToStartCar()* select the start algorithm!
  - Instead, the start algorithm should be put into each individual *Car*-type!
  - To make this work, we *@Override* the method *Car.startEngine()* for each individual *Car*-type to provide a special implementation:

```java
// <Garage.java>
public class Garage {
	public void tryToStartCar(Car car) {
		if (CarType.VINTAGE_CAR_TYPE == car.getCarType()) {
			VintageCar vintageCar = (VintageCar)car;
			if (!vintageCar.hasStarter()) {
				vintageCar.crankUntilStarted();
			} else {
				vintageCar.startEngine();
			}
		} else if (CarType.SOLAR_CAR_TYPE == car.getCarType()) {
			SolarCar solarCar = (SolarCar)car;
			solarCar.closeSwitch();
			solarCar.startEngine();
		} else {
			car.startEngine();
		}
	}
}
```

```java
// <Garage.java>
public class Garage {
	public void tryToStartCar(Car car) {
		car.startEngine();
	}
}
```

```java
// <VintageCar.java>
public class VintageCar extends Car {
	@Override
	public void startEngine() {
		if (!hasStarter()) {
			crankUntilStarted();
		} else {
			// startEngine();
		}
	}
}
```

```java
// <SolarCar.java>
public class SolarCar extends Car {
	@Override
	public void startEngine() {
		closeSwitch();
		// startEngine();
	}
}
```

- Now we have overridden *Car.startEngine()* with *VintageCar.startEngine()* and with *SolarCar.startEngine()*.
  - Notice, that the statement *startEngine()* is commented in each override, because it is not yet quite correct!

Overriding Methods – Our first Try – Part 2

- The new implementation of *Garage.tryToStartCar()* does now just call *startEngine()* on the passed *Car*.
  - The knowledge about and the responsibility for the start algorithm is fully delegated to the dynamic type "behind" the param *car*.
  - We have no type-check or downcasting in *Garage.tryToStartCar()*!
  - *Garage* only knows the type *Car*, the dynamic type of *car* is unknown to *Garage*!
  - Nevertheless, in *Garage.tryToStartCar()* the *startEngine()*-method of the dynamic type is called.

- The ability to call a method of a dynamic type through a reference of static type is called (object-oriented) polymorphism.

```
joesStation.tryToStartCar(seatLeon);
// >start Car
```

```
joesStation.tryToStartCar(new SolarCar());
// >start SolarCar
```

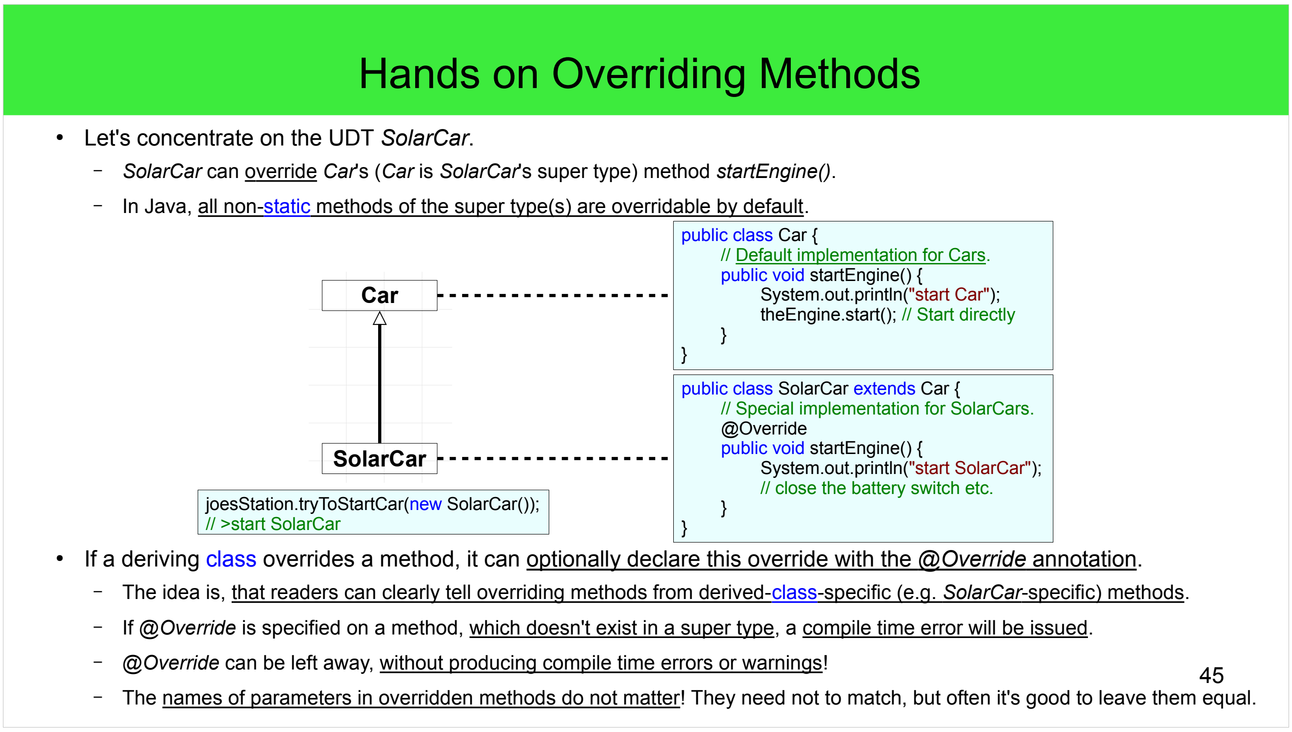
```
joesStation.tryToStartCar(fordTinLizzy);
// >start VintageCar
```

```
// <Garage.java>
public class Garage {
	public void tryToStartCar(Car car) {
		car.startEngine();
	}
}
```

```
// <Car.java>
public class Car {
	public void startEngine() { // Trace message:
		System.out.println("start Car");
	}
}
```

```
// <SolarCar.java>
public class SolarCar extends Car {
	@Override
	public void startEngine() { // Trace message:
		System.out.println("start SolarCar");
	}
}
```

```
// <VintageCar.java>
public class VintageCar extends Car {
	@Override
	public void startEngine() { // Trace message:
		System.out.println("start VintageCar");
	}
}
```

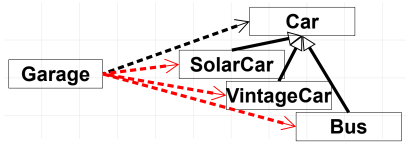


- Java 14 introduced the annotation *@Serial*, which works similar to *@Override*. It can optionally be specified at a set of 2 specific fields (e.g. *serialVersionID*) and 5 specific methods dealing with serialization to let the compiler check their correctness.

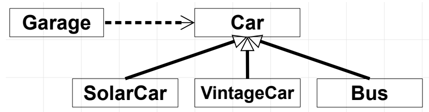
Where is Polymorphism happening? – Part 1

- With this implementation of *Garage*, we have a very good class hierarchy:

```java
// <Garage.java>
public class Garage {
	public void tryToStartCar(Car car) {
		if (CarType.VINTAGE_CAR_TYPE == car.getCarType()) {
			VintageCar vintageCar = (VintageCar)car;
		} else if (CarType.SOLAR_CAR_TYPE == car.getCarType()) {
			SolarCar solarCar = (SolarCar)car;
		} // ... and so forth
	}
}
```




```java
// <Garage.java>
public class Garage {
	public void tryToStartCar(Car car) {
		car.startEngine();
	}
}
```

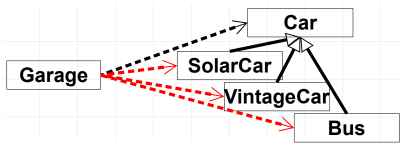


- Before: *Garage* needed to know *Car* and all subtypes of *Car*, which should be started in *Garage.tryToStartCar()*.
  - When a new *Car*-subtype *MagicCar* should be "startable" in *Garage.tryToStartCar()*:
    - *Garage.tryToStartCar()* must be changed considering all details how to start a *MagicCar* object and a new dependency must be added.
- After: *Garage* only has a dependency to *Car* and no dependencies to any (future) subtype of *Car*!
  - When a new *Car*-subtype *MagicCar* should be "startable" in *Garage.tryToStartCar()*:
    - *MagicCar* needs to @Override *Car.startEngine()* in an appropriate way. – Details how to start a *MagicCar* go into *MagicCar.startEngine()*'s override.
    - *Garage* needs not be changed!

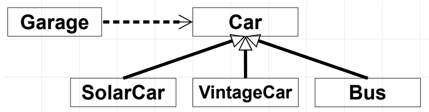
- Polymorphism:
  - In future, the implementation of the method *startEngine()* of any other sub type of *Car* could be called!
- When a method is called, the dynamic type of the object determines, which implementation of that method is really called.

Where is Polymorphism happening? – Part 2

```java
// <Garage.java>
public class Garage {
        public void tryToStartCar(Car car) {
                if (CarType.VINTAGE_CAR_TYPE == car.getCarType()) {
                        VintageCar vintageCar = (VintageCar)car;
                } else if (CarType.SOLAR_CAR_TYPE == car.getCarType()) {
                        SolarCar solarCar = (SolarCar)car;
                } // … and so forth
        }
}
```




```java
// <Garage.java>
public class Garage {
        public void tryToStartCar(Car car) {
                car.startEngine();
        }
}
```

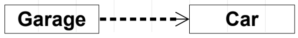


- Before: we hurt the Open Closed Principle (OCP)!
  - This principle tells us, that types should be closed for modification but open for extension.
  - We'd to modify *CarType* (add constants) and *Garage* (add branches for new *CarType* constants and add new *Car* sub classes).
- After: we just introduce new sub classes of *Car* and *@Override Car.startEngine()*. *Garage* just calls *Car.startEngine()*.
  - *Garage* is now closed for modification: it doesn't need to be changed, when new *Car* sub classed are added. *CarType* is gone!
  - *Car* is now open for extension: new *Car* sub classes just *@Override Car.startEngine()* their specific way.
  - The vector of (model/design) change is now in new *Car* sub classes, supporting the OCP!

Polymorphism – Handle Types and Body Types

- In our example *Car* is a <u>handle type</u> for *Garage*.

```
// <Garage.java>
public class Garage {
        public void tryToStartCar(Car car) {
                car.startEngine();
        }
}
```



With *Car* we have an oo-design, that has a clear location, in which it can be extended in future: by adding new *Car* sub classes. We already used the phrase <u>"vector of change"</u>: our <u>vector of change in this design</u> is when new *Car* types must be handled.

  - *Garage*, only knows *Car* and only has *Car* in its <u>interface</u> and only calls *Car*'s methods.
  - In terms of polymorphism we call the type *Car* a <u>handle type for *Garage*</u>.

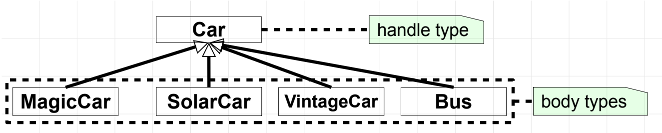
- The types of the objects being passed to *Garage.tryToStartCar()* are <u>behind</u> the handle type.

```
joesStation.tryToStartCar(new SolarCar());
// >start SolarCar
```

```
joesStation.tryToStartCar(fordTinLizzy);
// >start VintageCar
```

```
joesStation.tryToStartCar(seatLeon);
// >start Car
```

```
joesStation.tryToStartCar(new MagicCar());
// >start MagicCar
```

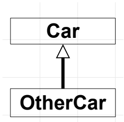


<u>Polymorphism:</u> *MagicCar* can be used with *Garage.tryToStartCar()*, because *Car* can be used with *Garage.tryToStartCar()*!

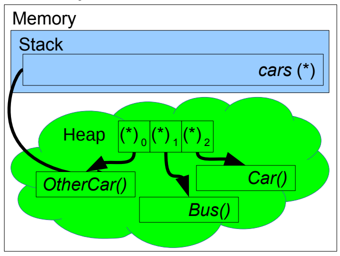
  - Subtypes of the handle type *Car*, are <u>unknown to *Garage*</u>. The connection between *Garage* and *Car*-subtypes is <u>only the handle type *Car*</u>.
  - In terms of polymorphism we call subtypes of *Car* <u>body types</u>.
  - The <u>handle type *Car* works like an isolator</u> between *Garage* and special body types.

Polymorphism and Arrays – Part 1

- When we define an array of super class objects, the elements can be of any derived class:




```
Car[] cars = {new OtherCar(), new Bus(), new Car()};
driveAway(cars);
```



  - The array-element type is the static type *Car*, but the types of the referenced objects can be different dynamic types (in the same array).

- This means, we can use the elements of *cars* polymorphically and call methods of the static type *Car* on each element:

```
public void startAll(Car[] cars) {
	for (Car car : cars) { // This works for each element in cars exploiting polymorphism:
		car.startEngine();
	}
}
```

- However, if we want to access the *Bus* element in *cars*, in order to call a special *Bus* methods we have to do a down cast:

```
Bus bus = (Bus) cars[1]; // This only works for the 2nd element of cars, because it refers to a Bus:
boolean canIEnter = bus.newPassengerCanEnter();
```

Polymorphism and Arrays – Part 2

```
Car[] cars = {new OtherCar(), new Bus(), new Car() };
```

```
Bus bus = (Bus) cars[1];
boolean canIEnter = bus.newPassengerCanEnter(); // This works, because cars[1] really refers to a Bus.
```

- The down cast "pulls" the *Bus* object out of the statically types *Car[] cars*.
  - If you will, we are applying cast contact lenses on *cars[1]* to get the *Bus* object behind this element.
- All right, but when we apply a wrong contact lens, i.e. try to cast down to the wrong type, we're in trouble:

```
Bus bus = (Bus) cars[0]; // java.lang.ClassCastException: OtherCar cannot be cast to Bus
boolean canIEnter = bus.newPassengerCanEnter();
```

- The example above is a typical problem of type safety. In Java we can make a <u>type check before trying to down cast</u>:

```
boolean canIEnterAnyCar = passengerCanEnterAnyCar(cars);
```

```
public boolean passengerCanEnterAnyCar(Car[] cars) {
	for (Car car : cars) {
		if (car instanceof Bus) {	// This (dynamic) type check evaluates
			Bus bus = (Bus) car;	// to true only for cars[1] because it is a Bus.
			if (bus.newPassengerCanEnter()) {
				return true;
			}
		}
	}
	return false;
}
```

  - Here we see a "reasonable" application of instanceof, which checks the run time type of a reference.
  - We just front load a check using instanceof, before we make the down cast.

Polymorphism and Arrays – Part 3

```
public boolean passengerCanEnterAnyCar(Car[] cars) {
	for (Car car : cars) {
		if (car instanceof Bus) { // This method is dependent of the special Car Bus, and not only on the super type Car!
			Bus bus = (Bus) car;
			if (bus.newPassengerCanEnter()) {
				return true;
			}
		}
	}
	return false;
}
```

- It should be said, that down casting as well as instanceof type checking should be avoided.
  - It requires code to know/assume the dynamic type of objects, which leads to oo-spaghetti effects, i.e. too many dependencies.
    - Here, *passengerCanEnterAnyCar()* must know *Bus* and not only *Car*. The down cast "pulls" the *Bus* object out of the statically types *Car[] cars*.
  - Down cast and instanceof both do an (expensive) type check! But the down cast may fail with an *Exception*!
    - This means, that applying instanceof and then applying the cast performs the same type check even twice!

- If down casts are "not good", how should such algorithms be implemented?
  - The answer is: "exploit polymorphism". We can program the algorithms with minimal or no type checks, which also means minimal dependencies to other types, esp. those, which change often (e.g. having to deal with new *Car* types).
  - This can be done by creating another layer of indirection to make the algorithm more flexible, but more robust against changes. This way to program is more complex and needs some experience, it involves so called oo design patterns.

Polymorphism and Arrays – Array-Covariance and its Problems

- Let's inspect another method, which accepts an array of *Car*s and sets the element on index *index* to a new *Car* instance:

```
public static void addCar(int index, Car[] cars) { // Yes, this methods looks harmless!
	cars[index] = new Car();
}
```

  - A special feature of Java is, that we can set/pass an array of sub class elements where an array of super class elements is declared.
  - Using special terms, we state, that Java's arrays are <u>covariant</u>, e.g. we could pass a *Bus[]* to *addCar()*:

```
Bus[] busses = { new Bus(), new Bus(), new Bus() };

addCar(0, busses); // java.lang.ArrayStoreException: Car
```

  - But … the call fails at run time! Why?

- The idea of covariant arrays is nice but not type safe!
  - *addCar()* awaits *cars* to refer to *Car*s or objects of more derived class.
  - Covariance allows to pass an array with more special element types to *addCar()*, e.g. *Bus[]*.
  - But in a *Bus[]* we cannot store a *Car* object, because a *Car* is more general than *Bus*, it throws an <u>*ArrayStoreException*</u>!

- Virtually, covariance of arrays is dubious, because <u>*Car* and *Bus* are related</u>, <u>but *Car[]* and *Bus[]* are not related</u>!

Polymorphism – chooses Implementation during Run Time

- The inherited implementation of a method could be inadequate!
  - *Car.startEngine()* inherited by *SolarCar* must close the battery switch, before starting the *Engine*!
- Inheriting UDTs can *@Override* inherited implementations of methods.
  - In Java, all non-static methods are overridable by default.
- Overriding (late binding) is not overloading (early binding)!
  - Overriding takes effect during run time, overloading during compile time!
  - Overrides in a subtype mustn't change the signature of the overridden method.
- Calling implementations of dynamic types on objects is called oo polymorphism.
- On a first sight, polymorphism looks strange, but some say, it is **the** core principle of oo.
  - It is the basis for so called oo design patterns.

Polymorphism – some Definitions

- In Java and most other oo languages only oo polymorphism is officially called "polymorphism":
  - Oo polymorphism: Objects of varying dynamic body type can be referenced by an object of a static handle type.
  - Oo polymorphism happens at run time, therefore it is sometimes called dynamic binding or late binding.

- No oo polymorphism, but "ad-hoc polymorphism":
  - Method overloading and operator overloading.
    - Java does not support operator overloading.
  - This happens at compile time in Java, therefore it is sometimes called static binding or early binding.

```
// two overloads of sum():
static String sum(String lhs, int rhs) {
	return lhs + rhs;
}

static int sum(int lhs, int rhs) {
	return lhs + rhs;
}

String text = sum("of age: ", 23);	// calls sum(String, int)
int result = sum(34, 67)		// calls sum(int, int)
```

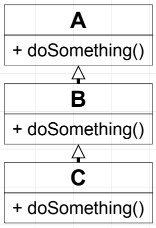
- No oo polymorphism, but parametric polymorphism:
  - Generic types and methods.
    - Java supports bounded parametric polymorphism.
  - Java resolves those at compile time.
  - We'll discuss those in a future lecture.

```
// the generic method single():
static <T> List<T> single(T one) {
	return List.of(one);
}

List<Car> singleCarList = single(new Car());
```

Polymorphism – Theory

- OO Polymorphism is a type of polymorphism related to generalization-specialization (via inheritance or Java interfaces).
- Compile time: A method is polymorphic, if it has different implementations with the same signature in a type hierarchy.
- Run time: having polymorphic methods in the type hierarchy, the actually called method is determined at run time.
  - This is called dynamic binding or late binding.
- In a multi-level polymorphic hierarchy, the most special method-override of the object's dynamic type will be called.
  - I.e. the @*Override*, that is in the type "closest" to the dynamic type from the hierarchy above. Consider:




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

  - => Sub class before super class:
    - If *C* derives *B* and *B* derives *A* and *C* and *B* both override *doSomething()*.
    - If an object is of dynamic type *C* and referenced by reference variable of static type *A*, *C*'s override would be called.

Thank you!