(4) Java Abstractions: Inheritance – Part 2

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!

Overriding Methods – Calling Methods of the super Class – Part 1

- In the last lecture we implemented *SolarCar*. In *SolarCar*s a switch needs to closed, before the engine can be started.
  - So our override of *SolarCar.startEngine()* could look like so:

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

```java
SolarCar solarCar = new SolarCar();
solarCar.startEngine();
System.out.println("SolarCar successfully started");
```

This statement will never be reached!

- It doesn't work! It doesn't work, because we're ending in an infinite recursion, i.e. *SolarCar.startEngine()* calls itself!

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

- We have to solve this: *SolarCar.startEngine()* needs to call *startEngine()* in the super class: it must call *Car.startEngine()*!

Overriding Methods – Calling Methods of the super Class – Part 2

- In Java we can call a method of the super class explicitly with the super keyword. This solves our recursion:

```java
// <Car.java>
public class Car { // UDTs Engine and Tyre elided.
	public void startEngine() {
		System.out.println("start Car");
		theEngine.start();
	}
}
```

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

```java
SolarCar solarCar = new SolarCar();
solarCar.startEngine();
System.out.println("SolarCar successfully started");
// >start Car
// >SolarCar successfully started
```

Car.startEngine() is called as intended and the recursion was resolved!

- Only methods of the direct super class can be called! I.e. there is no "super.super"!

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

Calling the Ctors of super Classes

- If the super classes have default ctors (dctors) they will be implicitly called by the ctors of subtypes:

```
public class Car { // (members hidden)
	public Car() {
		System.out.println("Car()");
	}
}
```

```
public class Bus extends Car { // (members hidden)
	/* pass */
}
```

```
Bus bus = new Bus();
// >Car() // Calls Car's dctor implicitly.
```

- If a super class doesn't provide a dctor, nothing can be implicitly called of course!

```
public class Car { // Car without dctor!
	public Car(double power) { /* pass */ }
}
```

```
public class Bus extends Car { // (members hidden)
	/* pass */
}
```

```
Bus bus = new Bus(); // Invalid! Super class Car provides no dctor.
```

  - Instead we have to call another ctor of the super class in the subtype's ctor explicitly.
    - Syntactically we have to call ctors of the super class in the subtype's ctor:

```
public class Bus extends Car {

	// Ctor, which calls one of Car's ctors.
	public Bus(double power) {
		super(power);
		//...
	}
}
```

```
Bus bus = new Bus(320); // Fine!
```

  - There are some peculiarities to be aware of:
    - Only the ctor of the direct super class can be called, i.e. there is no "super.super*(argument)*"!
    - The call to a super type's ctor must be the very first statement in the ctor!

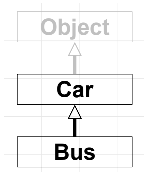
Everything is an Object – Part 1

- In the last lecture we introduced polymorphism as important principle of oo.
- In Java, polymorphism is very pervasive throughout the whole framework.
- In order to establish this pervasiveness, all objects in Java are implicitly inherited from a very super type: *Object*.
  - Yes, in Java there exists a type, i.e. a class, with the name *Object*.
- Virtually, our UDT *Car* is implicitly inherited from *Object*: A *Car* is an *Object*.

```
public class Car { // (members hidden)
}
```

```
public class Car extends Object { // (members hidden)
}
```

  - And the UDT *Bus* is transitively inherited from *Object*: A *Bus* is a *Car* and a *Car* is an *Object*, thus a *Bus* is also an *Object*!
  - => Every UDT is also of type *Object*, thus we can state, that "everything is an object/*Object*".
  - In upcoming class diagrams, we will leave the information about the super class *Object* away, it is implied!



- A framework, in which all UDTs inherit from the same type are sometimes said to have a "cosmic hierarchy".
  - In other words "cosmic hierarchy" means, that all objects have a common "root" type.

- Basically, a cosmic hierarchy is build up like: sun, solar system, galaxy, galaxy cluster, supercluster … There could be more levels in between, also the same levels enumerated here with different names, but the important idea is that the hierarchy starts with a single object on top, namely the sun. The word "cosmos" is greek for "order of the universe", its opposite is "chaos", which does not necessarily mean "un-order", but more or less "emptiness" or "the thing, which existed before cosmos".

Everything is an Object – Part 2

- The idea behind all UDTs being inherited from *Object* is simple: all UDTs will have a guaranteed default behavior!
  - In oo-terms "guaranteed behavior" means, that methods of *Object* are inherited to all subtypes and their presence is guaranteed.

- The type *Object* inherits some important methods to its subtypes:
  - *equals()* to compare objects for equality.
  - *hashCode()* to retrieve a unique integral code representing the object.
  - *toString()* to retrieve a textual representation of the object.

```java
// Somewhere in the JDK, the class Object is defined:
public class Object { // (declaration simplified)
	public boolean equals(Object other) { /* pass */ }
	public String toString() { /* pass */ }
	public int hashCode() { /* pass */ }
}
```

Object
+ equals(other : Object) : boolean + toString() : String + hashCode() : int

  - However, the important point is, that these three methods are inherited to all sub types, e.g. also to *Car*.

- To understand how these inherited methods make sense, we are going to discuss the concept behind *toString()*.
  - We'll discuss the other methods in upcoming lectures.

The Method Object.toString() inherits its Implementation

- In one of the past lectures we already called the method *toString()* when we used *StringBuilder*:

```
StringBuilder text = new StringBuilder(); // 1 (new StringBuilder object)
// Build the text:
text.append("a number: ");                // 2 append another String
text.append(42);                          // 3 append an integer
// Materialize the text into a new String object:
String effectiveText = text.toString();   // new String object
```

  - Hm, this code looks plausible and not very special. <u>However, meanwhile we know, that *toString()* is inherited from *Object*.</u>

- All right, because *Car* is also an *Object*, <u>we can expect *Car* having also the method *toString()*</u>! Let's call it!

```
Car car = new Car();
String carsStringRepresentation = car.toString();
System.out.println(carsStringRepresentation);
// >Car@4f4a7090  // The result of toString().
```

  - It looks really unspectacular! The *toString()*-implementation inherited from *Car* creates following <u>*String*-representation of an *Object*</u>:
    - The name of the object's <u>dynamic</u> type. "Car" in this case.
    - The symbol "@".
    - The hashcode of the object (the object's address in this case).

"<typeName>@<hashCode>"

- And now we're going to implement our own *toString()* in a <u>more suitable way for the UDT *Car*</u>.
  - Because the <u>inherited default-*toString()*-implementation from *Object* is insufficient</u>. The created *String*-representation is ugly!

Let's override Object.toString() in Car

- How to implement *toString()* for the UDT Car?
  - Let's add a field to *Car*, representing the vehicle identification number and this number may act as *String* representation.
  - This field should be set in a ctor of *Car*. It can be a blank final, because vehicle identification number won't change after creation.
  - A suitable *@Override* of *toString()* may look like this:

```
public class Car { // (other members omitted)
	private final String vehicleID;

	public Car(String vehicleID) {
		this.vehicleID = vehicleID;
	}
	@Override
	public String toString() {
		return "vehicleID: "+this.vehicleID;
	}
}
```

- Having this *@Override* of *toString()*, the *String* representation looks much more appealing than before:

```
Car car = new Car("W0L000051T2123456");
String carsStringRepresentation = car.toString();
System.out.println(carsStringRepresentation);
// >vehicleID: W0L000051T2123456 // Looks better, isn't it?
```

The Method toString() – Polymorphism and Pervasiveness

- The method *toString()* is an example of a method, that is really pervasive. Let's discuss some examples.

- Example 1: The method *toString()* is automatically called, when the operator-+ is called for *String* concatenation:

```
Car car = new Car("W0L000051T2123456");
String text = "Information about the car: "+car; // + calls toString() implicitly!
System.out.println(text);
// >Information about the car: vehicleID: W0L000051T2123456
```

  - As can be seen, *toString()* is not explicitly called here, the application of '+' calls *toString()* implicitly!
  - If '+' is applied on a null-reference (i.e. the lhs of '+' is null) the resulting *String* has the value "null".

Good to know

```
// '+' can be applied on null:
Car car = null;
String result = car + " test";
// result = "null test"

// But calling toString() on null
// throws an NPE:
String result2 = car.toString();
```

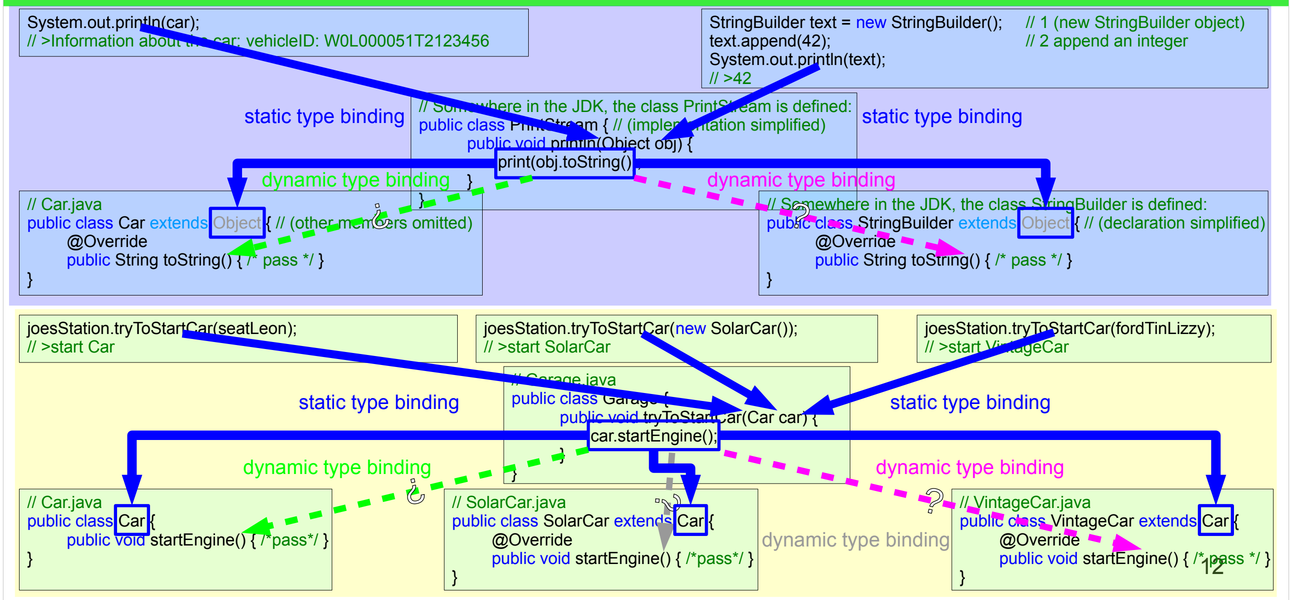
- Example 2: Let's exploit polymorphism to make matters even simpler!
  - The object *System.out* is of type *PrintStream*. *PrintStream* provides the method *println()* accepting an *Object*-argument:

```
// Somewhere in the JDK, the class PrintStream is defined:
public class PrintStream { // (declaration simplified)
        public void println(Object obj) { /* pass */ }
}
```

  - The interesting point is, that we can pass any object directly to *println(Object)*, because all UDTs are *Object*s!
  - Internally, *println(Object)* works in a very simple way: it just calls *toString()* on the passed argument!
  - I.e. we can call *System.out.println()* and pass the *car* object directly! *System.out.println()* calls *toString()* on *car* internally!

```
Car car = new Car("W0L000051T2123456");
System.out.println(car); // Calls toString() within PrintStream.println()!
// >vehicleID: W0L000051T2123456
```

The Application of Polymorphism is a Basis for Patterns



The Method toString() – … with Arrays – Part 1

- When we discussed arrays, we noticed, that the *String*-representation of arrays is pretty "unintuitive":

```
Car[] cars = {
        new Car("W0L000051T2123456"),
        new Car("W0L000041G2153456"),
        new Car("F0Z000030H2153972")
};
System.out.println("The cars: "+cars.toString());
// >The cars: [LCar;@17a7cec2
```

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
System.out.println("The numbers: "+numbers.toString());
// >The numbers: [I@6d03e736
```

  - What we are seeing here, is how the types *Car[]* and int[] *@Override Object.toString()*.
  - Yes, arrays of different element type are represented by different classes.

- We cannot influence the implementation of "Car[].toString()", "int[].toString()" and other array-types:
  - The basic format is: "[<typeName>@<hashCode>"
    - So, the prefixed "[" means "this is the *String* representation of an array" ...
  - If it is an array of primitive type, <typeName> can be "I" (int), "J" (long), "S" (short), "C" (char), "Z" (boolean) and "B" (byte).
    - It makes something like "[I@6d03e736" for an int[].
  - It it is an array of reference type/UDT the inner format of <typeName> is: "L<className>;"
    - So, the "L" after the "[" denotes a *String* representation of a UDT-array of element-type "<className>", incl. an extra ";".
    - It makes something like "[LCar;@17a7cec2" for a *Car[]*.

The Method toString() – … with Arrays – Part 2

- We cannot influence the implementation of "Car[].toString()", but we can call our *Car.toString()* for all elements in *cars*.
  - If we had to implement this on our own, we'd loop through *cars*, call *toString()* for each *Car* and build a *String* from the results.
  - But when we discussed arrays, we also learned about *Arrays.toString()*, which accepts an array, so no need for loops:

Arrays
+ toString(a : Object[]) : String

```
String carArrayStringRepresentation = Arrays.toString(cars);
System.out.println(carArrayStringRepresentation);
// >[vehicleID: W0L000051T2123456, vehicleID: W0L000041G2153456, vehicleID: F0Z000030H2153972] // Ok!
```

- Mind that we can pass a *Car[]* to *Arrays.toString()*, which awaits an *Object[]*, we already mentioned this effect: covariance.
  - It looks intuitive, if everything is an *Object*, why shouldn't an array of everything be an *Object[]*?
  - Indeed, it works as expected: *Arrays.toString()*'s polymorphic handling/calling of *toString()* for each *Car* in the *Object[]* "just works".

- Remember the deeper truth is the special substitutability of arrays in Java known as array covariance:
  - Java assumes that if *B* can be a substitute of *A*, also *B[]* can be a substitute of *A[]*. This is called covariance.
    - *Car* can be a substitute for *Object*, hence *Car[]* can substitute *Object[]*.
  - Although covariance of arrays is intuitive, it can become a problem, when code needs to set elements the array.
    - Mind that *Arrays.toString()* only needs to read elements in the array, which is no problem!

Overriding Object.equals() – Part 1

- Besides *toString()*, we will discuss another very important method derived from *Object*, namely *equals()*.

- Obviously, *equals()* is a method, which somehow deals with the equality of objects.

- However, the general question we have to answer is: how would we compare *Car*s, e.g. what is the equality criterion?
  - Let's e.g. add the *licencePlateID* to *Car*, it can check the equality of *Car*s. Then we compare *Car*s like this:

```
public class Car { // (other members omitted)
	private String licencePlateID;

	public void setLicencePlateID(String licencePlateID) {
		this.licencePlateID = licencePlateID;
	}
	public String getLicencePlateID() {
		return licencePlateID;
	}
}
```

```
Car car1 = new Car();
car1.setLicencePlateID("KL-EK 267");
Car car2 = new Car();
car2.setLicencePlateID("BIR-EL 954");

boolean carsAreEqual = car1.getLicencePlateID().equals(car2.getLicencePlateID());
// >carsAreEqual = false
```

- Of course, this is a solid solution. There is no need to change it basically!

- But, there is a standard way to implement checking for equality in Java, but why is it a good alternative?

Overriding Object.equals() – Part 2

- After a while, we decide, that *licencePlateID* is not a good criterion to compare *Car*s!
  - Does *licencePlateID* really represent the equality of a *Car*? What, if, e.g. a *Car*'s *licencePlateID* was doctored?
- An alternative idea is to use the *vehicleID* of a *Car*, which is difficult to erase or doctor.
  - The *vehicleID* is a blank final, which is only set in the ctor, so can't be changed or "doctored".
  - The only thing we have to add in *Car* is the method *Car.getVehicleID()* to access the *vehicleID* publicly.

```
public class Car { // (other members omitted)
	private final String vehicleID;

	public Car(String vehicleID) {
		this.vehicleID = vehicleID;
	}
	public String getVehicleID() {
		return vehicleID;
	}
}
```

- However, alas, all the code, which used *licencePlateID* to equality-compare *Car*s has to be changed to use the *vehicleID*:

```
Car car1 = new Car("W0L000051T2123456");
Car car2 = new Car("WVWZZZ1JZ3W386752");

boolean carsAreEqual = car1.getVehicleID().equals(car2.getVehicleID());
// >carsAreEqual = false
```

Overriding Object.equals() – Part 3

- All right, it means, that each time we change our opinion on how to compare *Car*s, other code must change as well.
  - When other code should compare *Car*s after the *licencePlateID*, we've to let people change their code using *getLicencePlateID()*.
  - When other code should compare *Car*s after the *vehicleID*, we've to let people change their code using *getVehicleID()*.
- In the *Car-Garage* example, we have already seen, that having dependencies to varying things is problematic.
  - New *Car* subtypes leaded to modify *Garage.tryToStartCar()*.
  - Here: our idea to change *Car*'s equality-comparison leads to have any code using comparison to change! – This is bad!
- We can use the same solution we've used for *Car-Garage*: let's encapsulate the equality-comparison-algorithm into *Car*!
- And the method to encapsulate equality-comparison is standardized in Java: *equals()*.

```
public class Car { // (other members omitted)
	private final String vehicleID;

	@Override
	public boolean equals(Object other) {
		Car otherCar = (Car)other;
		return this.vehicleID.equals(otherCar.vehicleID);
	}
}
```

```
Car car1 = new Car("W0L000051T2123456");
Car car2 = new Car("WVWZZZ1JZ3W386752");

boolean carsAreEqual = car1.equals(car2);
// >carsAreEqual = false
```

Overriding Object.equals() – Part 4

```
// Somewhere in the JDK, the class Object is defined:
public class Object { // (declaration simplified)
	public boolean equals(Object other) {
		/* pass */
	}
}
```

```
public class Car extends Object { // (other members omitted)
	private final String vehicleID;

	@Override
	public boolean equals(Object other) {
		Car otherCar = (Car)other;
		return this.vehicleID.equals(otherCar.vehicleID);
	}
}
```

Good to know:
The name of the parameter in overriding methods doesn't matter, it can differ from the one in the super class. Instead of *other* we could have used the name *otherObject*.

- All right, what is going on here?
  - First and foremost: *equals()* is already defined in *Car*'s implicit super class *Object*.
  - We are using the annotation *@Override* to signal, that we're going to override *Object.equals()* in *Car*.
  - *equals()* accepts an argument stored in the parameter *other*, which we have to down cast to the type *Car*.
  - After the down cast, we can equality-compare, what ever data/field/method-result we want between the two *Car* objects.
    - The two *Car* objects to compare are this and the *Car* object, which is contained in *other*.

- But why do we have deal with a parameter of type *Object* instead of *Car*? Why are we forced to cast down?

Overriding Object.equals() – Part 5

- Well, what happens, if we change the signature of *Car.equals()* to accept a *Car* instead of an *Object*:

```
public class Car { // (other members omitted)
	private final String vehicleID;

	@Override // Invalid: Method doesn't override method from its super class
	public boolean equals(Car otherCar) {
		return this.vehicleID.equals(otherCar.vehicleID);
	}
}
```

  - This will not compile! Why not?

- In Java, we have to @*Override* methods of the super class only with methods having the <u>same parameter types</u>!

```
// Somewhere in the JDK, the class Object is defined:
public class Object { // (declaration simplified)
	public boolean equals(Object other) {
		/* pass */
	}
}
```

<u>Notice:</u>
An @*Override* of *equals()* <u>must generally come together with an @*Override* of *hashCode()*</u>.
This topic will be discussed in the <u>"Collections and Algorithms"</u> training.

```
public class Car { // (other members omitted)
	@Override
	public boolean equals(Object other) {
		/* pass */
	}
}
```

```
public class Car { // (other members omitted)
	@Override
	public boolean equals(Car otherCar) {
		// Invalid: Method doesn't override
	}	// method from its super class
}
```

  - Among programmers we describe this restriction by stating that, "Java's @*Override*s have <u>invariant parameter types</u>".

Parameter Types in overridden Methods – Part 1

- But, why does it make sense to have invariant parameter types in @*Override*s?

- In short: the reason is compatibility. This code must effectively compile always for <u>all sub types of *Object*</u>:

```
void compareTest(Object left, Object right) {
	boolean areEqual = left.equals(right);
}
```

  – So, we can call *compareTest()* and pass two *Car* objects:

```
Car car1 = new Car("W0L000051T2123456");
Car car2 = new Car("WVWZZZ1JZ3W386752");
// Polymorphism in action: Car.equals(Object) is called, when we call left.equals(right):
boolean carsAreEqual = compareTest(car1, car2);
```

- If such an @*Override* would be legal:

```
public class Car { // (other members omitted)
	@Override
	public boolean equals(Car otherCar) {
		/* pass */
	}
}
```

  – this code can no longer call *Car.equals(Car)*:

```
void compareTest(Object left, Object right) {
	boolean areEqual = left.equals(right);
}
```

  – This is incompatible, because an *Object* cannot be passed to a *Car*, this is because an *Object* is more general than a *Car*!

Parameter Types in overridden Methods – Part 2

- Remember, that wherever an object of a more general type is awaited, an object of a more specialized type can be set:

```
Object car = new Car();   // Yes, we can assign a Car to an Object object. A Car is a substitute for an Object.
```

  - But not vice versa!

```
Car car = new Object();   // No, we cannot assign an Object to a Car object, because an Object is a too general type!
```

  - Therefore, this *@Override* cannot work:

```
public class Car { // (other members omitted)
	@Override
	public boolean equals(Car otherCar) {
		/* pass */
	}
}
```

  - because other parts of the program might try to pass all kinds of types derived from *Object*, matching the inherited signature.

- We have to *@Override Object.equals()* with the signature *equals(Object)* and keep the parameter type invariant.
  - But what if we don't pass a *Car* to *Car.equals(Object)* but something else inherited from *Object*?

```
@Override
public boolean equals(Object other) {
	Car otherCar = (Car)other;
	return this.vehicleID.equals(otherCar.vehicleID);
}
```

```
car.equals("a text"); // Throws ClassCastException
```

  - Of course it doesn't work (a *String* is no *Car*), but this one is a run time error. The cast from *other* to *Car* doesn't work.

Getting the Dynamic Type – The Class-Object of an Object

- All right! In order to understand more about dynamic types in Java, we'll learn how to get the type of an object.
  - Sorry? Sometimes we need to know the type of an object, esp. we want to know the dynamic type of an object.

- In Java, the type of an object is represented by an object of another special UDT: *Class*.
  - Yes, in Java a type is also represented by an object!
  - And: yes, Java provides a type, with the name *Class*!

```
// Somewhere in the JDK, the class Class is defined:
public class Class { // (declaration simplified)
}
```

- We can get the type of an object, by calling the method *getClass()*:

```
// Getting the Class object of the instance bus:
Bus bus = new Bus();
Class typeOfBus = bus.getClass();
System.out.println(typeOfBus);
// >class Bus
```

  - The method *getClass()* is inherited from the very super class *Object*, therefor every UDT provides the method *getClass()*.

```
// Object:
public class Object { // (declaration simplified)
	public Class getClass() { /* pass */ }
}
```

    - The method *getClass()* cannot be overridden in new UDTs!
  - The returned object is of type *Class*. *Class* is a so called meta-type: it is a type, which describes another type.

Getting the Static Type – The class-Literal of a Type

- Concerning polymorphism we've discussed objects' static and dynamic types.
  - In a former example we've used type flags to get information about dynamic types.
  - (But then we learned about overriding methods, which is the better alternative.)
- In Java we can directly get the dynamic type of an object of polymorphic type.
  - The dynamic type of an object can be retrieved with dynamic type checks:

```
Car car = new VintageCar(); // Let's car refer to a VintageCar.
System.out.println("car refers to a VintageCar: "+(VintageCar.class == car.getClass()));
// >car refers to a VintageCar: true

car = new Bus(); // Let's car refer to a Bus.
System.out.println("car refers to a Bus: "+(Bus.class == car.getClass())+"; car refers to a VintageCar: "+(VintageCar.class == car.getClass()));
// >car refers to a Bus: true; car refers to a VintageCar: false
```

Good to know:

```
// getClass() and class literals also work on array
// types and primitive types:
Class intClass = int.class;
Class intArrayClass = int[].class;
Class dynIntArrayClass = new int[]{1,2,3}.getClass();
// But not on primitive int objects:
Class dynIntClass = 3.getClass();
```

- Here we use a new syntax *Bus*.class, or more formal <*UDTName*>.class, which is called class literal.
  - *Bus*.class is the class literal of the UDT *Bus*.
  - The expression *Bus*.class evaluates to the *Class* object, representing the literal static type *Bus*.
  - The expression *car.getClass()* returns the *Class* object, representing *car*'s dynamic type.
  - Here we use class literals and the method *getClass()* to make the difference between static and dynamic type visible:
    - The reference *car* can point to an object of any subtype of *Car*. Here: *VintageCar*, then *Bus*.

```
Class typeOfBus = Bus.class;
Class dynTypeOfCar = car.getClass();
```


- Although we call these constructs "class literals", they can be used for interfaces and enums as well.
- Notice, that the result of class literals and the result of calling *getClass()* can be compared by reference! – I.e. both *Class* objects are identical!

Reimplementing Garage.tryToStartCar() with dynamic Type Checks

- It's possible to reimplement *Garage.tryToStartCar()* with dynamic type checks:

```
public class Garage {
    public void tryToStartCar(Car car) {
        if (VintageCar.class == car.getClass()) { // If car's dynamic type is VintageCar start it in a special way.
            VintageCar vintageCar = (VintageCar)car; // Downcasting!
            if (!vintageCar.hasStarter()) {
                vintageCar.crankUntilStarted();
            } else {
                vintageCar.startEngine();
            }
        } else {
            car.startEngine(); // Start other cars just by calling startEngine().
        }
    }
}
```

```
VintageCar fordTinLizzie = new VintageCar(1909);
// Ok! tryToStartCar() will pick the correct start-algorithm
// depending on the run time type!
joesStation.tryToStartCar(fordTinLizzie);
```

  - This implementation neglects the presence of overridden methods, but it uses dynamic type checks and down casts!
    - Mind, how this solution is virtually the same as the enum-solution. But in this case we are using the class literals as type flags.
    - We already learned that this is really bad: "pasta-object-orientation"!

- class literals and the result of *object.getClass()* can directly be compared by reference.
  - If this comparison evaluates to true, we still have to perform the down cast.

Overriding Object.equals() robustly with getClass()

- When we discussed overriding *Object.equals(Object)*, we encountered problems with run time errors:

```
@Override
public boolean equals(Object other) {
	Car otherCar = (Car)other; // This cast can fail.
	return this.vehicleID.equals(otherCar.vehicleID);
}
```

```
car.equals("a text");
```

  - The run time error occurs because of mismatching dynamic types: *equals()* awaits a *Car* "behind" *other*, but actually it is a *String*!
  - If the dynamic type behind *other* is not a *Car* (or sub type of *Car*) the "cast contact lens" fails with a *ClassCastException*!

- We can make *Car.equals(Object)* more robust by adding an extra type check:

```
public class Car { // (other members omitted)
	private final String vehicleID;

	@Override
	public boolean equals(Object other) {
		if (this.getClass() == other.getClass()) {
			Car otherCar = (Car)other;
			return this.vehicleID.equals(otherCar.vehicleID);
		}
		return false;
	}
}
```

Good to know:
Sometimes, often in books, you'll see usage of the operator instanceof (which we discuss in short) to compare dynamic types instead of *getClass()*, but this can lead to unexpected/wrong behavior of *equals()*. We'll not discuss the reason of this in this course, however it is related to the rules, which must be followed for a correct implementation of *equals()*.

  - We added a check to compare the *Class* of this and the passed parameter. If those don't match *equals()* evaluates to false.

- There are cases when using instanceof instead of *getClass()* is appropriate. E.g. when a proxy pattern is applied and instances of the proxy class must be treated as/compared like instances of the proxied class. This is the case if proxies are automatically generated, e.g. when the Spring Boot framework is used.

Overriding Object.equals() – the full Picture

- As mentioned in the "Good to know" box on the previous slide, overriding *Object.equals(Object)* must follow some rules.

- Without further ado, I'll present an *@Override* of *Object.equals(Object)*, that follows all rules:

```
public class Car { // (other members omitted)
    private final String vehicleID;

    @Override
    public boolean equals(Object other) {
        if (this == other) {
            return true;
        }
        if (null == other) {
            return false;
        }
        if (this.getClass() == other.getClass()) {
            Car otherCar = (Car)other;
            return this.vehicleID.equals(otherCar.vehicleID);
        }
        return false;
    }
}
```

checks for identity: identical objects are are equal the result is then true

checks for nullity (to avoid exceptions)

checks the dynamic type of this and the other object

the cast is type safe

the *vehicleID* fields of both objects are equality-compared

- Java also <u>requires</u> a type to *@Override Object.hashCode()*, if it *@Overrides Object.equals(Object)*.
  - We won't discuss this in this course.

Reimplementing Garage.tryToStartCar() with the instanceof Operator

- Instead of dynamic type checks with enums or *getClass()*/class literals, we can use the instanceof-operator:

```
public class Garage {
	public void tryToStartCar(Car car) {
		if (car instanceof VintageCar) { // If car's dynamic type is VintageCar start it in a special way.
			VintageCar vintageCar = (VintageCar)car; // Downcasting!
			if (!vintageCar.hasStarter()) {
				vintageCar.crankUntilStarted();
			} else {
				vintageCar.startEngine();
			}
		} else {
			car.startEngine(); // Start other cars just by calling startEngine().
		}
	}
}
```

- instanceof checks, whether the dynamic type of the left hand argument is equal to the UDTName on the right hand side.
  - If a dynamic type check evaluates to true, we still have to perform the down cast.
  - instanceof throws an NPE for null references.
- But using instanceof is basically the same as the class literal-else if-procedure, those are all dynamic type checks.
  - Better use overridden methods! -> Dynamic type checks should be avoided!

- There is the saying "polymorphism is always better than branching". In this example the usage of dynamic type checks (with class literals or instanceof) and if-else cascades would be the "branching". – This approach requires to add new dynamic types to be potentially handled in *Garage.tryToStartCar()* and this is a disapproving approach; always prefer polymorphism! – (Oo) Design patterns, which are often based on polymorphism, help to implement even complex problems without dynamic type checks.

Object-based Collections – Part 1

- Up the here, we mainly discussed *Object* being a very super class more or less as provider for common behavior.
  - We have discussed *Object.toString()* and *Object.equals()* we can *@Override* as we need it or leave the inherited behavior.
  - We can use *Object.getClass()* to get meta-information about the object in question.

- There is another view to the super class *Object*: it enables an ubiquitous substitution principle over all Java's types.
  - To understand this idea, we have to got back in time to Java version 1.4.

- In Java 1.4 the class definition of *ArrayList* looked like so:
  - It holds an *Object[]*.
  - Access- and modification methods accept/return *Object*s.

```
public class ArrayList { // very simplified definition
	private Object[] elementData; // the encapsulated array

	public boolean add(Object e) { /*pass*/ }
	public Object get(int index) { /*pass*/ }
}
```

- This is a genius idea! – It means that we can put any object into *ArrayList*.
  - Each type is inherited from *Object*, due to the substitution principle we can handle all thinkable objects:

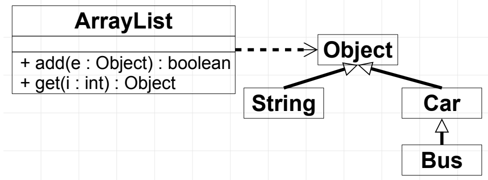
```
ArrayList busses = new ArrayList();
// Add elements:
busses.add(new Bus(150.0));
busses.add(new Bus(230.2));
// Get an element:
Bus firstBus = (Bus) busses.get(0);
```

```
ArrayList names = new ArrayList();
// Add elements:
names.add("Sally");
names.add("Peter");
// Get an element:
String firstName = (String) names.get(0);
```

Object-based Collections – Part 2

- Let's investigate the example storing *String*s into the *ArrayList* names. – Why does it work?

```
ArrayList names = new ArrayList();
// Add elements:
names.add("Sally");
names.add("Peter");
```



  - (1) It works because *ArrayList.add()* accepts *Object*
  - (2) and *String* is inherited from *Object*
  - (3) and due to the substitution principle we can pass a *String* to *ArrayList.add(Object)*.

- Getting an element from an *Object*-based collection looks differently, but bases on the same substitutability assumptions:
  - Obviously we have to do a down cast.

```
// Get an element:
String firstName = (String) names.get(0);
// firstName = "Sally"
```

  - (1) *ArrayList.get()* just returns an *Object*
  - (2) and because an *Object* can refer to any type, which inherits from *Object*, which is true for all objects of types in Java
  - (3) we have to cast the real dynamic type out of the *Object*-reference, which is a *String* in this case.
  - The crux on using an element stored in the *ArrayList* is, that we have to know the "real" dynamic type of the object in question.

Object-based Collections – Part 3

- However, when getting stored elements back from an *Object*-based collection, there are potential problems:

```
ArrayList busses = new ArrayList();
// Add elements:
busses.add(new Bus(150.0));
busses.add(new Bus(230.2));
```

  - Actually, we don't know the real dynamic type! – An *Object*-based collection could store any type of *Object*!
  - We easily provoke a problem:

```
// Get an element:
String firstName = (String) busses.get(0); // Bus cannot be cast to java.lang.String
```

    - The code postulates a *String* stored at the 1st slot in busses! – Sure, this is wrong, it only holds objects of type *Bus*.
    - The down cast fails and we end up with a *ClassCastException* at run time!

- Using *Object*-based collections is simple, the problem lies in handling dynamic types: type checks fail only at run time!

- Therefor, Java 5 added parameterized types to improve/replace *Object*-based code like *Object*-based collections.
  - Also *ArrayList* was upgraded to a parameterized type, namely *ArrayList<T>*.
  - The benefit over using *ArrayList<T>* instead of the *Object*-based *ArrayList* is, that the compiler helps us avoiding wrong typing:

```
ArrayList<Bus> busses = new ArrayList<>(); // Mind, that we have to specify the type stored in busses!
busses.add(new Bus(150.0));
busses.add(new Bus(230.2));
// Get an element (mind, that no explicit down cast is required):
String firstName = busses.get(0); // Invalid! Incompatible types: Bus cannot be converted to java.lang.String
```

Protected Family Secrets

- Let's review the type *VintageCar*:

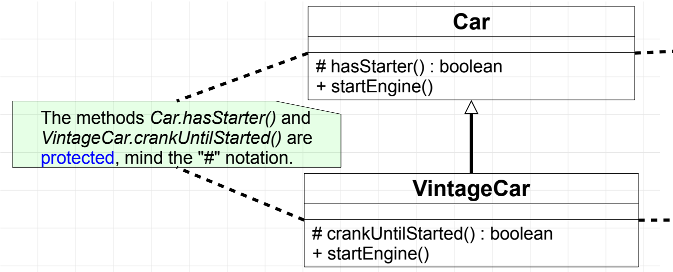
```java
public class VintageCar extends Car { // (members hidden)
	public boolean crankUntilStarted() { /* pass */ }
	public boolean hasStarter() { /* pass */ }
	public void startEngine() {
		if (!hasStarter()) {
			crankUntilStarted();
		} else {
			super.startEngine();
		}
	}
}
```

- There are some points to be thought of:
  - *VintageCar.hasStarter()* and *VintageCar.crankUntilStarted()* are only used in *VintageCar.startEngine()*.
  - *VintageCar.hasStarter()* could be defined in the super class *Car*, as each *Car* could have a starter.
  - *VintageCar.crankUntilStarted()* should be declared private, being encapsulated by *VintageCar*.

- But some problems arise with the proposed modifications:
  - *Car.hasStarter()* should not be visible to the public, but to its derived types (visible within *Car*'s "family").
  - Maybe *VintageCar.crankUntilStarted()* should be accessible by derived types also.
  - To solve these issues we can use Java's access specifier "protected".

Protected Methods

- Let's redesign *Car* and *VintageCar*:
  - Move *VintageCar.hasStarter()* to the type *Car* and mark this method as being protected.
  - Make *VintageCar.crankUntilStarted()* protected as well.
  - Then both methods are only visible in the "family", not to the public.



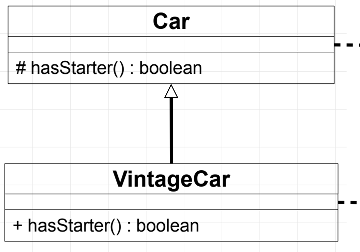

```java
public class Car { // (members hidden)
	protected boolean hasStarter() { /* pass */ }
	public void startEngine() { /* pass */ }
}
```

```java
public class VintageCar extends Car { // (members hidden)
	protected boolean crankUntilStarted() { /* pass */ }
	@Override
	public void startEngine() {
		if (!hasStarter()) {
			crankUntilStarted();
		} else {
			super.startEngine();
		}
	}
}
```

- In Java, the "family", i.e. valid accessor-UDTs of protected methods, are subtypes and other UDTs of the same package.
  - (The accessibility of protected members is strange to, e.g., C++ programmers, because it extends to UDTs in the package.)

Widening access Modifiers on Overrides

- When overriding methods, Java allows to <u>widen access modifiers</u>, e.g. from protected to public:




```
public class Car { // (members hidden)
	protected boolean hasStarter() { /* pass */ }
}
```

```
public class VintageCar extends Car { // (members hidden)
	@Override
	public boolean hasStarter() { /* pass */ }
}
```

- The opposite is not true, <u>an *@Override* cannot be more restrictive than the original</u>:

```
public class Car { // (members hidden)
	public void startEngine() { /* pass */ }
}
```

```
public class VintageCar extends Car { // (members hidden)
	@Override
	private void startEngine() { /* pass */ }
} // Invalid!  Attempting to assign weaker access privileges; was public
```

Intermezzo: Inheritance for white-box Reuse

- Aggregation: Using a *Car*'s public interface is black-box reuse.
  - All the critical aggregated stuff is encapsulated, encapsulated means being declared private or protected.
- Inheritance: Using a *Car*'s protected interface is white-box reuse.
  - Subtyping is generally needed to access the protected members.
  - Subtypes have to know how to work with protected methods!
  - And subtypes can also @*Override* protected methods!
  - Subtyping breaks encapsulation to certain degree!

Major Guideline:
Mainly use inheritance to express substitutability, not code reuse. – However, sometimes common code must be moved to an upper level in the hierarchy.

- Inherited/derived types have access to public and protected members of the super types.
  - protected members are only accessible within the "family".
    - E.g. accessing or handling the start-system of *Car*s (e.g. *hasStarter()*) is too critical to be public.
    - *Car's* subtypes must know how to use the start-system (e.g. the subtype *VintageCar* needed to handle the start-system in a different way w/ cranking).
- Another thing is, that using inheritance, common-ground code is not needed to be implemented again.
  - Inheritance could be used to express reuse, i.e. to reuse super class code.
  - This thinking can lead to weird ideas: if a method/code is used in more than one subclass -> move it to the super class.
  - Never ever use inheritance for plain reuse, this is an antipattern! – It is sometimes called the OOP DRY trap (Jeff Ward).

- In fact the protected members of our types should be as good documented (e.g. via (Javadoc) comments) as the public members to make white-box reuse possible!

Abstract Types – Part 1

- Now we're going to add a new method *drive()* to *Car*.
  - Of course, the idea of *drive()* is to let a *Car* drive, when it is called. *drive()* should be public, so everybody can call *drive()*:


```java
public class Car { // (members hidden)
	public void drive() {
		System.out.println("in Car.drive()");
	}
}
```

- Then its time to use oo principles and *@Override drive()* in sub classes of *Car*, so that this code works for all *Car*s:

```java
public void driveAway(Car car) {
	car.startEngine();
	car.drive();
}
```

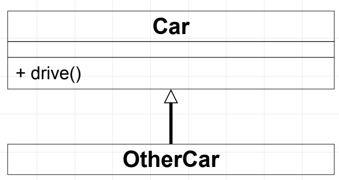
  - Consequently, every more special *Car*-class will or even must have a more special *@Override* of *Car*:

```java
public class Bus extends Car { // (members hidden)
	@Override
	public void drive() {
		setGear(1);
		releaseParkingBreak();
		releaseClutch();
	}
}
```

```java
public class VintageCar extends Car { // (members hidden)
	@Override
	public void drive() {
		setGear(1);
		removeChocks();
		releaseClutch();
	}
}
```

Abstract Types – Part 2

- The idea of *drive()* is of course very basic for *Car*s, so all new *Car* types must *@Override drive()* to do something useful!
  - If *drive()* is not overridden in a new *Car* type, the inherited implementation of *Car* (*Car.drive()*) will be effective:




```java
public class Car { // (members hidden)
	public void drive() {
		System.out.println("in Car.drive()");
	}
}
```

```java
public class OtherCar extends Car {
	// pass
}
```

```java
public void driveAway(Car car) {
	car.startEngine();
	car.drive();
}
```

```java
OtherCar car = new OtherCar();
driveAway(car);
// >in Car.drive()
```

  - But is that what we want? This is not very useful, *OtherCar* drives/behaves like *Car*, but it should define its own way to drive!
  - A matter of fact: *OtherCar* should *@Override drive()*!

- But how can we force developers to *@Override drive()*, when writing new sub classes of *Car*?

Abstract Types – Part 3

- We force @*Override*s of certain methods in inheriting classes by:
  - (1) Marking the respective method abstract in the super class and remove its implementation.
  - (2) Marking the respective enclosing super class as abstract class.

```
public abstract class Car { // (members hidden)
        public abstract void drive();
}
```

```
public class OtherCar extends Car {
        // pass
}
// Invalid! Error: java: OtherCar is not abstract and
// does not override abstract method drive() in Car
```

- *Car.drive()* is now an abstract method.
  - An abstract method doesn't have a body, i.e. it has no implementation.
  - If a class defines at least one abstract method, the class itself becomes an abstract class and must be declared abstract.

- What is the idea behind abstract methods and classes? When another class, such like *OtherCar*, extends *Car*:
  - (1) It can either @*Override* each inherited abstract method with a "meaningful" implementation to become a concrete class,
  - (2) or it @*Override*s only some or none abstract methods to also become an abstract class.

- Preliminary meaning of abstract: a "meaningful" sub class must @*Override* all abstract methods!

Abstract Types – Part 4

- But, what does that mean, "a meaningful sub class must *@Override* an abstract method"?

```
public abstract class Car { // (members hidden)
	public abstract void drive();
}
```

- A new class can *@Override* all inherited abstract methods with a meaningful implementation to become a concrete class.
  - Let's keep in mind, that we wanted to force programmers to code sub classes of *Car* to *@Override drive()* properly.
  - In Java, we can express this the "oo-way": *Car* is a class, that describes an abstract concept, by leaving away some definitions.
  - By "downgrading" *Car* to an abstract class, so by leaving away details, we force sub classes to fill, what the *Car*-concept left away.

```
public class OtherCar extends Car {
	@Override
	public void drive() {
		System.out.println("in OtherCar.drive()");
	}
}
```

```
public void driveAway(Car car) {
	car.startEngine();
	car.drive();
}
```

```
OtherCar car = new OtherCar();
driveAway(car);
// >in OtherCar.drive()
```

Good to know:
Abstract from latin *abstrahere* – to remove something

- After overriding all inherited abstract methods, *OtherCar* is a concrete class.

Abstract Types – Part 5

- What, if an inheriting class does not *@Override* all abstract methods of an abstract class?

```
public abstract class Car { // (members hidden)
	public abstract void drive();
}
```

- In this case, the inheriting class doesn't completely fill all the details we have left away in the abstract class.
  - When an inheriting class doesn't fill all details, it becomes itself an abstract class!
  - Then also the inheriting class must be defined as abstract class, in order be compiled successfully:

```
public abstract class OtherCar extends Car {
	// pass
}
```

- Defining a class as abstract has another consequence: we cannot create instances of abstract classes:

```
OtherCar otherCar = new OtherCar(); // Invalid! Error java: OtherCar is abstract; cannot be instantiated
```

  - In oo-terms we can say: "*OtherCar* is too abstract to have own instances"
  - If all abstract methods are overridden in *OtherClass* it becomes a concrete class, of which we can create instances:

Good to know:
Abstract means to leave things away. Perfection doesn't mean, that we can no longer add things, it means that we can no longer leave things away.

```
public class OtherCar extends Car {
	@Override
	public void drive() { // OtherCar is concrete now
		System.out.println("in OtherCar.drive()");
	}
}
```

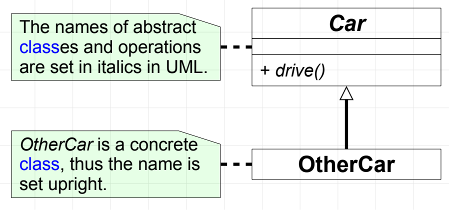
```
OtherCar otherCar = new OtherCar(); // Ok!
```

Abstract Types – UML Representation

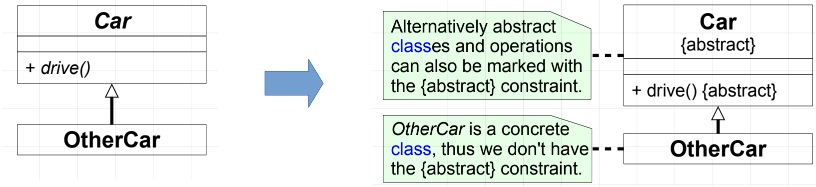
- In UML class diagrams the names of abstract classes and operations are set in italics, concrete ones are set upright.

```java
public abstract class Car { // (members hidden)
	public abstract void drive();
}
```

```java
public class OtherCar extends Car {
	@Override
	public void drive() {
		System.out.println("in OtherCar.drive()");
	}
}
```



- Alternatively, abstract classes and operations can be marked with the {abstract} constraint instead of setting text in italics:



  - Using {abstract} is practical when drawing class diagrams by hand, where setting text in italics is "challenging".

Implementation and Usage of abstract Types – Part 1

- So, abstract classes seem to be very limited in use, we cannot even create instances.

- What is the sense of abstract classes?
  - Up to now, we used them to force a derived class to @*Override* special methods to be a concrete class.
  - => And we can only instantiate concrete classes!

- More important are abstract classes when used for polymorphism:

```java
public void driveAway(Car car) {
	car.startEngine();
	car.drive();
}
```

```java
OtherCar otherCar = new OtherCar();
driveAway(otherCar);
// >in OtherCar.drive()
```

```java
Bus bus = new Bus();
driveAway(bus);
// >in Bus.drive()
```

  - We can pass any type to *driveAway()*, that is of type *Car*, i.e. each type, that is more specific than *Car*, each type, that is a *Car*.
  - In *driveAway()* we can safely call *car.drive()*, because Java guarantees, that we can only create objects of more specific *Car* type.

Implementation and Usage of abstract Types – Part 2

- abstract classes can only be used in four ways basically.

- (1) As super type of an inheriting class

```
public class OtherCar extends Car {
	// pass
}
```

- (2) On the left side of an assignment

```
Car car = new OtherCar();
driveAway(car);
// >in OtherCar.drive()
```

- (3) As parameter type

```
public void driveAway(Car car) {
	// pass
}
```

- (4) As return type

```
public Car pullOut() {
	// pass
}
```

- What can we conclude from these findings?
  - abstract classes can only be used as static types, thus there can be no instances (which would be created at run time).
  - The "type-character" of an abstract class is crucial, not the instance.
  - Java's concept of interfaces puts this to the next level, interfaces even get rid of fields and only represent concepts.

POJOs – Part 1

- After we have introduced many advanced things we can do with classes, we should take a look back.
  - We do this to introduce another Java-concept retrospectively, which underscores how useful simple classes can be.
- There exists a certain category of UDTs that only encapsulate fields via public getters/setters, e.g. like *Person*:

```java
// <Person.java>
public class Person {
    private String firstName;
    private String lastName;
    private int age;

    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) { this.firstName = firstName; }
    public String getLastName() { return lastName; }
    public void setLastName(String lastName) { this.lastName = lastName; }
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
}
```

  - Such a class is called Plain Old Java Object or "POJO".
- Ideally, a POJO only follows the Java Language Specification. It should not have to
  - extend classes different from *Object*,
  - implement interfaces (a concept we have yet to discuss)
  - and not to contain annotations (another concept we have yet to discuss).

POJOs – Part 2

- POJOs offer low coupling and high cohesion, "the ideal oo type": it is a highly reusable, maintainable and testable class.

- But, what does essentially remain in a POJO?
  - In the narrow sense, a POJO only has private fields, which have public getters/setters each:

```java
// <Person.java>
public class Person {
    private String firstName;
    private String lastName;
    private int age;

    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) { this.firstName = firstName; }
    public String getLastName() { return lastName; }
    public void setLastName(String lastName) { this.lastName = lastName; }
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
}
```

Good to know:
A private field together with its public getter/setter-pair is often called property. The term property is esp. used with the Java Beans concept.

  - *Person* is an extreme, for POJOs also offering methods beyond getters/setters is OK, also *Object*'s methods can be overridden.

- POJOs are the basis of another important Java concept: Java Beans.
  - Java Beans is Java's component technology for reusability beyond our own programs.
  - We will discuss Java Beans in a future lecture.

Why is Abstraction needed?

- OO programmers design models that simulate reality.
  - Abstraction means to leave out irrelevant details from the model.
  - Some degree of detail is sufficient. Capturing "all" details is impossible (computer memory, time and expert knowledge is limited).
  - OO seems to be appropriate for big projects to get rid of the "academic touch".

- A vendor's framework only abstracts a certain (the vendor's) view of a model.
  - Multiple abstraction solutions can be correct for a certain task.

- Delegation, information hiding, encapsulation and substitution help to abstract.
  - These concepts do also help to postpone or defer details.

- A core idea of oo development is to accept types as incomplete.
  - These types could be extended incrementally and iteratively.
    - This is called incremental and iterative software engineering.

- What does "incremental and iterative" mean?
  - Incremental: the engineering will be performed in steps.
  - Iterative: some parts of the engineering process will be repeated to improve these aspects.

Final Methods – Part 1

- Remember, when we added the field *licencePlateID* into the class *Car*, including a getter/setter pair:

```
public class Car { // (other members omitted)
	private String licencePlateID;

	public void setLicencePlateID(String licencePlateID) {
		this.licencePlateID = licencePlateID;
	}
	public String getLicencePlateID() {
		return licencePlateID;
	}
}
```

- Java allows to override the getter and the setter to show a different behavior for derived classes:

```
public class SuspiciousCar extends Car {

	@Override
	public void setLicencePlateID(String licencePlateID) { /* Do nothing! */ }
	@Override
	public String getLicencePlateID() { return "FAKE"; }
}
```

```
Car car = new SuspiciousCar();
car.setLicencePlateID("KUS-TT 295");
String licencePlateID = car.getLicencePlateID();
// licencePlateID = "FAKE"
```

  - What we have done here is very interesting! We have effectively deactivated *licencePlateID's* getter/setter in *SuspiciousCar*!
  - The setter just ignores the parameter and doesn't set anything, the getter always returns the *String* "FAKE"!

Final Methods – Part 2

- However, the general idea of the *licencePlateID* is very important, it should not be overridden by derived classes!

- Java provides a special feature to forbid a sub class to *@Override* specific methods. We can declare them as final methods:

```
public class Car { // (other members omitted)
	private String licencePlateID;

	public final void setLicencePlateID(String licencePlateID) {
		this.licencePlateID = licencePlateID;
	}
	public final String getLicencePlateID() {
		return licencePlateID;
	}
}
```

Good to know:
So, Java's keyword final can be used to define constants and to declare methods being non-overridable. When the same keyword can be used with different meanings in different places, we call it a contextual keyword.

- Having declared *setLicencePlateID()* and *getLicencePlateID()* as final methods, they can no longer be overridden:

```
public class SuspiciousCar extends Car {

	@Override
	public void setLicencePlateID(String licencePlateID) { /* Do nothing! */ }
	// Invalid! Error: java: setLicencePlateID(String) in SuspiciousCar cannot override setLicencePlateID(String) in Car overridden method is final
	@Override
	public String getLicencePlateID() { return "FAKE"; }
	// Invalid! Error: java: getLicencePlateID(String) in SuspiciousCar cannot override getLicencePlateID() in Car overridden method is final
}
```


- An important example of a final method is *getClass()*, that is defined in *Object*.

Final Classes

- When we discussed final methods, we understood, that overriding methods can be unwanted or even dangerous.
  - => Consider that overriding methods means, that we can deactivate functionality of a super class in a derived class.
  - Because of substitutability we can pass an object of a derived class, were an object of the super class is awaited, we can trick other algorithms to use our override instead of the super classes method.
  - To avoid this, we can use final methods to forbid overriding of individual methods.
- Java also allows to forbid sub-classing of a class completely. In this case the class cannot be used as super class.
  - And this also avoids any substitutability to take place!
- We disallow sub classing a class in Java by declaring a class as final class. E.g. Java's class *String* is final:

```java
// Somewhere in the JDK, the class String is defined:
public final class String {
	// pass
}
```

```java
// Invalid! Error: java: cannot inherit from final String
public class MyString extends String {
	// pass
}
```

Good to know:
Indeed, Java's contextual keyword final can even be used in three ways (or contexts)! It can be used to define constants and to declare methods being non-overridable and to forbid a class to be inherited at all.

  - Currently, the UML doesn't offer a notation for final classes and methods.
- Remember the discussion about the type *String* in a previous lecture. We said *String* is highly optimized.
  - That is possible, because *String* efficiently and firmly encapsulates its internal management.
  - But this comes with a prize: we cannot inherit new types from *String*, the class *String* is a final class.

- The ability to inherit *String*, so that instances of such a new class could be passed wherever a *String* is awaited, would open the door to disaster. E.g. we could inherit *String* and *@Override length()* so that calling *length()* sends a mail with the *String*'s content to someone else… Just mind, how often *length()* will be called in other code.
- enums are implicitly final.
- In a sense, final is the opposite of abstract: final -> mustn't inherit/*@Override* versus abstract -> must inherit/*@Override*.
- Usually own UDTs should not be declared final! – It avoids the application of the open close principle (OCP)!
  - final makes sense to be applied for very fundamental, non-substitutable UDTs, esp. for security reasons, such as *String* or *Class*.

The SOLID Principle

- Robert Cecil Martin ("Uncle Bob") et al. collected five principles for oo design.
  - These principles guide us through the design of "good" oo code.
  - Good code: not rigid, a change does not effect other parts of the program (-> not fragile), the structure clarifies what the code does, not the debugging activity.
- Single Responsibility Principle (SRP)
- Open-Close Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

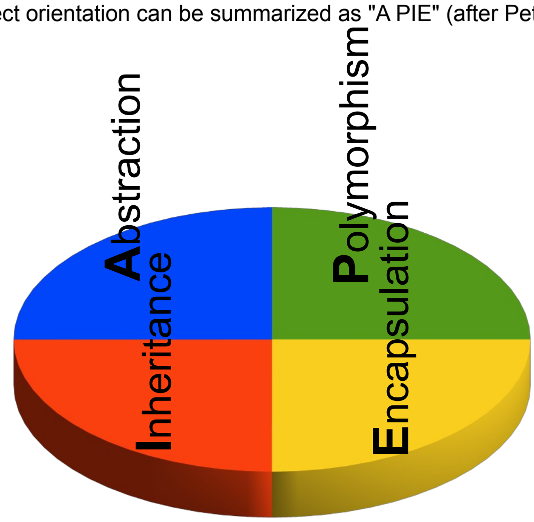
- SRP: Types should have only one reason to change. This requirement also applies to methods.
- OCP: Open types for extension, but close them for modification, i.e. extend them w/o modification of the core code, rather code will be added (e.g. via polymorphism). If we need to modify a type's interface, the calling code needs to be modified (ripple effect), which is expensive. Separate things that change, from others that don't (i.e. find the vector of change to drive design patterns). This principle also implies, that we should rather avoid final classes, unless we have a very good reason to "close" classes.
- LSP: Exploit sub typing to put the substitution principle into effect. Realization: a class can be exchanged with another class, as long as it provides the same functionalities, this a kind of backward-compatibility. Inheritance establishes class-relations, which guarantees backward-compatibility implicitly. Sometimes types are not related, but still need to be backward compatible. A subtype must not restrict the super type, therefor square shouldn't be a subtype of rectangle: square restricts free setting of the sides a and b!
- ISP: Use small type interfaces to which callers depend on, so that changes in other type interfaces don't bother them. Smaller interfaces are more stable than bigger ones.
- DIP: Let types only depend on abstract types (the contracts), never on concrete types (the implementation) to reduce coupling. Layers of lower abstraction (more concrete type) depend on layers of higher abstraction (more abstract types), never vice versa. Interface/abstract types are sometimes called "handle types", concrete implementations are sometimes called "body types".
  - Why is is called "inversion"? The idea is that the actual control flow at run time accesses code in the "body type" through the more abstract "handle type", but the dependency is directed into the inverse direction of the control flow, namely from the "body type" to the "handle type" and the "body type" can be varied.
- The SOLID principle enumerates a set of guidelines, not laws.
- Additionally, there exists the principle of Command Query Separation (CQS) formulated by Bertrand Meyer: Do getters only query values, but don't change state? Do setters not return values?

So, what is oo Programming all about?

- The reality can be simulated to a good degree with object oriented types!
  - Esp. many real associations can be expressed with aggregation and specialization.
  - Even customers can understand, how object oriented types work (UML)!
- Thinking in type interfaces rather than in algorithms and procedures:
  - Interface and implementation can be separated, but types are complete.
  - First define all interfaces, then implement the type completely.
- Instead of algorithms with static methods, we can build architectures with UDTs.
  - Recurring UDTs architectures are called design patterns.
  - There exist simple and complex design patterns.
  - Many design patterns were/are integrated as first-class features in present/upcoming (oo) programming languages.
- Successful (oo) programmers identify and use design patterns (even if they don't know they are design patterns).

APIE

- The core principles of object orientation can be summarized as "A PIE" (after Peter van der Linden):



Thank you!