

HBV401G SOFTWARE DEVELOPMENT

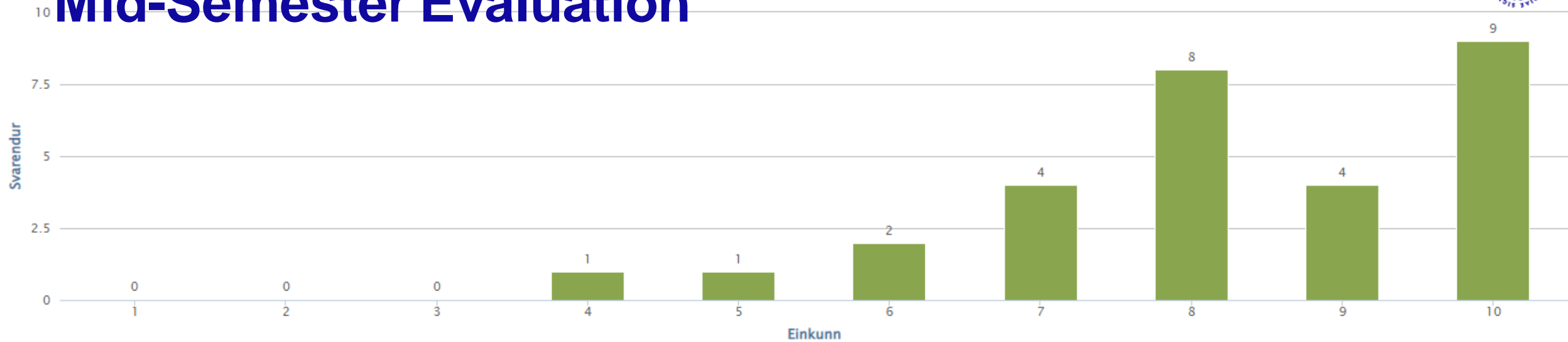
7. Generalization & Specialization

Matthias Book
Spring 2022

**FACULTY OF INDUSTRIAL ENGINEERING, MECHANICAL
ENGINEERING AND COMPUTER SCIENCE**

Mid-Semester Evaluation

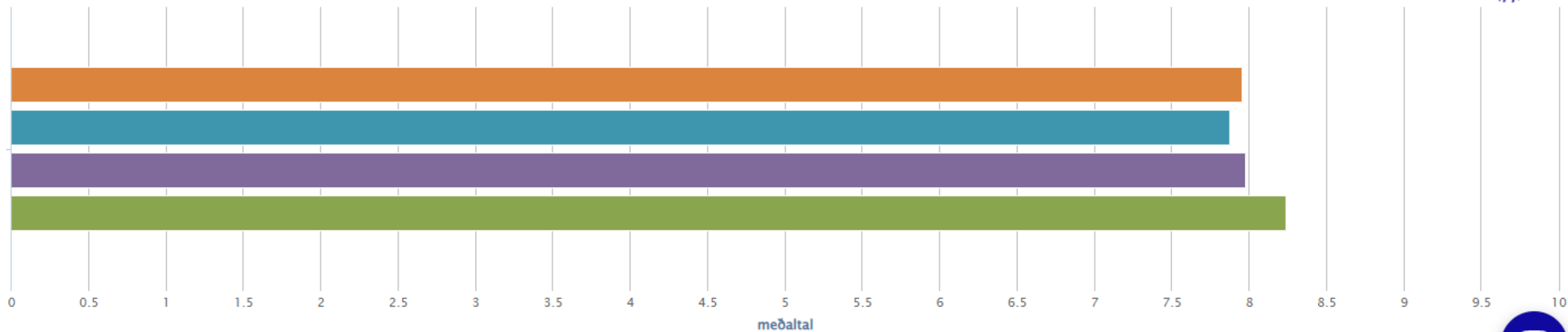
Gefðu námskeiðinu einkunn:
Dreifing einkunnna



HBV401G – Þróun hugbúnaðar

Takk fyrir! 😊

Please grade the course:
Samanburður



Participation: 29/112

HBV401G Iðnaðarverkfræði-, vélaverkfræði- og tölvunarfræðideild Verkfræði- og náttúruvísindasvið Háskóli Íslands



What people like

- Good course organization
- Interesting course material
- Remote teaching format
- Teaching staff's commitment

What could be improved

- More practical coding instructions
- Uncertainty about programming structure, complexity, effort
 - This is intentional – real-world challenge
 - Primary learning goal of this course is to design your own software
 - Programming finesse is secondary
 - Start exploring technical solutions early
 - Talk to teams in your cluster early
 - Ask your tutors for advice
 - Use Discord to ask technical questions
- 3x40 min. online lectures are too long

In-Class Quiz #6 Solution: Object Orientation

- Fill in the blanks with the words (A)tttributes, (C)lass, (I)mplementations, (M)ethods, (O)bjects, or (V)alues:
 - a) In the OO paradigm, we design a system as a set of collaborating **OBJECTS**.
 - b) A **CLASS** defines the common characteristics of a set of **OBJECTS**.
 - c) A **CLASS** is defined at a system's design time, while **OBJECTS** are created and destroyed at run time of the system.
 - d) An object's state is defined by the **VALUES** of its **ATTRIBUTES**, while its behavior is defined by the **IMPLEMENTATIONS** of its **METHODS**.
 - e) The **ATTRIBUTES** of different objects of the same class can have different **VALUES**, but their **METHODS** have the same **IMPLEMENTATIONS**.
 - f) **ATTRIBUTES** should only be accessed by **METHODS** of the same **CLASS**.

Recap: Team Assignment #2: OO Domain Model

- Create a domain model that defines:
 - The application-domain concepts that your system will need to handle (→ classes)
 - The properties that are characterizing instances of those classes (→ attributes)
 - The operations that can be performed on instances of those classes (→ methods)
 - The association, aggregation or composition relationships between those classes
- By **Sun 27 Feb**, submit in Canvas (as a PDF document):
 - A **UML class diagram** showing your project's **domain model**:
 - All classes that describe your team's application domain
 - including their attributes and methods (no need for data types)
 - Relationships (association, aggregation, composition) between classes
 - with multiplicities where appropriate
 - **Note: Associations may be undirected (without an arrow tip) in a domain model**
- On **Wed 2 Mar**, present and **explain** your model to your tutor:
 - Why did you structure the classes, their attributes and associations the way you did?
 - How will your system work with these classes via their methods?

Recommended simple and free drawing tool:
<http://draw.io>

Recap: Team Assignment #2: OO Domain Model

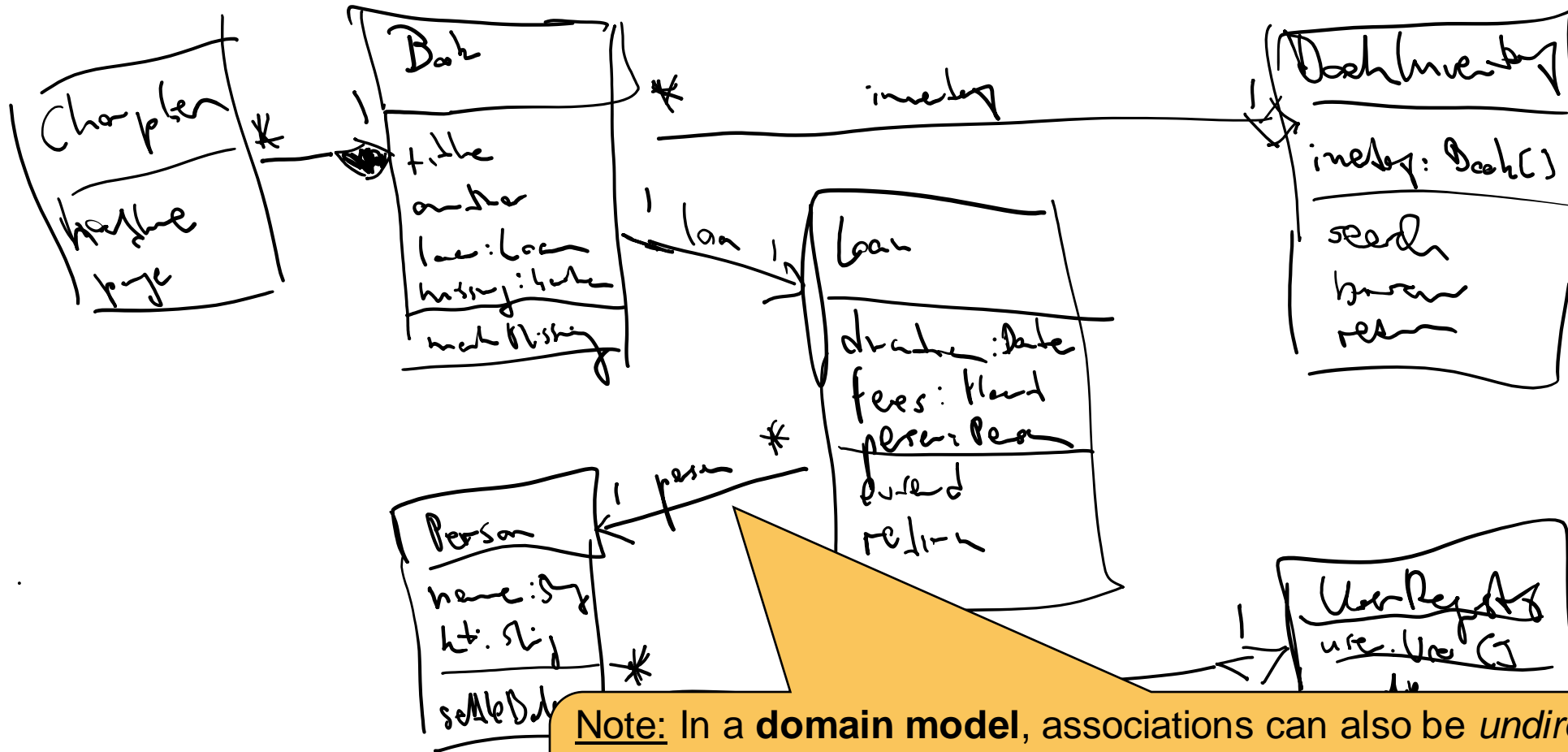
■ Grading criteria

- Domain model is a syntactically correct UML class diagram (10%)
- Domain model is complete (esp. with regard to classes and attributes) (45%)
- Relationships between classes, and placement of attributes and methods in classes, are plausible (45%)

■ Deadlines (**mandatory for assignment grade** / **optional for bonus grade**)

- by **Wed 23 Feb 12:00**, submit your anonymized draft to the Peer Feedback Assignment #2
 - the earlier you submit, the more time your reviewers have to give you feedback
- on **Wed 23 Feb 15:00-18:15**, discuss your draft with your tutor
- by **Fri 25 Feb 23:59**, submit your peer feedback on the drafts assigned to you
 - the earlier you submit, the more time the other students have to incorporate your feedback
- by **Sun 27 Feb 23:59**, submit your final PDF document to the Team Assignment #2
- by **Wed 2 Mar 12:00**, rate the quality / usefulness of the peer feedback you received
- on **Wed 2 Mar 15:00-18:15**, present and explain your model to your tutor

Recap: Example Domain Model of a Library



Note: In a **domain model**, associations can also be *undirected* (without arrow)

- If you just want to specify that two classes are associated...
- ...but not which one is maintaining a reference to the other.

In a **design model**, associations should always be directed (with arrow)

Generalization & Specialization

see also:

Learning UML 2.0, Chapters 4 and 5



Recap: Classes and Objects

■ Class

- A definition of the common characteristics of a set of objects, i.e.
 - the types of its attributes
 - the implementations of its methods
- In software: Defined at design-time



■ Object

- An individual instance of a class in which all of the class' characteristics manifest themselves
- In software: Created and destroyed at run-time

- **A class is a “blueprint” for the creation of similar objects.**



What if There Are Different Types of Vehicles?

- Some of their attributes are shared (generic), e.g.
 - registration, wheel size, ...
- Some of their attributes are differing (specific), e.g.
 - for fuel trucks:
tank volume, number of axles
 - for cooler trucks:
current temperature, number of axles
 - for jeeps:
number of doors, fording depth
 - for convertibles:
type of top, number of doors
- Some of their methods are shared (generic), e.g.
 - accelerate, decelerate, ...
- Some of their methods are differing (specific), e.g.
 - for fuel trucks:
load, unload
 - for cooler trucks:
load, unload, cool
 - for jeeps:
lockDifferential, unlockDifferential
 - for convertibles:
openTop, closeTop

Generalization Example

- Specifying all these attributes and operations in individual classes would create lots of specification, implementation and maintenance redundancy.
 - Imagine fixing the same bug in all four acceleration methods...
 - ...or adding a new method to all four classes.

FuelTruck
registration wheelSize tankVolume numAxles
accelerate decelerate load unload

CoolerTruck
registration wheelSize currTemp numAxles
accelerate decelerate load unload cool

Jeep
registration wheelSize numDoors fordingDepth
accelerate decelerate lockDiffGear unlockDiffGear

Convertible
registration wheelSize numDoors topType
accelerate decelerate openTop closeTop

Generalization Example

MotorVehicle
registration wheelSize
accelerate decelerate

FuelTruck
tankVolume numAxles
load unload

CoolerTruck
currTemp numAxles
load unload cool

Jeep
numDoors fordingDepth
lockDiffGear unlockDiffGear

Convertible
numDoors topType
openTop closeTop

- Some attributes and methods can be generalized to any type of motor vehicle...

Generalization Example

Truck
numAxles
load unload

MotorVehicle
registration wheelSize
accelerate decelerate

Car
numDoors

FuelTruck
tankVolume

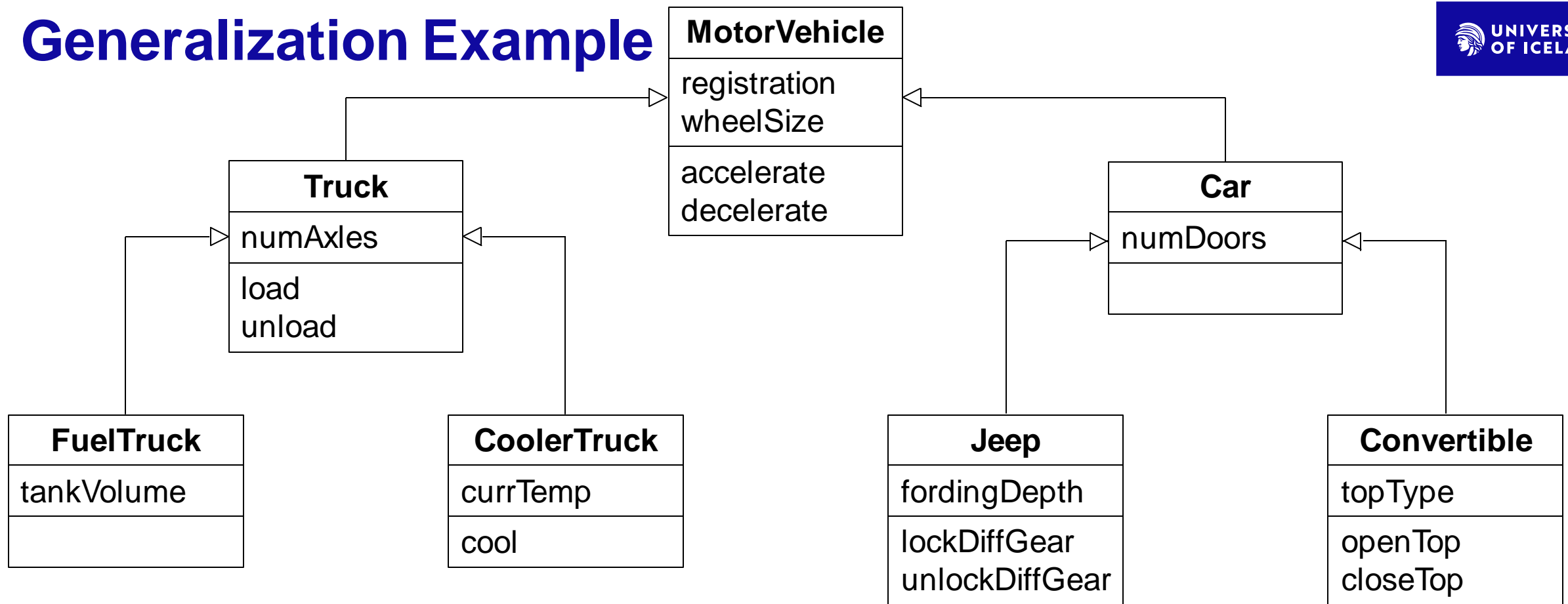
CoolerTruck
currTemp
cool

Jeep
fordingDepth
lockDiffGear unlockDiffGear

Convertible
topType
openTop closeTop

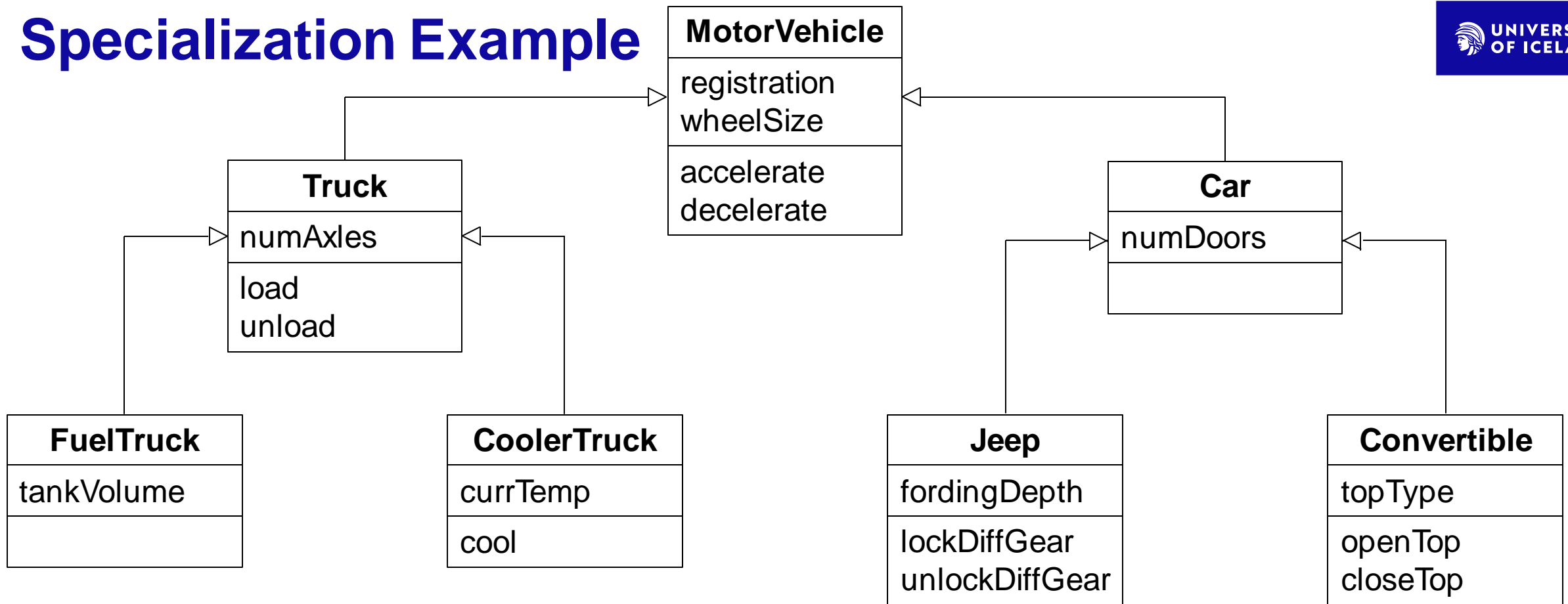
- ...and some to specific types (classes) of motor vehicles
 - which are still more generic than fuel and cooler trucks, jeeps and convertibles

Generalization Example

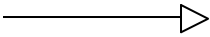


- A car is a motor vehicle, so it inherits all characteristics of a motor vehicle.
- A jeep is a car, so it inherits all characteristics of a car (and thus a motor vehicle)

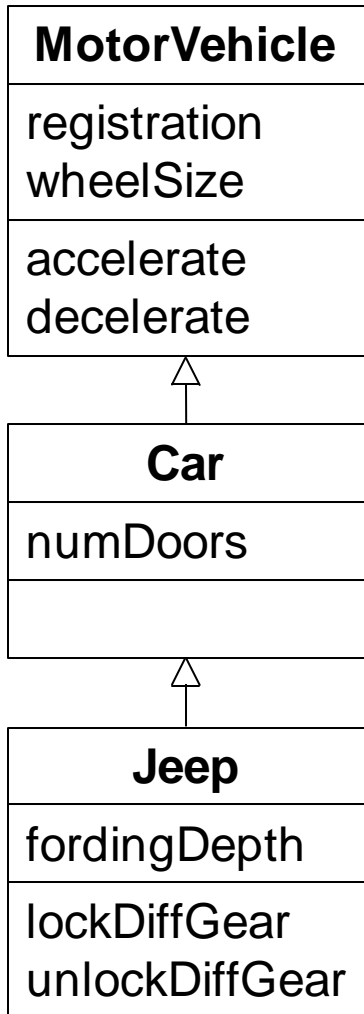
Specialization Example



- A cooler truck extends a truck's characteristics with the ability to cool its freight.
- A jeep extends a car's characteristics with the ability to lock its differential gear.

- Generalization is a relationship between a superclass and a subclass
- Specified in UML as an arrow with a hollow closed tip pointing to the superclass
 - Read: “is a” or “inherits from” or “extends” 
- “is a” relationship: An object of the subclass is also of the type of the superclass
 - e.g. “a jeep is a car”
 - Transitive relationship: “A car is a motor vehicle; a jeep is a car; so a jeep is a motor vehicle.”
- Subclasses inherit attributes and methods from superclasses...
 - Nice: This is not just a conceptual convention in object-oriented modeling notations, but automatically occurs in the execution of object-oriented programming languages!
- ...and can extend/change the superclass’ capabilities with new implementations

Generalization and Specialization in Java

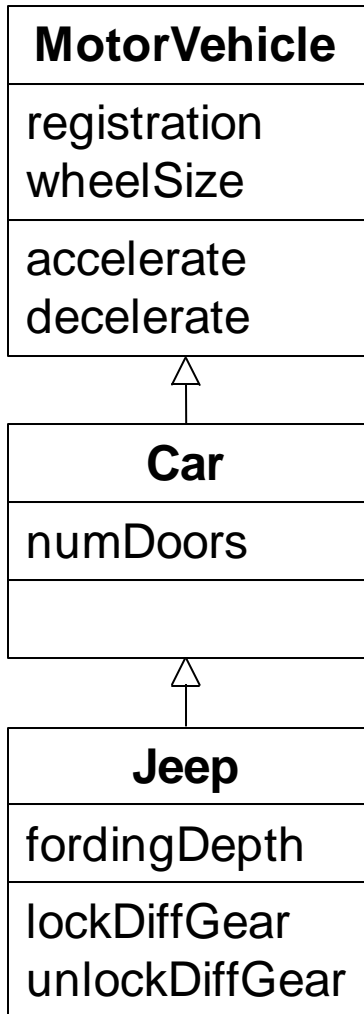


```
public class MotorVehicle {
    public void accelerate(float kmh) {...}
    public void decelerate(float kmh) {...}
}
```

```
public class Car extends MotorVehicle {
    private int numDoors;
}
```

```
public class Jeep extends Car {
    private int fordingDepth;
    public void lockDiffGear() {...}
    public void unlockDiffGear() {...}
}
```

Generalization and Specialization in Java



```
public class TestDriver {

    public static void main(String[] args) {
        Jeep cherokee = new Jeep();
        cherokee.lockDiffGear();
        cherokee.accelerate(50);
        crashTest(cherokee);
    }

    public void crashTest(Car testCar) {
        testCar.decelerate(100);
        testCar.unlockDiffGear();
    }
}
```

This works:
testCar is a Car,
which is a MotorVehicle,
so it “knows” how to decelerate,
and cherokee is a Jeep,
which is also a Car

This doesn't work:
testCar is a general Car, which is not a Jeep
and doesn't “know” a Jeep's methods,
even when it is “unknowingly” passed
a reference to a Jeep object

Benefits of Generalization / Specialization

- Avoiding redundant code
- Ensuring code changes are reflected in all subclasses
- Reducing model and software complexity through multi-level abstraction
- Expressing structural relationships of domain objects in code, and leveraging those relationships in program structures
- High flexibility and extensibility through “type-agnostic programming”:
Instances of subclasses can be treated just like instances of superclasses!

Break



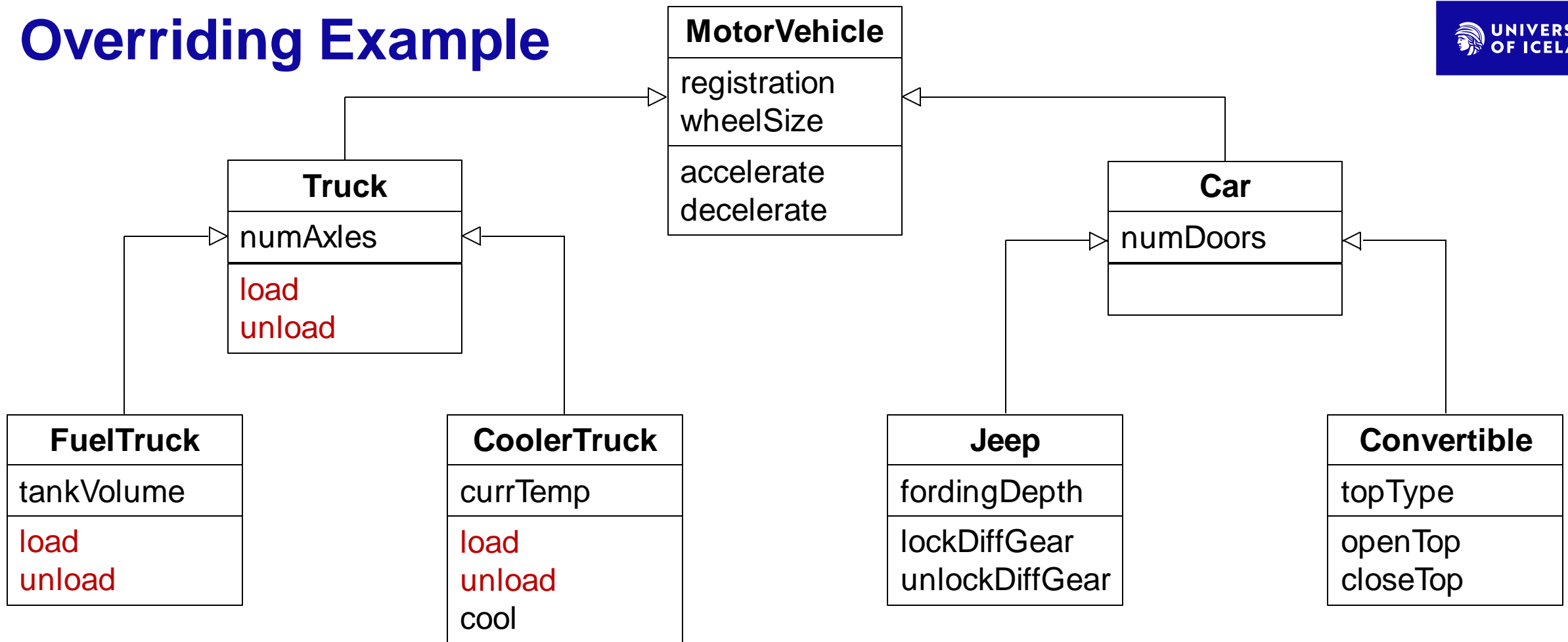
Overriding Methods, Abstract Classes

see also:

Learning UML 2.0, Chapters 4 and 5



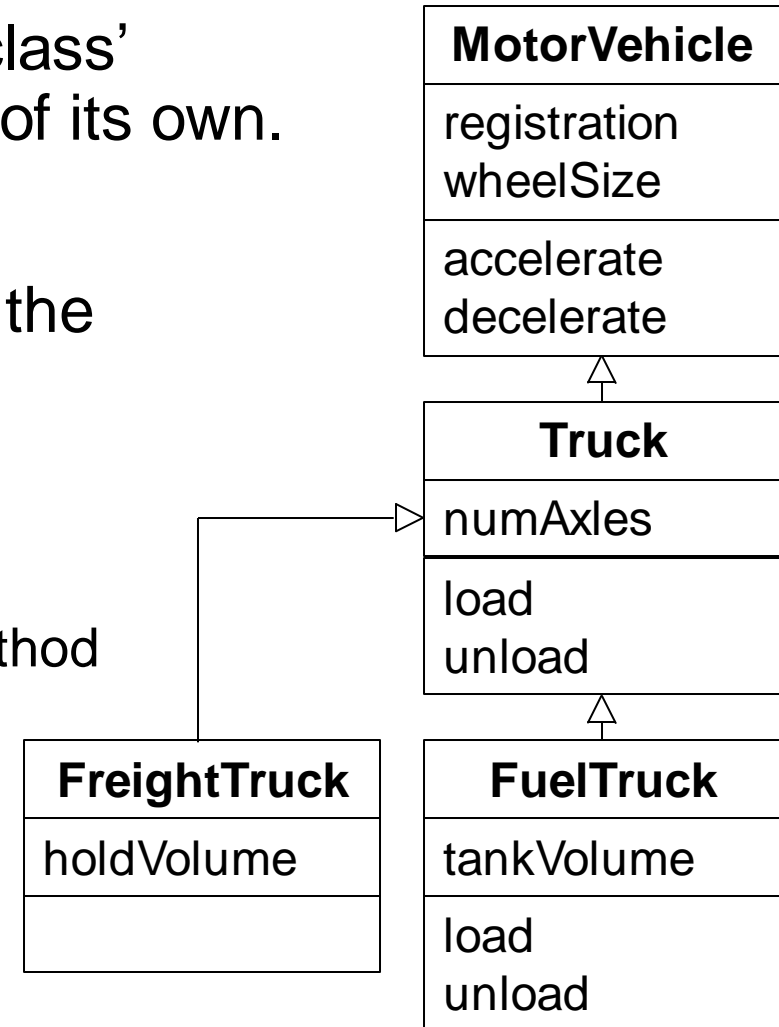
Overriding Example



- General Trucks have general loading and unloading procedures.
- FuelTrucks and CoolerTrucks have different, specific (un)loading procedures that **override** the general procedures.

Overriding Methods

- A subclass can override the implementation of a superclass' general method by providing a specific implementation of its own.
- Instantiated objects will then use the implementation of the "closest" class.
- Examples:
 - A FreightTruck instance will use MotorVehicle's accelerate method
 - A FreightTruck instance will use Truck's load method
 - A FuelTruck instance will use its own load method
 - A Truck instance will use its own load method



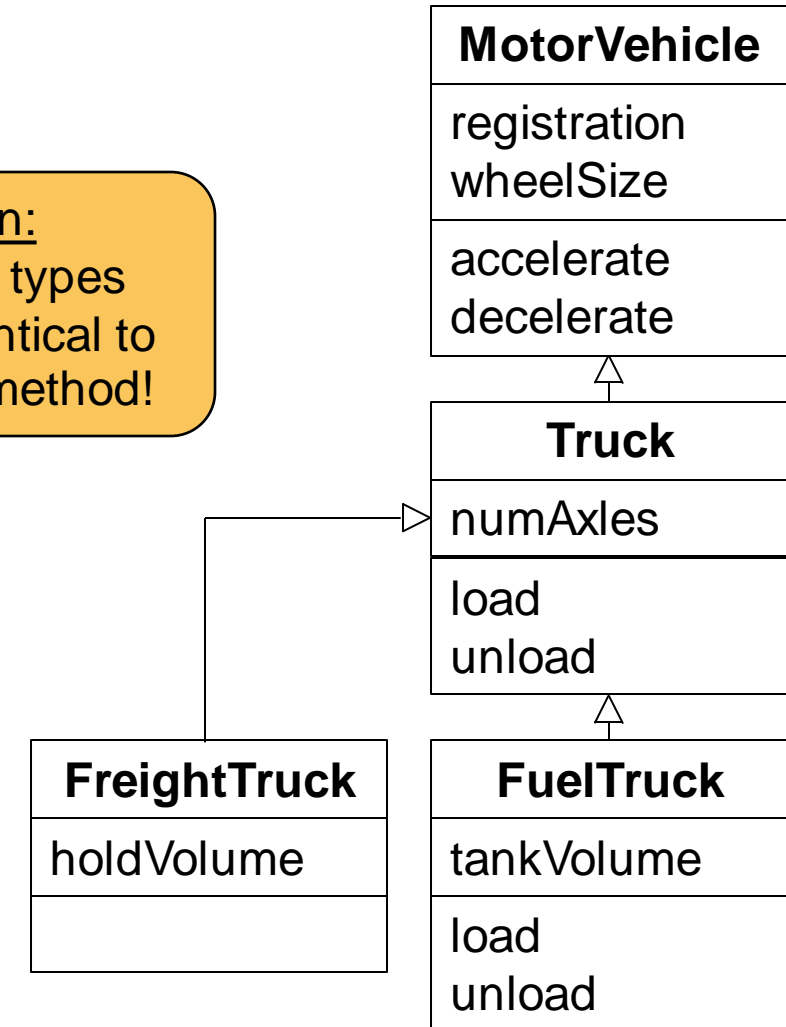
Overriding in Java

```
public class Truck {  
    public void load(Cargo c) {  
        /* general implementation */  
    }  
    ...  
}
```

```
public class FuelTruck extends Truck {  
    @Override  
    public void load(Cargo c) {  
        /* specialized implementation */  
        super.load(...);  
        ...  
    }  
    ...  
}
```

Caution:
Parameter types
must be identical to
overridden method!

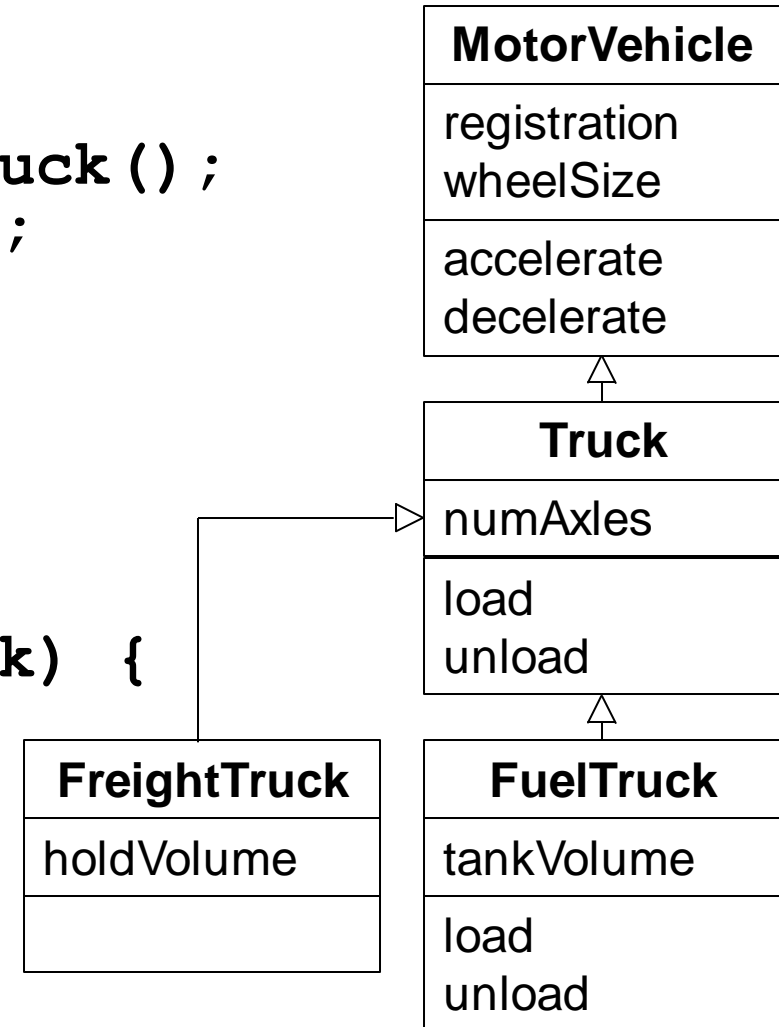
only if necessary:
Call the superclass'
load implementation



Polymorphism in Java

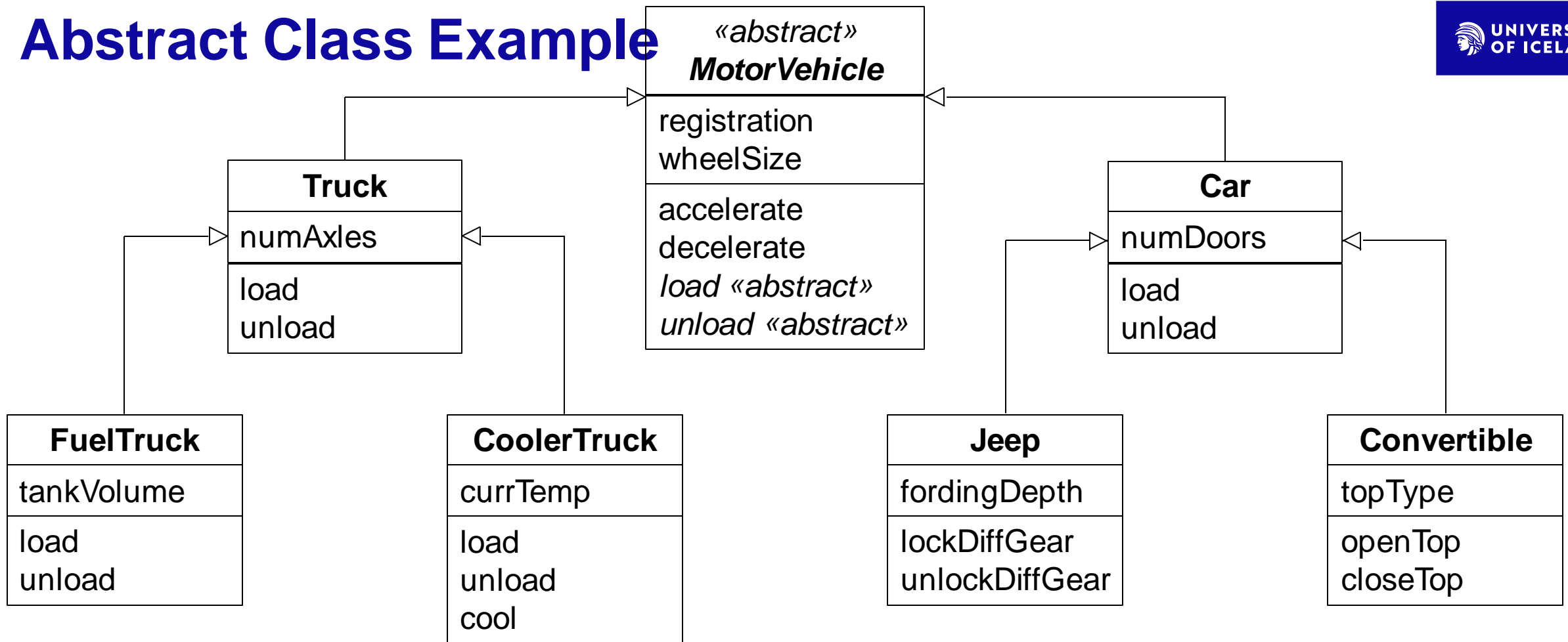
```
public class TestDriver {  
    public static void main(String[] args) {  
        FreightTruck freighter = new FreightTruck();  
        CoolerTruck cooler = new CoolerTruck();  
        FuelTruck tanker = new FuelTruck();  
        customsInspection(freighter);  
        customsInspection(cooler);  
        customsInspection(tanker);  
    }  
  
    public void customsInspection(Truck truck) {  
        truck.unload();  
        ...  
    }  
}
```

Nice:
Automatically uses the correct
`unload` implementation
for each call at run-time!



- Overridden methods exist in several forms – they are “polymorphic”.
- We have the choice of treating objects in their specific or generic form:
 - If we need to work with an object’s specific features, we refer to it through the interface provided by its own class definition (e.g. FuelTruck).
 - If we do not care about an object’s specific features, but only its general characteristics, we can refer to it through the interface provided by a superclass that is appropriate for our purpose (e.g. Truck or even MotorVehicle).
 - In that case, we do not even have to care about which specific class our object is an instance of – we simply refer to it as if it was an instance of the appropriate superclass.
 - Helpful e.g. when storing objects of related types, cycling through lists of related objects, etc.
- This gives us great flexibility in working with objects in a “natural” way, and in adding or changing specific implementations without having to touch most other code that only relies on the generic aspects.
 - One of the key benefits of object-oriented design and programming

Abstract Class Example



- All MotorVehicles can be loaded and unloaded somehow, but we can't define a general procedure for it on this high level.
- And there are no concrete "MotorVehicles" on our roads, just Trucks and Cars.

- Occasionally, we introduce a superclass only for structural purposes
 - i.e. we don't want to create instances of it, but bundle some general characteristics that will be inherited and extended by specialized subclasses.
- And/or we might not be able to provide general implementations for some of a superclass' methods, but still want to ensure that those methods will be provided in specialized form by the subclasses
 - i.e. we want to enforce an interface that all subclasses must conform to
- In these cases, we can mark the method and/or class as “abstract”
 - Meaning: No concrete instances of the class can exist
- A class must be abstract if at least one of its methods is abstract
 - But it can also be declared abstract even if all its methods are implemented
- Subclasses must provide implementations for their superclass' abstract methods or be declared as “abstract” themselves

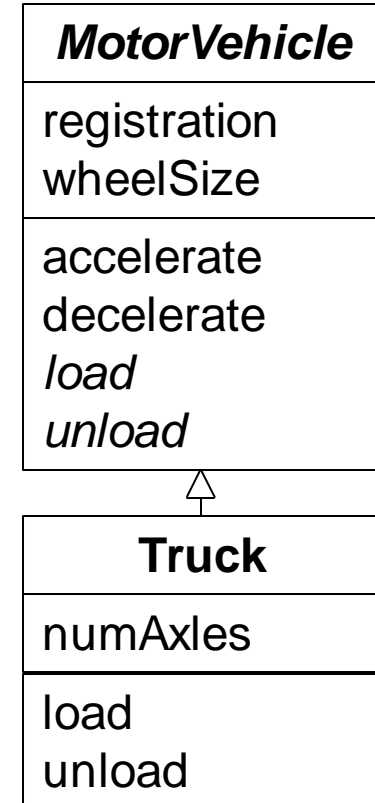
Abstract Classes in Java

```
public abstract class MotorVehicle {  
    ...  
    public abstract void load(...);  
    public abstract void unload(...);  
}
```

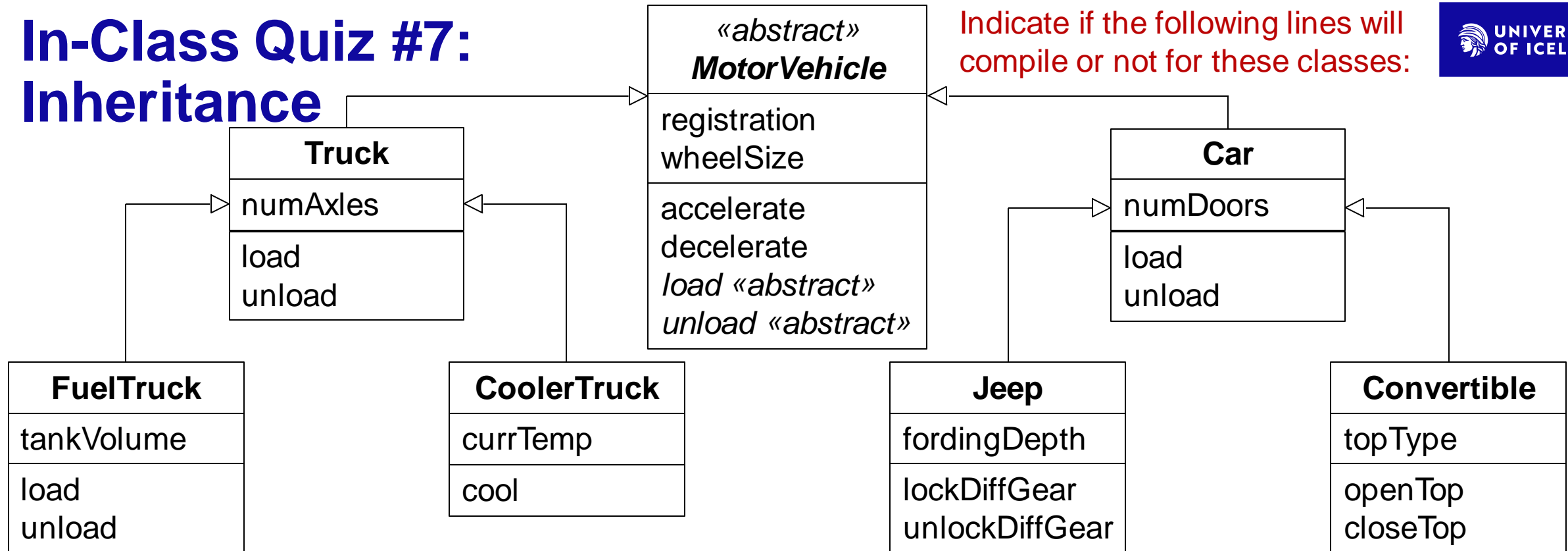
Note:
No method body!

```
public class Truck extends MotorVehicle {  
    ...  
    public void load(...) {  
        /* specialized implementation */  
    }  
    public void unload(...) {  
        /* specialized implementation */  
    }  
}
```

Compiler error if we omit any
of these, unless we declare
Truck as **abstract** as well



In-Class Quiz #7: Inheritance



- a) `Jeep j = new Jeep(); j.lockDiffGear();`
- b) `MotorVehicle mv = new MotorVehicle(); mv.accelerate();`
- c) `Convertible conv = new Convertible(); conv.load();`
- d) `Car car = new Convertible(); car.openTop();`
- e) `CoolerTruck ct = new Truck(); ct.currTemp = -18;`
- f) `Truck t = new FuelTruck(); t.numAxles = 3;`
- g) `CoolerTruck cool = new CoolerTruck(); cool.unload();`

Thank you!

book@hi.is

