SEN9110 Simulation Masterclass
Lecture 6. Object-Oriented Simulation

Prof.dr.ir Alexander Verbraeck
a.verbraeck@tudelft.nl

Brightspace: SEN9110

Agenda of this lecture

- Final questions about DEVS paradigm
- Questions about object-oriented papers
- Principles of object-orientation
- Object-orientation in simulation
- Example: DSOL

1.

Principles of object-orientation

Object Orientation: what is it?

- Object Orientation is a **modeling approach** [Booch:1999]
- We **choose** to see the world object oriented because it helps us to understand the systems we are designing and analyzing
- Choosing an OO-approach has strong **consequences** (our promise: you will never be able to see the world not-OO based again)

Characteristics of an object

- **Identity**: you can name an object or distinguish it from other objects

- **State**: the state of an object is defined by its attributes (i.e. age, speed, weight, size, etc.)

- **Behavior**: you can do things to the object (invoke its methods) and the object can invoke methods on other objects.

Objects as the basis for OO approach

source: msnbc website

Object: a traffic jam

Object: a car

Object: a tire

Principles of object orientation (1/9)
1. The ability to specify classes

- Modeling perspective: template for a type of objects
- Design perspective: a class is just another object
- Implementation perspective: global object

Principles of object orientation (2/9)
2. Information hiding

- An object explicitly describes which attributes are publicly visible and therefore accessible
- Public, protected and private modifiers enable information hiding

Principles of object orientation (3/9)
3. Encapsulation

- Attributes and methods uniquely belong to an object
- No need to check access! Access is controlled in the class

Principles of object orientation (4/9)
4. Polymorphism

- A programming language's ability to process objects differently depending on their class. The ability to redefine methods for derived classes.

Principles of object orientation (5/9)
5. Inheritance

- Classes can be organized in a hierarchical structure. In such a structure the subclass inherits the **protected and public attributes and methods from the superclass.**

Principles of object orientation (6/9) 6. Delegation

- An object passes the invocation of a method on to another object that actually fulfils the invoked method

Principles of object orientation (7/9)
7. Asynchronous communication

- An object invokes a method on another object where it does not expect immediate result

Principles of object orientation (8/9) 8. Late binding (or dynamic or run-time binding)

- Support for the ability to determine the specific class and thus the specific specification at runtime. This mechanism enables polymorphism.

Principles of object orientation (9/9)
9. Design by contract

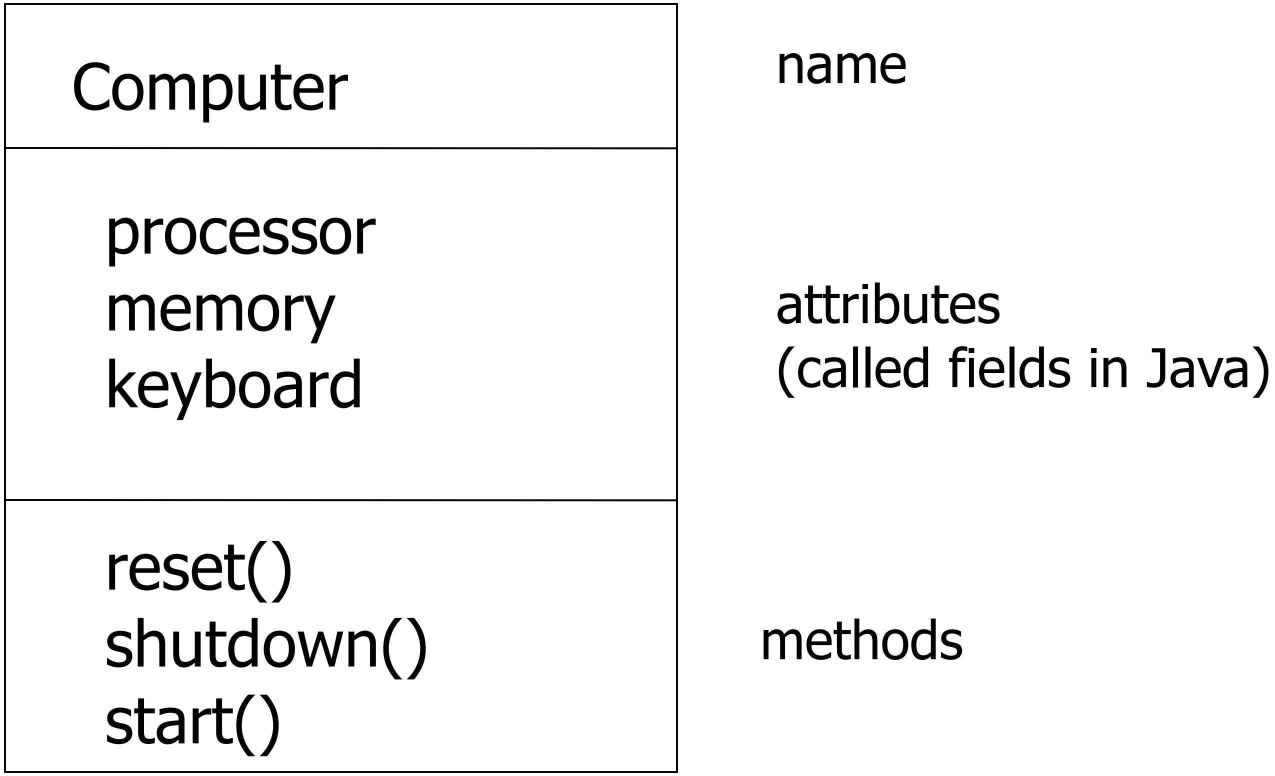
- The ability to design a set of methods as a contract via the usage of interfaces.

Classes as templates for objects

- **Classes** are the most important building block of any object-oriented system.

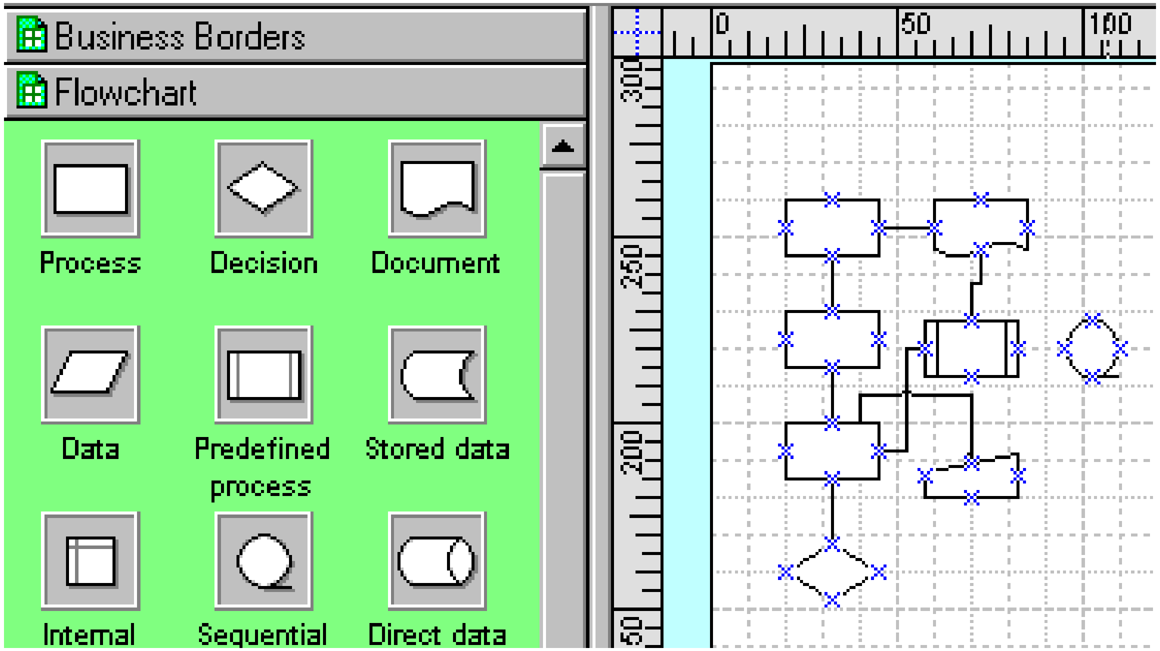
- A **class** is a template for a set of objects that share the same attributes (defining their state), operations, relations and semantics.

The illustration of a class



Instantiate objects from a class

Analogous to instantiating a shape in Visio:



Class libraries

- nl.abnamro.management.**Chair** class versus nl.pastoe.furniture.**Chair**

- **Libraries** group a logical set of classes. Statistical library, graph library, hi-speed math library, supply chain library, automated guided vehicle library

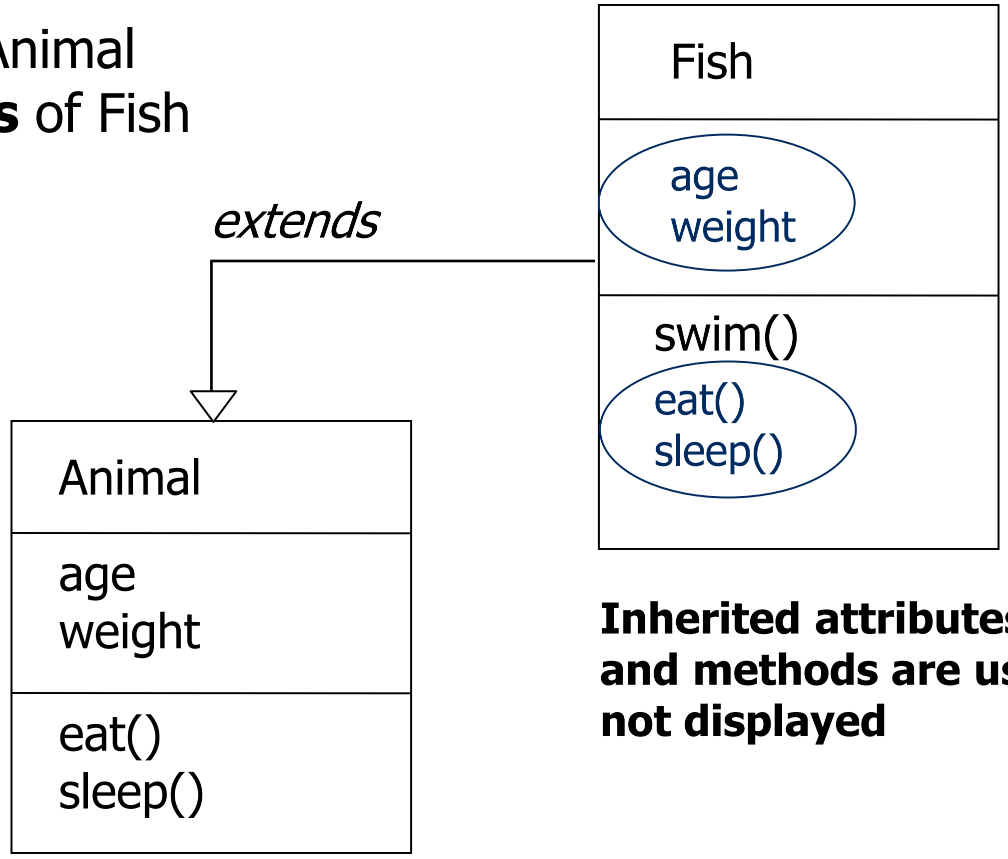
Classes : subclass & superclass

- A **subclass** is a specification of a class. We design a subclass whenever extra operations, relations or attributes describe a subset of objects. Fish **is** a subclass of Animal **when** we define the operation swim().

- A **superclass** is the more general "parent" class from which operations, methods and relations are **inherited**

Classes: subclass & superclass

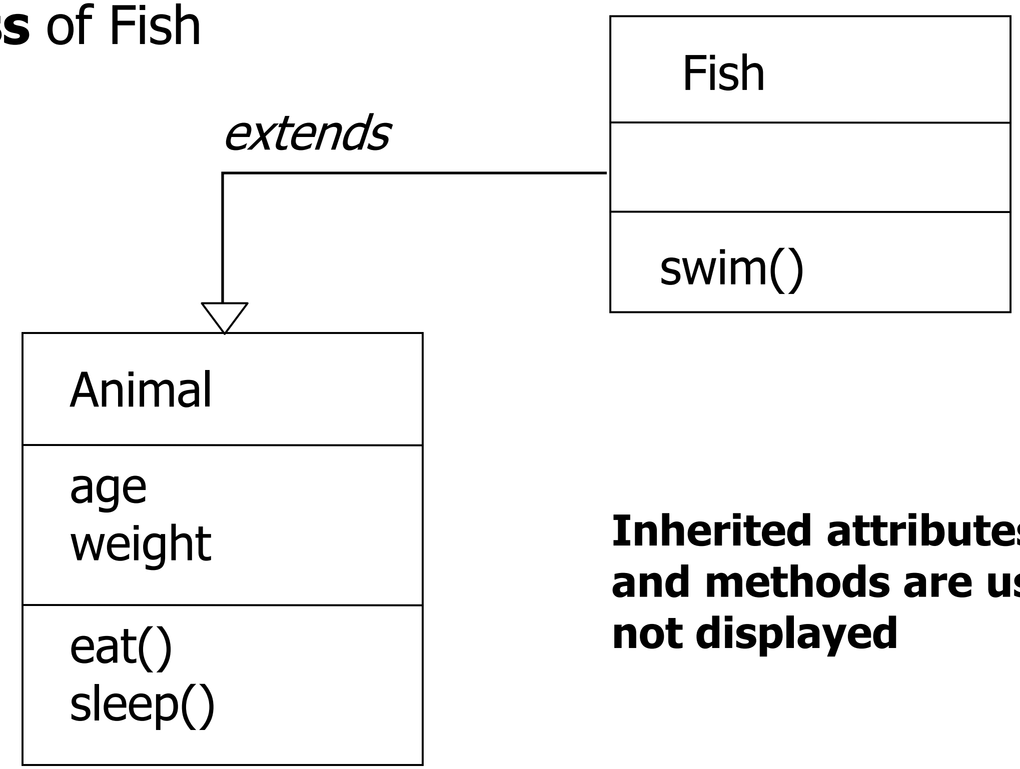
Fish is a **subclass** of Animal
Animal is a **superclass** of Fish



Inherited attributes and methods are usually not displayed

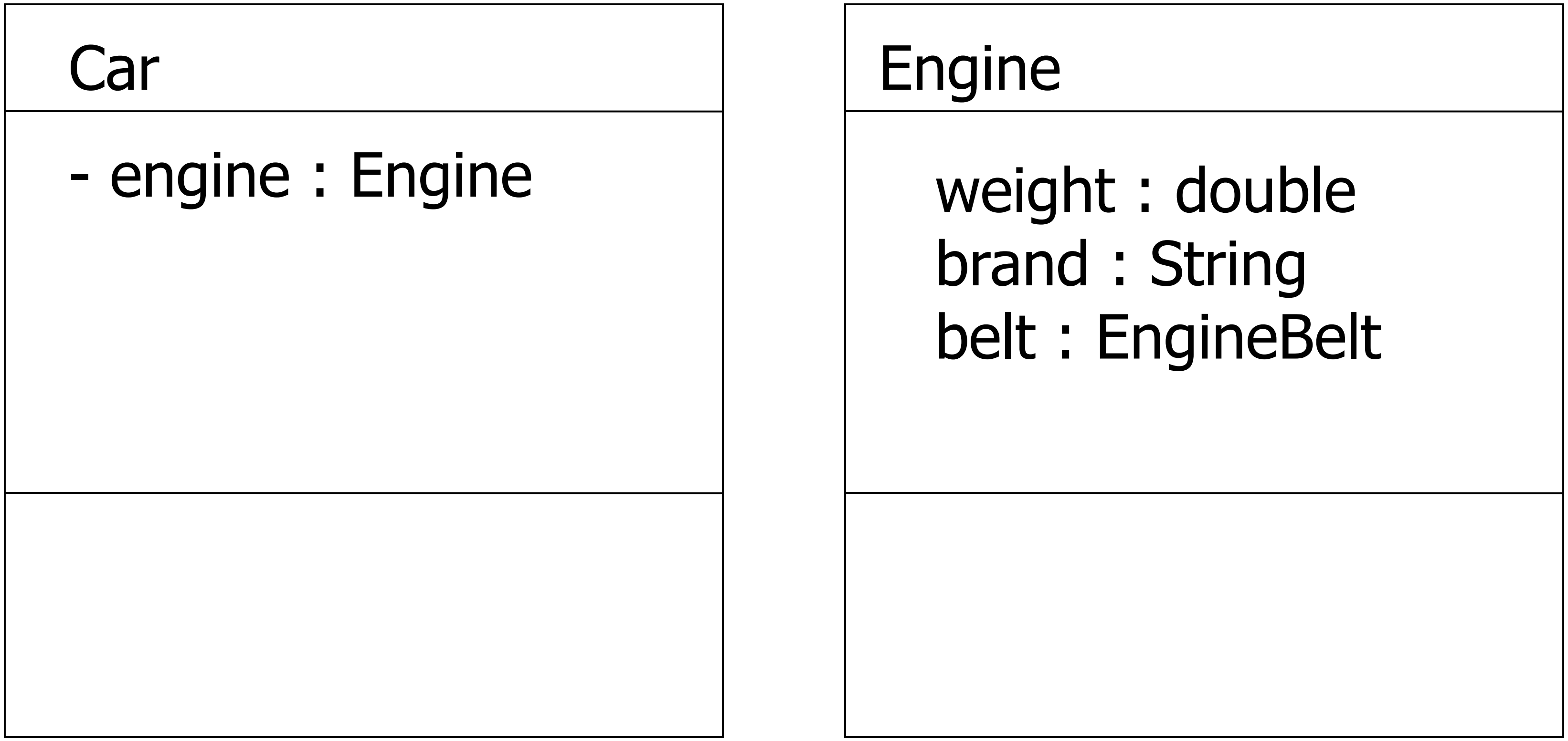
Classes: subclass & superclass

Fish is a **subclass** of Animal
Animal is a **superclass** of Fish

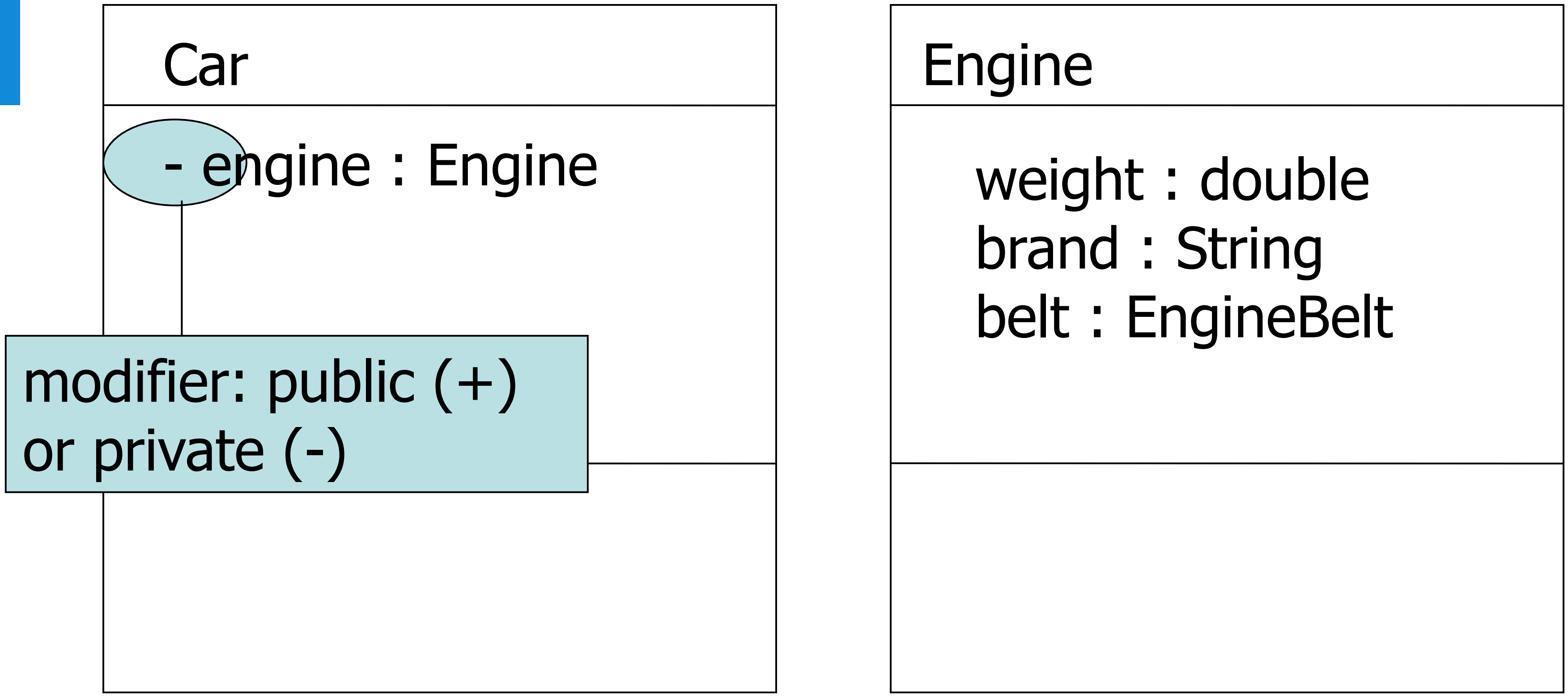


Inherited attributes and methods are usually not displayed

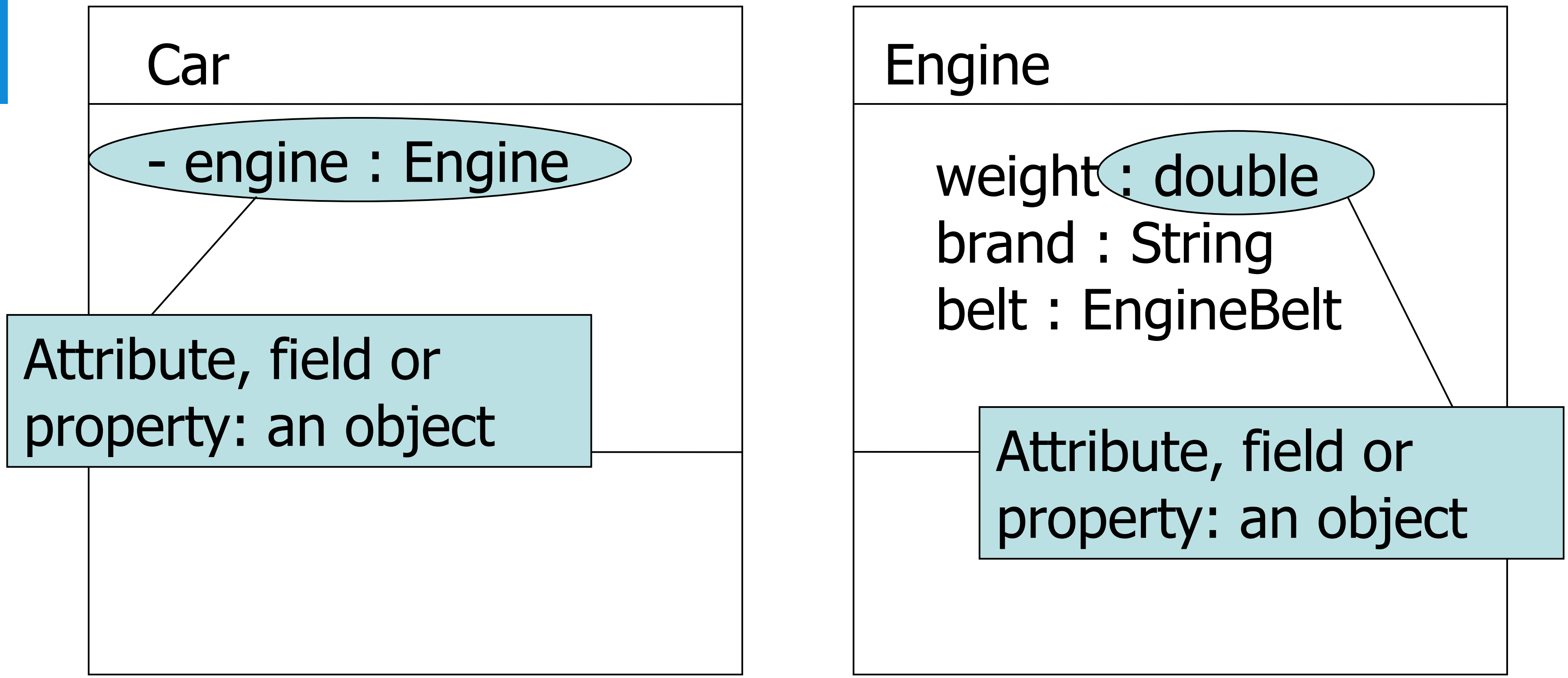
Classes and their relations (1/10)



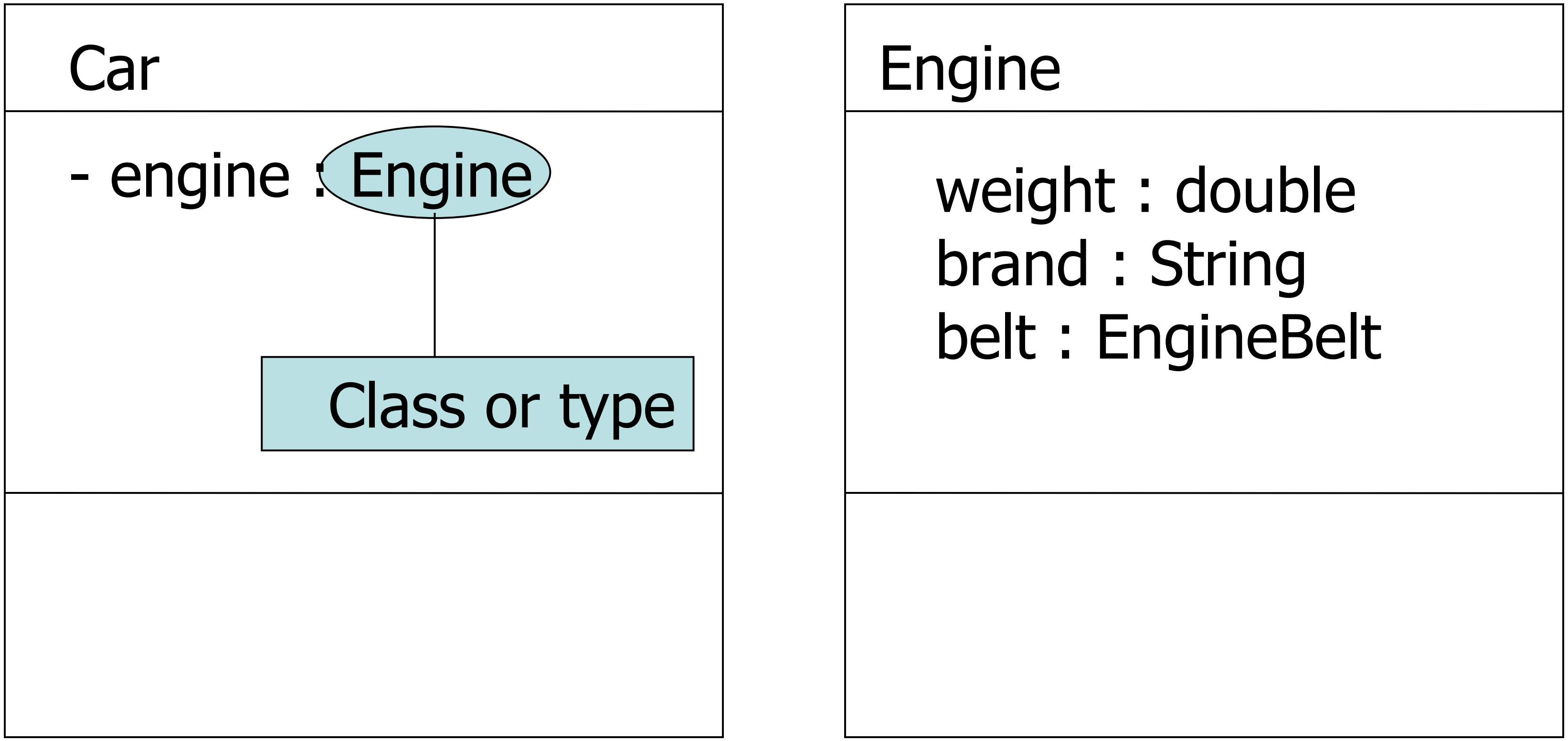
Classes and their relations (2/10)



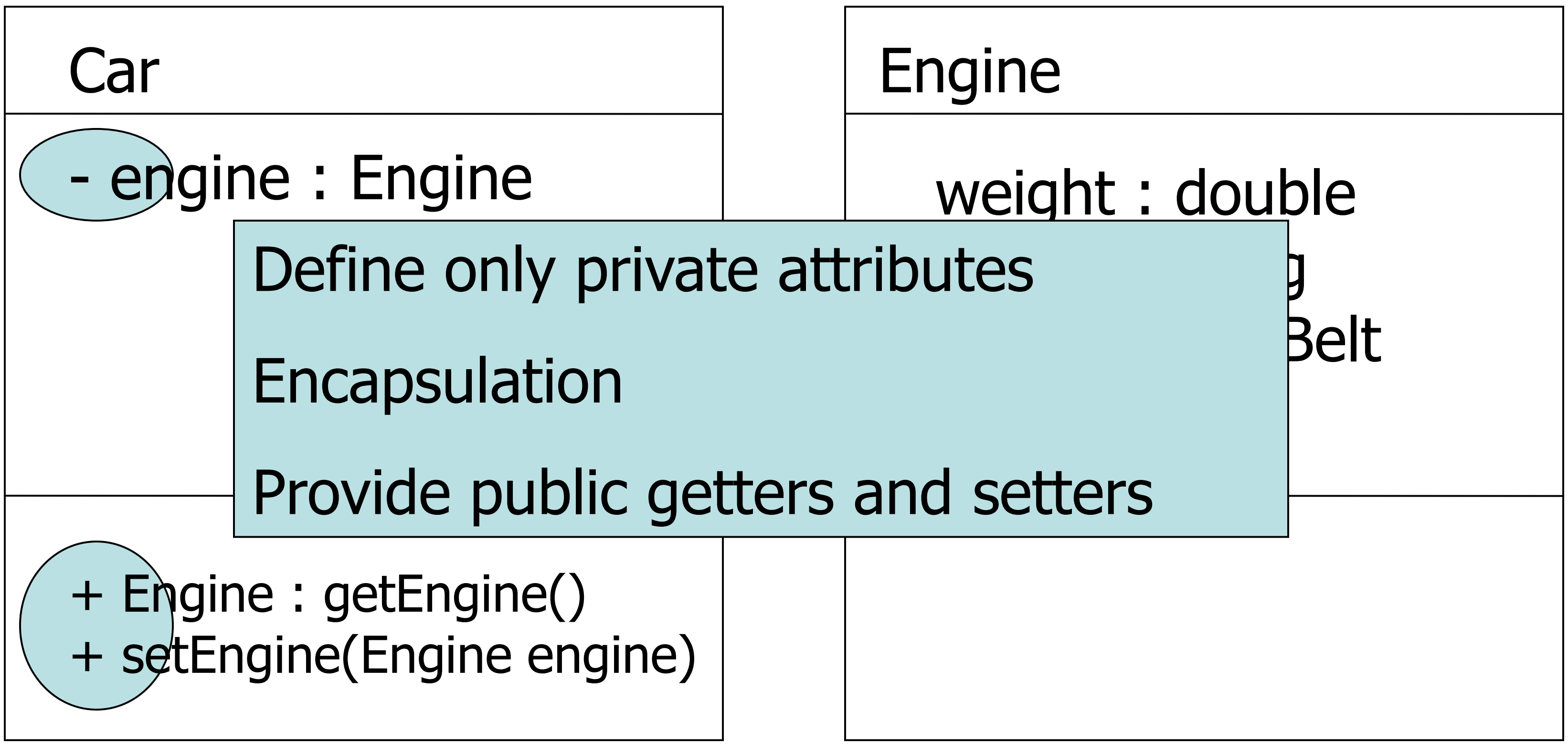
Classes and their relations (3/10)



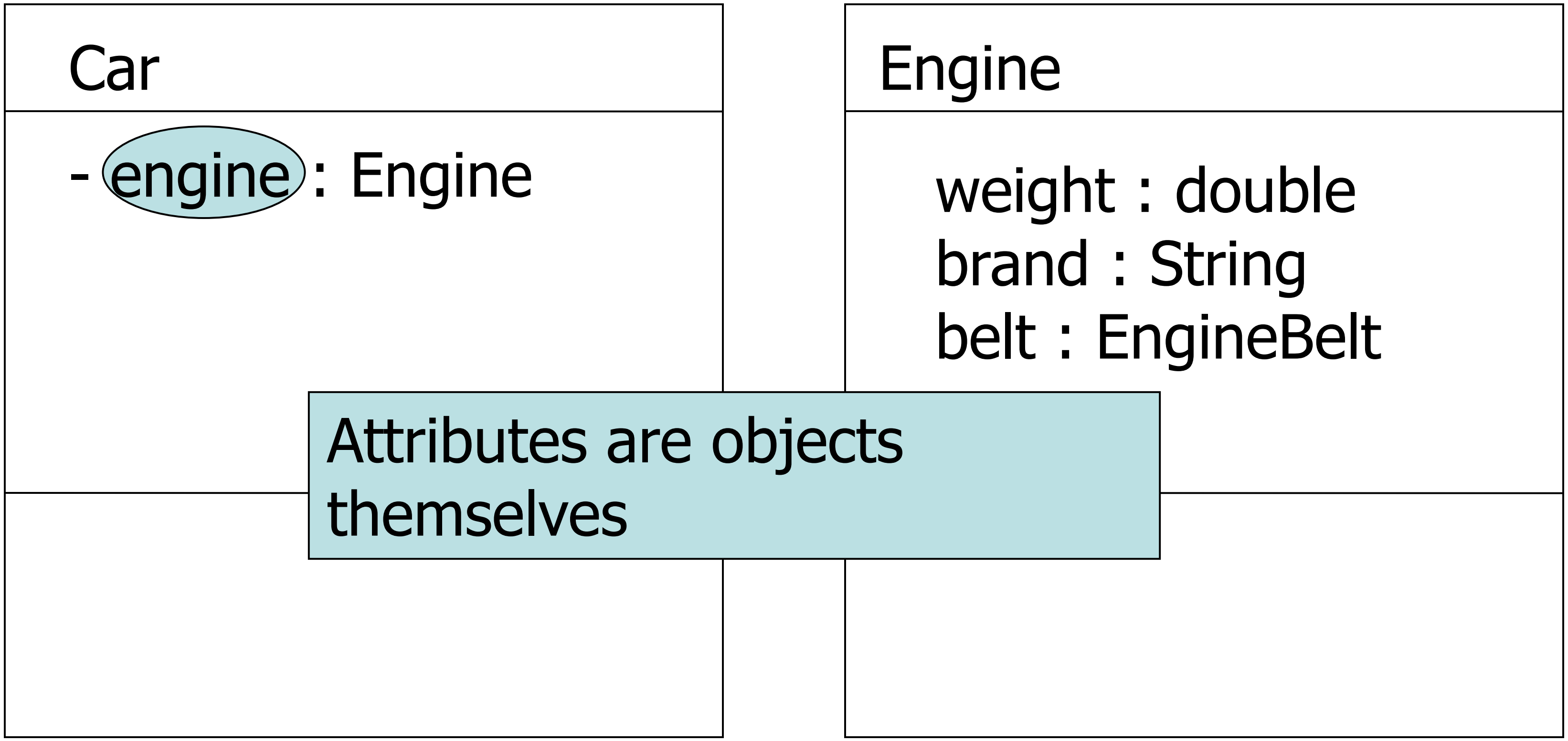
Classes and their relations (4/10)



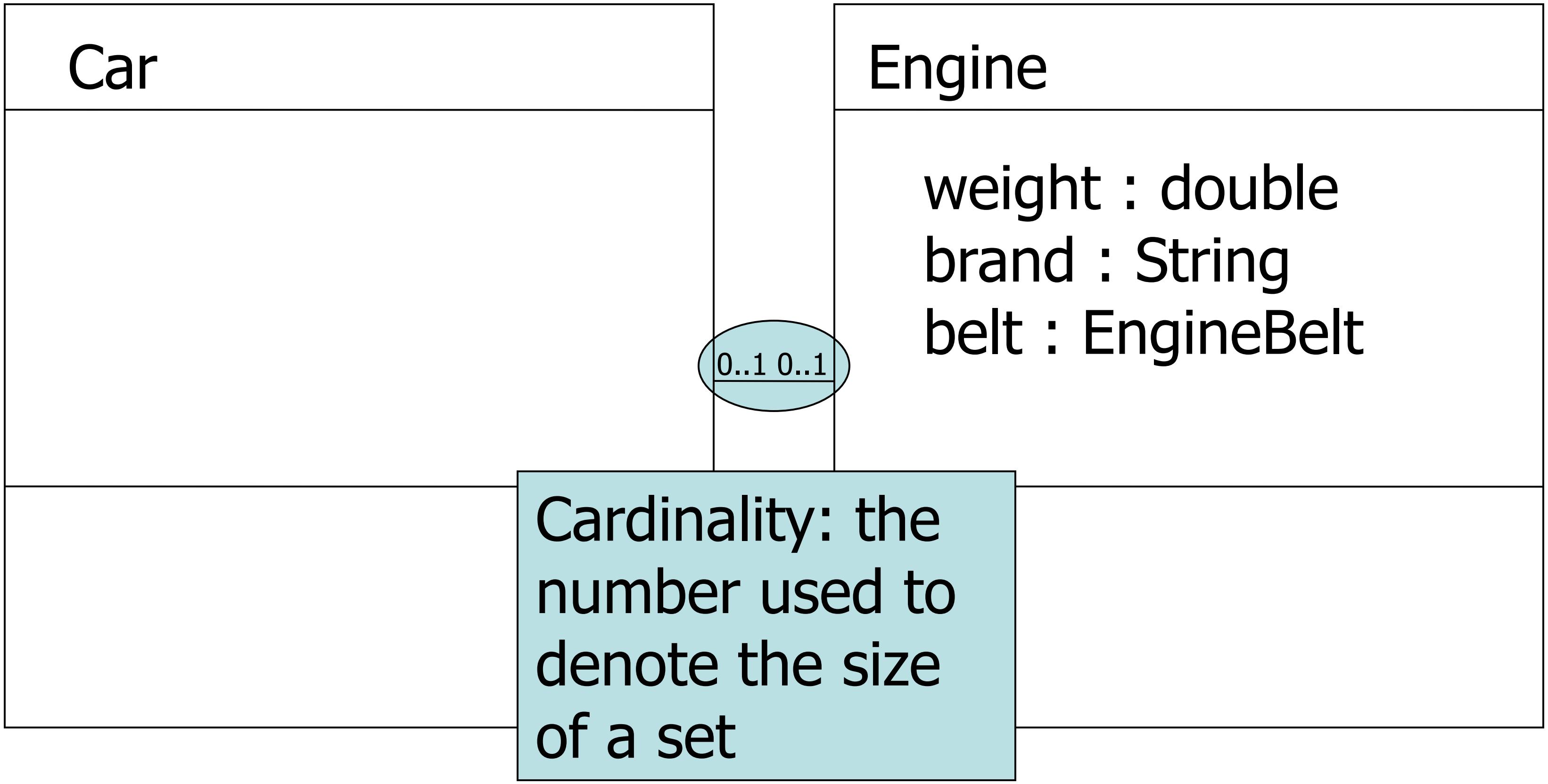
Classes and their relations (5/10)



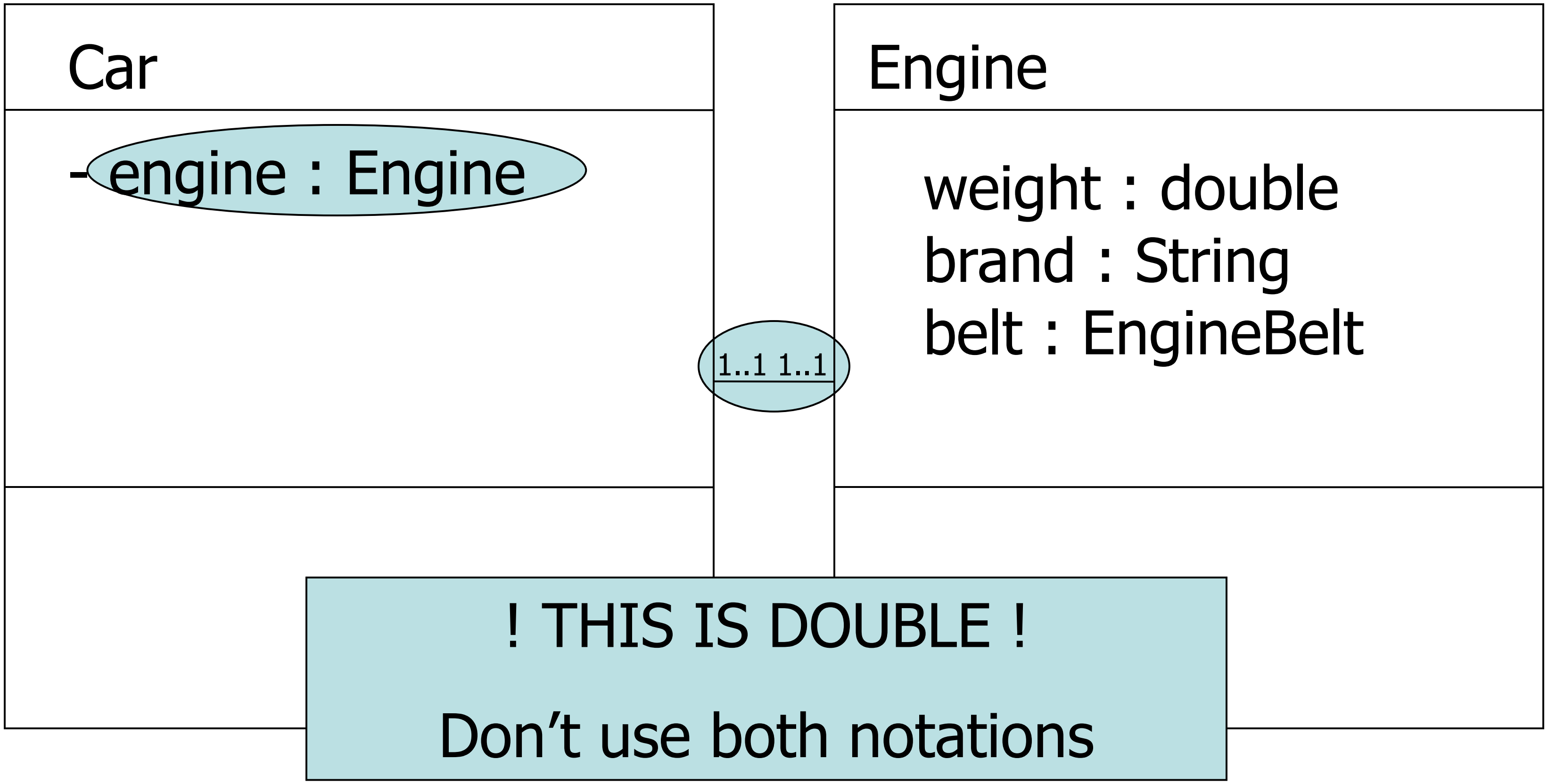
Classes and their relations (6/10)



Classes and their relations (7/10)



Classes and their relations (8/10)



Classes and their relations (9/10)

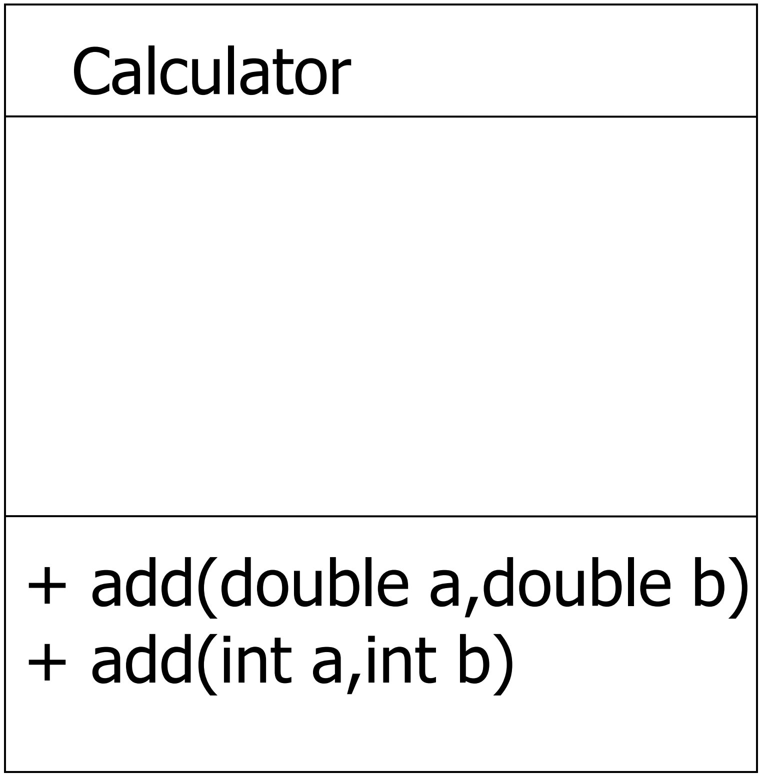
- **Aggregation** : a car physically consists of an engine, tires, steering wheel, etc.

- **Association**: a newspaper and a customer are associated by a subscription

- In OO languages such as Java there is (often) **NO DISTINCTION** between aggregation and association

- This is a **BIG PROBLEM** for simulation
  - WHY?

Classes and their relations (10/10) Polymorphism



Polymorphism enables the distinction of method invocation based on different 'signatures'

Polymorphism is method overloading

Be smart! It helps preventing redundant specification

2.

Object orientation in Simulation

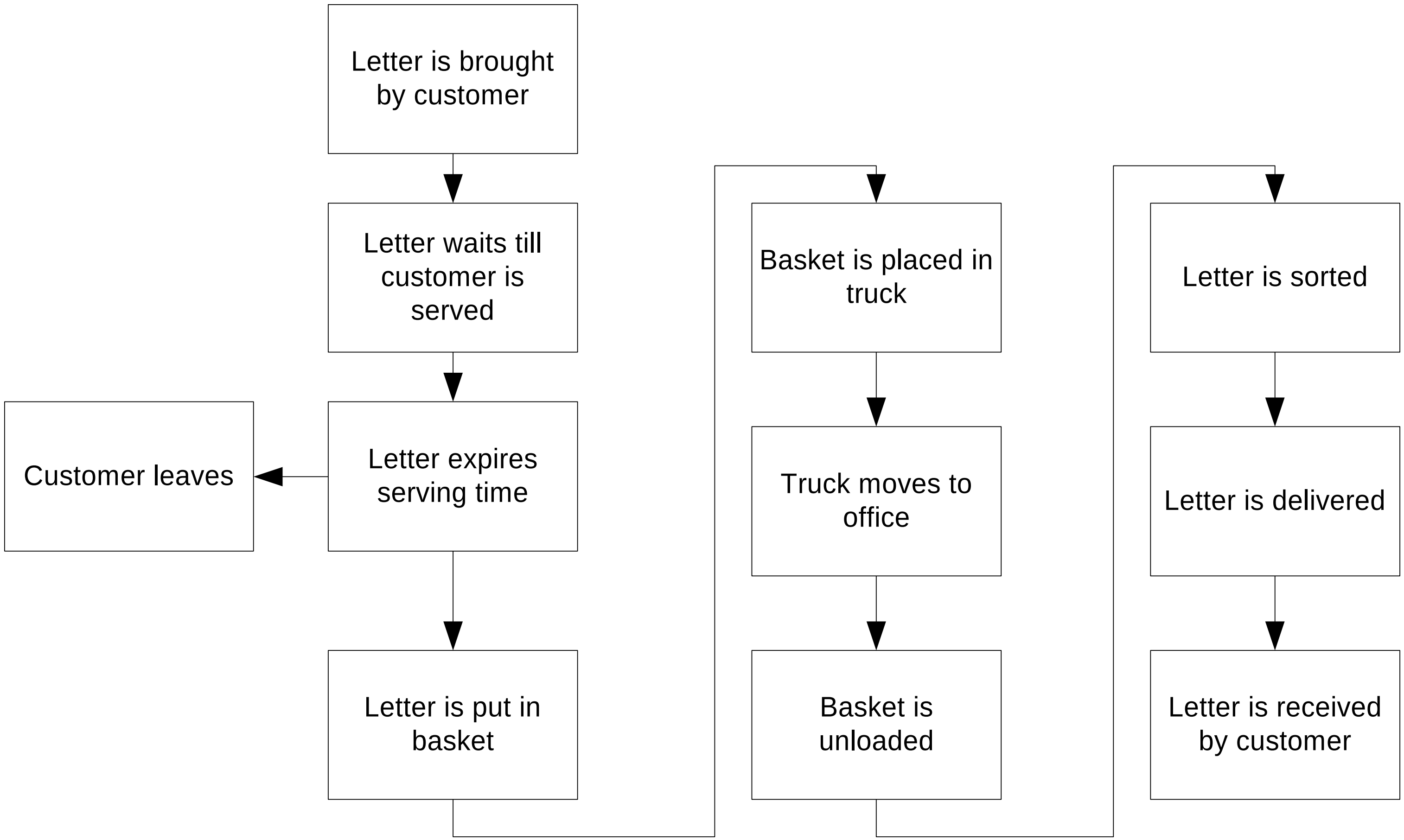
Simulation models

- Models are described in several formalisms. How do they relate or comply with OO?

- Flow based models (Arena)
- Activity based models

- Arena is coded in C++, an object-oriented language. Same for Simio, which is coded in C#. Are their models therefore OO?

A flow based overview of posting a letter (1/5)



OO based specification of posting a letter (2/5)

We create instances for our simulation model. See the difference between instances and classes.

(Capital S, C, P means class, small s, c, p means instance)

```
Simulator simulator = new Simulator()
Customer customer = new Customer()
Postbox postbox = new Postbox()
Postman postman = new Postman()
PostOffice postOffice = new PostOffice()
```

OO based specification of posting a letter (3/5)

- simulator → customer.moveTo(postbox)
- customer → postbox.receive(letter)

- simulator → postman.moveTo(postbox)
- postman → postbox.empty()
- postman → this.moveTo(postOffice)
- postman → postOffice.receive(collectedLetters)

OO based specification of posting a letter (4/5)

simulator → customer.moveTo(postbox)
customer → postbox.receive(letter)

simulator → postman.moveTo(postbox)
postman → postbox.empty()
postman → this.moveTo(postOffice)
postman → postOffice.receive(collectedLetters)

Event scheduling in a simulation

OO based specification of posting a letter (5/5)

- simulator → customer.moveTo(postbox)
- customer → postBox.**receive**(letter)

- simulator → postman.moveTo(postbox)
- postman → postbox.empty()
- postman → this.moveTo(postOffice)
- postman → postOffice.**receive**(collectedLetters)

Postbox and PostOffice both implementing a receiver

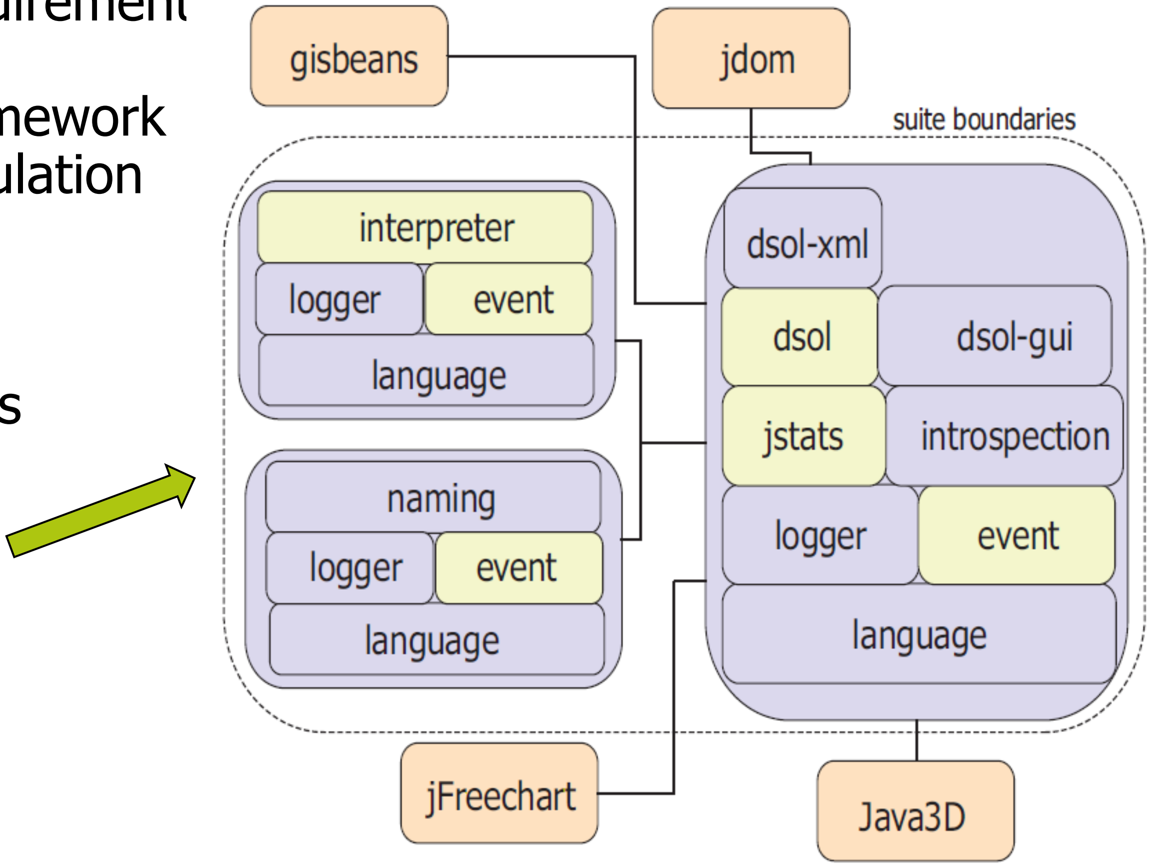
3.

Object orientation in Simulation Languages: DSOL

Aims of this part of the presentation

Discuss the 3 main requirement

- **Distributing** the framework for modeling and simulation
- Providing enough **formalisms** for the construction of models
- Implements DSOL in a **service oriented** architecture



The problems we aim at in modern simulations:

- Multiple **distributed** stakeholders
- System conceptualization in a **system** of models
- System specification is tightly linked with **underlying IT infrastructure**
- **Collaborative** model construction and problem solving
- The construction of **complex** models, in terms of time, aspects and scope

A vision on service based simulation (1)

Current simulation tools	Service based simulation
1 stakeholder is locally supported by a simulation study	N stakeholders are web-enabled and simultaneously supported
1 model formalism is allowed in a simulation study	N formalisms can be used to express parts of the model
1 simulation expert can simultaneously work on a simulation model	N simulation experts can collaboratively work on a model

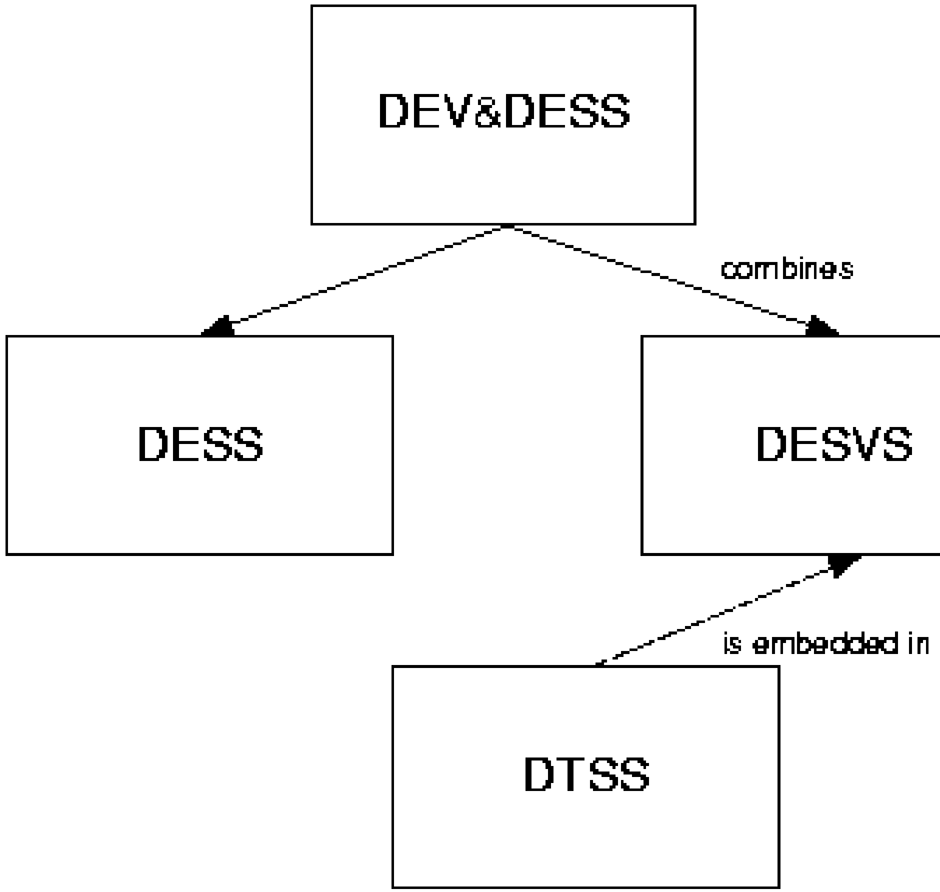
A vision on service based simulation (2)

Current simulation tools	Service based simulation
1 set of tools can be used for reporting, animation, etc. etc.	N services can interact to present, report, compute the study
1 processor can be used to deploy the simulation model	N computational processors can be used to deploy models, services and simulators
monolithic models are developed, globally accessible members	modular models are developed, encapsulated members

Distribution in the framework for M&S.

- Distributed DSOL **application**: Open remote experiments, report to the web, view animation remotely. Supported in DSOL : **YES**!
- Distributed **model** specification: Entity and simulator distributed, customer and resource distributed, etc. Supported in DSOL : **YES**!
- Distributed **simulator**: HLA / IEEE 1516, time synchronization between simulators. Supported in DSOL : **YES**!

Formalisms in the framework for M&S



Relations between formalisms

- **Combining** relation : A combines B and C whenever B and C in A
- **Embedding** relation: A embeds B whenever any model expressed in B can be expressed in A
- **Root formalism**: there is no formalism which can be embedded by root formalism A

See also the papers from Vangheluwe & De Lara that we will cover for multi-paradigm simulation.

Formalisms in DSOL

- DEV&DESS root formalisms is supported by a unique simulator. This **guarantees** multiformalism!
- Several non-root formalisms are nevertheless supported with unique simulator for **performance optimization**. (DESS, DEVS)
- Several non-root formalisms are supported by libraries **embedding** the formalism (Flow is embedded in DEVS)

Service oriented system design (I)

- **Object orientation** is the de facto modeling and programming language for the design of information systems

- A **service** is a **contractually** defined behavior that can be implemented and provided by any component for use by any other component, based solely on the contract. This contract is also referred to as the **interface** of a service

Service oriented system design (II)

- The **encapsulation** of states. Since a service only describes methods, the state of a component providing the service is shielded to those components invoking it
- The prevention of relations between objects or components. Service oriented design encourages objects to relate to an **interface**. Whenever the particular specification or embodiment of the service changes, this has no consequences for the relation
- **Collaborative** system specification. Whenever the interfaces of a system are designed, several engineers can collaboratively and concurrently implement parts of the system without potential integration problems

DEVS implementation of event scheduling

- Delayed method invocation

- Any method can be called based on a delay relative to the time base

```
simulator.scheduleEvent(delay, class, "methodName", arguments);

private startService()  {
  double hour = TimeUnit.convert(simulator, 1.0, TimeUnit.HOURS);
  simulator.scheduleEvent(hour, this, "serviceReady", null);
}

private newCustomer()  {
  Customer customer = new Customer(simulator);
  DistContinuous interArrivalTime = new DistExponential(5.0);
  double iat = TimeUnit.convert(simulator, interArrivalTime.draw(),
    TimeUnit.MINUTE);
  simulator.scheduleEvent(iat, this, newCustomer, null);
}
```

Conclusions

- DSOL is a Java-based distributed application,
  with pyDSOL containing a Python version of DSOL
- DSOL is an object-oriented simulation framework for distributed modeling
- DSOL supports several formalisms among which:
  DESS, DEVS, DTSS, DEV&DESS, DSDEVS, QDEVS
- DSOL and pyDSOL are open source and published under BSD on github