UML for code design

Paolo Burgio
paolo.burgio@unimore.it

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

High Performance
Real Time Lab


TECHNICAL DEBT
I DON'T UNDERSTAND WHY IT TAKES SO LONG TO ADD A NEW WINDOW.
@VINCENTDNL

UML (standard) diagrams

Structural diagrams

- Use-cases/scenarios
- Notations for classes/objects/packages/components – From OOP
- Deployment/components } won't see these

Behavioral diagrams

- Sequence diagrams
- State diagrams
- Activity diagrams

Now, let's code

UML provide abstractions to design how the code should look like

- From this perspective, what matter is **data**
- Previously, we focused on entities as system parts/units/components
- We modeled their behavior with sequence diagrams, under different use cases (streamlined from the requirement analysis)

Now, we need to switch to a lower abstraction level, and "look" at entities

- ER diagrams for DBs (already covered in other course)
- Z diagrams for algorithm
- Class/objects[/packages] for OOP

Remember, the goal of UML is to clearly define

what every part of the system does,

and how it interacts with the other parts

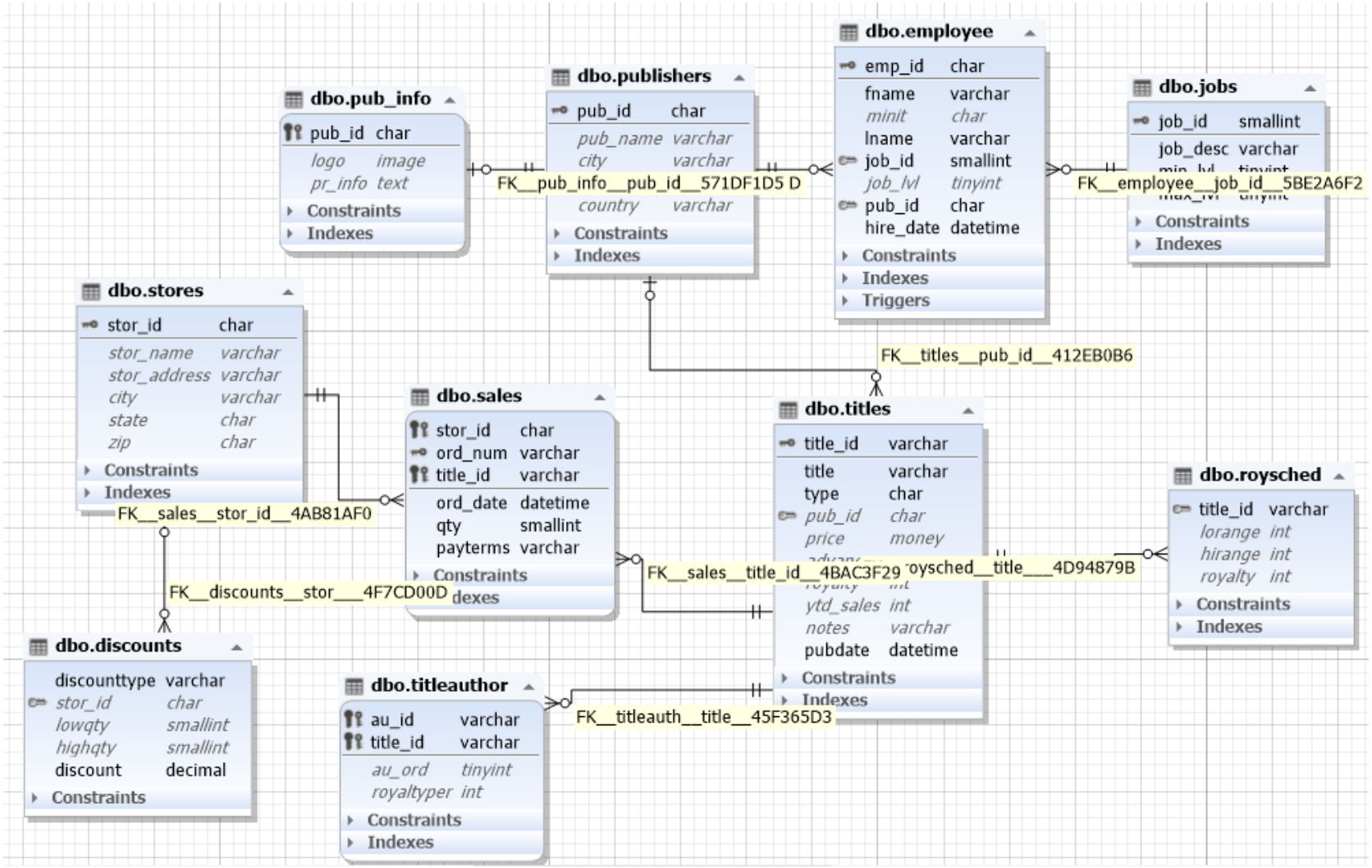
ER model from the SW engineering perspective

- The result of our analysis phase
- Identifies what data is created, and required by business processes
- Describes system components as blocks, and the relations among them

Typically, used to model our DB

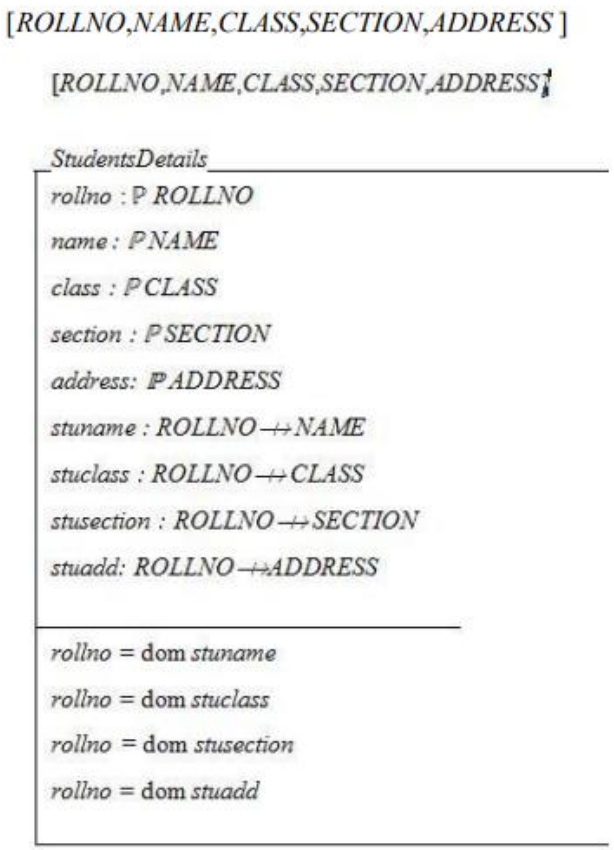
- Remember, we are **data architects**
- We can use tool to generate the code (model and driver) to interact with the DB

ER model from the SW engineering perspective



Z diagrams

- Formal specification language that aims at clearly identifying what a computer system does, and how it is done
- Based on mathematical concepts such as lambda calculus, first order predicate logic…
- ISO13568 (2002)



Pros

- Typed formalism
- Formally correct

Cons

- Complex to use

Class diagrams

Class diagrams

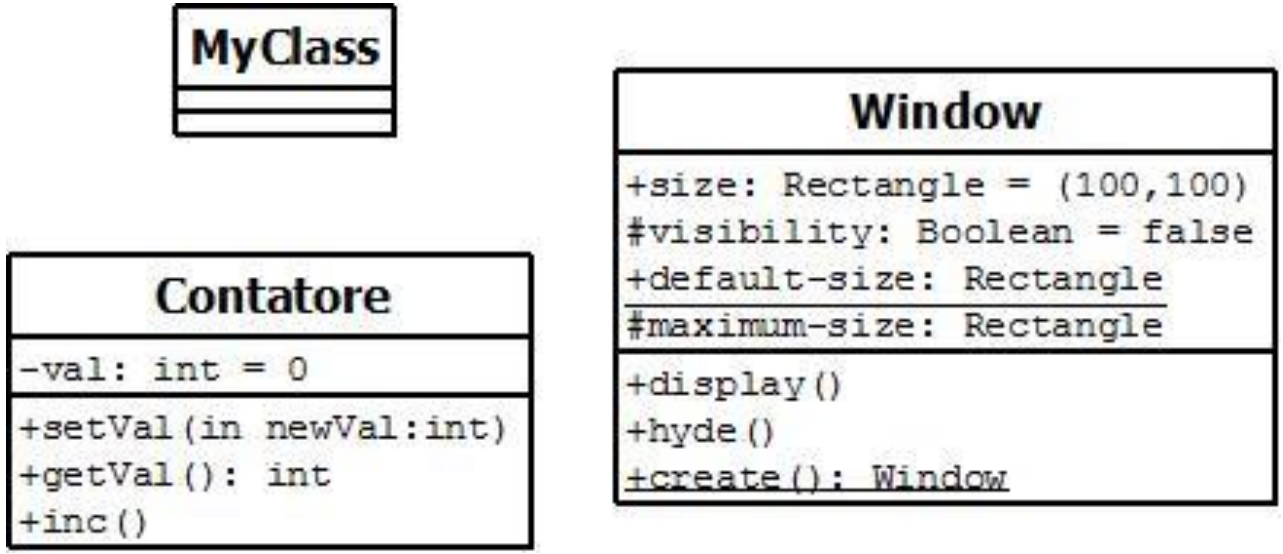
- A graph that (clearly) describes classes/interfaces and their relations by means of nodes and arcs
- Can be used to group elements within packages, or subsystems
- Used to model the static behavior of our system (i.e., classes), something that we can define at compile time

Why is it so important to define what happens at compile time vs. what happens at run-time?

The right abstraction

We need to model each entity (here, class) by the only properties that are of our interest

- E.g., in a gym club, we might want to model people age, weight, height…, while our bank account only models our age
- The only way to master complexity…is to reduce it!



OOP recap

What is a class?

- **Abstract** concept
- A descriptor of a set of object with common attributes, operations, relations, and behavior
- Groups data (fields) and operations (methods) for a specific set of objects
- *Philosophical pills: this breaks **S**OLID…we will see it later*

What is an object?

- Is a **concrete** instance of a class

Why are them …and OOP… so powerful?

- Enable classifying, and organizing, the knowledge of our problem
- Coming from the analysis/system design phase
- Ultimately, to correctly translating them into a code artifact

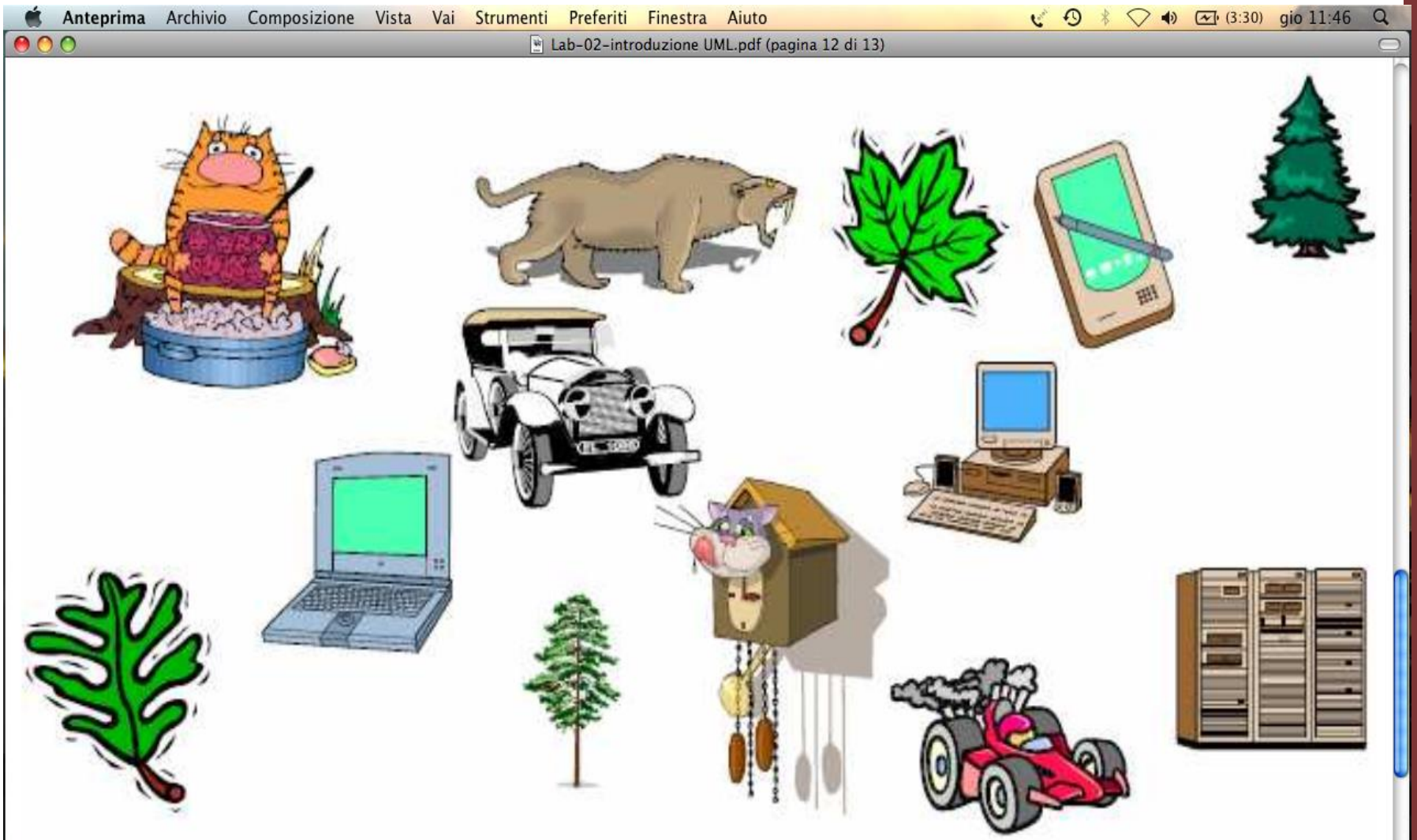
OOP recap

What is a class?

What is an object?

Why are them ...and OOP... so powerful?

Example: how many classes?



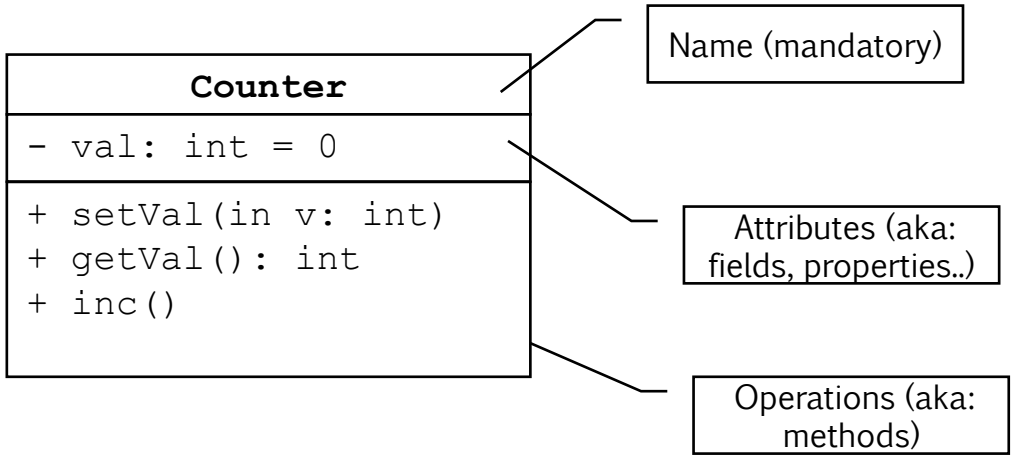
› We are forced to identify a proper abstraction level!

Basic representation

- "A descriptor of a set of object with common attributes, operations, relations, and behavior"
- A rectangle divided in three parts (similar to object diagrams…we'll see them later)

Note how they introduce the concepts of

- data type
- assignment
- Visibility (+ for public, - for private)

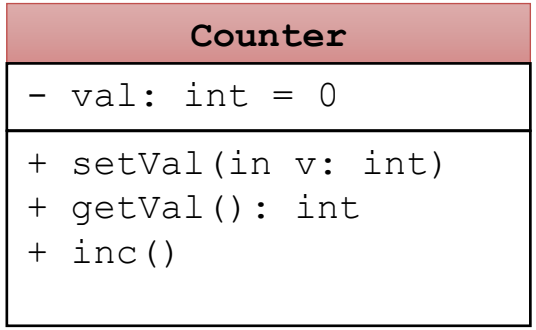


Name

A string of text

- Must start with capital letter (Java-style)
- Can be *prefix* + "`::`" + *name*
- No special characters ($, %, &)
  - why?

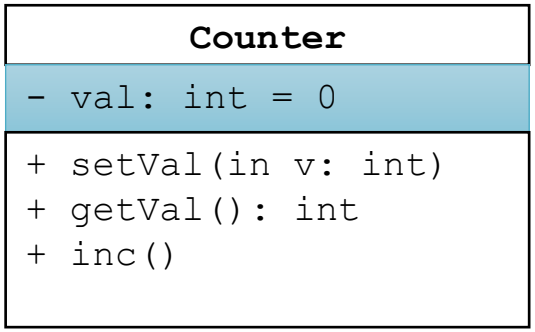
- Non-ambiguous



Attributes

<visibility> <name> <[cardinality]> **:** *<type>* **=** *<initial value>*

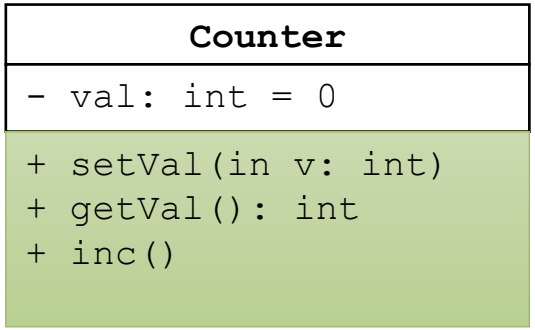
- Name is mandatory
- Visibility
  - **-** for private
  - **+** for public
  - **~** for package
  - **#** for protected
- Cardinality `[n]` for arrays
- `Static` attributes are <u>underlined</u>



Operations

<visibility> <name> (<param>: <type>, ..): <ret val type>

- Only name is mandatory
- `Static` methods and `ctors` are underlined
- Parameters can be preceded by a modifier: `in`, `out`, `inout`
- Non-Java style



Types of operations

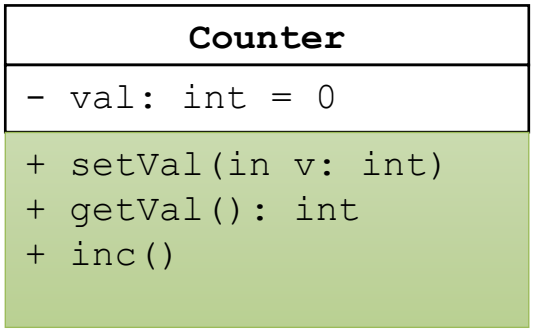
Queries

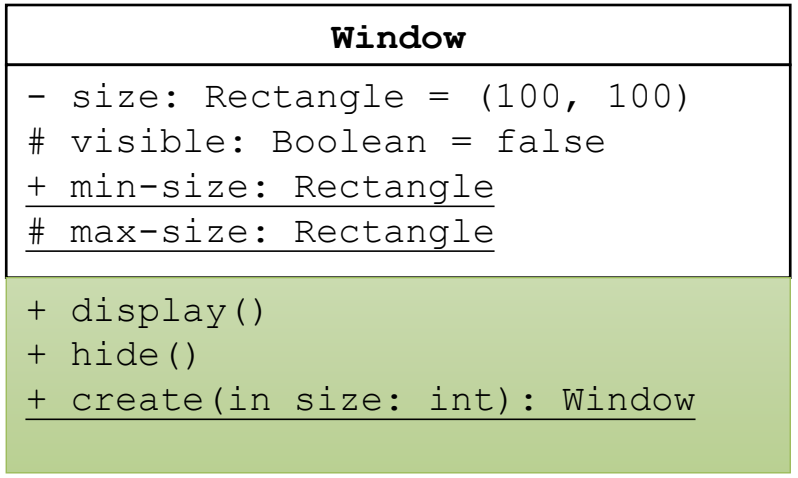
- (ex: Get-ters)
- Do not modify the status

Modifiers

- (ex: Set-ters)
- Modify the status

Ctors to create new instances

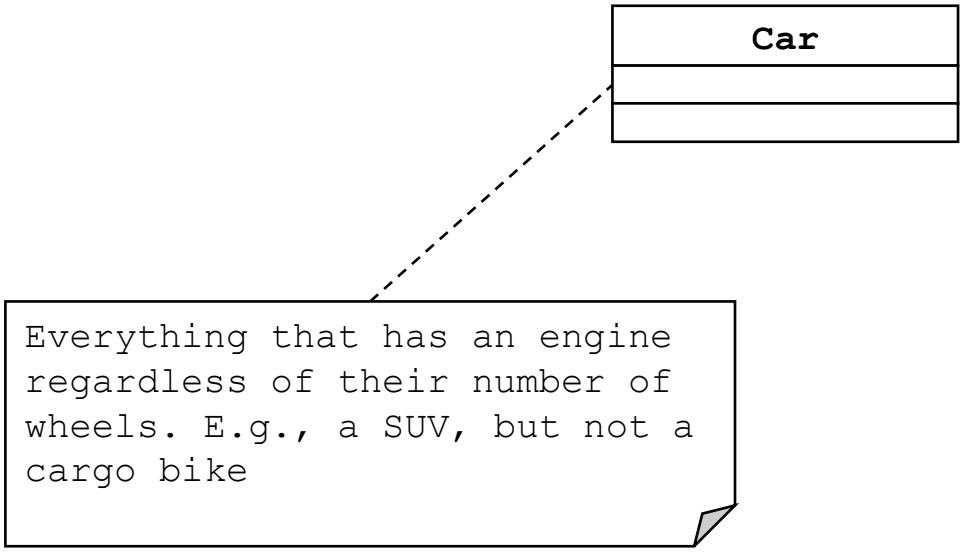




Adding notes and comments

Do not underestimate comments!!

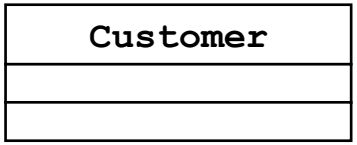
- We can automatically generate code (and their comments) by this
- We can automatically generate (technical) documentation by code comments
- We will have a dedicated lesson on that



Relation between classes

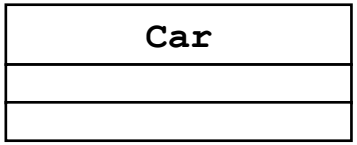
Associations

› Simple, aggregation, and composition



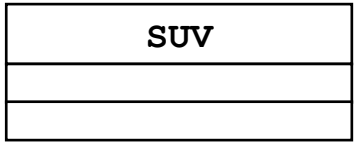
Dependency

› "Uses"



Generalization/specialization

› Has to do with inheritance and interfaces



Simple association

A solid line between classes

› Arrows specify directions (No arrow: bidirectional) – aka *navigability*

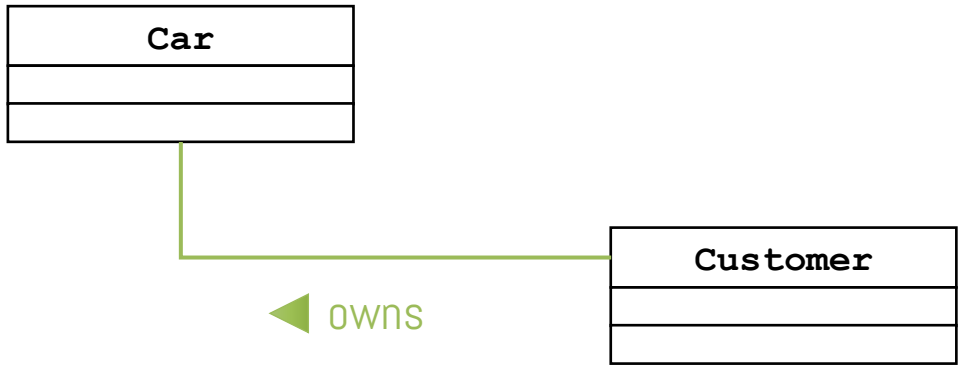
Features

› Name

› Roles

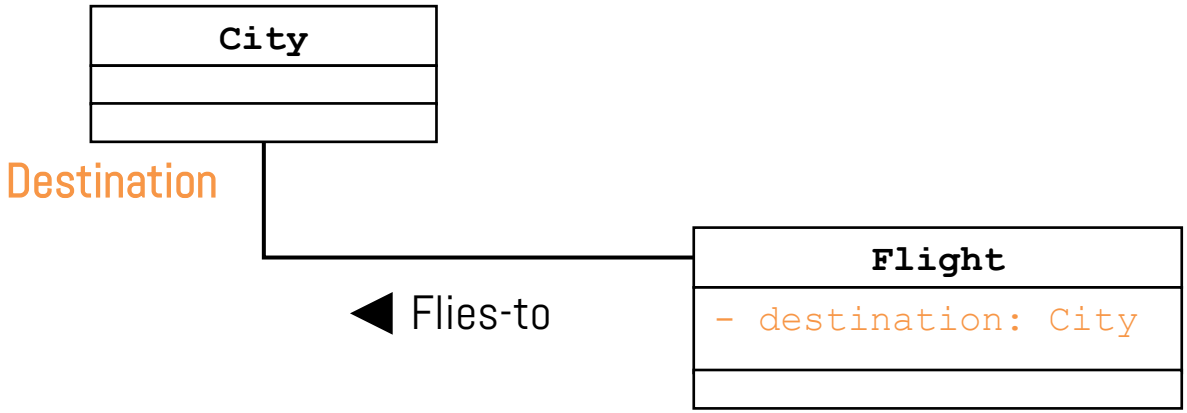
› Cardinality

› Navigability



Association: role names

- Goes in the direction of creating reference/fields
- Mandatory for reflective relations (between the same class) – we'll see them soon



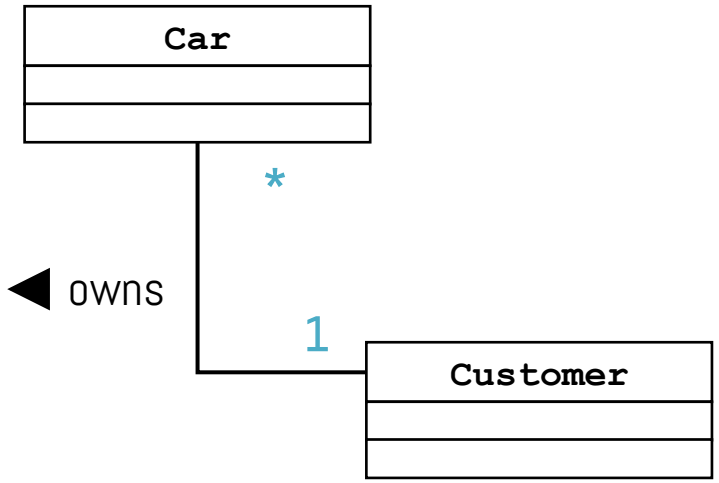
Association: cardinality

- Gives an information/bound to the number of objects that can participate to an association
- Useful information for programmers!!

Can be

- A symbol (*0 1 **)
- An interval (*1…6* means "*from one to six*")
- Comma-separated list

  (*1..3, 10…20* means *"1..3 or 10…20"*)

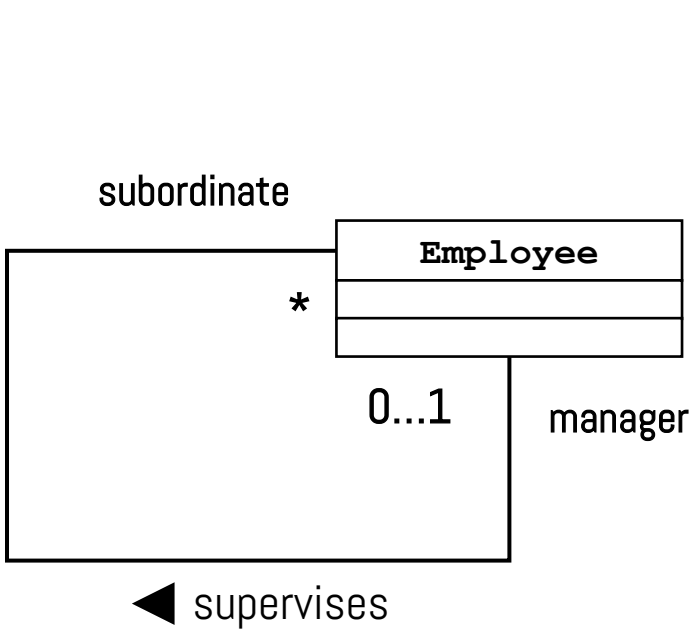


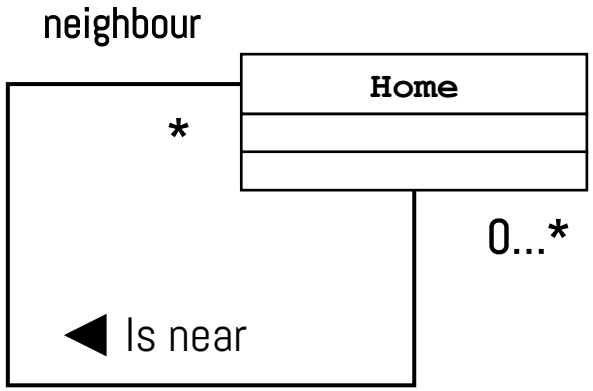
Notes

- Use *to specify any number
- **and *0…** are the same thing
- Often, *replaced with *N*

Association: reflective

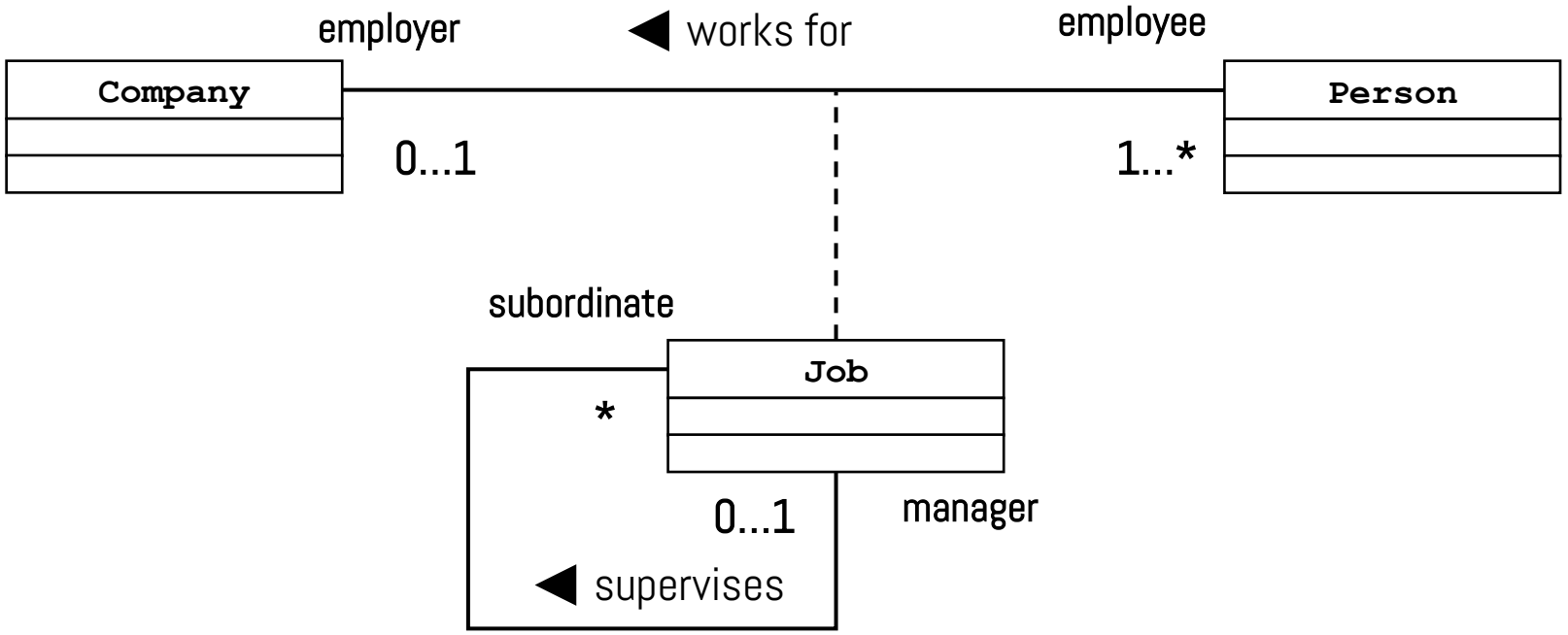
› In this case, roles are mandatory





Association classes

- Is a class that specifies an association
- Dotted line
- Defines operations, and attributes for that association



Aggregations

A class that contains another class (logically or even physically)

› Indicates that objects of that class are part of objects of another class

What makes them different by “normal” fields?

› The contained class has its own lifecycle

› Should ring a bell..

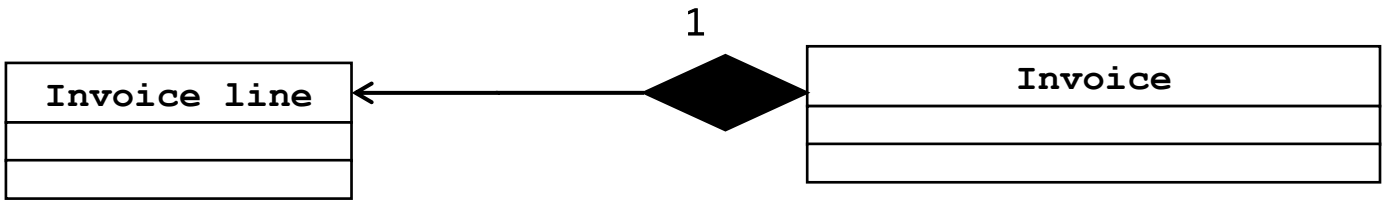
Modeled as **empy** rhombus (with cardinality) close to the containing class

Compositions

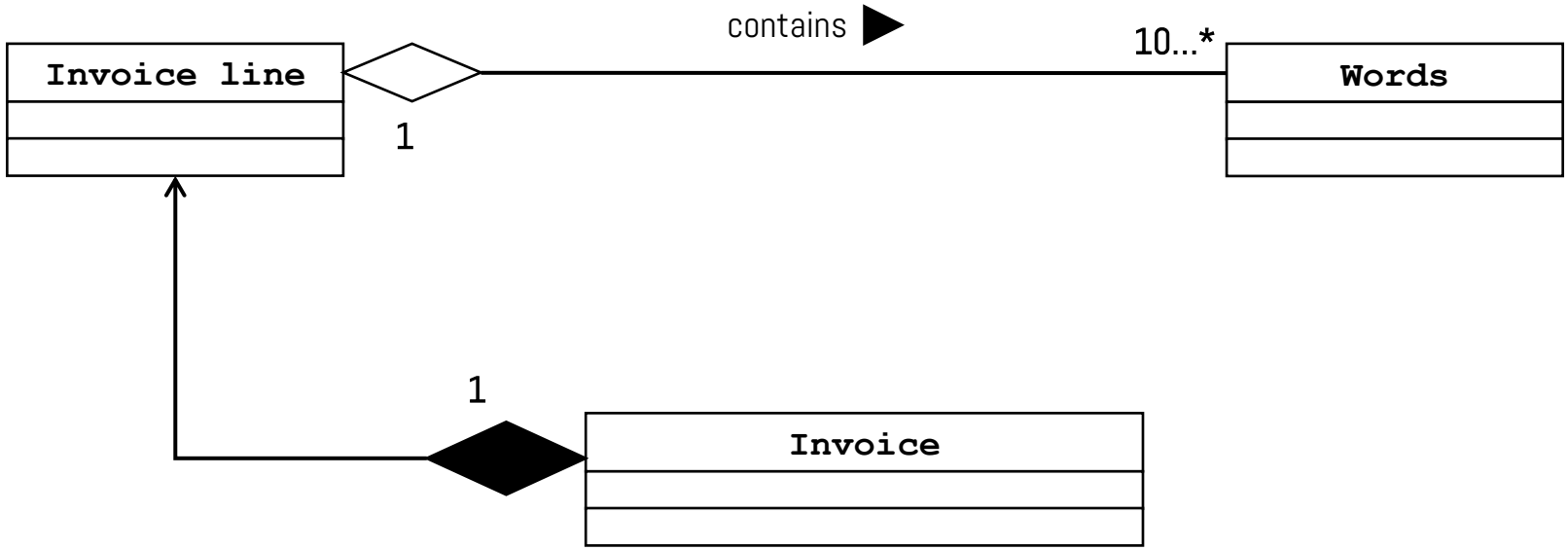
Are strong aggregations

- Indicates that objects of that class are part of objects of another class
- The contained class **doesn't have** has its own lifecycle: only containing object can create and destroy its parts
- Cardinality is 1 (in every instant) – *"Every cost entry can belong only to one invoice"*

Modeled as **filled** rhombus (with cardinality) close to the containing class



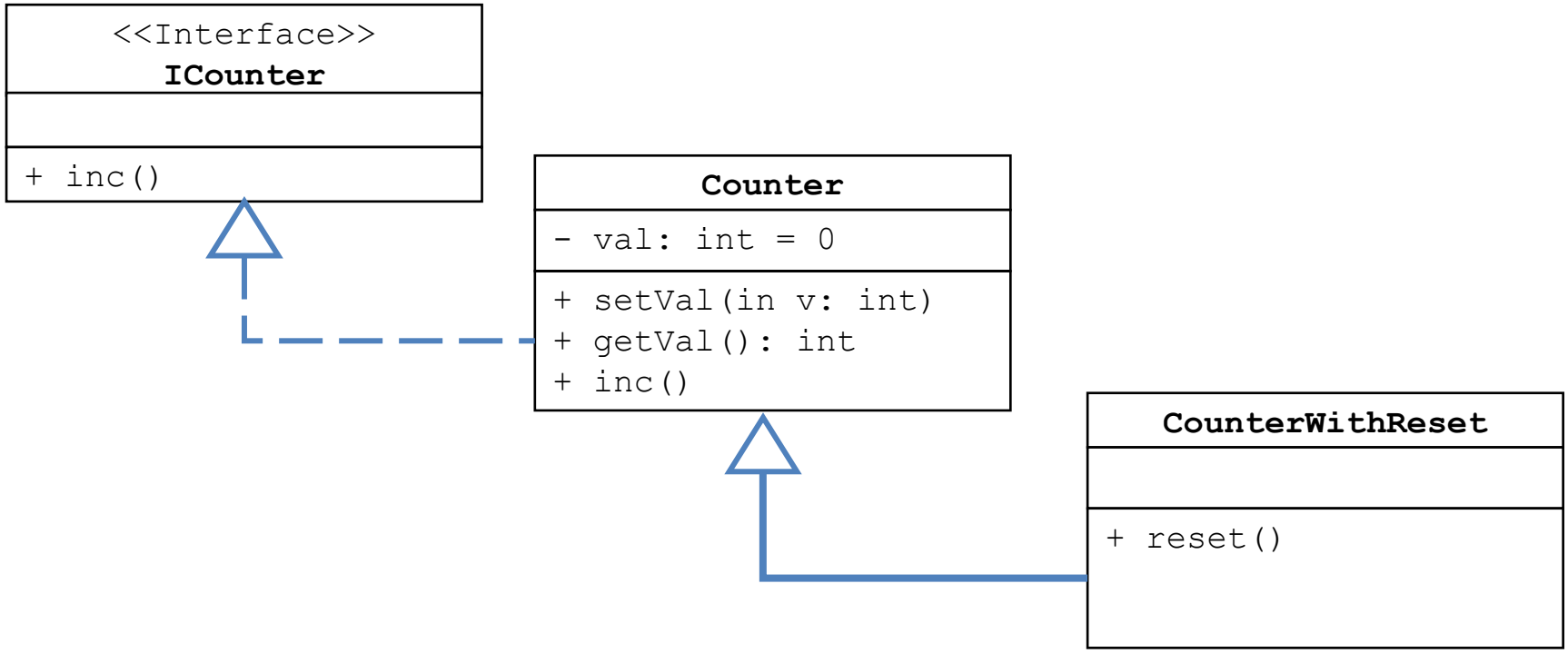
Compositions...and aggregations



Generalization/specialization

The typical relation in OOP

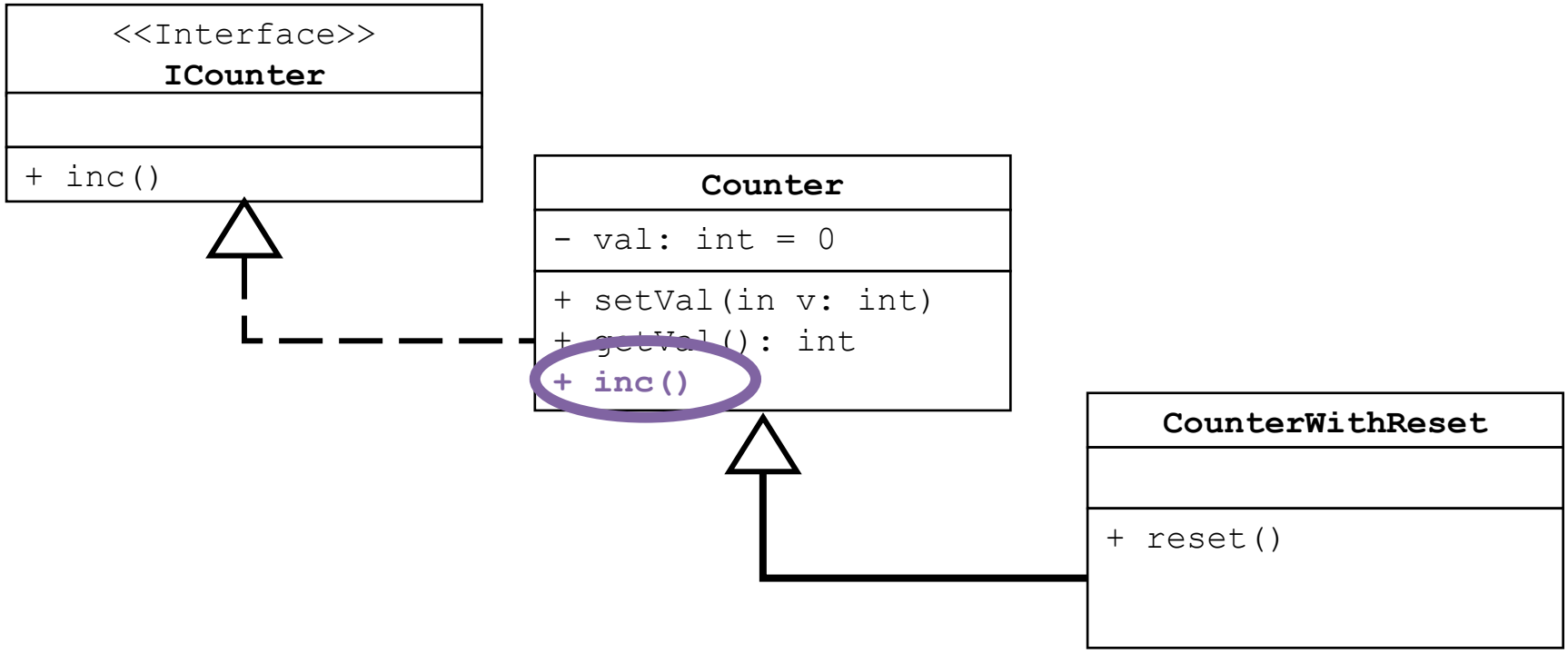
- Models "parent-child" relations, where child class(es) specify (*"override"*) the behavior
- Not limited to OO code!
- Dashed (subclass) or dotted (interfaces) line with empty arrow



Inheritance

Basic principles

- Properties in super-classes are also in sub-classes
- We do not write it (unless we override it)
- Visibility rules apply

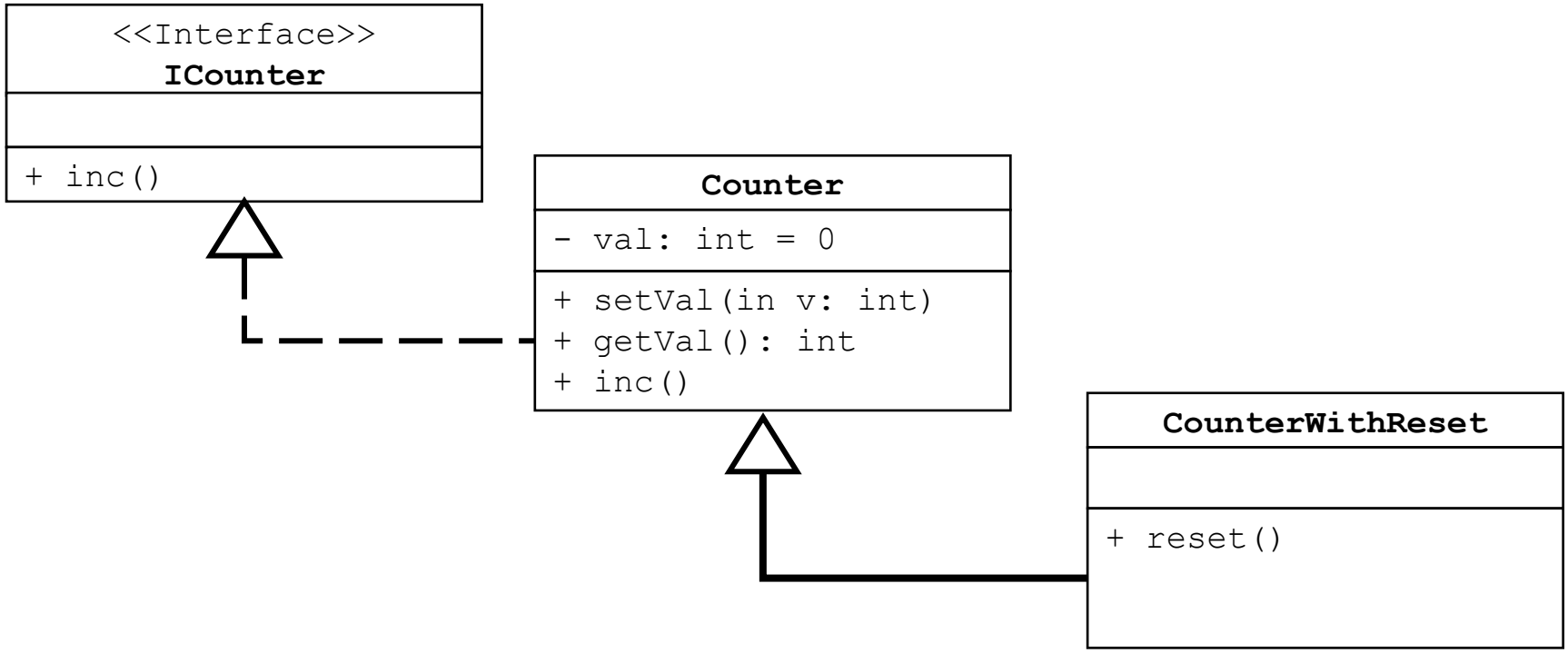


Inheritance

Basic principles

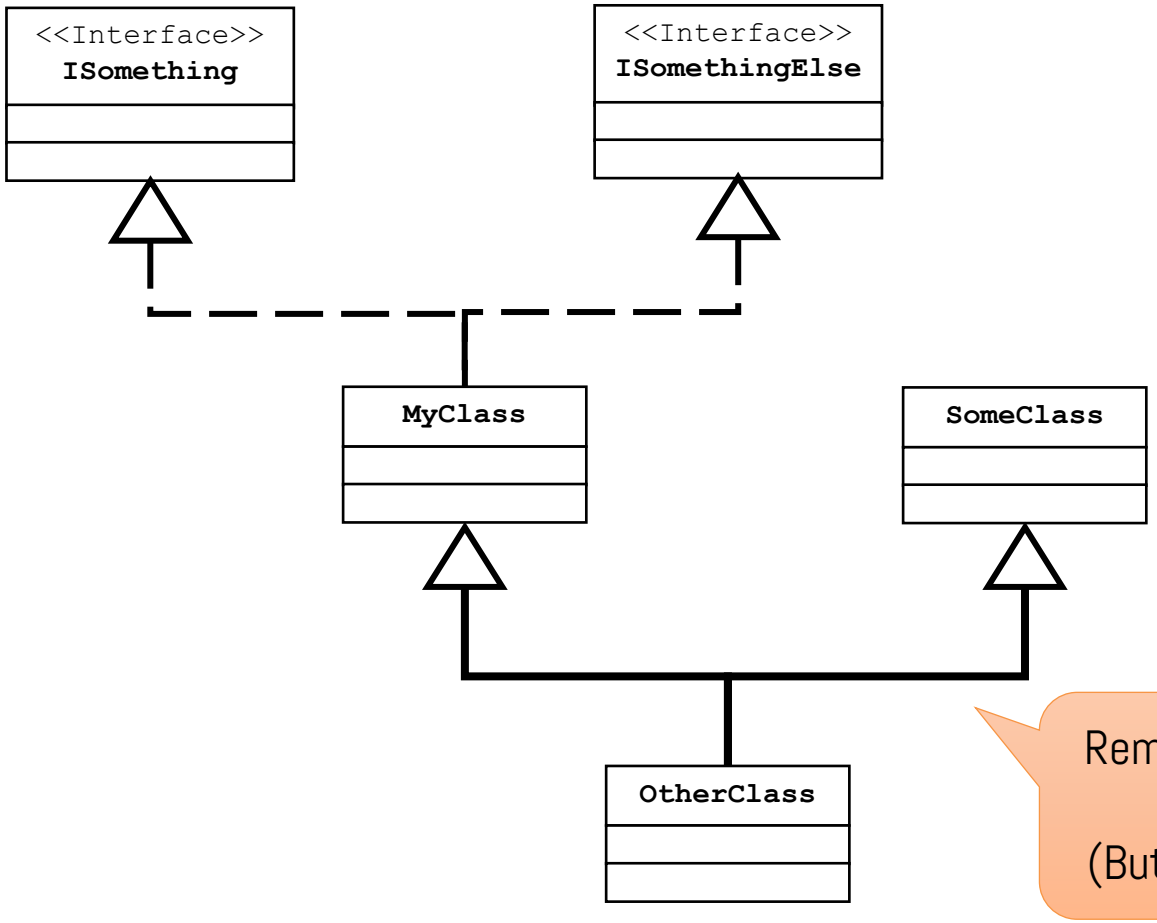
- Properties in super-classes
- We do not write it (unless w
- Visibility rules apply

Super	Sub
Public	Public
Private	Not accessible by subclasses
Protected	Not accessible by other classes, but only by sub-classes



Multiple inheritance

- We can create trees or graphs (res: simple and multiple inheritance)

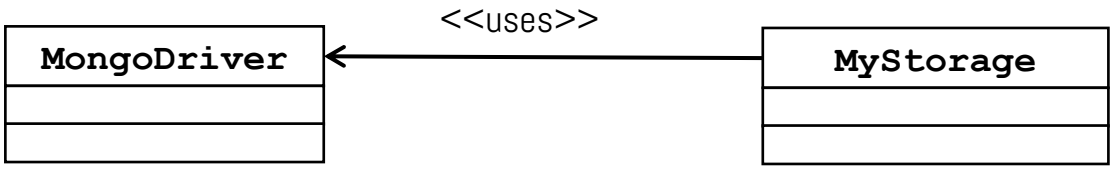


Remember this is not legal in Java, nor in C#!
(But it is in C++ and Python)

Dependency

Semantic relations: one element requires another element

- Models a “client-server” relation
- Dotted line + stereotype *“<<uses>>”* (which can be omitted)



Template classes

Some properties, or ret vals in classes that are generic

- I.e., not specified as we create the class
- We specify them when we create the object
- Useful especially for data structures

Featured in every OO-language with static typing

- Java and C#: `generics`
- C++: `templates`

JS & Python have dynamic typing

- Class properties are simply implemented as dictionaries
- Did you ever notice this?

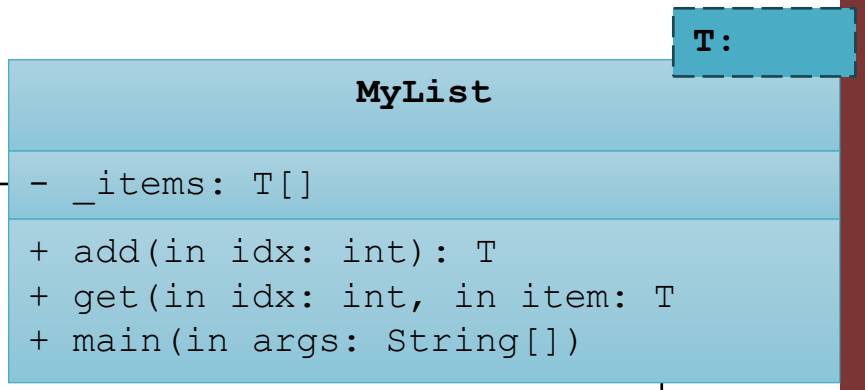
Templates in Java

- C# syntax is nearly identical

```
public class MyList<T> {
  private []T _items;
  public T add(int idx) {
    return this._items[idx];
  }
  public void get(int idx, T item) {
    this._items[idx] = item;
  }


  public static void main(String args[]){
    // Create a list for Strings
    MyList<String> list1 = new MyList<String>();
    list1.add(0, "Alessandro Del Piero");

    // Create a list for Integers
    MyList<Integer> list12 = new MyList<Integer>();
    list2.add(0, 10);
  }
}
```

Templates in UML

- A dotted rectangle, on top-left of the class

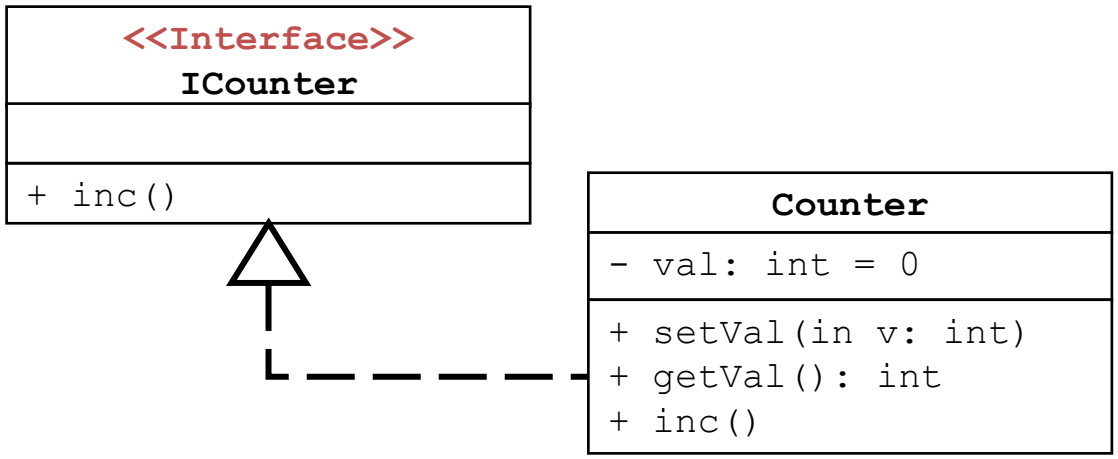



```
public class MyList<T> {
  private []T _items;
  public T add(int idx) {
    return this._items[idx];
  }
  public void get(int idx, T item) {
    this._items[idx] = item;
  }


  public static void main(String args[]){
    // Create a list for Strings
    MyList<String> list1 = new MyList<String>();
    list1.add(0, "Alessandro Del Piero");

    // Create a list for Integers
    MyList<Integer> list12 = new MyList<Integer>();
    list2.add(0, 10);
  }
}
```

Stereotypes

Extend notation with custom concepts

- E.g., `<<Interface>>`
- Each class can have at most 1
- Partly already saw

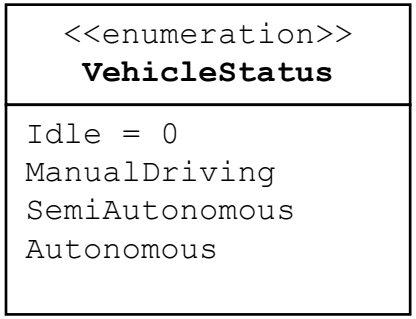


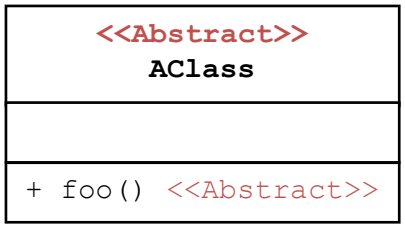
There are even more notations!

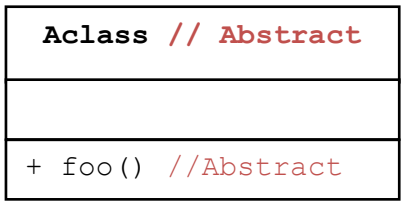
Won't see them here, they simply are too many

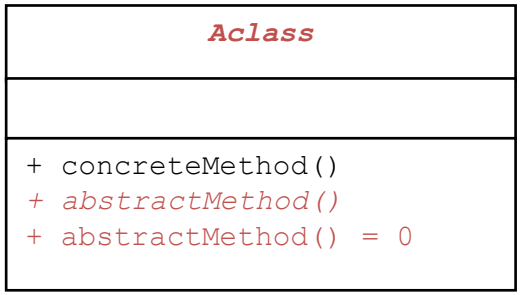
Few relevant

- Enums have the stereotype <<enumeration>>
- Active classes have double vertical borders
  - Classes with their own execution flow
- Abstract classes are a mess!
  - Everyone uses their notation
  - Classes that are only partly implemented
  - Not implemented methods are, in turn, called Abstract methods









Object diagrams

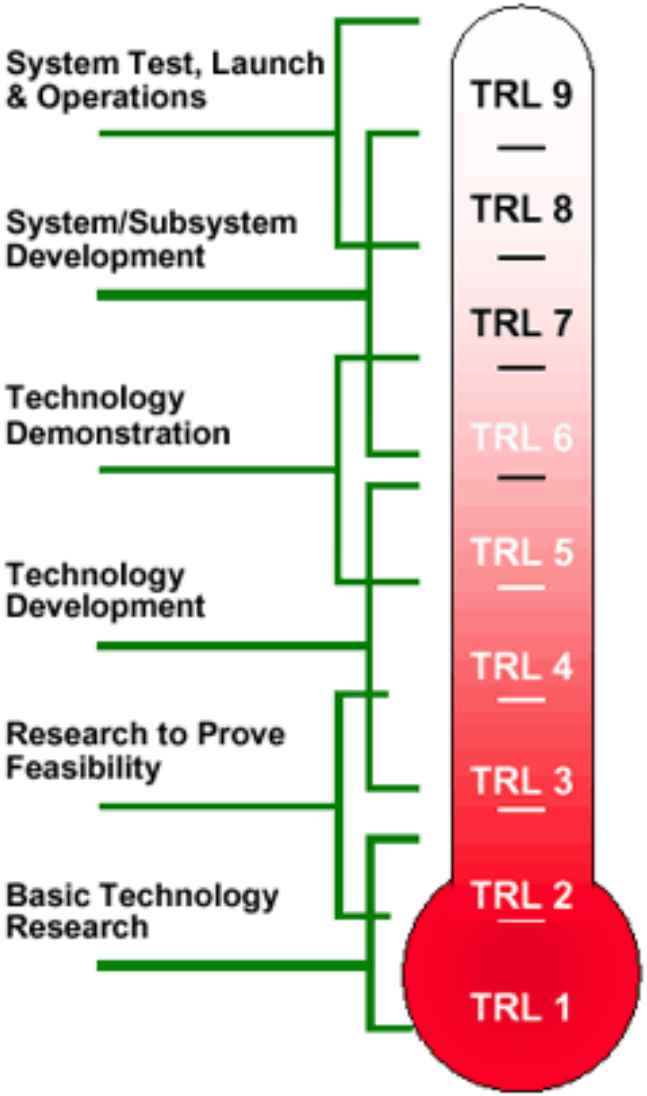
Package diagrams

TRL – Technology readiness level

A Number from 1 to 9 estimating the maturity of a technology

› Developed by NASA in 1970s for the space missions

› **Note**. I will move this in the "03 - Requirements" deck of slides



Assessing TRL of SW

TRL	NASA usage[4]	European Union[5]
1	Basic principles observed and reported	Basic principles observed
2	Technology concept and/or application formulated	Technology concept formulated
3	Analytical and experimental critical function and/or characteristic proof-of concept	Experimental proof of concept
4	Component and/or breadboard validation in laboratory environment	Technology validated in lab
5	Component and/or breadboard validation in relevant environment	Technology validated in relevant environment (industrially relevant environment in the case of key enabling technologies)
6	System/subsystem model or prototype demonstration in a relevant environment (ground or space)	Technology demonstrated in relevant environment (industrially relevant environment in the case of key enabling technologies)
7	System prototype demonstration in a space environment	System prototype demonstration in operational environment
8	Actual system completed and "flight qualified" through test and demonstration (ground or space)	System complete and qualified
9	Actual system "flight proven" through successful mission operations	Actual system proven in operational environment (competitive manufacturing in the case of key enabling technologies; or in space)

› Always remember: it depends on the Operational Scenario! F1/10 example…

References

Course website

- http://hipert.unimore.it/people/paolob/pub/ProgSW/index.html

Book

- I. Sommerville, "Introduzione all ingegneria del software moderna", Pearson
  - Chapter 3

My contacts

- paolo.burgio@unimore.it
- http://hipert.mat.unimore.it/people/paolob/
- https://github.com/pburgio