

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

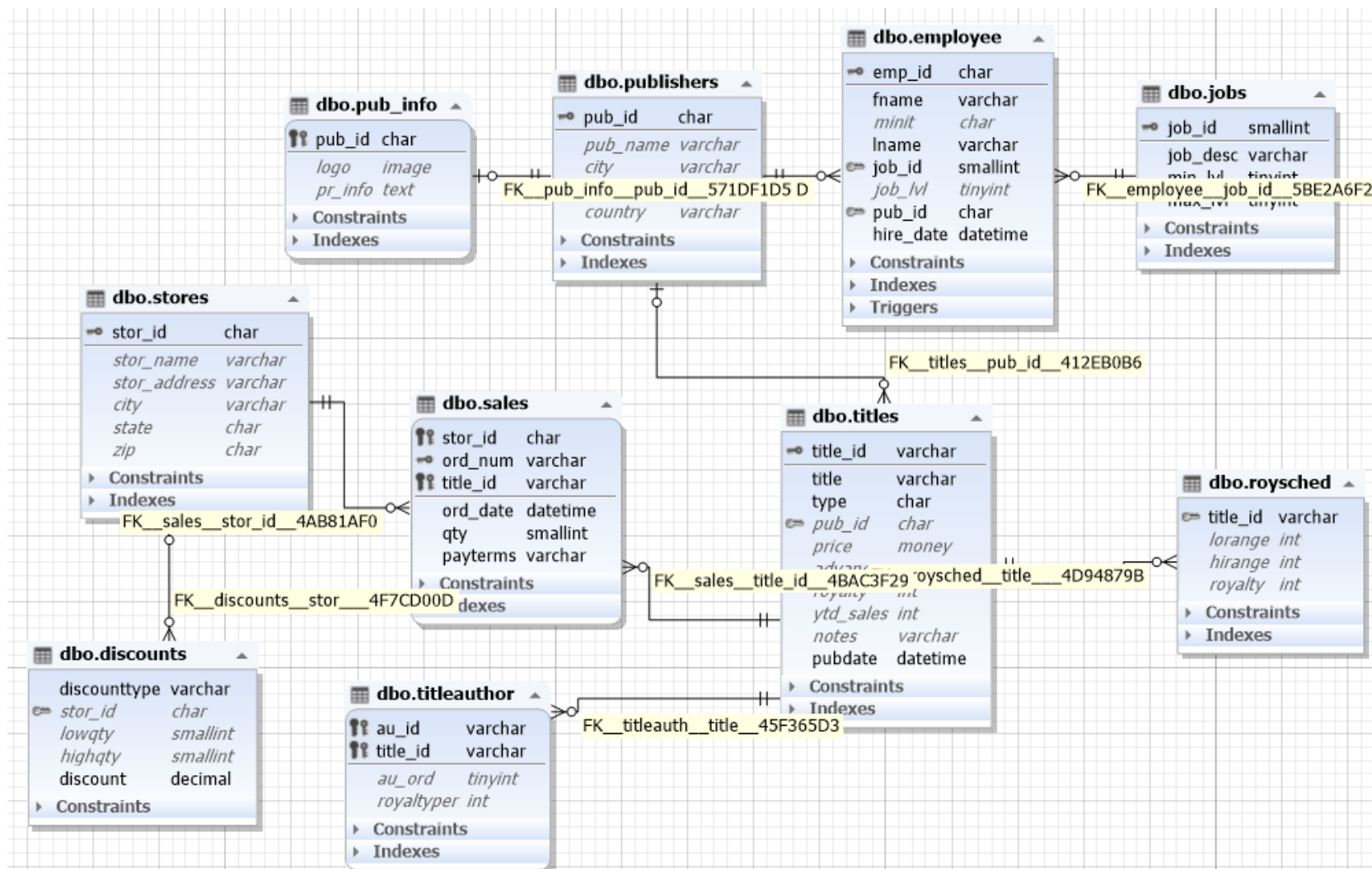
- › The result of our analysis phase
- › Identifies what data is created, and required by business processes
- › Describes system components as blocks, and the relations among them

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

$[ROLLNO, NAME, CLASS, SECTION, ADDRESS]$

$[ROLLNO, NAME, CLASS, SECTION, ADDRESS]$

StudentsDetails

rollno : $\mathbb{P} ROLLNO$

name : $\mathbb{P} NAME$

class : $\mathbb{P} CLASS$

section : $\mathbb{P} SECTION$

address : $\mathbb{P} ADDRESS$

stuname : $ROLLNO \twoheadrightarrow NAME$

stuclass : $ROLLNO \twoheadrightarrow CLASS$

stusection : $ROLLNO \twoheadrightarrow SECTION$

stuadd : $ROLLNO \twoheadrightarrow ADDRESS$

rollno = dom *stuname*

rollno = dom *stuclass*

rollno = dom *stusection*

rollno = dom *stuadd*



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

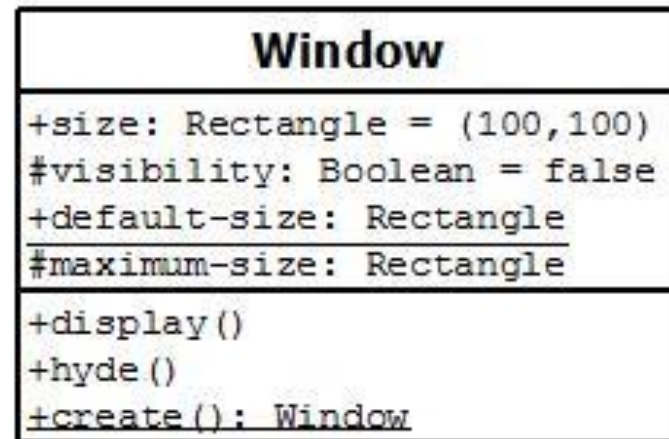
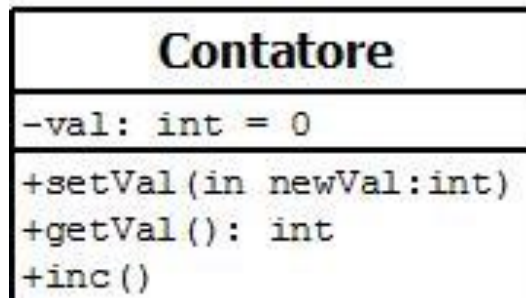
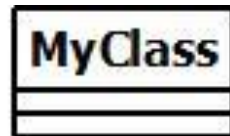
*Why is it so important to define what happens
at compile time vs. what happens at run-time?*



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?

What is an object?

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



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 SOLID...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 domain, and knowledge of our problem
- › Coming from the analysis/system design phase
- › Ultimately, to correctly translating them into a code artifact



Example: how many classes?



› We are forced to identify a proper abstraction level!

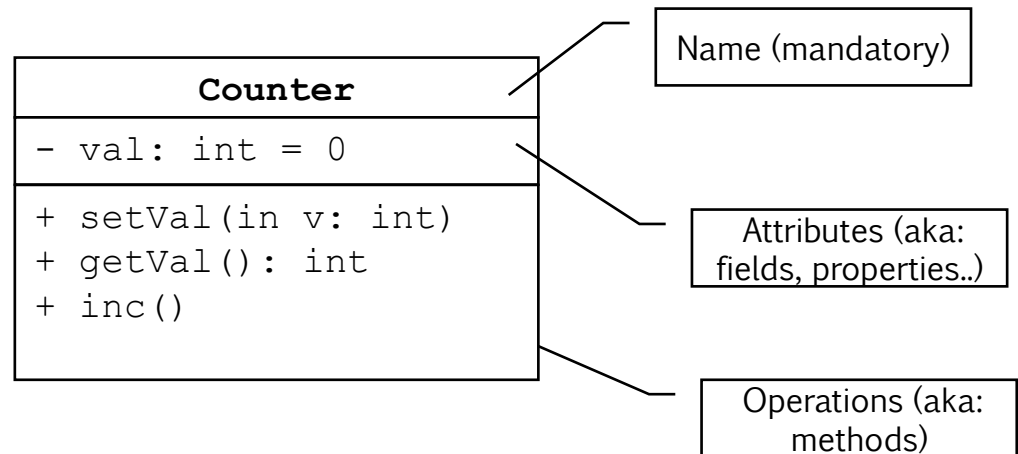


Basic representation

- › “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

Counter
- val: int = 0
+ setVal(in v: int) + getVal(): int + inc()



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 underlined

Counter
- val: int = 0
+ setVal(in v: int)
+ getVal(): int
+ inc()



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

Counter
- <code>val: int = 0</code>
+ <code>setVal(in v: int)</code>
+ <code>getVal(): int</code>
+ <code>inc()</code>



Types of operations

Queries

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

Counter
- val: int = 0
+ setVal(in v: int) + getVal(): int + inc()

Modifiers

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

Window
- size: Rectangle = (100, 100) # visible: Boolean = false + <u>min-size: Rectangle</u> # <u>max-size: Rectangle</u>
+ display() + hide() + <u>create(in size: int): Window</u>

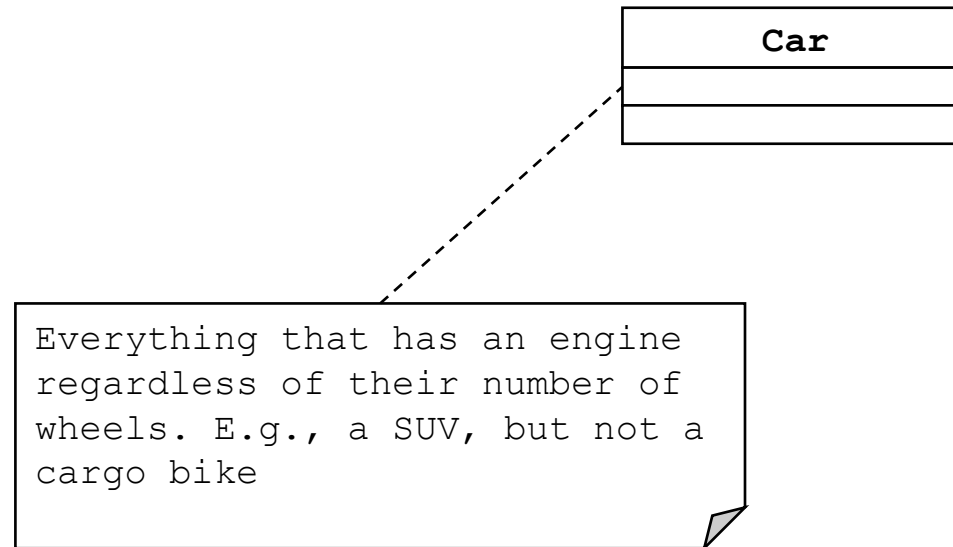
Ctors to create new instances



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

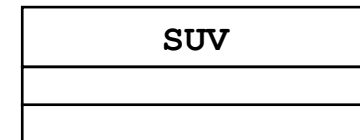
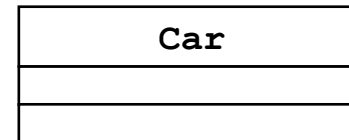
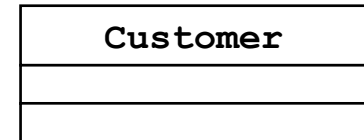
- › Simple, aggregation, and composition

Dependency

- › “Uses”

Generalization/specialization

- › Has to do with inheritance and interfaces





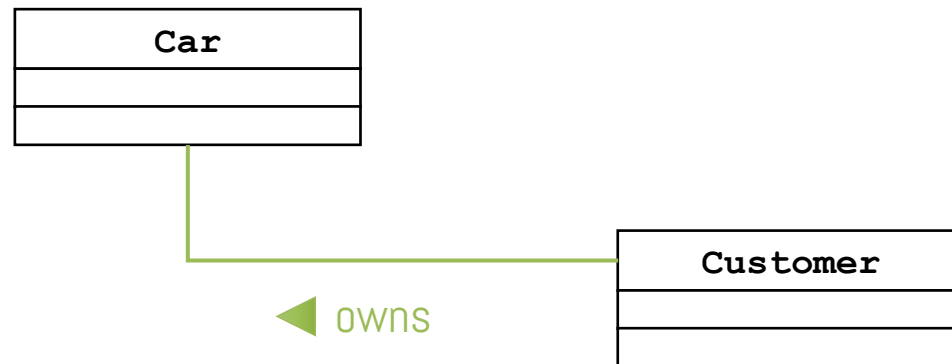
Simple association

A solid line between classes

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

Features

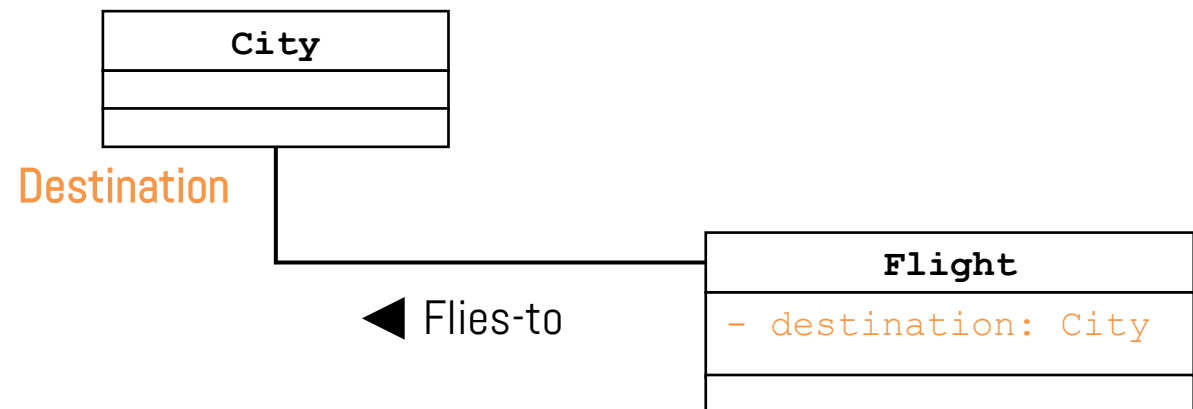
- › Name
- › Roles
- › Cardinality
- › Navigability





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

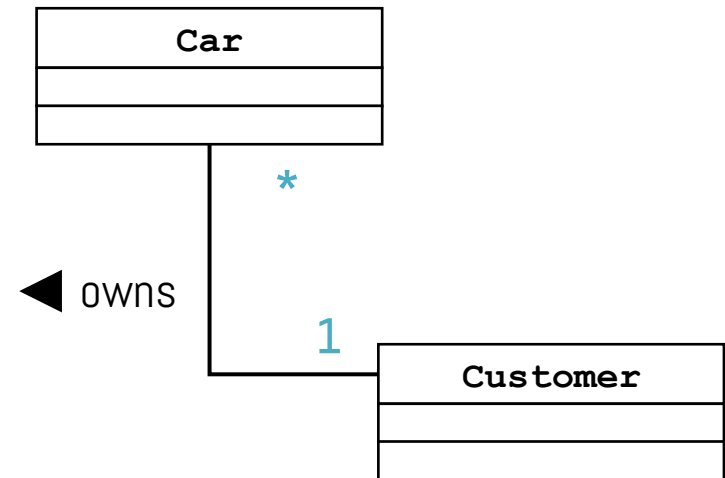
- › Gives an information/bound to the number of objects that can participate to an association
- › Useful information for programmers!!

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

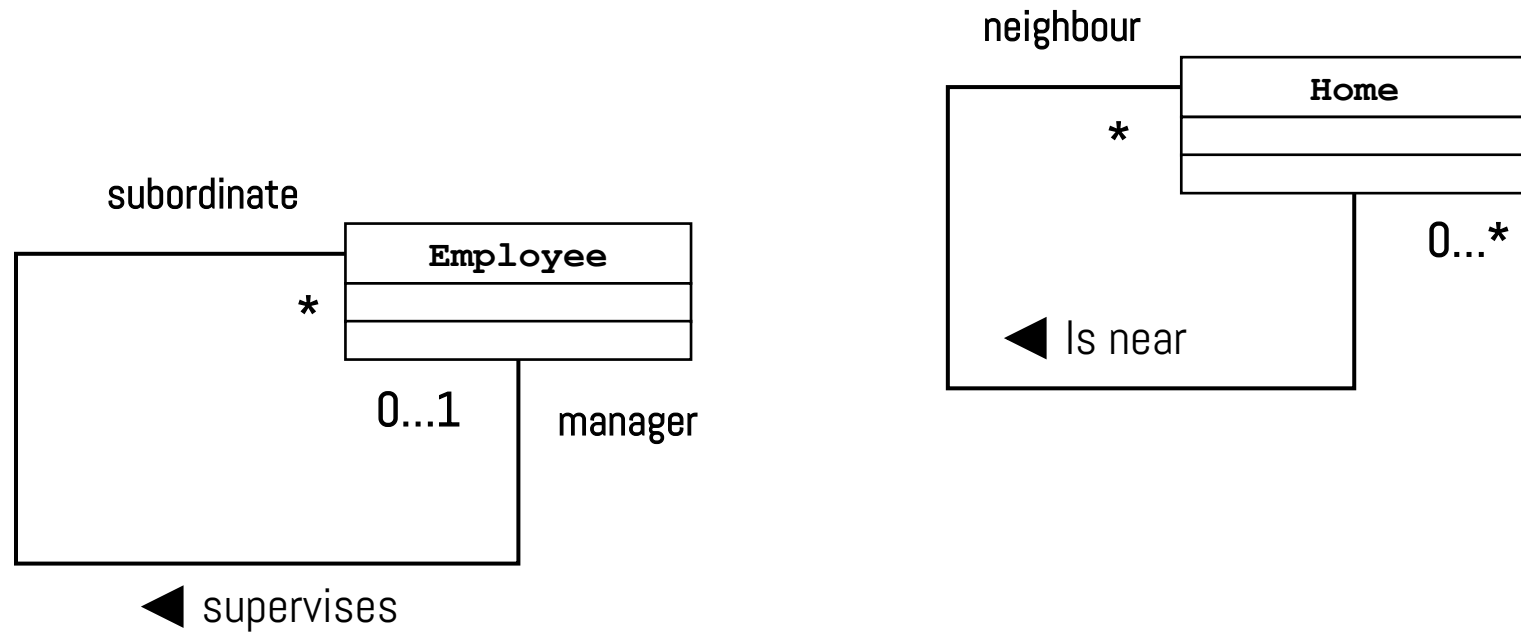
- › Use $*$ to specify any number
- › $*$ and $0...*$ are the same thing
- › Often, $*$ replaced with N





Association: reflective

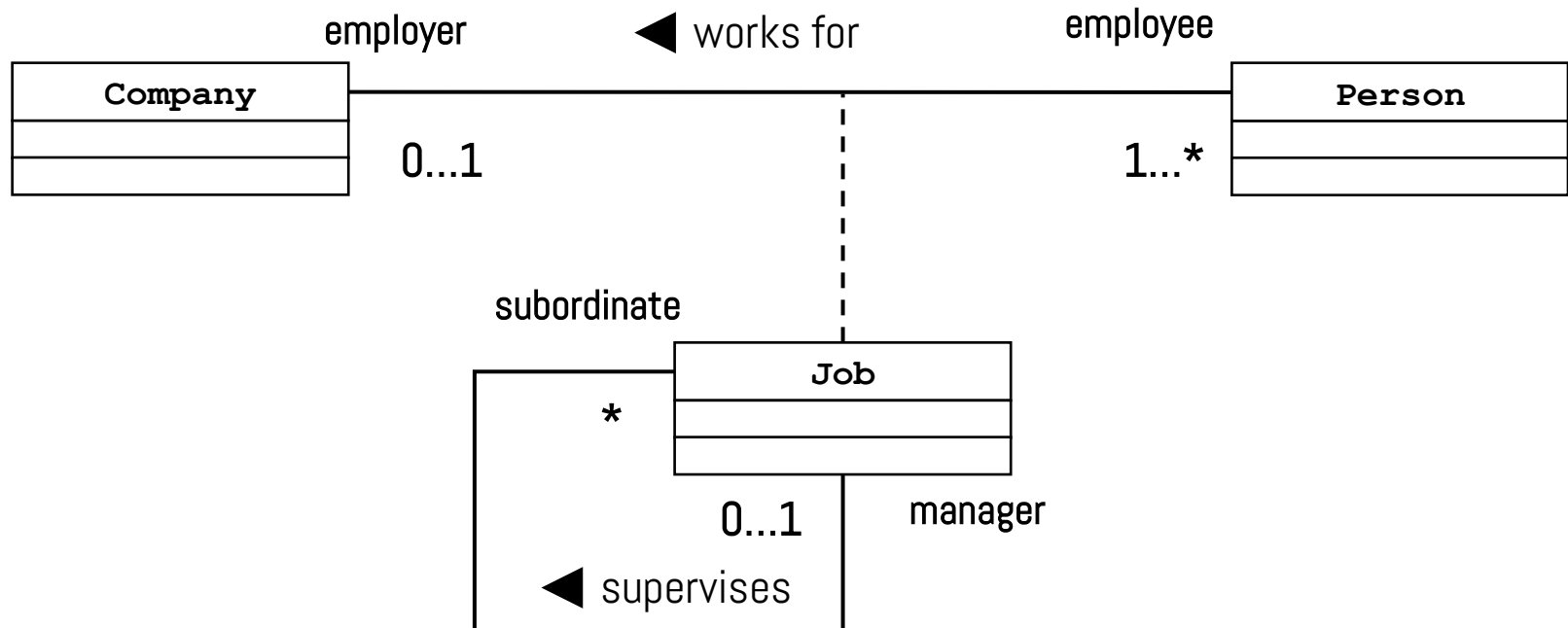
- › In this case, roles are mandatory





Association classes

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





Aggregations

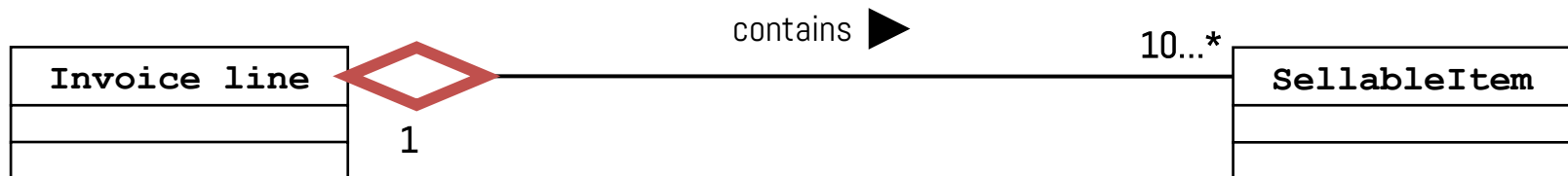
A class that contains another class (logically or even physically)

- › 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 **empty** rhombus (with cardinality) close to the containing class



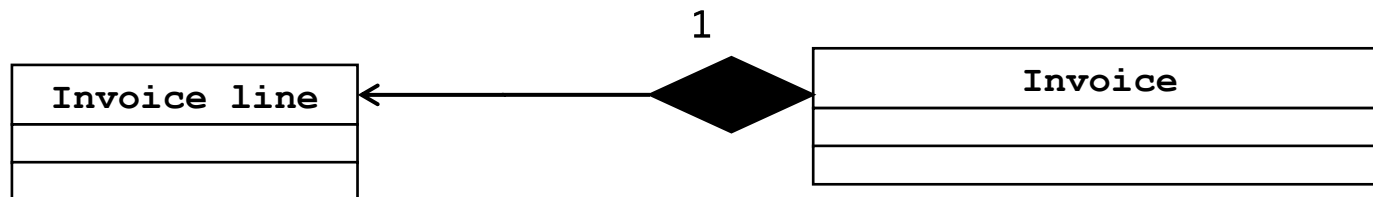


Compositions

Are strong aggregations

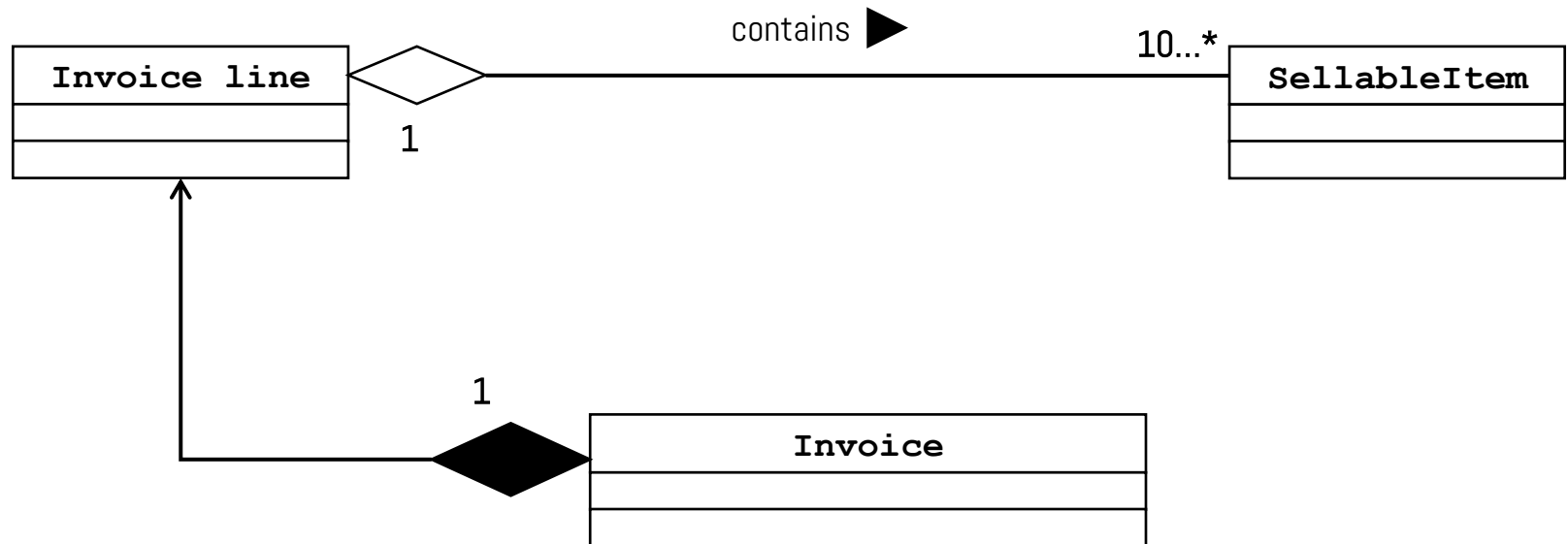
- › Indicates that objects of that class are part of objects of another class
- › The contained class **doesn't have** its own lifecycle: only containing object can create and destroy its parts
- › Cardinality is 1 (in every instant) – *“Every cost entry can belong only to one invoice”*

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





Compositions...and aggregations

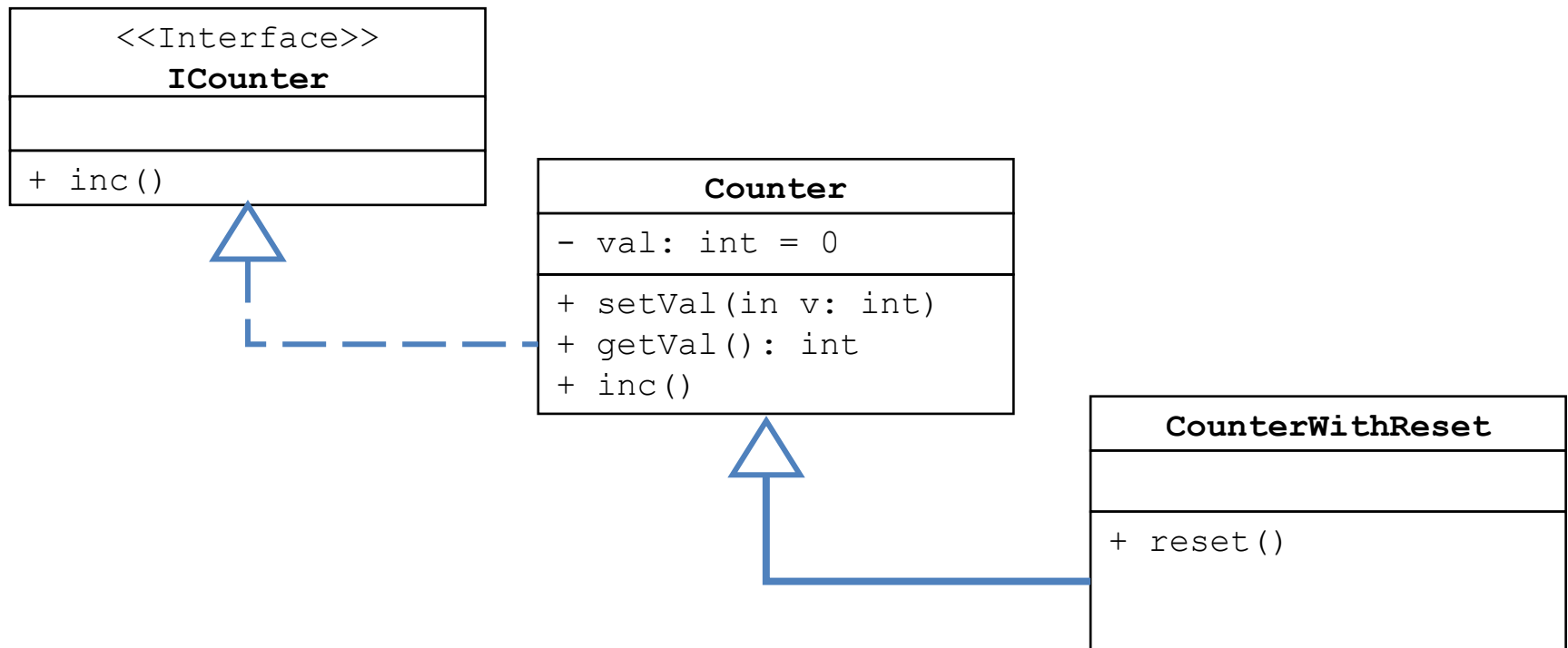




Generalization/specialization

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

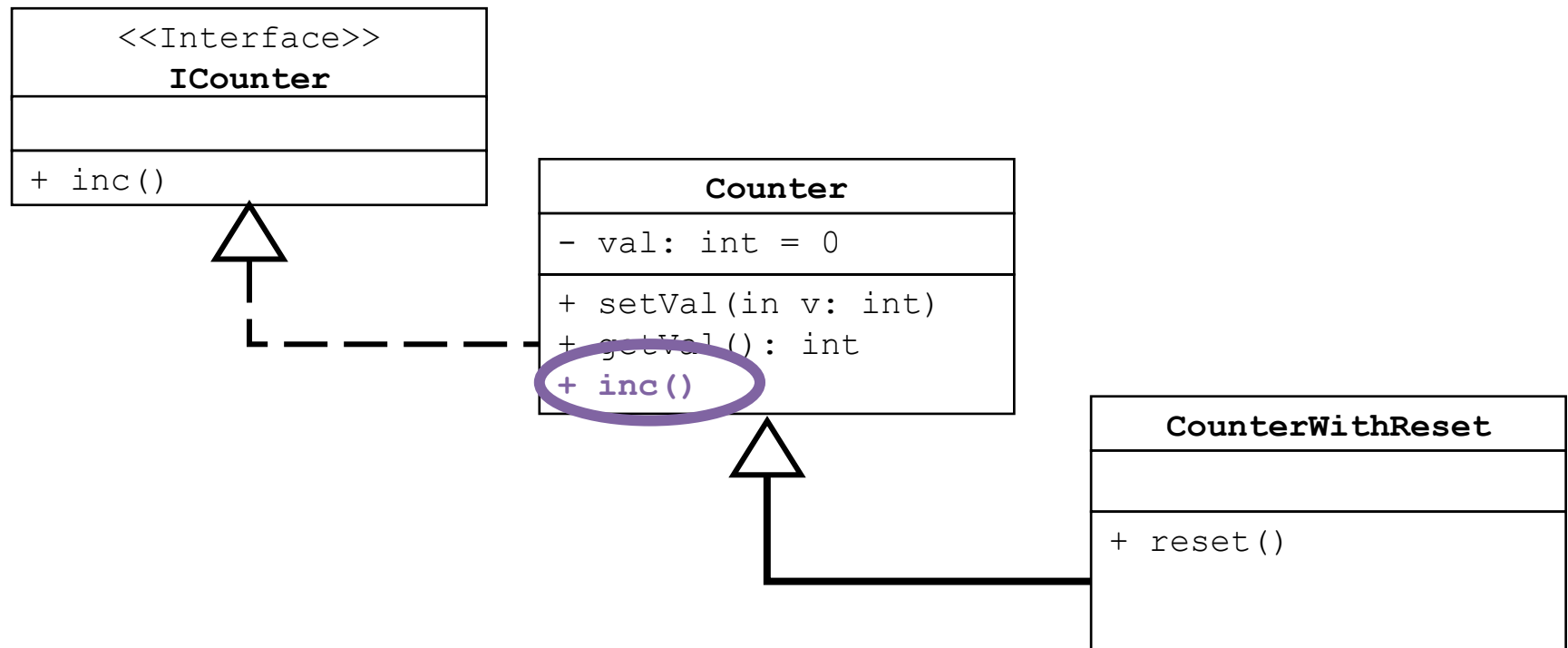




Inheritance

Basic principles

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



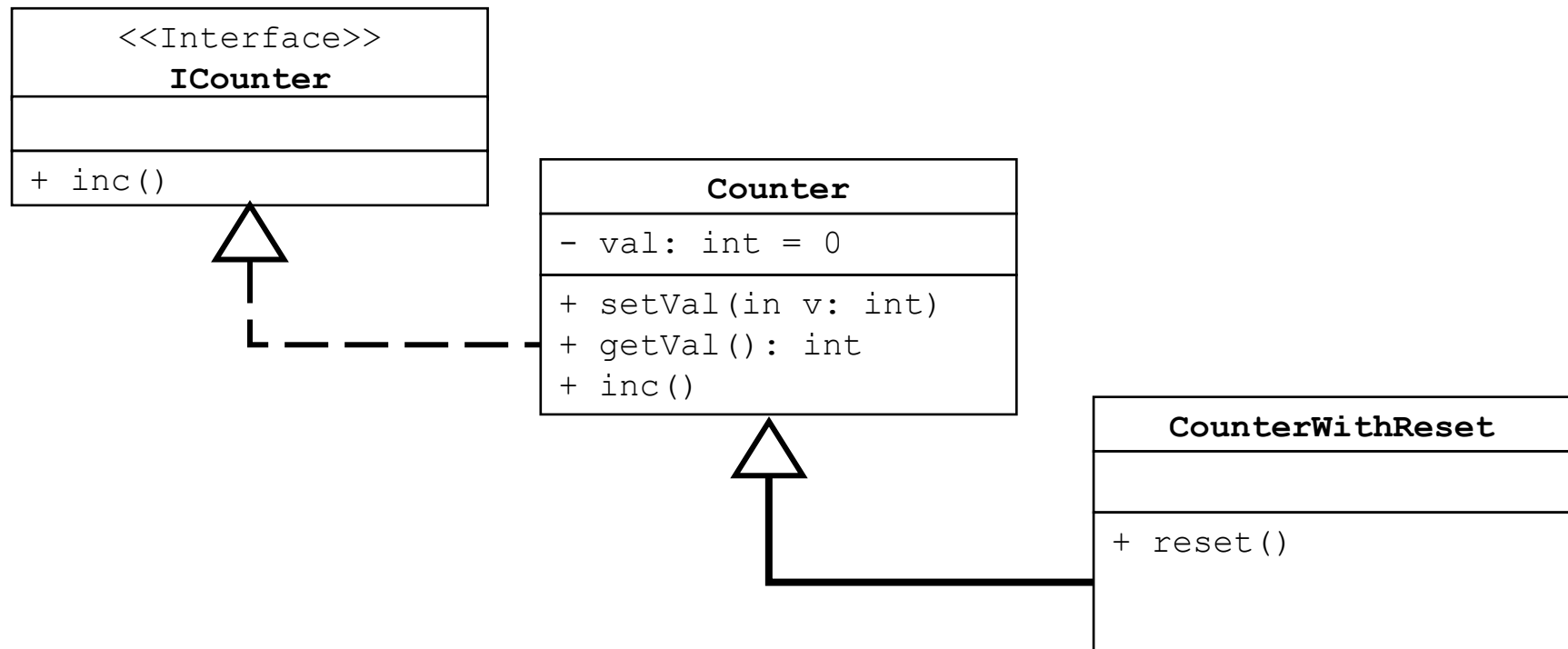


Inheritance

Basic principles

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

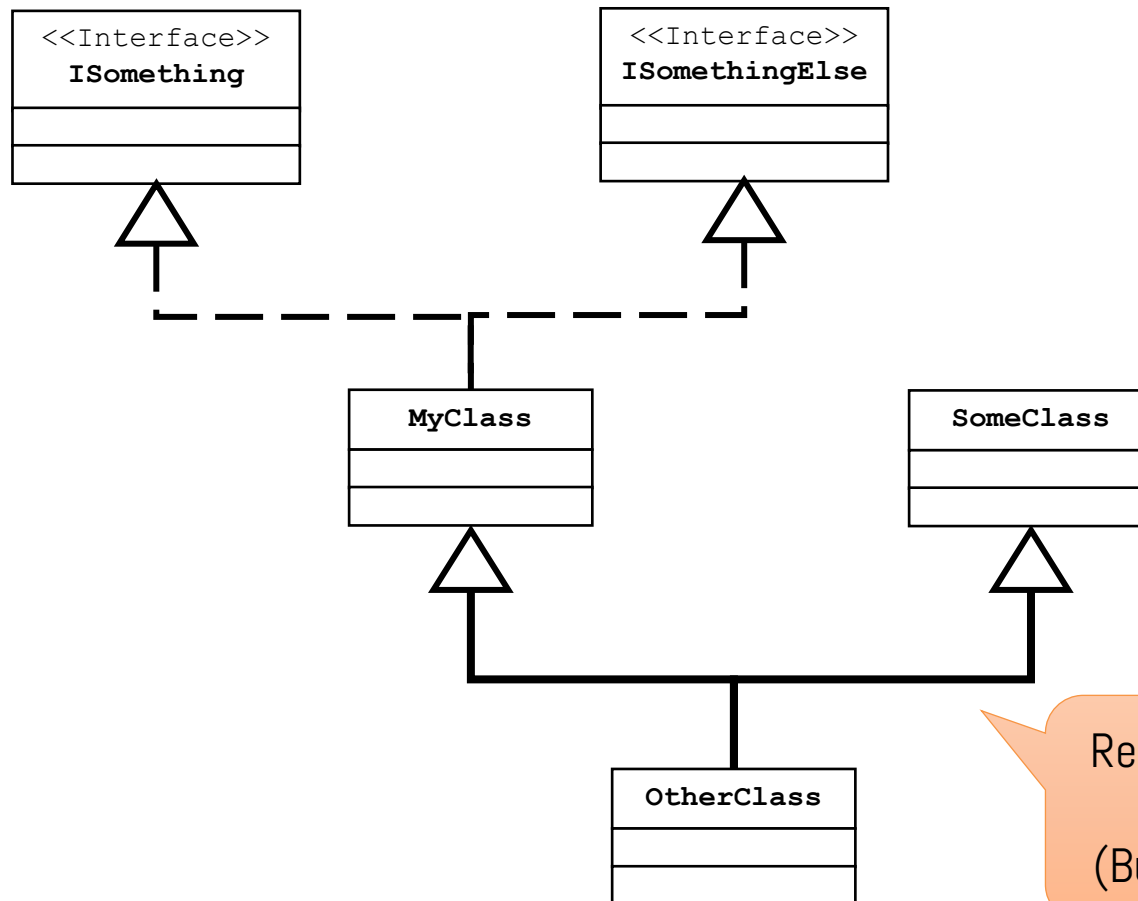
Super	Sub
Public	Public
Private	Not accessible by subclasses
Protected	Not accessible by other classes, but only by sub-classes





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)



The importance of interfaces

- › Enable contract-based interaction between components (i.e., Classes)

```
///  
/// Implement this to enable  
/// serialization  
/// within a framework  
///  
public interface ISerializable {  
    void serialize();  
}  
  
///  
/// Basic counter functionality  
///  
public interface ICounter {  
    void inc();  
}  
  
///  
/// Implement this to enable printing  
///  
public interface IPrintable {  
    void print();  
}
```

```
void serializeAndSendViaSerial(ISerializable s) {  
    Serial.send(s.serialize());  
}  
  
void foo(ICounter c) {  
    c.inc();  
}  
  
void print(IPrintable obj) {  
    obj.print();  
}
```

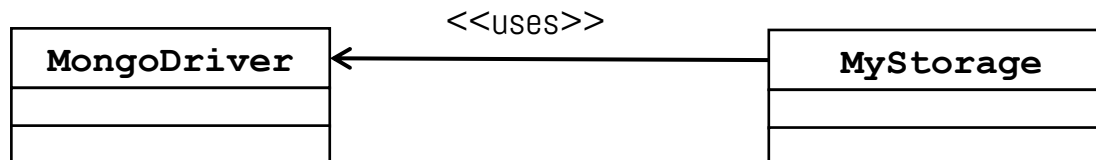
```
public static void main() {  
    Counter c = new Counter();  
  
    foo(c);  
  
    serializeAndSendViaSerial(c);  
  
    print(c);  
}
```



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> list2 = 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> list2 = new MyList<Integer>();  
        list2.add(0, 10);  
    }  
}
```

T:

MyList

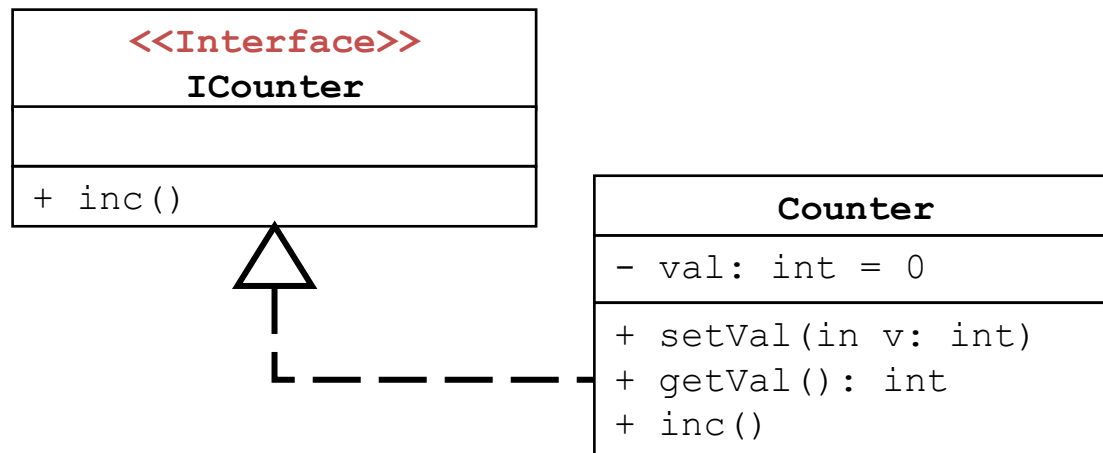
```
- _items: T[]  
  
+ add(in idx: int): T  
+ get(in idx: int, in item: T  
+ main(in args: String[])
```



Stereotypes

Extend notation with custom concepts

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



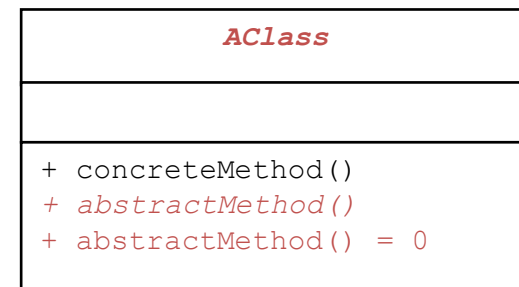
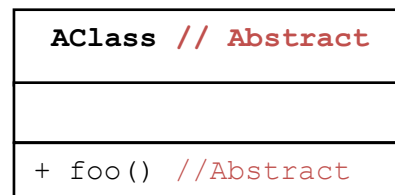
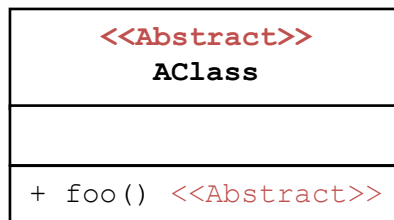
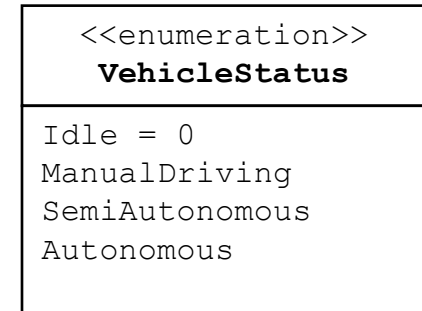


There are even more notations!

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



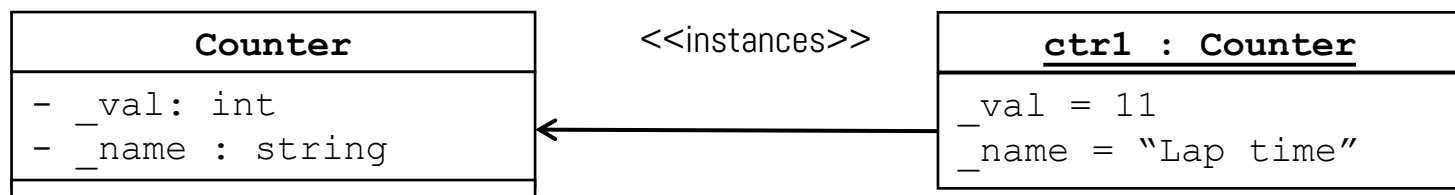
Objects un UML

Instances of classes

- › At run-time, they store the status of given (atomic) entities of our model/representation
 - But not static properties...do you remember the difference?
- › And...the methods to access to data
- › At least, they give us the chance to do so...

Notation is similar to the class diagram

- › Also here, language-specific notations/conventions might apply
- › Underlined

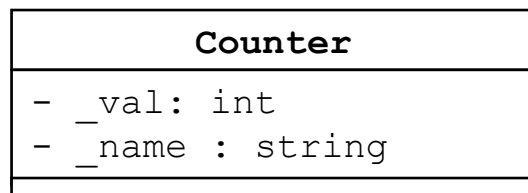




Objects in UML

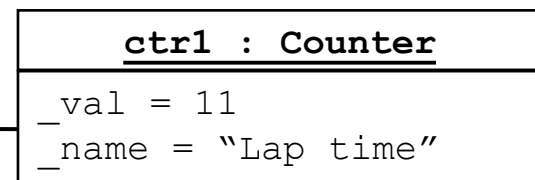
Find the difference

Static



<<instances>>

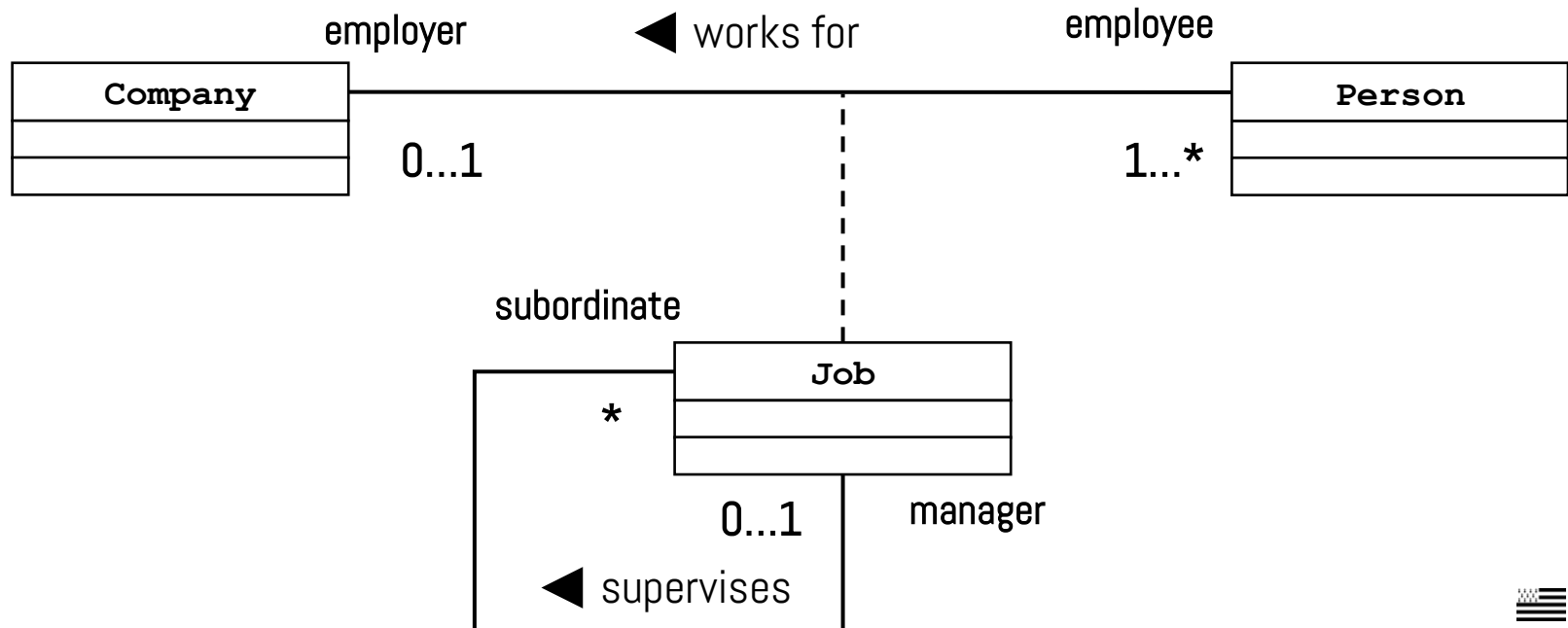
Dynamic





Relations between objects

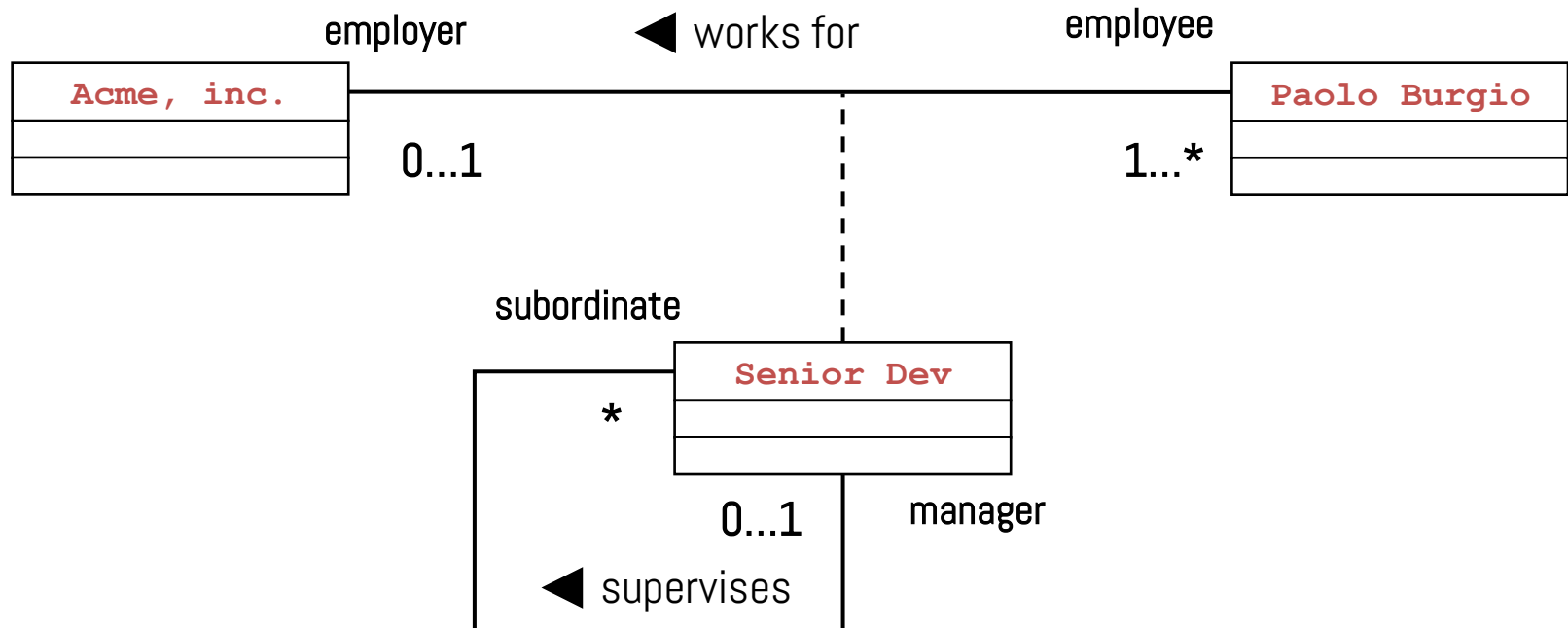
- › More or less the same than the ones between classes





Relations between objects

- › More or less the same than the ones between classes





Package diagrams



What is a “package”

...ok, we are experts in Java..

- › We can **group** our entities (i.e., classes) to structure our code
- › Follows up by *divide-et-impera*

This is not so simple as it seems!

- › Which classes do we group together? What are our “semantic boundaries”
- › Do we group them by functionality?
- › This is a design choice

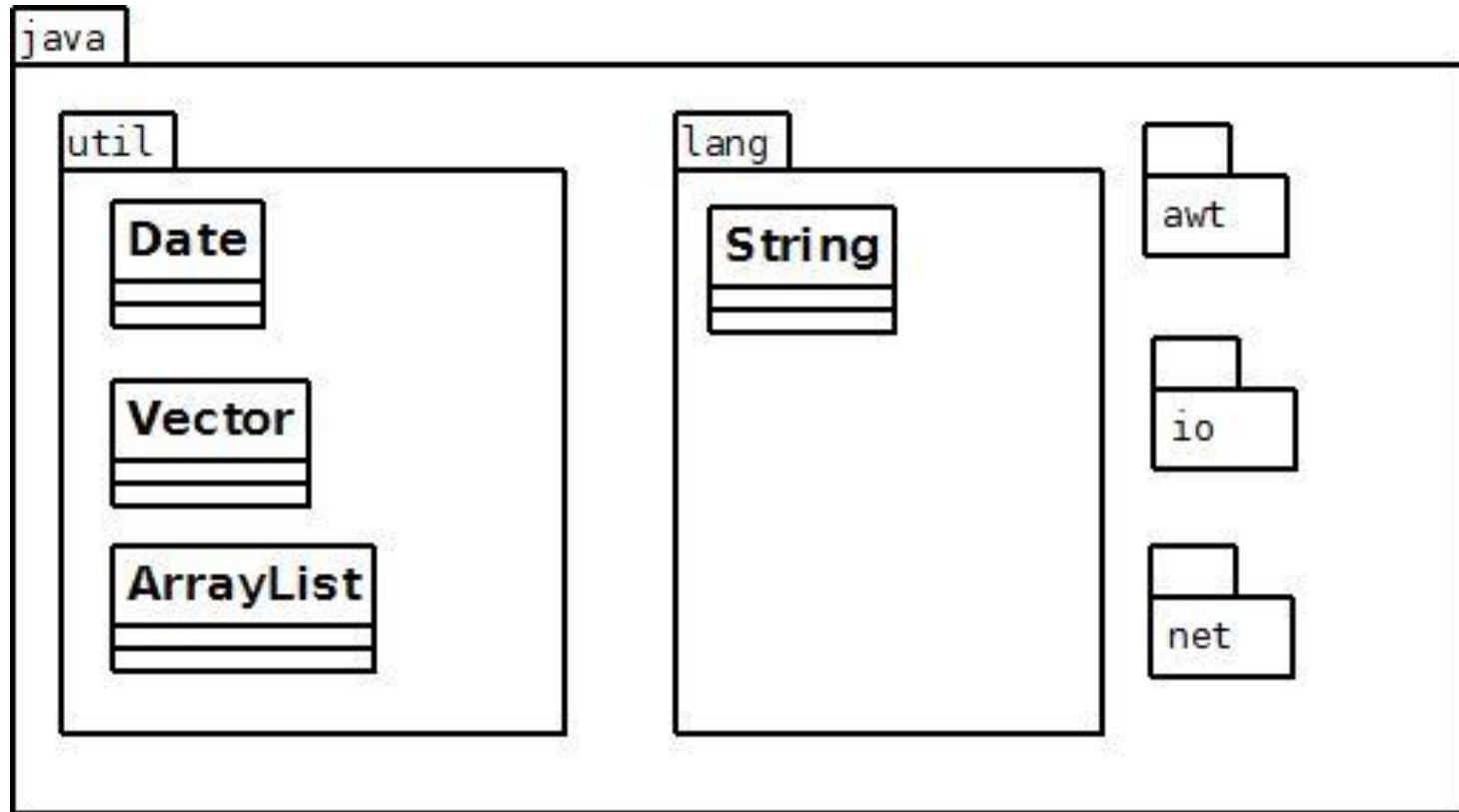
Practical example

- › Java is “nice at us”, as it forces us mapping packages on folders and sub-folders
- › C# gives us more freedom...but also more responsibilities



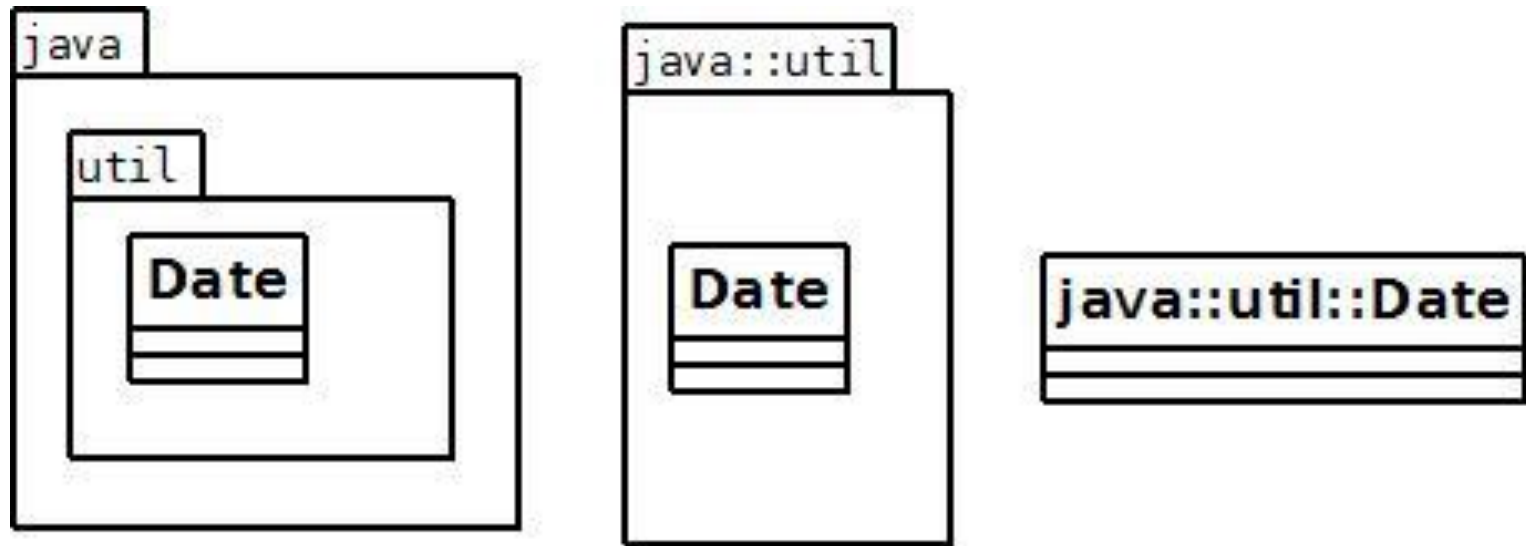


Practical example in Java



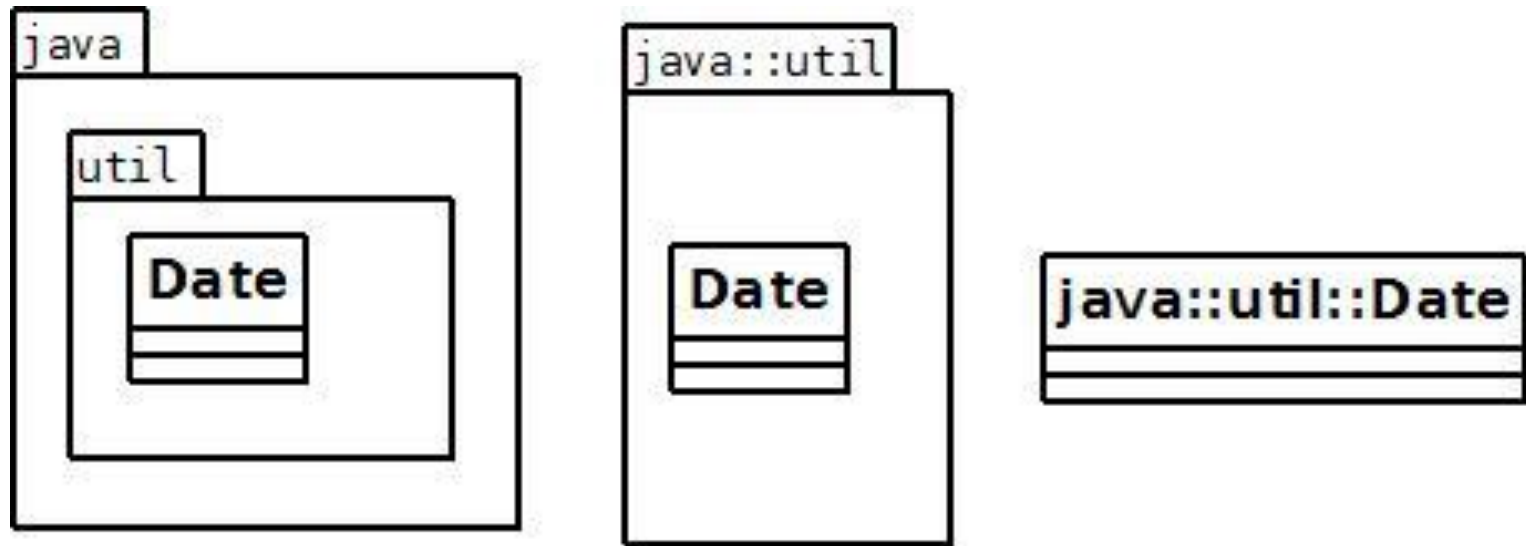


Nested packages, and class diagram





Visibility recap



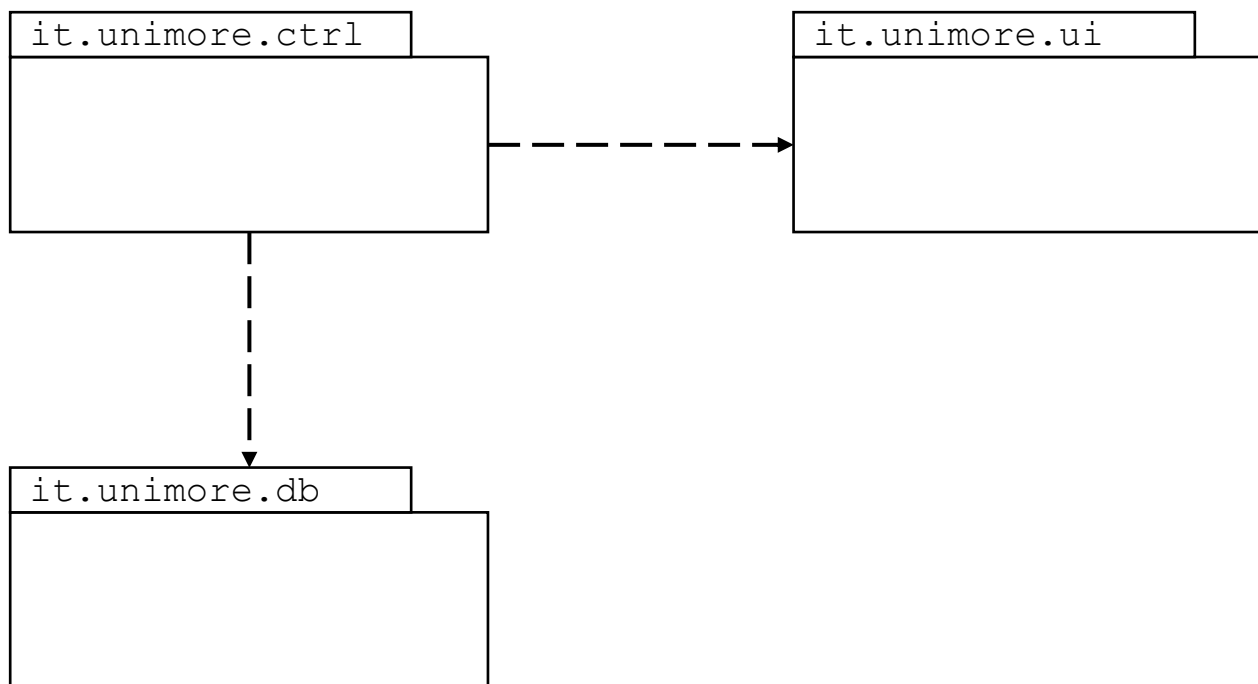
- › There is only one “`util`” package within the “`java`” package
- › There is only one “`Date`” class within the “`java::util`” package
- › There is only one “`java::util::Date`” class



Dependencies between packages

Dotted lines

- › `<<uses>>` (default, omitted)
- › Here, MVC as an example





Dependencies between packages

Usage

- › `<<uses>>` (default)
- › Client-server relation between packages

Import

- › `<<imports>>`
- › Provider package namespace becomes part of client package namespace

Access

- › `<<accesses>>`
- › Elements from client package can access elements of provider package
- › E.g., `friend` classes
- › Recap: these are **not** transitive



Recap (...?) Friend classes

Methods from friend class can access private fields and member of target class

- › Target class explicitly declares this
- › (...yes, it's a design pattern...)
- › Disclaimer: **they only exists in C++**



Recap (...?) Friend classes

```
#include <iostream>
using namespace std;

class Goo
{
    private:
        int private_variable;
    public: Goo() { private_variable = 10; }

    // friend class declaration
    friend class Foo;
};

class Foo
{
    public:
        void display(Goo& t)
        {
            cout << "The value of Private Variable = "
                 << t.private_variable << endl;
        }
};

int main()
{
    Goo g;
    Foo fri;
    fri.display(g);
    return 0;
}
```



Visibility between packages

For elements (classes, class methods...) within packages

- for private

+ for public

~ for package

for protected (visible only to child packages)

References



Course website

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

My contacts

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