OO Paradigm and UML

Object Oriented Programming

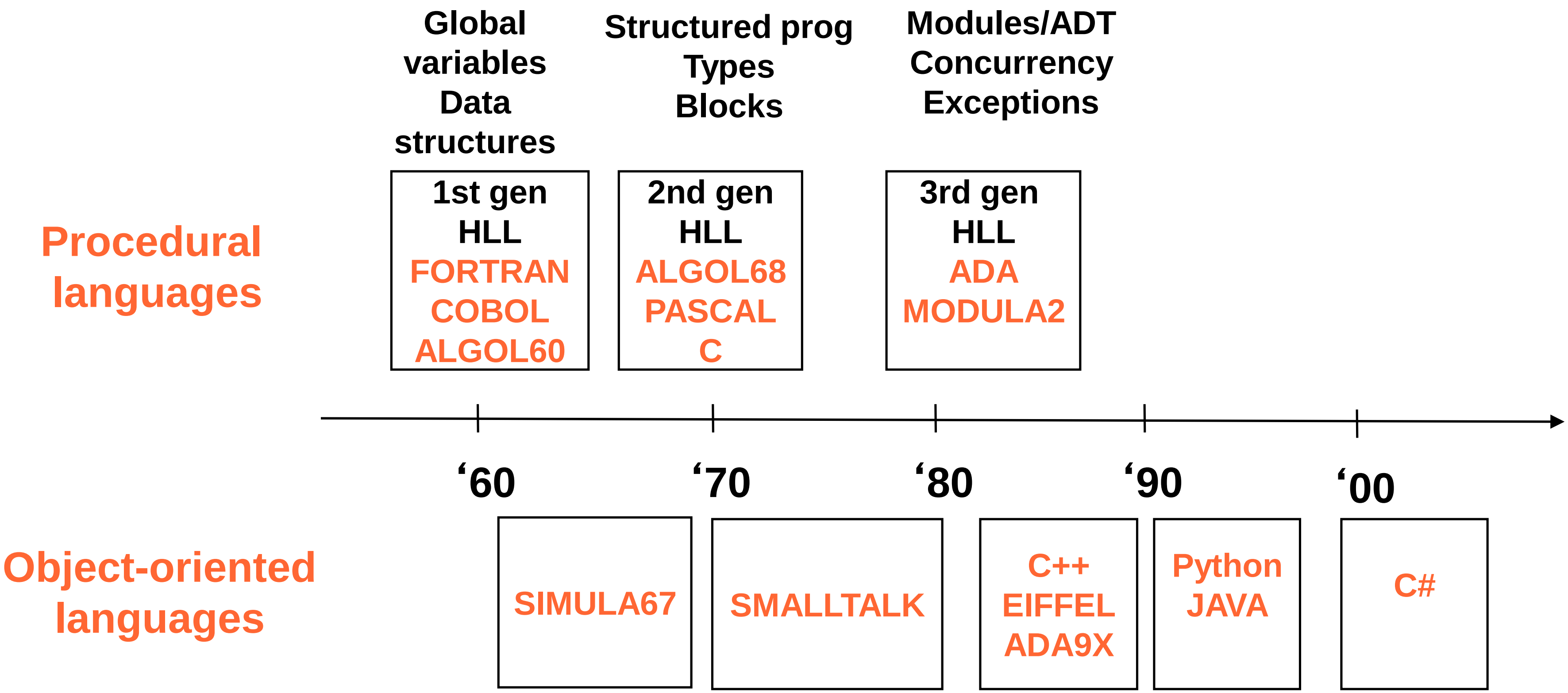
http://softeng.polito.it

Licensing Note

Programming paradigms

- Procedural (Pascal, C,…)
- Object-Oriented (C++, Java, C#,…)
- Functional (LISP, Haskell, SQL,…)
- Logic (Prolog)

Languages timeline



Procedural languages

Global variables Data structures

1st gen HLL

FORTRAN COBOL ALGOL60

Structured prog Types Blocks

2nd gen HLL

ALGOL68 PASCAL C

Modules/ADT Concurrency Exceptions

3rd gen HLL

ADA MODULA2

‘60 ‘70 ‘80 ‘90 ‘00

Object-oriented languages

SIMULA67

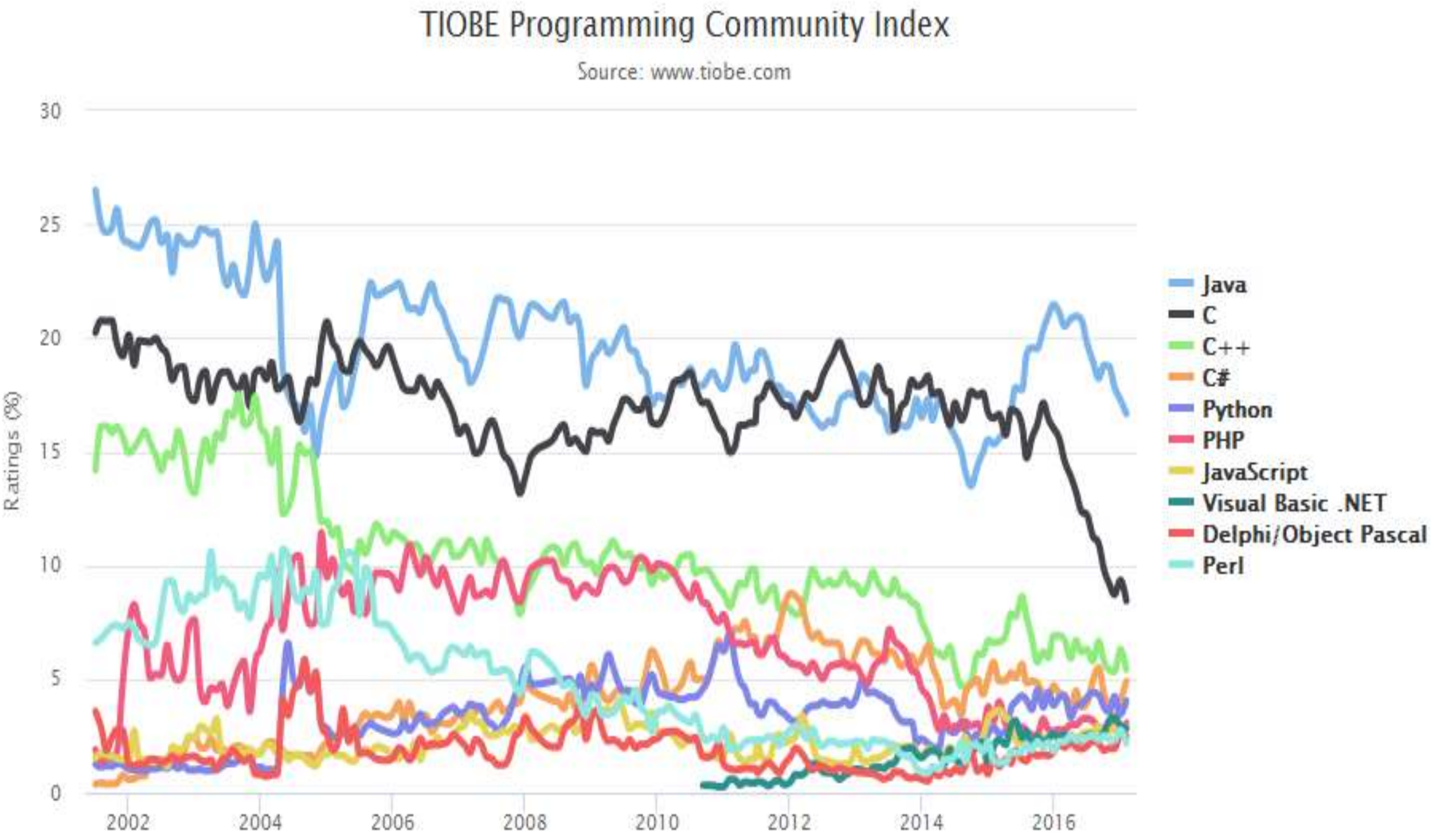
SMALLTALK

C++ EIFFEL ADA9X

Python JAVA

C#

Popularity of Languages



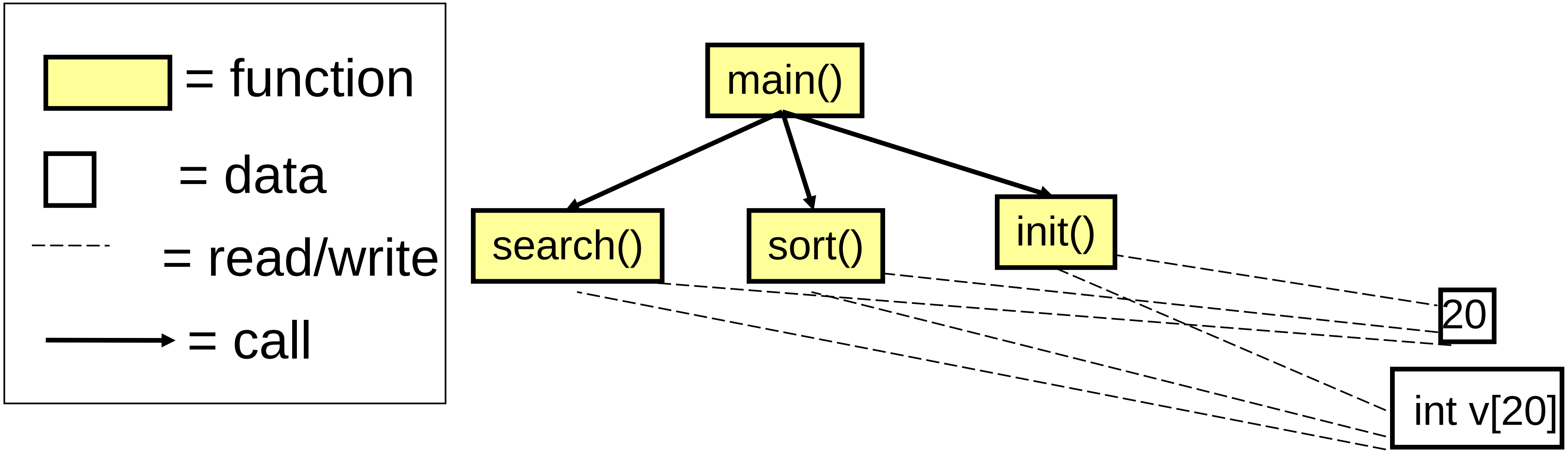
Procedural

```
int vect[20];
void sort() { /* sort */ }
int search(int n){ /* search */ }
void init() { /* init */ }
// …
void main(){
  int i;
   init();
   sort();
   i = search(13);
}
```

Modules and relationships

- Modules:
  - Data
  - Function (Procedure)

- Relationships
  - Call
  - Read/write



Problems

- There is no syntactic relationship between:
  - Vectors (int vect[20])
  - Operations on vectors (search, sort, init)

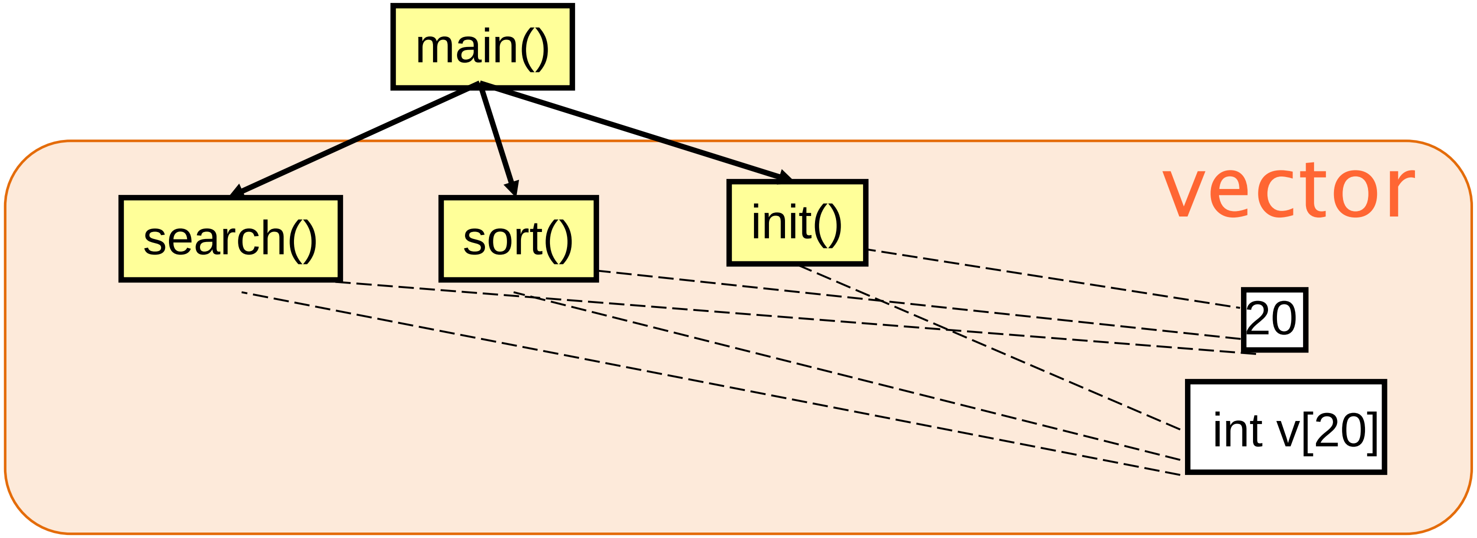
- There is no control over *size*:

```
for (i=0; i<=20; i++) {  vect[i ]=0; };
```

- Initialization
  - Actually performed?

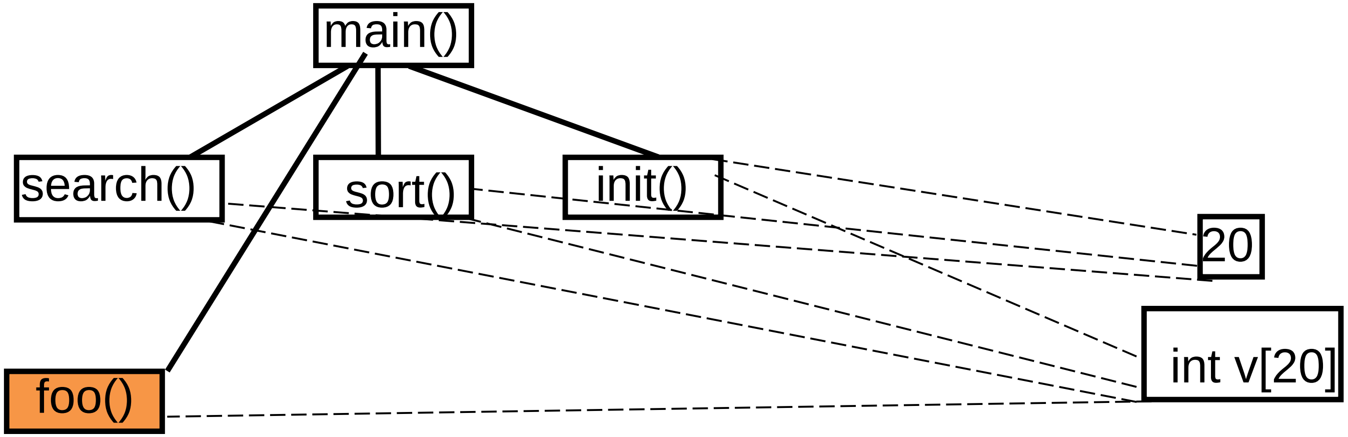
The vector

- It’ s not possible to consider a vector as a primitive and modular concept
- Data and functions cannot be modularized properly



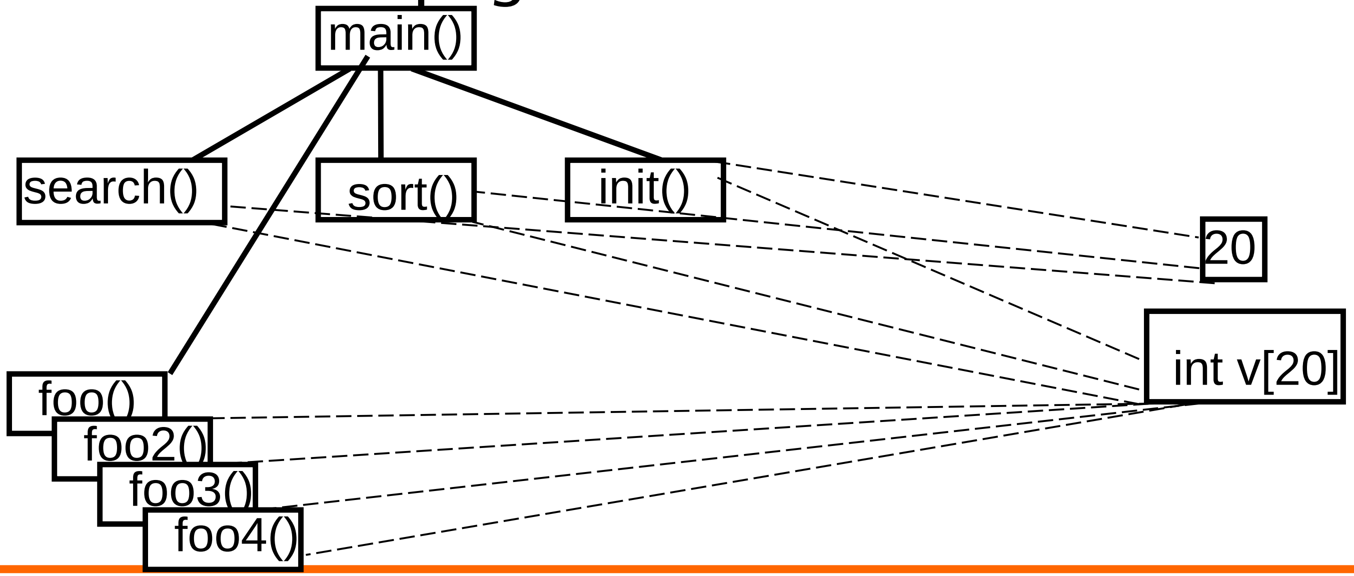
Procedural – problems

- No constraints on read/write relationships
- External functions can read/write vector’ s data



Procedural – In the long run

- (All) functions may read/write (all) data
- As time goes by, this leads to a growing number of relationships
- Source code becomes difficult to understand and maintain
  - Problem known as “Spaghetti code”



What is OO?

- Procedural Paradigm
  - Program defines data and then calls subprograms acting on data
- OO Paradigm
  - Program creates objects that encapsulate both the data and the procedures operating on data

- OO is simply a new way of organizing a program
  - Cannot do anything using OO that can’t be done using procedural paradigm

Why OO?

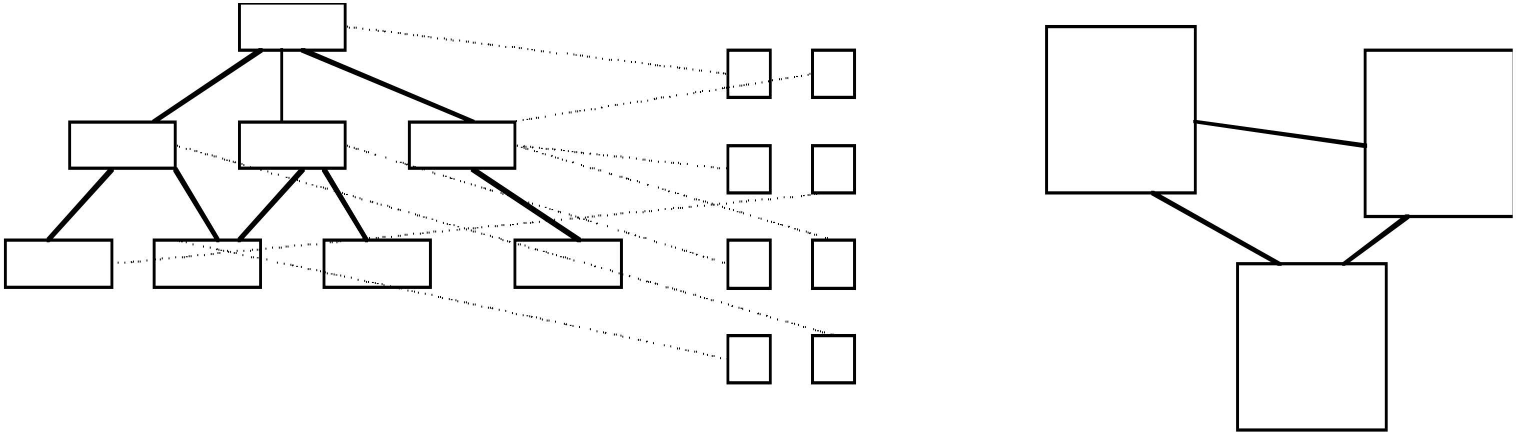
- Programs are getting too large to be fully comprehensible by any person
- There is a need for a way of managing very-large projects
- Object Oriented paradigm allows:
  - programmers to (re)use large blocks of code
  - without knowing all the picture
- OO makes code reuse a real possibility
- OO simplifies maintenance and evolution

Why OO?

- Benefits only occur in larger programs
- Analogous to structured programming
  - Programs < 30 lines, spaghetti is as understandable and faster to write than structured
  - Programs > 1000 lines, spaghetti is incomprehensible, probably doesn’t work, not maintainable
- Only programs > 1000 lines benefit from OO really

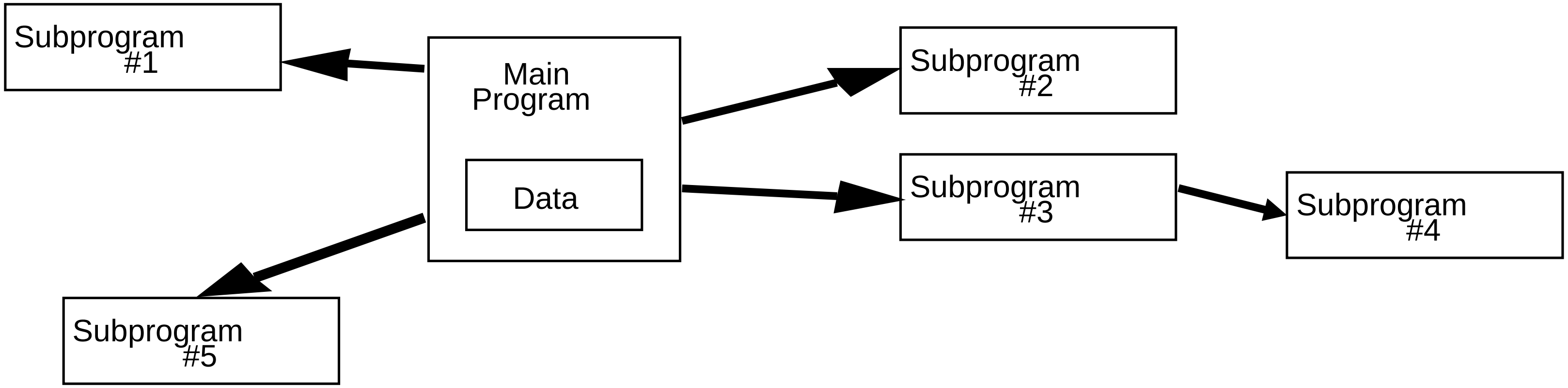
Object-Oriented Design

- Objects introduce an additional aggregation construct
- More complex system can be built

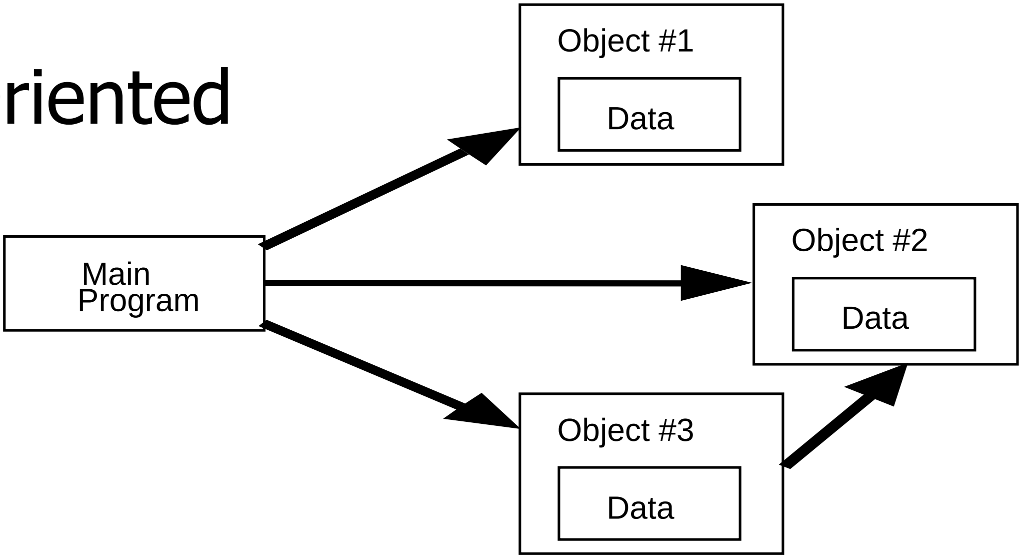


Procedural vs. OO

Procedural



Object Oriented



Object-Oriented approach

- Defines a new component type
  - Object (and class)
  - Both data and functions operating on data are within the same module
  - Allows defining a more precise interface
- Defines a new kind of relationship
  - Message passing
  - Read/write operations are limited to the object scope

Classification of OO languages

- **Object-Based** (Ada)
  - Specific constructs to manage objects
- **Class-Based** (CLU)
  - + each object belongs to a class
- **Object-Oriented** (Simula, Python)
  - + classes support inheritance
- **Strongly-Typed O-O** (C++, Java)
  - + the language is strongly typed

UML

- Unified Modeling Language
- Standardized modeling and specification language
  - Defined by the Object Management Group (OMG)
- Graphical notation to specify, visualize, construct and document an object-oriented system
- Integrates the concepts of Booch, OMT and OOSE, and merges them into a single, common and widely used modeling language

UML

- Several diagrams
  - Class diagrams
  - Activity diagrams
  - Use Case diagrams
  - Sequence diagrams
  - Statecharts

UML Class Diagram

- Captures
  - Main (abstract) concepts
  - Characteristics of the concepts
    - Data associated to the concepts
  - Relationships between concepts
  - Behavior of classes

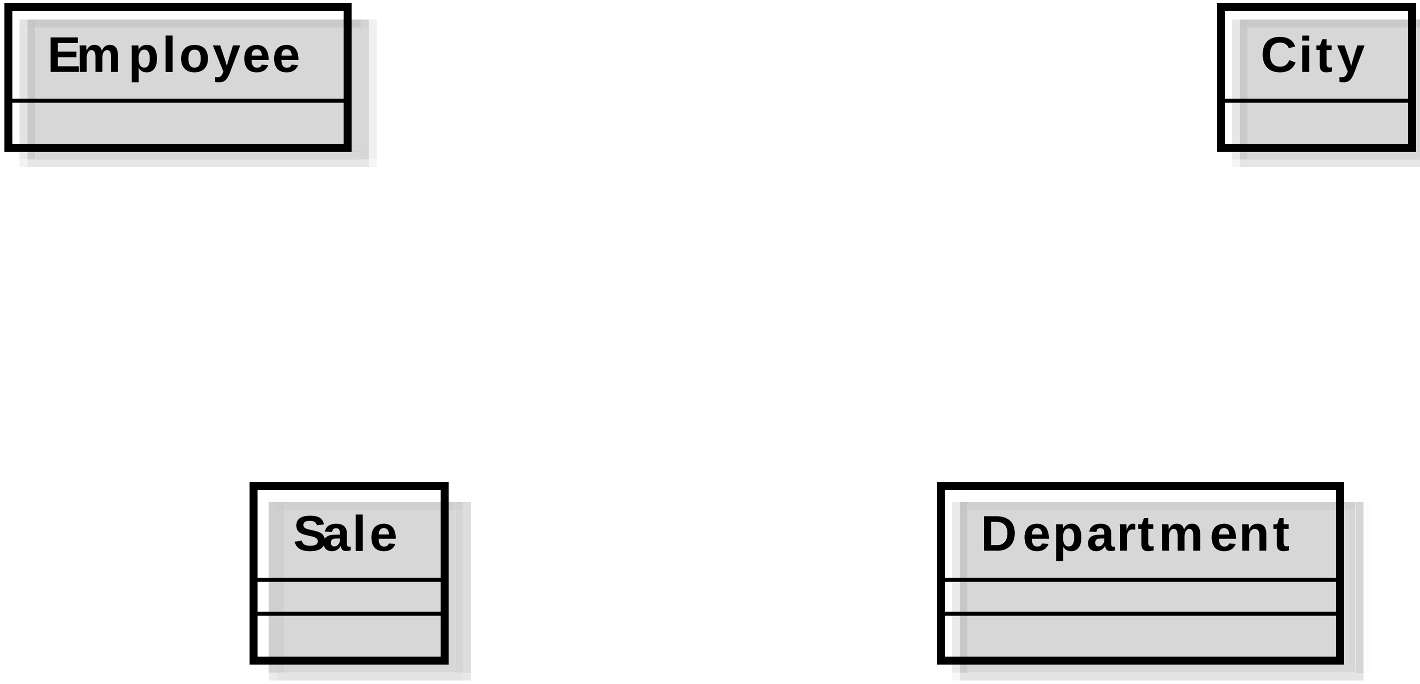
Abstraction levels

Abstract	Concept Entity Class Category Type
Concrete	Instance Item Object Example Occurrence

Class

- Represents a type of objects
  - Common properties
  - Autonomous existence.
  - E.g. facts, things, people
- An instance of a class is an object of the type that the class represents.
  - In an application for a commercial organization CITY, DEPARTMENT, EMPLOYEE, PURCHASE and SALE are typical classes.

Class – Examples



Employee

City

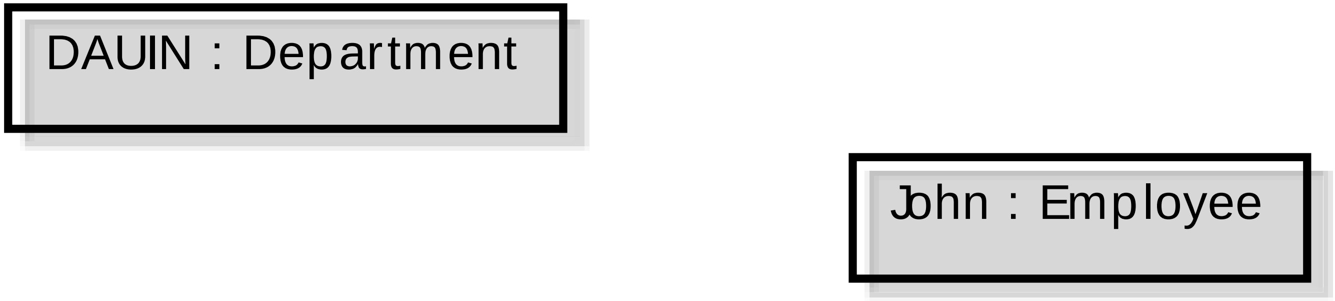
Sale

Department

Object

- Model of a physical or logical item
  - ex.: a student, an exam, a window
- Characterized by
  - identity
  - attributes (or data or properties or status)
  - operations it can perform (behavior)
  - messages it can receive

Object



DAUIN : Department

John : Employee

Class and Object

- Class (the description of object structure, i.e. *type*):
  - Data (ATTRIBUTES or FIELDS)
  - Functions (METHODS or OPERATIONS)
  - Creation methods (CONSTRUCTORS)
- Object (class instance)
  - State and identity

Class and object

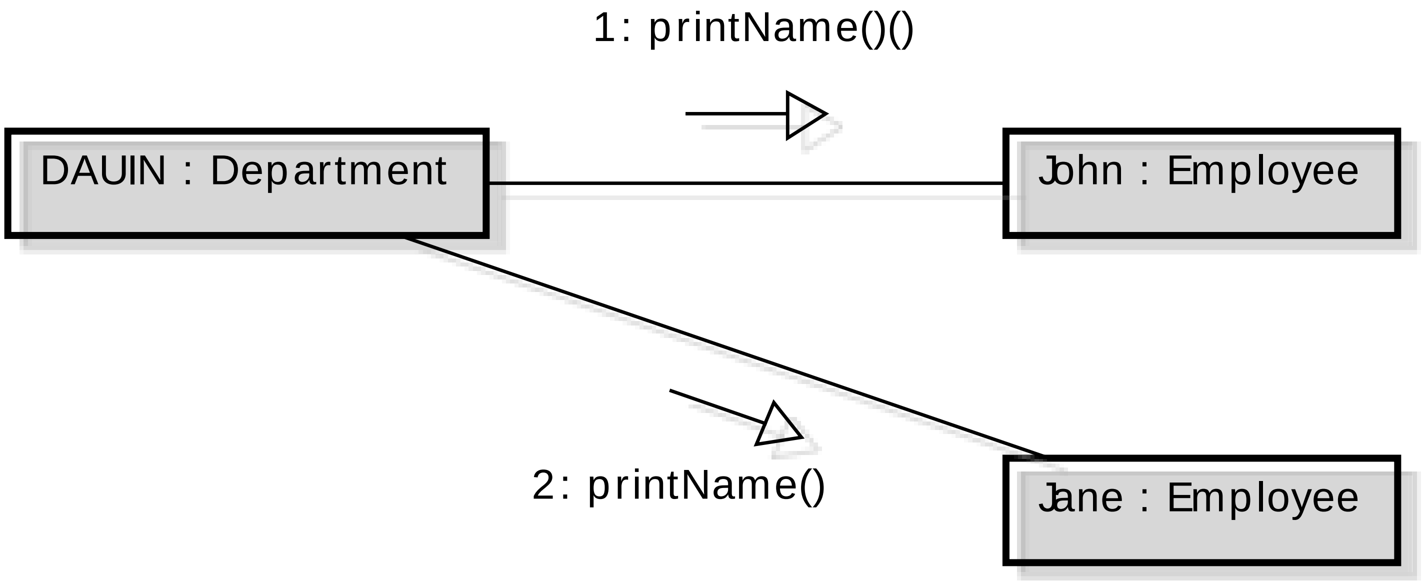
- A class is a type definition
  - Typically no memory is allocated until an object is created from the class
- The creation of an object is called instantiation. The created object is often called an instance
- There is no limit to the number of objects that can be created from a class
- Each object is independent. Interacting with one object doesn't affect the others

Message passing

- Objects communicate by message passing
  - Not by procedure call
  - Not by direct access to object's local data
- A message is a service request

Note: this is an abstract view that is independent from specific programming languages.

Object

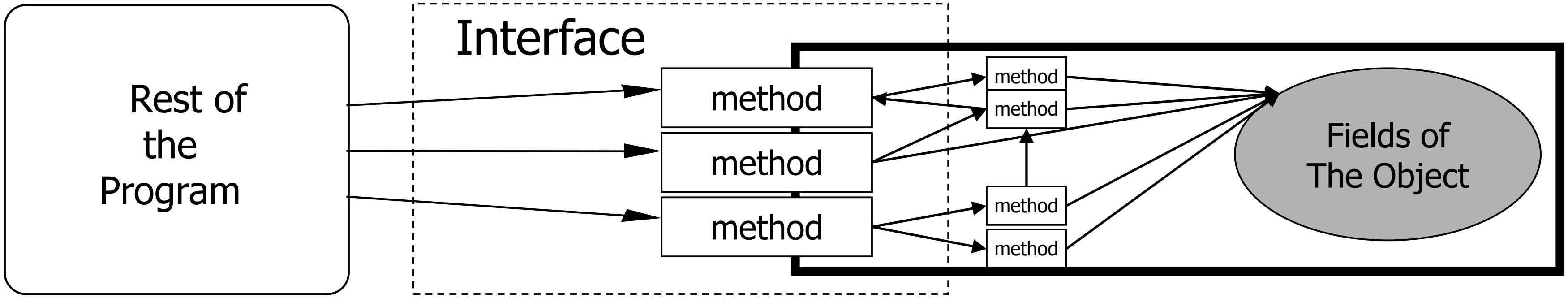

1: printName()()
DAUIN : Department
John : Employee
2: printName()
Jane : Employee

Interface

- Set of messages an object can receive
- Any other message is illegal
- The message is mapped to a function within the object
- The object is responsible for the association (message, function)

Interface

- The interface of an object is simply the subset of methods that other "program parts" are allowed to call
    - Stable



Benefits of encapsulation

- To use an object, the user need only comprehend the interface. No knowledge of the internals are necessary
- Self-contained. Once the interface is defined, the programmer can implement the interface (write the object) without interference of others

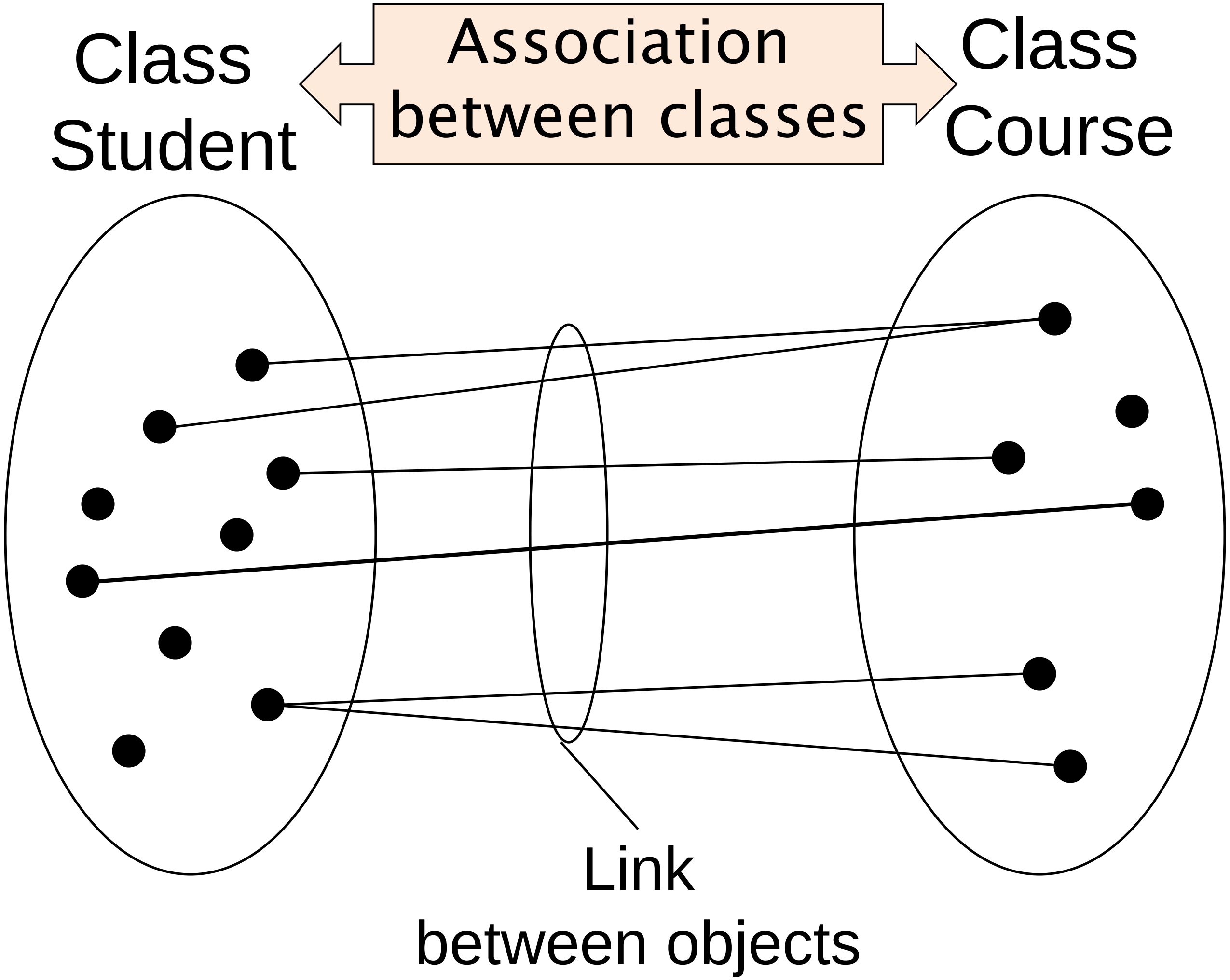
Benefits of encapsulation

- Implementation can change at a later time without rewriting any other part of the program (as long as the interface doesn't change)
- Any change in the data structure means modifying the code in one location, rather than code scattered around the program (error prone)

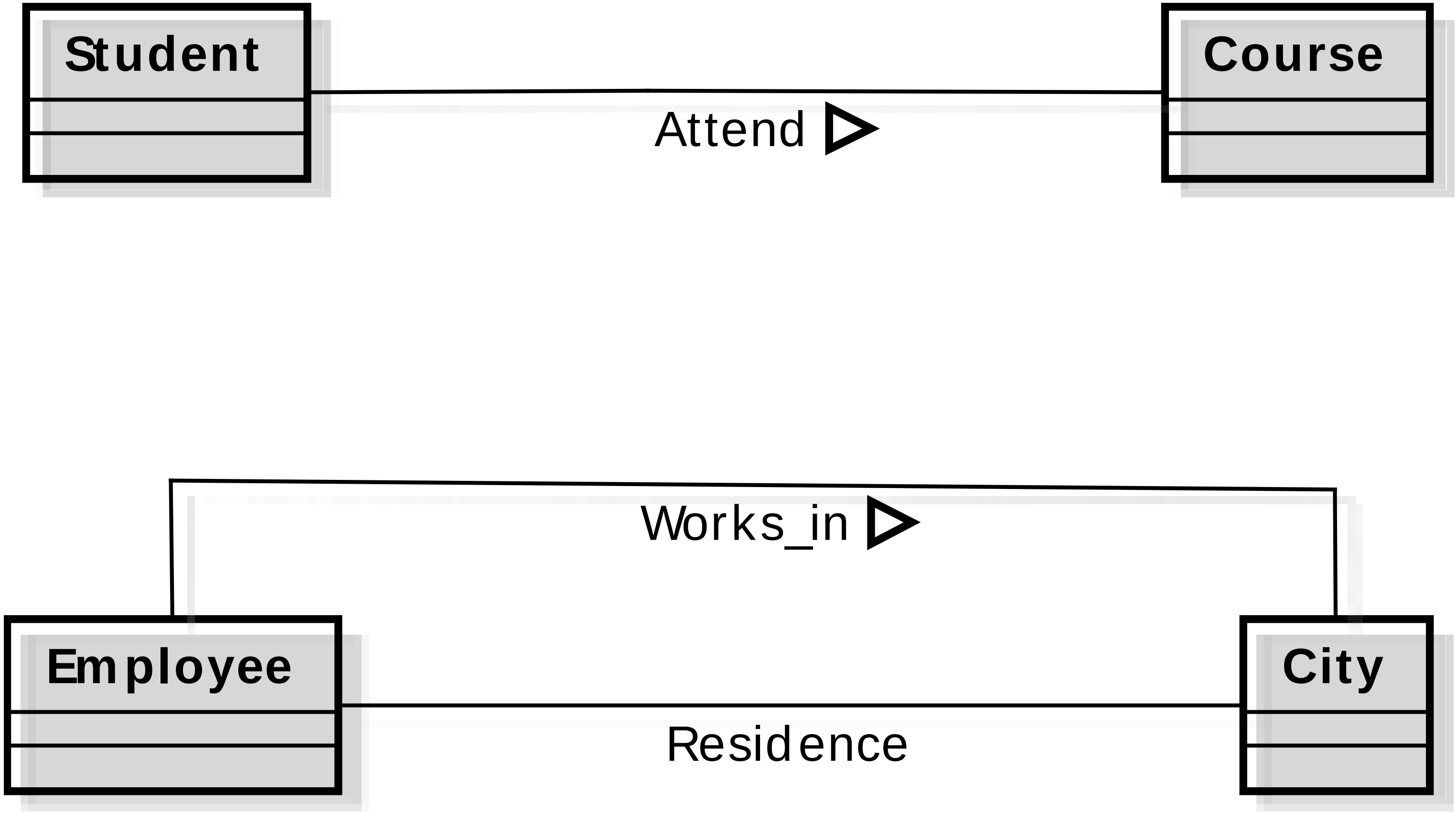
Association

- Represents the logical links between two classes.
- An occurrence of an association is a pair made up of the occurrences of entities, one for each involved class
    - Residence can be an association between the classes City and Employee;
    - Exam can be an association between the classes Student and Course.

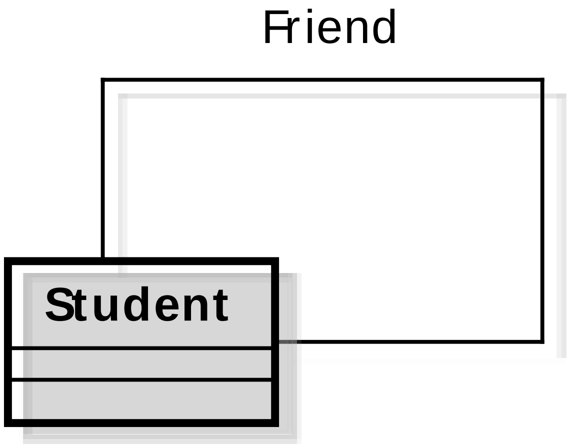
Associations

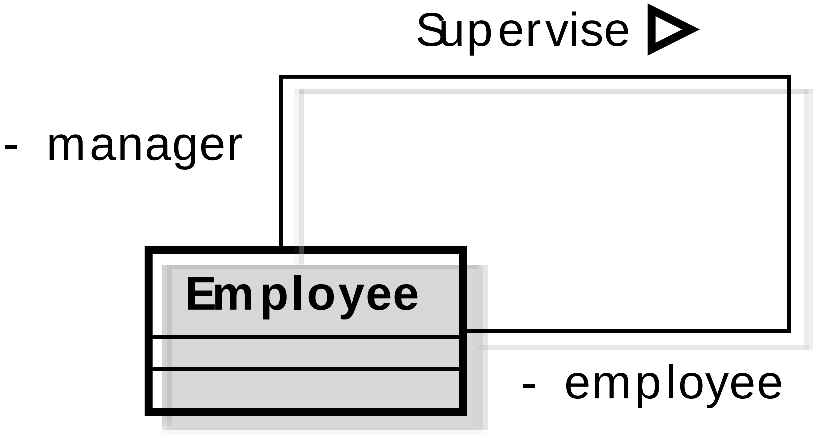


Association – Examples



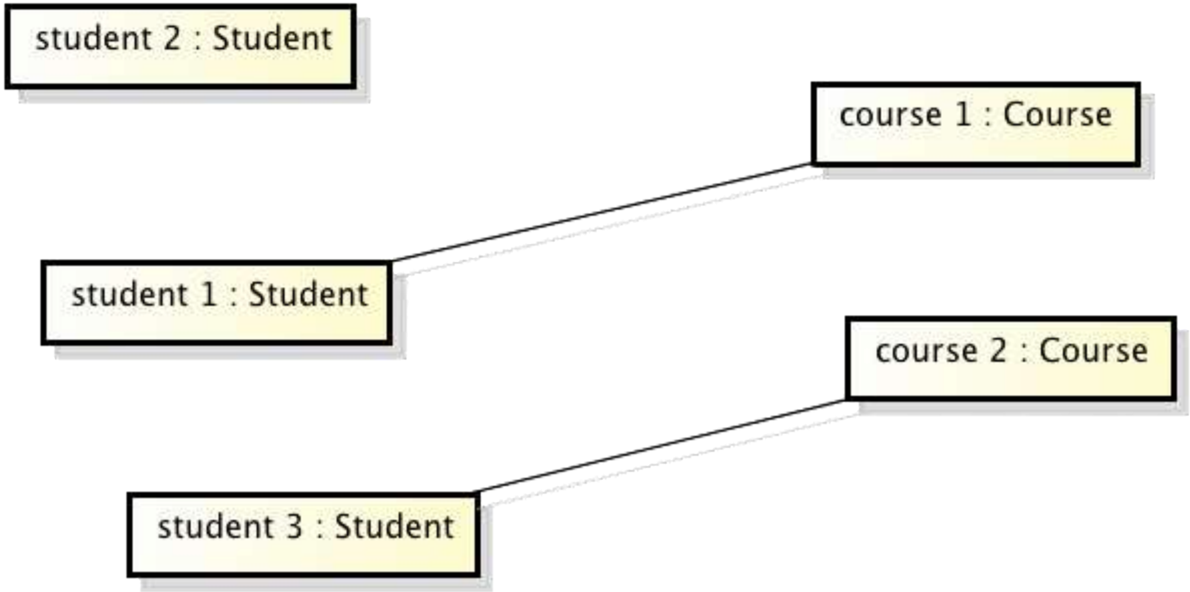
Recursive association–Samples





Link

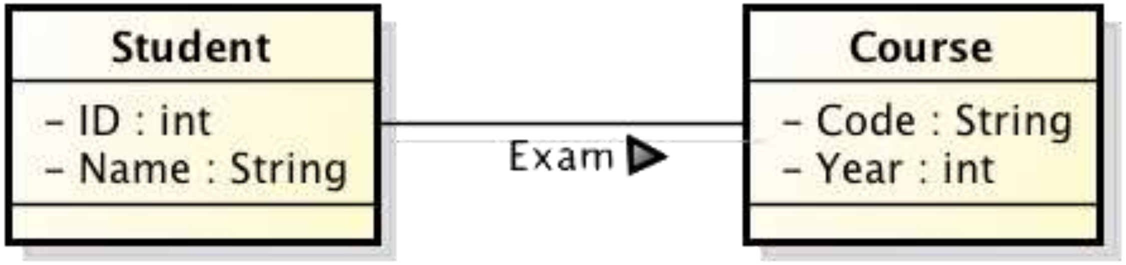
- Model of association between objects

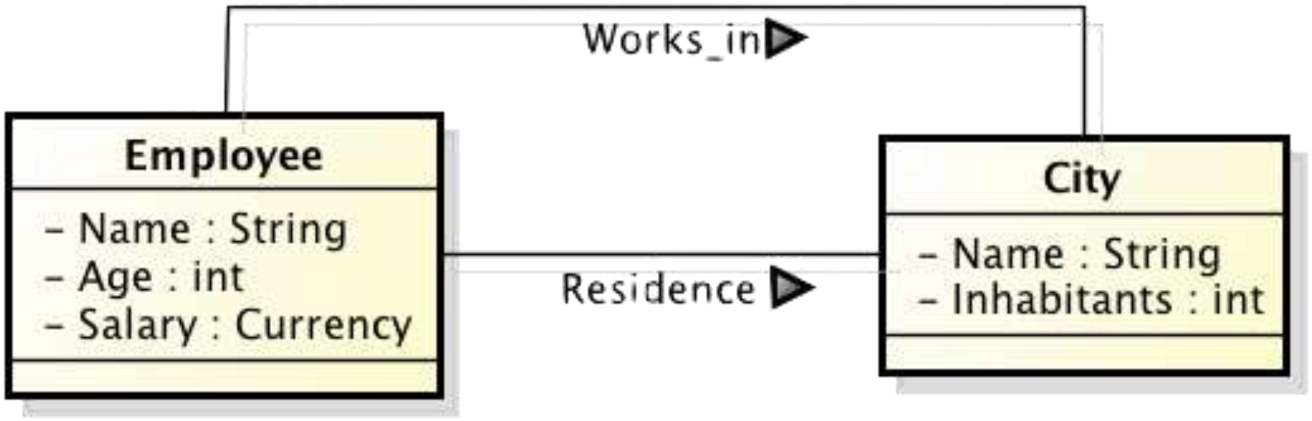


Attribute

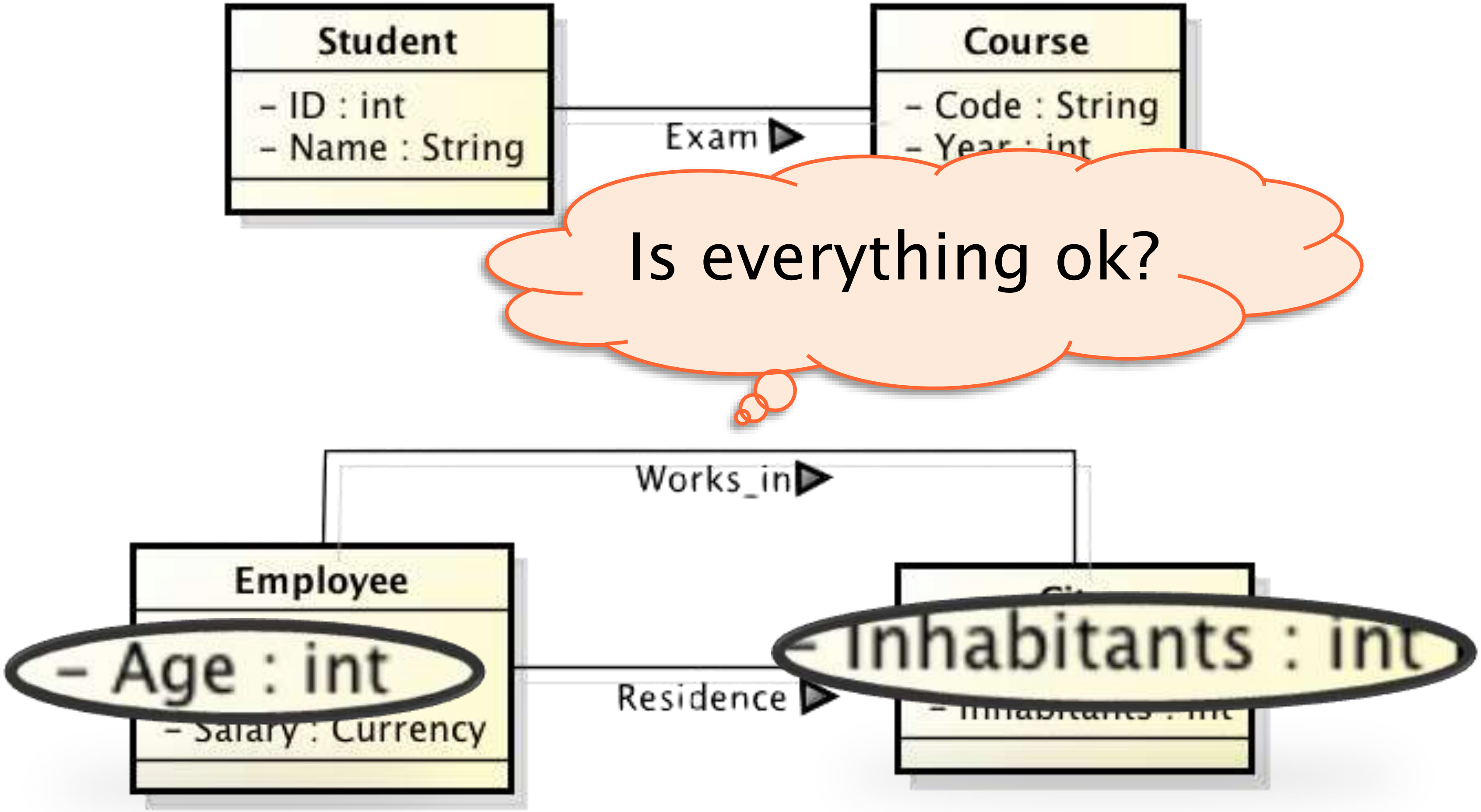
- Elementary property of classes
  - Name
  - Type
- An attribute associates to each object (occurrence of a class) a value of the corresponding type
  - Name: String
  - ID: Numeric
  - Salary: Currency

Attribute – Example





Attribute – Example

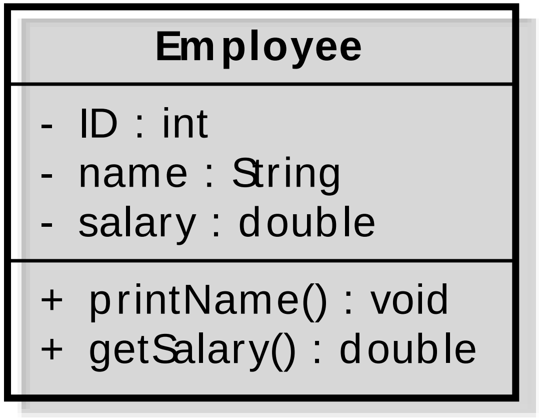


Is everything ok?

Method

- Describes an operation that can be performed on an object
  - Name
  - Parameters
  - Similar to functions in procedural languages
- It represent the means to operate on or access to the attributes

Method – Example



Employee

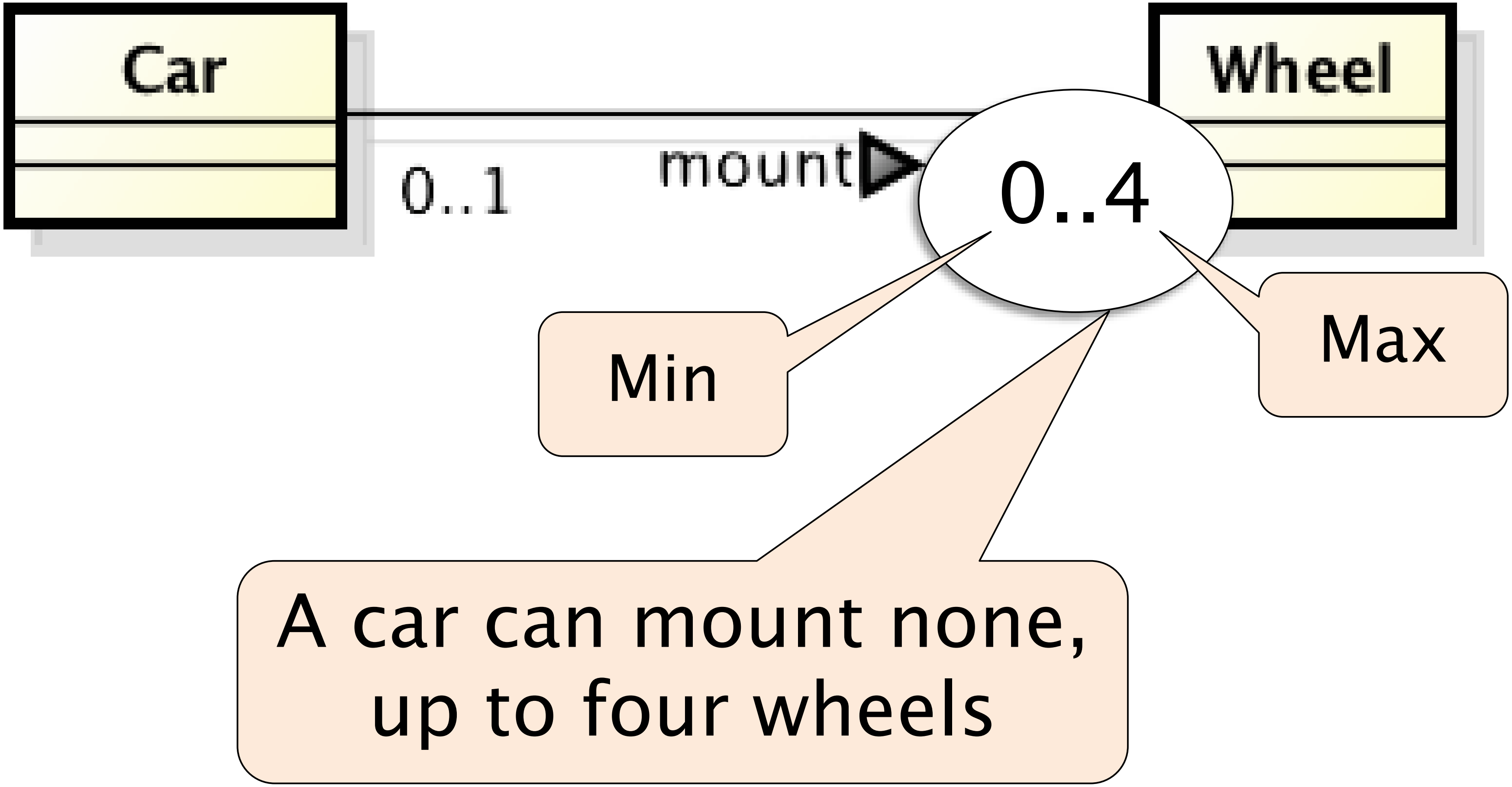
- - ID : int
- - name : String
- - salary : double

- + printName() : void
- + getSalary() : double

Multiplicity

- Describe the maximum and minimum number of links in which a class occurrence can participate
  - Undefined maximum expressed as *
- Should be specified for each class participating in an association

Multiplicity – Example

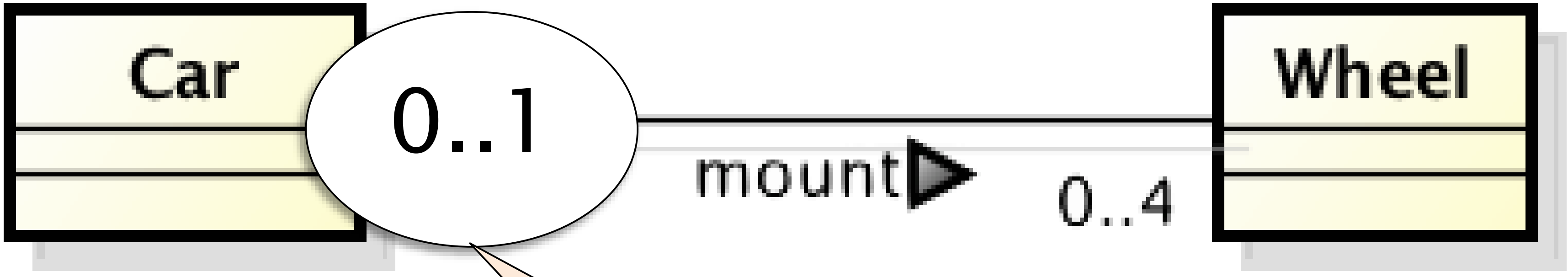


Min

Max

A car can mount none, up to four wheels

Multiplicity – Example

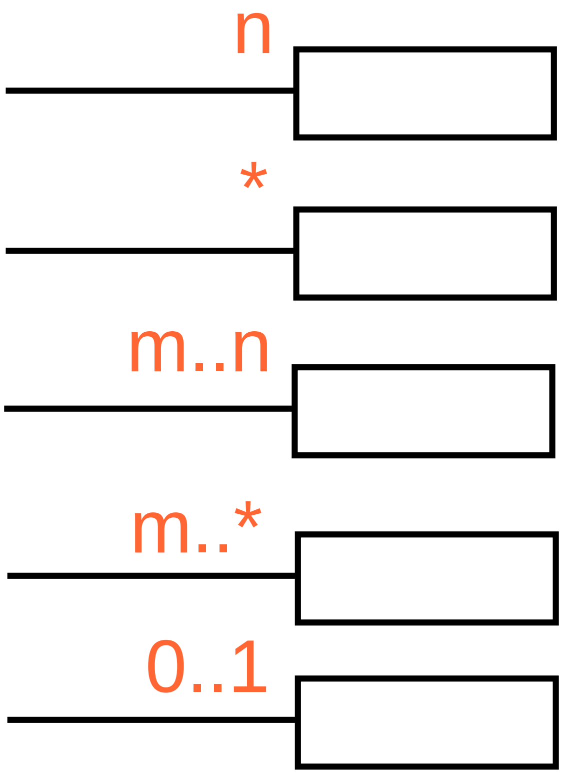


A wheel can be mounted on none or at most one car

Multiplicity

- Typically, only three values are used: 0, 1 and the symbol * (many)
- Minimum: 0 or 1
  - 0 means the participation is *optional,*
  - 1 means the participation is *mandatory;*
- Maximum: 1 or *
  - 1: each object is involved in at most one link
  - *: each object is involved in many links

Multiplicity



Exactly n

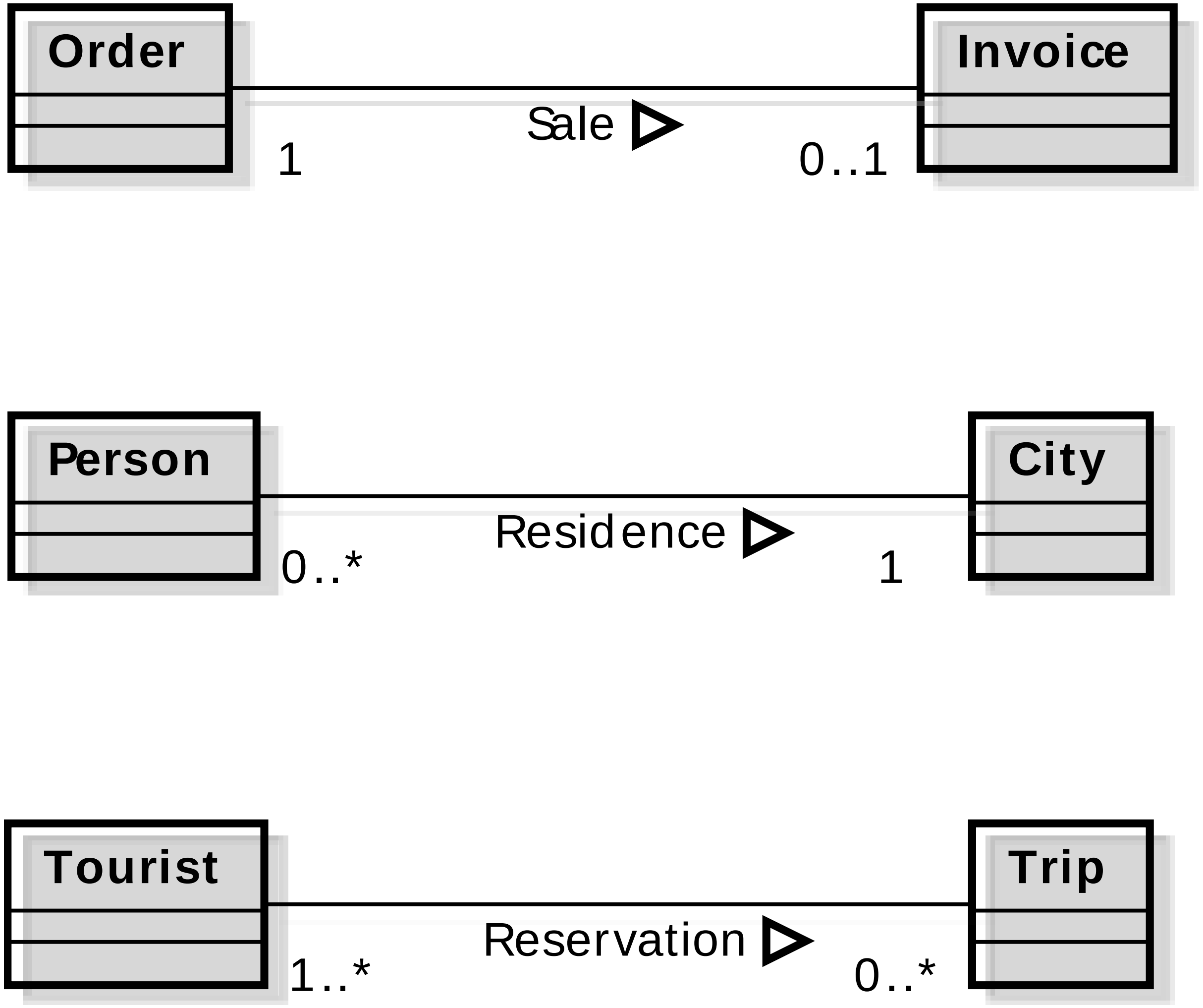
Zero or more

Between m and n (m,n included)

From m up

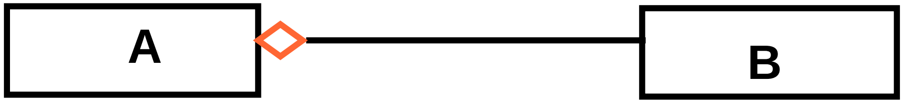
Zero or one (optional)

Multiplicity

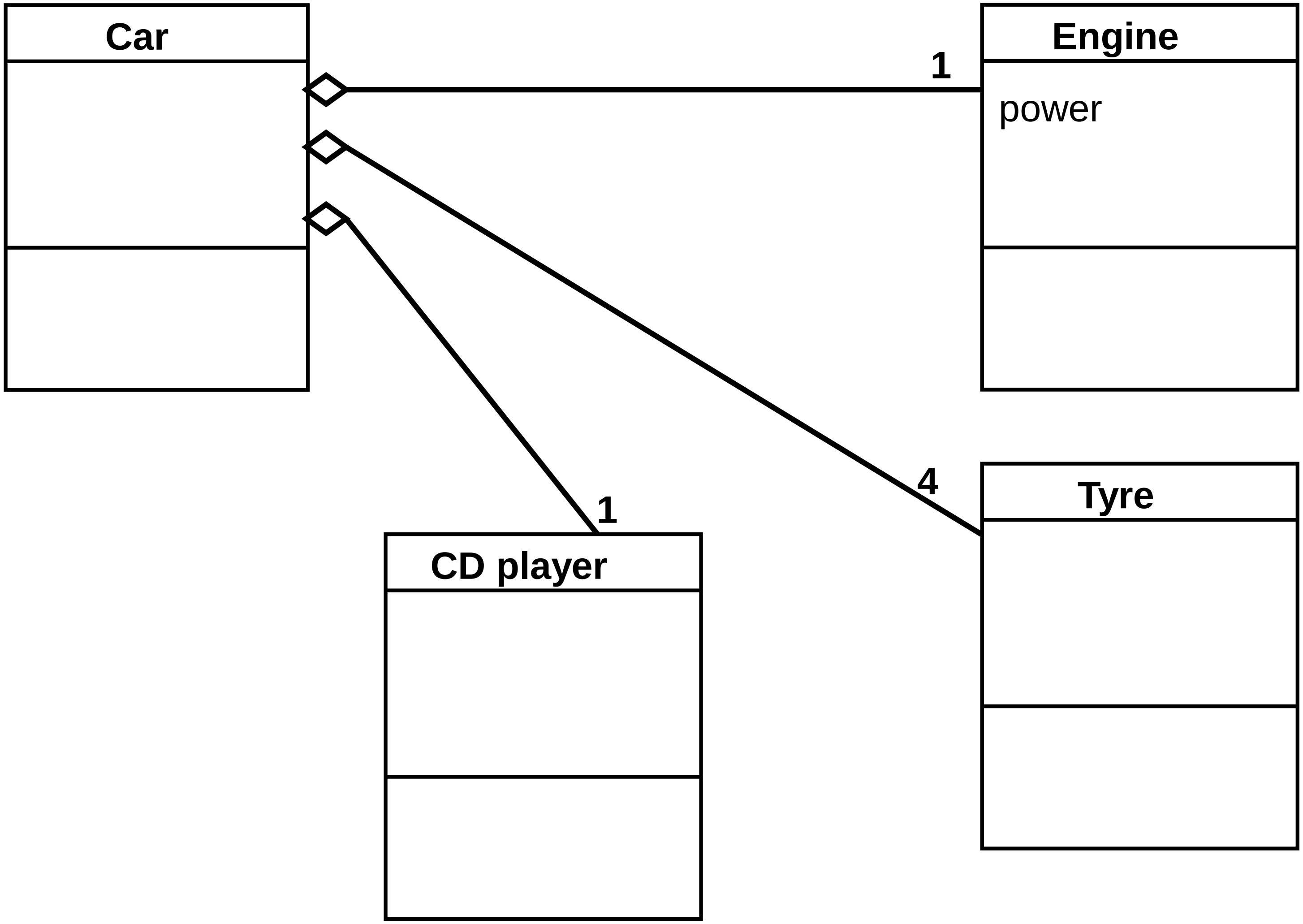


Aggregation

- B *is-part-of* A means that objects described by class B can be attributes of objects described by A



Example



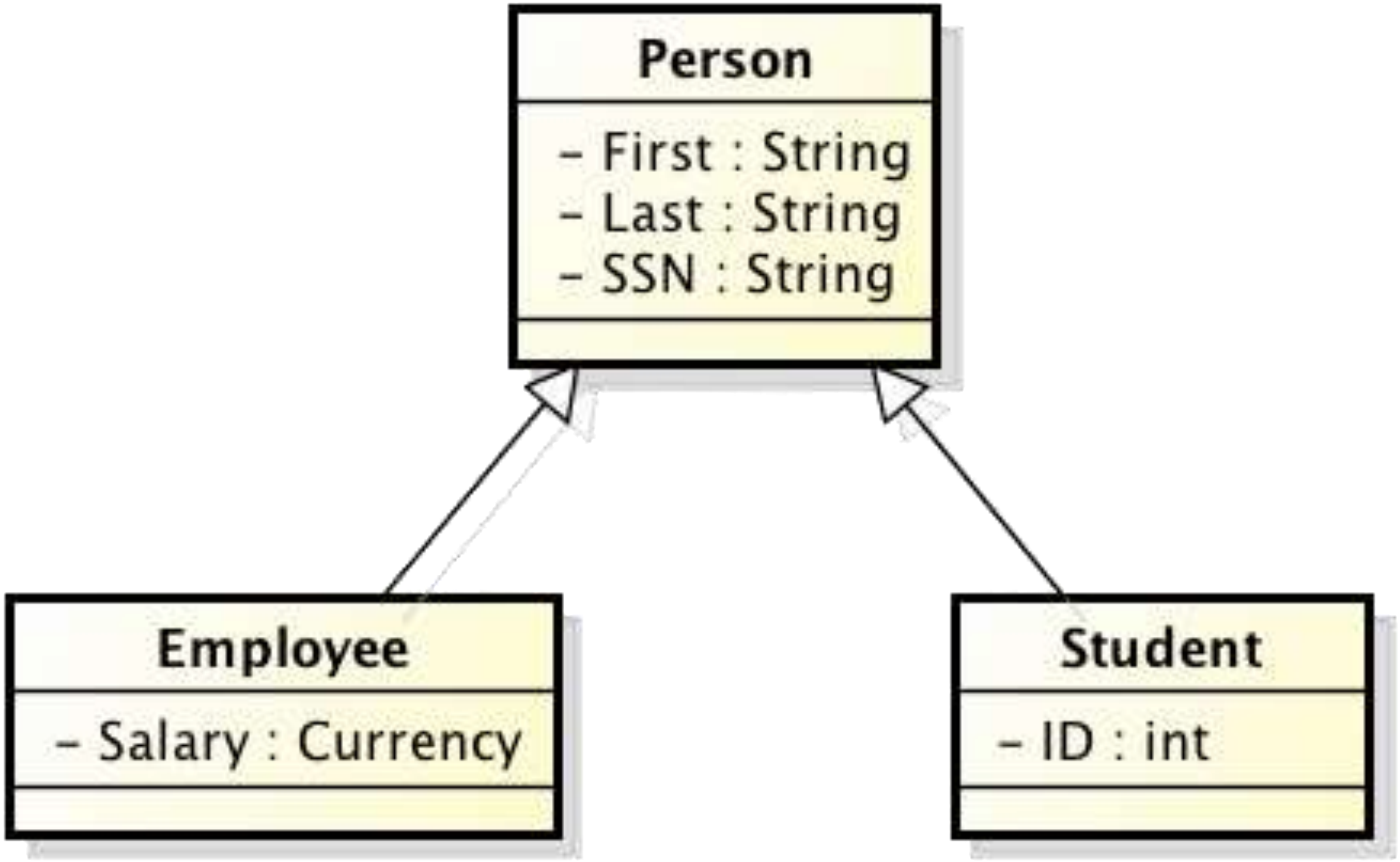
Inheritance

- A class can be a sub-type of another class
- The inheriting class contains all the methods and fields of the class it inherited from plus any methods and fields it defines
- The inheriting class can override the definition of existing methods by providing its own implementation
- The code of the inheriting class consists only of the changes and additions to the base class

Specialization / Generalization

- B *specializes* A means that objects described by B have the same properties of objects described by A
- Objects described by A may have additional properties
- B is a special case of A
- A is a generalization of B (and possible other classes)

Generalization



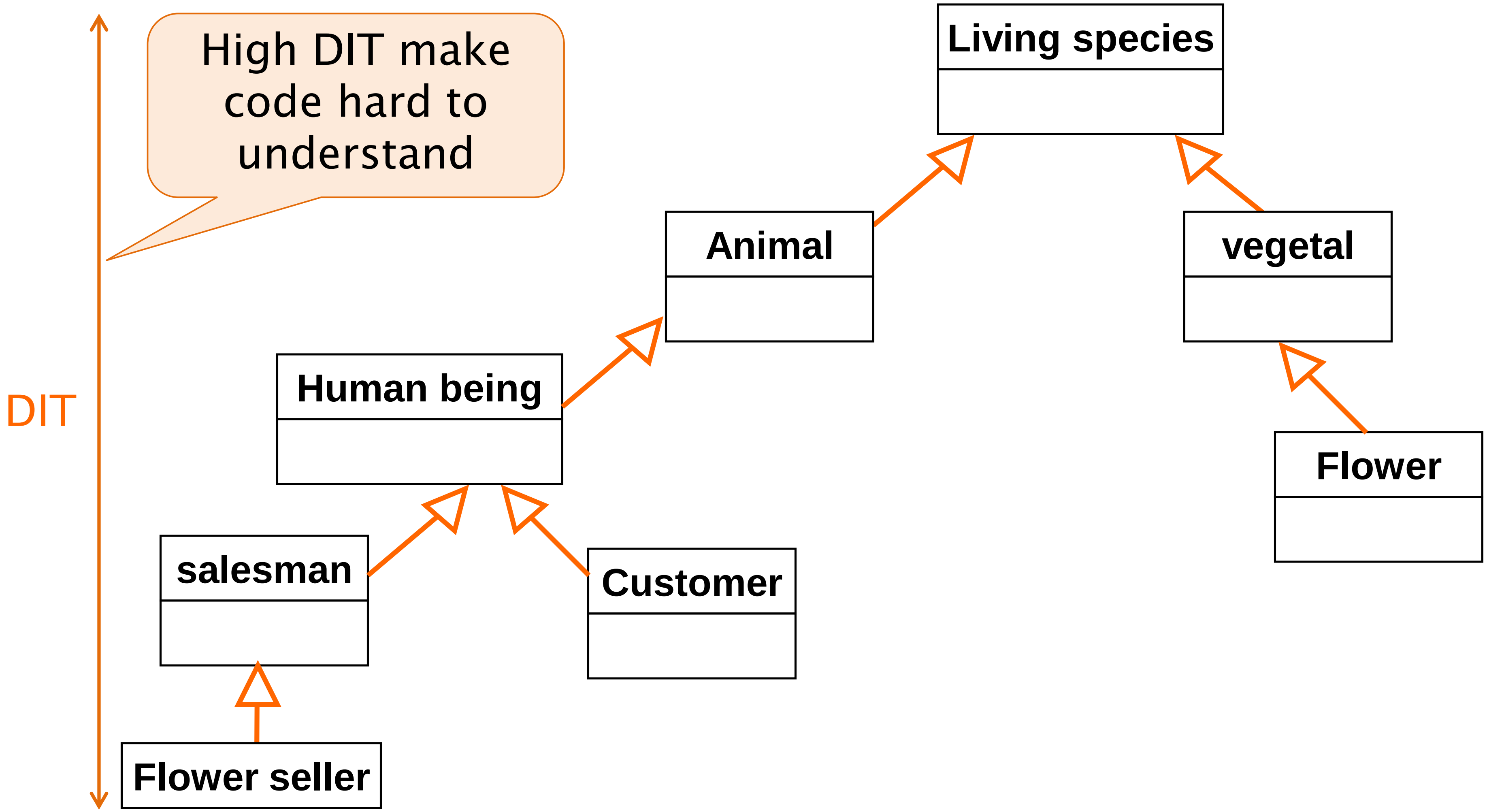
Inheritance terminology

- Class one above
  - Parent class
- Class one below
  - Child class
- Class one or more above
  - Superclass, Ancestor class, Base class
- Class one or more below
  - Subclass, Descendent class, Derived class

Why inheritance

- Frequently, a class is merely a modification of another class. In this way, there is minimal repetition of the same code
- Localization of code
  - Fixing a bug in the base class automatically fixes it in the subclasses
  - Adding functionality in the base class automatically adds it in the subclasses
  - Less chances of different (and inconsistent) implementations of the same operation

Example of inheritance tree



References

- Fowler, M. “UML Distilled: A Brief Guide to the Standard Object Modeling Language – 3rded.”, Addison-Wesley Professional (2003)