OO Paradigm and UML

Object-Oriented Programming

SoftEng
http://softeng.polito.it

Licensing Note

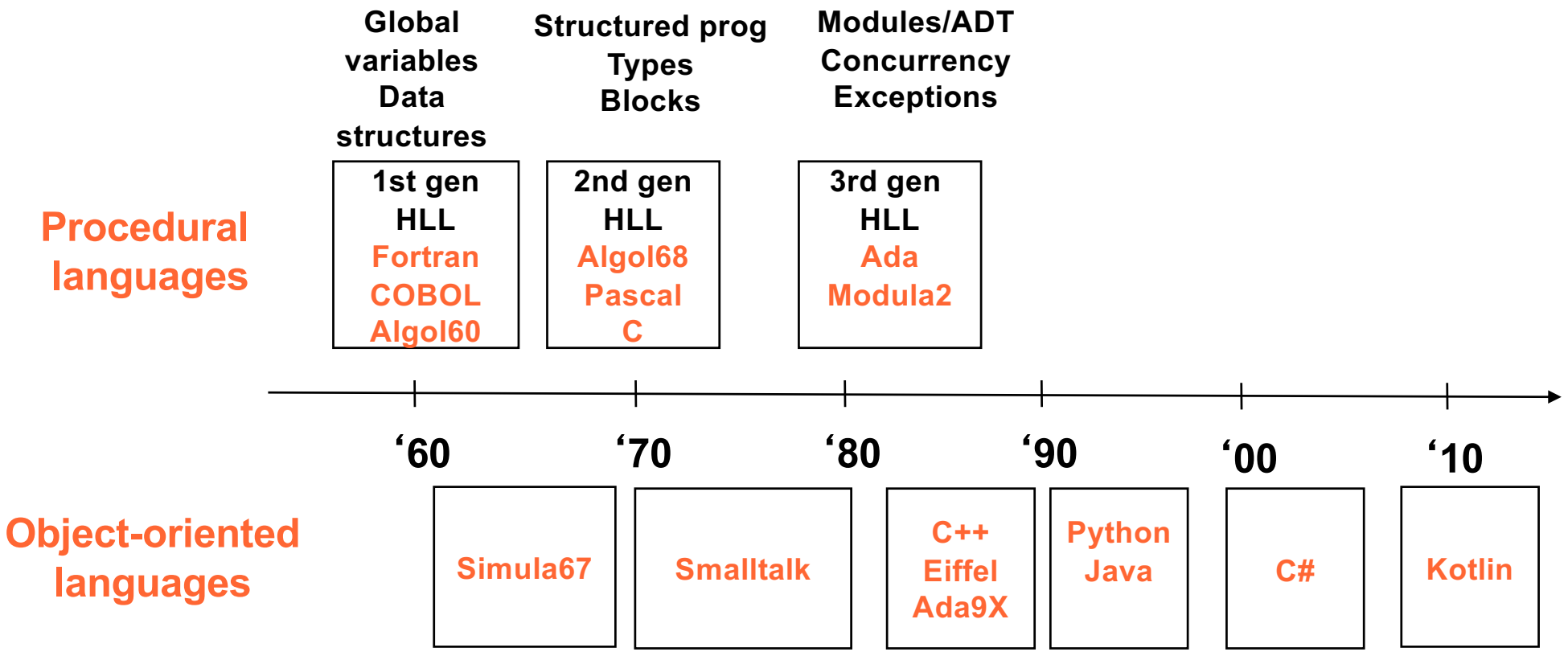
From procedural to object oriented programming paradigm

OBJECT ORIENTED PARADIGM

Programming paradigms

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

Languages timeline



Example – Receipt

- Cash registers emit purchase receipts
- A receipt is made up of items
- Every item correspond to a product that has a name and a price
- Products' info is stored in a price list
- Any time a new product code is entered the corresponding item is added to the receipt
- After the last item is entered, a list of the items (with product name and price) are printed together with the total sum.

Example: Shop Receipt

- Input:
  - 13
  - 57
  - 123
  - 0 (end of receipt)
- Output

```
Receipt:
       ID13 : 16.62
       ID57 :  9.73
      ID123 :  0.06
------
Number of items: 3
           Total: 26.41
```

Procedural (C)

```
float prices[MAX_LIST];
char* names[MAX_LIST];
int receipt[MAX_RCPT];
int n_items;
void add(int) {/* add item to receipt */ }
void print(){ /* print receipt */ }
void init() { /* initialize */ }
int read() { /* read item code */
int main(){
    init();
    int code;
    while( (code = read()) ){ add(code); }
    print();
}
```

Modules and relationships

Modules:

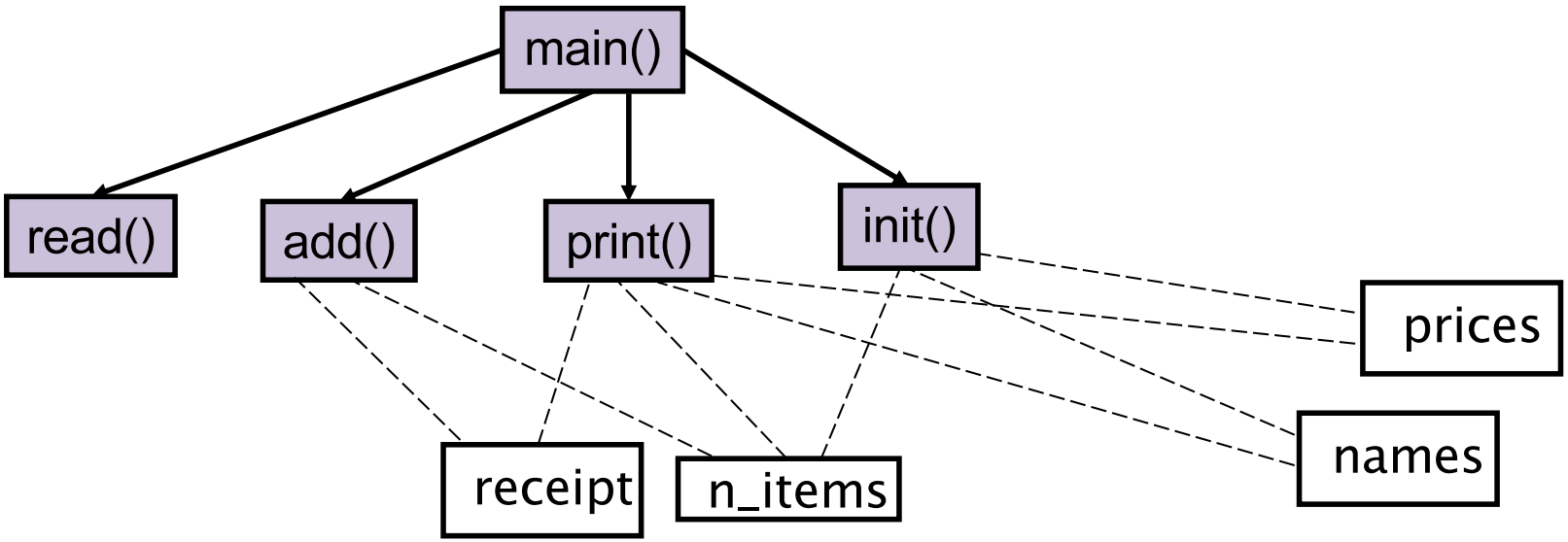
Data

Function (Procedure)

Relationships

Call

Read/write



Problems

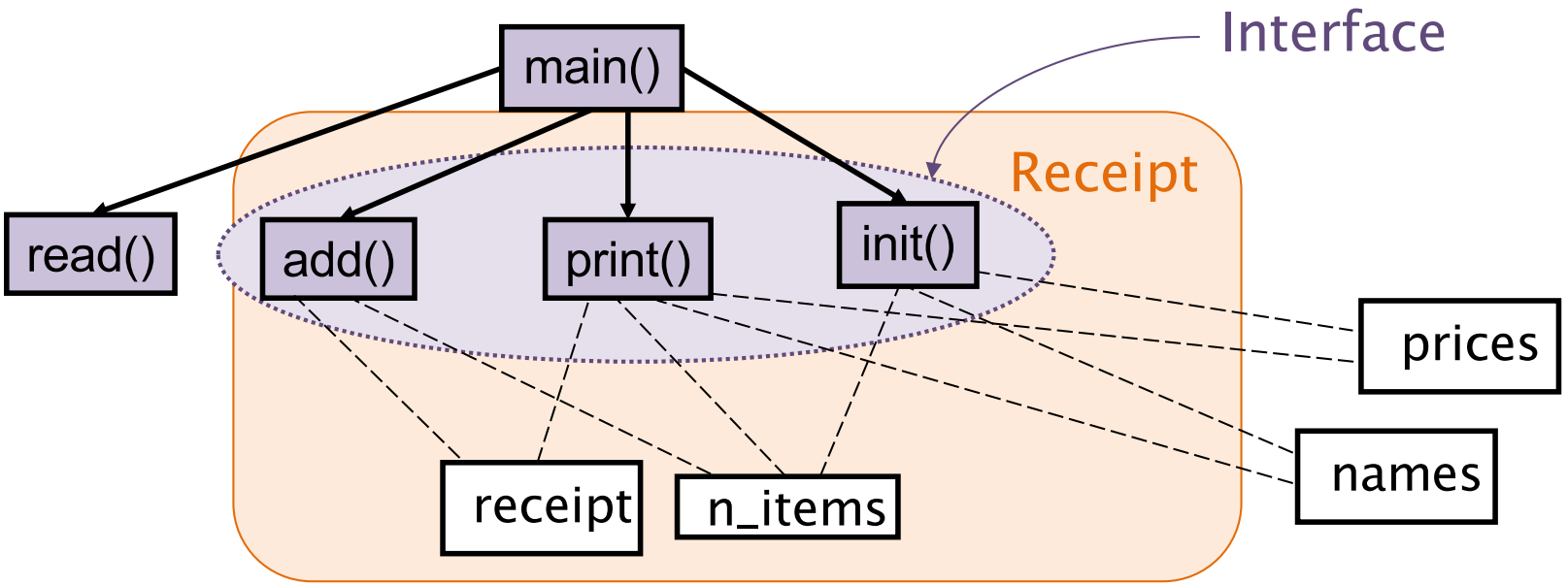
- No syntactic relationship between:
  - Arrays (**`receipt, prices, names`**)
  - Relative operations (**`add`**, **`print`**, **`init`**)
- Lack of link between coupled arrays (**`prices, names`**)
- No control over *size*:

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

- No guarantee on initialization
  - Actually performed?

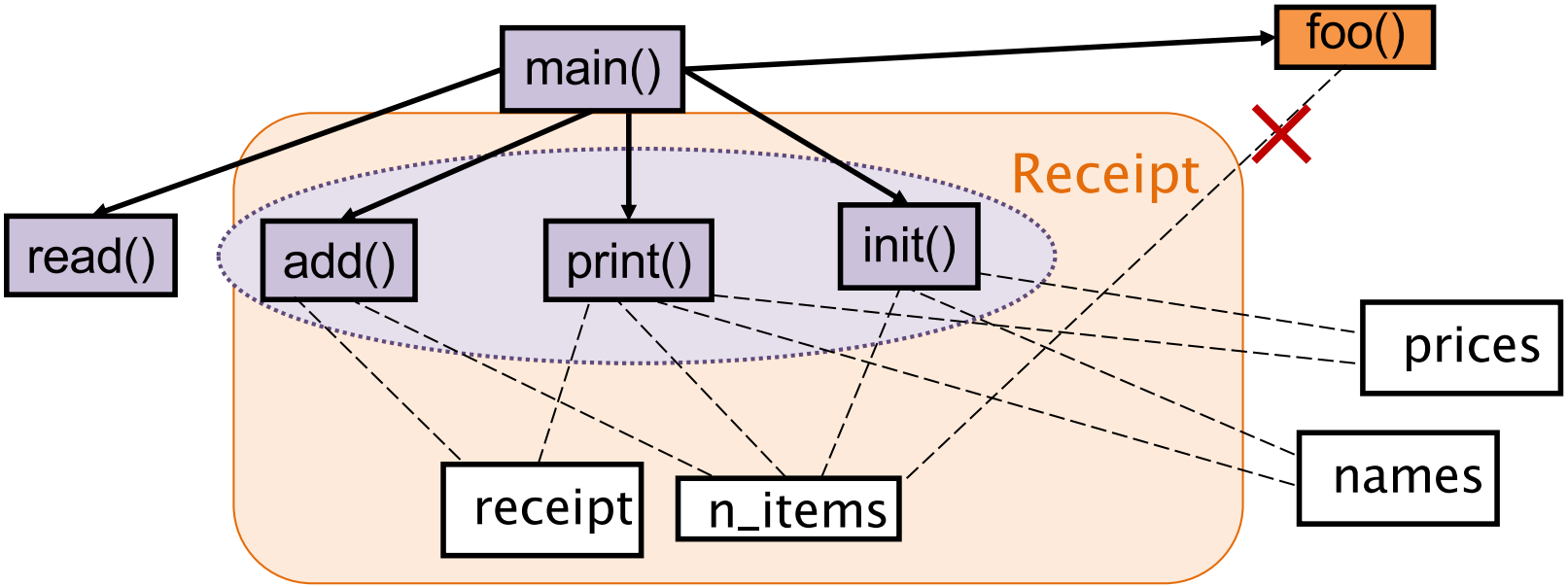
Objects – Encapsulation

- Bring together code and data
  - E.g. **`add()`** + **`receipt`** + **`n_items`**



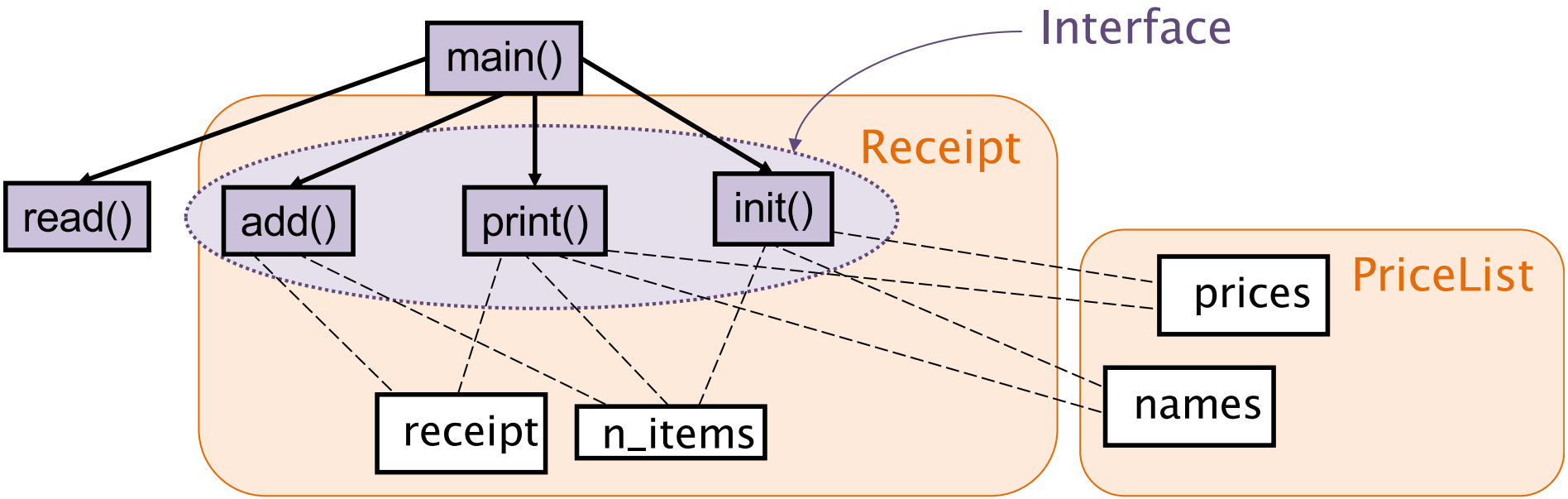
Objects – Information Hiding

- Hide object information from external modules
  - The only way to access data within an object is through its interface



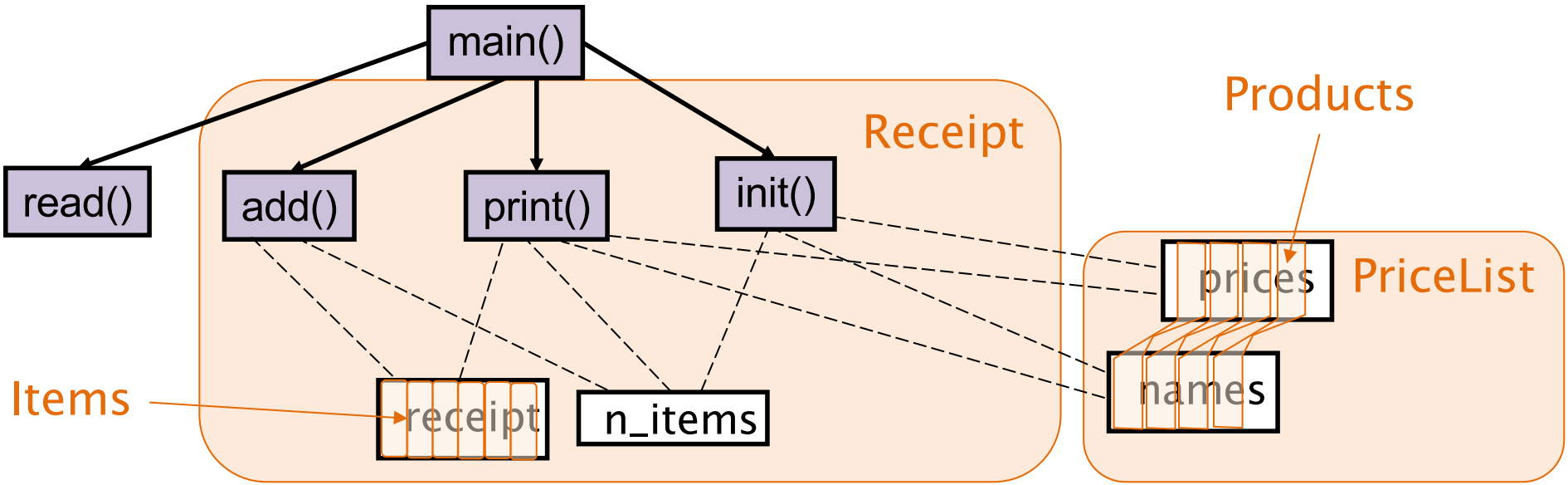
Objects

- Tie related data elements
  - E.g. **`prices`** + **`names`**



Objects

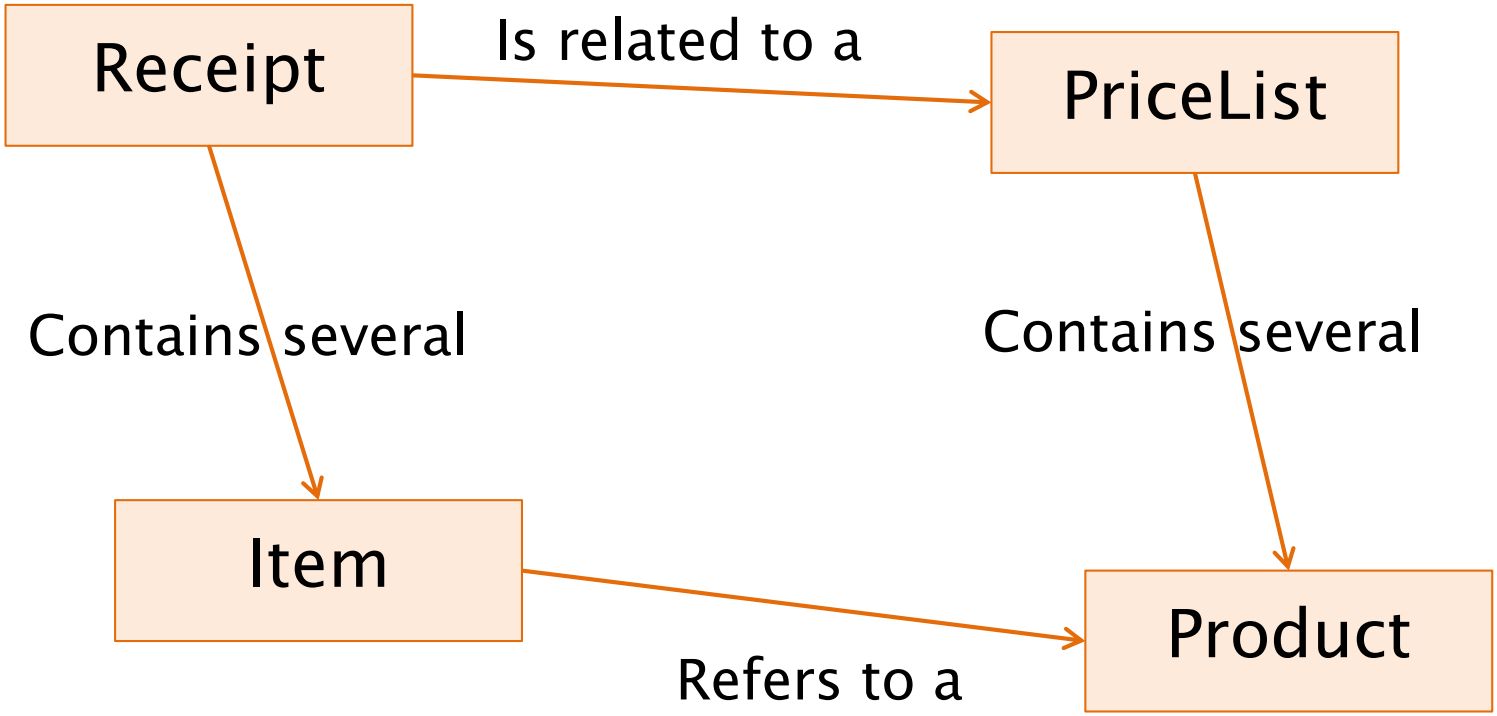
- Represent semantically consistent elements that map to problem-domain concepts
  - E.g., items and products



Classes

- Represent high level concepts
  - Often taken from problem domain
- Are instantiated into Objects
  - Define common features of Objects
- Are related to each other
  - Define links and communication patterns among their instances
- Can be defined by specialization
  - Specific classes inherit from general ones

Classes


Receipt
Is related to a
PriceList
Contains several
Contains several
Item
Refers to a
Product

Object and Classes

- Objects are created as instances of classes
- The structure of an object is defined by its class
  - The class is the *type* of the object
- A class defines how an object is built through a *constructor* function
  - Initialization is performed automatically

Object-Oriented approach

- Defines a new component type
  - Object (and class)
  - Both data and functions accessing it 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 same object scope

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

When OO?

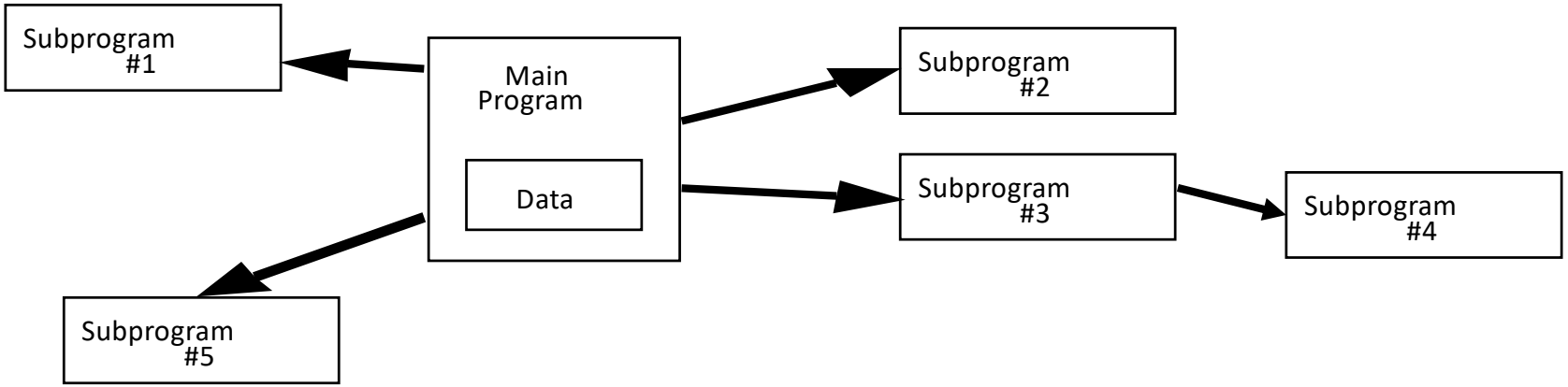
- Benefits only occur in larger programs
- Analogous to structured programming
  - Programs < 30 lines, spaghetti is as understandable and faster to write than structured
  - Programs > 1000 lines, spaghetti is incomprehensible, probably doesn't work, not maintainable
- Only programs > 1000 lines benefit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

An engineering approach

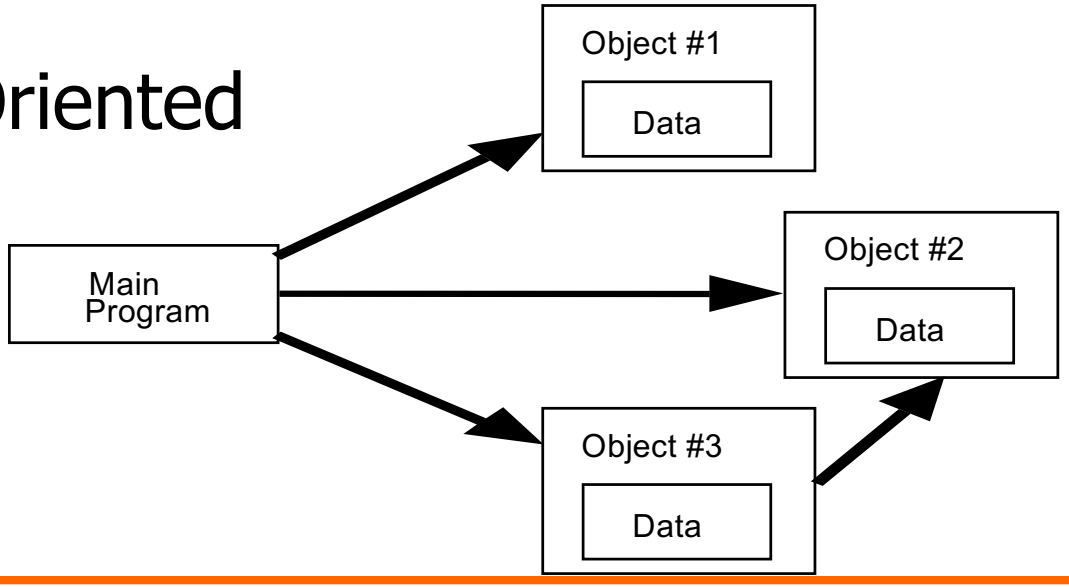
- Given a system, with components and relationships among them, we shall:
  - Identify the components
  - Define component interfaces
  - Define how components interact with each other through their interfaces
  - Minimize relationships among components

Procedural vs. OO

Procedural



Object Oriented

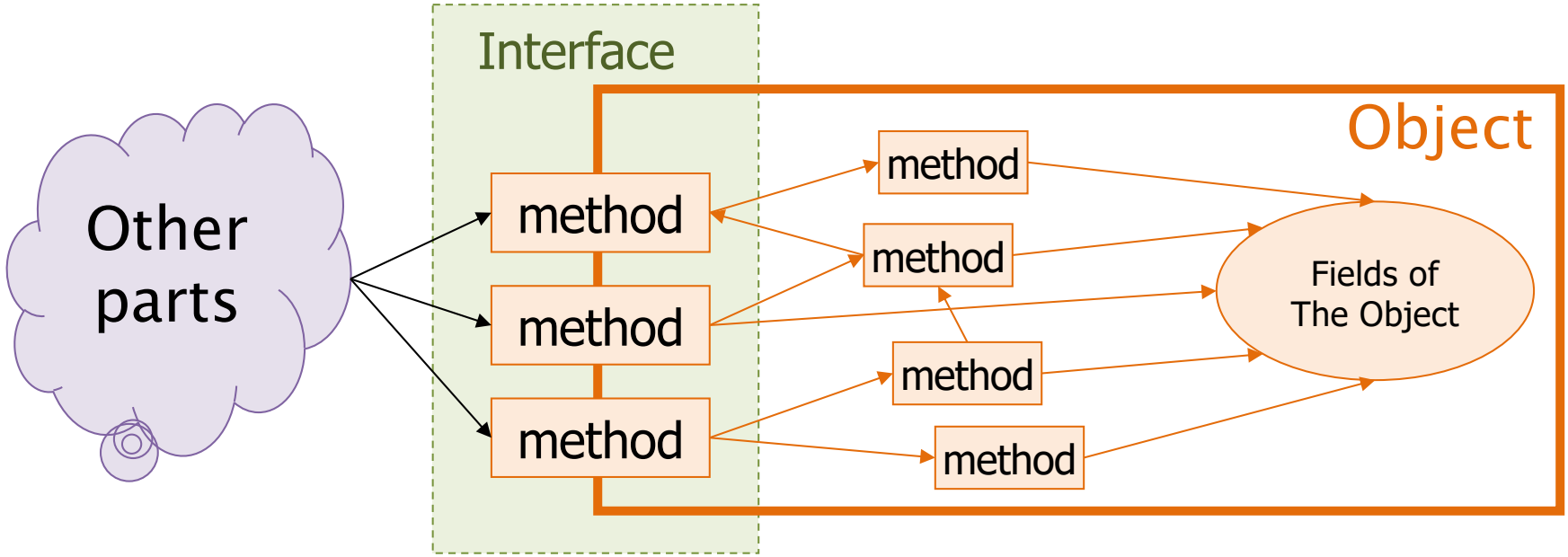


Interface

- Set of messages an object can receive
  - Each message is mapped to an internal "function" within the object
  - The object is responsible for the association (message → function)
  - Any other message is illegal
- The interface
  - Encapsulates the internals
  - Exposes a standard boundary

Interface

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



Encapsulation

- Simplified access
  - 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

Encapsulation

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

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

The Object-Oriented Paradigm

UML and Modeling

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

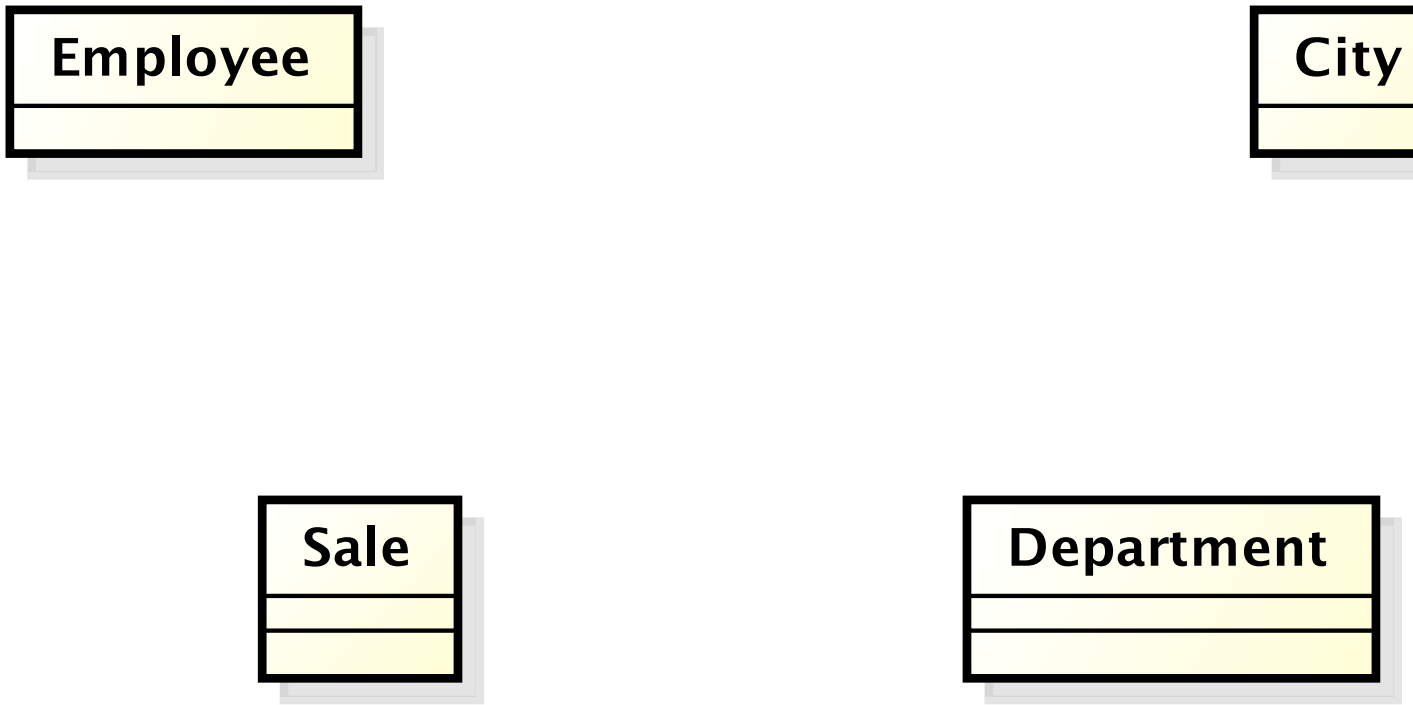
Abstraction levels

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

Class

- Represents a set 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



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

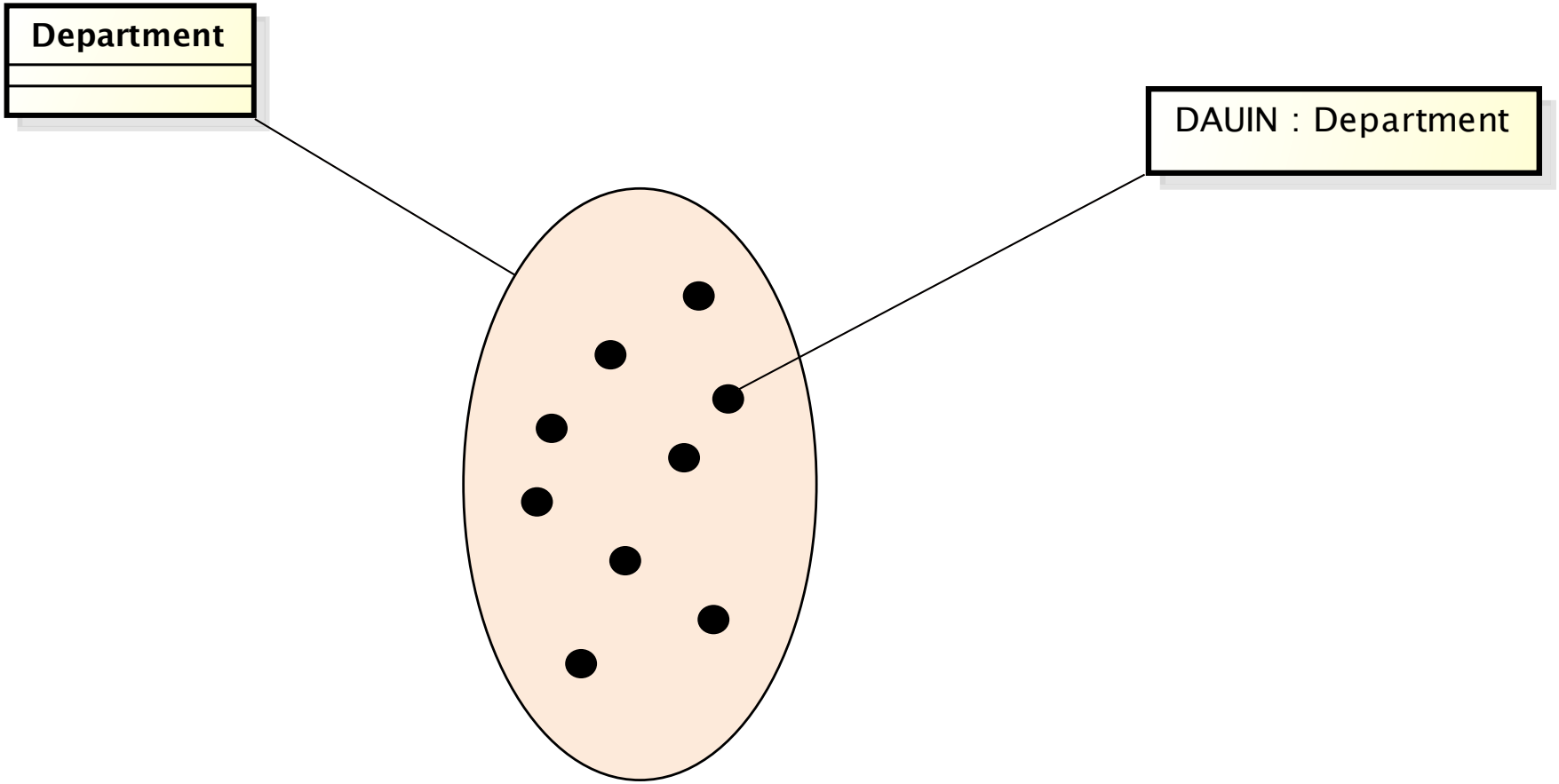
Class and Object

- 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

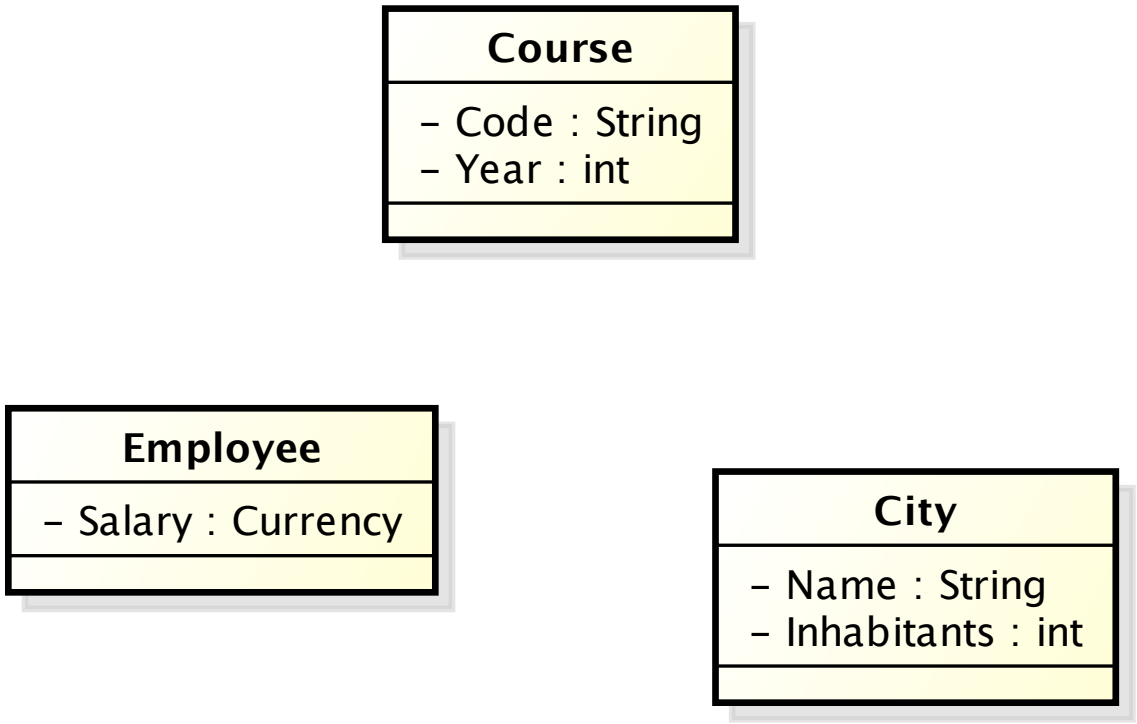
Classes and objects



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

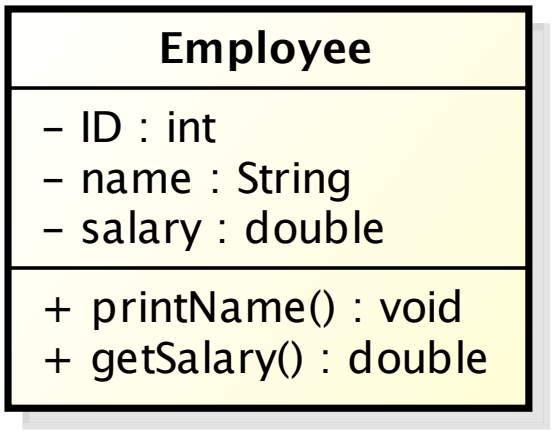
Attribute – Example



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

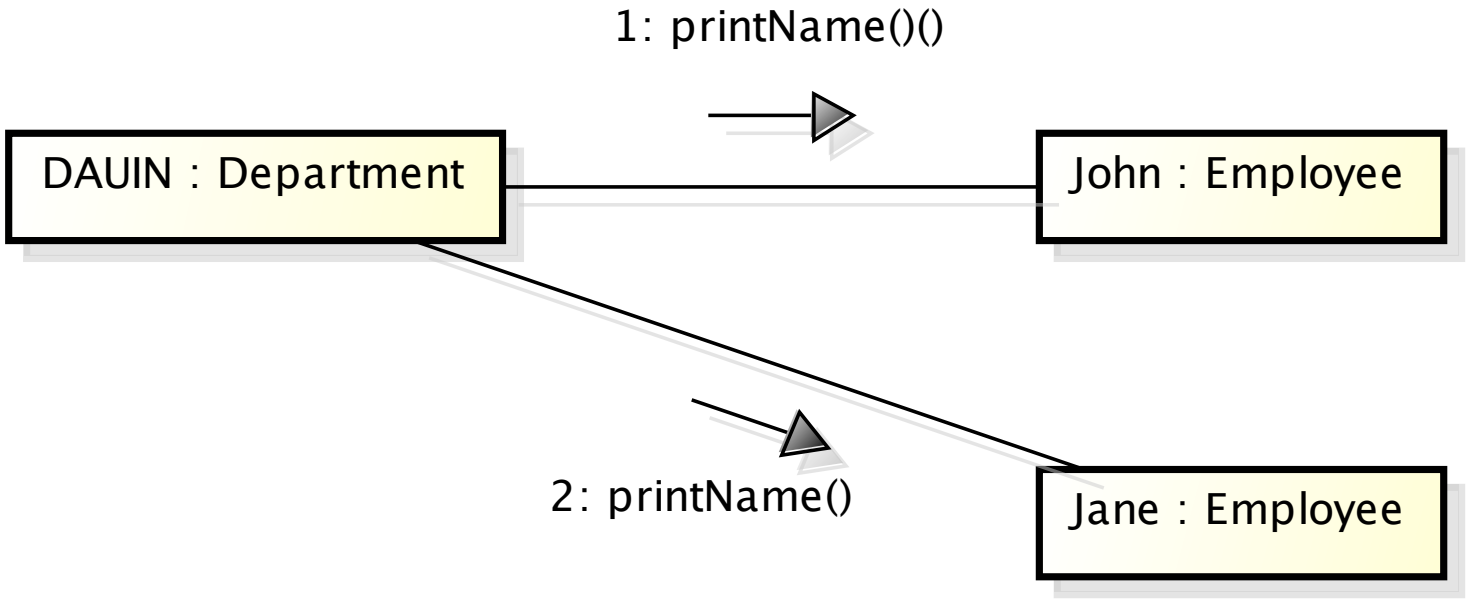


Message passing

- Objects communicate by message passing
  - 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.

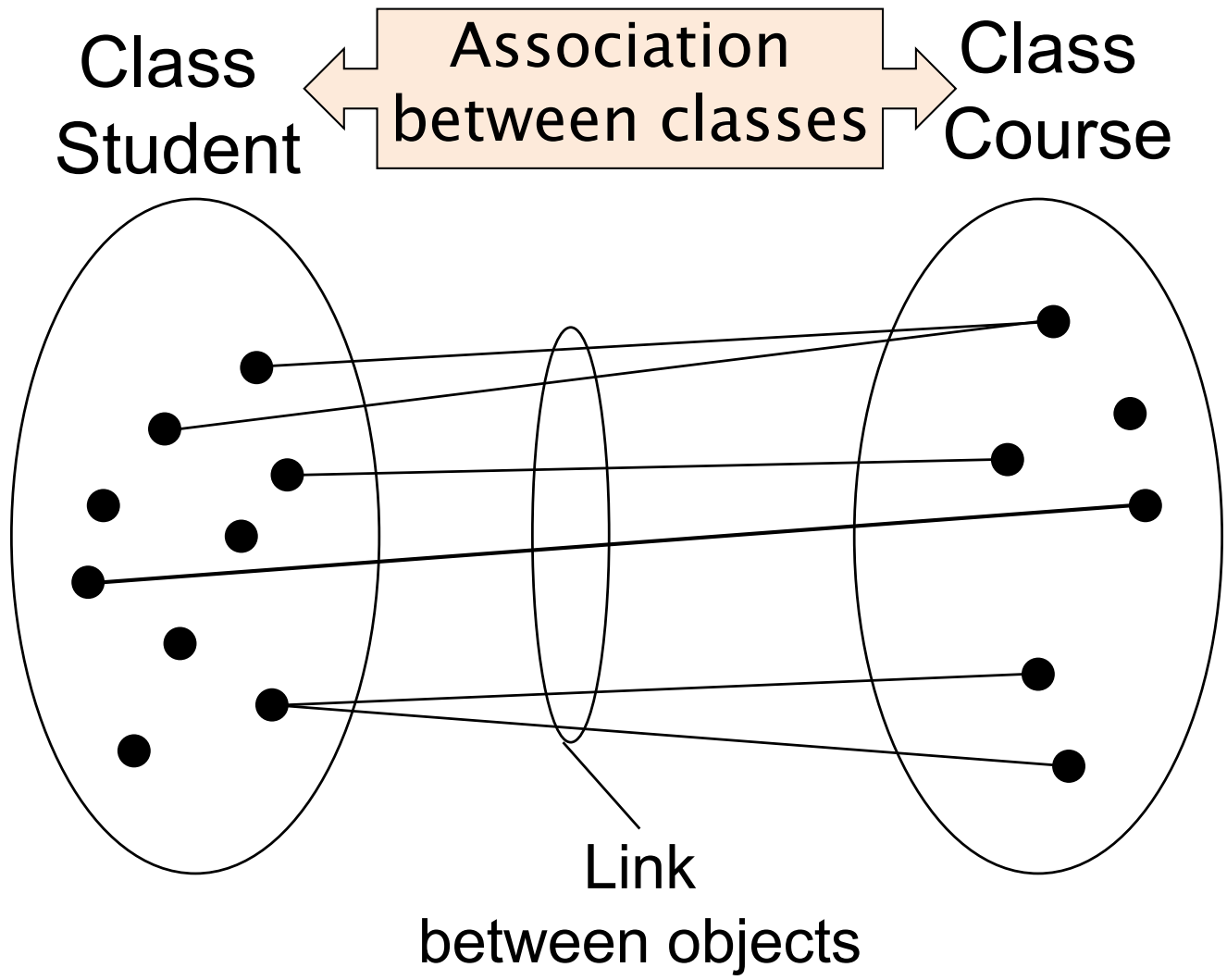
Messages


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

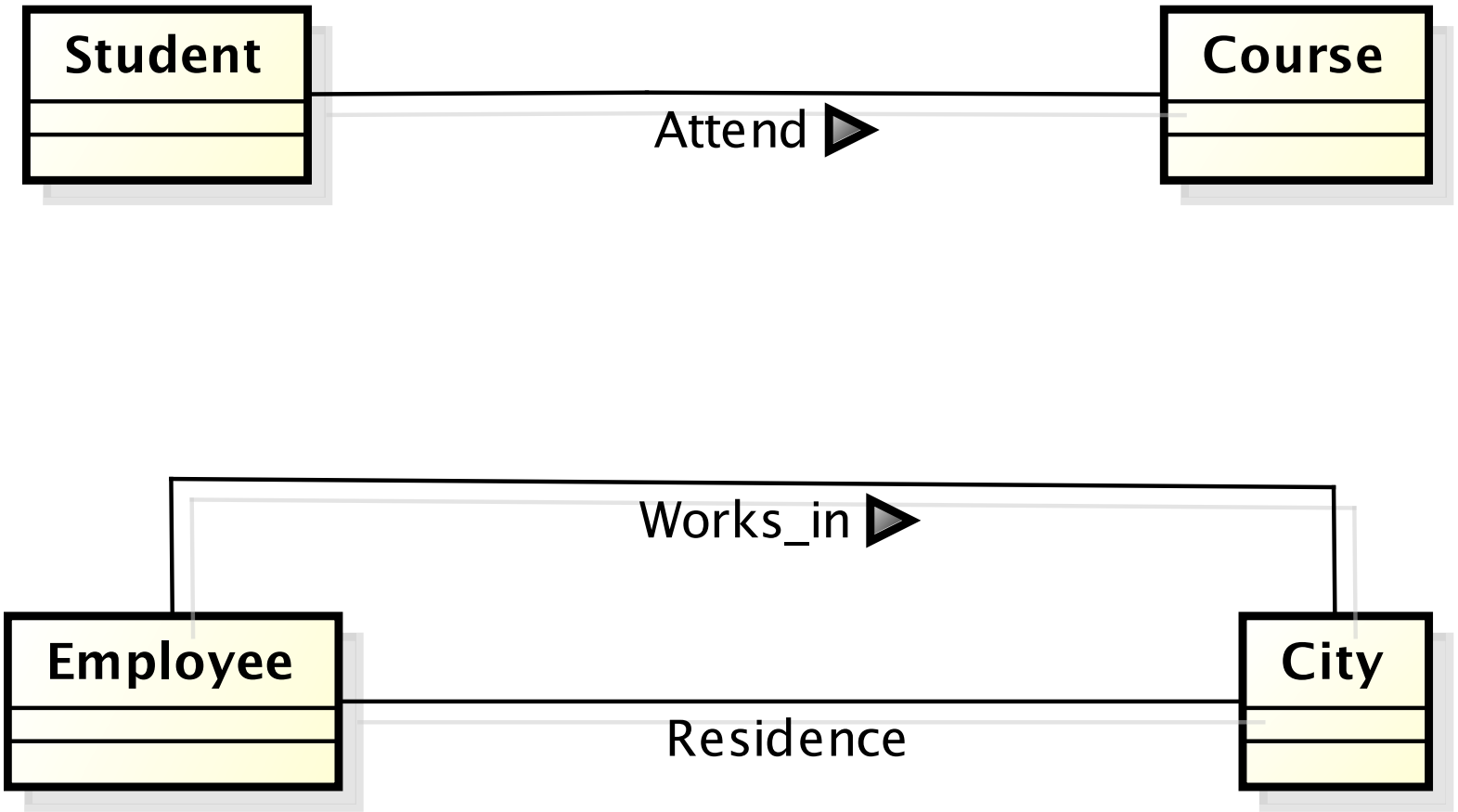
Association

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

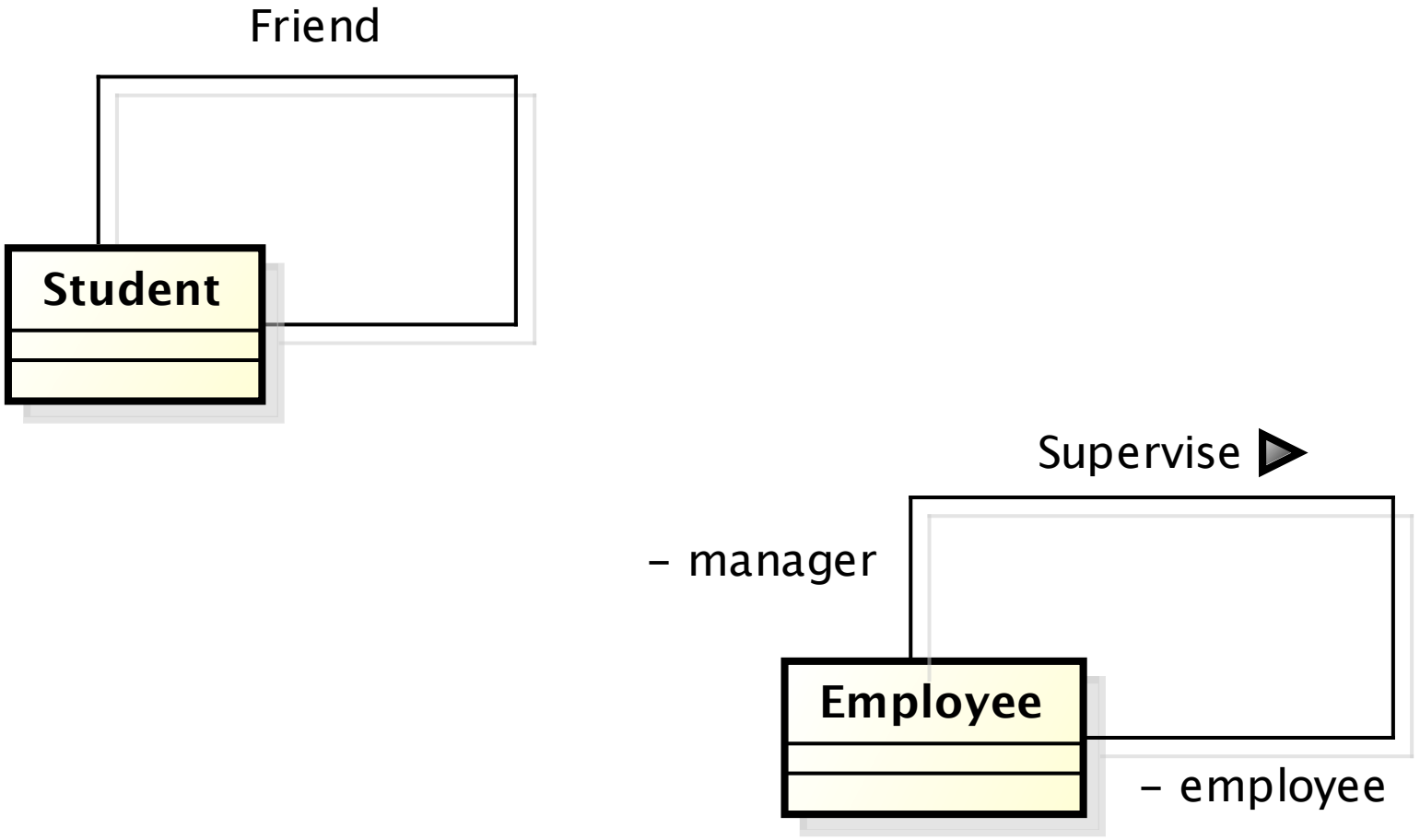
Associations



Association – Examples

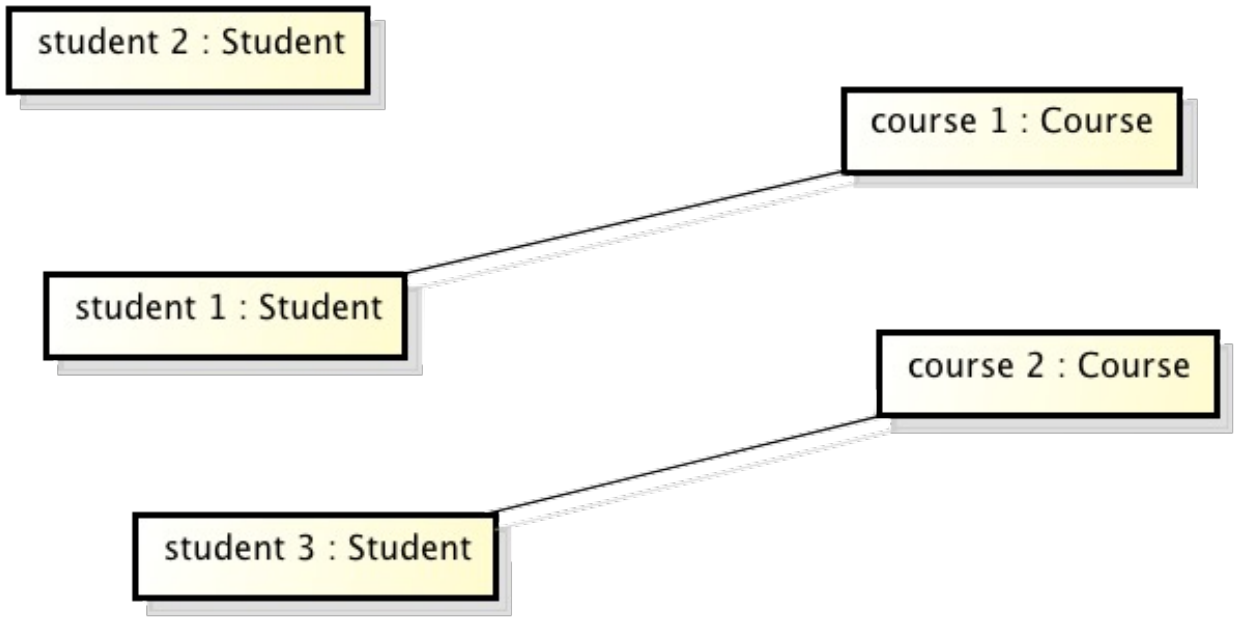


Recursive association-Samples



Link

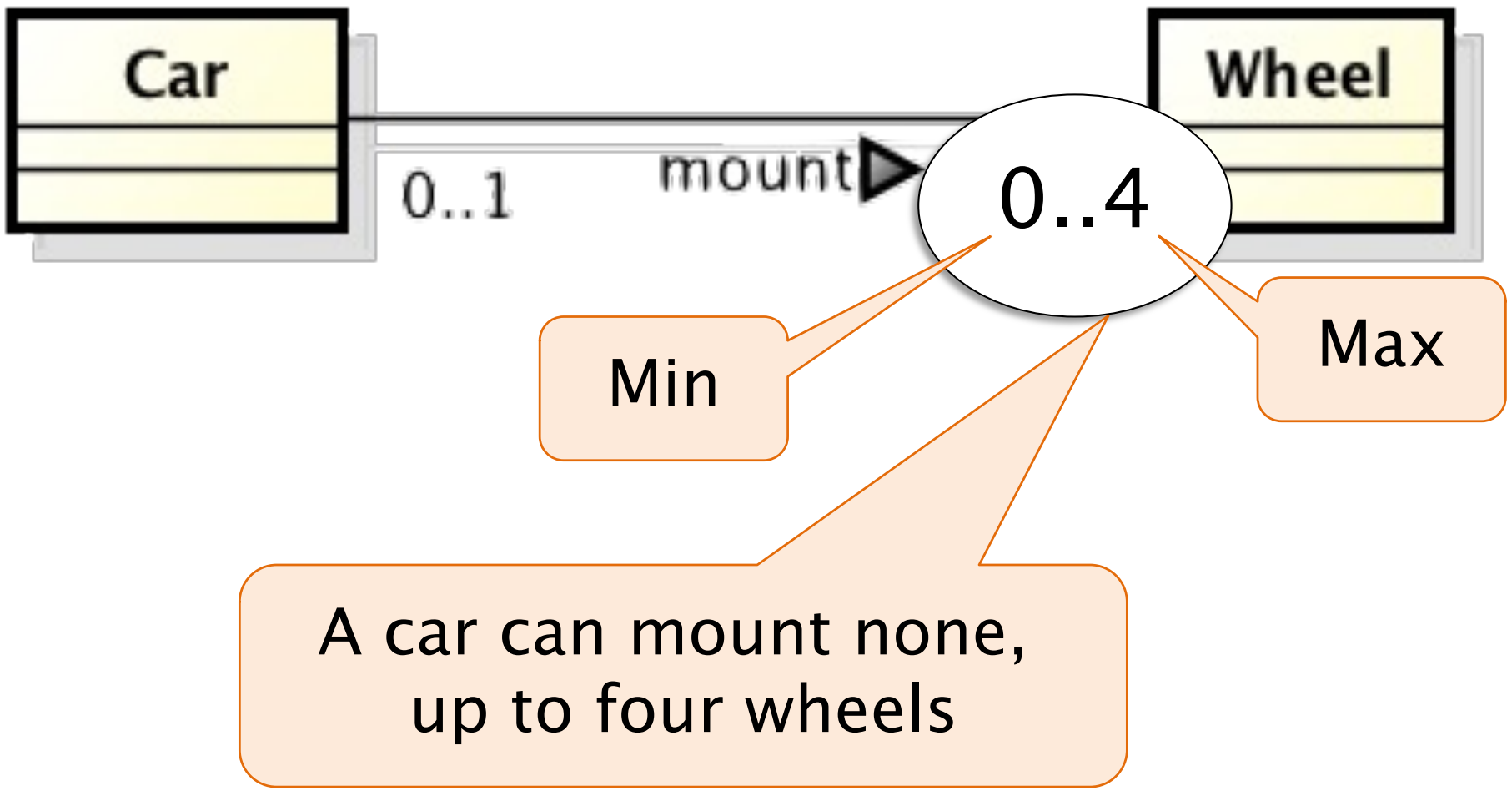
- Model of association between objects



Multiplicity

- Describes 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



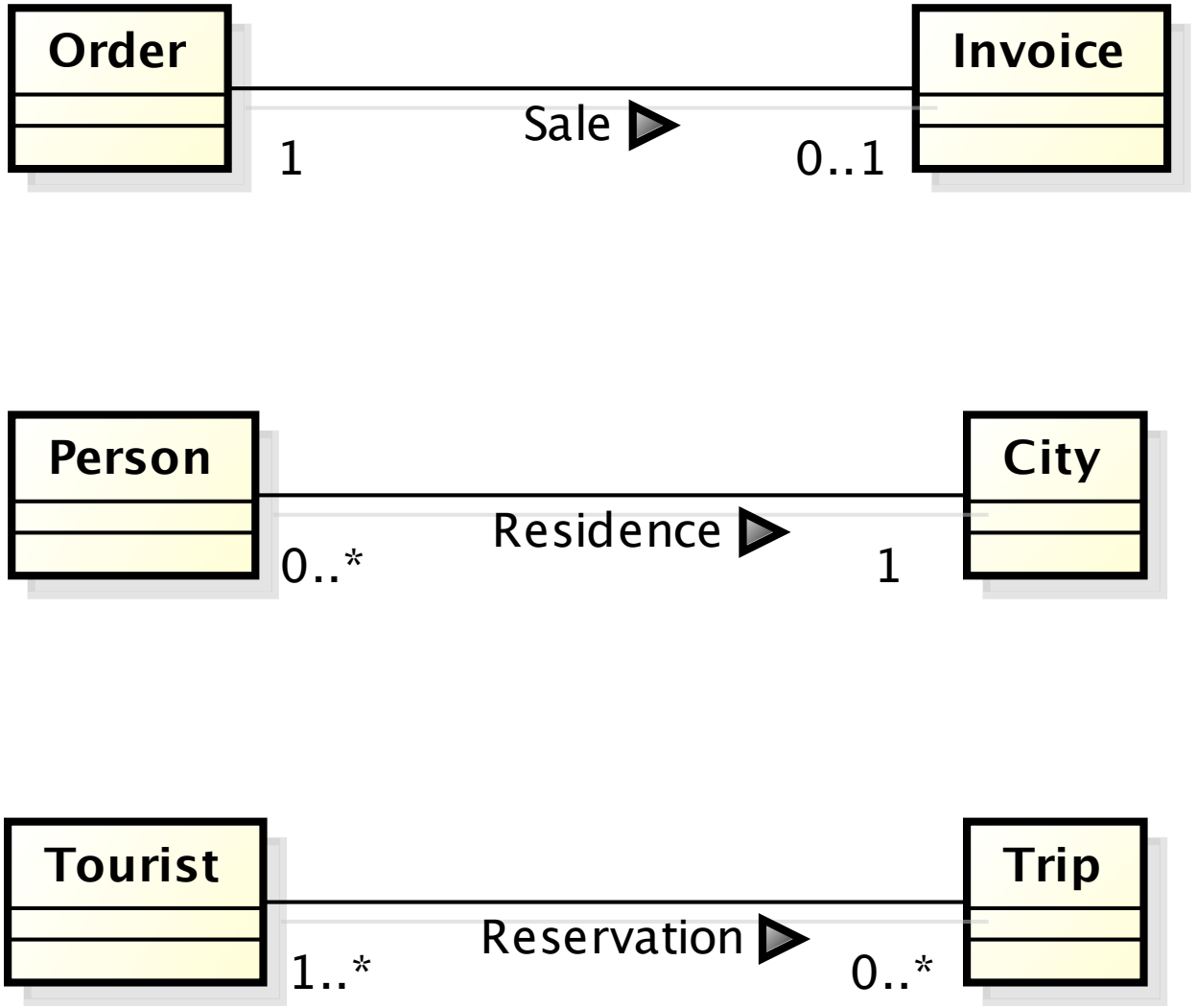
Multiplicity – Example

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: object is involved in at most one link
  - *: each object is involved in many links

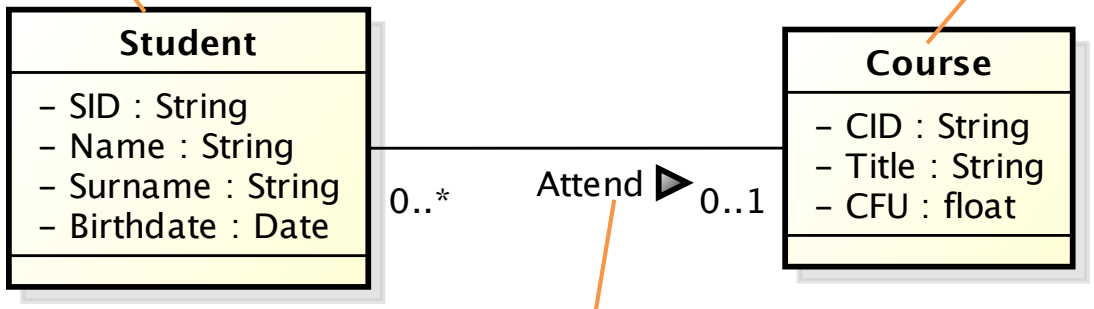
Multiplicity



Operational interpretation

SID	Name	Surname	Birthdate
S2345	John	Smith	1990-4-12
S1234	Jane	Brown	1991-7-11
S5678	Mario	Rossi	1991-11-5

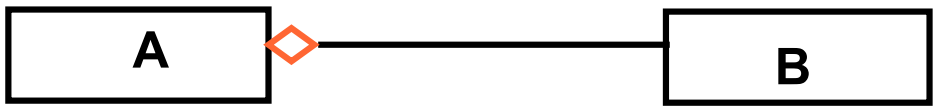
CID	Title	CFU
C001	Information Systems	8
C002	Advanced Programming	10
C003	Calculus	10



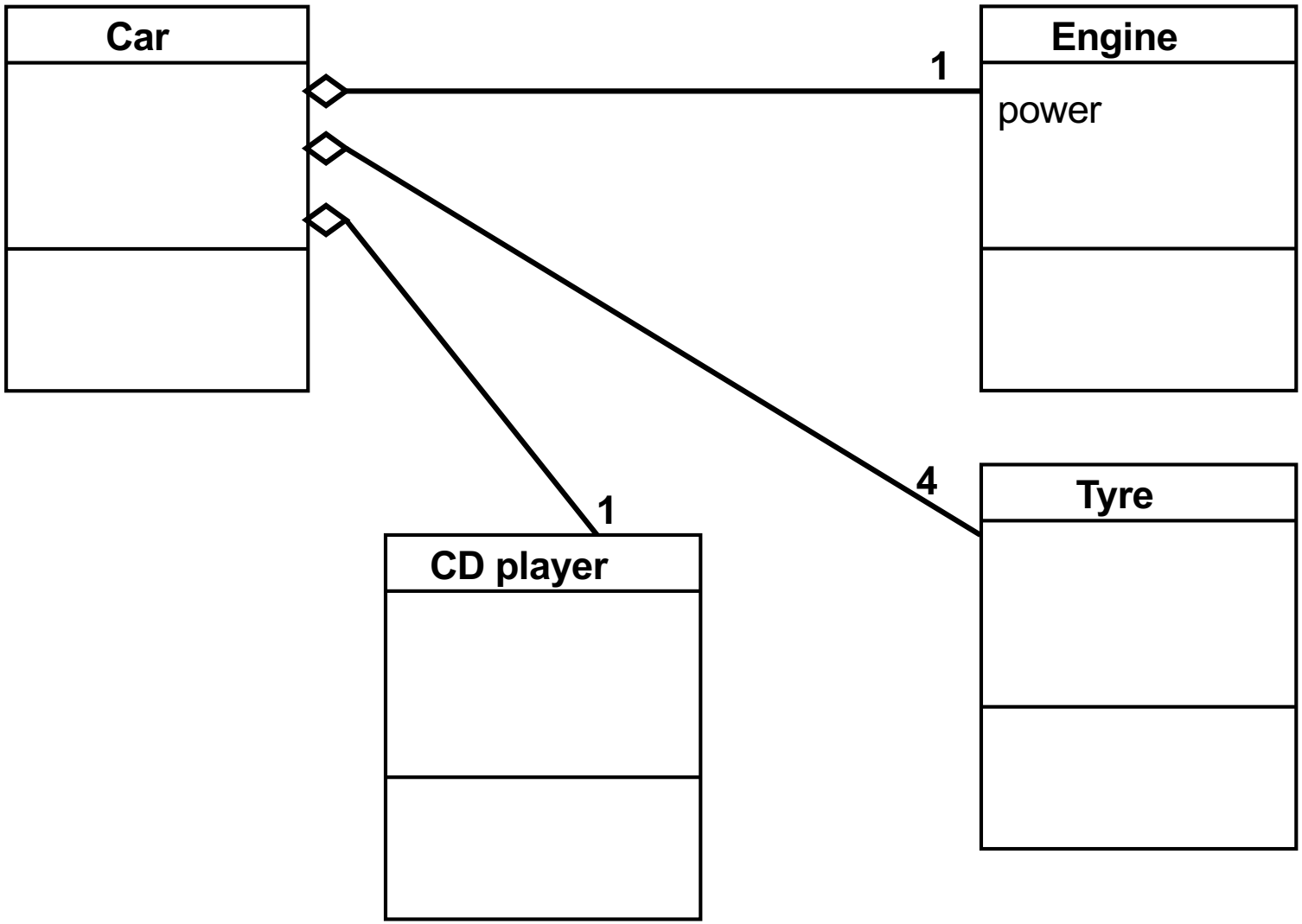
	C001	C002	C003
S2345			X
S1234	X		
S5678	X		

Aggregation

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

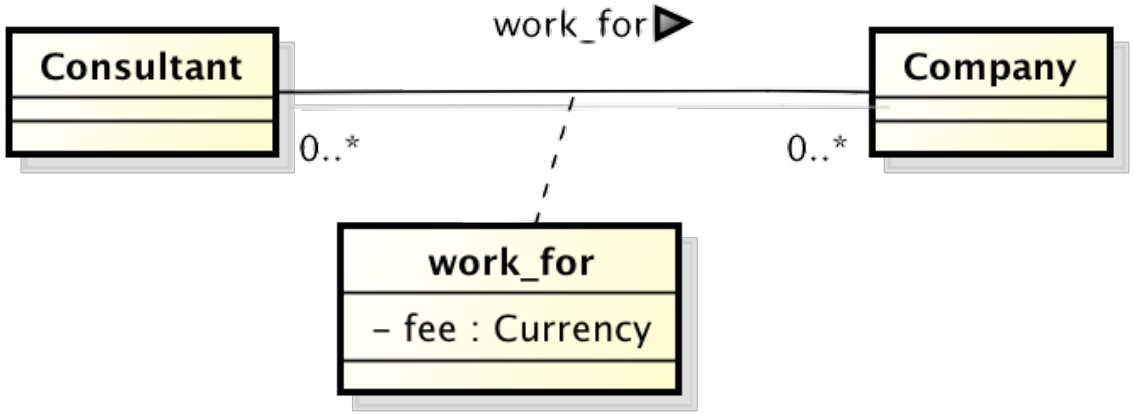


Example

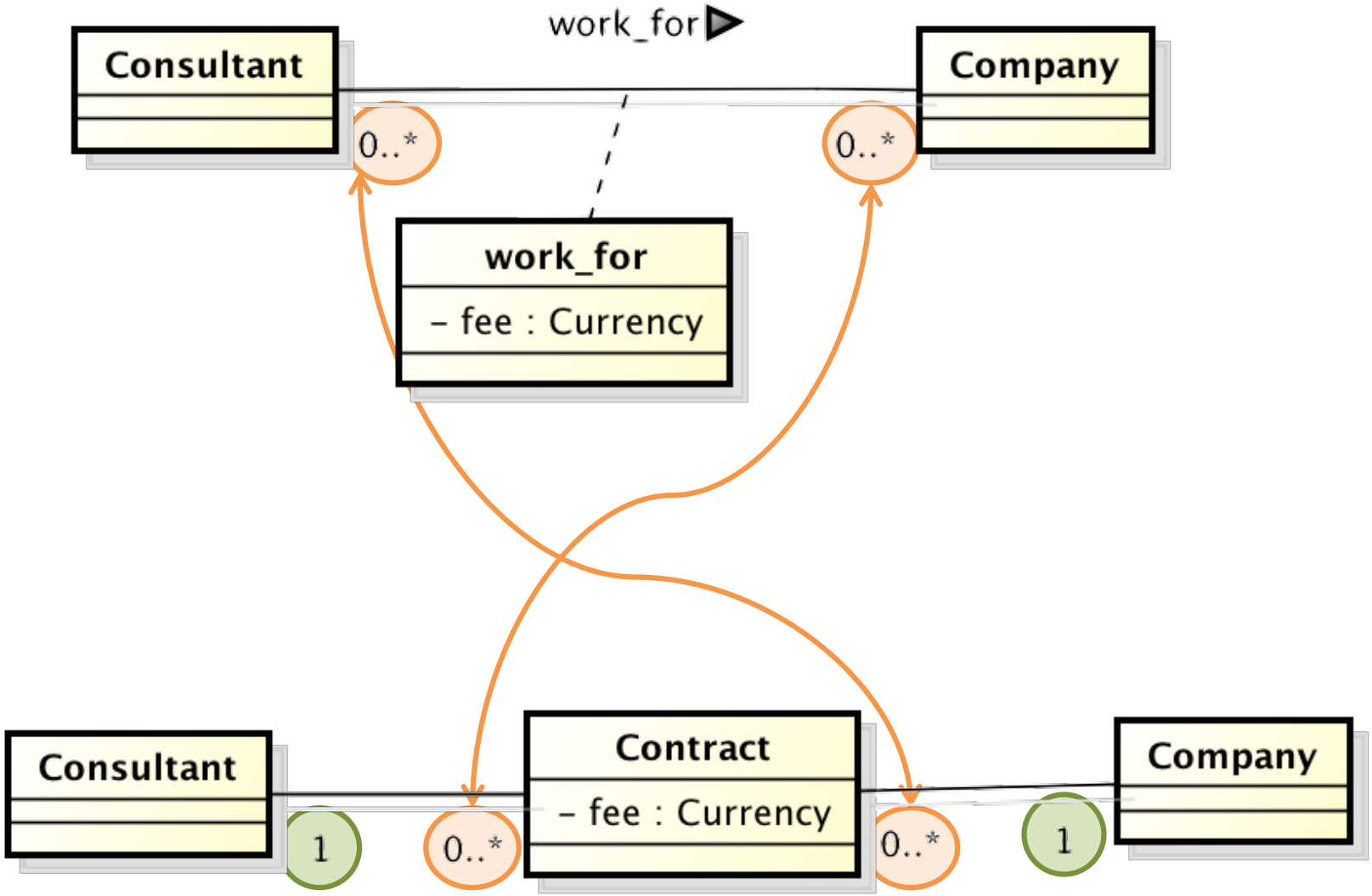


Association Class

- The association class define the attributes related to the association
- A link between two object includes
  - The two linked objects
  - The attributes defined by the association class



Association class – Equivalence



Association Class Limitations

- Association class
  - Fee is a function of consultant and company
  - fee (Consultant , Company)
- Intermediate class
  - Fee is a function of the contract
  - fee (Contract)

Association class limitation

- Case
  - Consultant working several time for the same Company
- Cannot be represented by association class
- Only representable through intermediate class

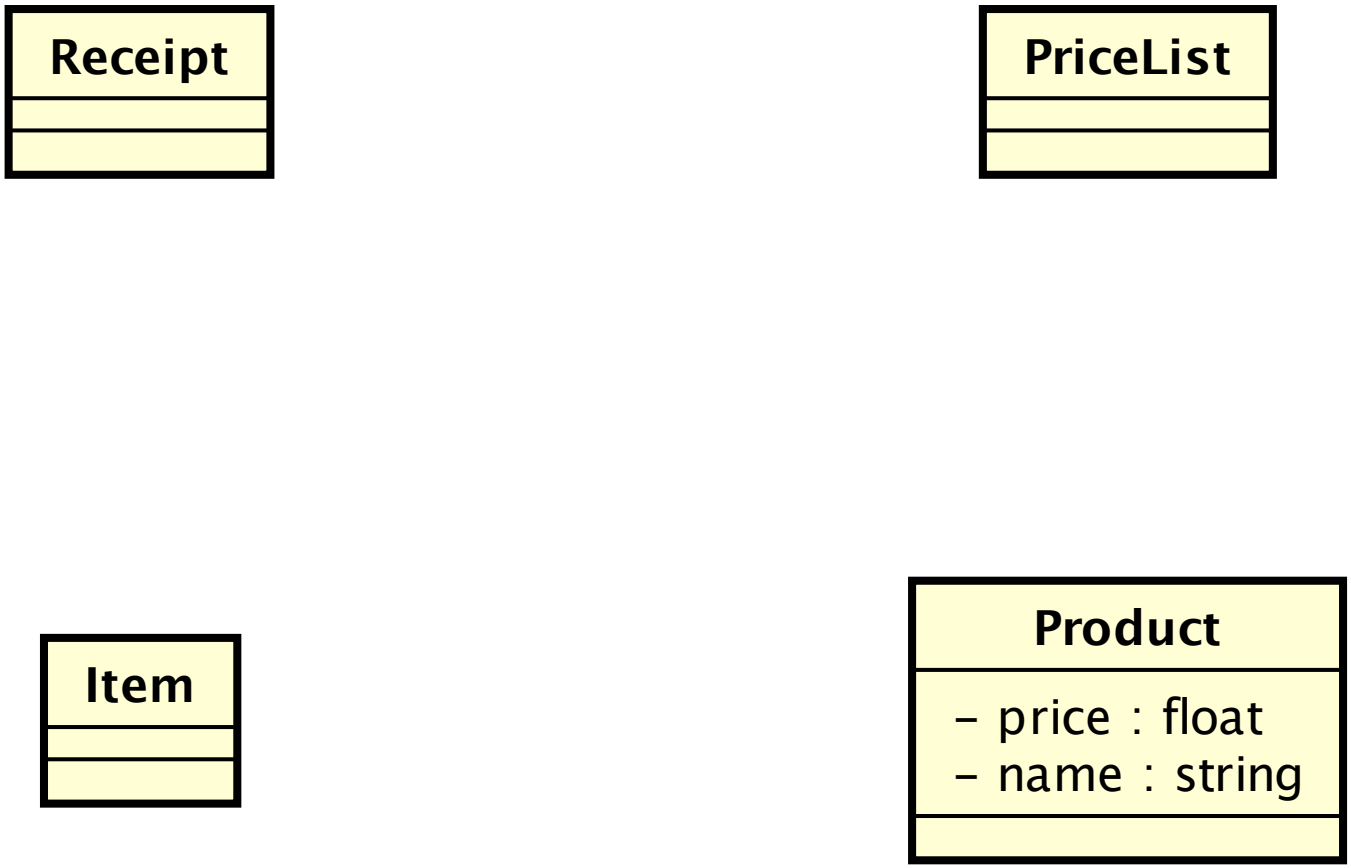
Essential guidelines

- If a concept has significant properties and/or describes types of objects with an autonomous existence, it can be represented by a class.
- If a concept has a simple structure, and has no relevant properties associated with it, it is likely an attribute of a class.
- If a concept provides a logical link between two (or more) entities, it is convenient to represent it by means of an association.
- Any operation that implies access to the attributes of a class should be defined as a method.

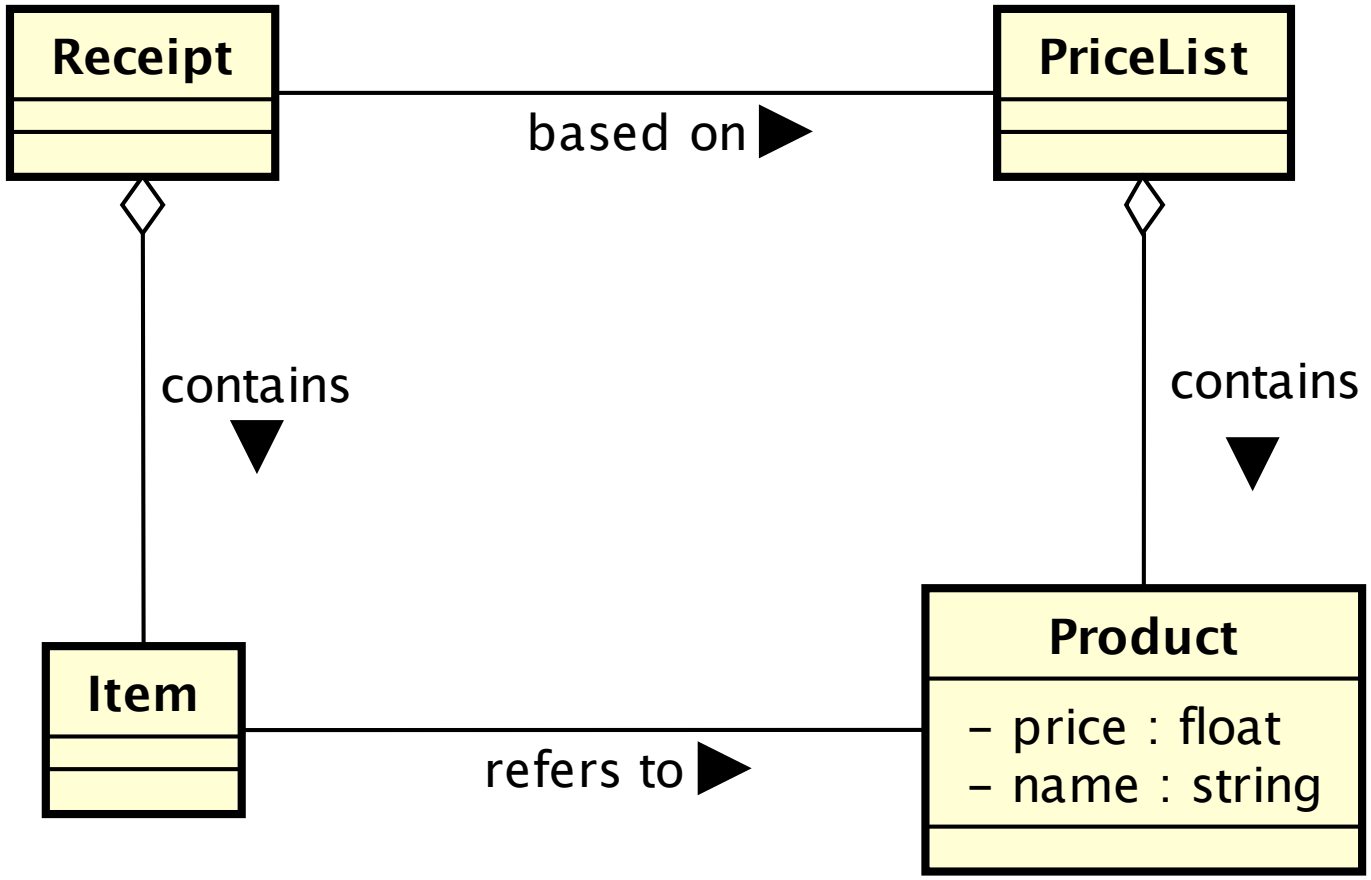
Example – Receipt

- Cash registers emit purchase receipts
- A receipt is made up of items
- Every item correspond to a product that has a name and a price
- Products' info is stored in a price list
- Any time a new product code is entered the corresponding item is added to the receipt
- After the last item is entered, a list of the items (with product name and price) are printed together with the total sum.

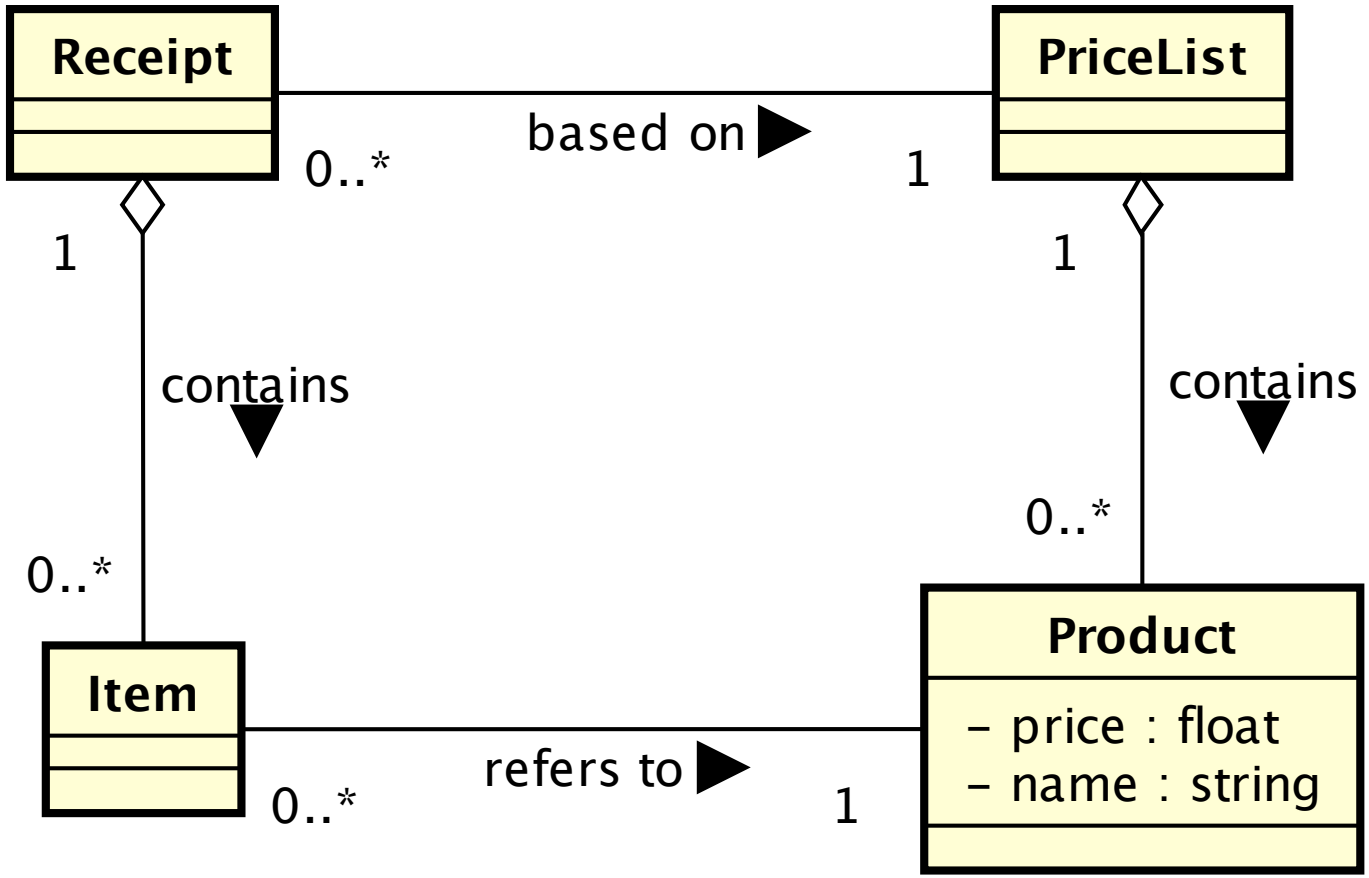
Example – Classes



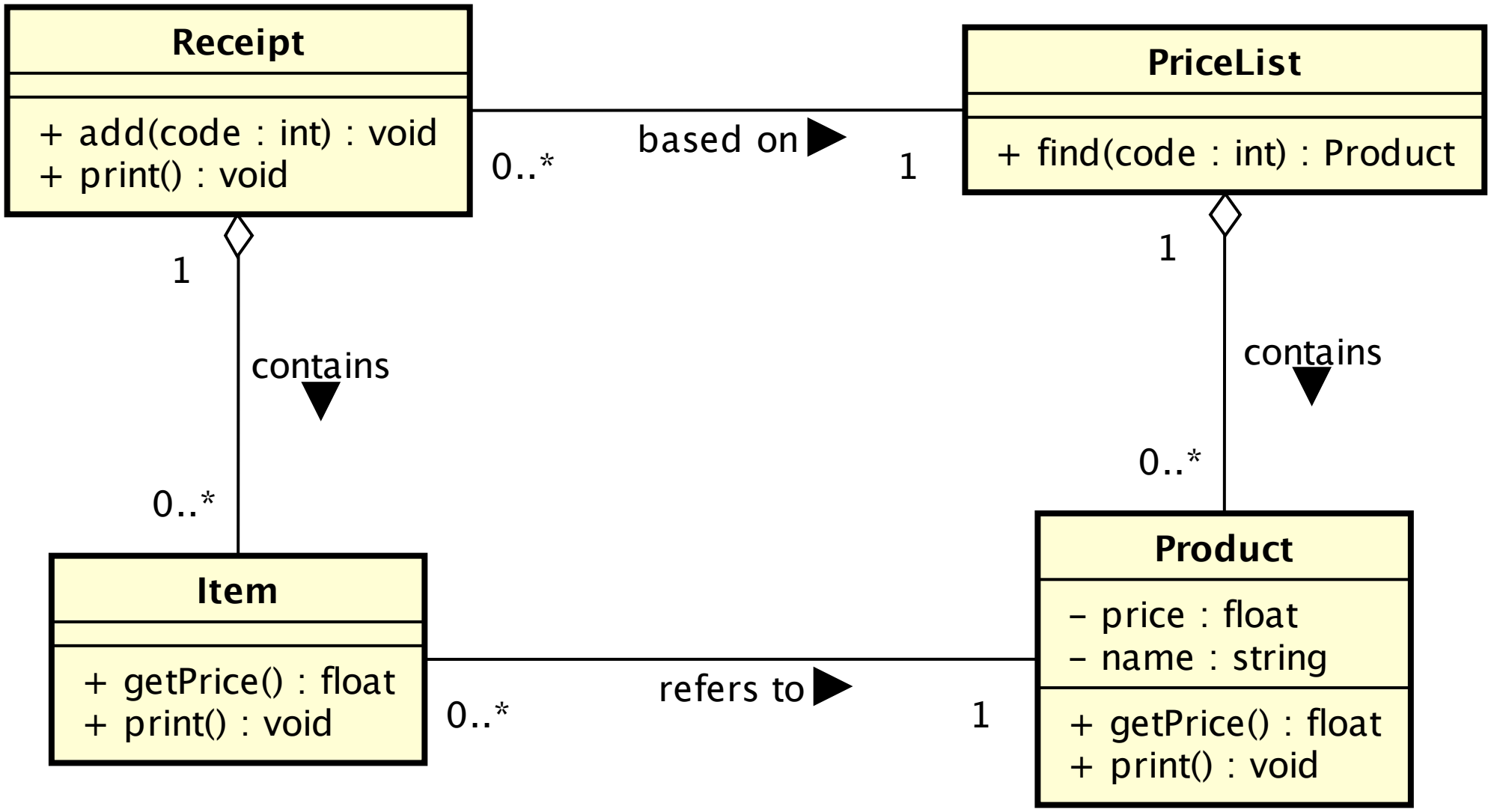
Example – Associations



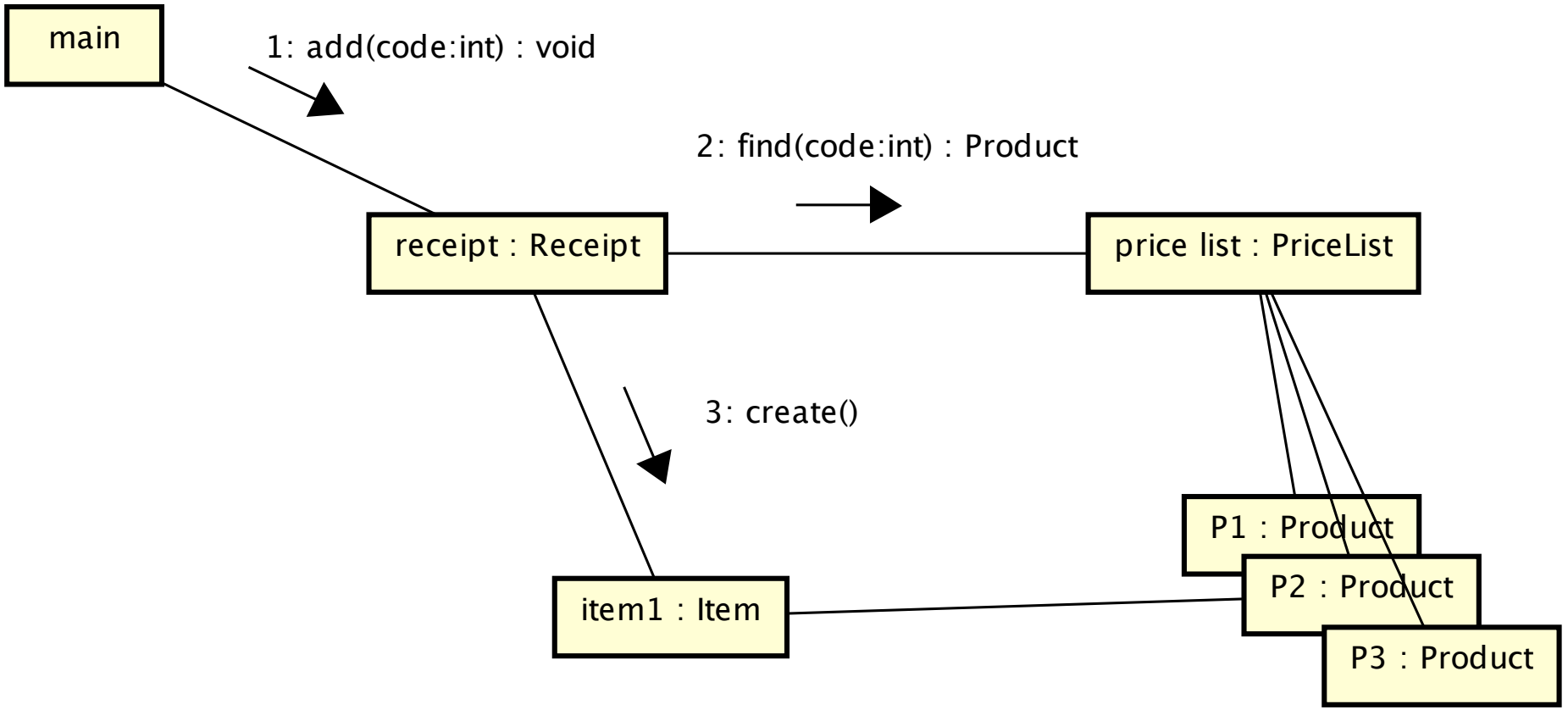
Example – Multiplicity



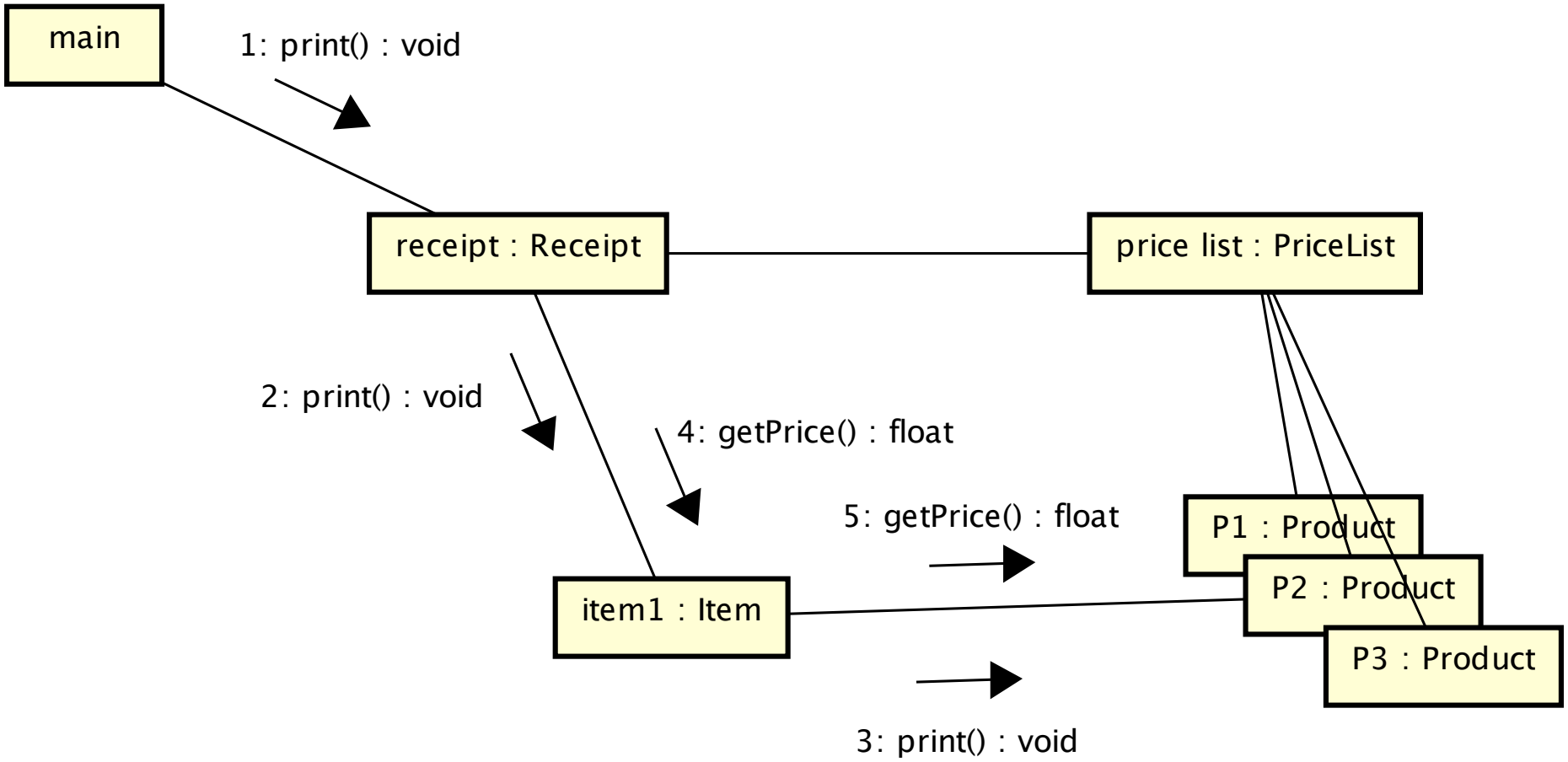
Example – Methods



Example – Messages (Add)



Example – Messages (Print)



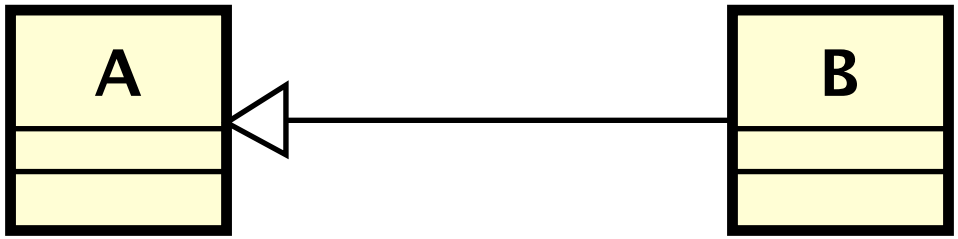
INHERITANCE

Inheritance

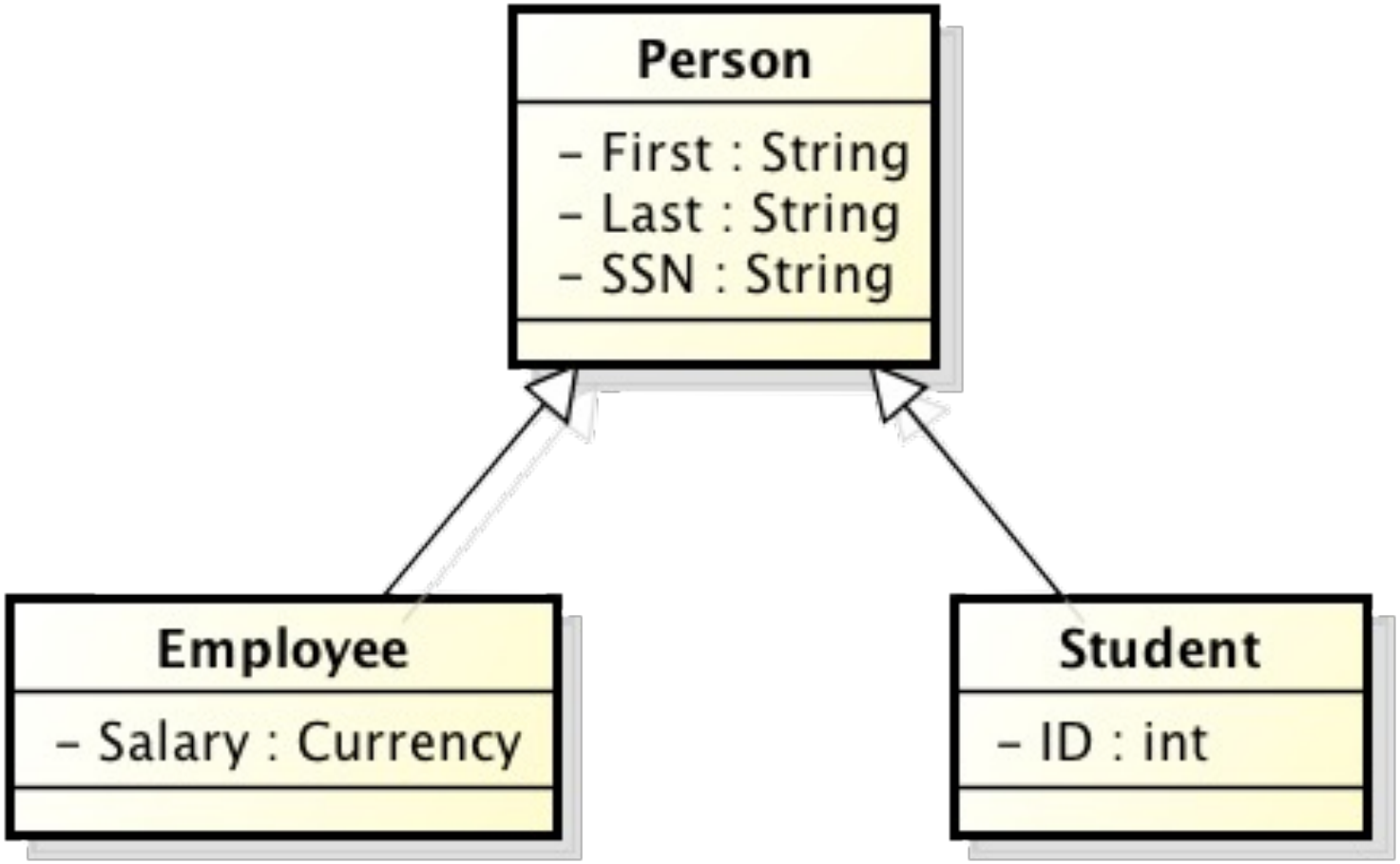
- A class can be a sub-type of another class
- The inheriting class contains all the methods and fields of the class it inherited from plus any methods and fields it defines
- The inheriting class can override the definition of existing methods by providing its own implementation
- The code of the inheriting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

Specialization / Generalization

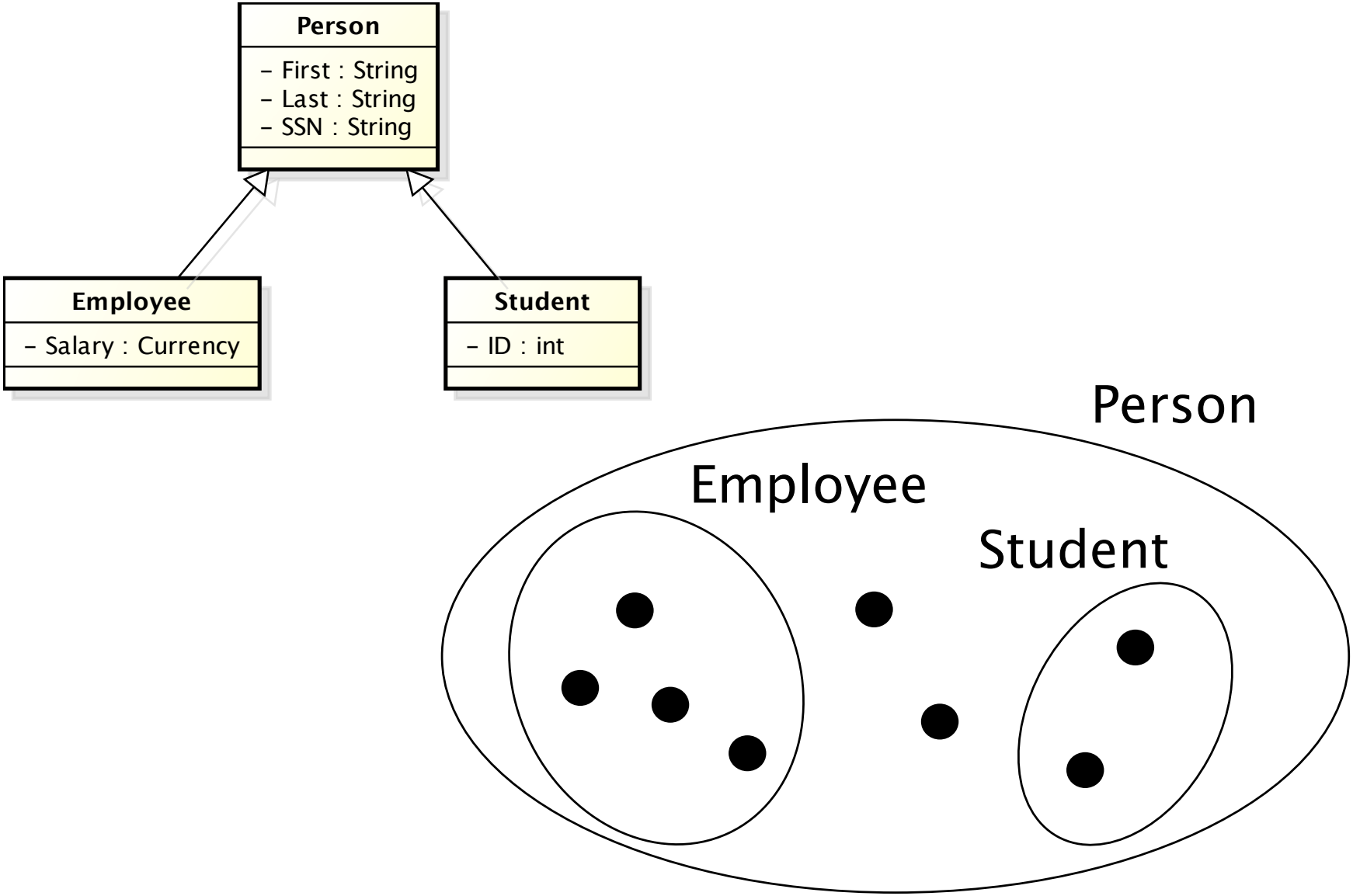
- B *specializes* A means that
  - B has the same characteristics as A
    - Attributes
    - Participation in associations
  - B may have additional characteristics
  - B is a special case of A
  - A is a generalization of B



Generalization



Set-Specialization



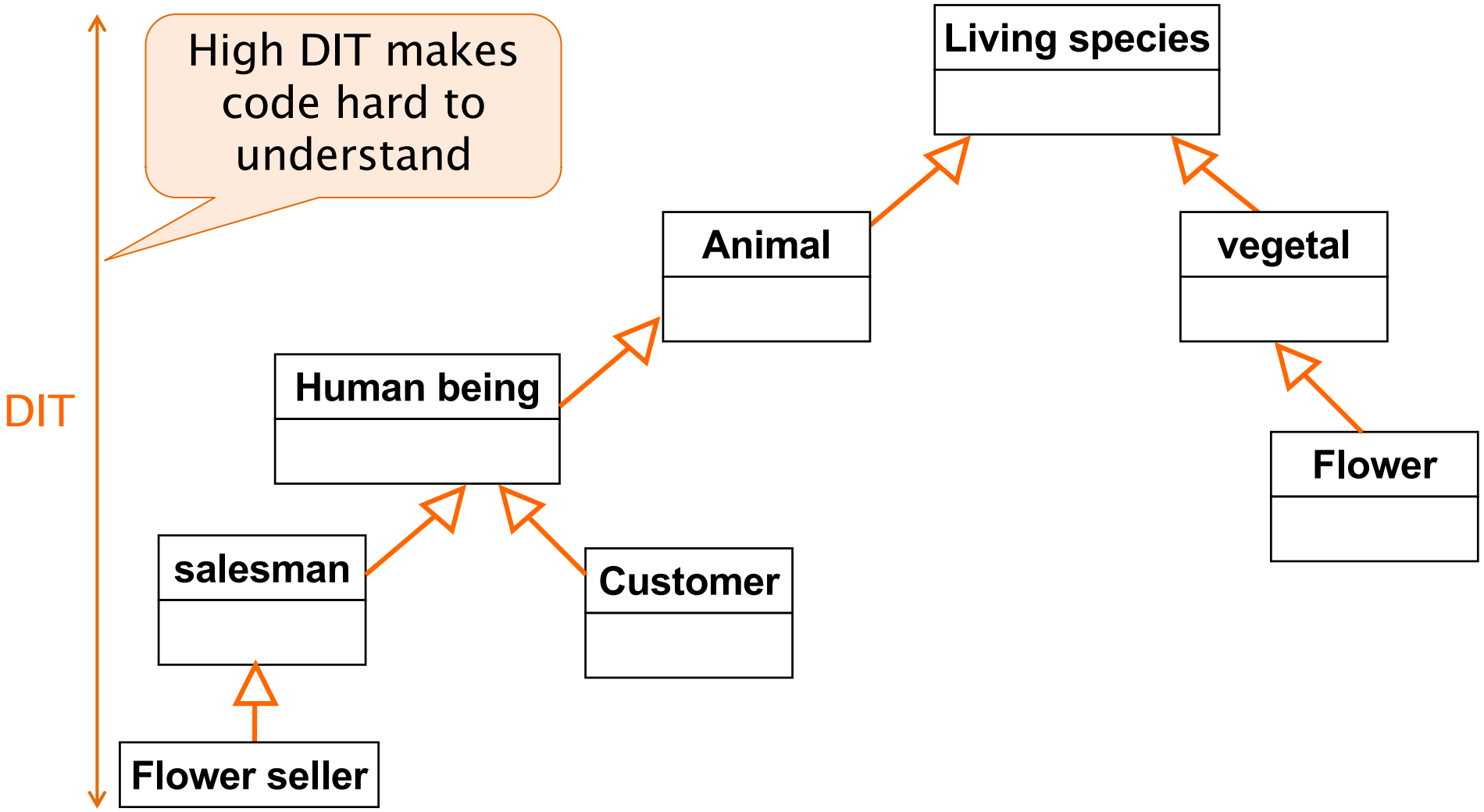
Inheritance terminology

- 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

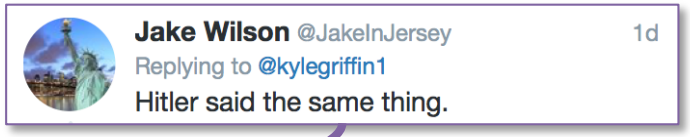


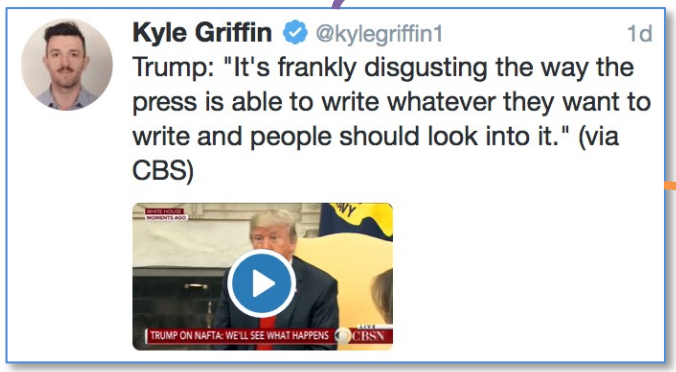
Twitter (simplified)

- A registered user can
  - Post a tweet
  - Follow another user
  - Reply to a tweet
  - Add a like to a tweet

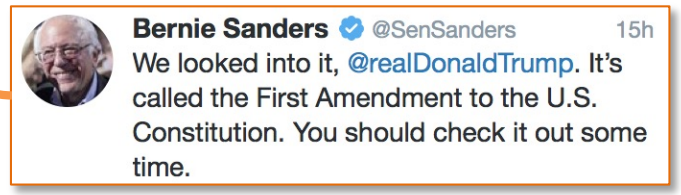
Example

Reply



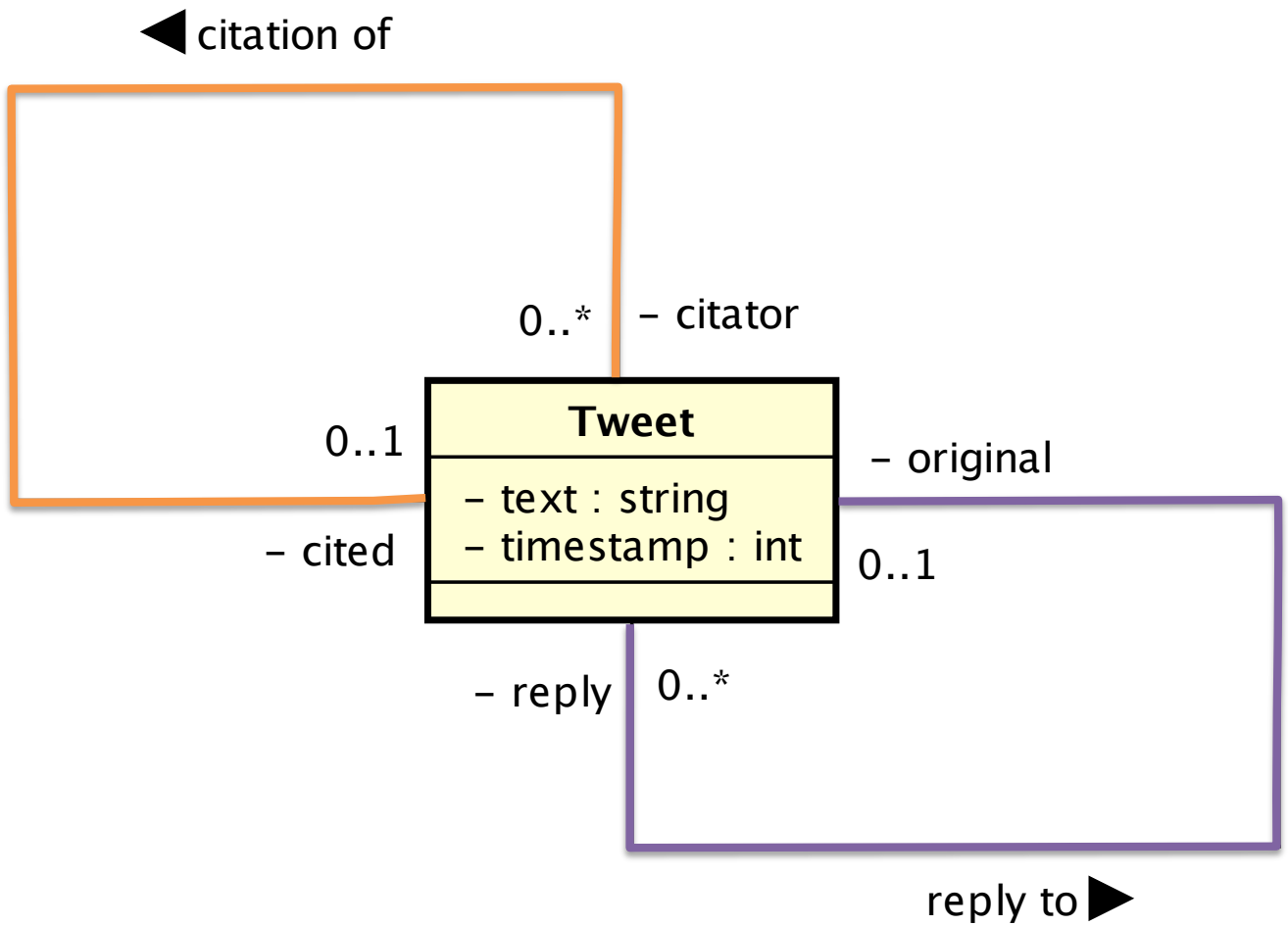


Original

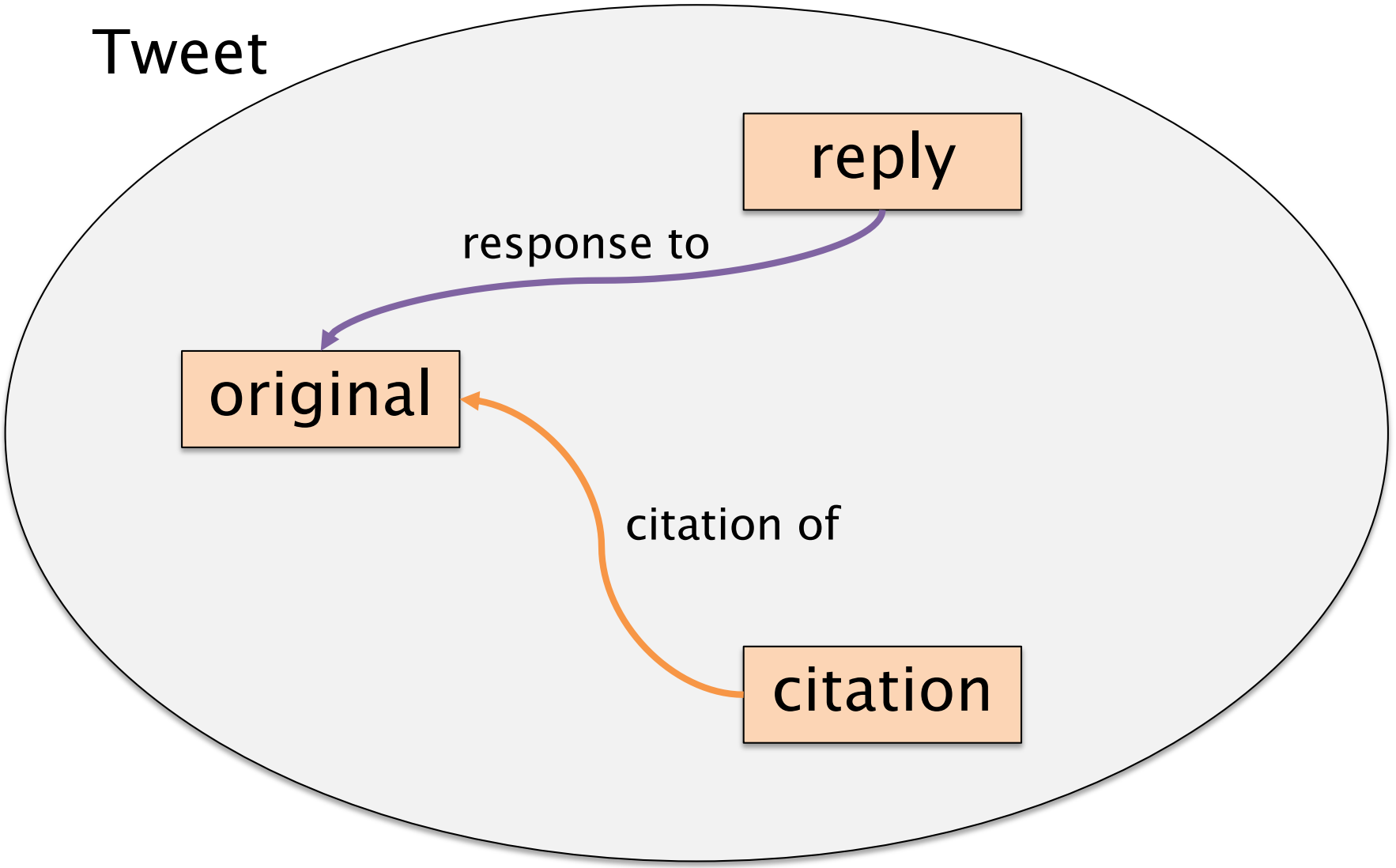


Citation

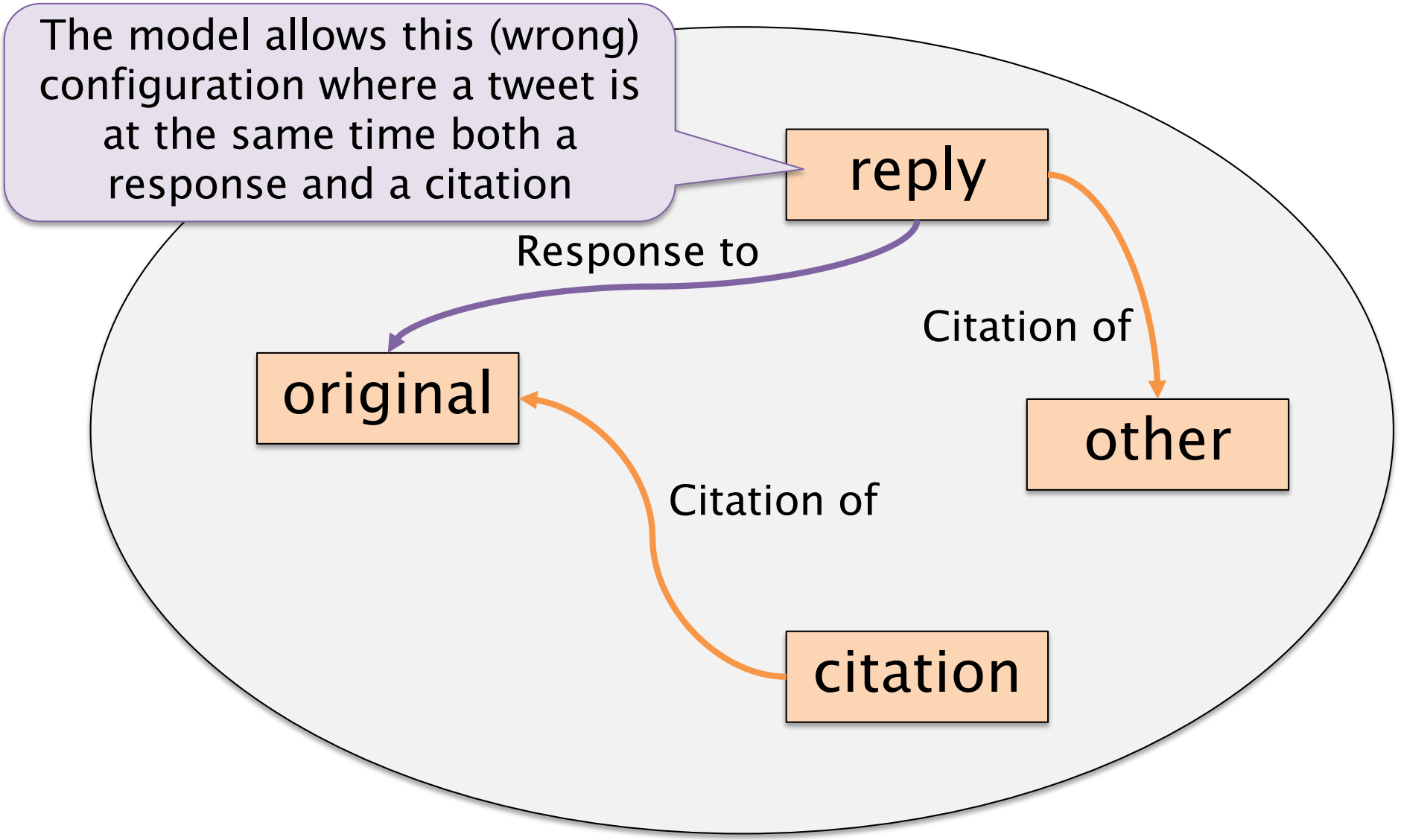
Optional Recursive Associations



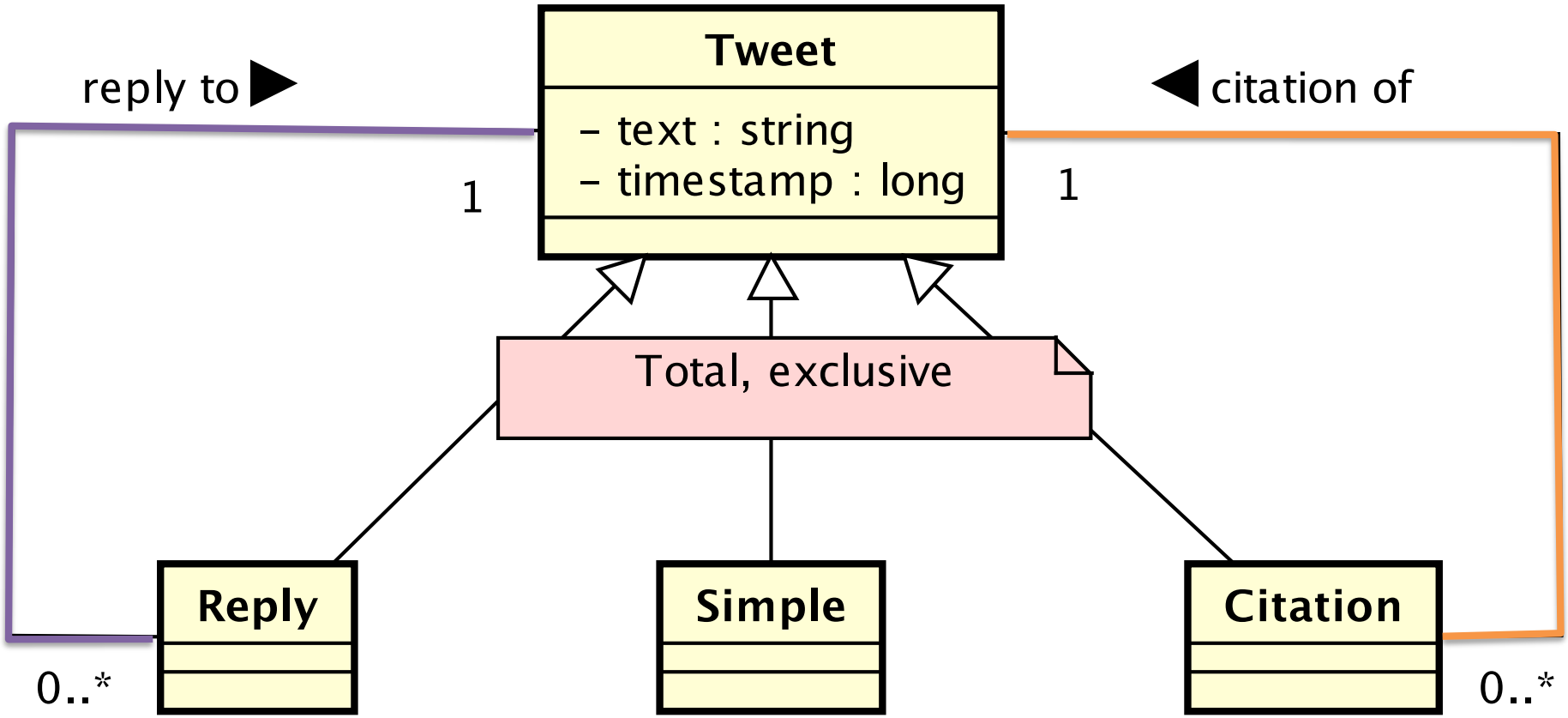
Optional Recursive Associations



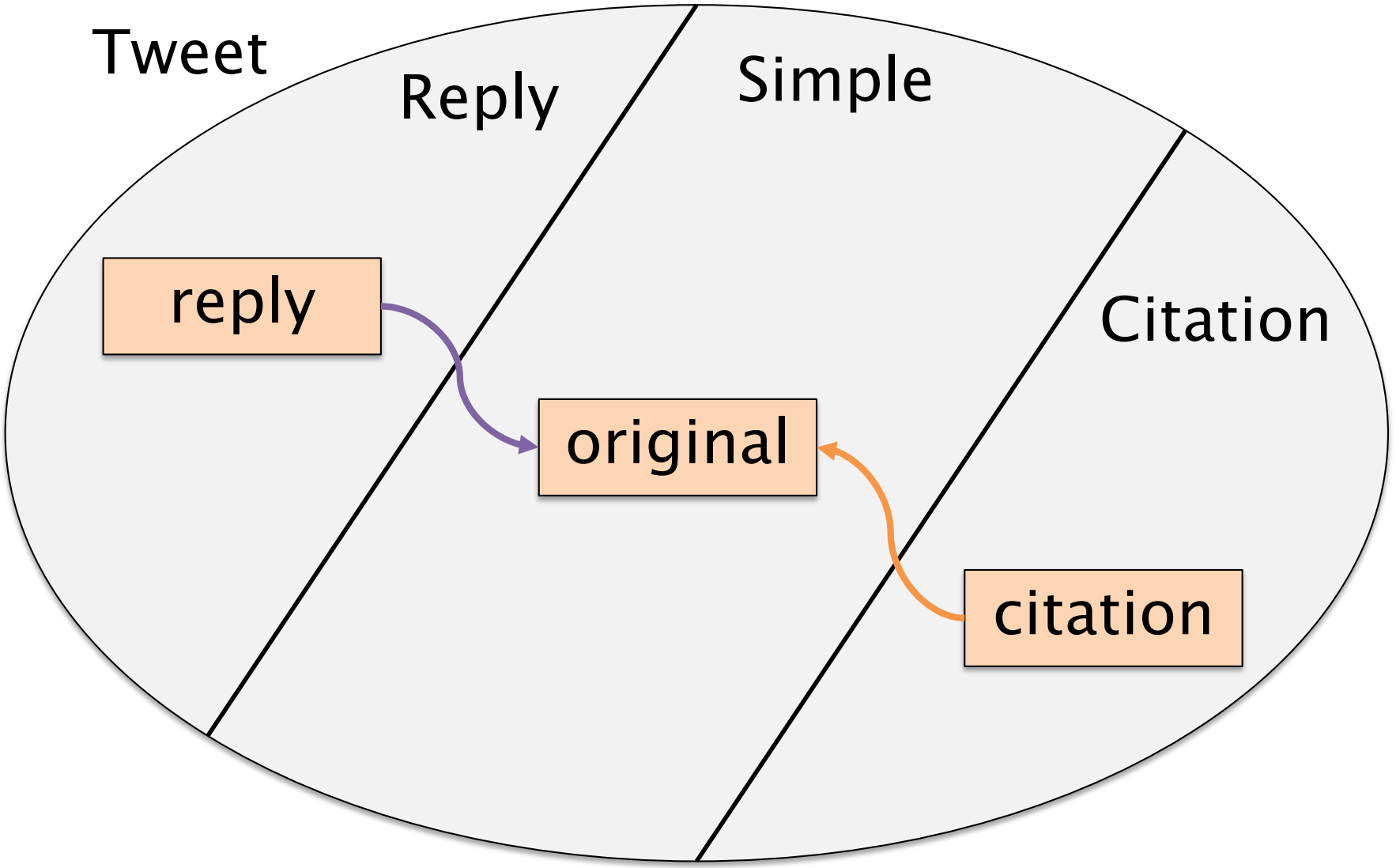
Optional Recursive Associations



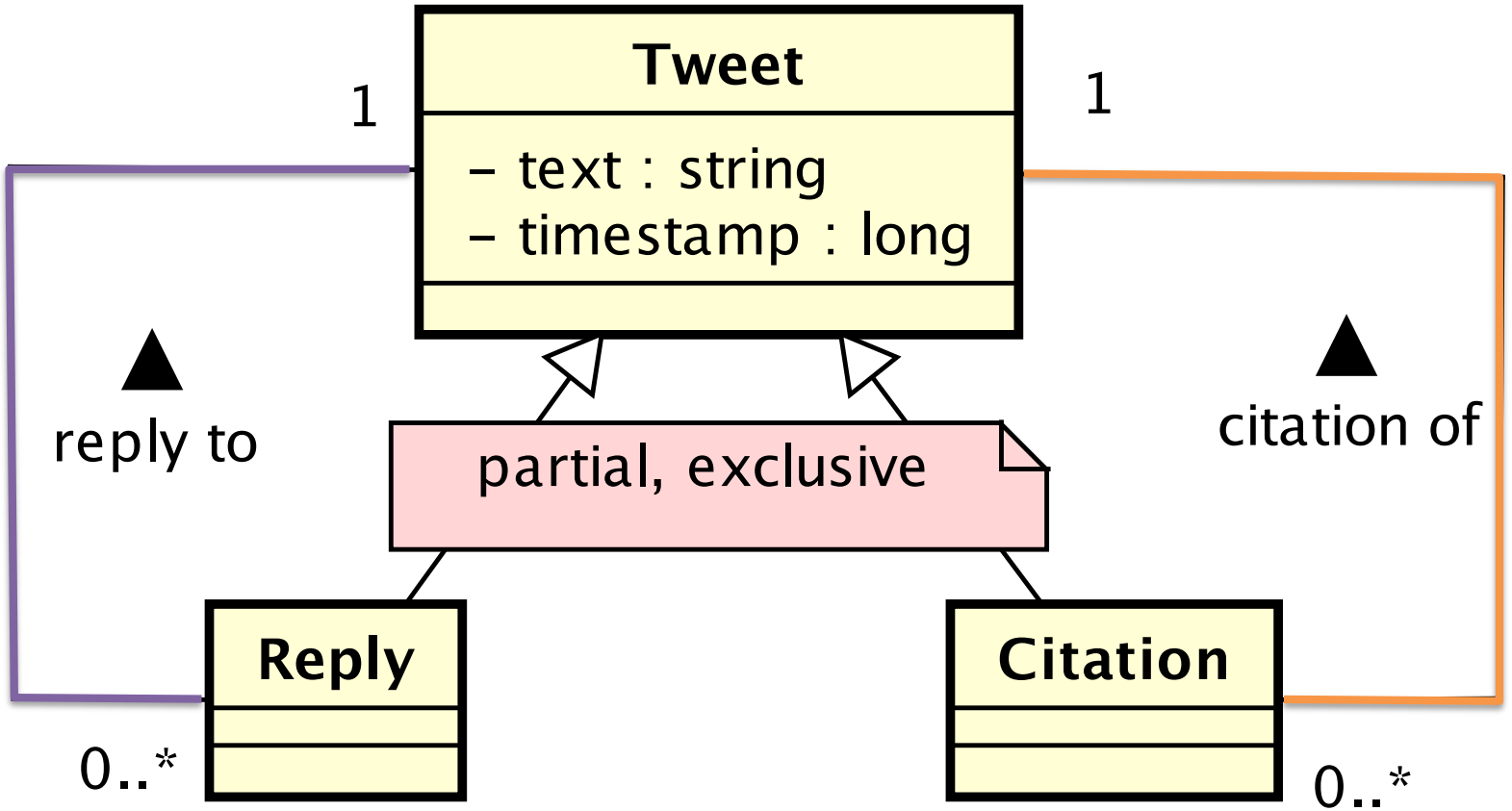
Specialization



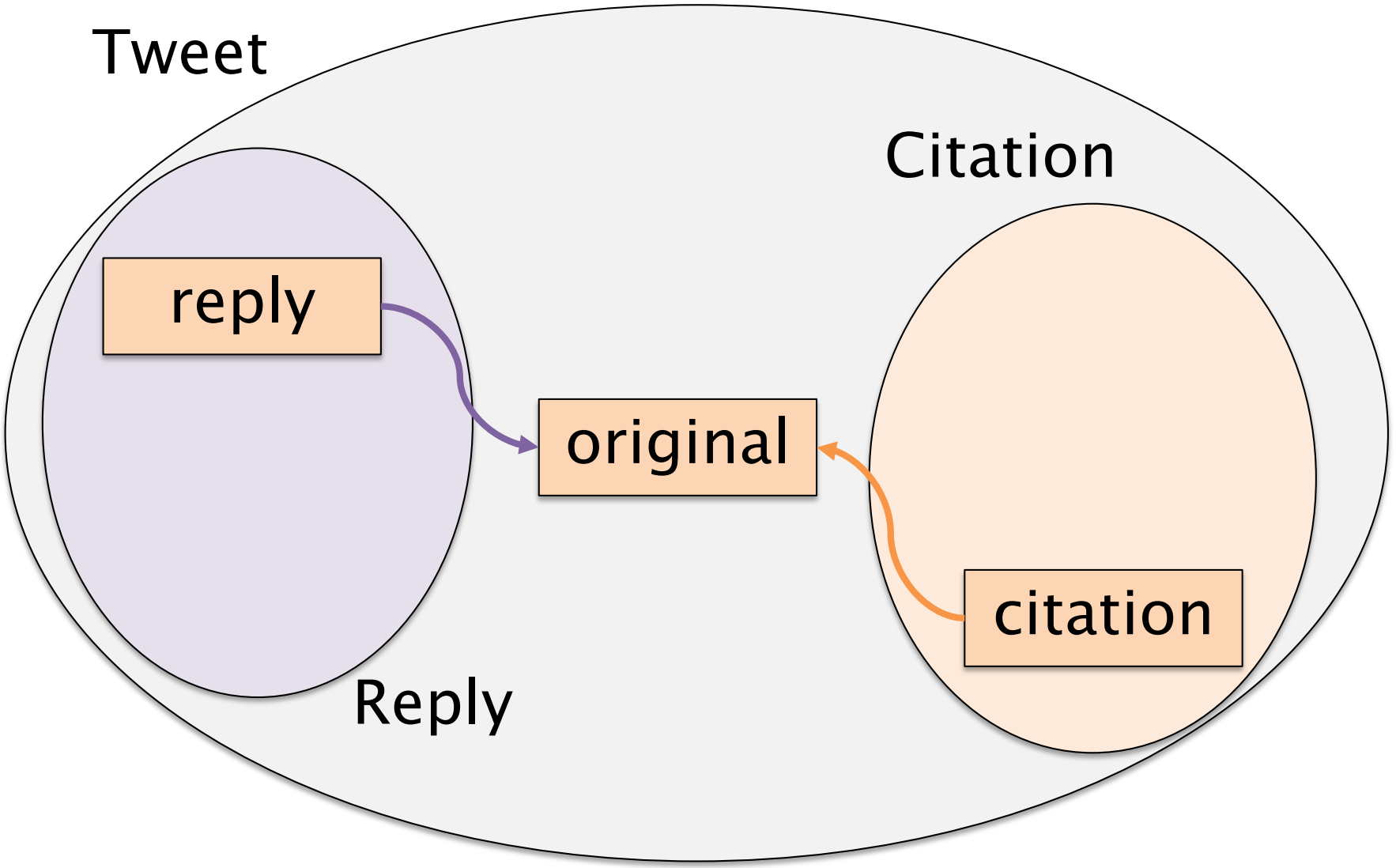
Specialization



Partial Specialization



Partial Specialization



Essential guidelines (II)

- If one or more concepts are special cases of another concept, it is convenient to represent them by means of a generalization.
- When distinct classes may play the same role w.r.t. an association to a given class it is common to represent this commonality by generalization
  - Inheritance includes also associations

Modeling strategies

- Top-down
  - Start with abstract concepts and perform successive refinements
- Bottom-up
  - Start with detailed concepts and proceed with integrating different pieces together
- Inside-out
  - Like bottom-up but beginning with most important concepts first
- Hybrid

Model quality

- Correctness
  - No requirement is misrepresented
- Completeness
  - All requirements are represented
- Readability
  - It is easy to read and understand
- Minimality
  - There are no avoidable elements

References

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