Object Oriented Programming

UML

UML (1)

- A **modelling language** describes the models of the application to develop.
  - Increases readability (less information than code, enabling a global view of the application).
  - Shows the structure of the application, without implementation details.
  - Graphic representation increases semantic clearness.
  - Due to complexity, OO programs are described through diverse models, where each model describes a particular aspect of the application.

UML (2)

- **UML** (*Unified Modeling Language*) combines techniques from different analyses systems:
  - Booch (G. Booch)
    - Focused on the design, that is, in the passage from the problem space to the solution space.
  - *Objected Oriented Software Engineering,* OOSE (I. Jacobson)
    - First methodology that included steps from specification to implementation.
  - *Object-modeling technique,* OMT (J. Rumbaugh)
    - Focused on the analysis for the identification, specification and description of the functional requirements of the system
- 1st proposal made available in 1997
  - 1999 - v1.1, 2000 - v1.3, 2001 - v1.4, 2003 - v1.5, Abril 2004 - **v2.0**
- Tools: Rational Rose, Visual Paradigm, ArgoUML, Dia, etc

UML (3)

- UML v2 makes available 13 diagrams, grouped in:
  - Structural diagrams (package, class, object, composite structure, component, deployment and profile).
  - Behaviour models (use case, activity, state machine, communication, sequence, timing, interaction overview).

- In OOP we cover only UML central diagrams, in the sequence:

  concept ⇒ UML representation ⇒ Java implementation

Classes – definition

- A **class** is a template for objects defined by:
  - **Identifier**
  - **Attributes/fields** (that define the state of the objects) with possible values:
    - primitive types (integers,…)
    - references to other objects (identifying relations between objects)
  - **Methods** (operations that might update the state of the objects)

- Attributes and methods are designated as **members of the class**.

Classes – example (1)

"A bank account contains always a balance and belongs to a person. A person has a name, a phone number and an ID number. It is possible to deposit and withdraw money to and from the bank account, respectively. The owner of the account may consult its balance".

Classes

- Account
- Person

Primitive attributes

- Account: balance (float)
- Person: name (string), phoneNb (long), idNb (long)

Classes – example(2)

Reference attributes

- Account: owner (instance of a Person)

Methods

- Account:
  - withdraw (parameter: amount to withdraw)
  - deposit (parameter: amount to deposit)
  - balance (return: account balance)
- Person:
  - phoneNb (return: phone number)
  - idNb (return: ID number)

Methods – definition (1)

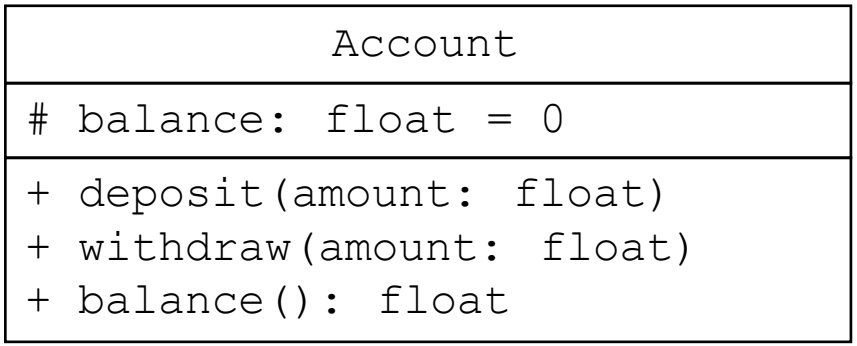
- A **method** is a sequence of actions, executed by an object, that may read and/or write the state of the object (value of the attributes).
- The value of the attributes reside in the object, whereas the methods reside in the class.
- **Signature** of a method:
  - method identifier;
  - identifier and type of the parameters;
  - return value.

Methods – definition (2)

- Methods are categorized as:
  - **Constructor**: executed when building the object.
    - Usually it has the same identifier as the class.
    - Never return a type.
    - Cannot be called explicitly.
    - Commonly used to initialize the attributes of the object.
  - **Destructor**: executed when destroying the object.
  - **Setter**: updates the value of the attributes.
  - **Getter**: returns the value of the attributes, without updating them.

Classes – UML

- Represented by a rectangle, divided in 3 regions:
  - Identifier of the class
  - Attributes (primitive only!)
  - Methods



Note: Only primitive type attributes are represented, reference attributes are represented as associations.

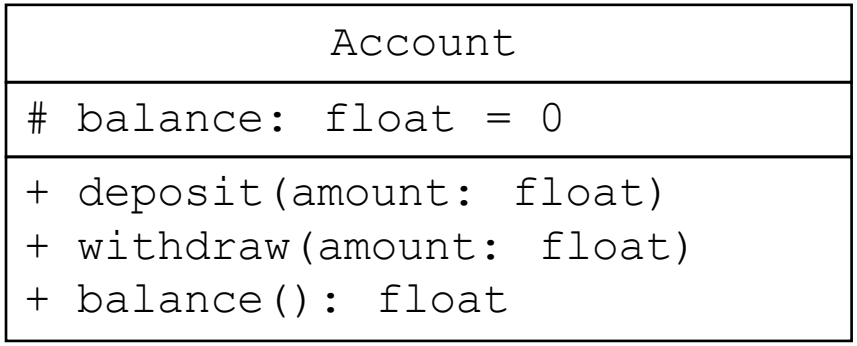
- The attribute and method regions are optional.
- A method and an attribute might have the same identifier.

Attributes – UML

Sintax

Visib Ident: Type [=Init][{Prop}]

- Visib: visibility
- Ident: attribute identifier
- Type: usual data types include primitive types (boolean, int, char, float,…)
- **Init**: initialization
- Prop: additional property

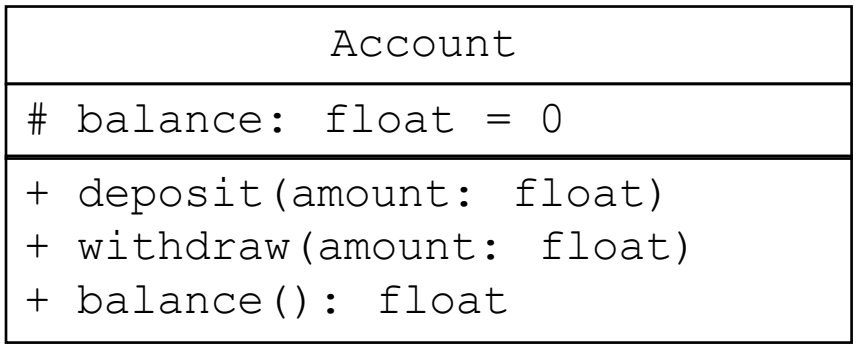


Methods – UML

Sintaxe

Visib Ident([id:ParamType [, id:ParamType]*])
[:RetType] [{Prop}]

- Visib: visibility
- Ident: method identifier
- id: parameter identifier
- ParamType: parameter type
- RetType: return type
- Prop: additional property



Visibility of attributes and methods – UML

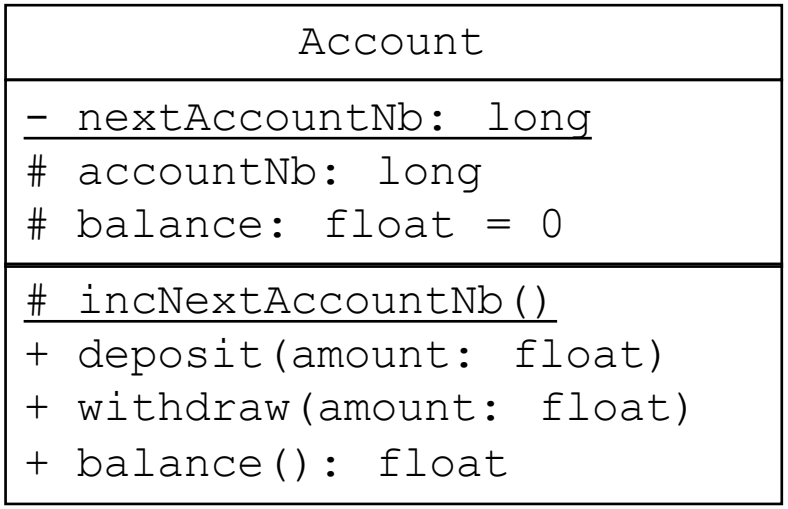
- Visibility of attributes and methods represented by a character before the identifier, and it determines the access permissions:
  - **public:** visible outside of the class (+)
  - **private**: visible only inside the class (–)
  - **protected**: visible in class and subclasses (#)
  - **package**: visible in all classes of the same package (~)

Class attributes and class methods – definition

- The attributes and methods can be:
  - instance: one for each instance of the class
  - class: one for all instances of the class

Class attributes and class methods – UML

- Represented with an underline in the class diagram.

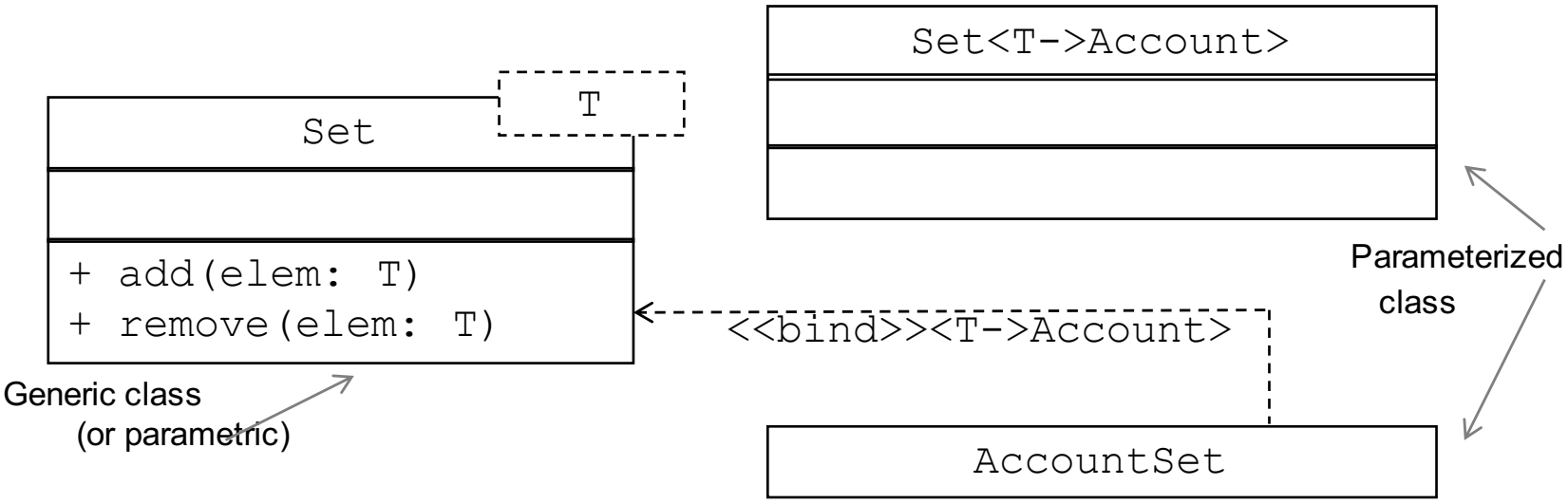


Generic classes – definition

- A **generic class** (or **parametric class**) is a class that receives as argument other classes. Instances of **generic classes** are denominated **parameterized classes**.
- Generic classes are commonly used to define collections (sets, lists, queues, trees, etc).

Generic classes – UML

- Represented in UML with a dashed box in the superior right corner of an UML representation of a class. The dashed box contains a list with the parameter types to pass to the generic class.



Relationships – definition

- Objects do not live isolated and they establish cooperative relationships within applications.
- A **relationship** is a connexion between elements. There are different kinds of relationships:
  - **Association**: relates objects between themselves.
  - **Composition/Aggregation**: relationship that denotes that the *whole* is constituted by *parts*.
  - **Inheritance**: mechanism of generalization-specialization of classes.
  - **Realization**: a class implements a functionality offered by an interface.

Association – UML (1)

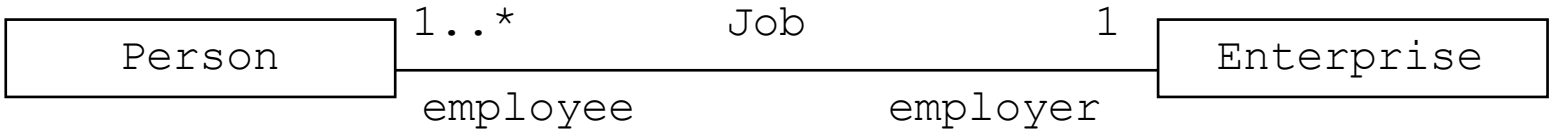
- An **association** represents a reference between objects.
- In an association is defined:
  - **Identifier** – term that describes the association.
  - **Role** – roles represented by the association in each of the related classes.
  - **Multiplicity** – number of objects associated in each of the related classes.

Association – UML (2)

- The identifier and the roles are optional.
- Associations might have diverse multiplicities:
  - exactly one (`1`).
  - zero or one (`0..1`).
  - zero or more (`0..*`).
  - one or more (`1..*`).
  - more complex multiplicities are also accepted, for instance, `0..1, 3..4, 6..*`, to say any number greater than 0 except 2 and 5.

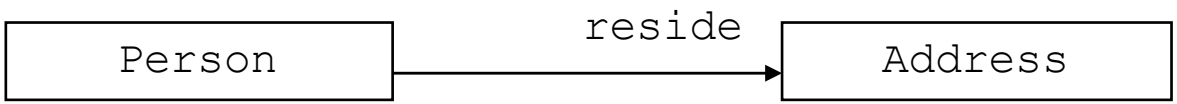
Association – UML (3)

- The association is represented by a solid line between the associated classes.
- The association identifier appears on top of the association line.
- The role of each class in the association appears in the respective endpoints of the association.
- The multiplicities also appear in the endpoints.



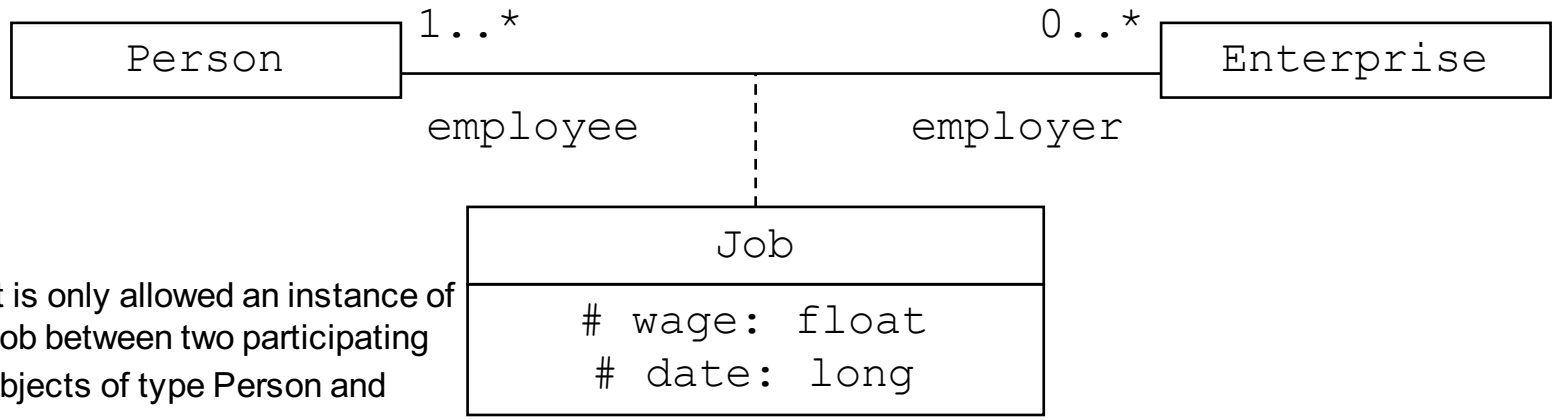
Association – UML (4)

- Associations might be **directed**, and in that case they are represented by an arrow.
- The direction of an association is related with implementation issues.



Association – UML (5)

- The associations might, by themselves, have extra information, being in that case an **association class** linked with a dashed line to the association.
- The association identifier is in that case given by the identifier of the associated class.



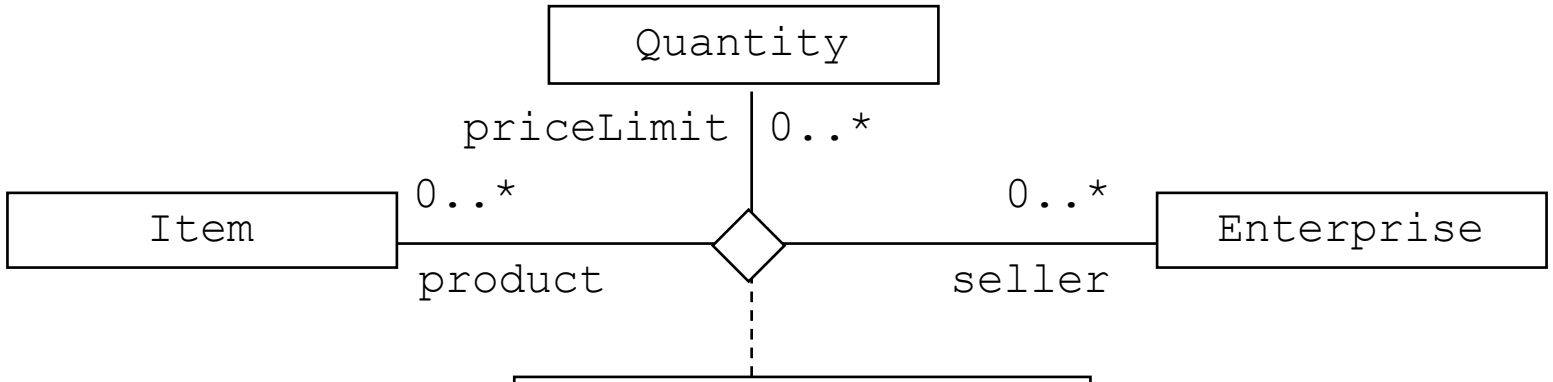
Note: it is only allowed an instance of Job between two participating objects of type Person and Enterprise!

Association – UML (6)

- By default, an association is:
  - bi-directional;
  - from one to one;
  - does not have extra information.

Association – UML (7)

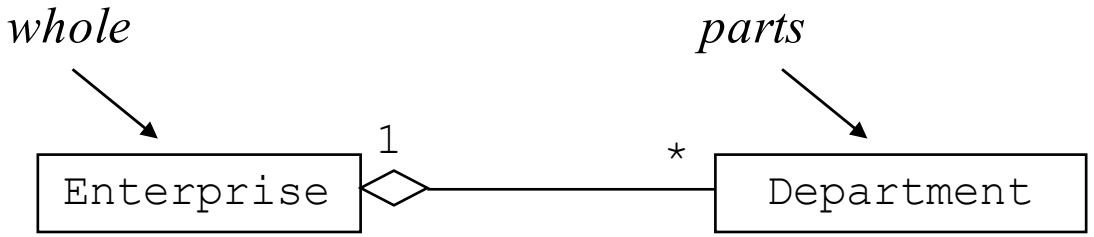
- **Ternary associations** represented by an unfilled diamond that links different solid lines of the associated classes.



Note: once again, given a fixed triple (item, quantity, enterprise) there could be only a unique price!

Aggregation/Composition – UML (1)

- An **aggregation** is an association which denotes that the *whole* is formed by *parts* – **does not imply ownership**.
- The aggregation is said to be as a relationship of "**has-a**".
- Represented as an association with a solid line, but with an unfilled diamond in the endpoint related to the *whole.*



Aggregation/Composition – UML (2)

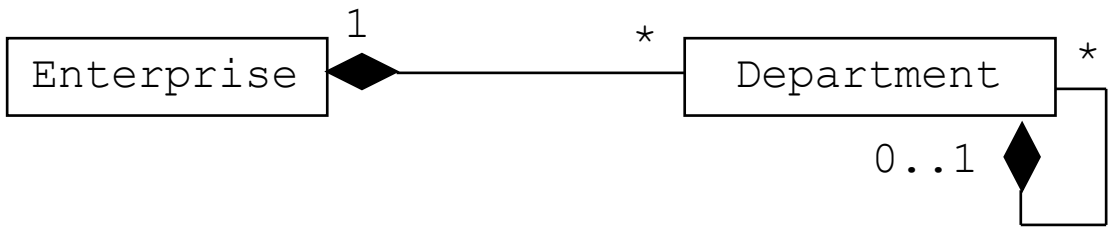
- In **composition**, when the owning object is destroyed, so are the contained objects – **no sharing**.
- Represented as an aggregation but the diamond is filled.

Aggregation/Composition – UML (3)

- Generally, both aggregation and composition do not have identifier, as the meaning of these relationships are implicit in the whole-part relation.
- The multiplicity should appear in both endpoints of the related classes. By default, multiplicity of exactly one (1) is considered.

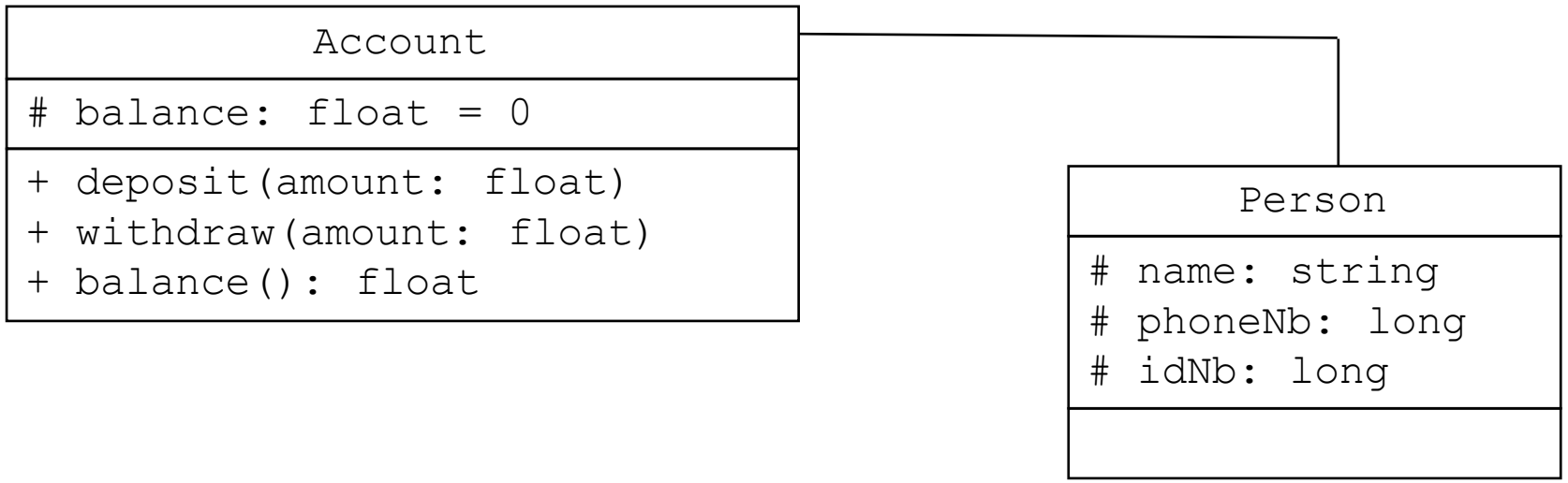
Reflexivity in relations – UML

- Relationships of association, aggregation and composition might be reflexive, with an element composed of several similar elements.



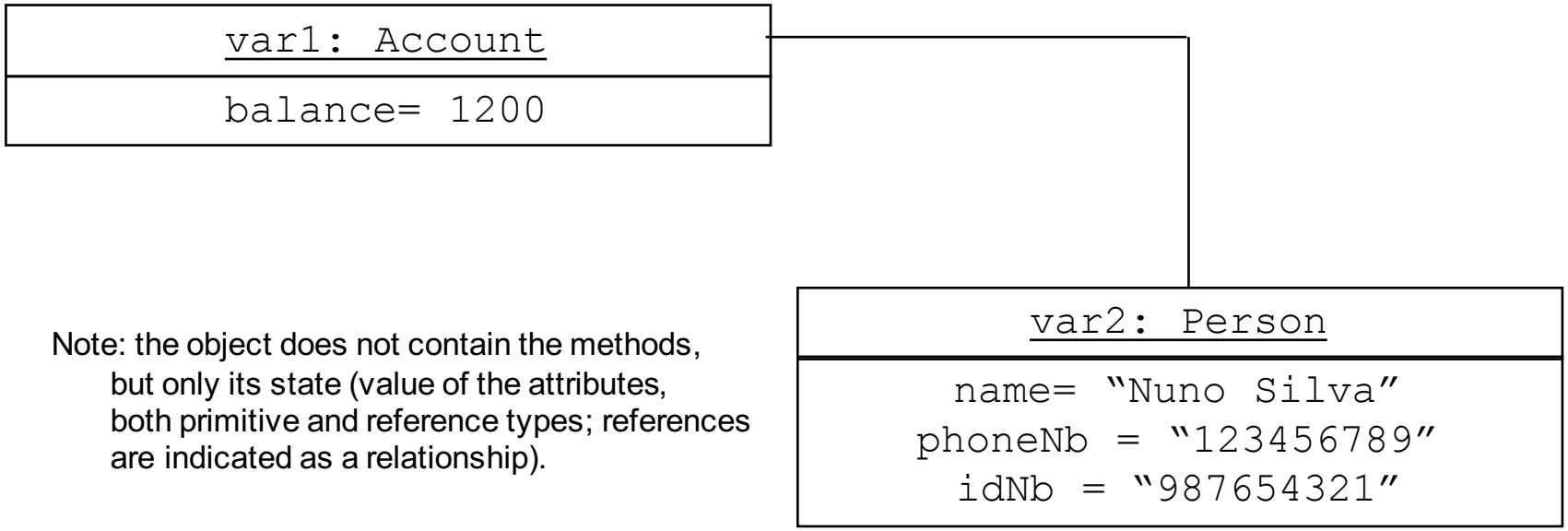
Relations – example (1)

"A bank account contains always a balance and belongs to a person. A person has a name, a phone number and an ID number. It is possible to deposit and withdraw money to and from the bank account, respectively. The owner of the account may consult its balance".



Objects – UML

- Represented by a rectangle, divided in 2 regions:
  - idObject:IdClass (or solely :IdClass)
  - Attributes initialization, one per line, as Id=Init



Note: the object does not contain the methods, but only its state (value of the attributes, both primitive and reference types; references are indicated as a relationship).

Inheritance – definition (1)

- **Open-closed principle**
  - **Software entities should be open for extension, but closed for modification.**
    - When designing a class, all attributes and methods should be offered and the class should be closed to future updates.
    - Classes should be open to extension, so that new requisites are incorporated with minimal impact in the system.

Inheritance – definition (2)

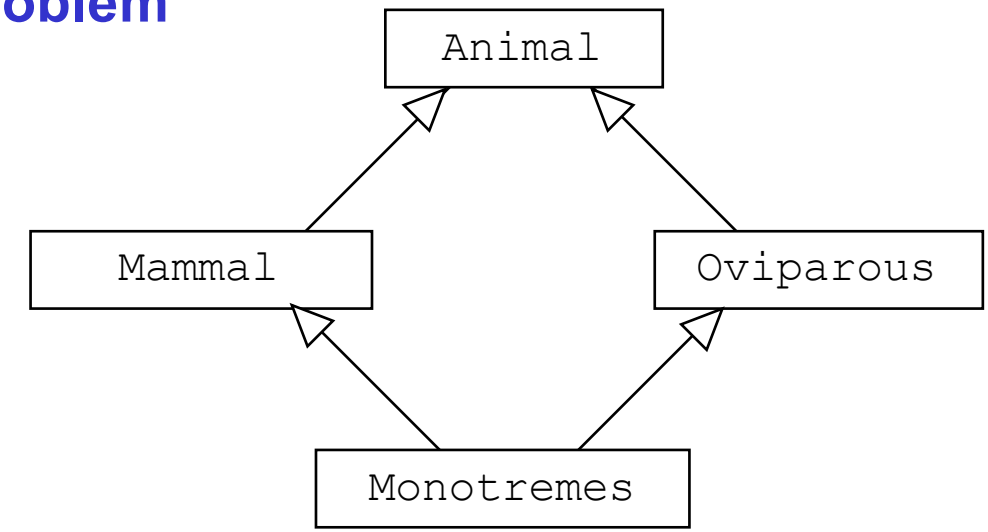
- Inheritance is a mechanism where the **subclass** constitute a specialization of a **superclass**. The superclass might be seen as a **generalization** of its subclasses.
- Inheritance is said as a relationship of **"is-a"**.
- Subclasses inherit the attributes and methods of superclasses. The inherited methods might be modified. New attributes and methods might be added to the subclasses.

Inheritance – definition (3)

- **Polymorphism** occurs when there is **redefinition** of methods (methods with the same signature) of the superclass in the subclass.
- In OO, polymorphism is usually implemented through **dynamic binding**, i.e., the method being executed is determined only in runtime (and not in compile time).

Inheritance – definition (4)

- In **simple inheritance** each subclass has only a direct superclass.
- In **multiple inheritance** a subclass might have more than one direct superclass.
  - **Diamond problem**



Inheritance – definition (5)

- Advantages:
  - More readability, as superclasses describe common aspects.
  - Easier to incorporate new requisites, as usually they coincide in particular aspects.
  - Promote reuse of the code of the superclasses.
- Disadvantages:
  - Need to detect for common aspects.

Inheritance – example (1)

"A bank account does not receive interest rates. A time deposit receives interest rates in the end of a time period, except if a withdraw transaction is made before the end of that period. In that case, the immobilization period is restarted".

Subclasses

- TimeDeposit (superclass: Account)

Inheritance – example (2)

Added attributes

- TimeDeposit: interest_rate (type float), begin (type long), period (type int)

Updated/redefined methods

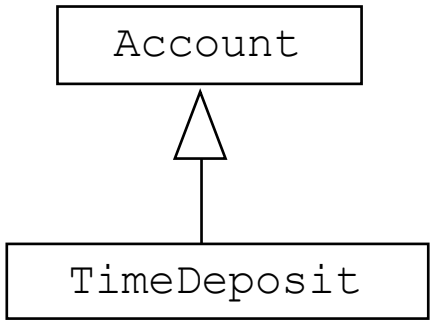
- TimeDeposit: withdraw

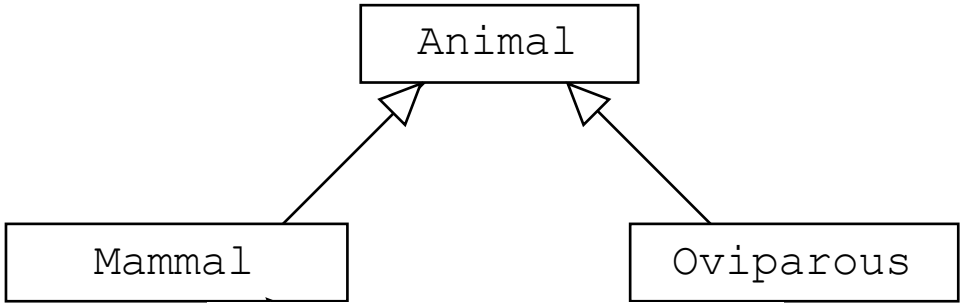
Added methods

- TimeDeposit: interest_rate

Inheritance – UML (1)

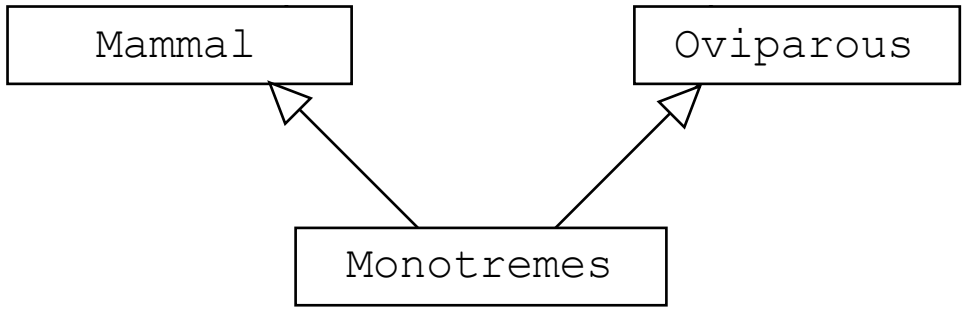
- Simple inheritance is represented with an unfilled arrowhead from subclasses towards superclass.





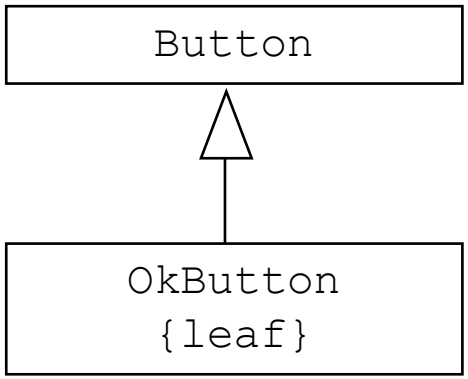
Inheritance – UML (2)

- Multiple inheritance is represented as an unfilled arrowhead from subclasses towards superclasses.



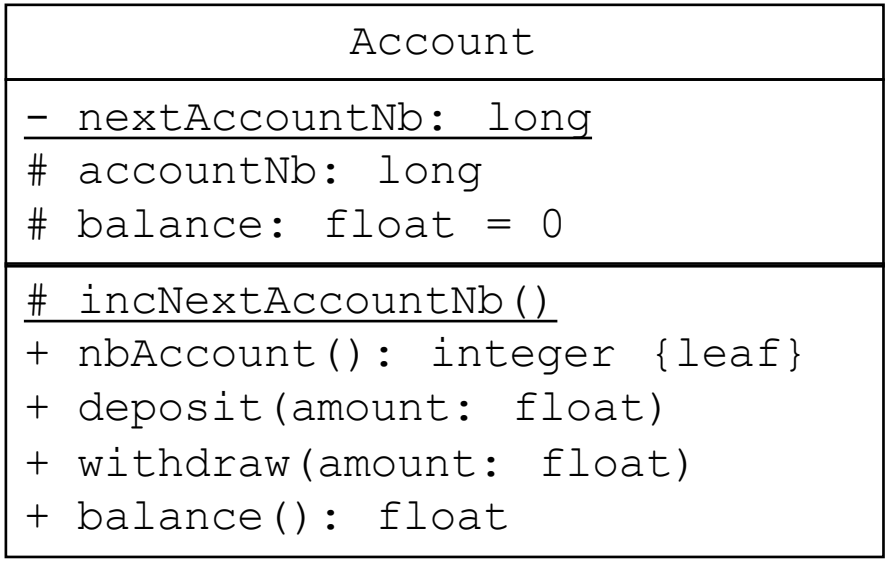
Inheritance – UML (3)

- **Not extensible classes**:
  - **Class without superclasses**: depicted with the property {root} written under the class identifier.
  - **Class without subclasses**: depicted with the property {leaf} written under the class identifier.



Inheritance – UML (4)

- UML also allows to specify that a certain method cannot be redefined in subclasses. Such methods are represented with the property {leaf} written after the signature of the method.



Inheritance – UML (5)

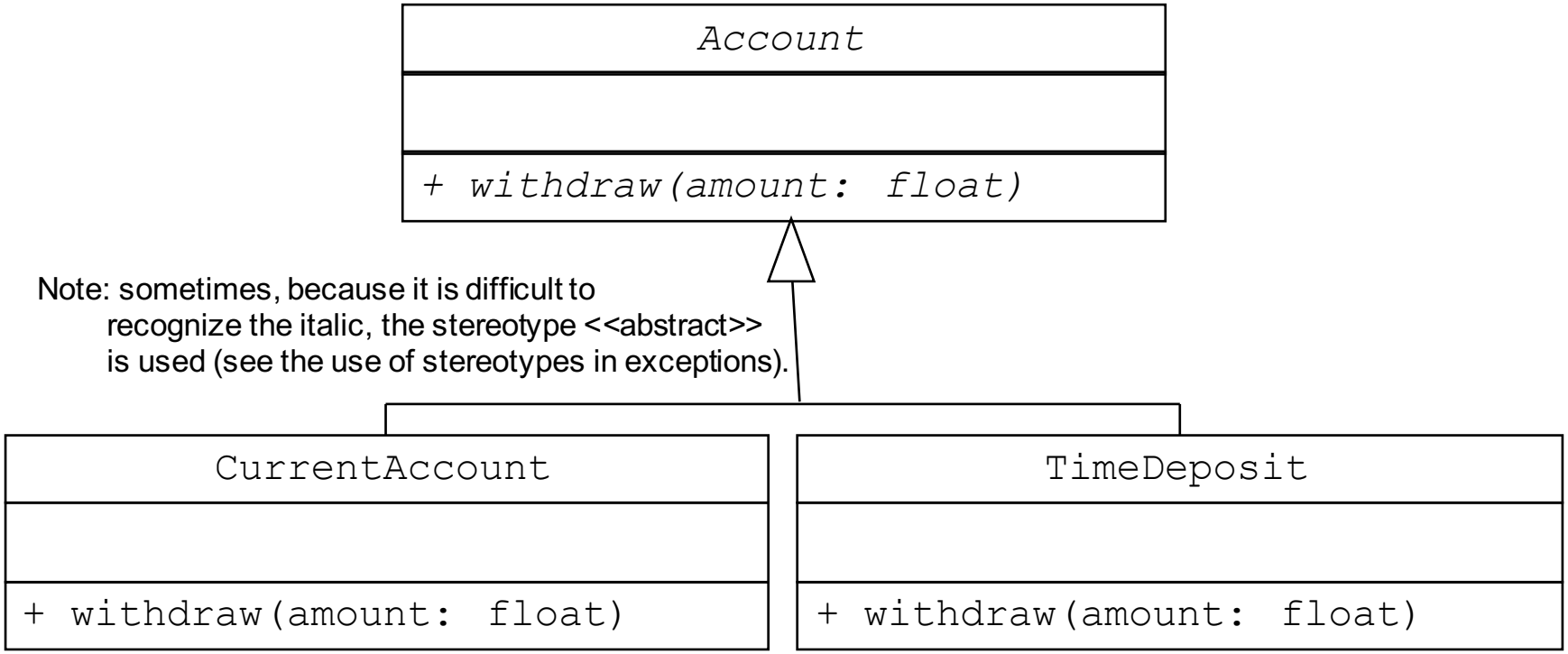
- Visibility of attributes and methods represented by a character before the identifier, and it determines the access permissions:
  - **public:** visible outside of the class (+)
  - **private**: visible only inside the class (–)
  - **protected**: visible in class and subclasses (#)
  - **package**: visible in all classes of the same package (~)

Abstract methods and classes – definition

- An **abstract method** is a method without implementation (it is only a prototype).
- An **abstract class** is a class that cannot be instantiated.
  - A class that has at least one abstract method (defined within the class or inherited from a superclass, direct or indirect), is an abstract class.

Abstract methods and classes – UML

- The abstract methods/classes are represented with their signature/identifier in italic.



Exceptions – definition (1)

- Frequently, applications are subject to many kind of errors:
  - Mathematic errors (for instance, divide by 0 arithmetic).
  - Invalid data format (for instance, integer with invalid characters).
  - Attempt to access a null reference.
  - Open an unexisting file.
  - …

Exceptions – definition (2)

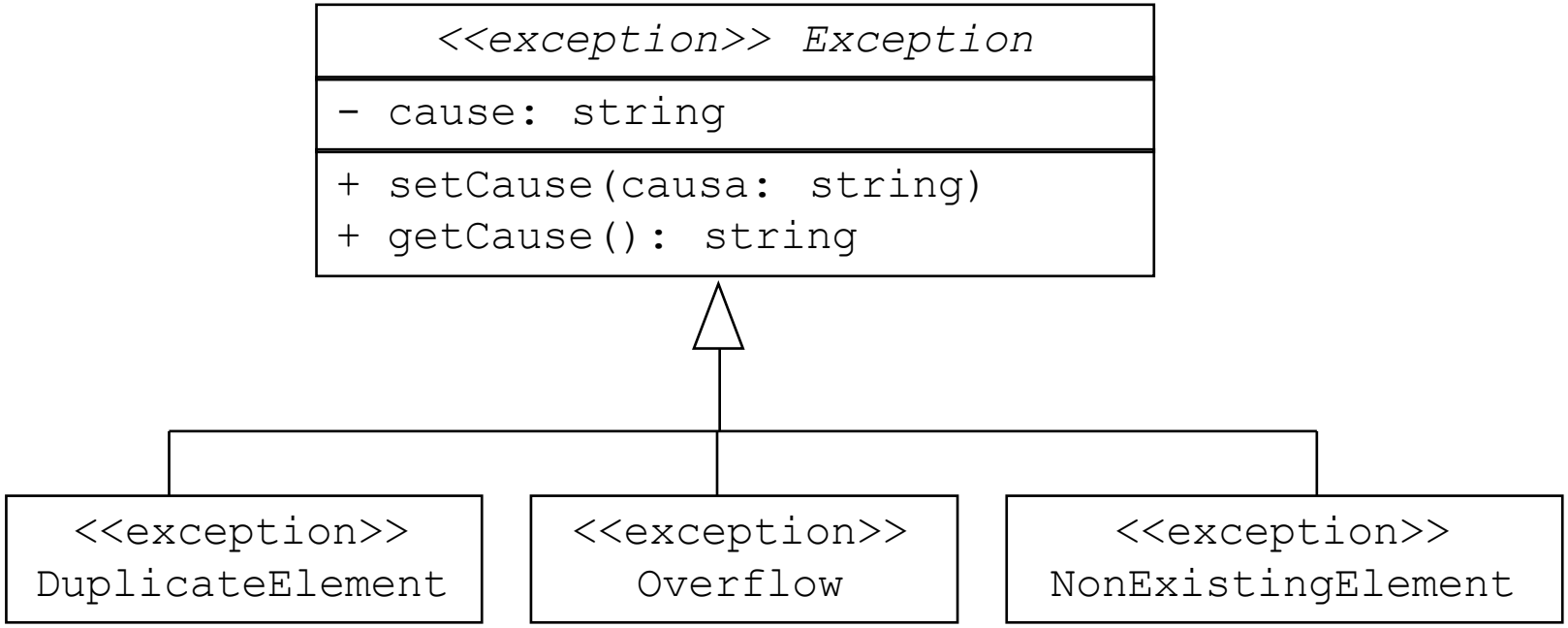
- An **exception** is a signal that is thrown when an unexpected error is encountered.
  - The signal is synchronous if it occurs directly as a consequence of a particular user instruction.
  - Otherwise, it is asynchronous.
- Exceptions can be handled in different ways:
  - Terminating abruptly execution, with warning messages and printing useful information (unacceptable in critical systems).
  - Being managed in specific places, denominated **handlers**.

Exceptions – definition (3)

- Advantages:
  - Provide a clean way to check for errors without cluttering code.
  - Provide a mechanism to signal errors directly, rather than indirectly with flags or side effects such as fields that must be checked.
  - Make the error conditions that a method can signal an explicit part of the method signature.

Exceptions – UML (1)

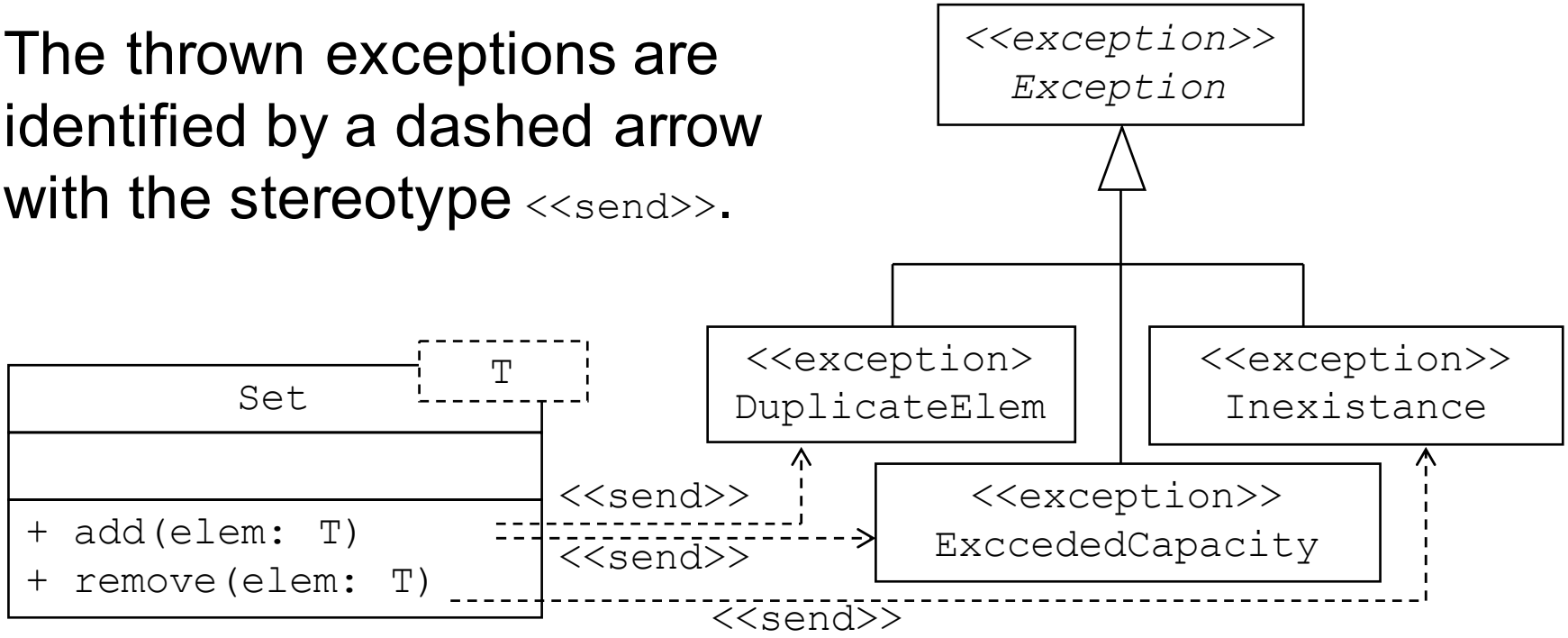
- Exceptions are represented as classes with the stereotype `<<exception>>`.



Exceptions – UML (2)

- The exception classes model the exceptions that an object can throw in the execution of their methods.

- The thrown exceptions are identified by a dashed arrow with the stereotype `<<send>>`.



Interfaces and Packages – definition

- Interfaces and packages are useful mechanisms to develop high dimensional systems:
  - The **interfaces** decouple the specification from the implementation.
  - The **packages** group distinct elements in a same unit.

Interfaces – definition (1)

- An **interface** is a collection of operations (without implementation) that are used to specify a service of a class:
  - Interfaces do not contain attributes, except for constants.
  - An interface may be realized (or implemented) by a **concrete class**. In this **implementation** or **realization**:
    - The attributes needed to the correct implementation of the interface are determined.
    - The code of the methods made specified by the interface is provided.

Interfaces – definition (2)

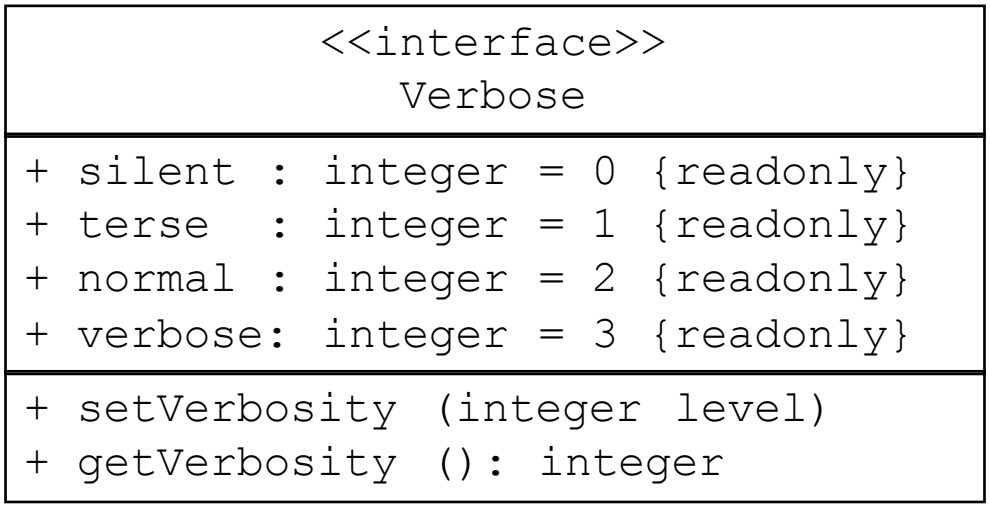
- An interface may inherit the definitions of another interface.
  - Interfaces may use polymorphism.
  - If a class realizes more than one interface, and different interfaces have methods with the same signature, the class should provide only one implementation of those methods.
- An interface cannot be instantiated.

Interface vs abstract class

- Similarities:
  - Both interfaces and abstract classes cannot be instantiated directly.
- Differences:
  - An abstract class may have attributes (constants or not), whereas an interface can only have constant attributes.
  - An abstract class may have methods with implementation.

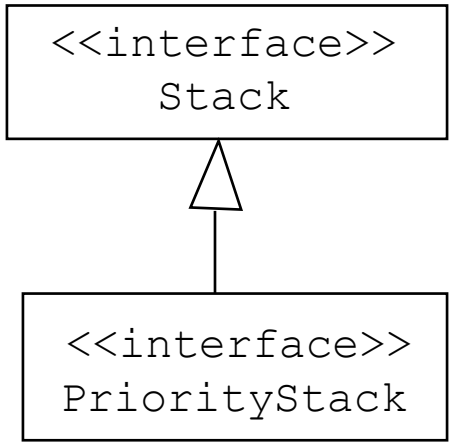
Interfaces – UML (1)

- Interfaces are represented as classes with the keyword `<<interface>>` above the name.
- An interface, besides the method prototypes, can only have **constant attributes**, represented with the property `{readonly}`.



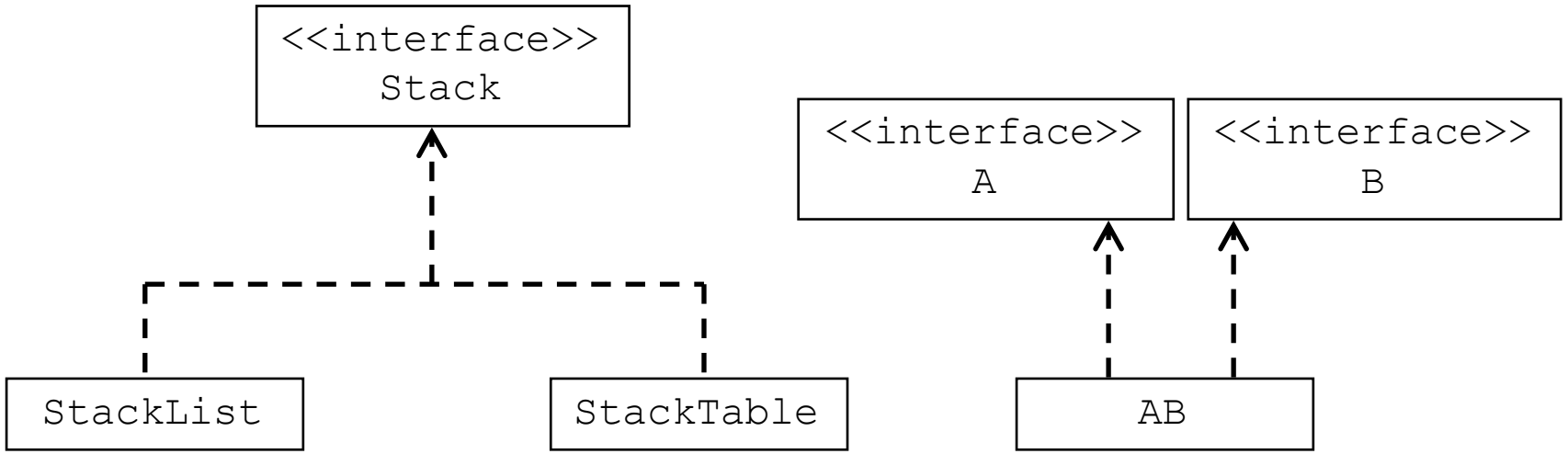
Interfaces – UML (2)

- Interfaces may participate in inheritance relationships (simple or multiple).



Interfaces – UML (3)

- An implementation class is linked to the interface by a relation of **realization**, graphically represented with a dashed arrow.
- One class may realize one or more interfaces.



Packages – definition (1)

- Packages are a mechanism to group information:
  - Packages may contain other packages, classes, interfaces and objects.
  - The package defines a **namespace**, so its members need to have unique identifiers (for instance, in a package two classes with the same name cannot exist).
  - The identifier of a package may consist in a **simple name** or in a **qualified name**. The qualified name corresponds to the simple name prefixed with the name of the package where it resides, if it exists. It is common to use :: to split the simple names.

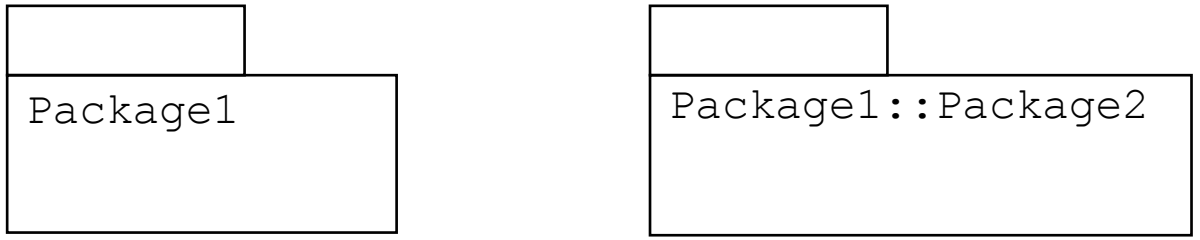
Packages – definition (2)

- An **import** adds the content of the imported package to the namespace of the importing package, so that the members of the imported package need not to be used by their qualified name.
- Importing is not transitive.
  - If package B imports package A, and package C imports package B, in package C the members of A are not imported.
  - If package C also wants to import the members of A, two imports are needed, one to import package A and other to import package B.

- The set of methods of a package is referred as **API** (*Application Programmer Interface*).

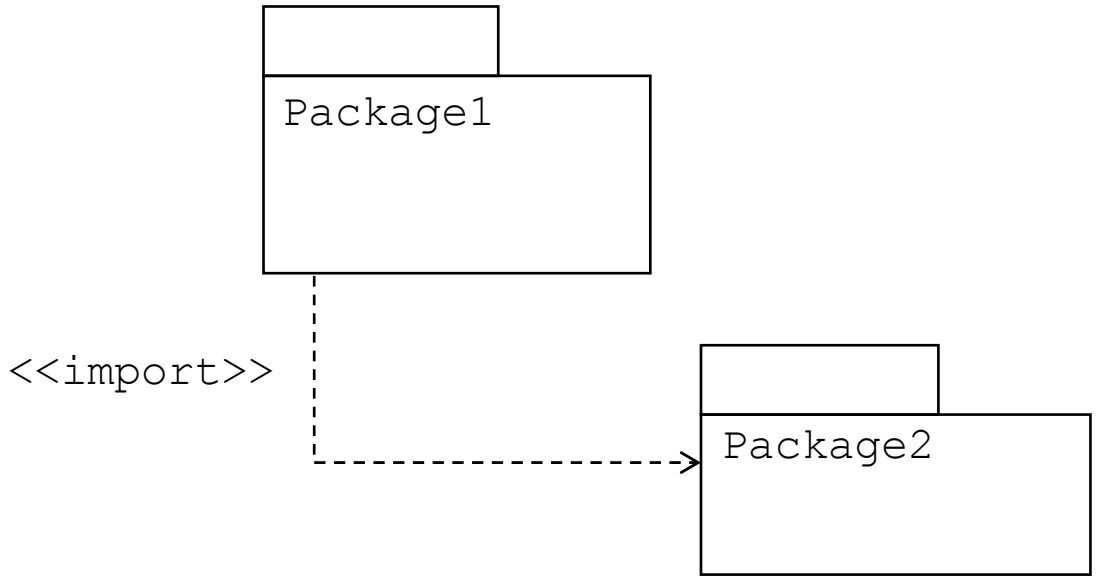
Packages – UML (1)

- Packages are represented as folders, identified by the respective name.



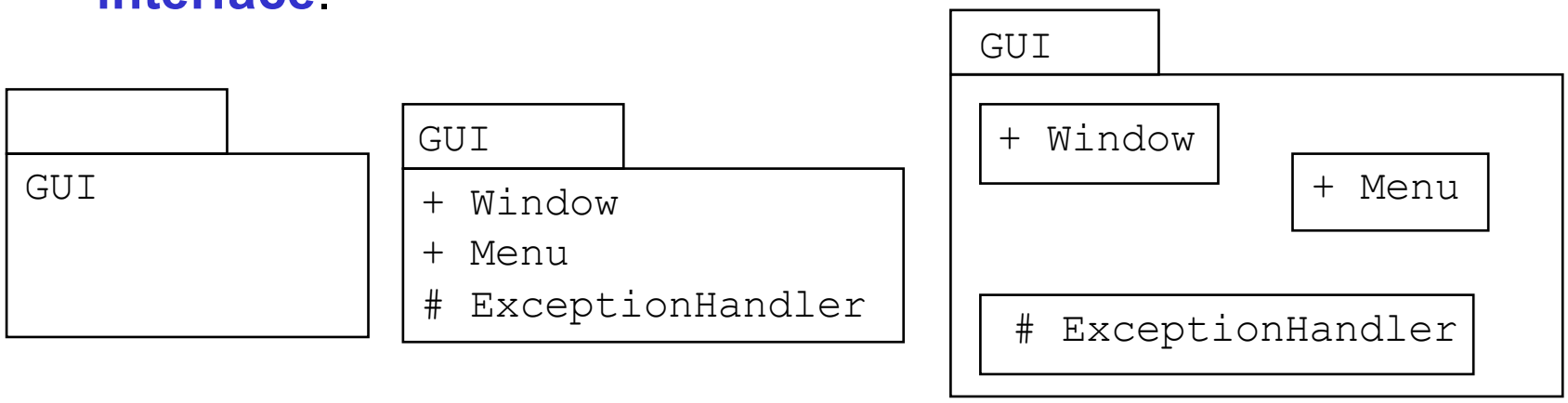
Packages – UML (2)

- An import is represented as a dashed arrow with the stereotype `<<import>>`.



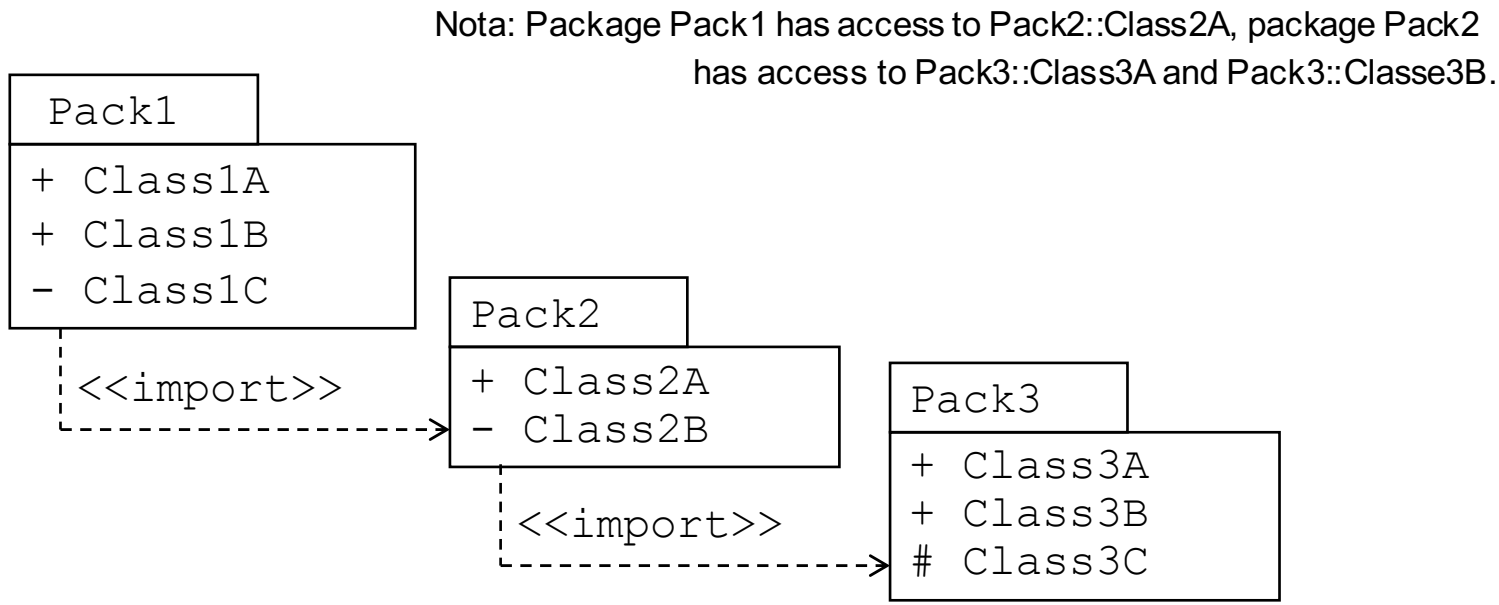
Packages – UML (3)

- The elements of a package may have diverse visibility:
  - **public**: elements visible to the package and to the importing packages (+)
  - **private**: elements not visible outside the package (-)
  - **protected**: elements visible in the package and subpackages (#)
  - **package**: elements visible only inside the package (~)
- The public members of a package constitute the **package interface**.



Packages – UML (4)

- A package **exports** only the public members.
- The exported members of a package are only visible by packages that explicitly import them.



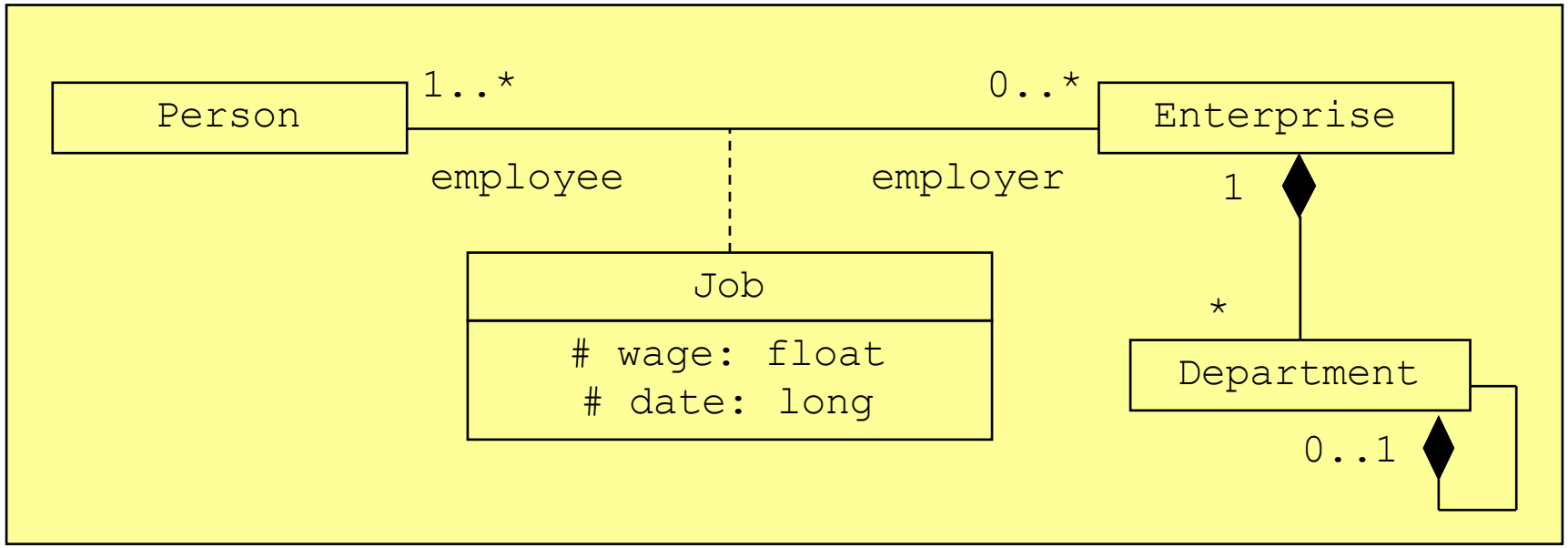
UML diagrams

- UML v2 makes avaliable13 diagrams, however, in OOP class only 1 is studied:
  - Structural diagrams:
    - **Class diagram**: classes, interfaces e relationships.
    - **Object diagram**: objects and relationships.
    - **Package diagram**: packages.

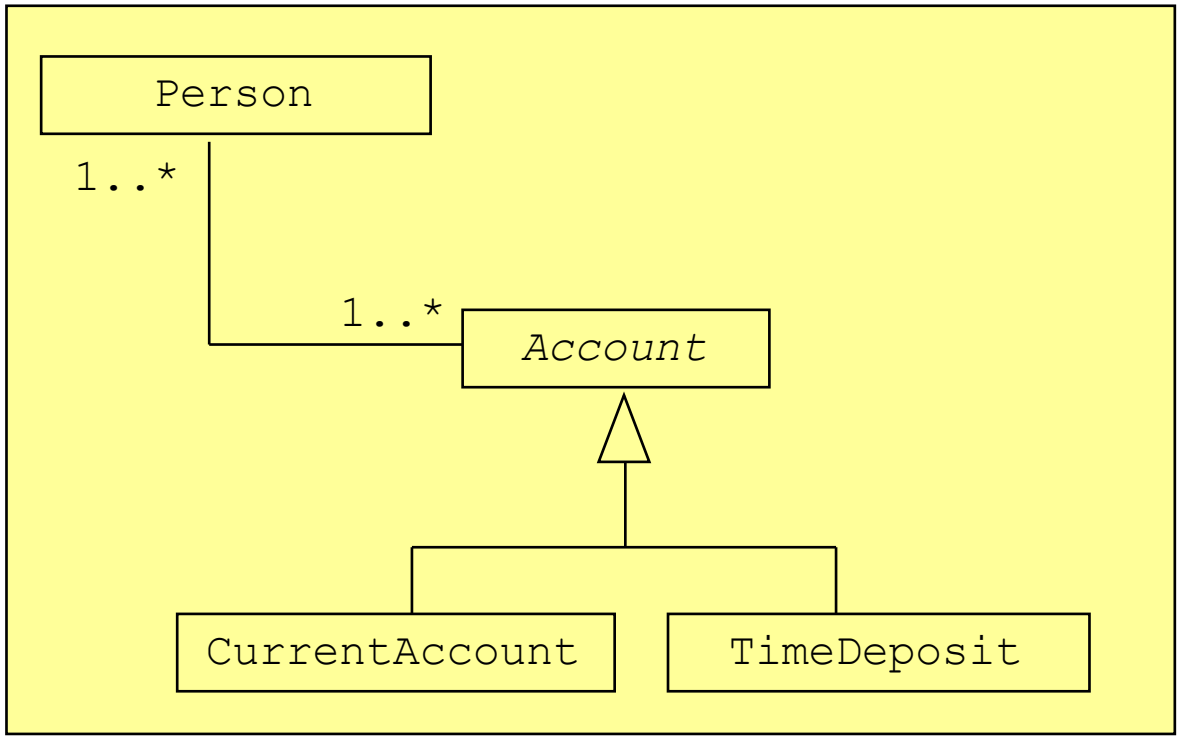
Class diagrams – definition

- **Class diagrams** are used to model:
  - Collaboration between classes.
  - Data base schemas:
    - Usually, information systems contains persistent objects.
    - The class diagram is a superset of **ER diagrams** (*entity-relationship*), a common model for logical database design. The ER diagrams focus only the data, whereas class diagrams allow, besides the data, to model behaviour.
- Typically, class diagrams contain classes, interfaces and relationships. It may also contain packages and objects.
- Class diagrams are the most common diagrams when modelling object oriented systems.

Class diagrams – UML (1)



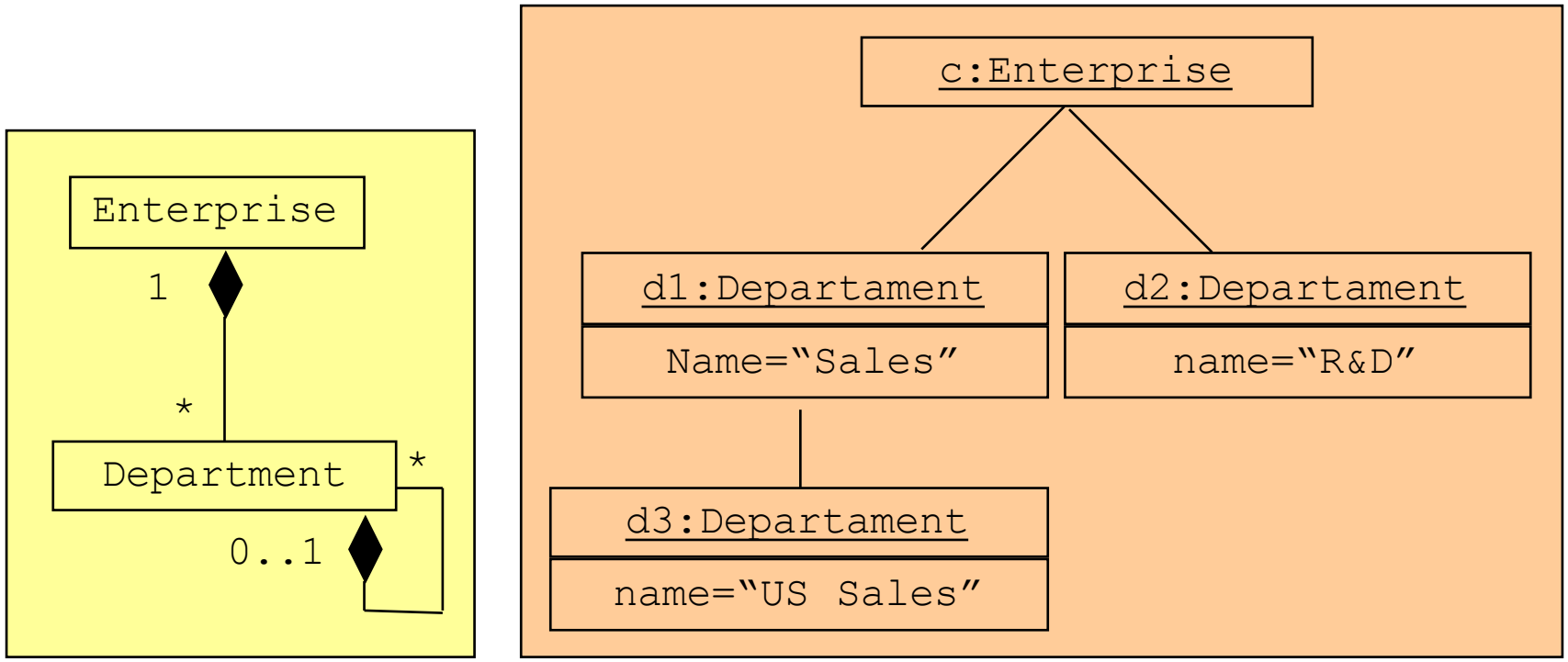
Class diagrams – UML (2)



Object diagrams – definition

- **Object diagrams** shows a set of objects and their relationships (a snapshot at a certain time instant).
- A class diagram captures a set of abstractions that are interesting, exposing their semantics and their relationships to other abstractions.

Object diagrams – UML



Package diagrams – UML

- A package diagram shows the decomposition of the model itself into organization units (packages) and their dependencies (imports).