Introduction to UML

Review: elements of OOP

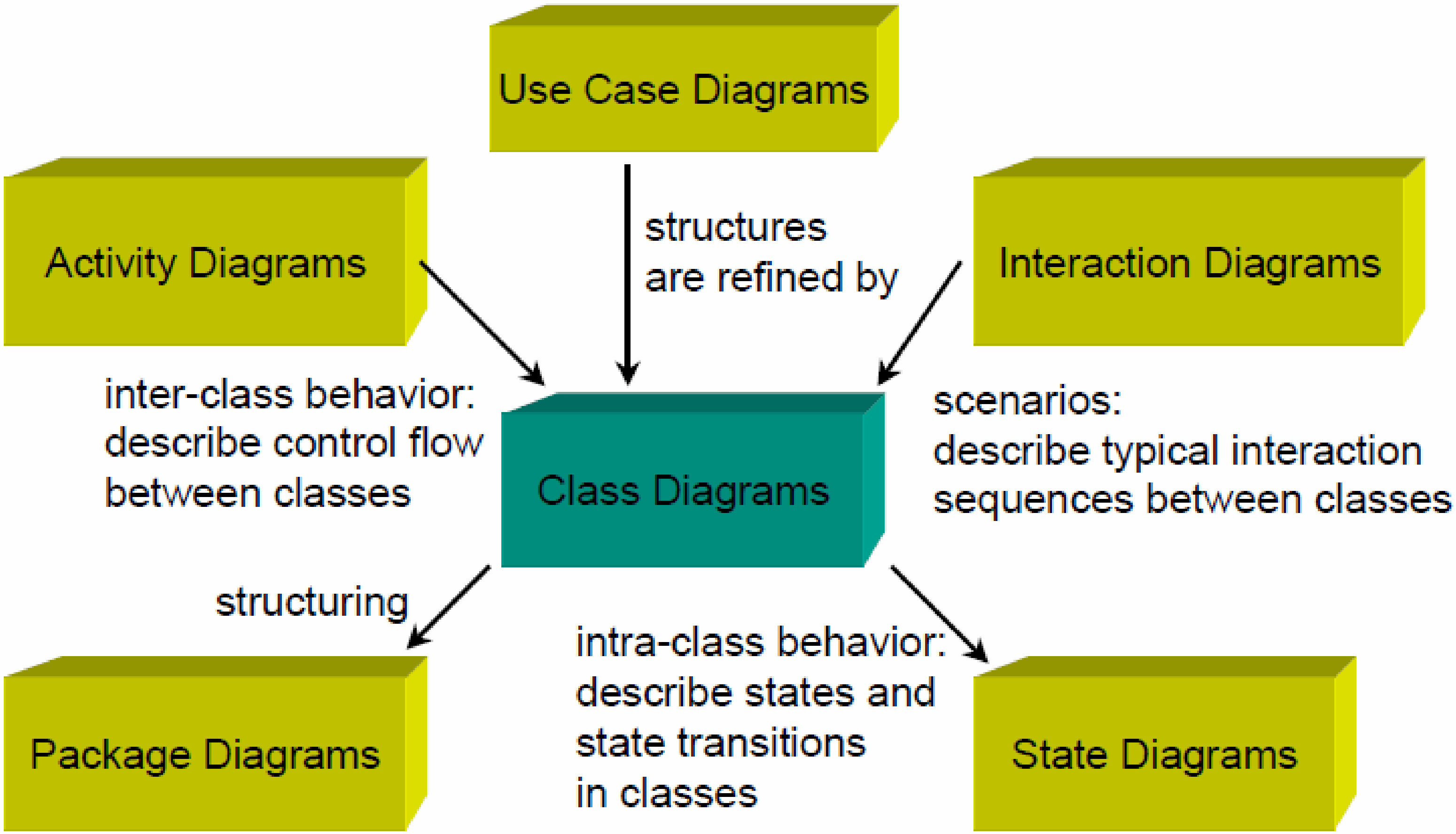
- Object
- Class
- Attribute
- Method/Operation
- Encapsulation
- Inheritance

Class Diagrams: Overview

Class diagrams describe the **types** of **objects** in the system and the various kinds of **static relationships** that exist among them.

- There are two principal kinds of static relationships:
  - associations
    - "a customer may rent a number of videos"
  - subtypes
    - "a student is a kind of person"
- Class diagrams also show the **attributes** and **operations** of a class and the **constraints** that apply to the way objects are connected.

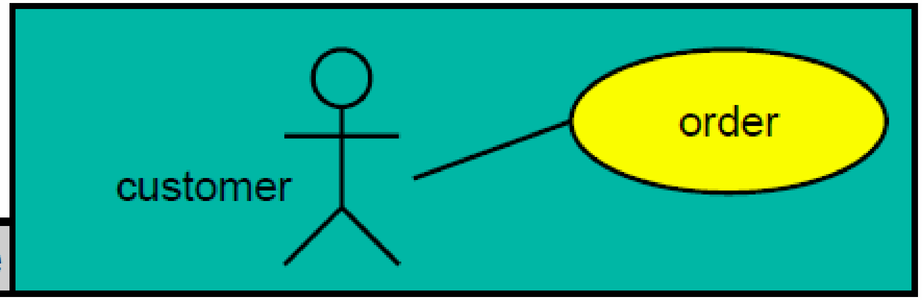
Role of Class Diagrams



Class diagrams are **central** for analysis, design and implementation.
Class diagrams are the **richest** notation in UML.

From Use Cases to Class Diagrams

The requirements list of a company includes the following textual description of the use case „order“:



Textfile

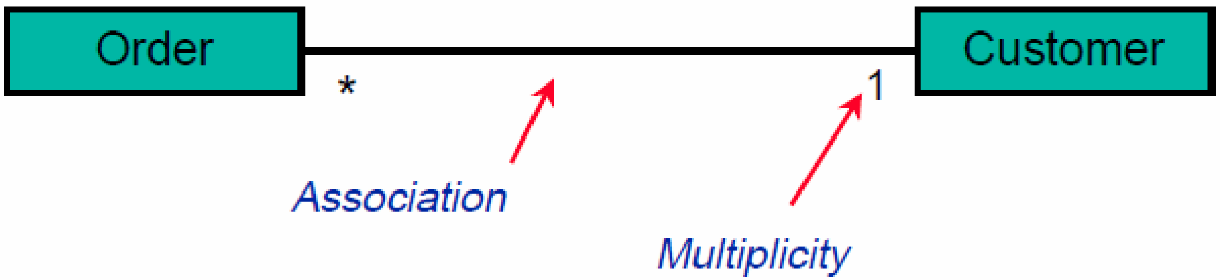
Textfile

<u>Order:</u>

We have customers who order our products.
We distinguish corporate customers from personal customers, since corporate customers are billed monthly whereas personal customers need to prepay their orders.

We want our orders to be lined up product by product.
Each line should contain the amount and the price of each product.

Example: Order - Associations

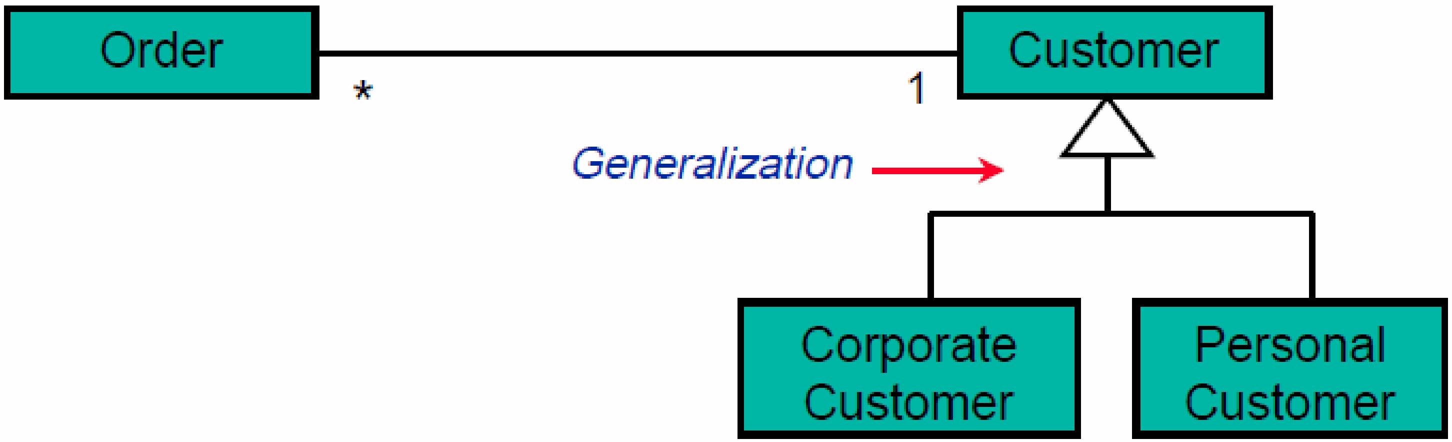


Order:

We have customers who may order several products.
We distinguish corporate customers from personal customers, since corporate customers are billed monthly whereas personal customers need to prepay their orders with a credit card.

We want our orders to be lined up product by product.
Each line should contain the amount and the price of each product.

Example: Order - Generalization

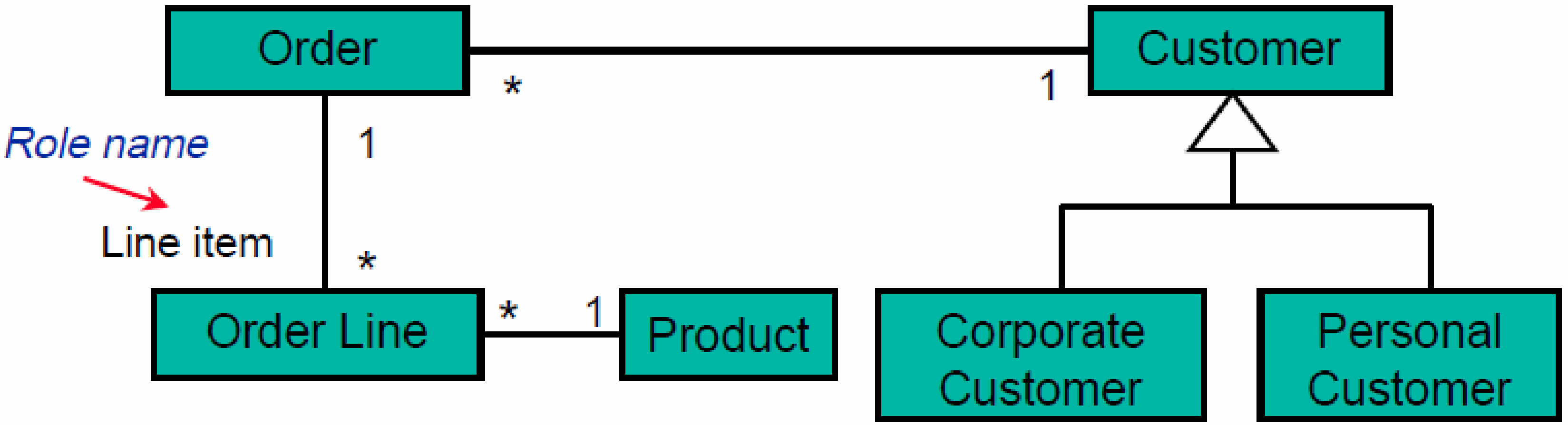


Order:

We have customers who order our products.
We distinguish corporate customers from personal customers, since corporate customers are billed monthly whereas personal customers need to prepay their orders with a credit card.

We want our orders to be lined up product by product.
Each line should contain the amount and the price of each product.

Example: Order - More Associations

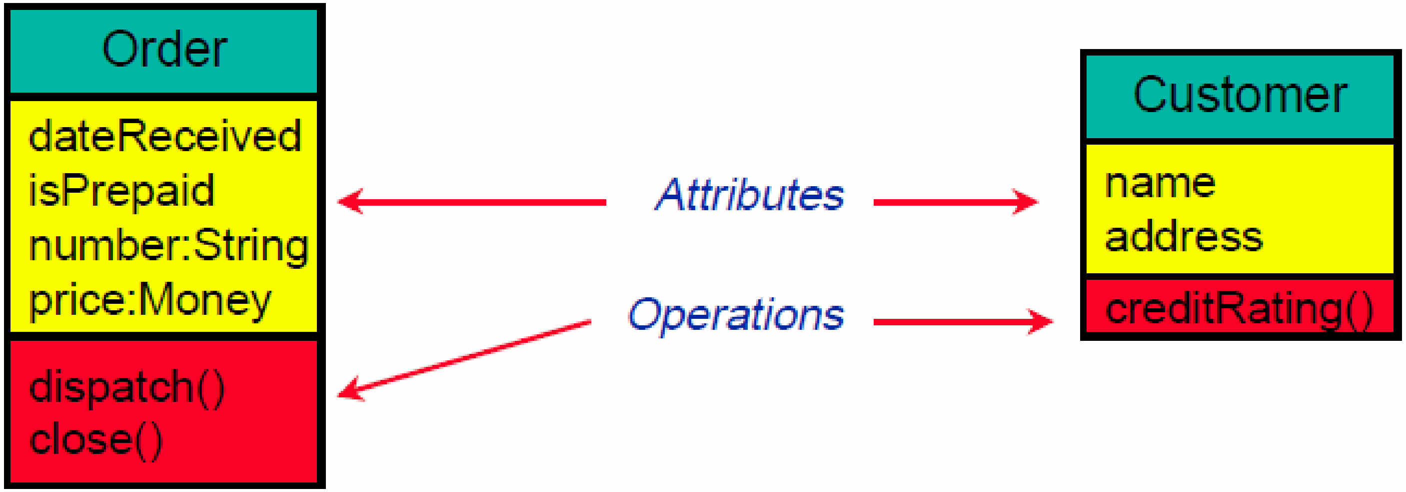


Order:

We have customers who order our products.
We distinguish corporate customers from personal customers, since corporate customers are billed monthly whereas personal customers need to prepay their orders with a credit card.

We want our orders to be lined up product by product.
Each line should contain the amount and the price of each product.

Example: Order- Attributes & Operations

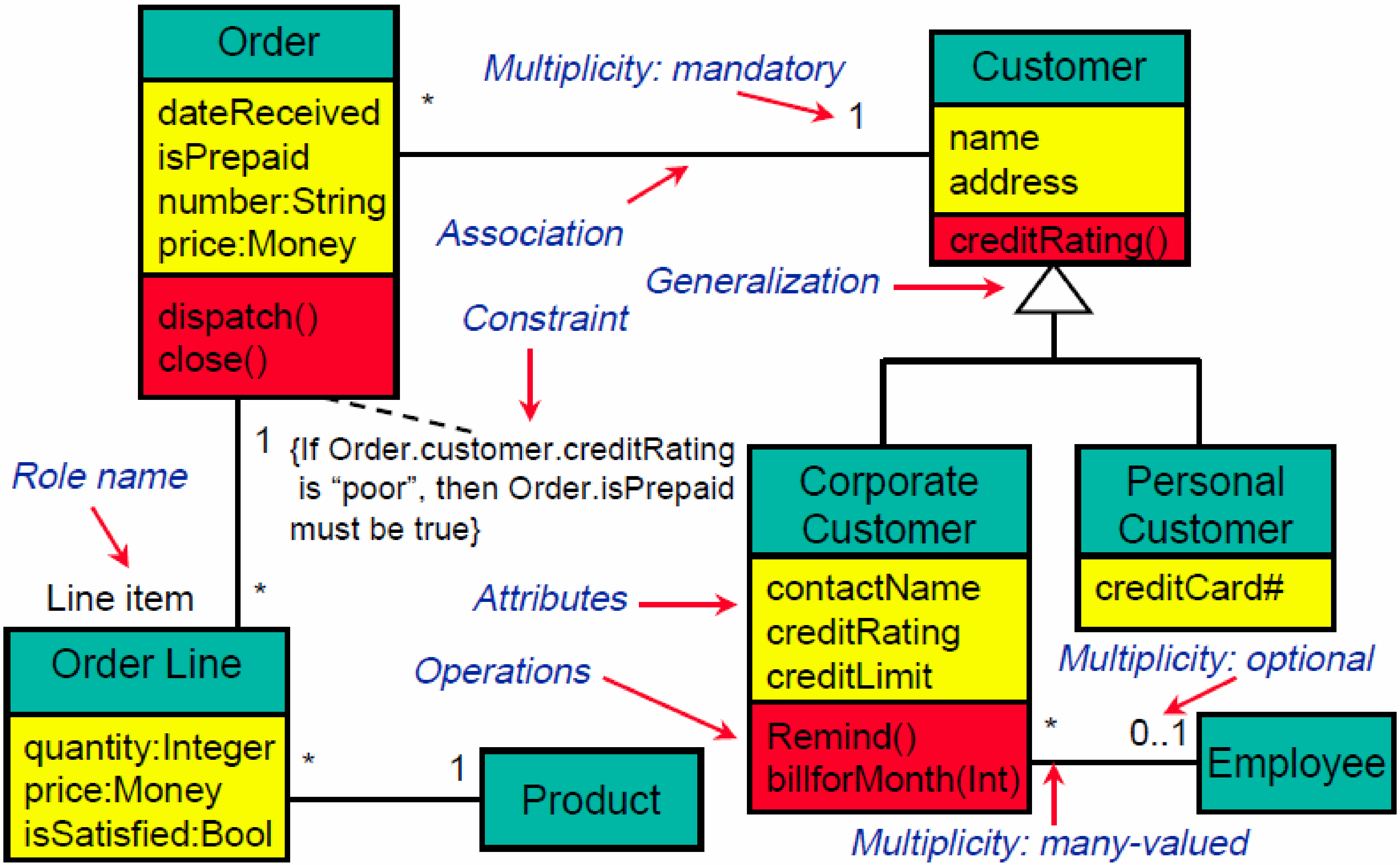


Order:

We have customers who order our products.
We distinguish corporate customers from personal customers, since corporate customers are billed monthly whereas personal customers need to prepay their orders with a credit card.

We want our orders to be lined up product by product.
Each line should contain the amount and the price of each product.

Example: Order - Full Class Diagram

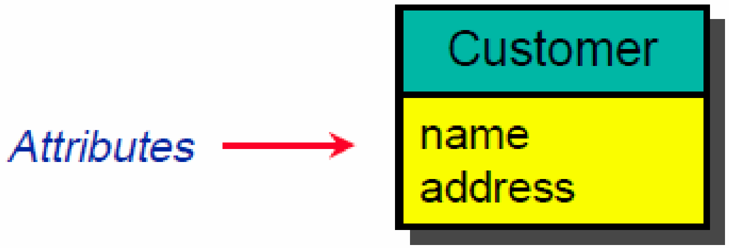


Perspectives

There are **three** perspectives (views) you can use in drawing class diagrams:

- **Conceptual**
  - represents the concepts relating to the classes
  - provides language independence
- **Specification**
  - represents the software **interfaces**
  - hides the implementation
- **Implementation**
  - shows the real classes used in the programming language
  - maps directly to the implementation

Attributes

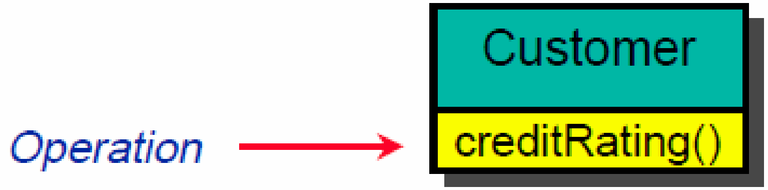


- Attributes may be specified at different levels of detail:
  - At the **conceptual** level a customer's name attribute indicates that customers have names.
  - At the **specification** level, this attribute indicates that a customer object can tell you its name and you can set the name.
  - At the **implementation** level, a customer has a field or an instance variable for its name.
- The UML syntax for attributes, depending on the level of detail:

visibility name: type = default-value

+ identifier : String = “Mr. Noname”

Operations



- Operations are the **processes** that a class knows to perform.
- They correspond to the **methods** of a class in an OO language.
- At **specification** level operations correspond to **public methods** on a class.
  - Normally you do not show those methods that simply set or get attribute values.
- In the **implementation** view usually private and protected methods are shown.
- The use of UML syntax for operations may vary with the level of detail:

visibility name(parameter-list) : return-type-expression {property string}

+ creditRating(for : Year) : Rating {abstract}

UML Meta Description

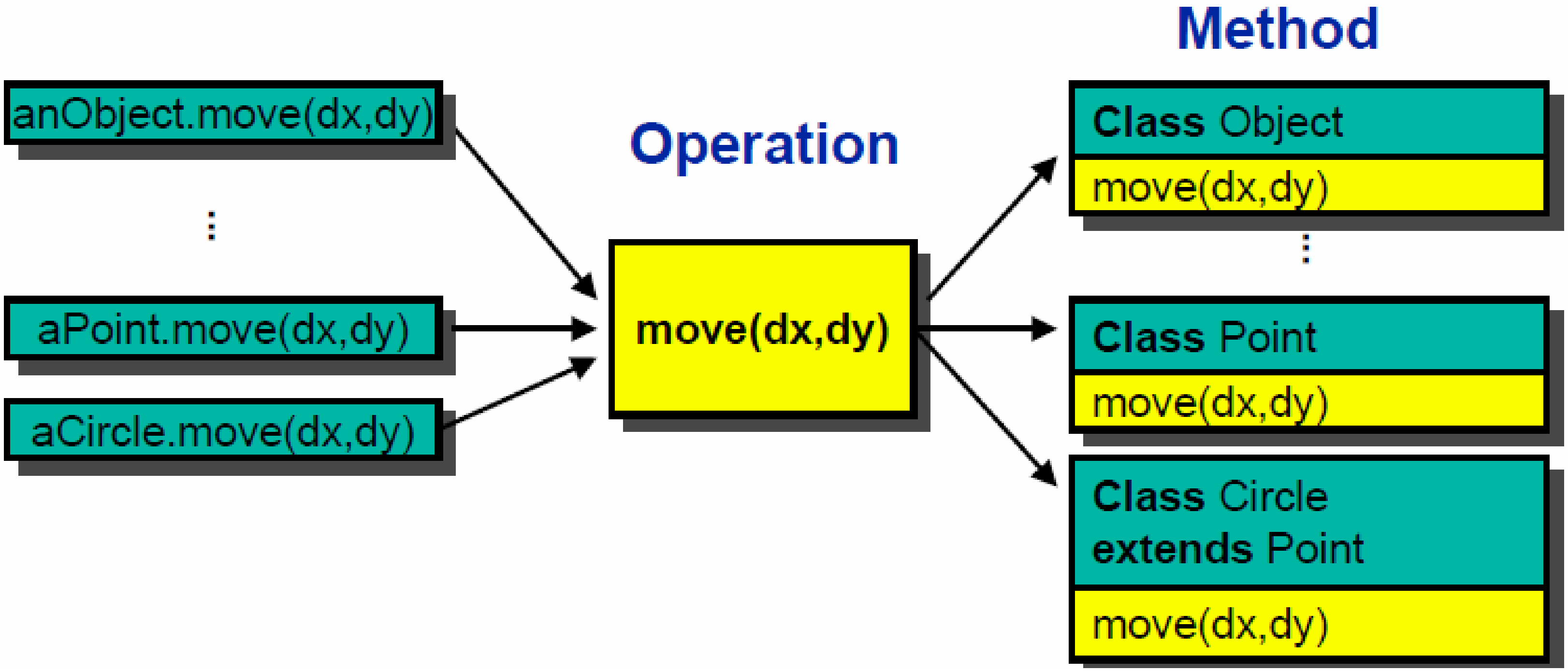
visibility name(parameter-list) : return-type-expression {property string}

- *Visibility* is + : for public, i.e., every other class can see this.
  – : for private, i.e., only this class can see this.
  # : for protected, i.e., only subclasses can see this.
- *Identifier* is defined by a string.
- *Parameter-list* contains (optional) arguments whose syntax is the same as that for attributes, i.e., name, type and default value.
- *Return-type-expression* is an optional, language-dependent specification that specifies the type of the return value (if any).
- *Property-string* indicates property values that apply to the given operation, e.g., if this operation is abstract (not implemented in this class, but in subclasses).

+ creditRating(for : Year) : Rating {abstract}

Operations vs. Methods

- An **operation** is something that is **invoked** on an object (or a **message** that is **sent** to an object) while
- a **method** is the **body** of a procedure, i.e., the implementation that realizes the operation or method.
- This distinction facilitates **polymorphism**.

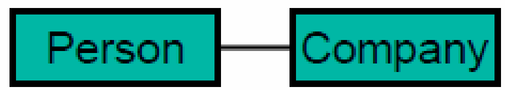


Associations

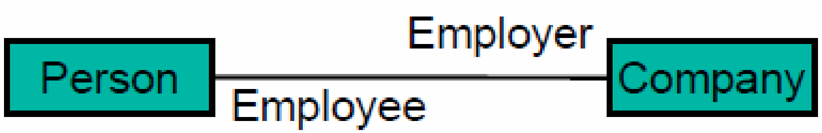
- *Associations* represent relationships between **<u>instances</u>** of classes.
  - "Peter and Mary *are employed* **by** IBM."

Instances are marked by <u>underlining</u>.

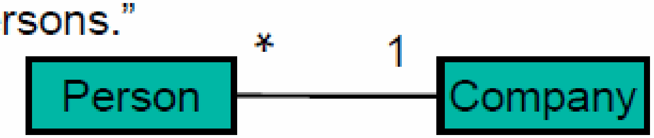
- From the conceptual perspective, associations represent conceptual relationships between **classes**.
  - "Companies *employ* persons."



- Each association has two **roles** that may be named with a label.



- **Multiplicities** indicate how many objects may participate in a relationship.
  - "A person is employed by a (exactly one) company."
  - "A company may employ many persons."

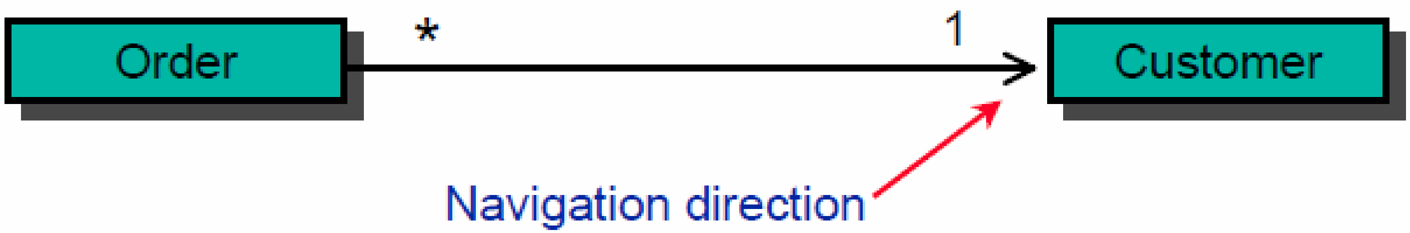


Associations: Multiplicities

- The * represents the range 0..*Infinity*.
- The 1 stands for 1..1.
  - "An order must have been placed by exactly one customer."
- For more general multiplicities you can have
  - a **single** number like 11 soccer players,
  - a **range**, for example, 2..4 players for a canasta game,
  - a discrete **set** of numbers like 2,4 for doors in a car.

Navigability

- To indicate navigability with associations, arrows are added to the lines.



- In a **specification** view this would indicate that an order has a **responsibility** to tell which customer it is for, but a customer has no corresponding ability to tell you which orders it has.
- In an **implementation** view, one would indicate, that order contains a **pointer** to customer but customer would not point to orders.
- If a navigability exists in only one direction it is called

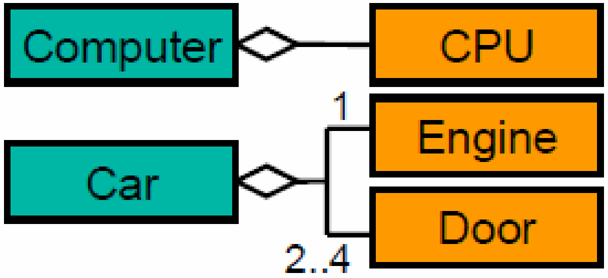
  uni-directional association

  otherwise

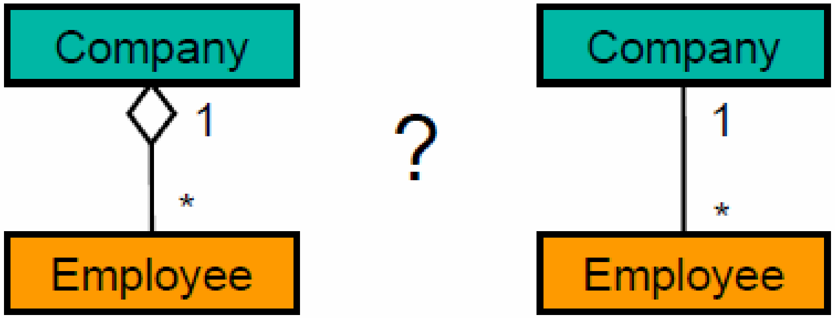
  bi-directional association.

Aggregation

- Aggregation is the **part-of** relationship.
  - "A CPU is part of a computer."
  - "A car has an engine and doors as its parts."

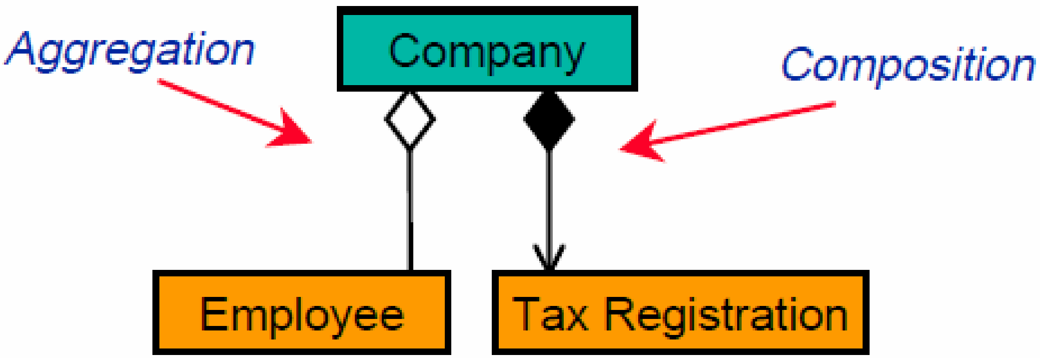


- Aggregation vs. attributes :
  - **Attributes** describe **properties** of objects, e.g. speed, price, length.
  - **Aggregation** describe **assemblies** of objects.
- Aggregation vs. **association**:
  - Is a company an aggregation over its employees or is it an association between its employees?



Composition

- Composition is a **stronger** version of **aggregation**:
  - The part object may **belong to only one** whole.
  - The parts usually live and **die** with the whole.
- Example:
  - **Aggregation**: A company has employees. The employees may change the company.
  - **Composition**: The company has a tax registration. The tax registration is tied to the company and dies with it.

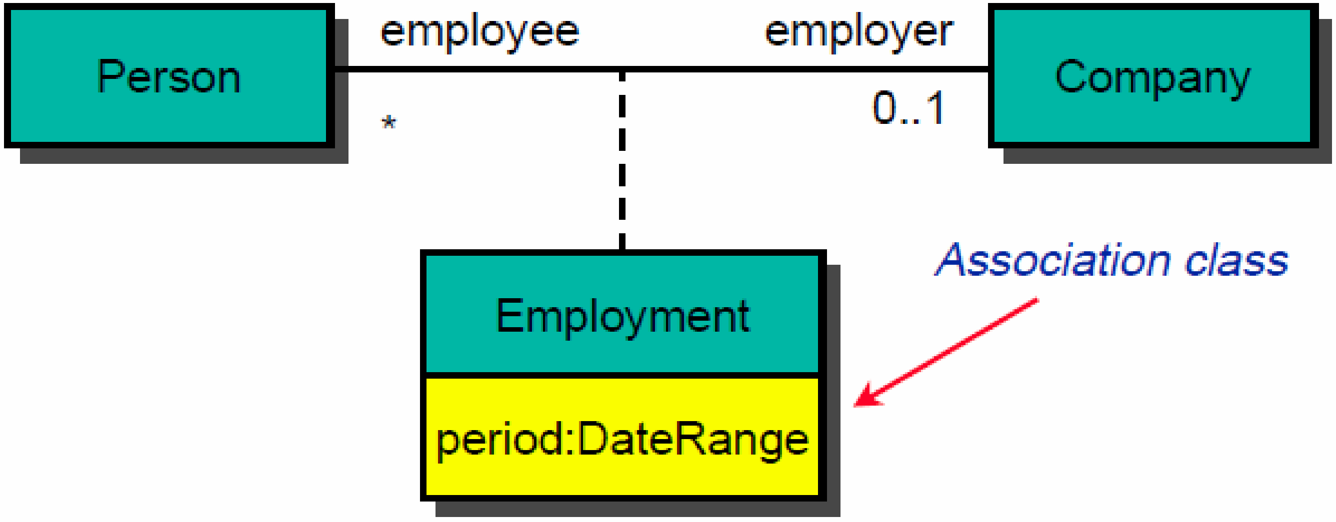


Association Classes

Example: Persons are employed by companies for a period of time.

Question: Where does the period attribute go?

Association classes allow you to model associations by classes, i.e., to **add attributes**, operations and other features to associations.



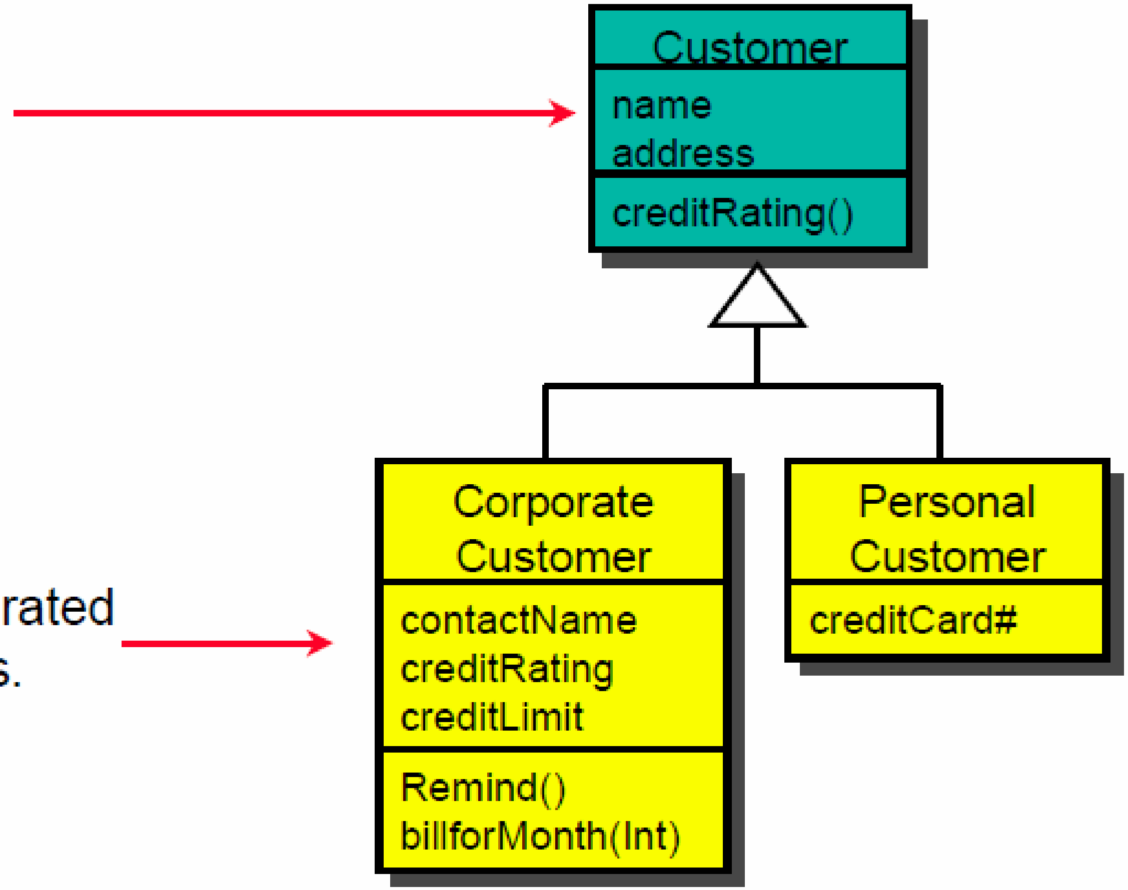
Note: a person and a company are associated only by **one** employment period here.

Generalization

Generalization captures **similarities** between several classes in a superclass
Specialization refines and adds **differences** in subclasses.

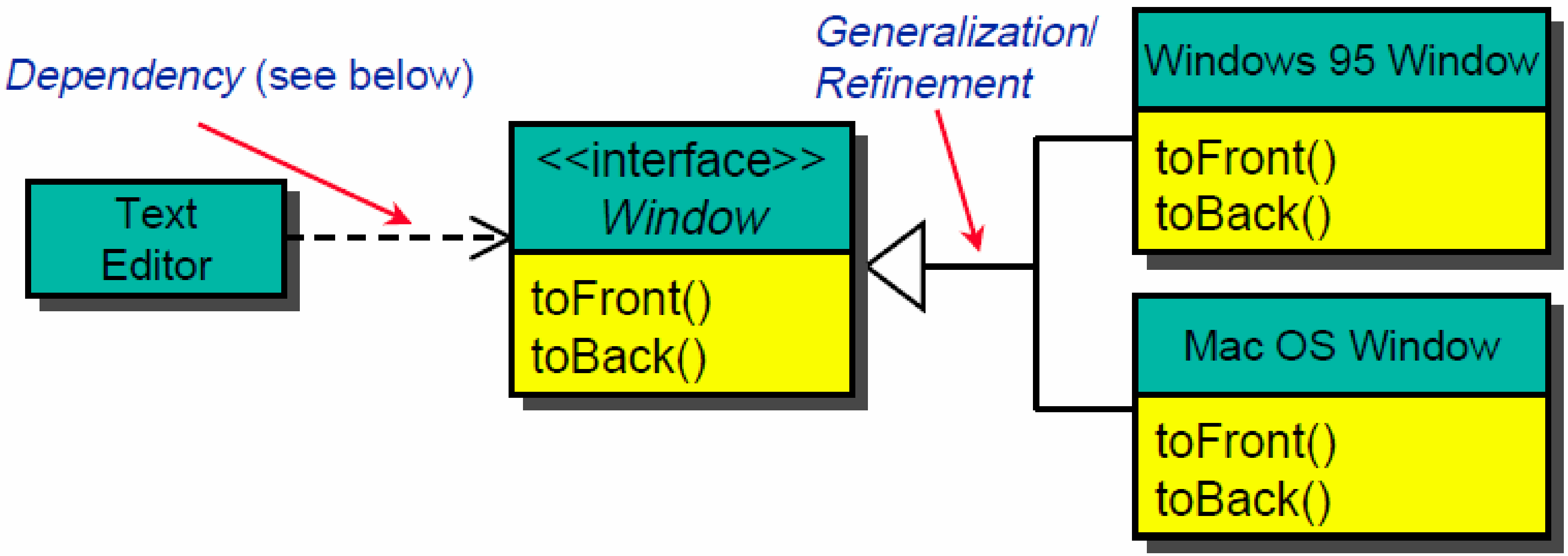
Similarities are placed in a general superclass.

The differences are separated in specialized subclasses.



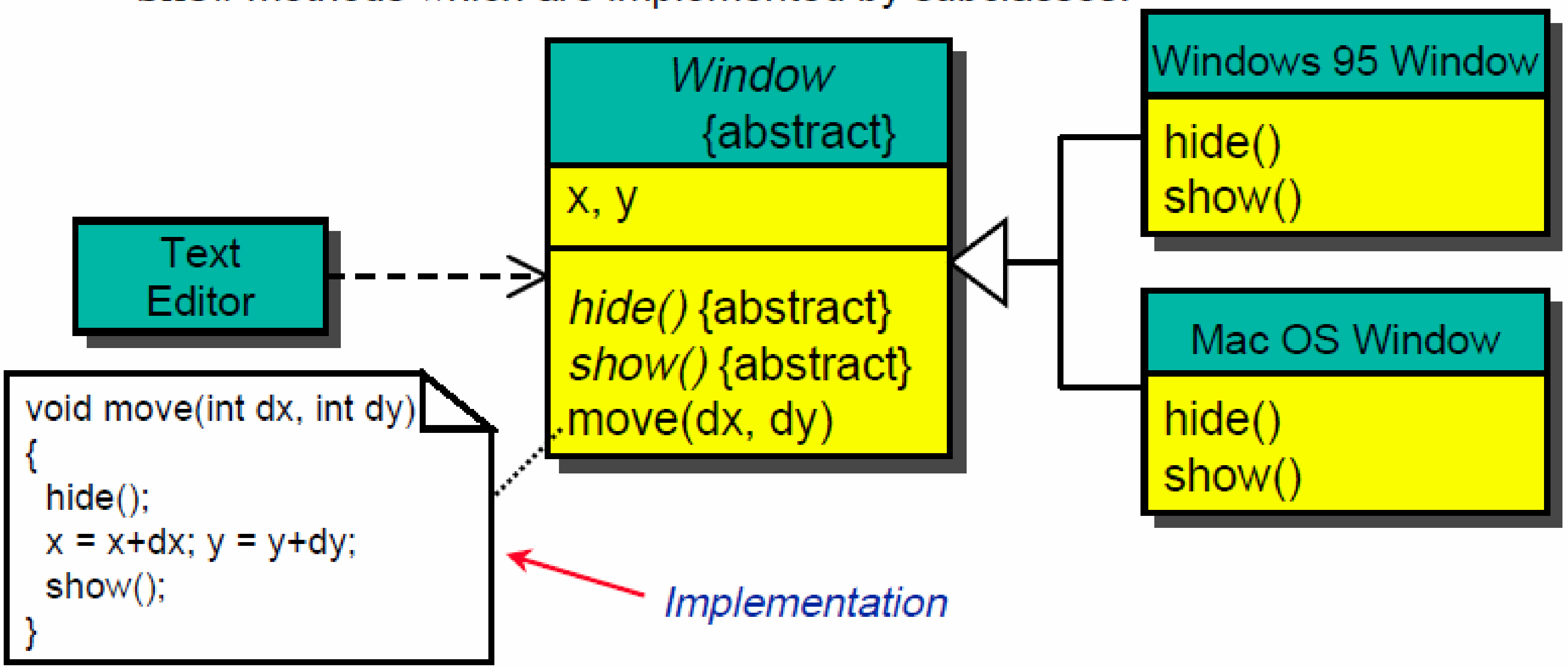
Interfaces

- An **interface** is a (abstract) class with **no implementation.**
  - An interface is implemented (refined) by (different) classes.
  - The implementation can be changed without changing the clients.
- Example: A portable text editor displays its windows using a window interface that is implemented differently for Windows 95 and Mac OS.



Abstract Classes

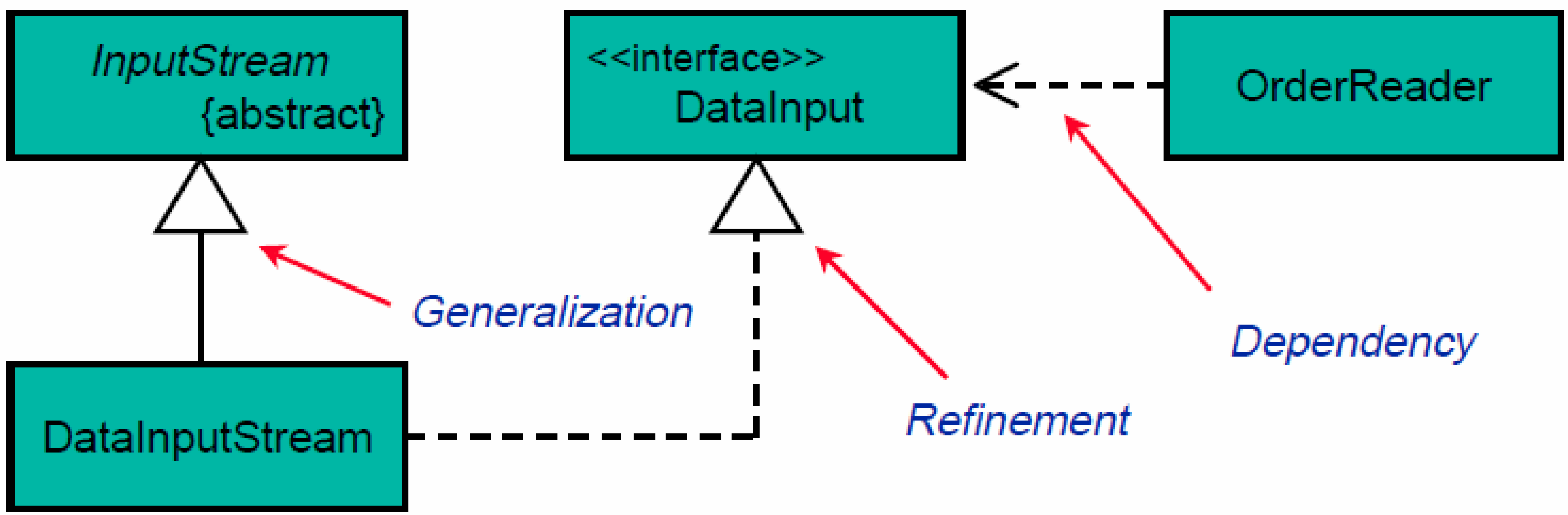
- An **abstract** class is a class without a **(full) implementation**.
  - Some **methods** are **deferred**, i.e., they are not implemented.
  - The deferred methods are implemented by **subclasses** only.
- Example: The window `move` operation is implemented by using `hide` and `show` methods which are implemented by subclasses.



Example: Interfaces and Abstract Classes

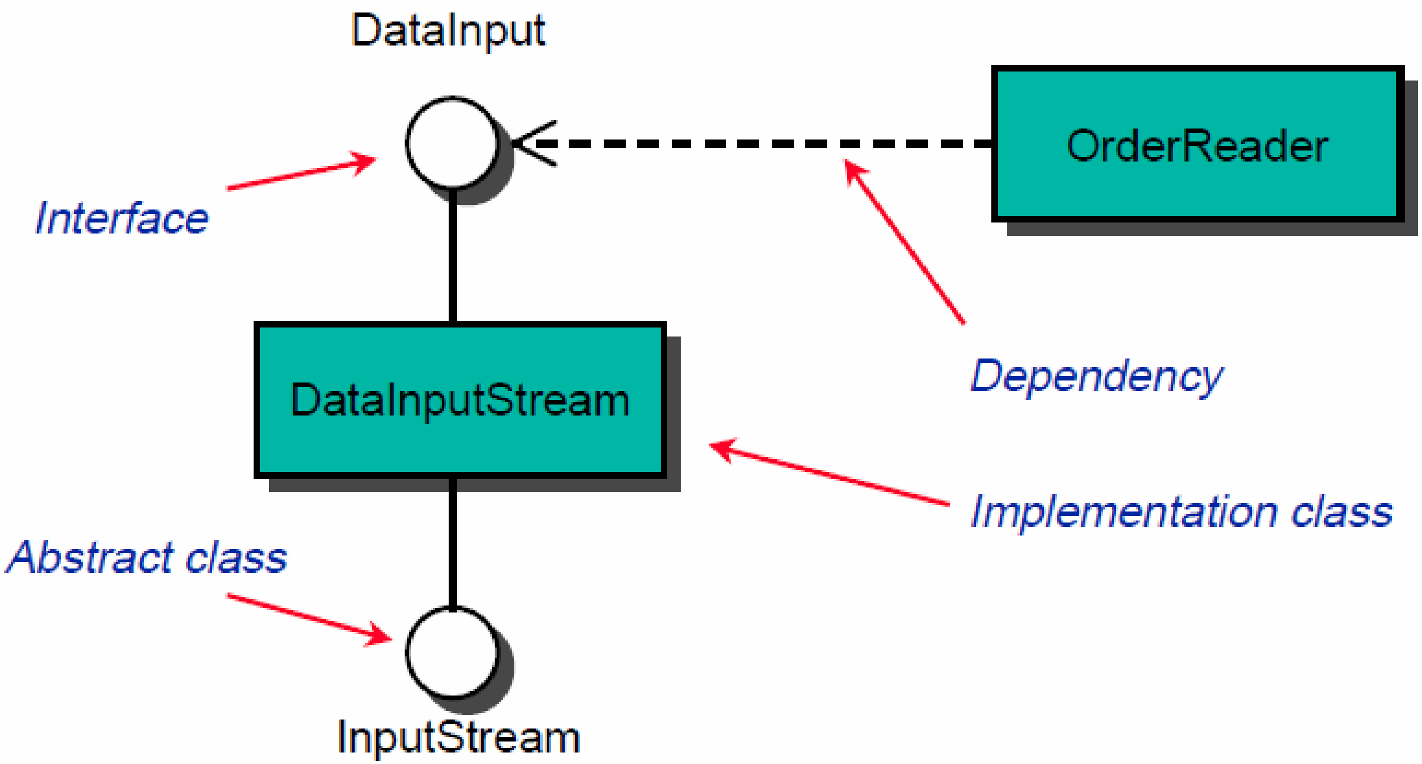
Example from Java class libraries:

- `InputStream` is an abstract class, i.e., some methods are deferred.
- `DataInput` is an interface, i.e., it implements no methods.
- `DataInputStream` is a subclass of `InputStream`; it implements the deferred methods of `InputStream`, and the methods of the interface `DataInput`.
- `OrderReader` uses only those methods of `DataInputStream` that are defined by the interface `DataInput`.



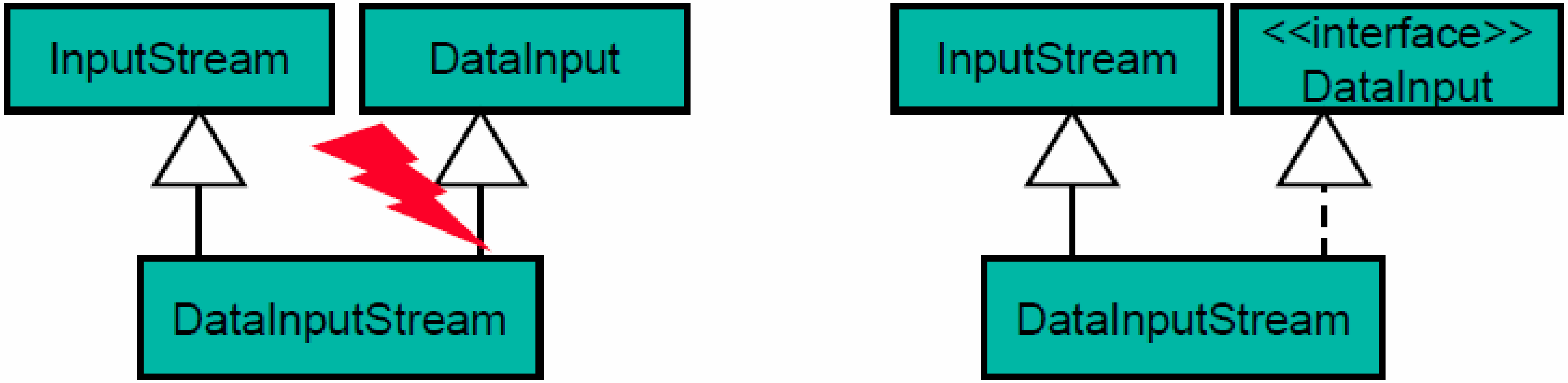
Lollipop Notation

The interfaces or abstract classes are represented by small circles (lollipops), coming off the classes that implement them.



Interfaces vs. Abstract Classes

- There is no distinction between **refining** an interface and **subclassing** an abstract class.
- Both define an interface and defer implementation.
- However, abstract classes allow to add implementation of some methods.
- An interface forces you to defer the implementation of **all** methods.
- Interfaces are used to emulate multiple inheritance, e.g., in Java.
  - A (Java) class cannot be subclass of many superclasses.
  - But it can implement different interfaces.



When and How to Use Class Diagrams

Class diagrams are the **backbone** of nearly all object-oriented methods. Especially they facilitate **code generation**.

The trouble with class diagrams and their rich notation is that they can be **extremely detailed** and therefore confusing.

- Do not try to use all the notations available to you, if you do not have to.
- Fit the **perspective** from which you are drawing the diagrams to the stage of the project.
  - If you are in **analysis**, draw **conceptual** models.
  - When working with **software**, concentrate on **specification**.
  - Draw **implementation** models only when you are illustrating a particular implementation **technique**.
- Don't draw diagrams for everything; instead **concentrate** on the **key** areas.

Any Question?