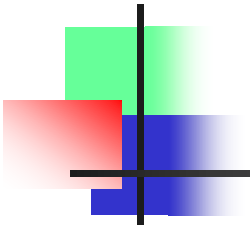
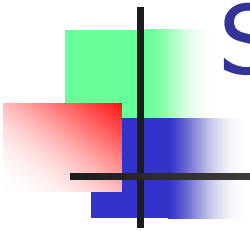
Object Oriented System Design Class Diagrams



- Marc Conrad
  - D104 (Park Square Building)
  - Email: Marc.Conrad@beds.ac.uk
  - WWW: http://perisic.com/marc

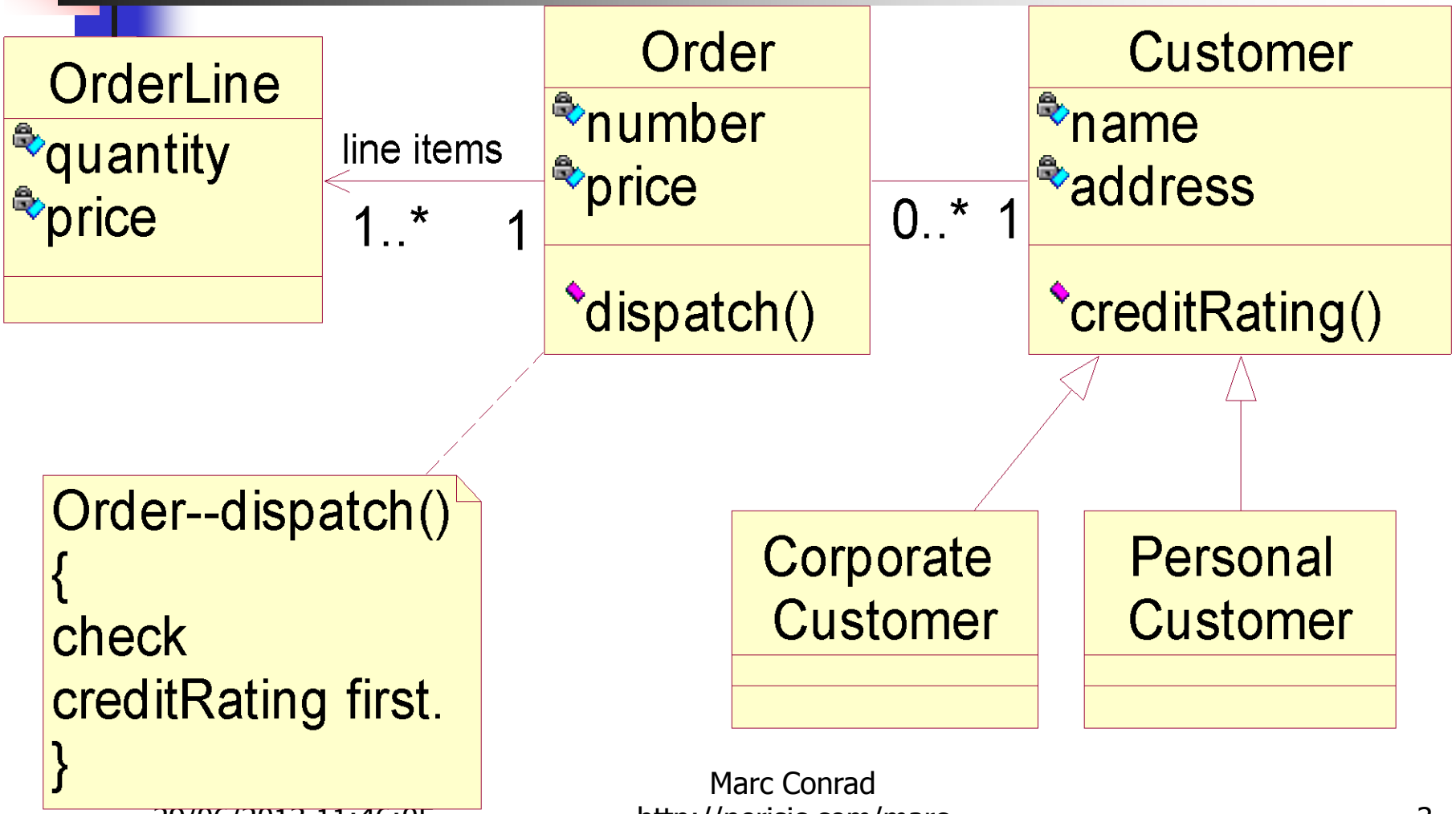


Static Models and Dynamic Models

- Class diagrams model the *static behaviour* of objects, i.e.
  - Attributes of objects
  - Operation of objects
  - Relationships between objects.

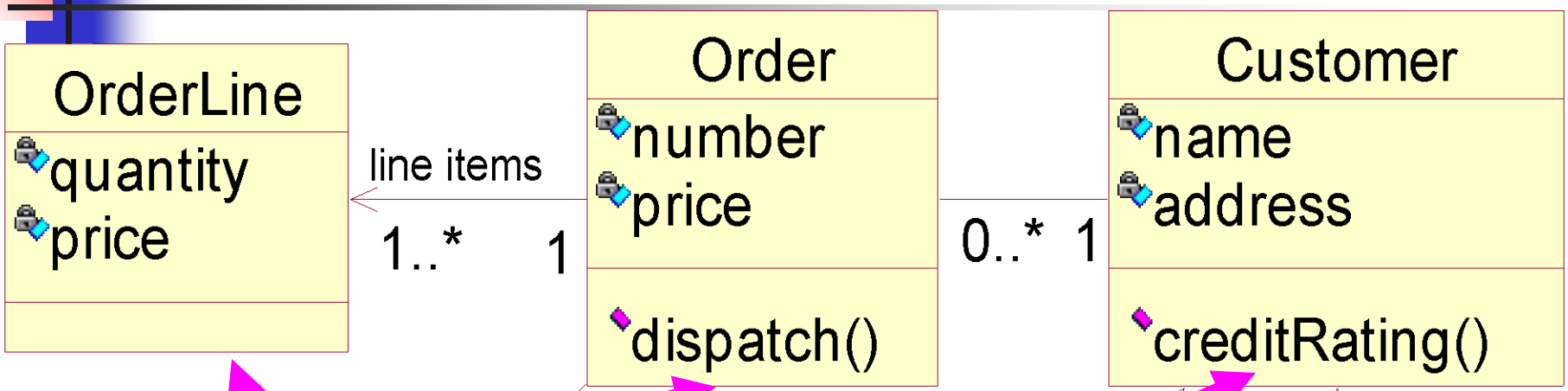
(Dynamic behaviour is modelled in a state diagram)

Example of a class diagram

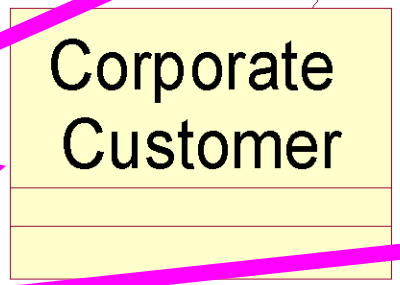


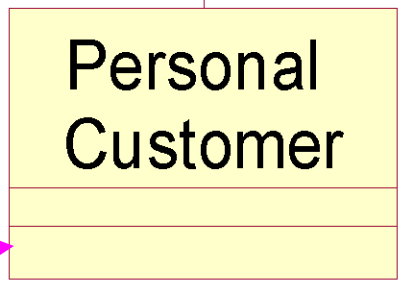
More: http://perisic.com/oosd

Elements of a class diagram
Classes

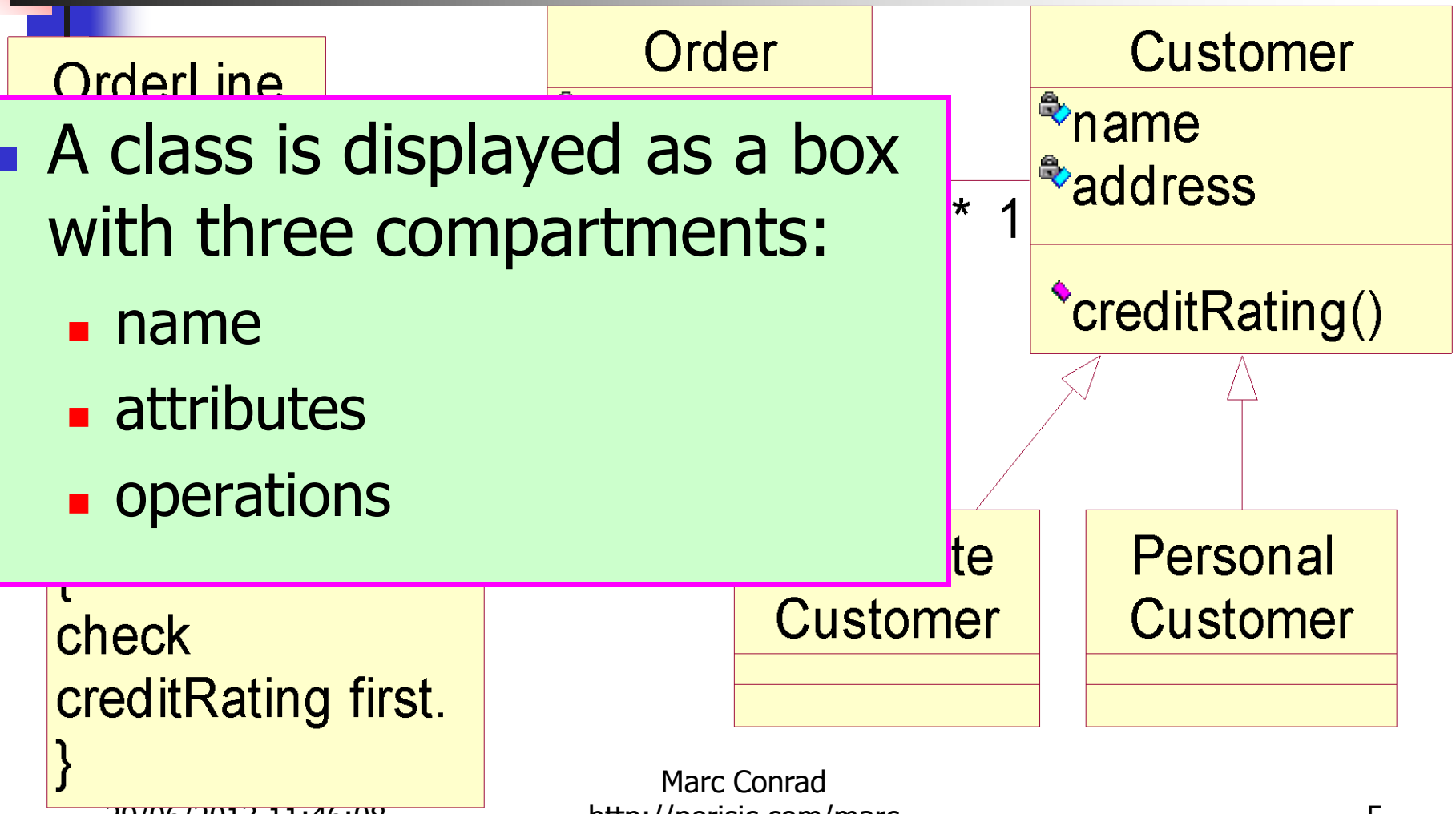


Classes are the fundamental elements in a class diagram.





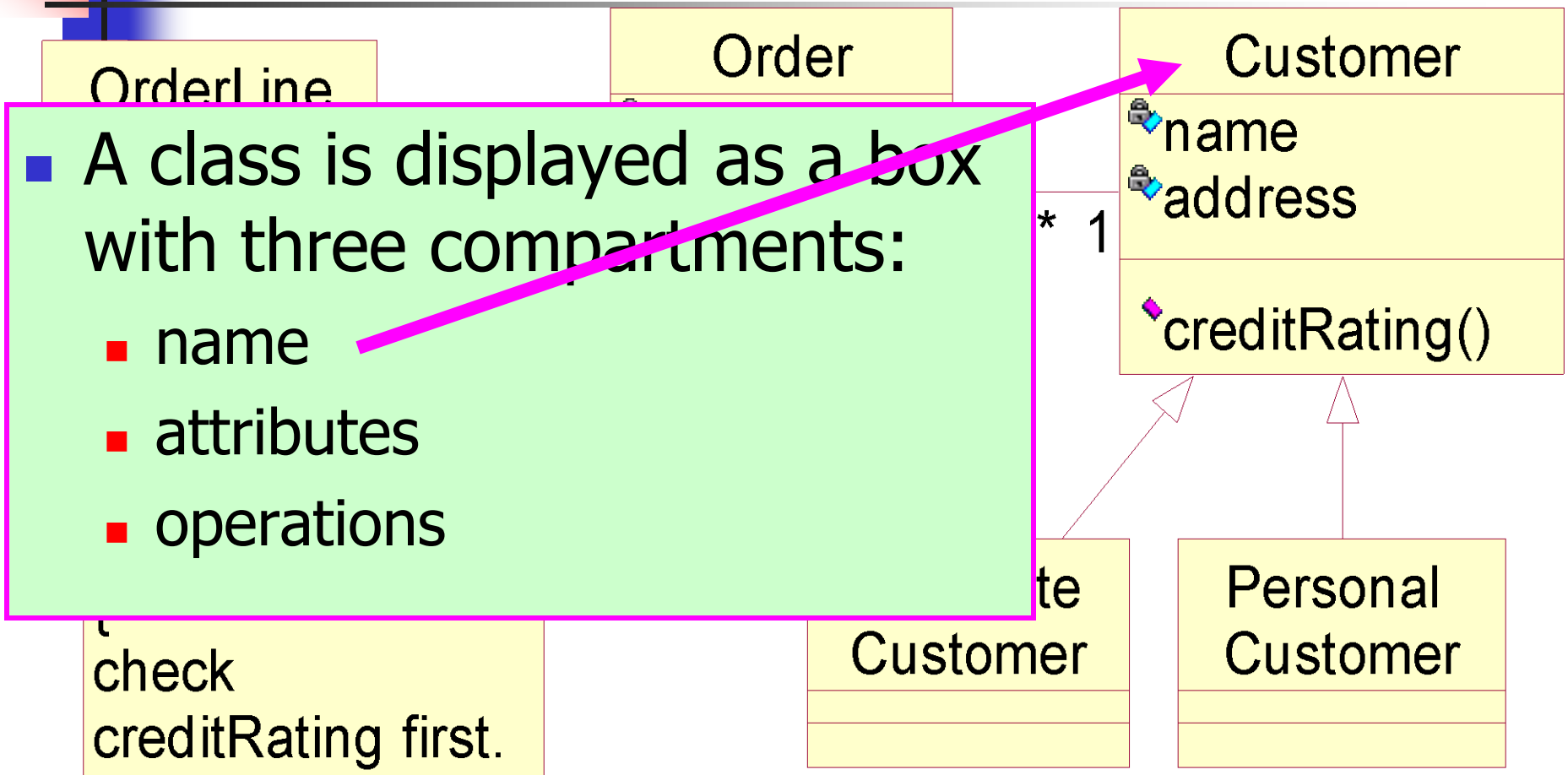
Elements of a class diagram
Structure of a class



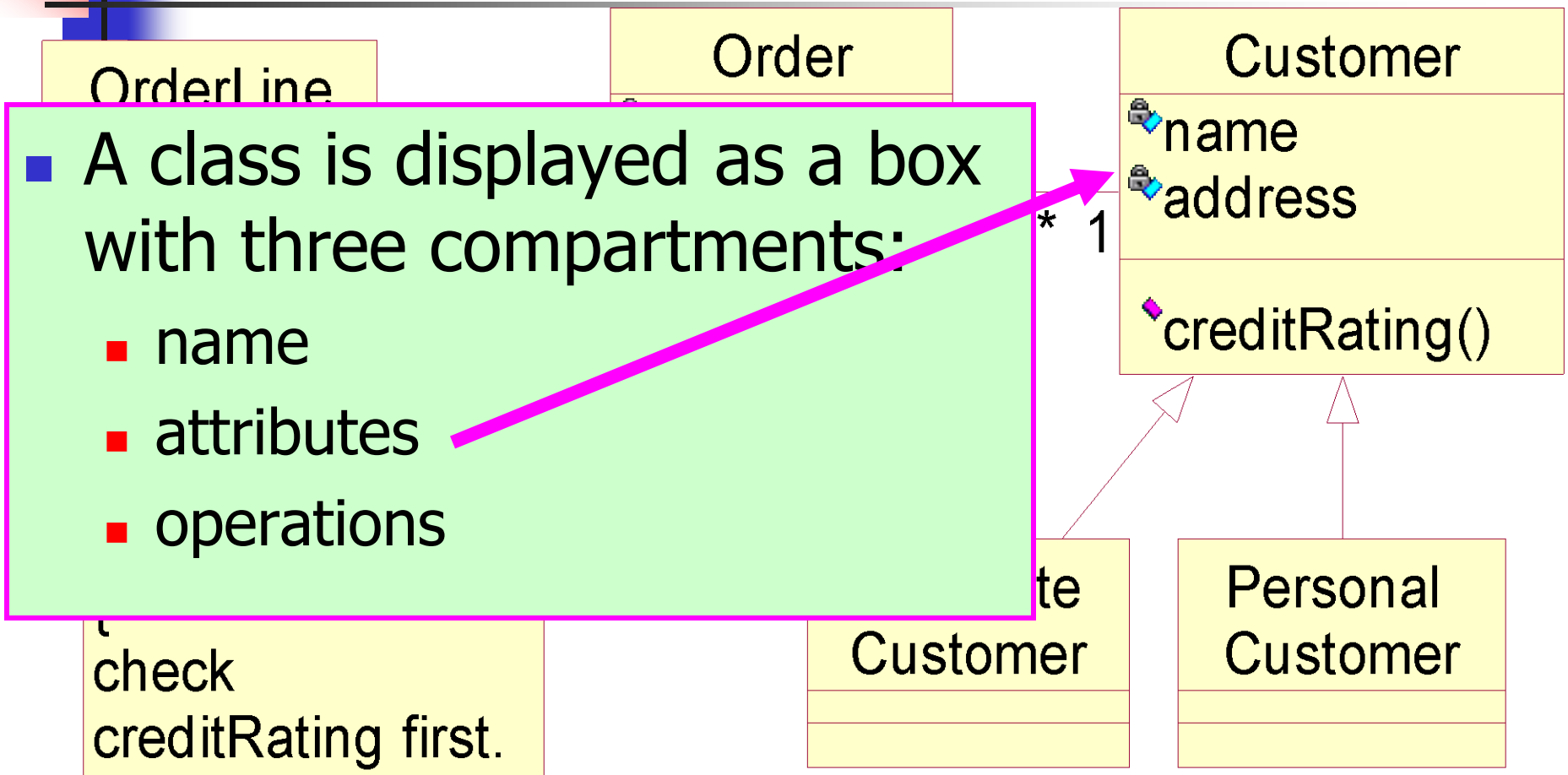
- A class is displayed as a box with three compartments:
  - name
  - attributes
  - operations

Elements of a class diagram
Structure of a class

- A class is displayed as a box with three compartments:
  - name
  - attributes
  - operations

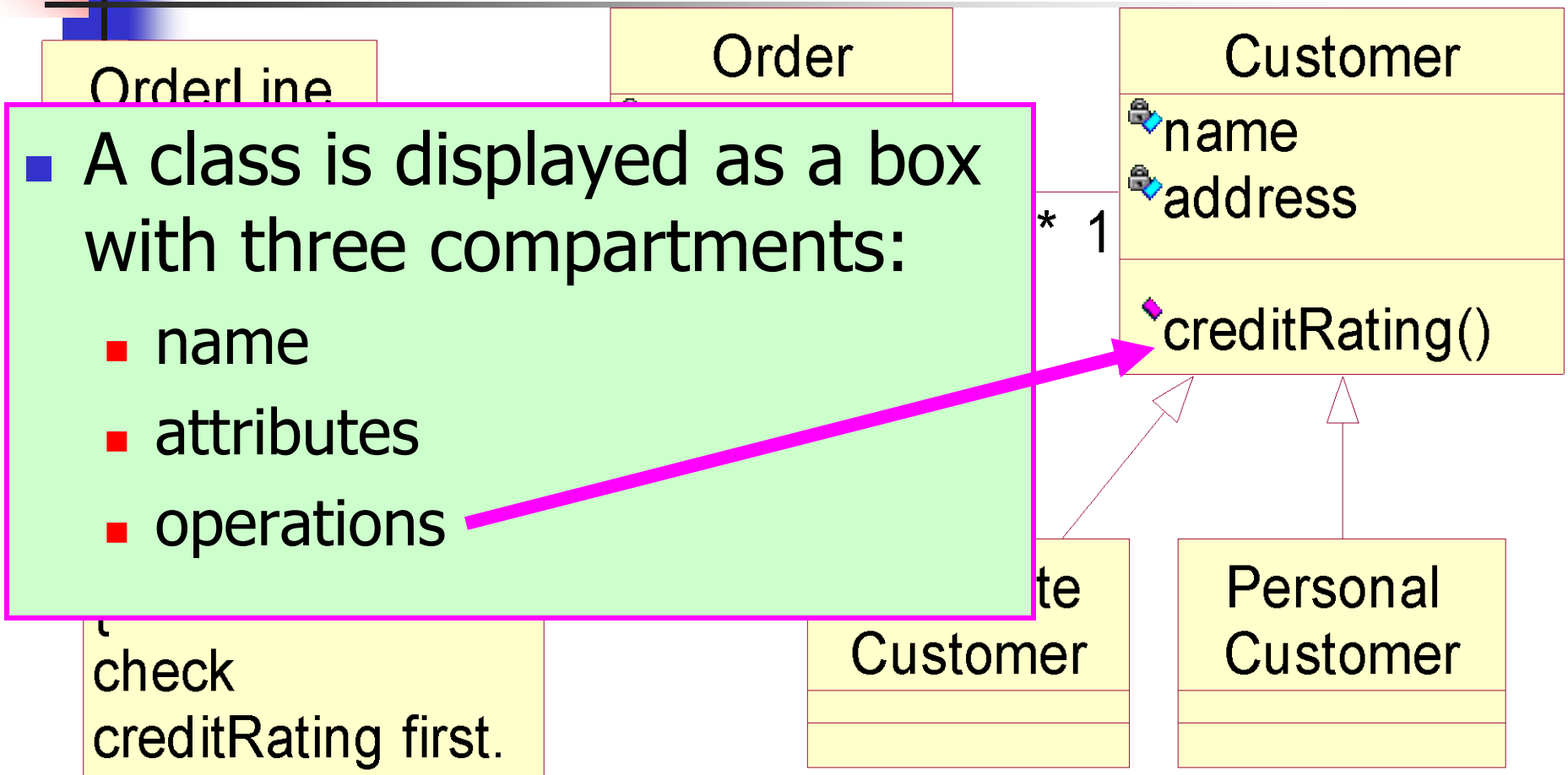


Elements of a class diagram
Structure of a class



- A class is displayed as a box with three compartments:
  - name
  - attributes
  - operations

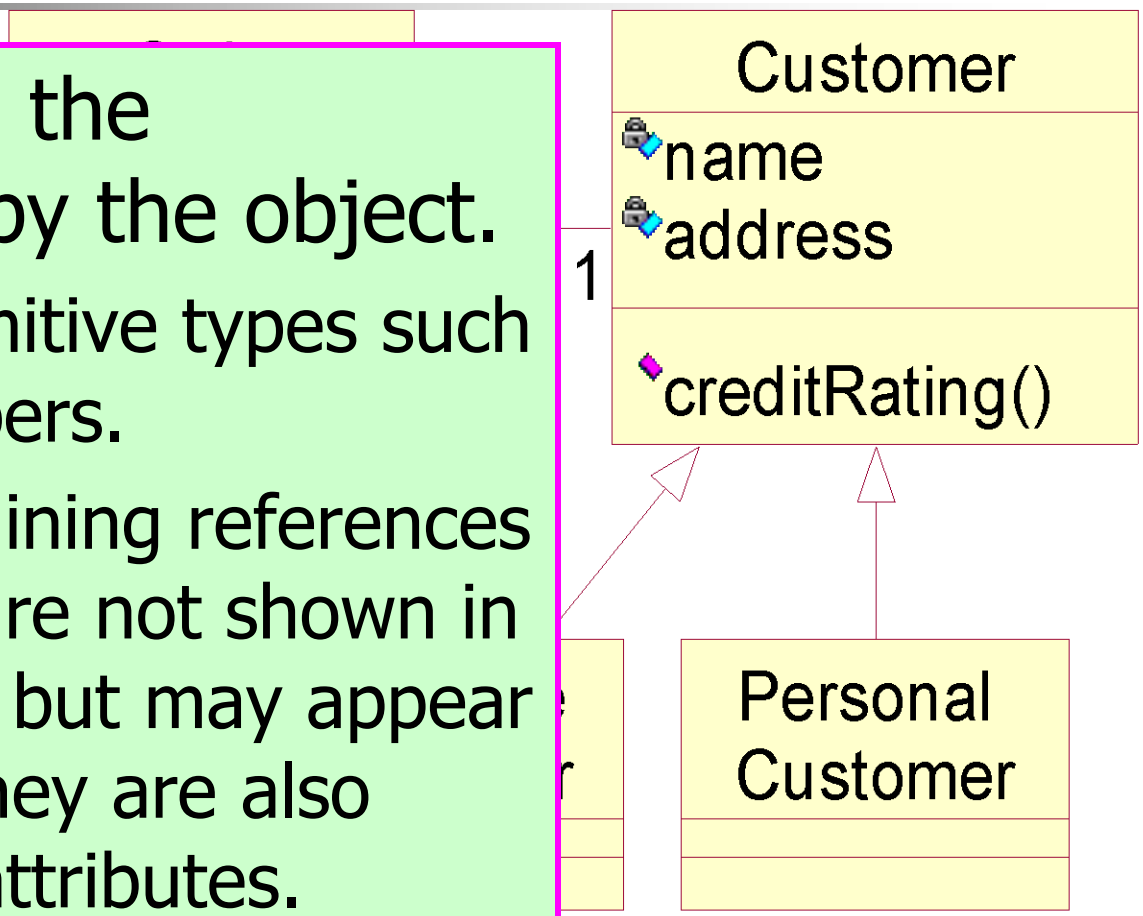
Elements of a class diagram
Structure of a class



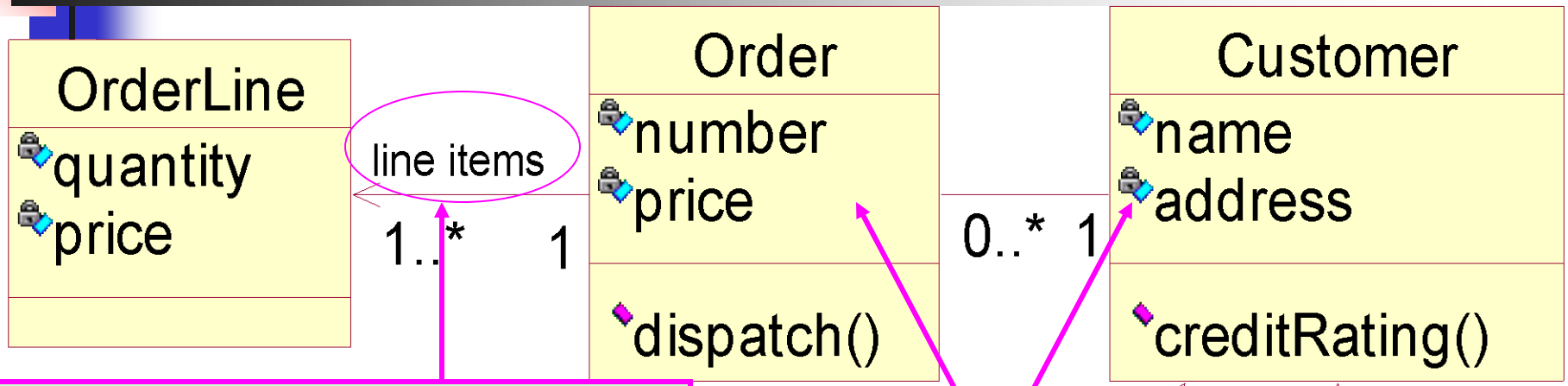
- A class is displayed as a box with three compartments:
  - name
  - attributes
  - operations

Elements of a class diagram Attributes

- Attributes contain the information held by the object.
  - They refer to primitive types such as string or numbers.
  - "Attributes" containing references to other objects are not shown in the compartment but may appear as role names. They are also called reference attributes.

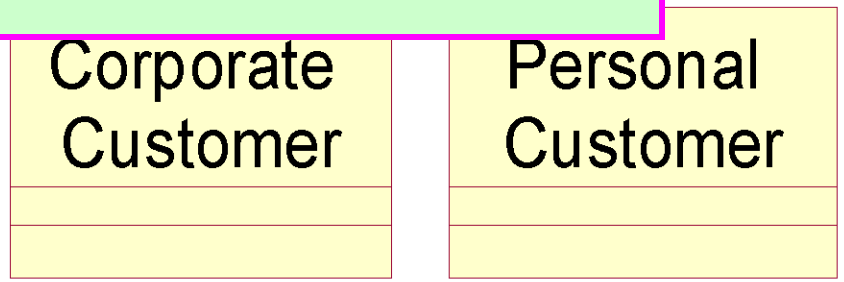


Elements of a class diagram roles and attributes



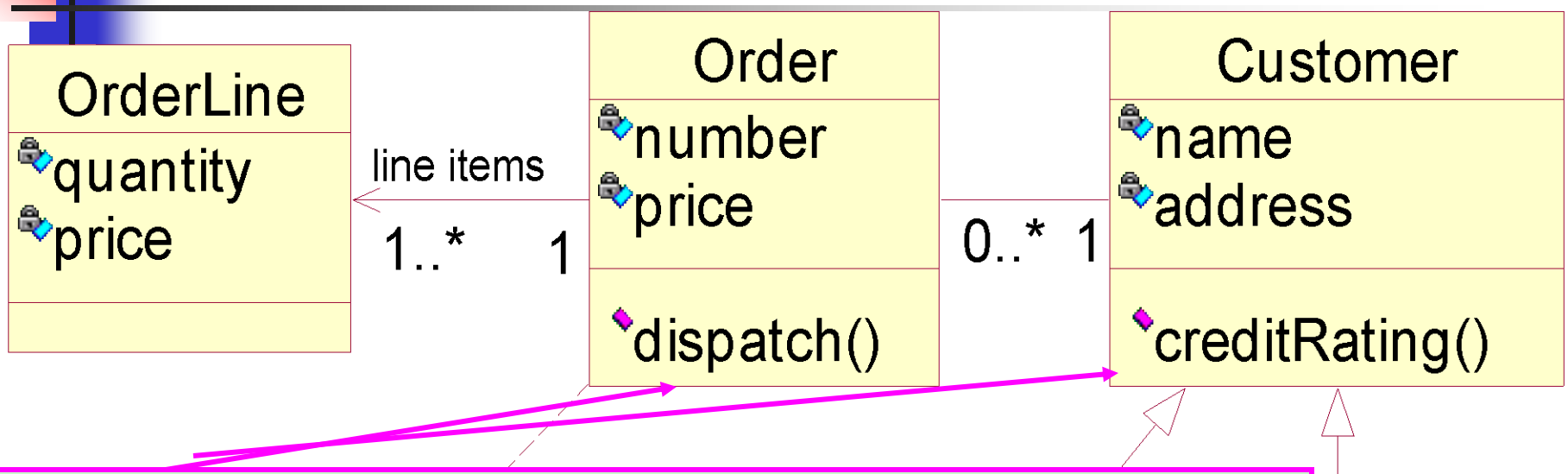
A role name (will be implemented as a reference attribute in the Order class).

Primitive types



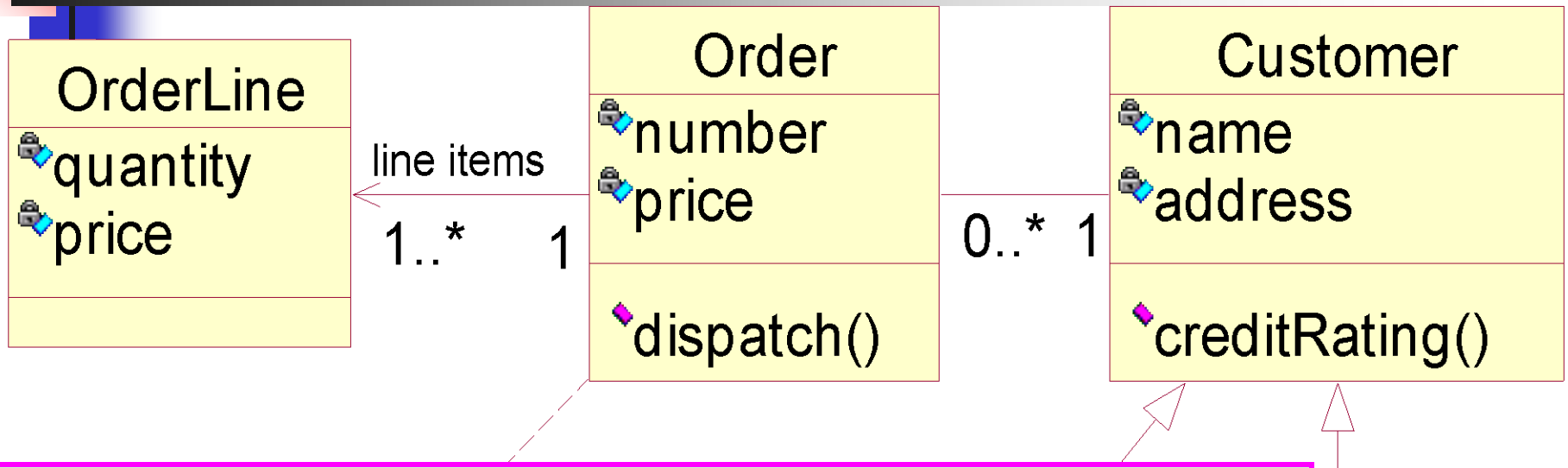
More: http://perisic.com/oosd

Elements of a class diagram Operations



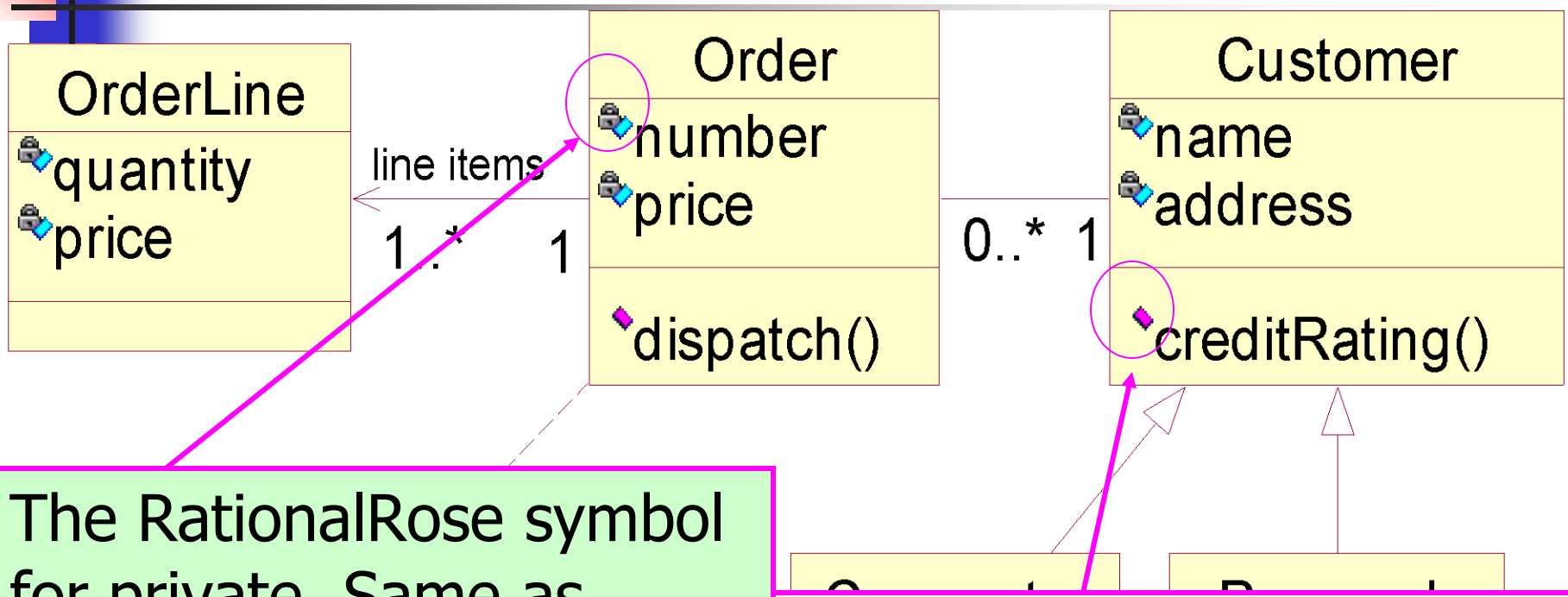
- Operations (Methods)
  - They refer to the *behaviour* of the object.
  - Operations are implied by the sequence of events in a sequence diagram.

Elements of a class diagram Operations & Attributes



- Operations and Attributes can be private, protected or public. This is reflected by the symbols: +, #, -. RationalRose uses other symbols.

Elements of a class diagram private/public



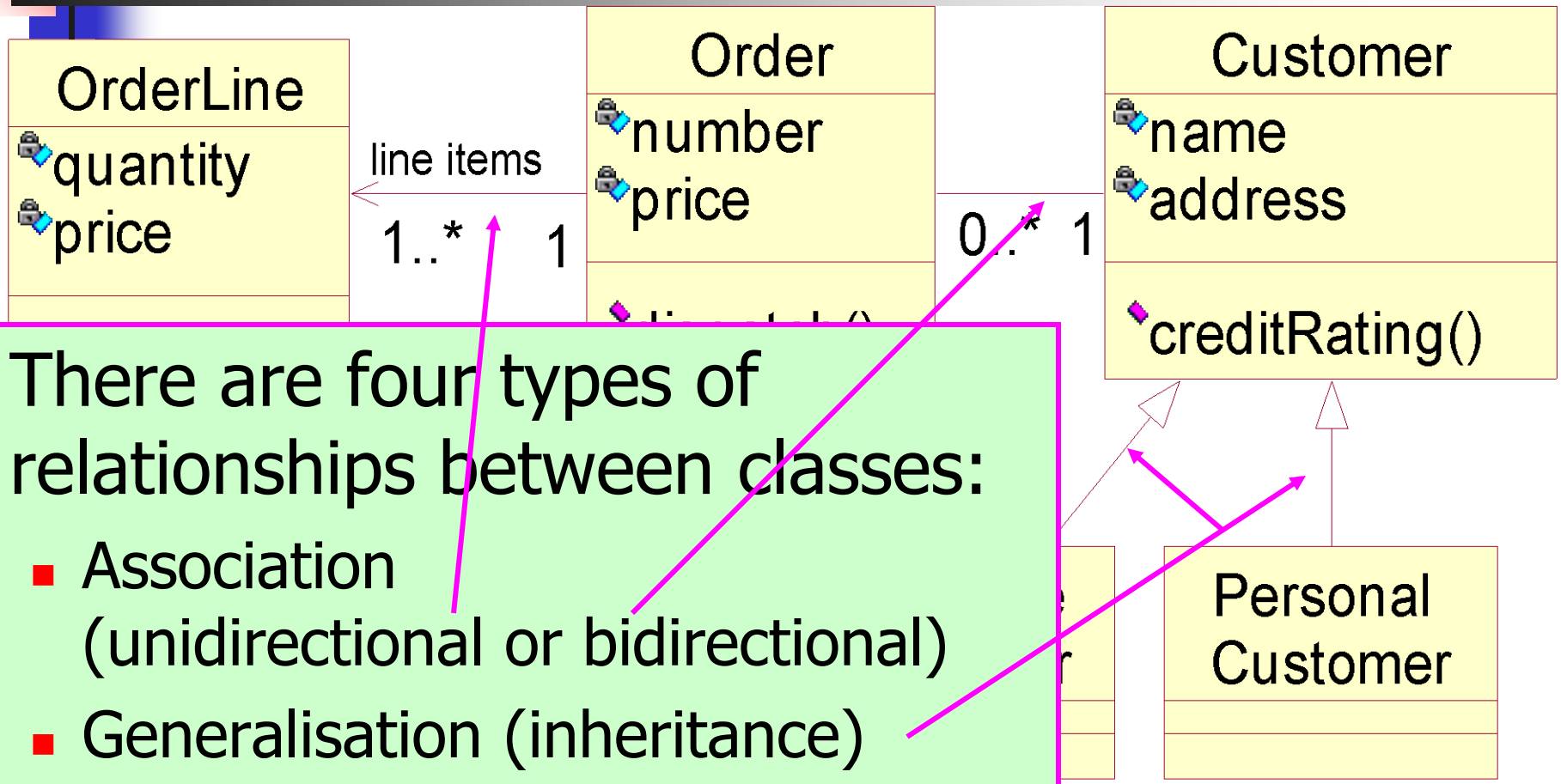
- The RationalRose symbol for private. Same as "-number"

- The RationalRose symbol for public. Same as "+creditRating()"

creditRating first.
}

More: http://perisic.com/oosd

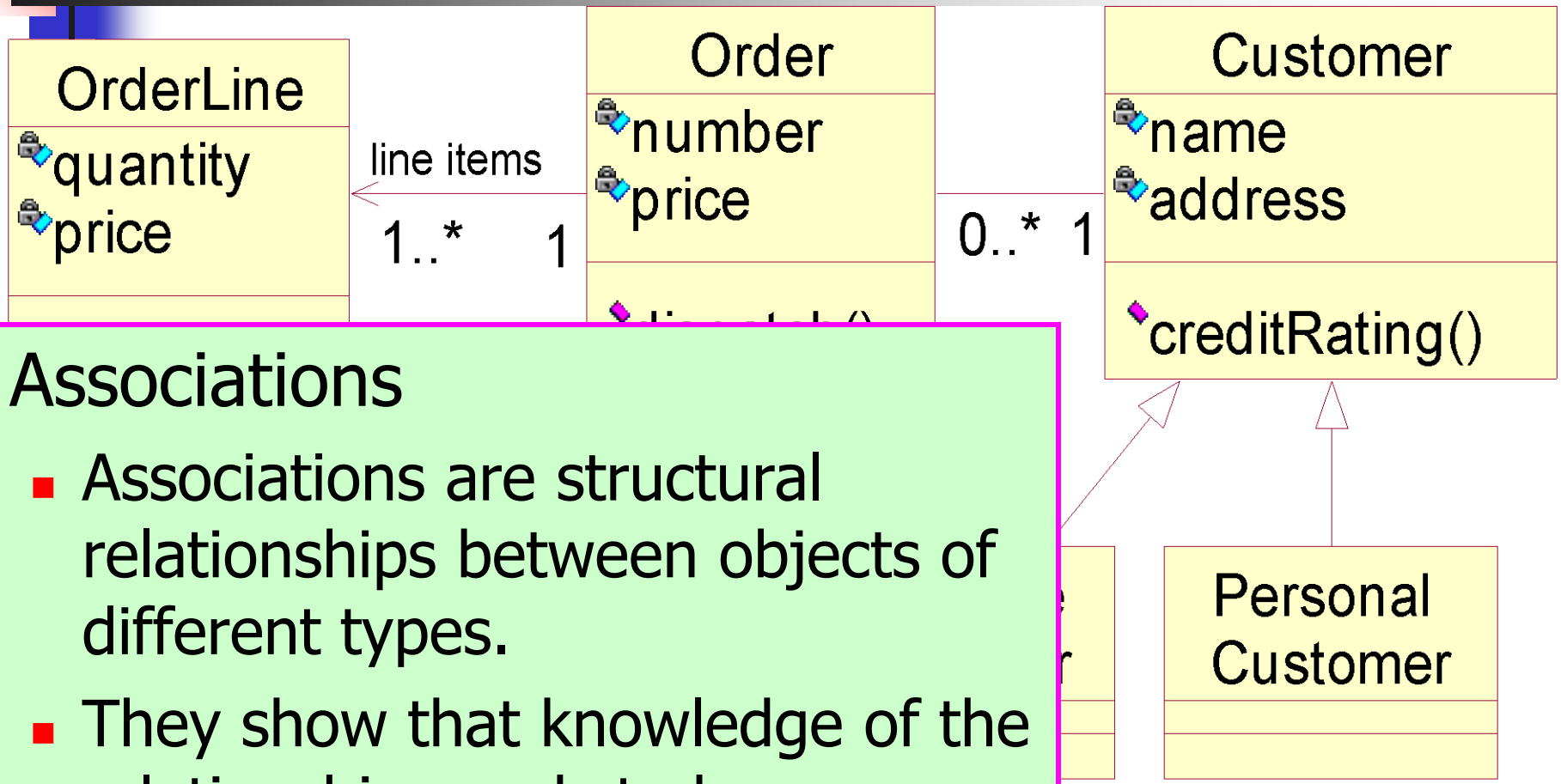
Elements of a class diagram Relationships



- There are four types of relationships between classes:
  - Association (unidirectional or bidirectional)
  - Generalisation (inheritance)
  - Dependencies & Aggregation

More: http://perisic.com/oosd

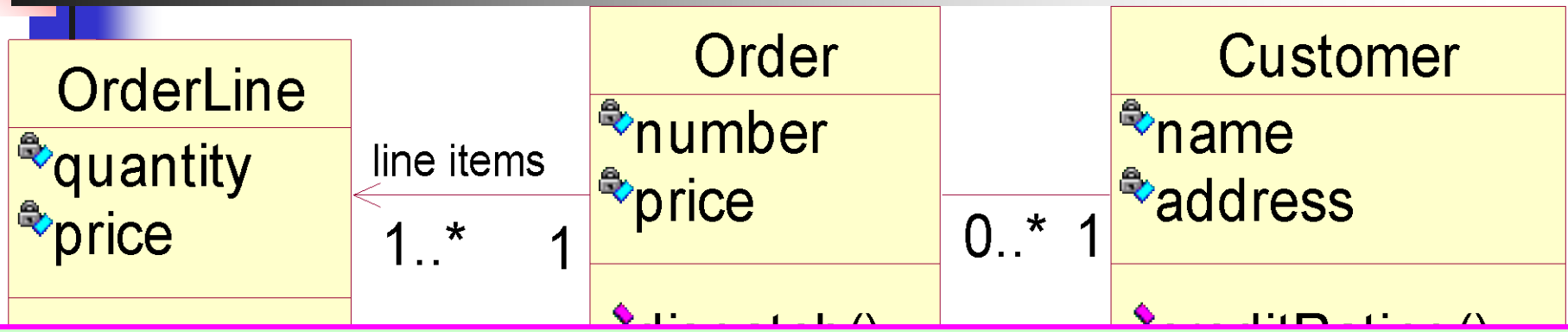
Elements of a class diagram Associations



- Associations
  - Associations are structural relationships between objects of different types.
  - They show that knowledge of the relationship needs to be preserved for some duration.

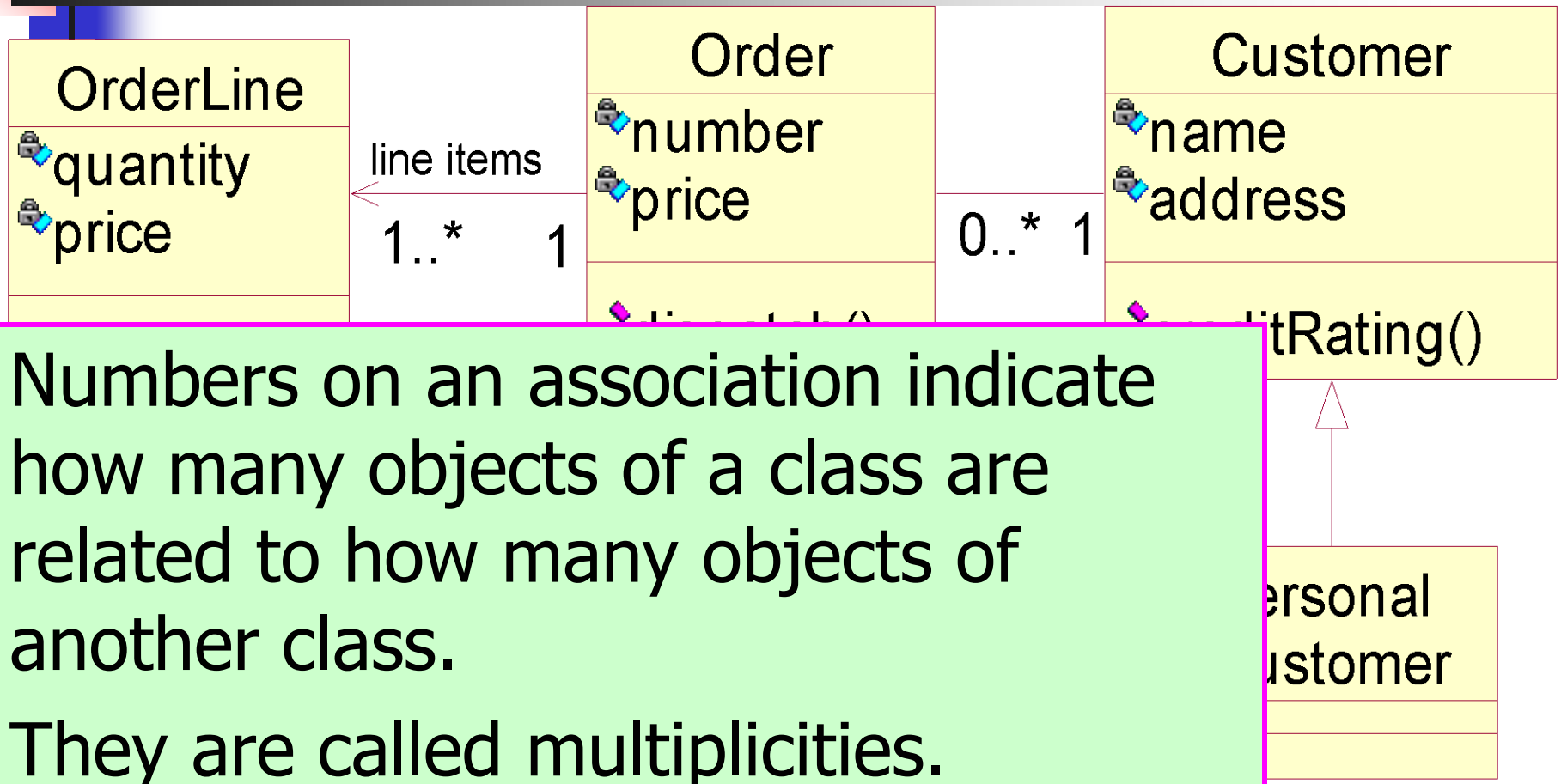
More: http://perisic.com/oosd

Elements of a class diagram Arrows on Associations



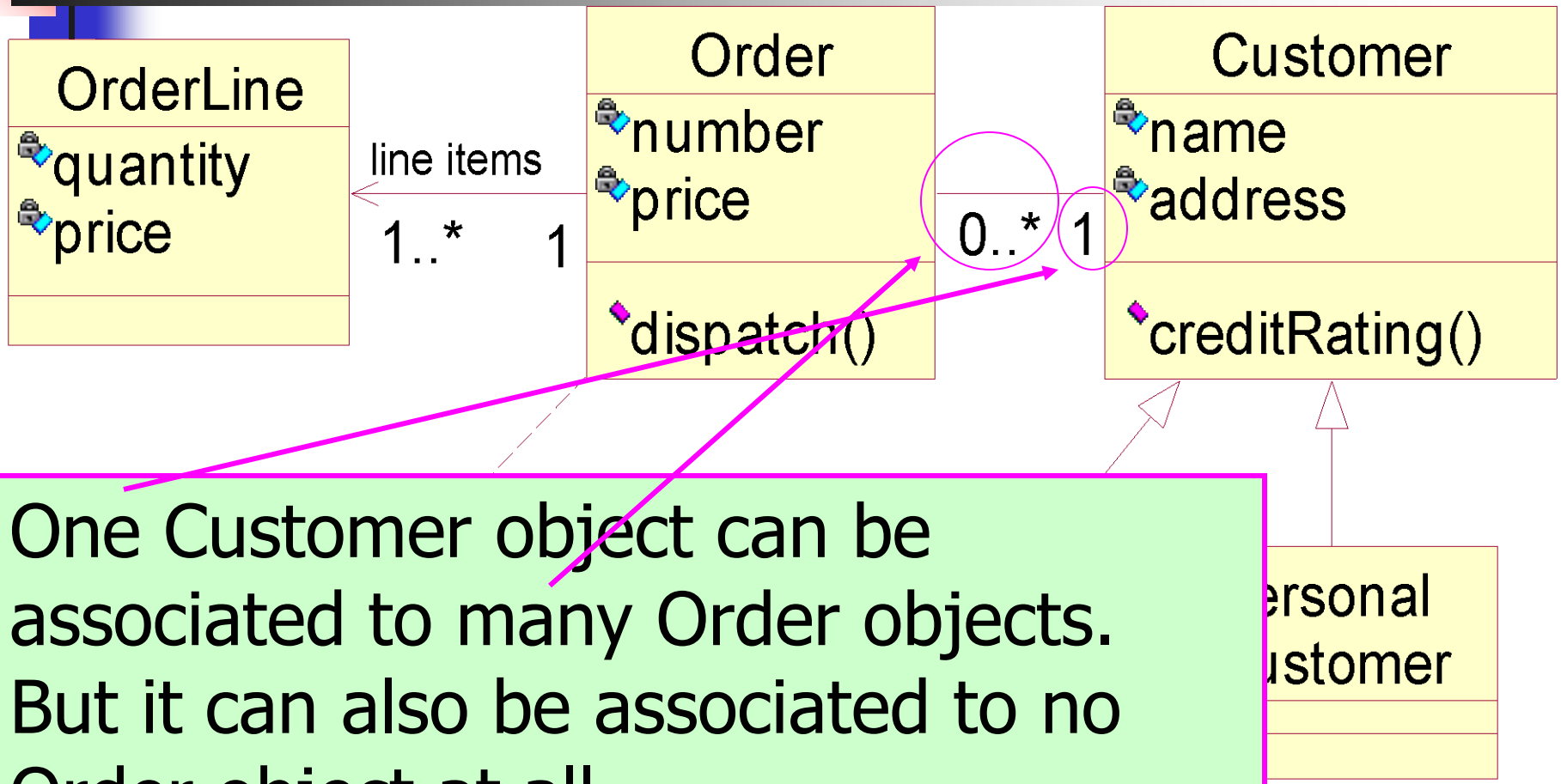
- Arrows on Associations.
  - The arrow on an association indicates a visibility relationship. OrderLine is visible by Order.
  - No arrow on an association means visibility in both directions. Order knows about Customer and Customer knows Order.

Elements of a class diagram Multiplicities



- Numbers on an association indicate how many objects of a class are related to how many objects of another class.
- They are called multiplicities.

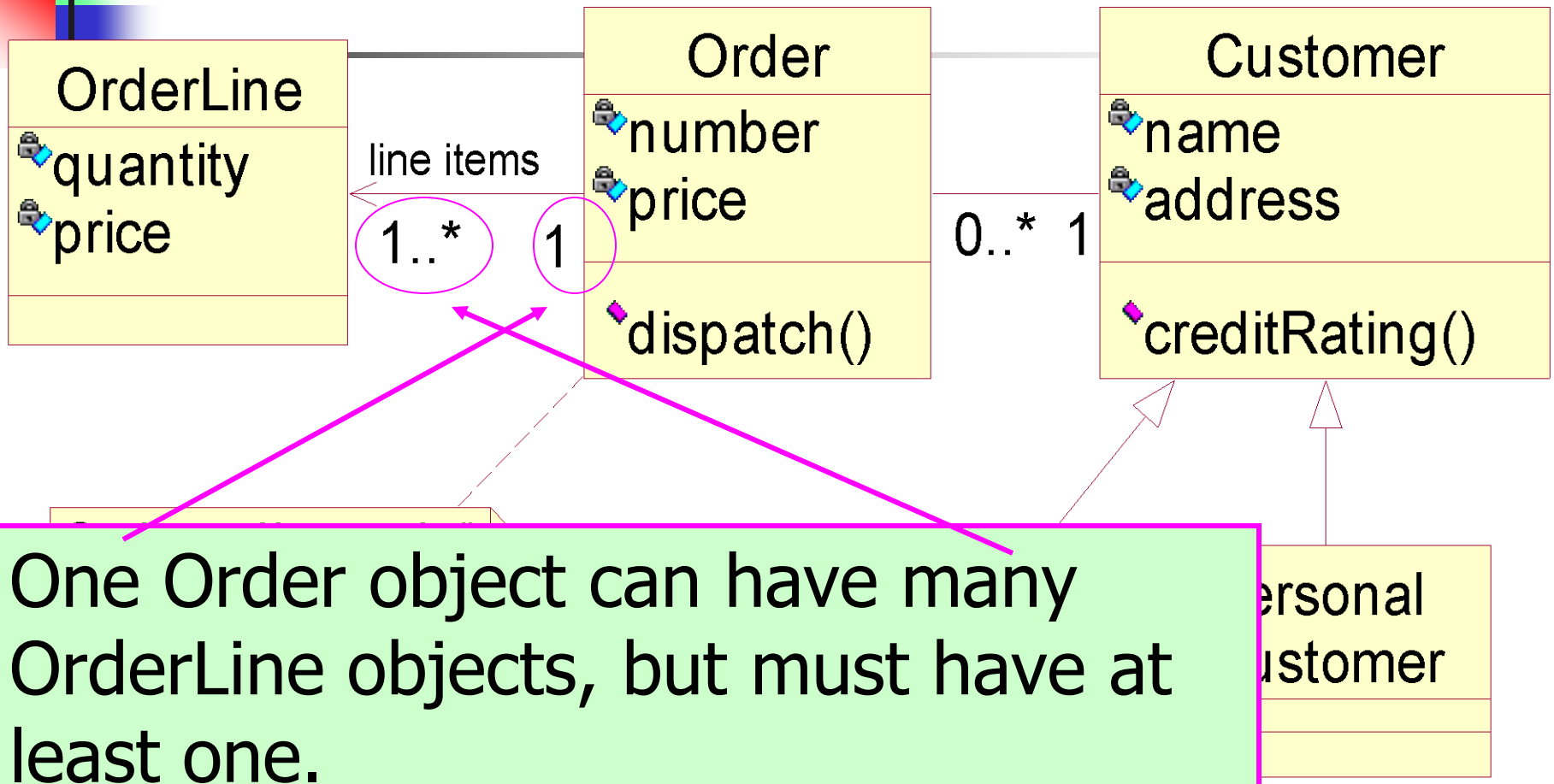
Elements of a class diagram Multiplicities



- One Customer object can be associated to many Order objects. But it can also be associated to no Order object at all.

More: http://perisic.com/oosd

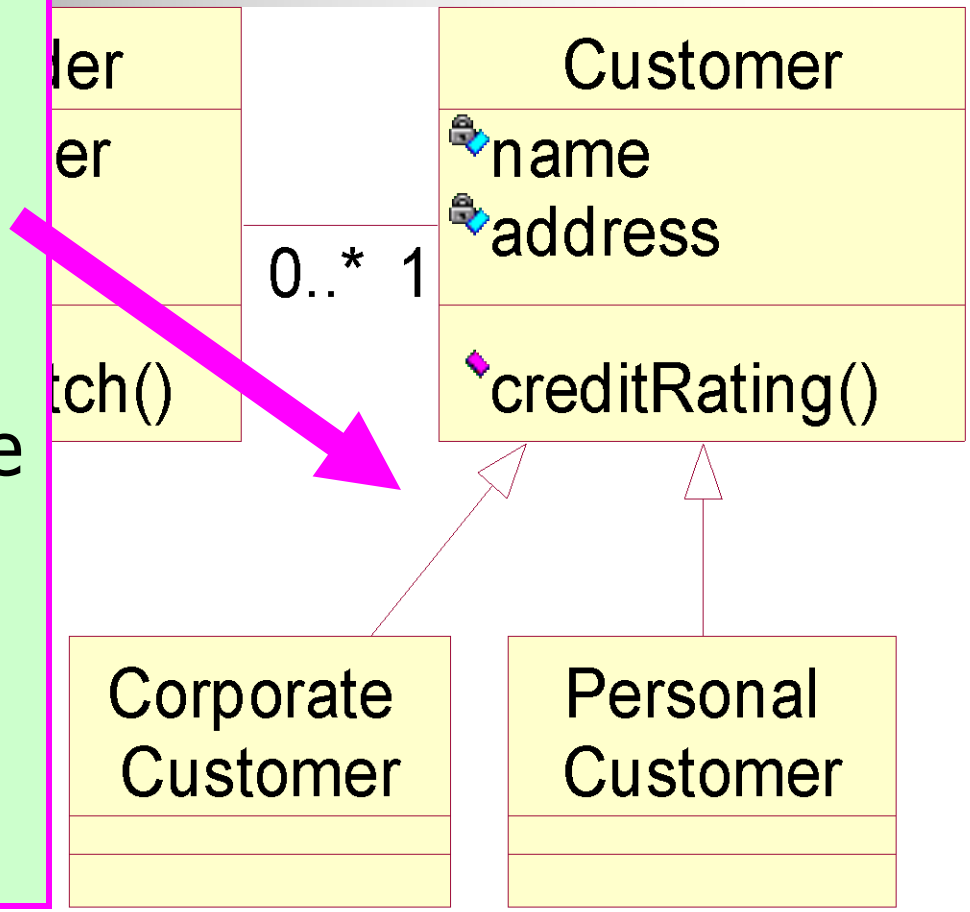
Elements of a class diagram Multiplicities



- One Order object can have many OrderLine objects, but must have at least one.

a class diagram Generalisation

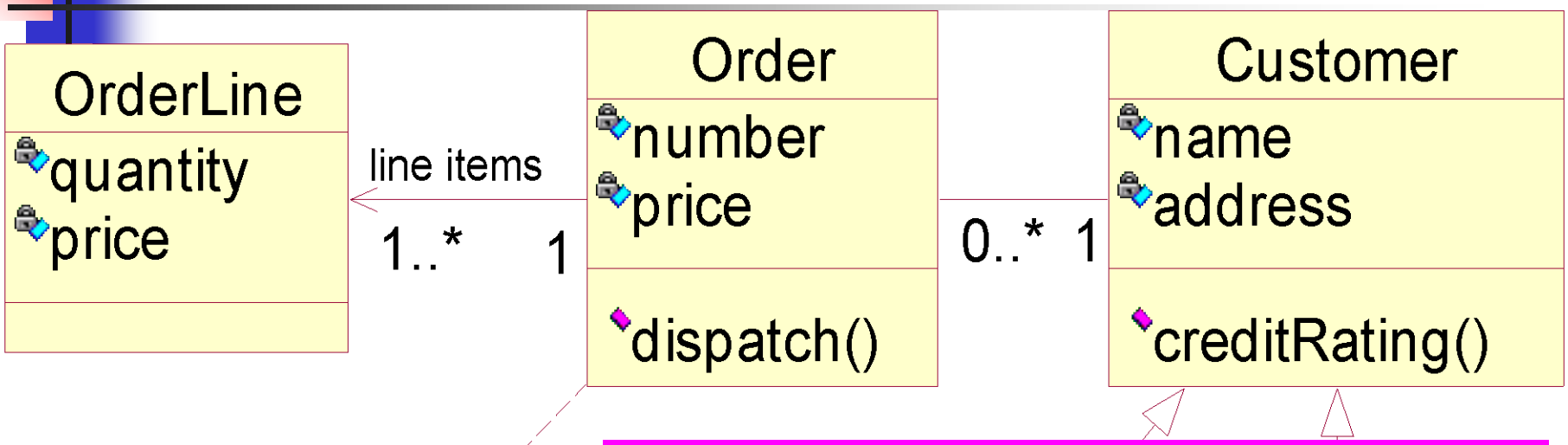
- Generalisation
- If two or more classes have some common attributes and methods, these attributes and methods can be collected and placed in a super class (parent class).
- Generalisation reflects the inheritance relationship known from C++ and Java.




```
}
```

More: http://perisic.com/oosd

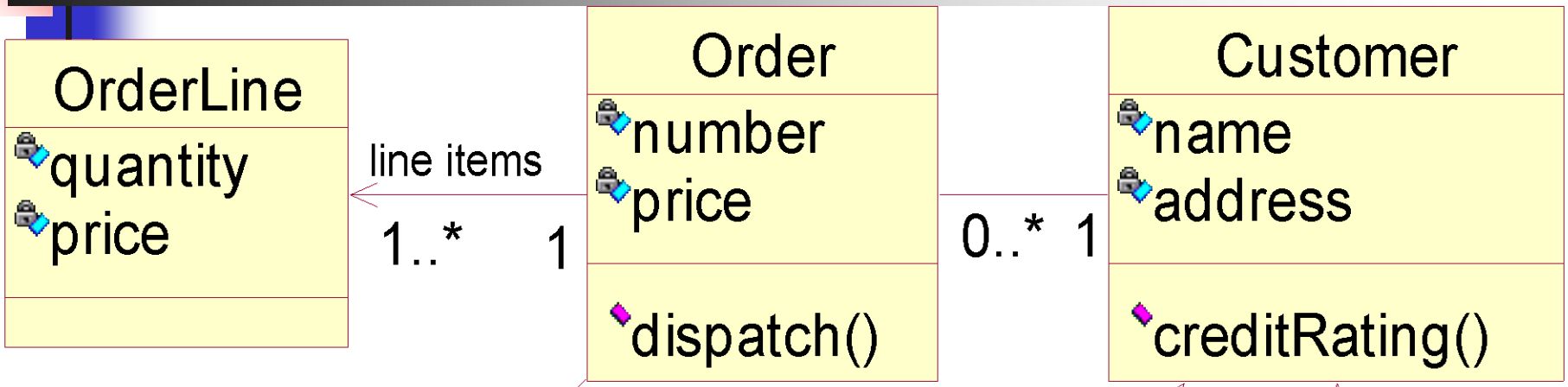
Elements of a class diagram Constraints



Order--dispatch()
{
check
creditRating first.
}

- Constraints
  - A constraint is attached to an element. It has semantic influence on the element.

Elements of a class diagram Constraints

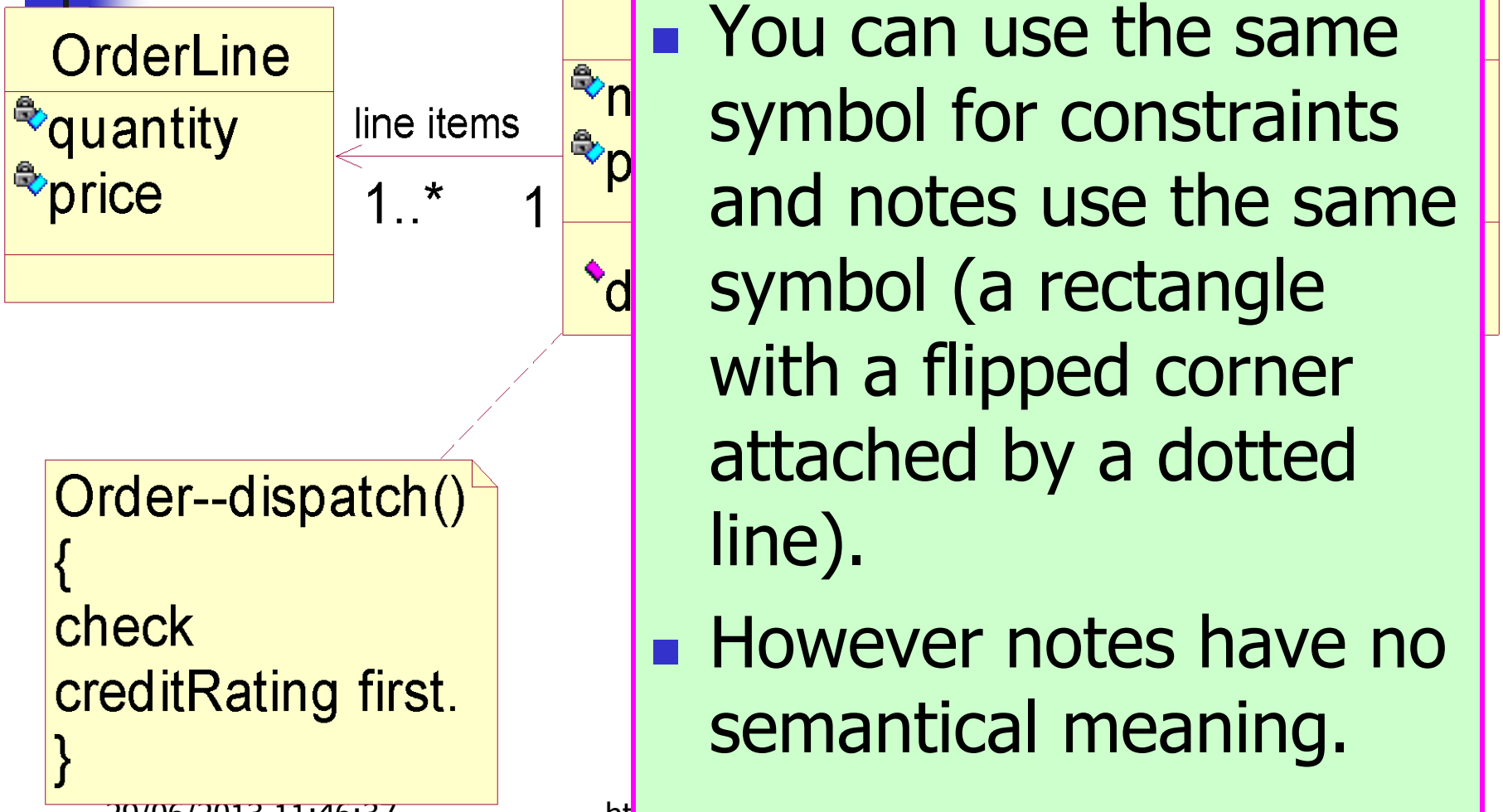



```
Order--dispatch()
{
check
creditRating first.
}
```

- Pre-condition
  - The condition of an operation before being executed.
- Post-condition
  - The expected consequence of an operation.

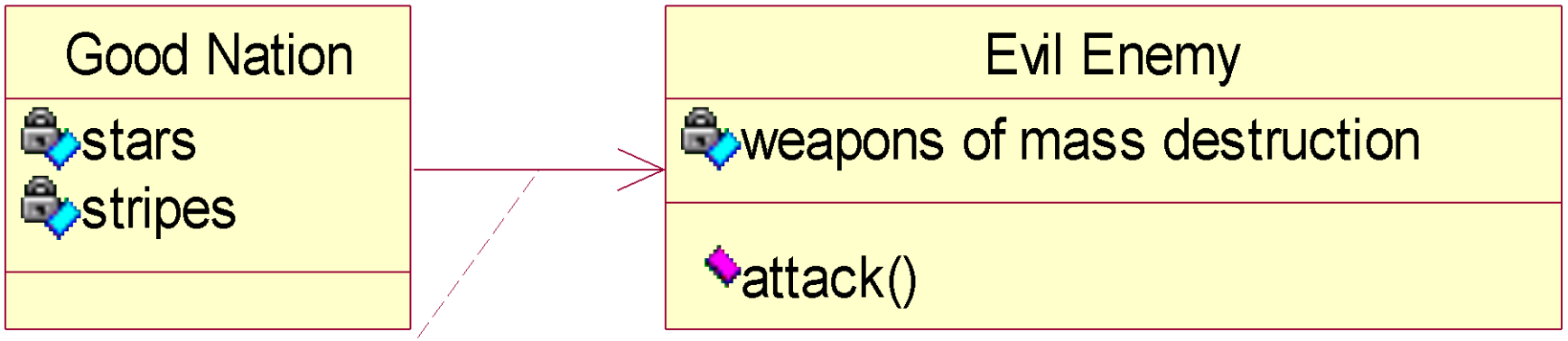
More: http://perisic.com/oosd

Elements of a class diagram Constraints & Notes



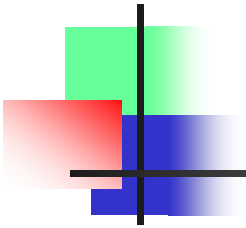
- You can use the same symbol for constraints and notes use the same symbol (a rectangle with a flipped corner attached by a dotted line).
- However notes have no semantical meaning.

Element of a class diagram Notes and Constraints



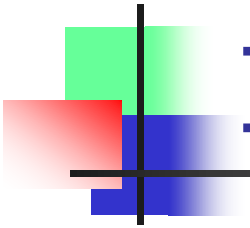
Enemy-attack(){
get UN resolution
first
}

Misinterpreting constraints as simple notes may lead to major problems



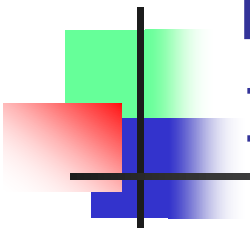
How to make a class diagram.

1. Identify all the classes participating in the software solution (analysis & sequence diagrams).
2. Draw them in a class diagram.
3. Identify the attributes.
4. Identify the methods (from the sequence diagram).
5. Add associations, generalisations, aggregations and dependencies.
6. Add other stuff (roles, constraints, ...)



Class diagrams and Interaction diagrams.

- In practice class diagrams and interaction diagrams are usually created in parallel.
- Many classes, methods, etc. may be sketched out in a class diagram prior to drawing a sequence diagram.
- Note also that a conceptual model and/or the analysis model can be used as starting points for the class diagram.

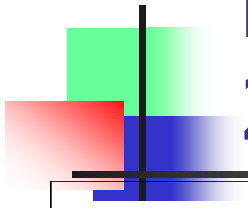


Example (how to make a class diagram) 1. Identify classes

We investigate the "return item" Use Case of the Recycling machine.

- From the analysis we find the following classes:
  - Customer Panel
  - Deposit item receiver
  - Receipt basis
  - Deposit item
  - Receipt printer
  - Can, Bottle, Crate

Example (how to make a class diagram)
2. Draw them in a class diagram



Customer Panel

Deposit item receiver

Receipt basis

Deposit item

Receipt printer

Can

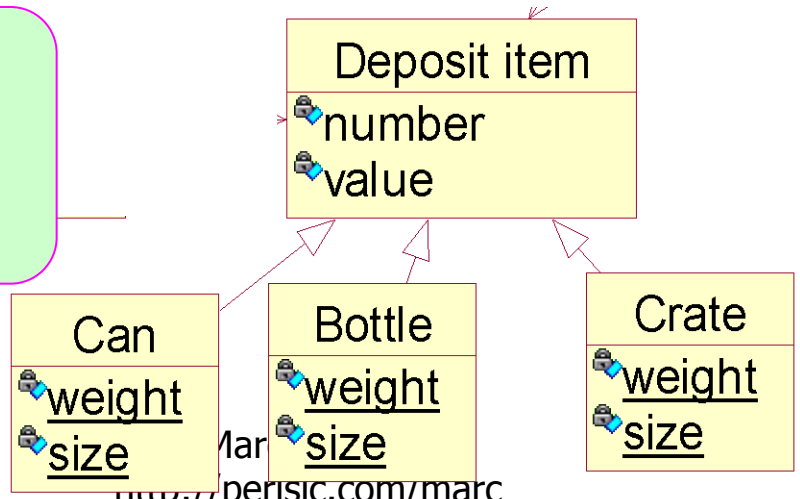
Bottle

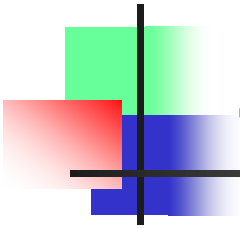
Crate

Example (how to make a class diagram) 3. Identify attributes

- Classes which contain data are in the Deposit item hierarchy.
  - For checking & classifying an item we need the *weight* and size of a Can, Bottle, and Crate.
  - For collecting the data at the Receipt basis each Deposit Item gets a *number* and a *value*.

Underlined atteributes show class (static) variables.



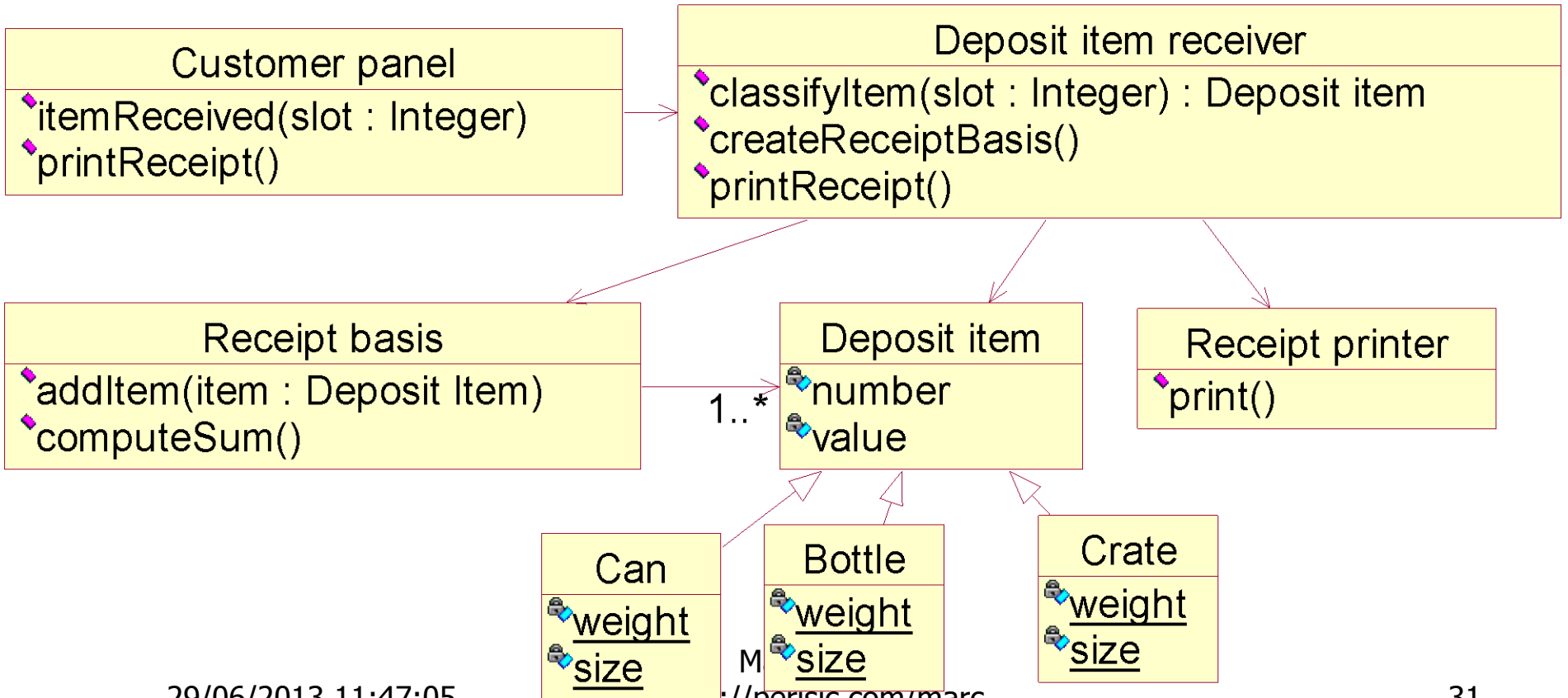


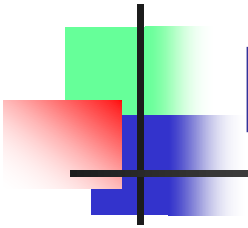
Example (how to make a class diagram) 4. Identify methods

- The return item use case suggests the following two methods for the Customer Panel:
  - itemReceived(slot : Integer)
  - printReceipt()
- Following the sequence of events in the sequence diagram we obtain then:
  - *Deposit item receiver:* classifyItem(), createReceiptBasis(), printReceipt()
  - *Receipt basis:* addItem(), computeSum(),
  - *Receipt printer:* print().
- We don't show accessor and modifier methods in order to keep the diagram simple.

Example (how to make a class diagram) 5. Add associations

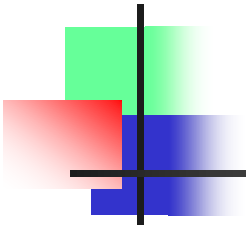
- Associations show navigability between classes





Relationships between classes

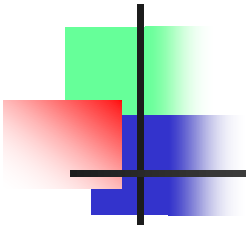
- There are four possible relationships between classes.
  - Association
  - Dependency
  - Generalisation
  - Aggregation



Relationships between classes

- There are four possible relationships between classes.
  - Association
  - Dependency
  - Generalisation
  - Aggregation

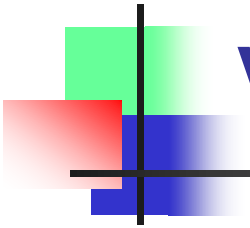
- Association and dependency are in the context of visibility.



Relationships between classes

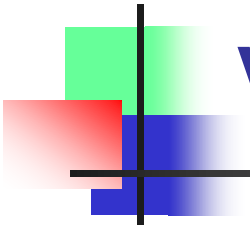
- There are four possible relationships between classes.
  - Association
  - Dependency
  - Generalisation
  - Aggregation

- Generalisation and aggregation may be considered as special versions of association.



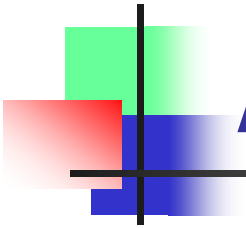
Visibility

- Why do we consider visibility?
- Object Oriented design is about sending messages between objects.
- For an object A to send a message to an object B, B must be visible to A.
- Example: The Deposit Item Receiver cannot send a message to the Printer, if it is not visible for the Deposit Item Receiver



Visibility

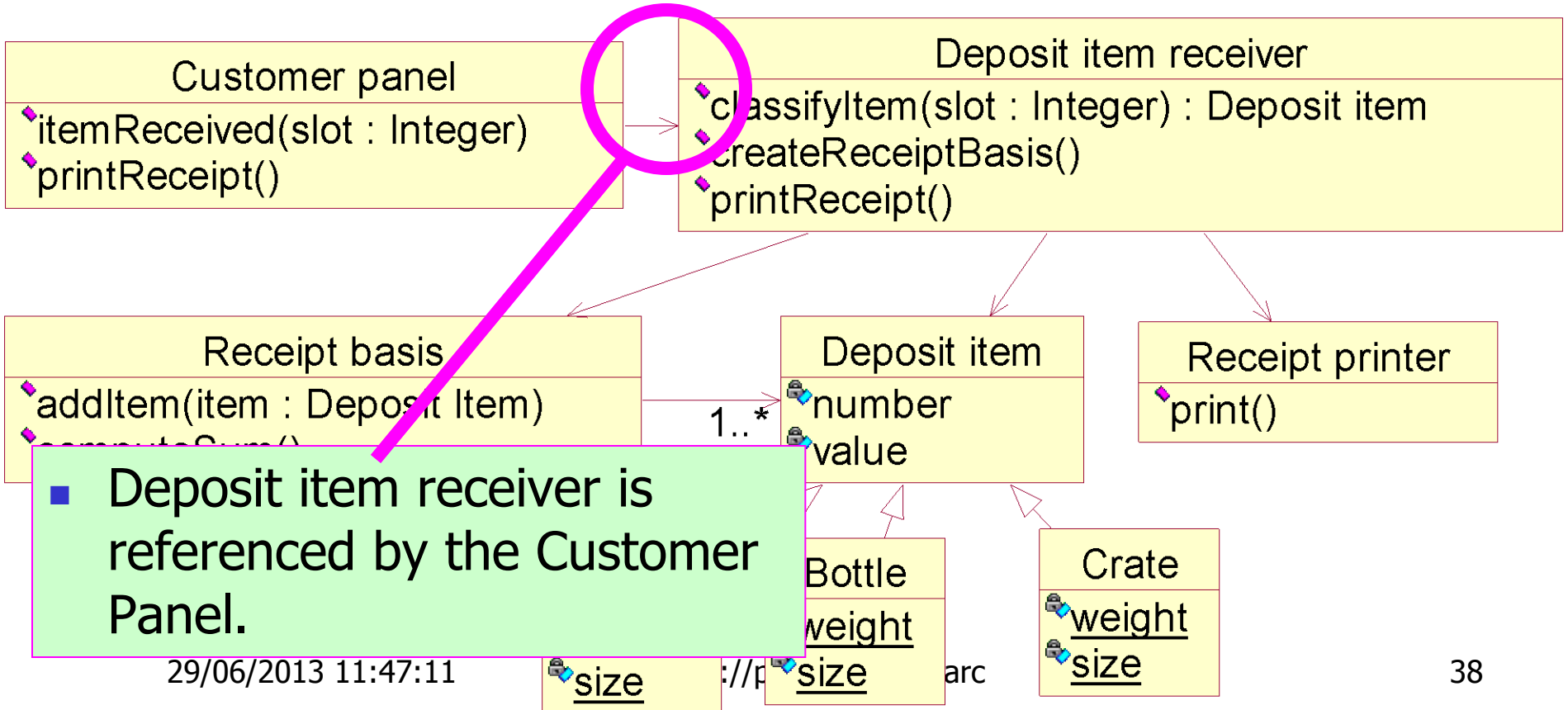
- There are four types of visibility:
  - *Attribute visibility* - B is a (reference) attribute to A.
  - *Parameter visibility* – B is a parameter of a method of A.
  - *Locally declared visibility* – B is declared as a local object in a method of A.
  - *Global visibility* – B is in some way globally visible.



Attribute Visibility

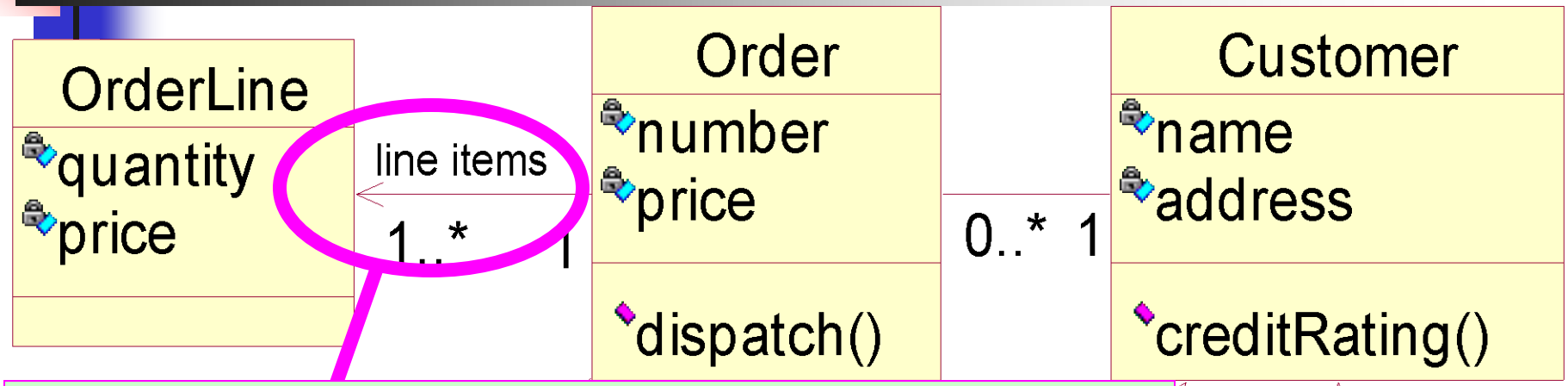
- *Attribute visibility* from A to B exists when B is a (reference) attribute of A.
- It persists as long as A and B exist.
- It is a very common form of visibility in object-oriented systems.
- In the implementation usually A has a reference (Java) or a pointer (C++) variable of B.

Attribute Visibility – Example



- Deposit item receiver is referenced by the Customer Panel.

Attribute visibility – Example 2

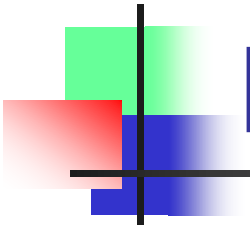


- The role name already suggests a name for the reference in the implementation, e.g. (Java)
  - public class Order {

    OrderLine [] line_items;

    ...

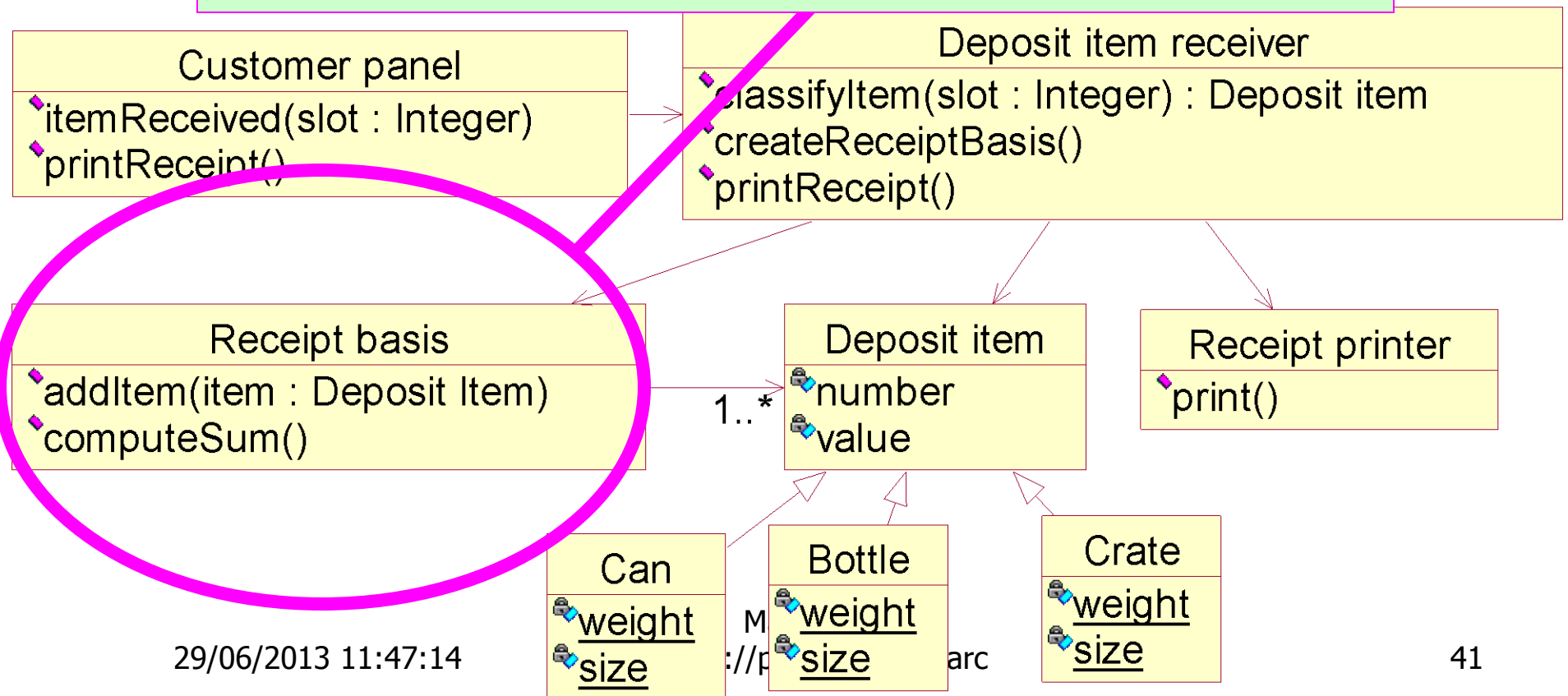
    }

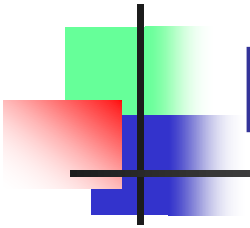


Parameter visibility

- Parameter visibility exists when B is passed as a parameter to a method of A.
- It is a relatively temporary visibility because it persists only in the scope of the method.
- It is common to transform parameter visibility into attribute visibility (see example).

- Parameter visibility (example):
  - Deposit item is passed as a parameter in the addItem method of the Receipt basis.
  - The parameter *item* will then become an attribute of Receipt basis.



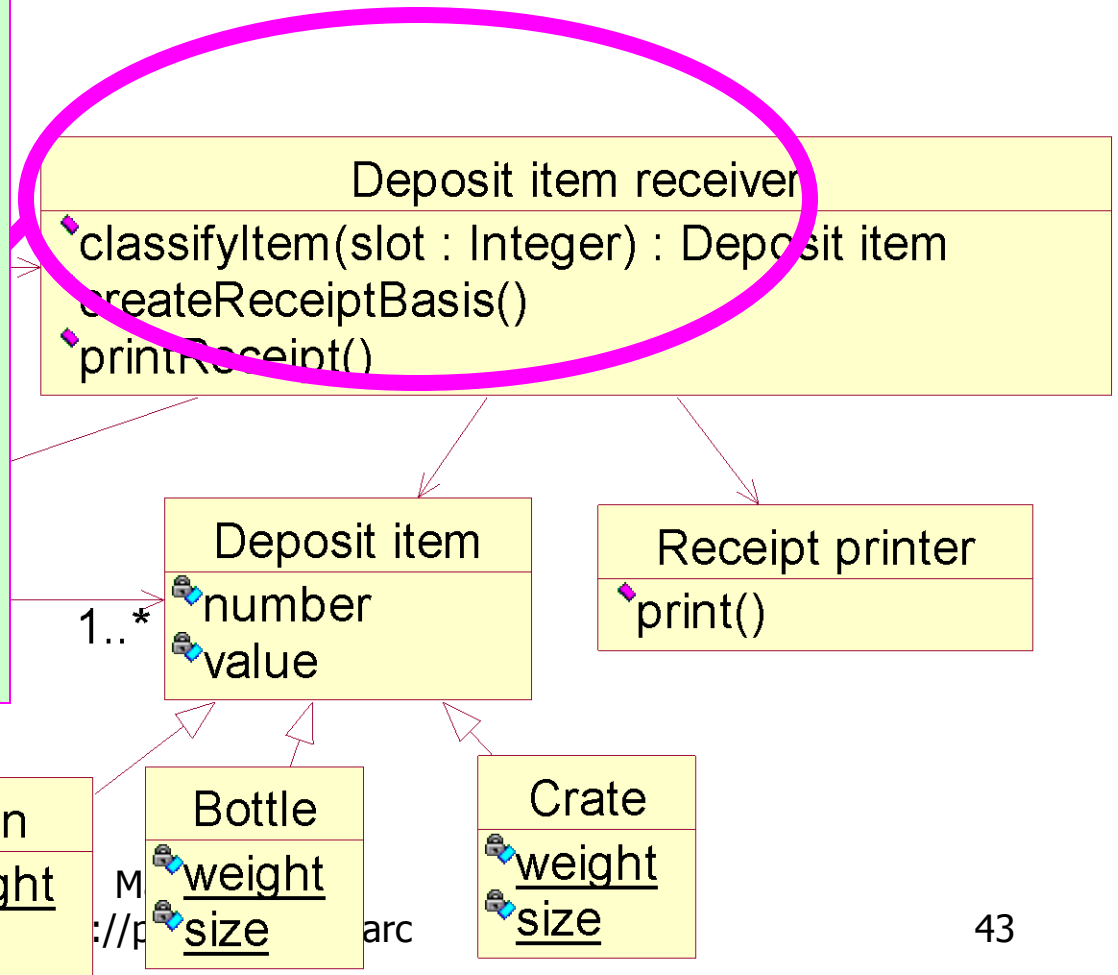


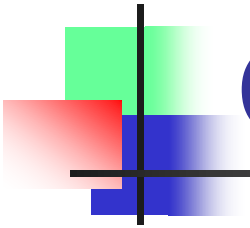
Locally Declared Visibility

- Locally declared visibility from A to B exists when B is declared as a local object within a method of A.
- Two common means:
  - Create a new local instance and assign it to a local variable.
  - Assign the return object from a method invocation to a local variable.

Locally Declared Visibility – Example

- The classifyItem() method generates an instance of Deposit item (Can, Bottle or Crate, depended of the slot) and returns it.
- In this method the Deposit item is locally visible.



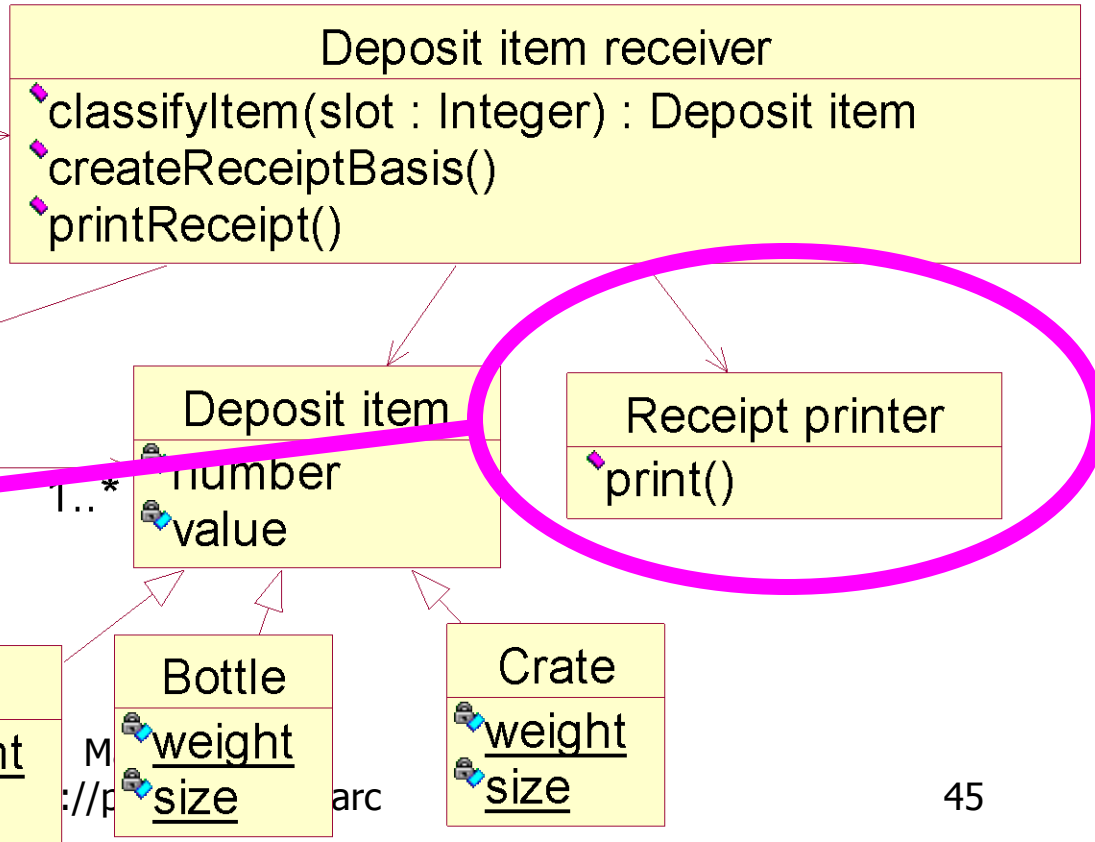


Global Visibility

- Global visibility from A to B exists when B is global to A. In object oriented systems it is the least common form of visibility.
- Global visibility can be implemented via
  - the return value of a class (static) method.
  - the return value of a non-member function (C++).
  - as a public static attribute in Java.

Global Visibility – Example

- As the printer is unique in the system and may be used also by other classes than Deposit item receiver (e.g. in the daily report use case) we can design it as a global object.

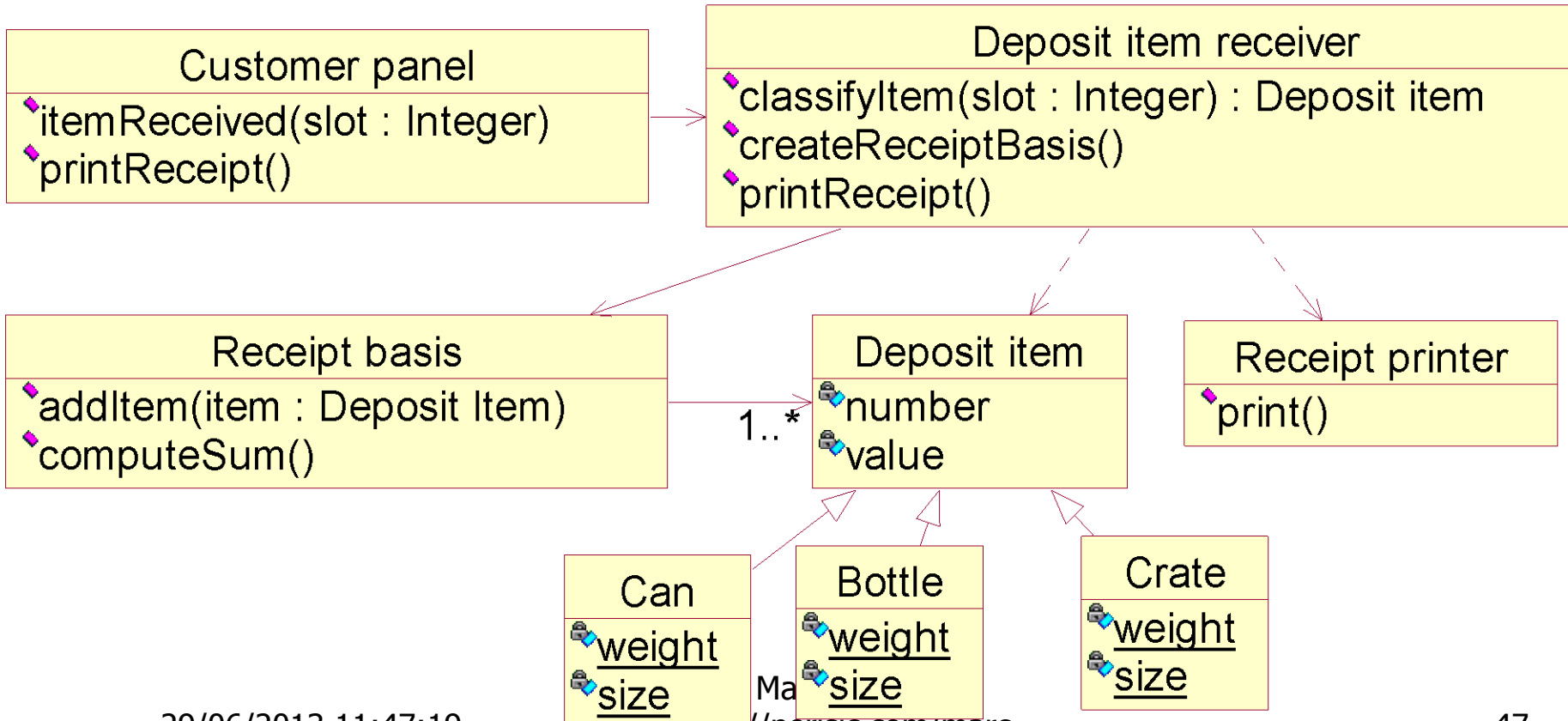


Visibility, Association & Dependency

- Attribute visibility between classes is always considered as an *association*. UML uses a solid arrow to denote associations:

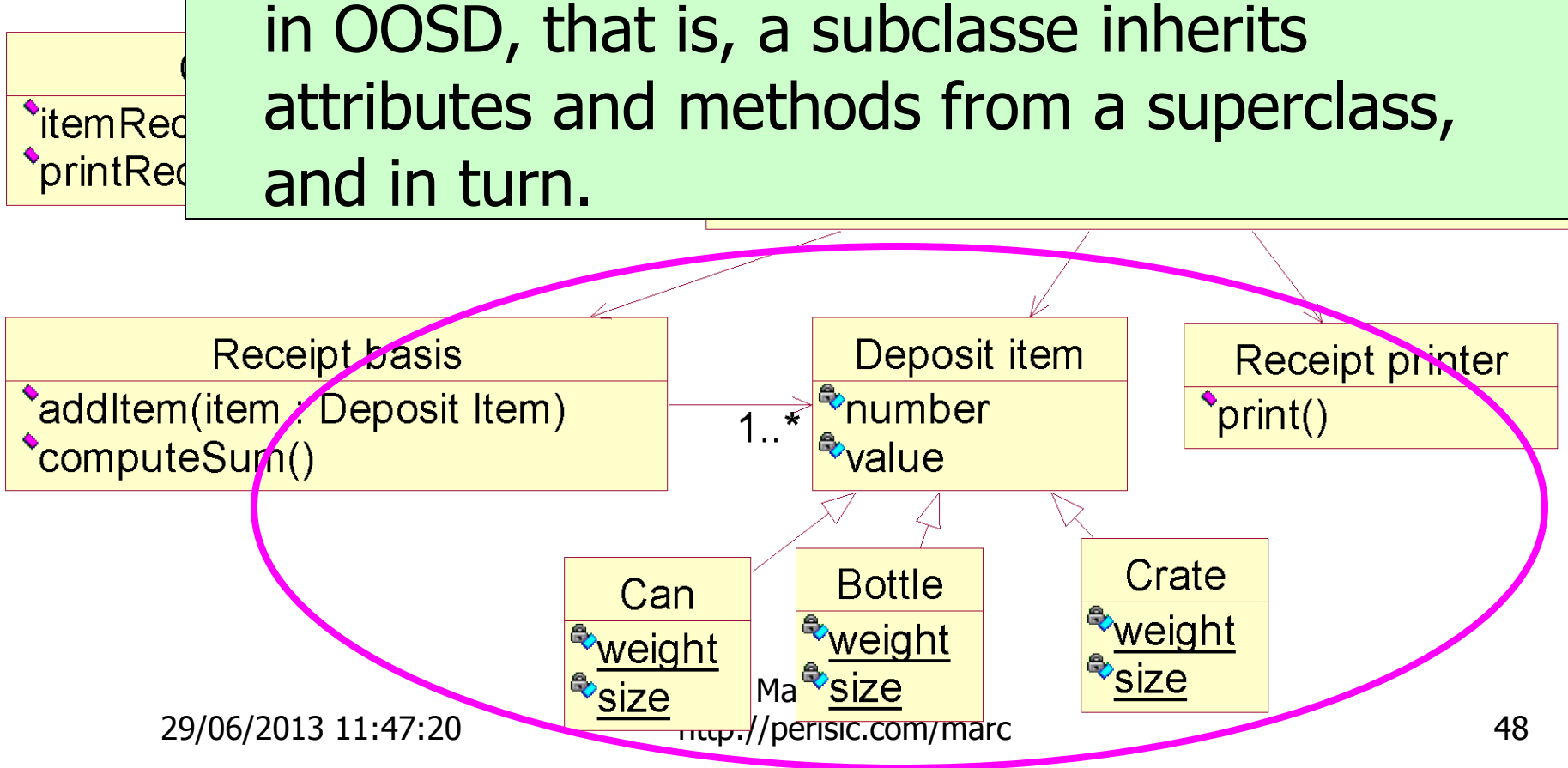
- Parameter, local, and global visibility is considered as a *dependency.* UML uses as dashed arrow for dependencies:

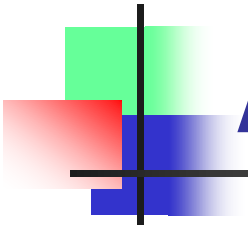
Revised example:



Generalisation

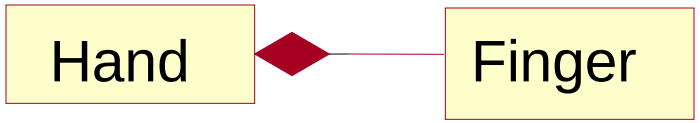
- Generalisation -- used to refer to inheritance in OOSD, that is, a subclasse inherits attributes and methods from a superclass, and in turn.



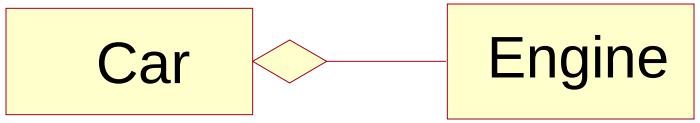


Aggregation and Composition

- *Aggregation* is a kind of association used to model whole-part relationships between things – A "has a" relationship. The whole is generally called the *composite* (the parts have no standard name)
- Aggregation is shown with a hollow or filled diamond:
  - Composite Aggregation:



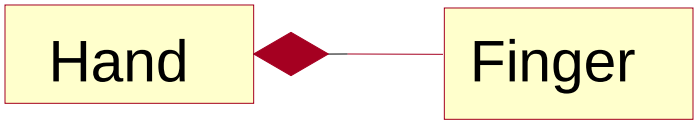
  - Shared Aggregation:



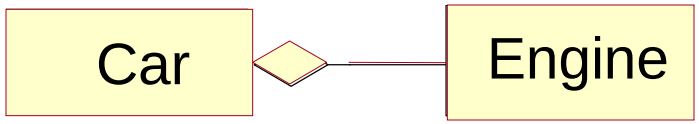
- Aggregation is a property of an association role (as multiplicity, name, multiplicity)

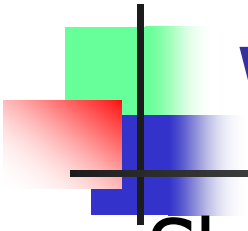
Composite Aggregation vs. Shared Aggregation

- Composite aggregation (also known as composition) means that the composite solely owns the part.



- Shared aggregation means that the part may be in many composite instances.





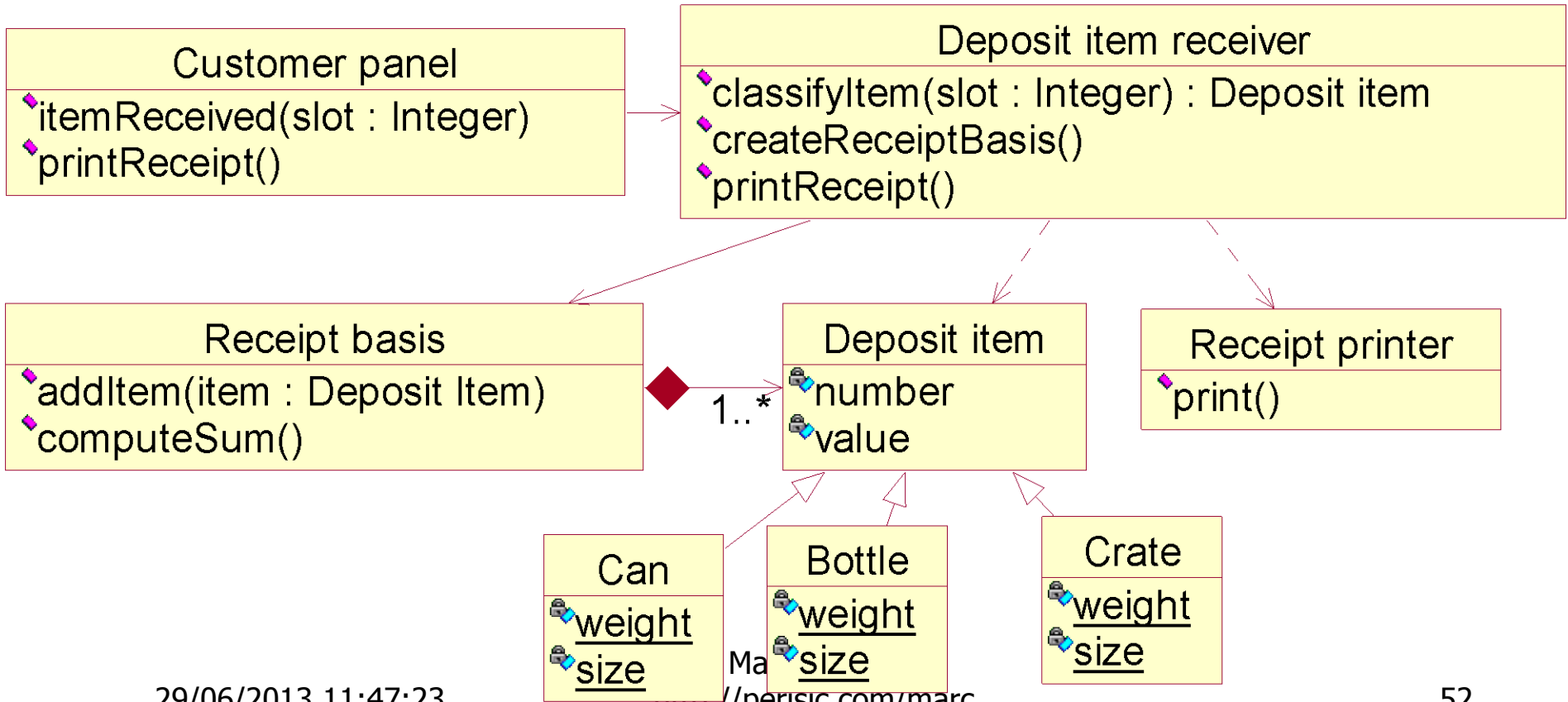
When to show aggregation?

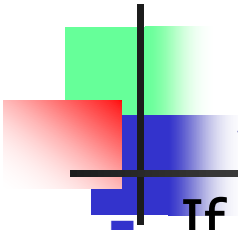
- Show aggregation when:
  - The lifetime of the part is bound within the lifetime of the composite.
  - There is an obvious whole-part physical or logical assembly.
  - Some properties of the composite propagate to the parts.
  - Operations applied to the composite propagate to the parts.

Rule of thumb: If in doubt, leave it out.

Aggregation (Example)

- The Deposit item may be considered as part of a composite Receipt basis.





Abstract classes & Interfaces.

- If every member of a type T must also be a member of a subtype, then type T is called an *abstract type,* and the type name is *italized* in the class diagram
- If an abstract type is implemented in software as a class during the design phase, it will usually be represented by an *abstract class,* meaning that no instances may be created for the class.
- An *abstract method* is one that is declared in an abstract class, but not implemented; in the UML it is also notated with italics.
- Classes containing only abstract methods are known as *interfaces* (denoted by a dotted generalisation arrow).

Abstract classes

- Deposit item may be considered as an abstract class as it only exists as a Can, Bottle, or Crate. Therefore Deposit item is *italized.*