Requirements Analysis and system design-Developing Information Systems with UML

Article · January 2001

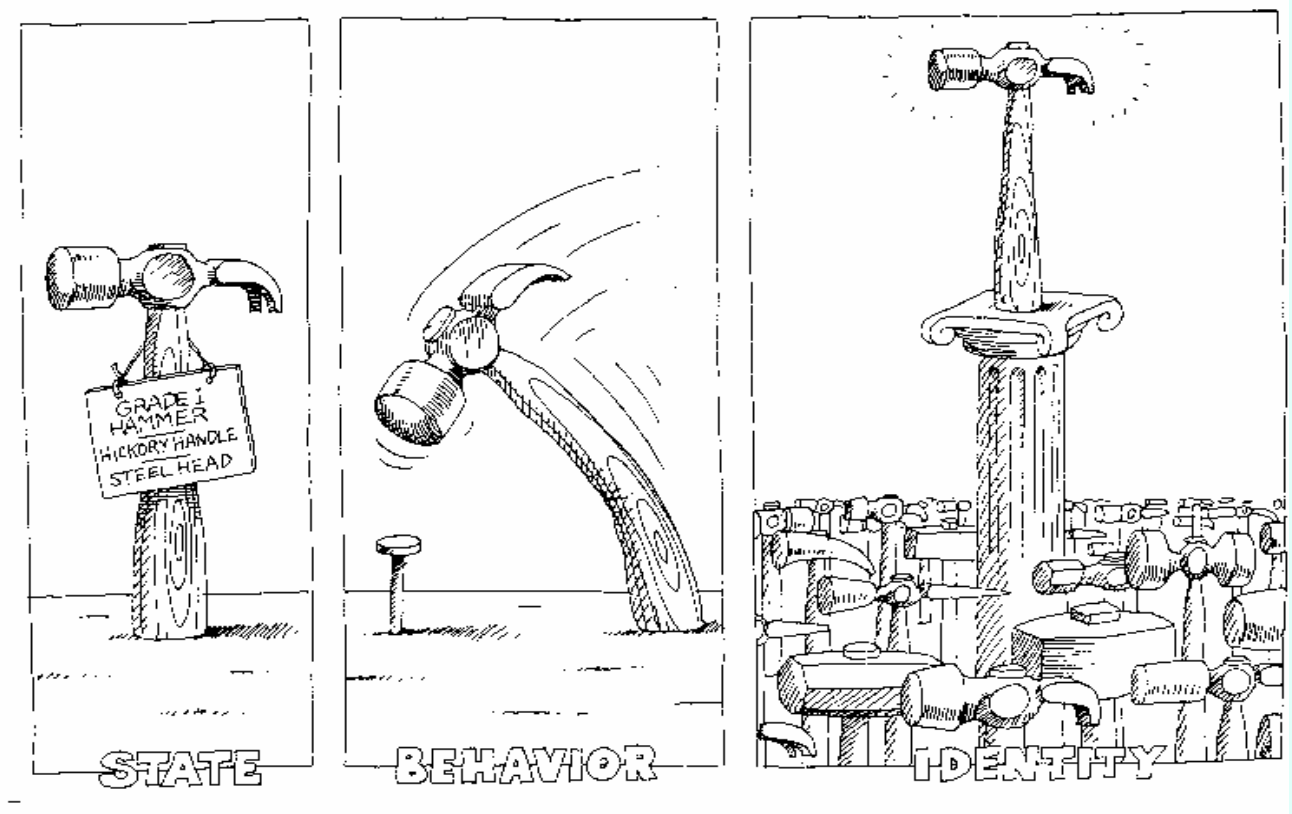
MACIASZEK, L.A. (2007):
Requirements Analysis and System Design, 3rd ed.
Addison Wesley, Harlow England
ISBN 978-0-321-44036-5

Appendix
Fundamentals of Object Technology

Topics

- Instance object
- Class
- Variables, methods, and constructors
- Association
- Aggregation and composition
- Generalization and inheritance
- Abstract class
- Interface

Fundamentals of OT

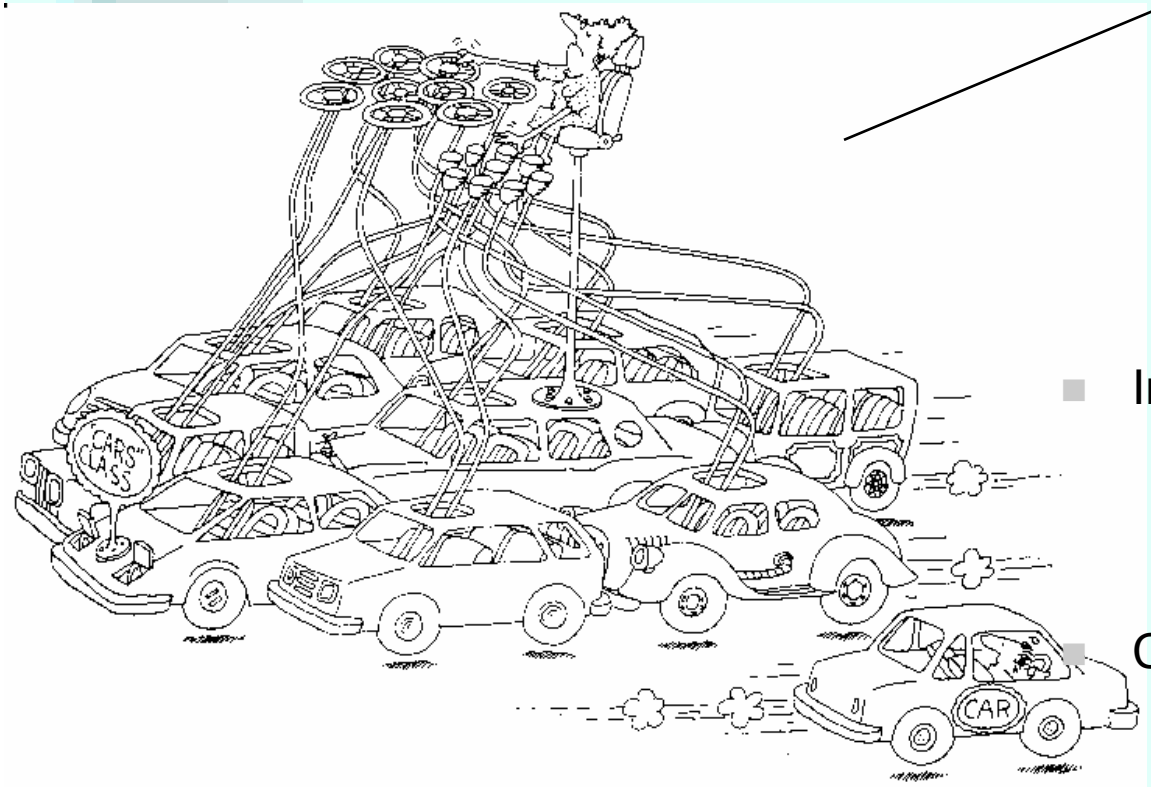


- Object has
  - State
  - Behavior
  - Identity

    (equal ≠ identical)
- Objects and natural systems

BOOCH, G. (1994): *Object Oriented Analysis and Design with Applications*, 2nd ed, The Benjamin/Cummings Publ

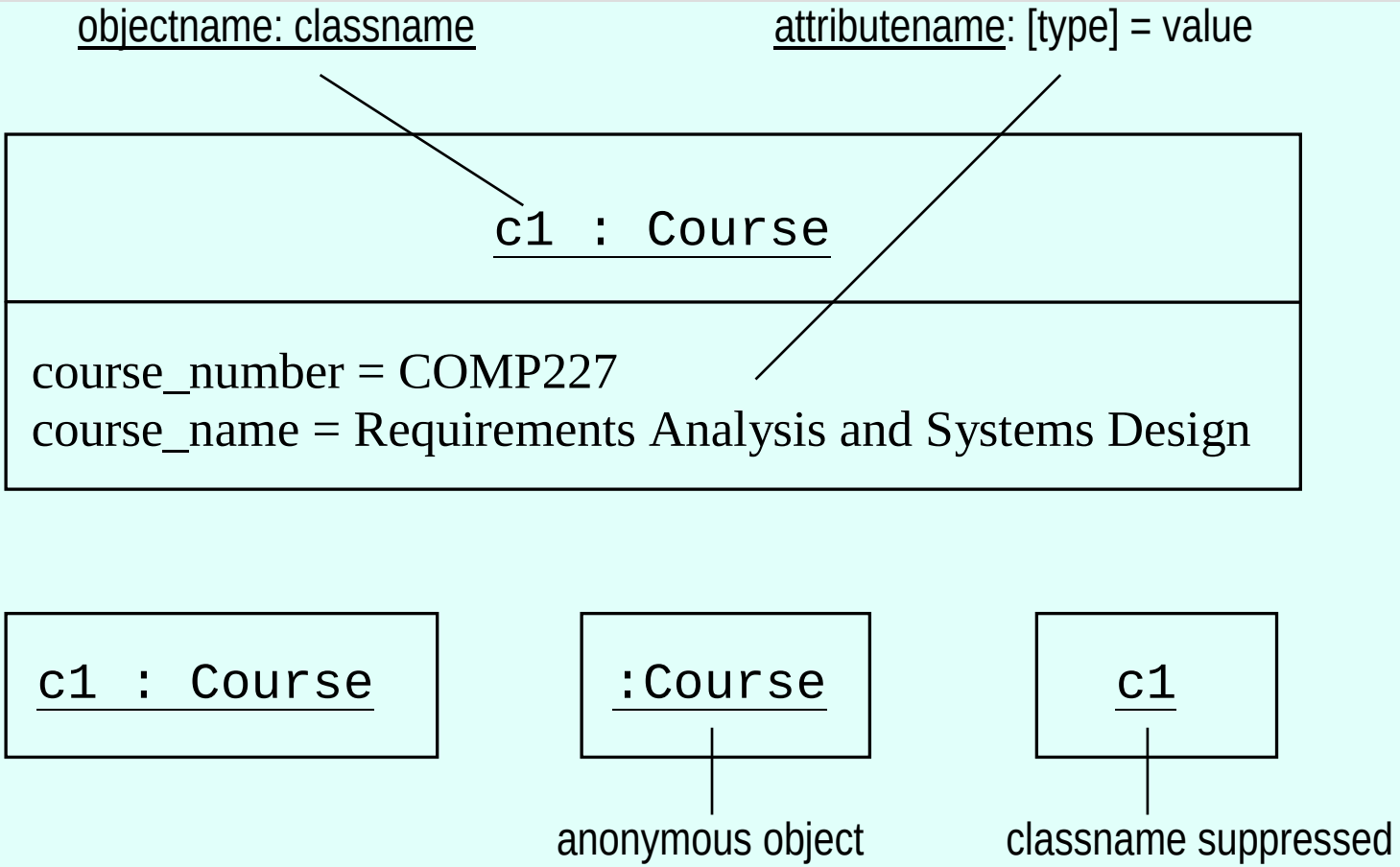
Instance object



BOOCH, G. (1994): *Object Oriented Analysis and Design with Applications*, 2nd ed, The Benjamin/Cummings Publ

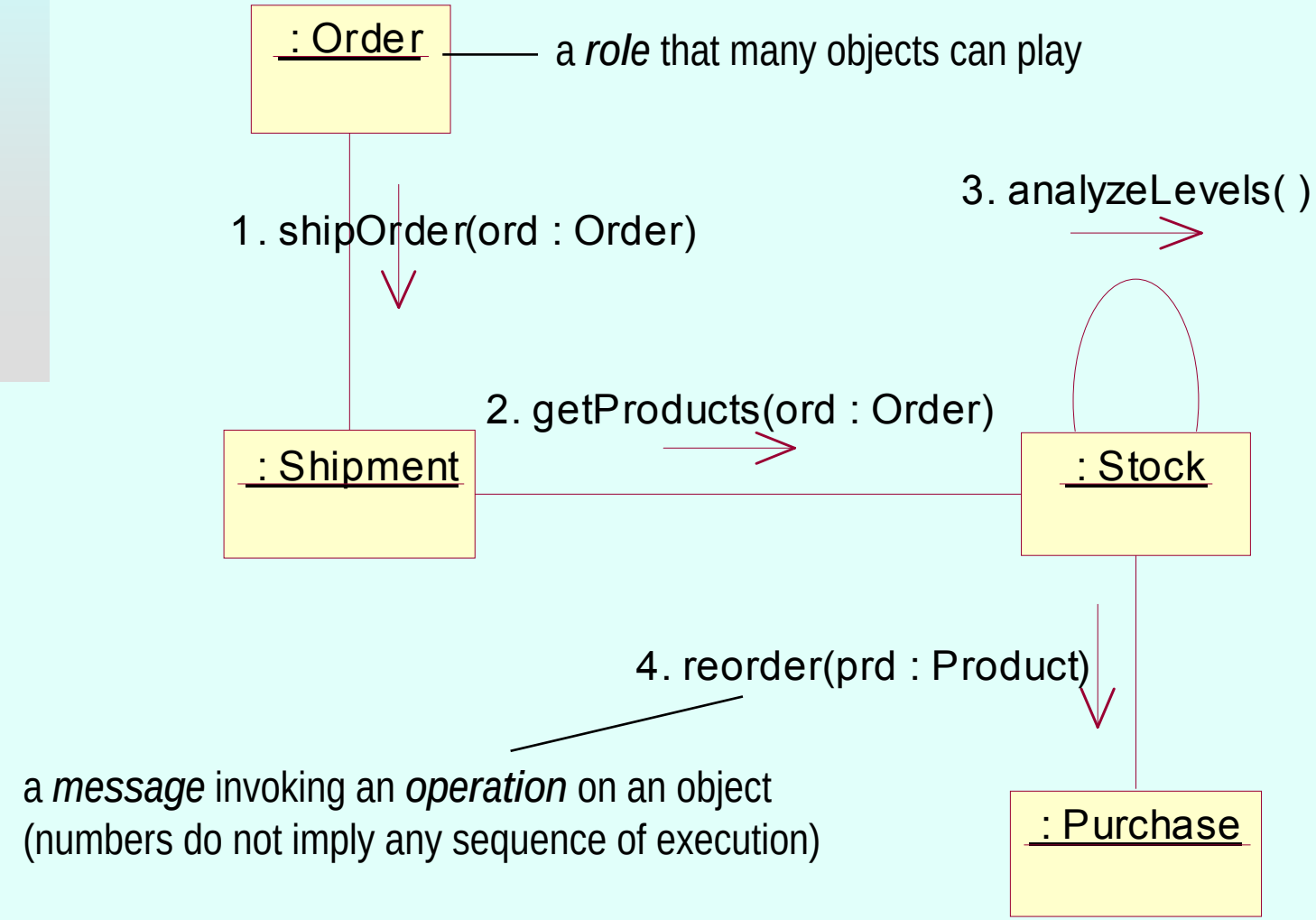
- Class
  - an abstraction (generic description) for a set of objects
  - a class for objects (but better to avoid the term “object class” because a class can be instantiated into an object – “object class object’ would sound a bit strange)
- Instance object
  - an instance of a class
  - object, instance, but not object instance
- Class object
  - an instantiated class (everything in an object oriented system is an object)

Object notation



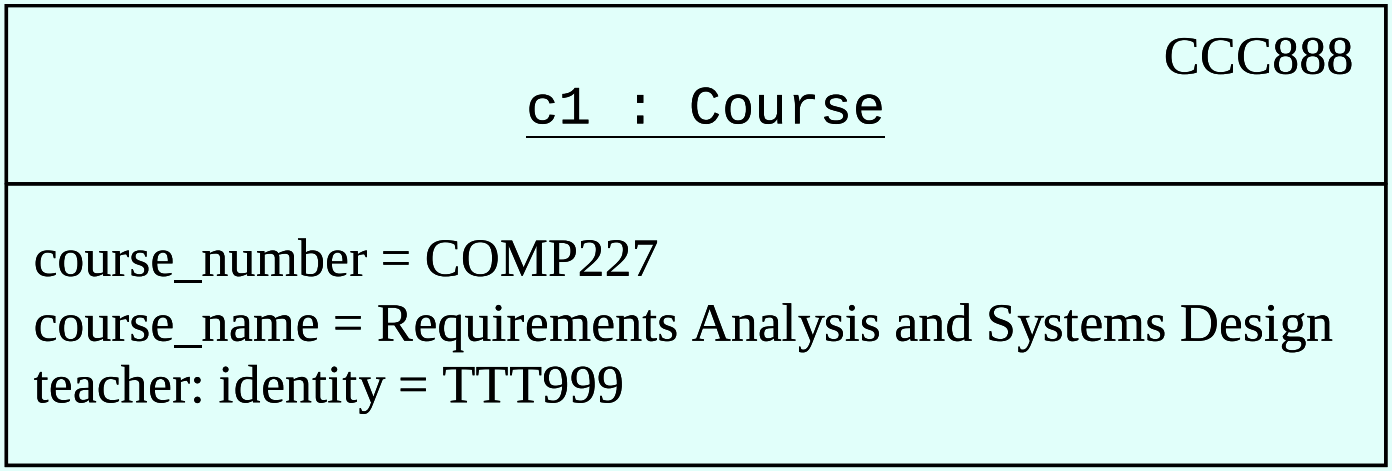
No compartment for operations in objects!
Exception – prototypical (delegation-based) languages, such as Self or Newton

How do objects collaborate?

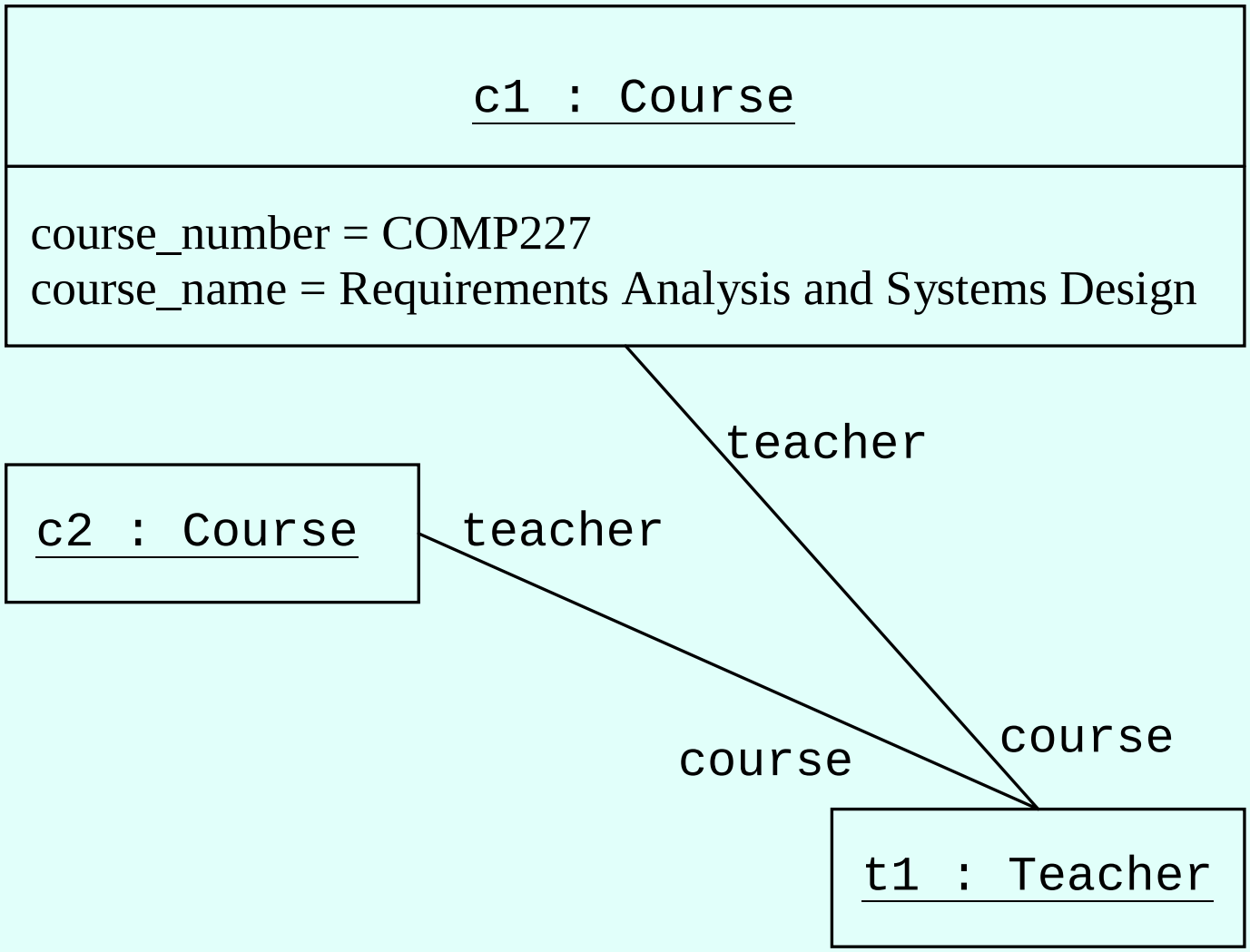


How objects identify each other?

- object identifier - OID
- Object longevity
  - Persistent object
    - outlives the execution of the program
    - swizzling to convert persistent OID (disk address) to transient OID (memory address)
  - Transient object



OIDs to implement associations



Transient link

- How does an object know the OID of another object if there is no persistent link?
  - Previous access to an object still “memorized” in some program variable
  - Search on the database
  - A “map” object that associates objects to other objects by logical identifiers (primary keys) or similar means
  - Creating a new object

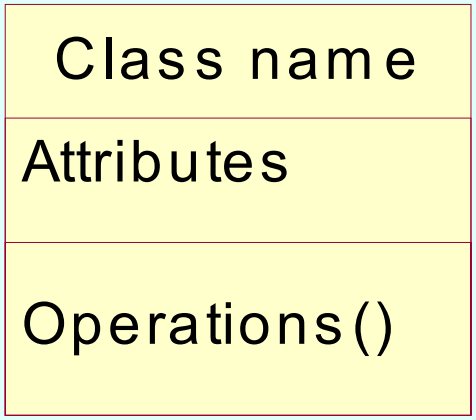
Message passing

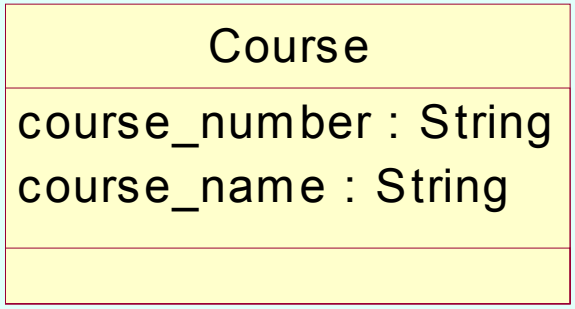
```
crs_ref.getCourseName(out crs_name)
```

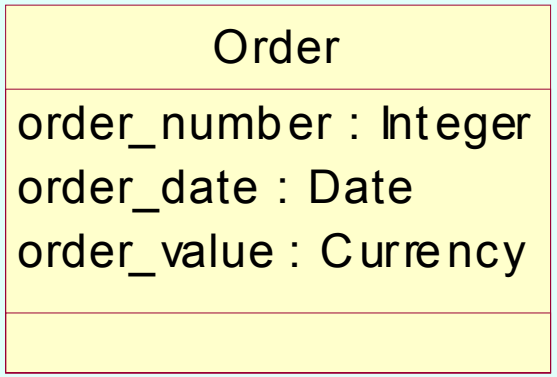
- in, out, and in/out arguments
  - Most OO languages do not make such distinction explicit
  - In Java:
    - message arguments of primitive data types are passed by value
      - act as **input arguments**
      - the change of argument values not possible because the operation acts on a copy of the argument
    - "pass by value" for non-primitive data types results in the operation receiving the reference of the argument, not its value
      - the reference can be used to access and possibly modify the attribute values within the passed object
      - the above eliminates the need for explicit **in/out arguments**
    - return type of an operation substitutes a need for explicit **out arguments**
      - operation can return only one return value or no value at all (void)
      - however, it can return non-primitive type (i.e. an object of any complexity)

Class, attributes

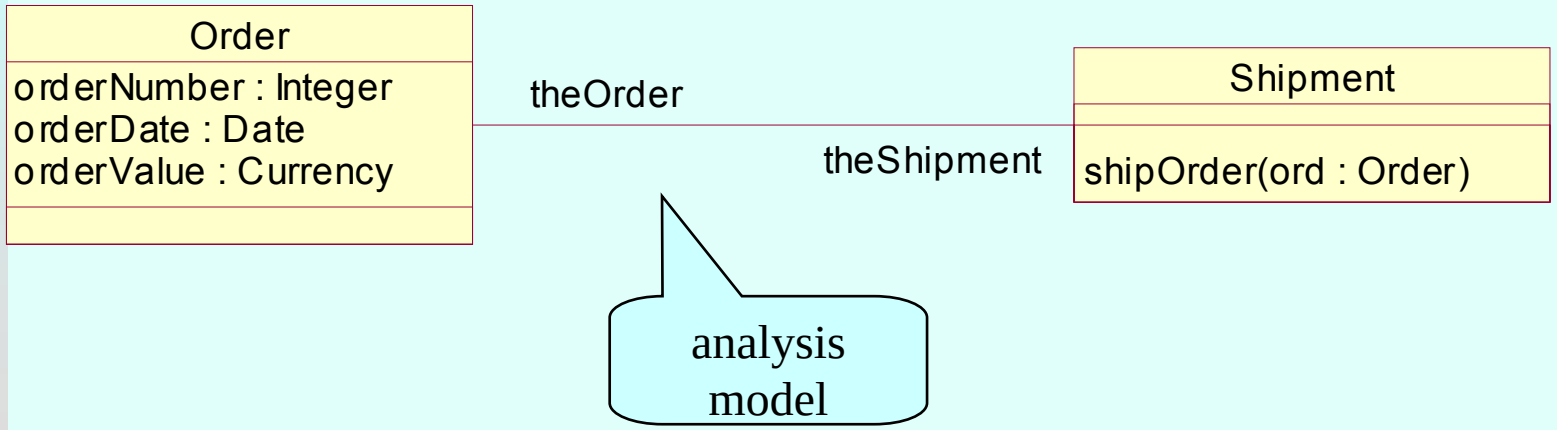
Class – an “overloaded” term!

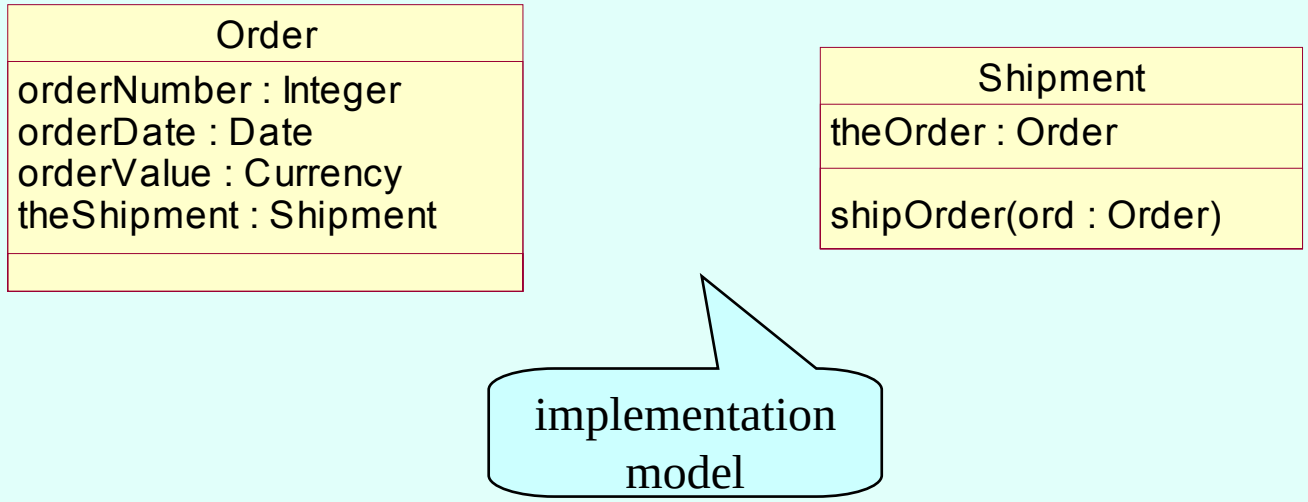






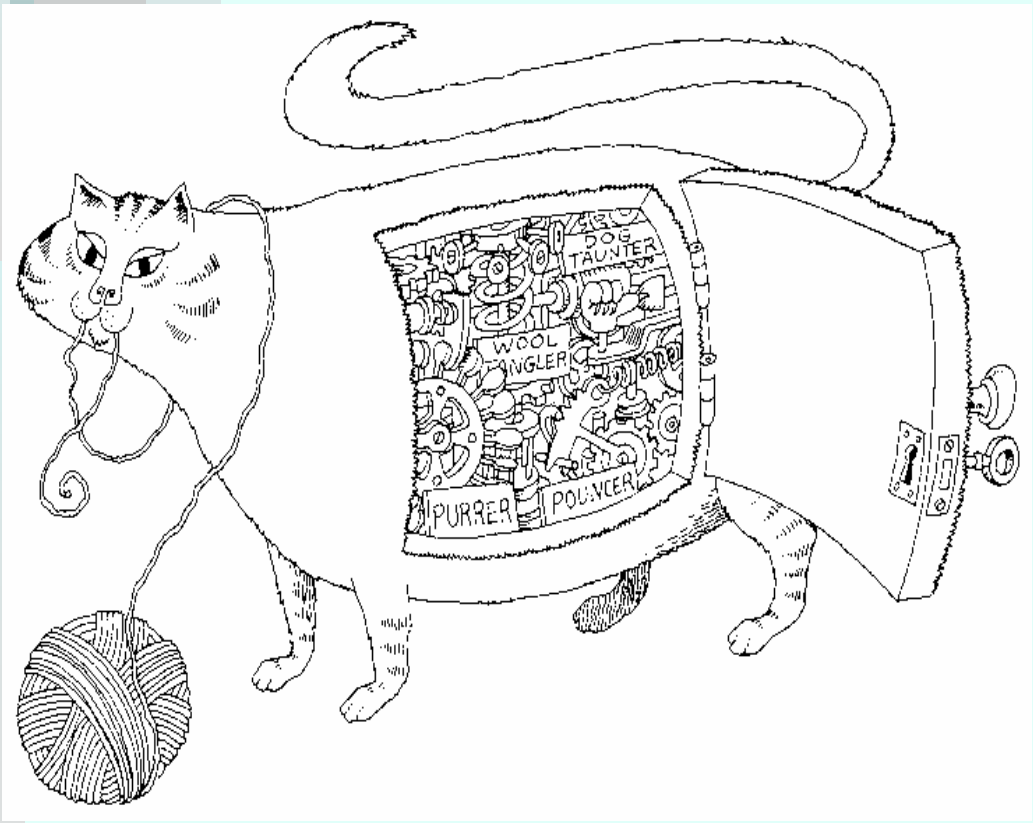
Role names / attributes designating a class





Attribute visibility

- private attributes and public operations
- operations **encapsulate** attributes

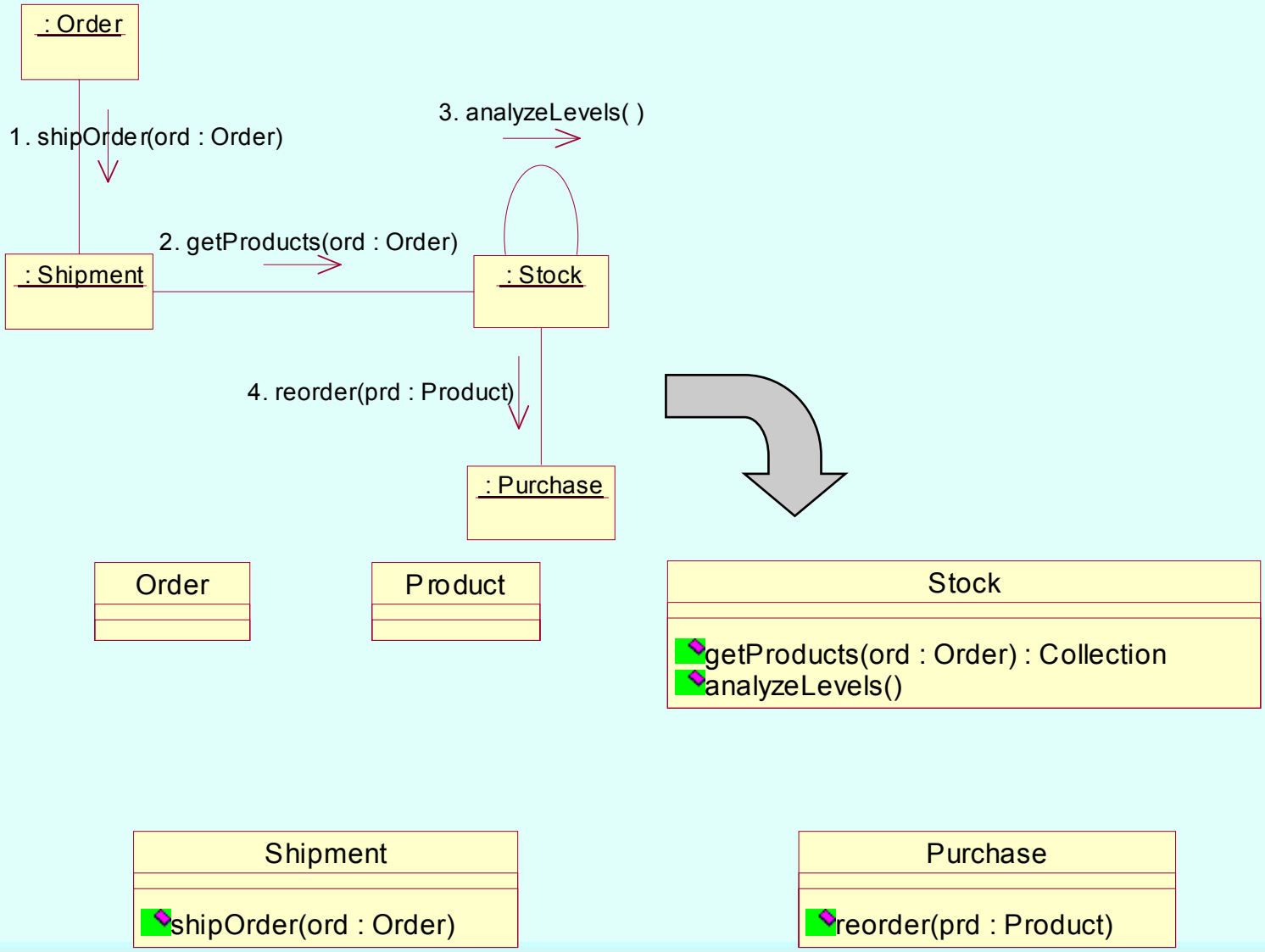


Purchase
- purchaseNumber : String - purchaseDate : Date - purchaseValue : Currency
+ reorder(prd : Product)

Purchase
purchaseNumber : String purchaseDate : Date purchaseValue : Currency
reorder(prd : Product)

BOOCH, G. (1994): *Object Oriented Analysis and Design with Applications*, 2nd ed, The Benjamin/Cummings Publ

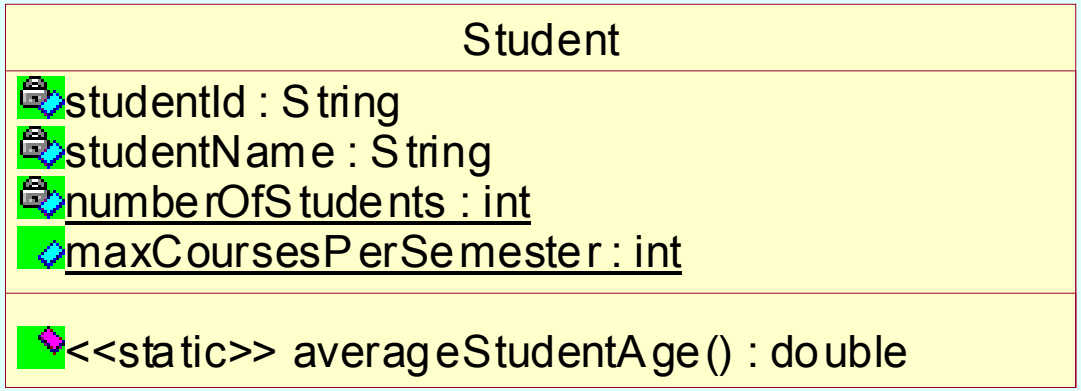
Operations in object collaboration



Operation, visibility, scope, class object

- **method** – procedure that implements an **operation**
- message name = method name
- **signature** = list of formal arguments of a method
- visibility ≠ scope
  - **instance scope** when operation invoked on instance object (findEmpAge())
  - **class scope** when operation invoked on class object (findAverageAge())
- **Class object**
  - most prog. lang. do not instantiate class object
  - they only provide a syntax to refer to the class name in order to access a class-scope attribute or call a class-scope operation → **static** attribute/operation
- Example → next slide

Class-scope attributes and operations - example



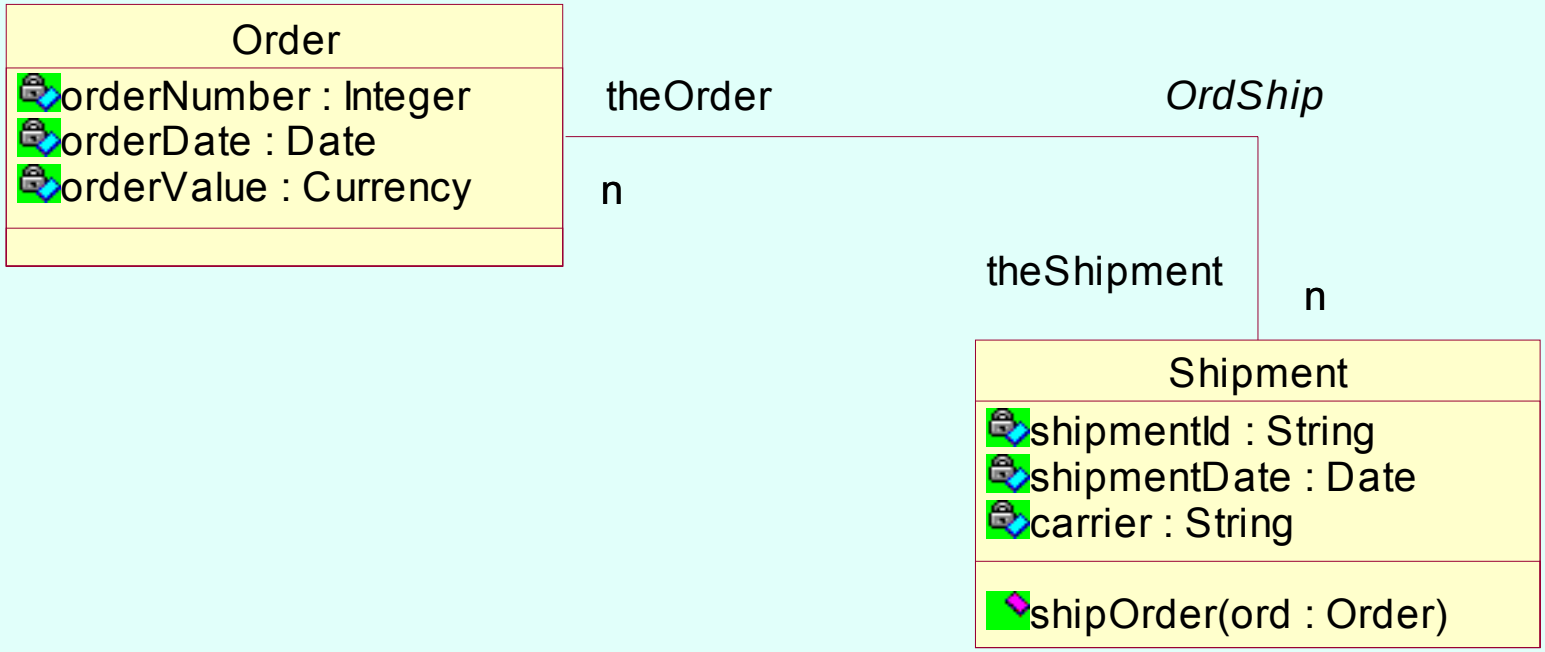

```
public class Student
{
   private String studentId;   //accessible via Student's operations
   private String studentName; //accessible via Student's operations
   private static int numberOfStudents;
   //accessible only to Student's static methods, such as averageStudentAge()
   public static int maxCoursesPerSemester;
   //accessible via Student:: maxCoursesPerSemester

   public static double averageStudentAge()
   { implementation code here }
   //callable by referring to the class name – Student::averageStudentAge()
   //callable also with an object of the class – std.averageStudentAge()
}
```

Variables, methods, constructors

- **Variable** – name for a storage space
  - **data member** – variable declared in a class
    - instance variable (instance scope)
    - class variable (class scope)
  - **local variable** – variable declared in a method body
- **Method** – implementation of an operation
  - method **prototype** = name + signature + return type
  - **overloaded** methods – same names, different signatures
- **Constructor**
  - special “method” to instantiate objects of the class
  - constructor name = class name
  - constructor has no return type
  - class must have at least one constructor
  - invoked with the new keyword

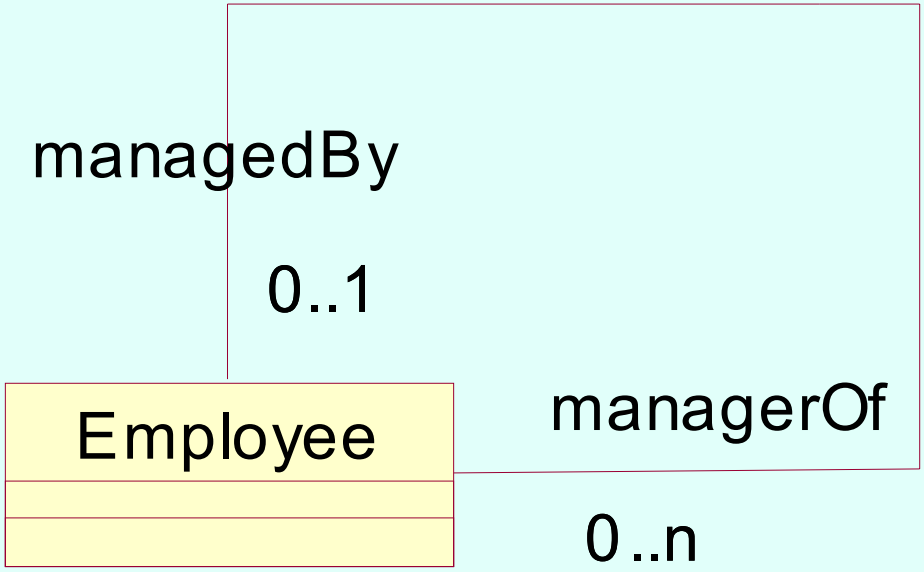
```
Student std22 = new Student(); //default constructor
```

Association



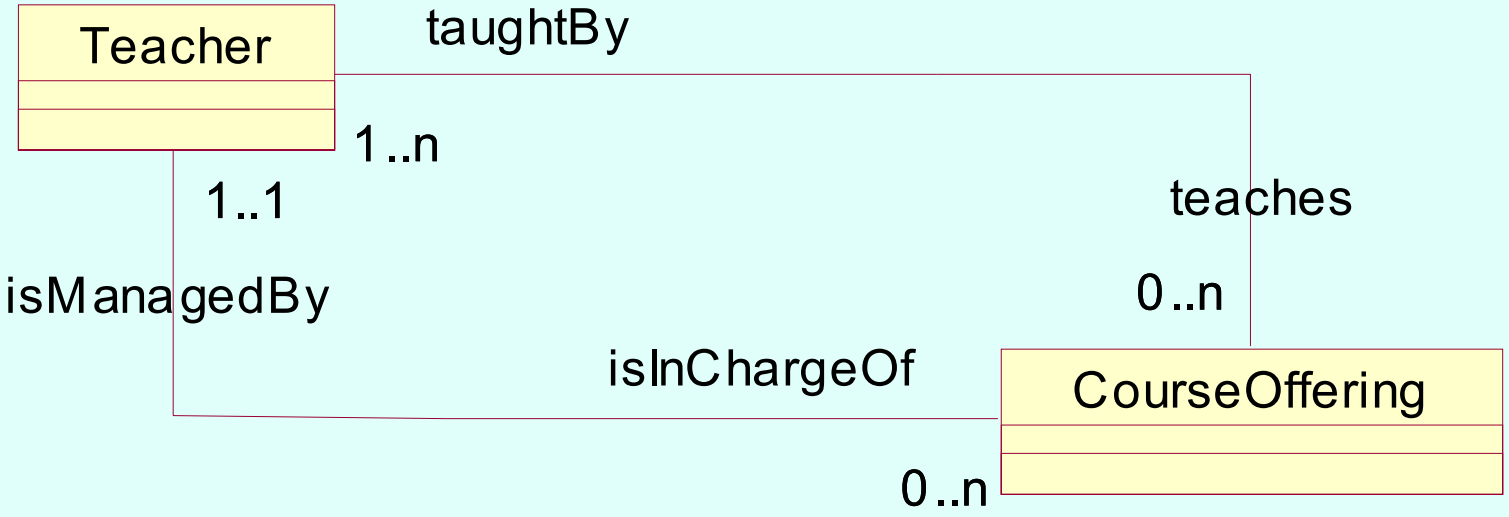
Association degree

- Binary
- Unary (singular)
- Ternary



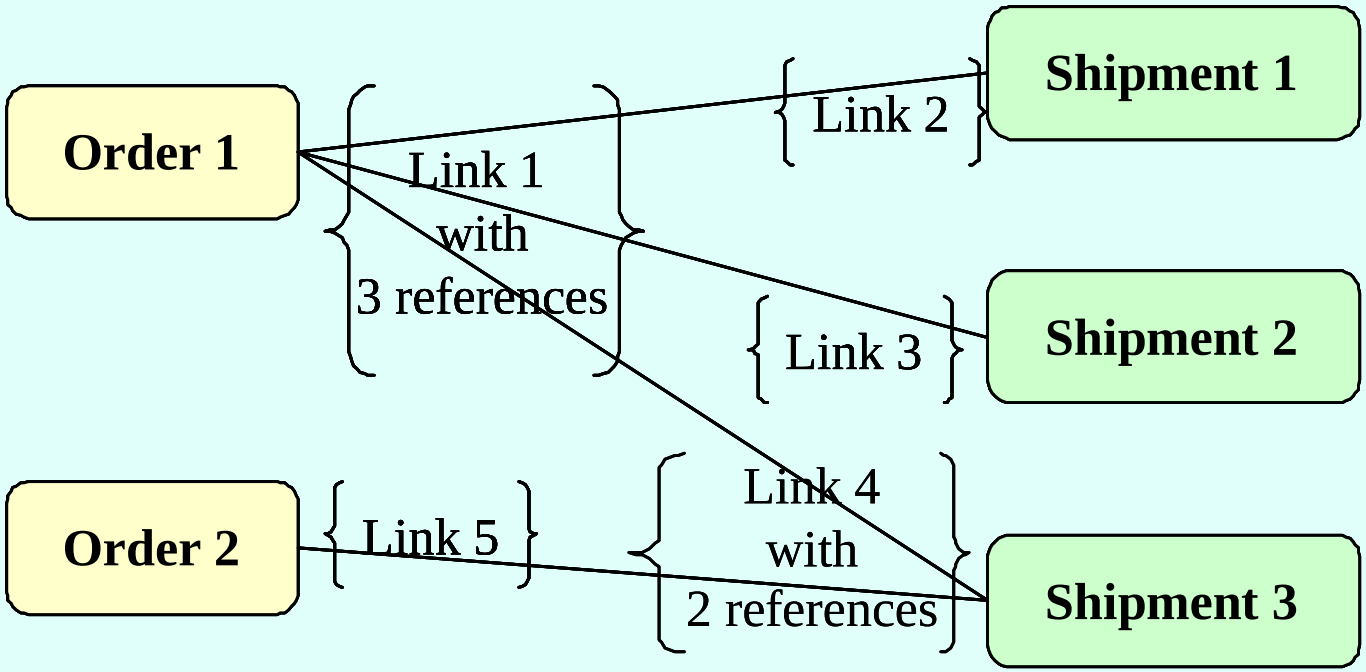
Association multiplicity

- 0..1
- 0..n
- 1..1
- 1..n
- n

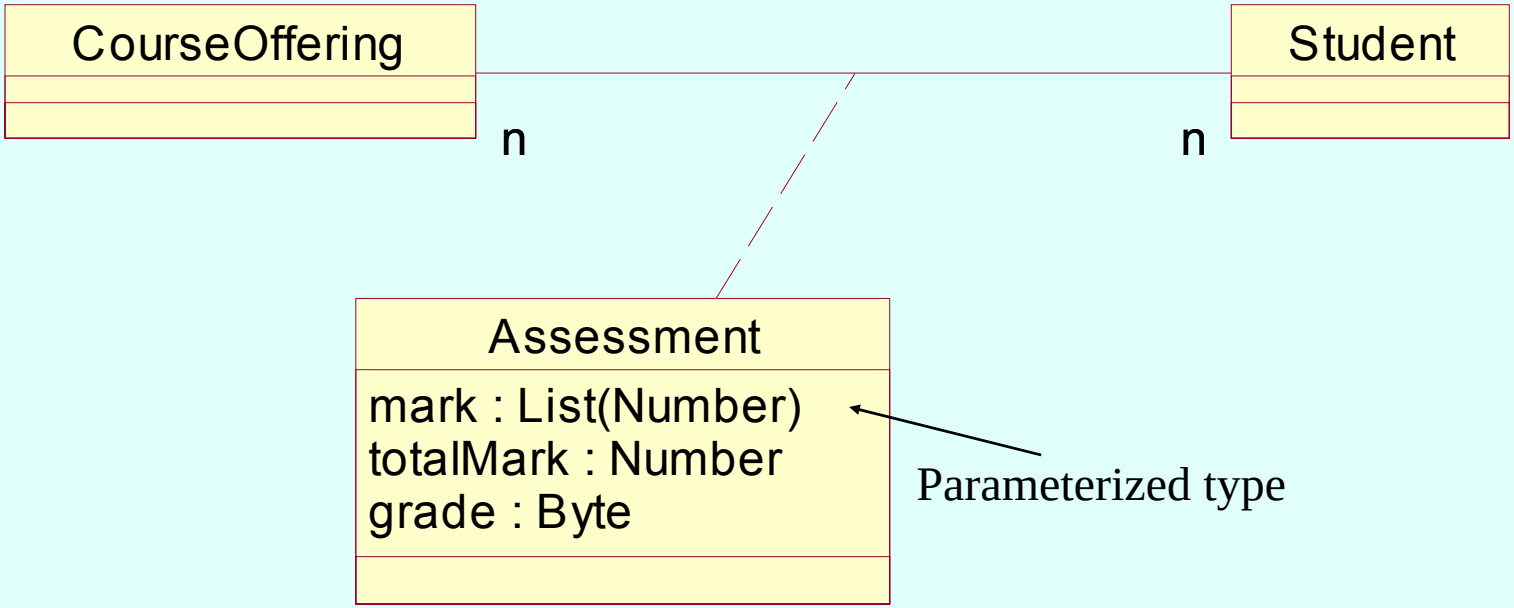


Association link and extent

- Link – association instance
  - represents a rolename
  - can be a collection of references
- Extent – set of association instances
  - in the figure, the extend of the association is five (five links)

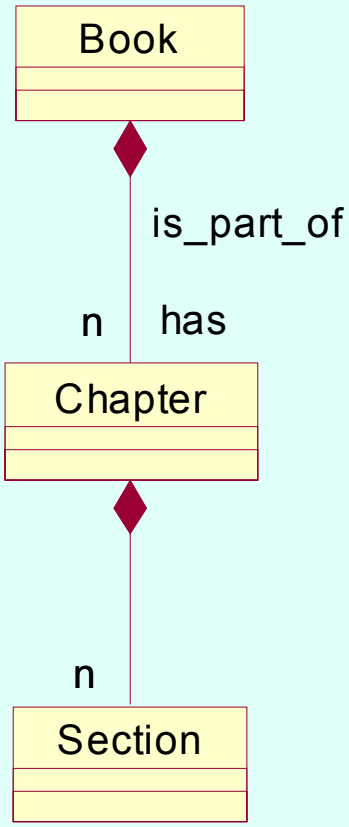


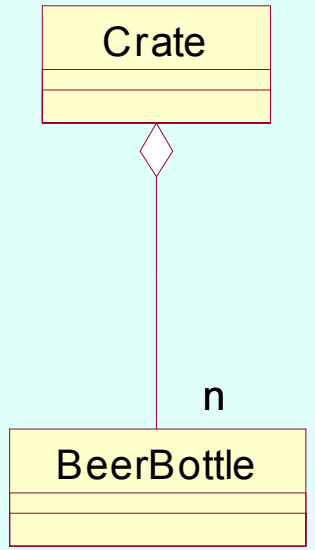
Association class



Composition and aggregation

- Composition – aggregation by value
- Aggregation – aggregation by reference
- Properties:
  - Transitivity
  - Asymmetry
  - Existence dependency
- Implemented by buried references or inner classes → next slides

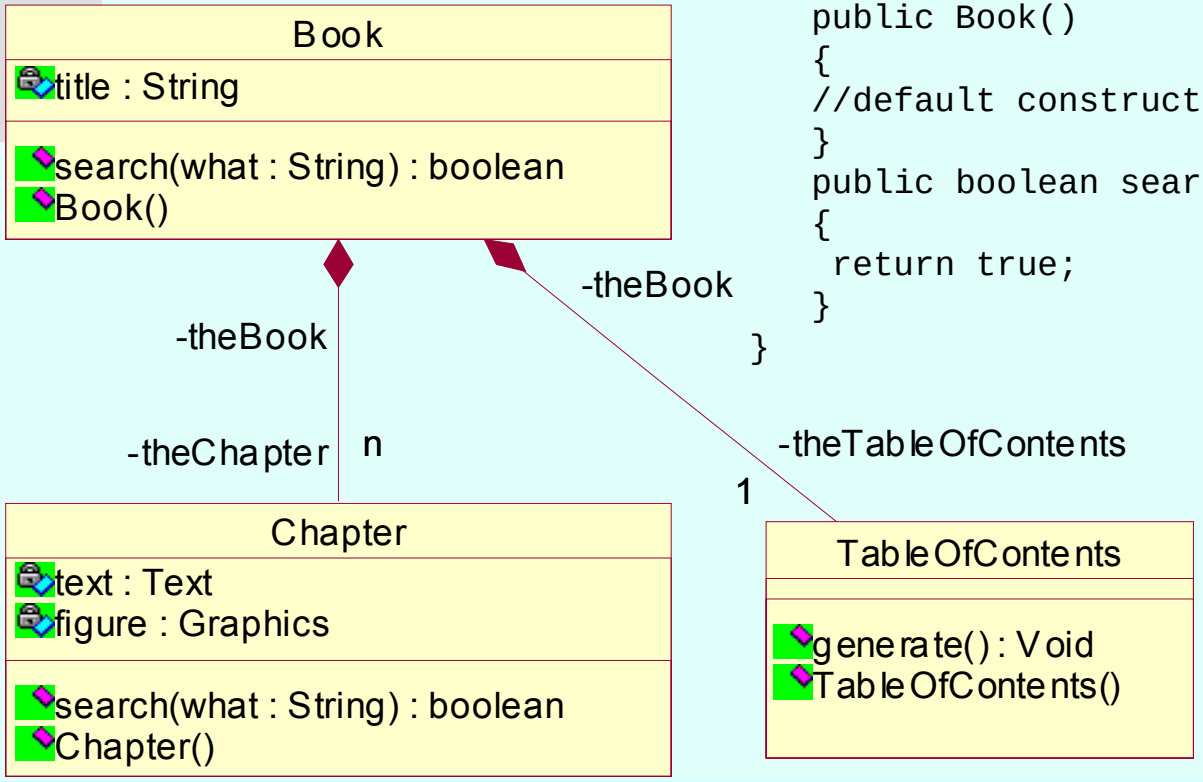




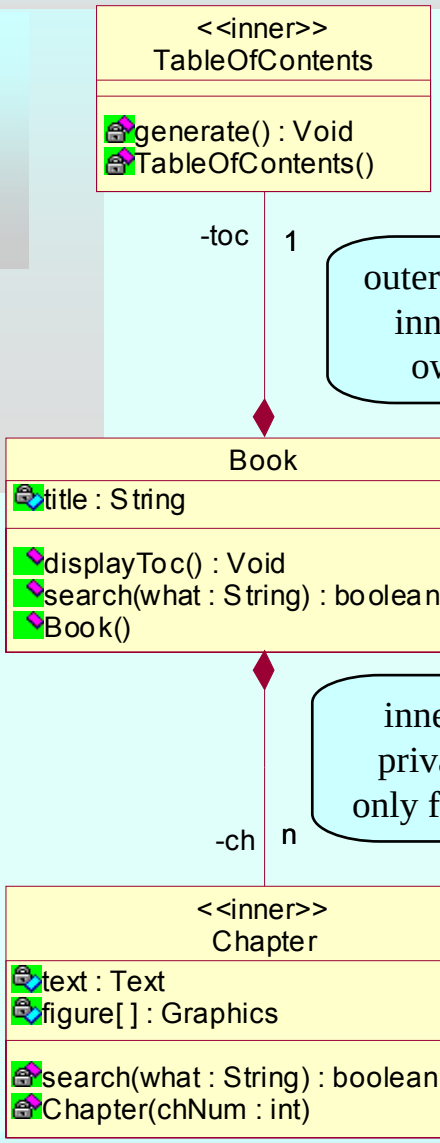
Implementing aggregation by buried reference

- *not different to association*
- *by means of variable with private visibility*
- *however, in Java, classes cannot have private visibility*
  - *subset classes must have public or package visibility*

```
public class Book
{
    private String title;
    private Chapter theChapter[];
    private TableOfContents theTableOfContents;
    public Book()
    {
    //default constructor
    }
    public boolean search(String what)
    {
     return true;
    }
}
```



Implementing aggregation by inner classes



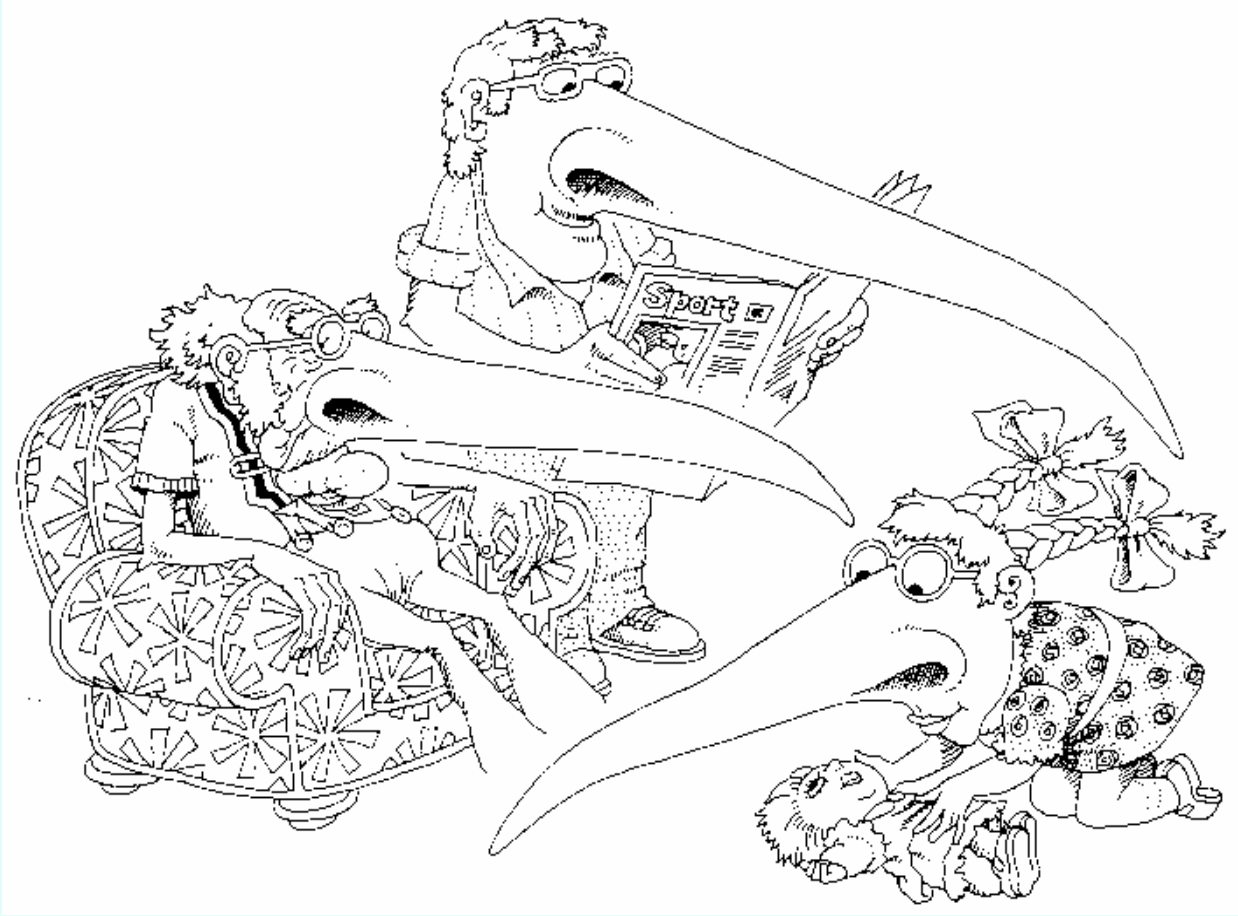
outer class instantiates inner objects in its own constructor

inner class can be private (accessible only from outer class)

```
public class Book
{
   private String title;
   private Chapter[] ch;
   private TableOfContents toc;
   public Book(...)
   { ...
      toc = new TableOfContents();
      ch = new Chapter[numberChapters];
      for (int i=0; i<numberChapters; i++)
         ch[i] = new Chapter();
   }
   public Void displayToc()
   {
      toc.generate();
      return;
   }
private class TableOfContents
   {
      private Void generate()
      {...
      }
   }
}
```

Generalization

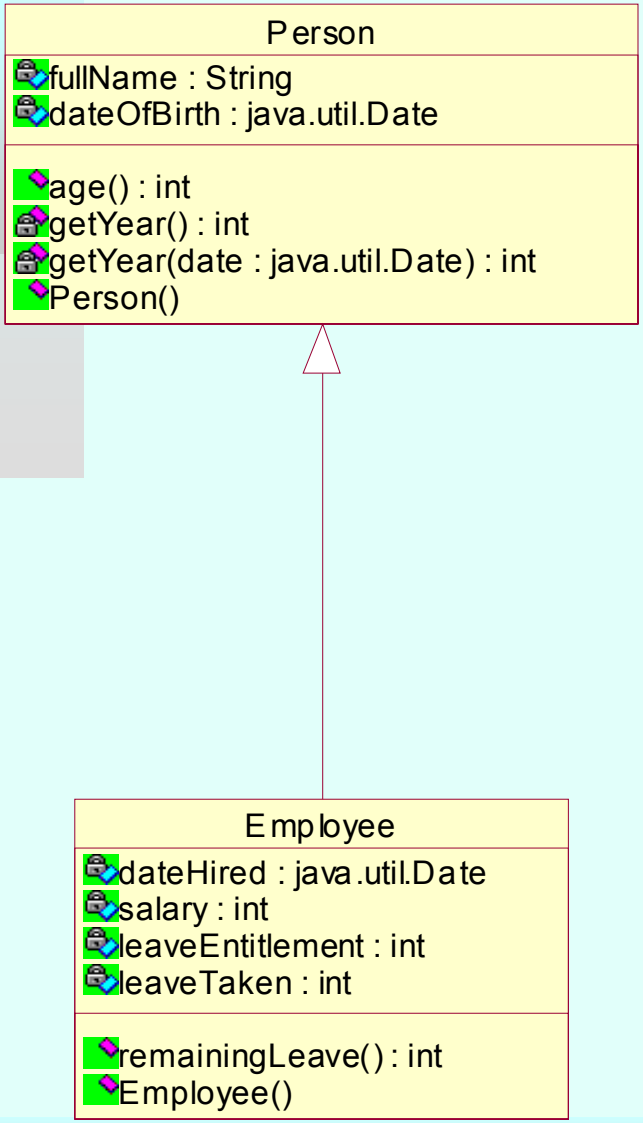
- A subclass inherits the structure and behavior of its superclass



Not a good visualization of generalization. Subclasses inherit types, not values (a nose not a long nose)!

BOOCH, G. (1994): *Object Oriented Analysis and Design with Applications*, 2nd ed, The Benjamin/Cummings Publ

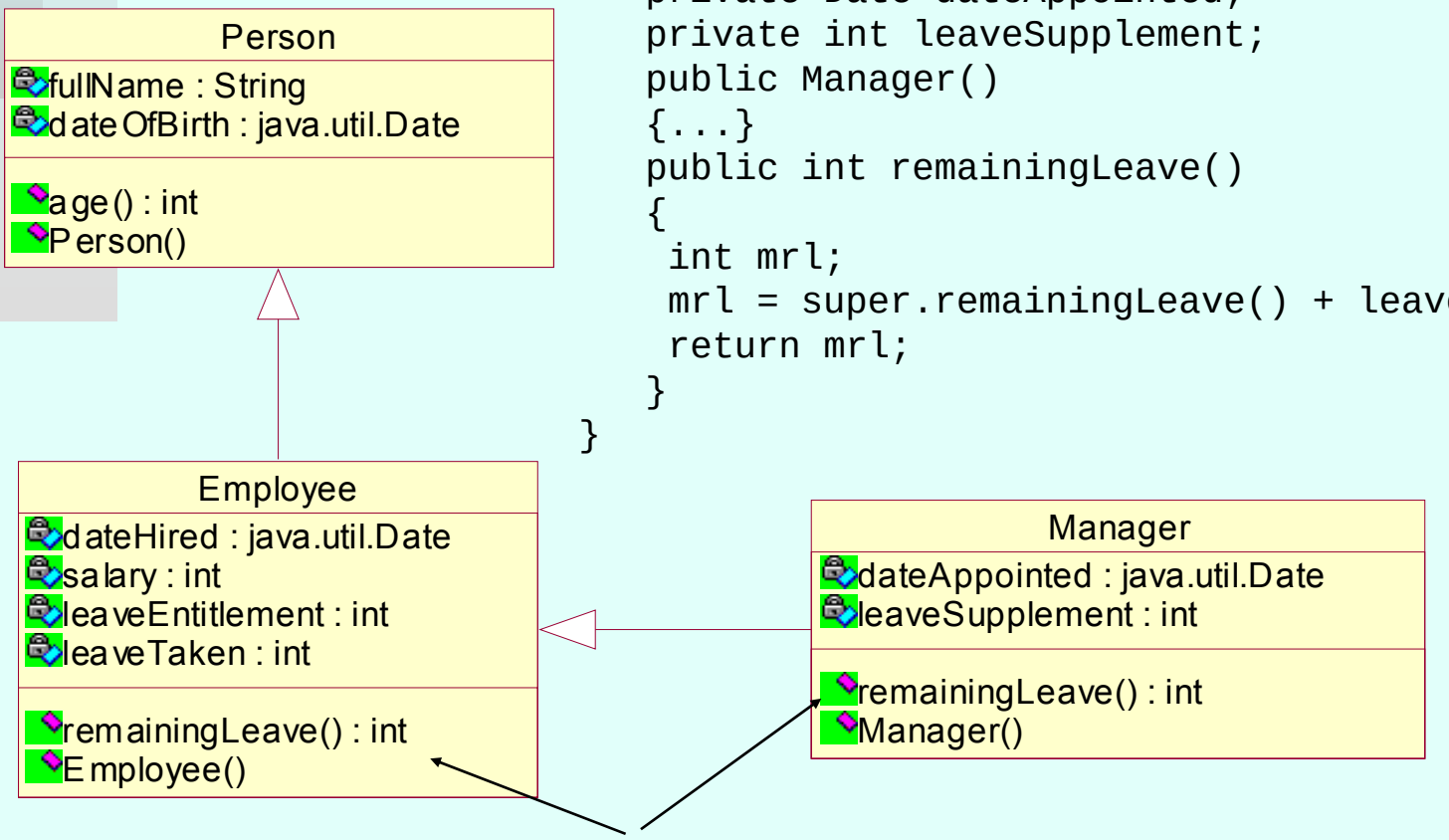
Generalization




```
public class Person
{
   private String fullName;
   private Date dateOfBirth;
   public Person()
   {...}
   public int age(){
      return getYear() - getYear(dateOfBirth);
   }
}

public class Employee extends Person
{
   private Date dateHired;
   private int salary;
   private int leaveEntitlement;
   private int leaveTaken;
   public Employee()
   {...}
   public int remainingLeave(){
      return leaveEntitlement - leaveTaken;
   }
}
```

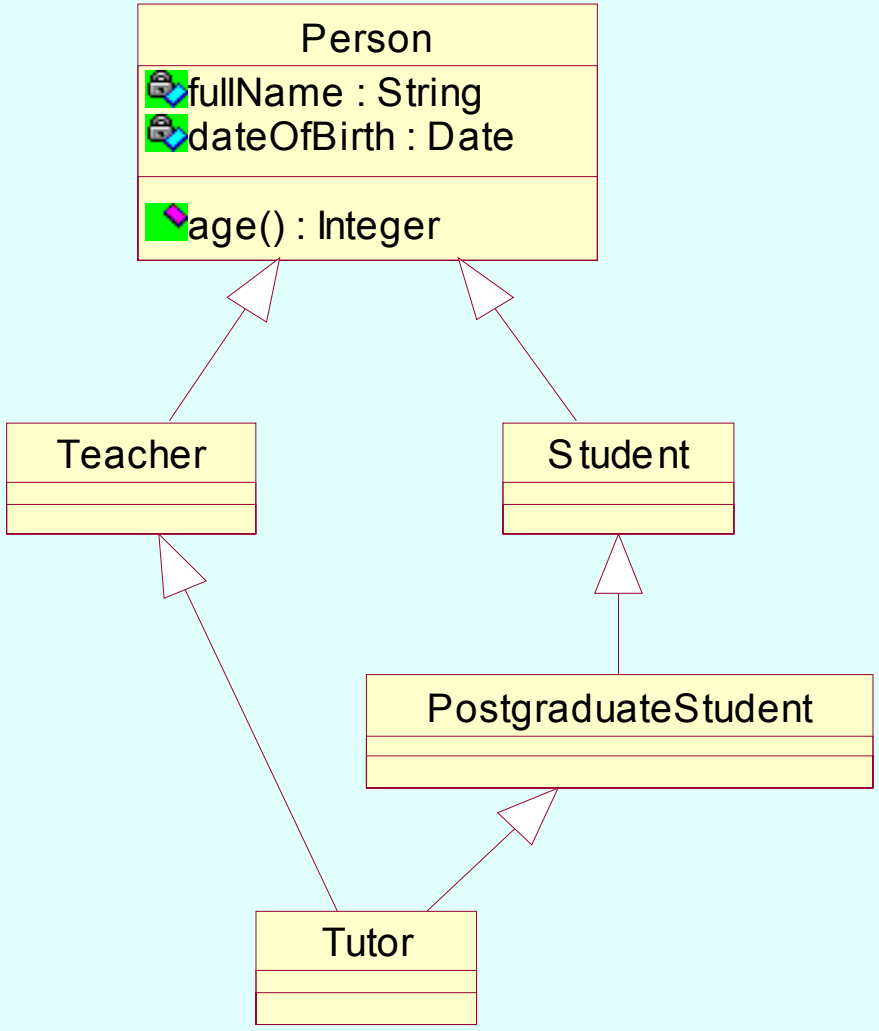
Polymorphism

```
public class Manager extends Employee
{
   private Date dateAppointed;
   private int leaveSupplement;
   public Manager()
   {...}
   public int remainingLeave()
   {
    int mrl;
    mrl = super.remainingLeave() + leaveSupplement;
    return mrl;
   }
}
```



The same signature
(operation name and the number and type of arguments)

Multiple inheritance



Multiple classification

- Multiple inheritance
  - A class may have many superclasses, but a single class must be defined for each object
- Multiple classification
  - An object is simultaneously the instance of two or more classes

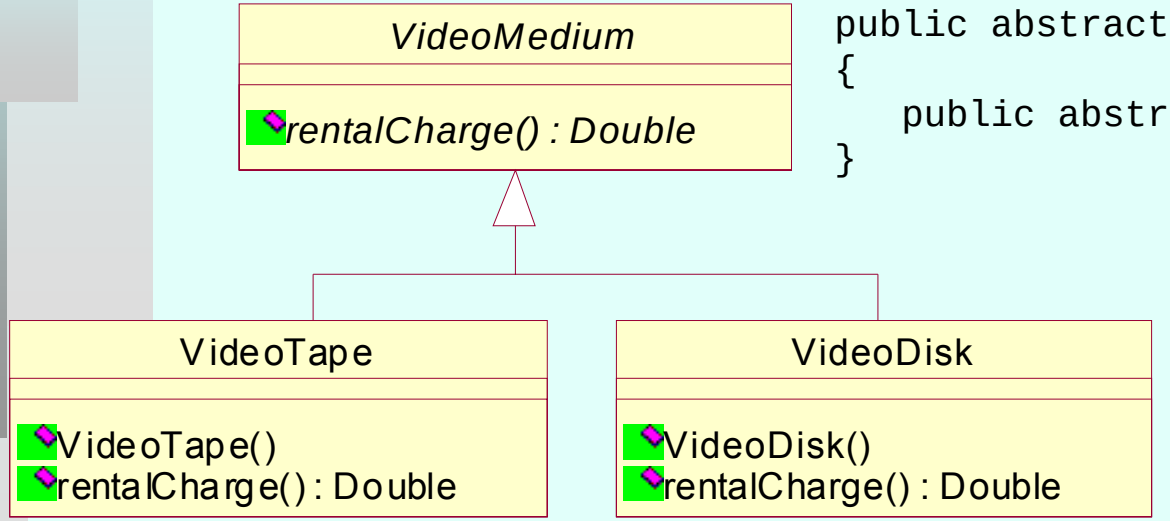
- *The problem arises if `Person` is specialized in few orthogonal hierarchies*
  - *`Person` can be `Employee` or `Student`, `Male` or `Female`, `Child` or `Adult`, etc.*
- *Without multiple classification*
  - *need to define classes for each legal combination between the orthogonal hierarchies*
  - *`ChildFemaleStudent` etc.*

Dynamic classification

- An object does not only belong to multiple classes but it can gain or lose classes over its lifetime
- A `Person` object can be just an employee one day and a manager (and employee) another day
- In most current object-oriented programming environments, an object cannot change its class after it has been instantiated (created)

Abstract class

- **Parent class** that will not have direct instance objects
- **Abstract class** cannot instantiate objects because it has at least one **abstract operation**



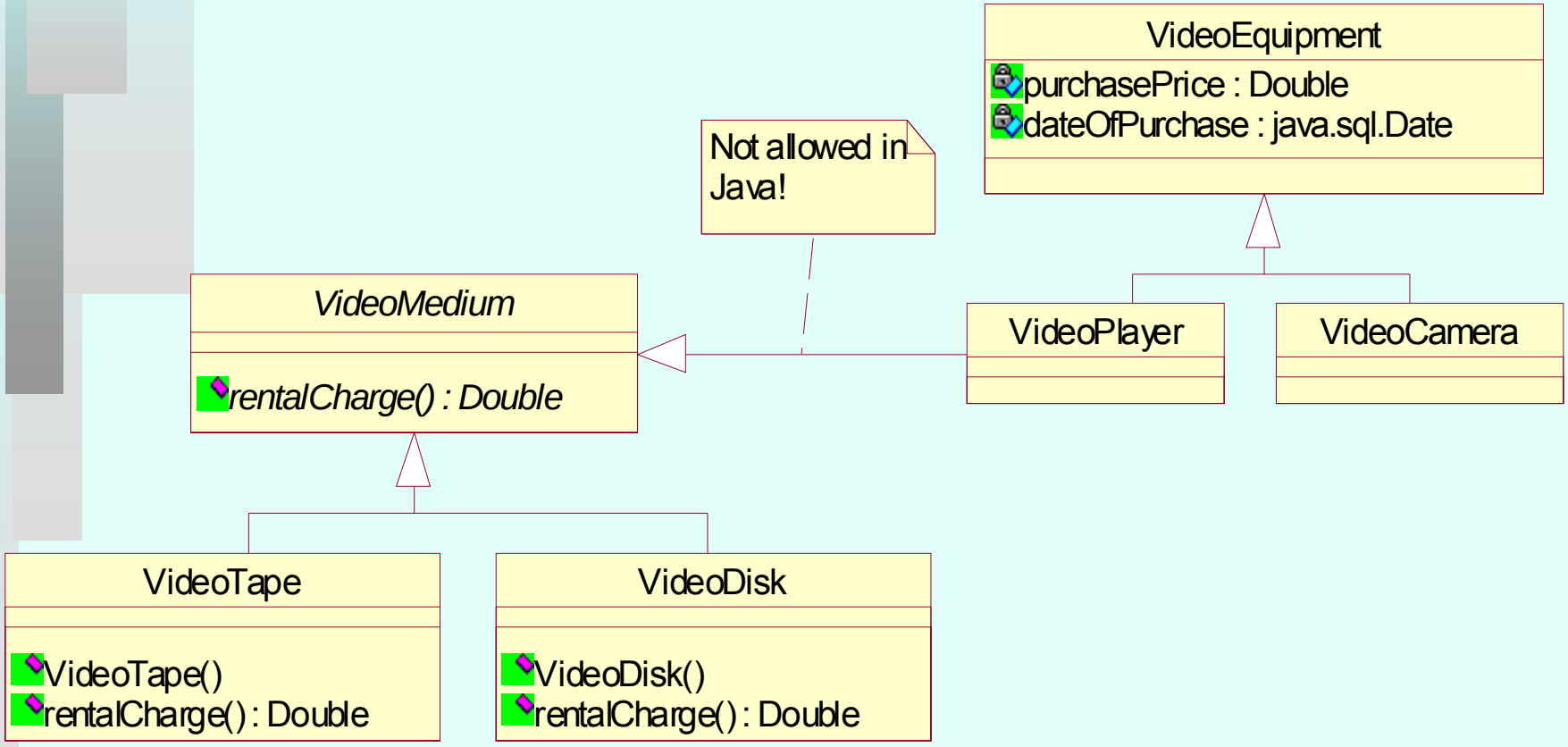

```
public abstract class VideoMedium
{
   public abstract Double rentalCharge();
}
```

```
public class VideoDisk extends VideoMedium
{
   public VideoDisk()
   {...}
   public Double rentalCharge()
   {
    return null;
   }
}
```

Interface vs abstract class

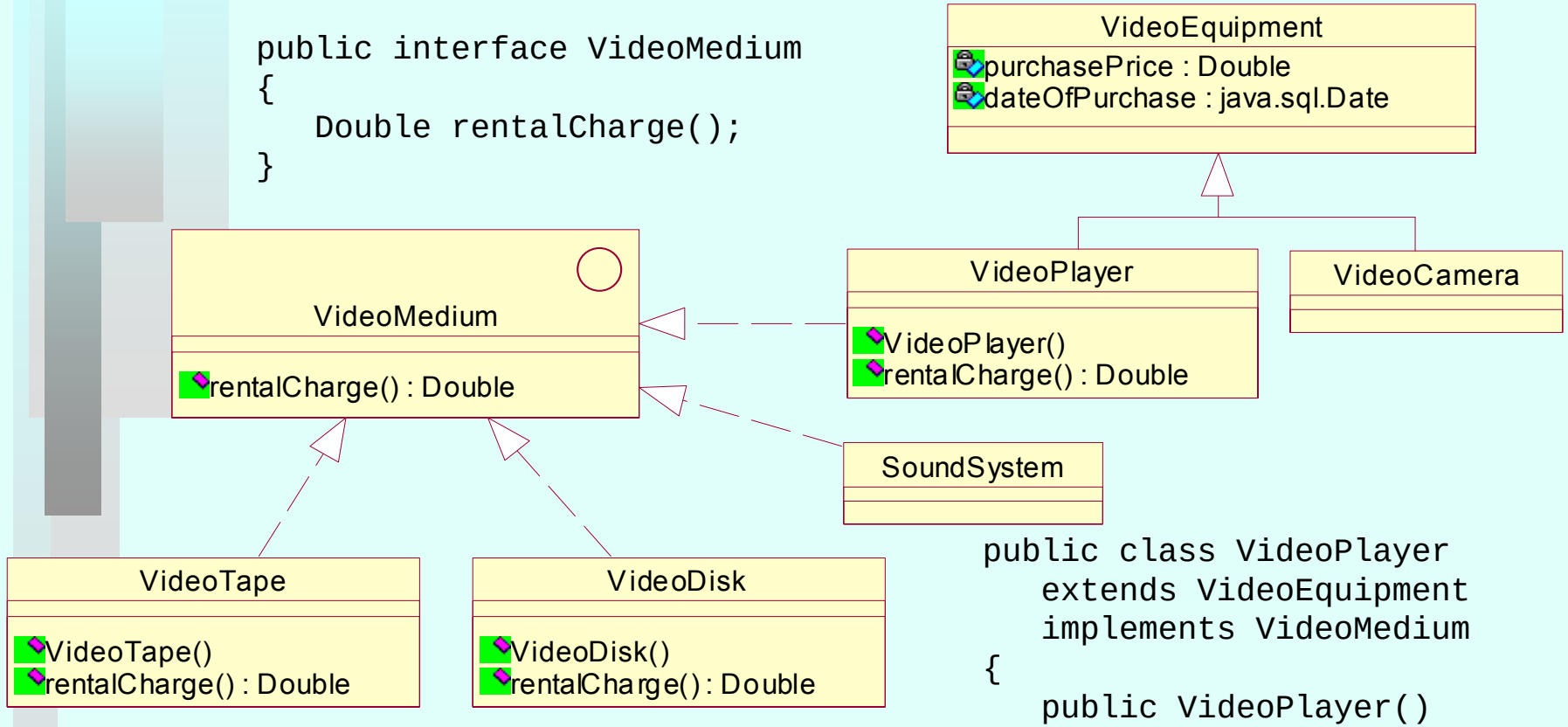
- **Interface** – a definition of a semantic type with attributes (constants only) and operations but without actual declarations (implementations) of operations
  - classes that implement the interface provide the declarations
- **Abstract class** has undesirable effect of the **fragile base class** problem
- Unlike abstract classes
  - **interfaces** are helpful for modeling situations that ask for multiple inheritance
  - **interface** does not implement (even partially) any of its methods
    - but still, **pure abstract class ≠ interface**
      - in case of interface, any class in the system can implement it, not just the subclasses
      - a class can implement any number of interfaces
  - interface defines a **reference type** that separates client objects from the implementation changes in the supplier objects
    - the implementation of the interface can change and the client object may not be affected

Multiple implementation inheritance not allowed in Java



Implementing Java interface

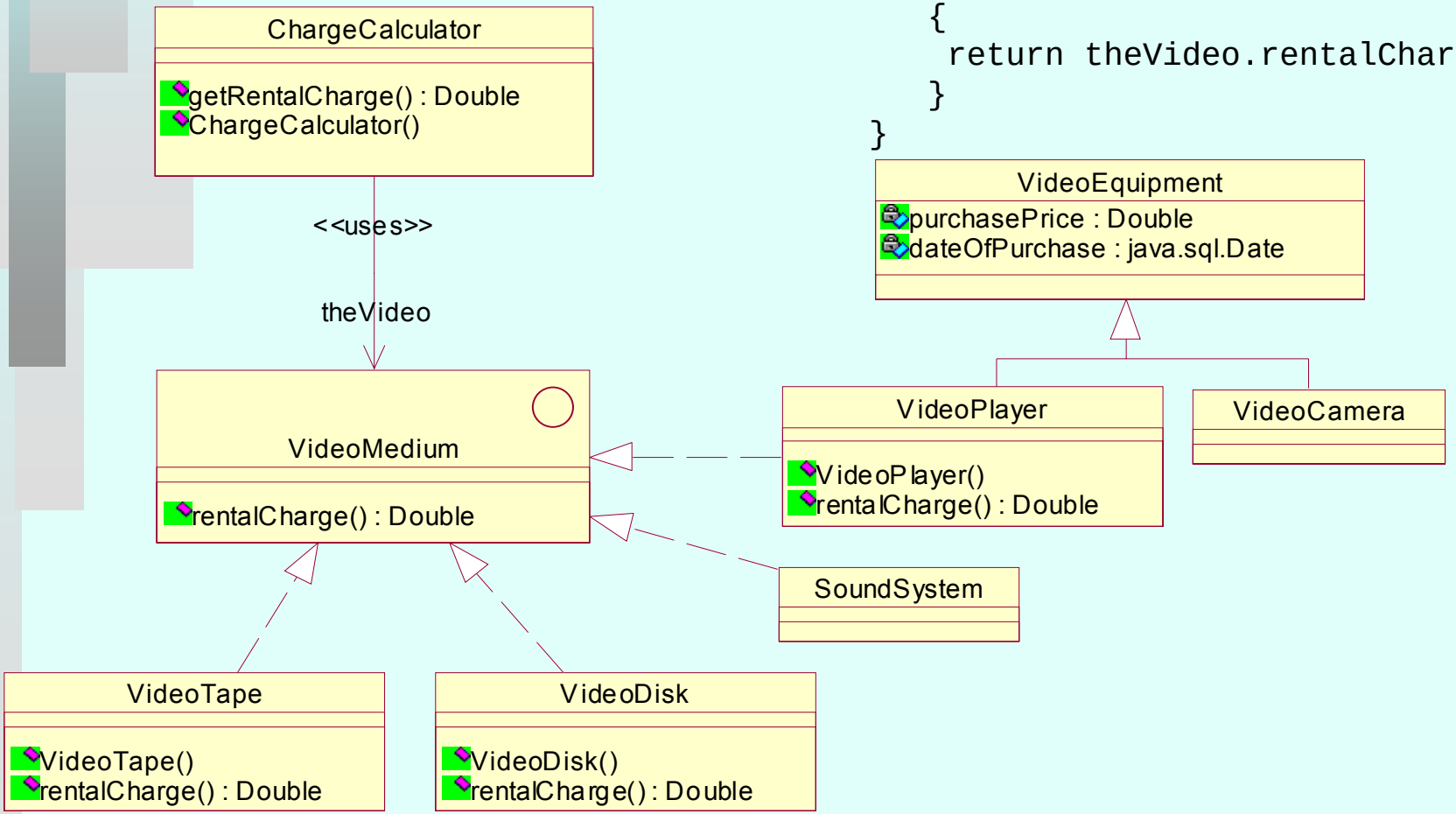
```
public interface VideoMedium
{
    Double rentalCharge();
}
```




```
public class VideoPlayer
    extends VideoEquipment
    implements VideoMedium
{
    public VideoPlayer()
    {...}
    public Double rentalCharge()
    {
     return null;
    }
}
```

Using interface to eliminate dependency to supplier

```
public class ChargeCalculator
{
   VideoMedium theVideo;
   public Double getRentalCharge()
   {
    return theVideo.rentalCharge();
   }
}
```



Summary

- Each **object** has a state, behavior and identity
- There are **instance objects** and **class objects**
- **Class** defines attributes and operations
- There are three kinds of **relationships** – association, aggregation, generalization
- **Generalization** provides the basis for polymorphism and inheritance
- Commercial programming environments support **multiple inheritance** directly (C++, C#) or by means of interfaces (Java)
- **Multiple and dynamic classification** is still not supported commercially
- **Abstract classes** and **interfaces** are important in modeling