Lesson 2
Object Orientation

Software Analysis and Design

2nd Year, Computer Science

Universidad Autónoma de Madrid

Index

- **Object oriented design concepts**
  - **Comparison with structured programming**
- Objects and classes
- Encapsulation
- Inheritance and polymorphism
- Summary and conclusions

Object orientation

- Programming paradigm that considers applications as a set of interacting objects
  - Objects may model concepts of the real world, like a bank account, or a person
  - They contain data and methods (functions, procedures)

- Attempt to improve the development and maintenance process of software
  - Reuse, extensibility

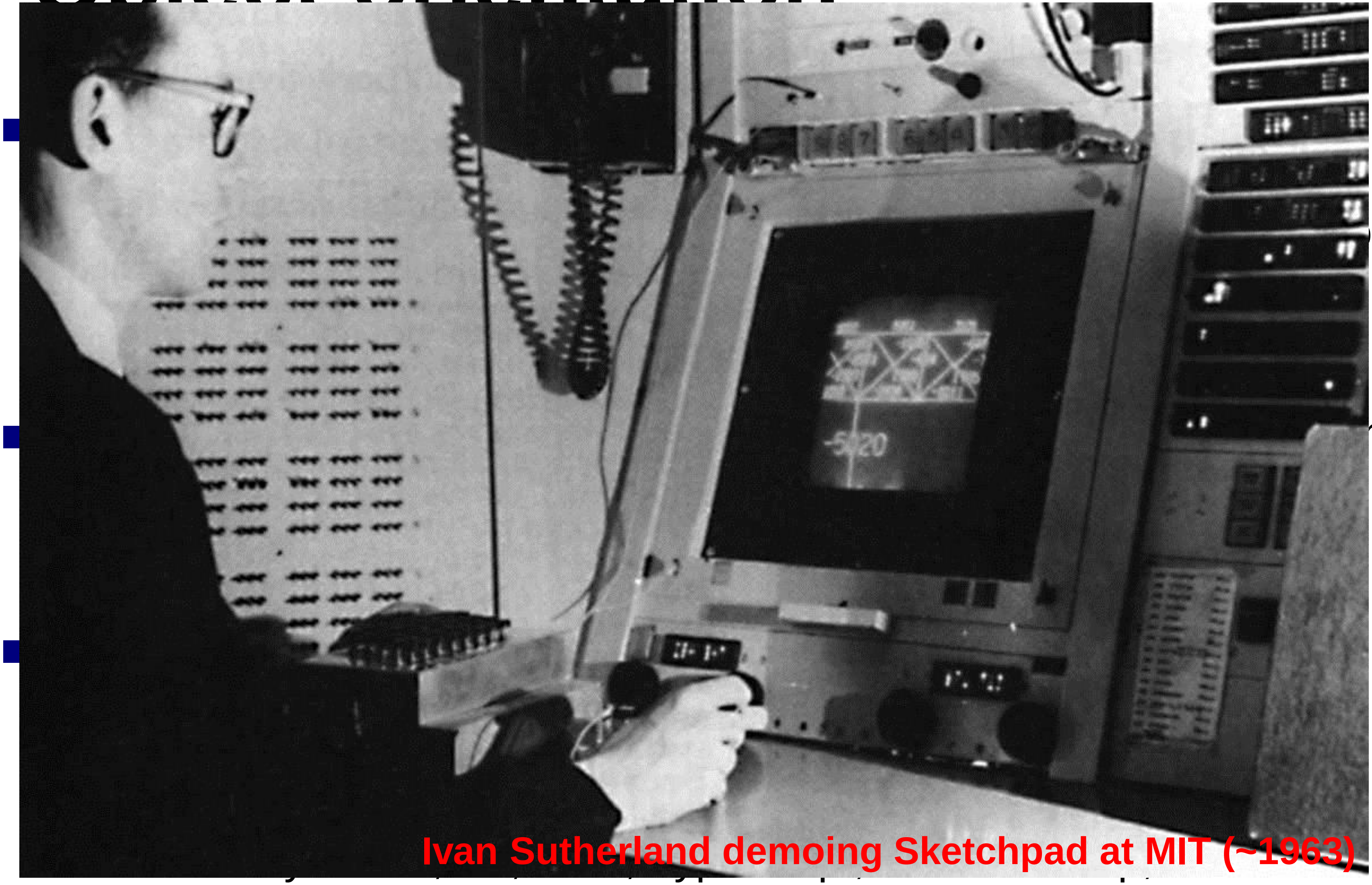
- Origin in the 60'
  - Sketchpad (MIT), ALGOL (ACM & GAMM)
  - Simula67 (Dahl, Nygaard)
  - Smalltalk (Xerox PARC) in the 70s (Kay)
  - Nowadays: C++, C#, Java, TypeScript, Common Lisp, etc

Object orientation

set

unt,

nce



Ivan Sutherland demoing Sketchpad at MIT (~1963)

Object orientation

- *“Add behaviour (methods) to data types (e.g., structs in C)”*

- Fundamental concepts
  - **Class**
    - “template” describing data and behaviour of a set of objects
  - **Object**
    - Run-time instance of a class
  - **Encapsulation**
    - Information hiding. Show only the interface of the object
  - **Polymorphism**
    - Refinement/generalization, inheritance
    - Safely use a more specialized object in place of a more general one

Example

- Using object orientation, design an application to manage the employees and clients of a company

- We need to store the name and birth date of every person. Of clients, we need the name, company and phone number

- The application needs to show the personal data of every person

- Employees have a gross salary, and a department. We want to calculate the net salary

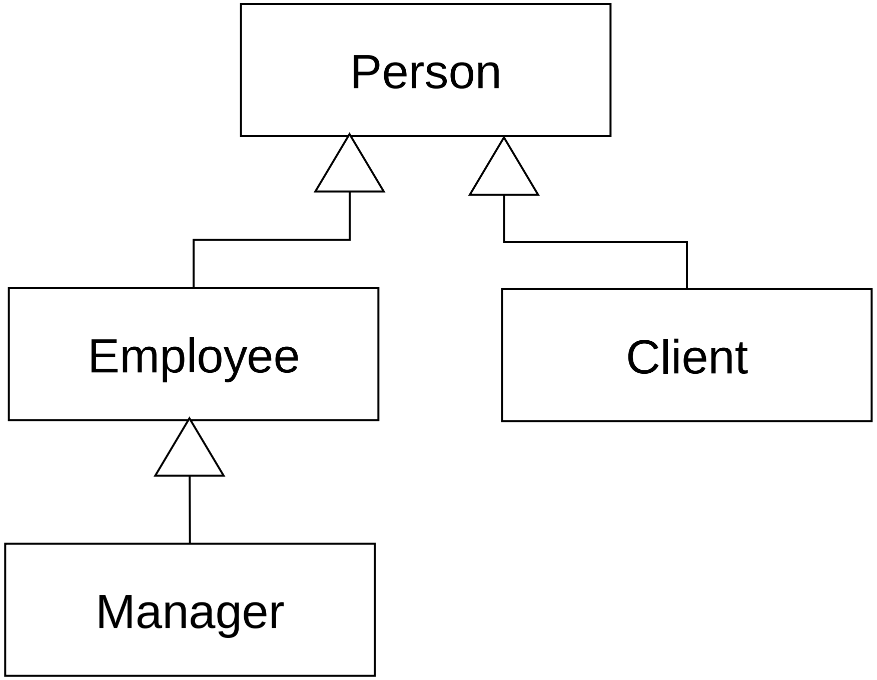
- Some employees are managers, and these have a category

Example

Designing the classes

- Identify relevant classes in the domain (nouns)

- Group classes with shared data and behaviour
    - Inheritance hierarchy

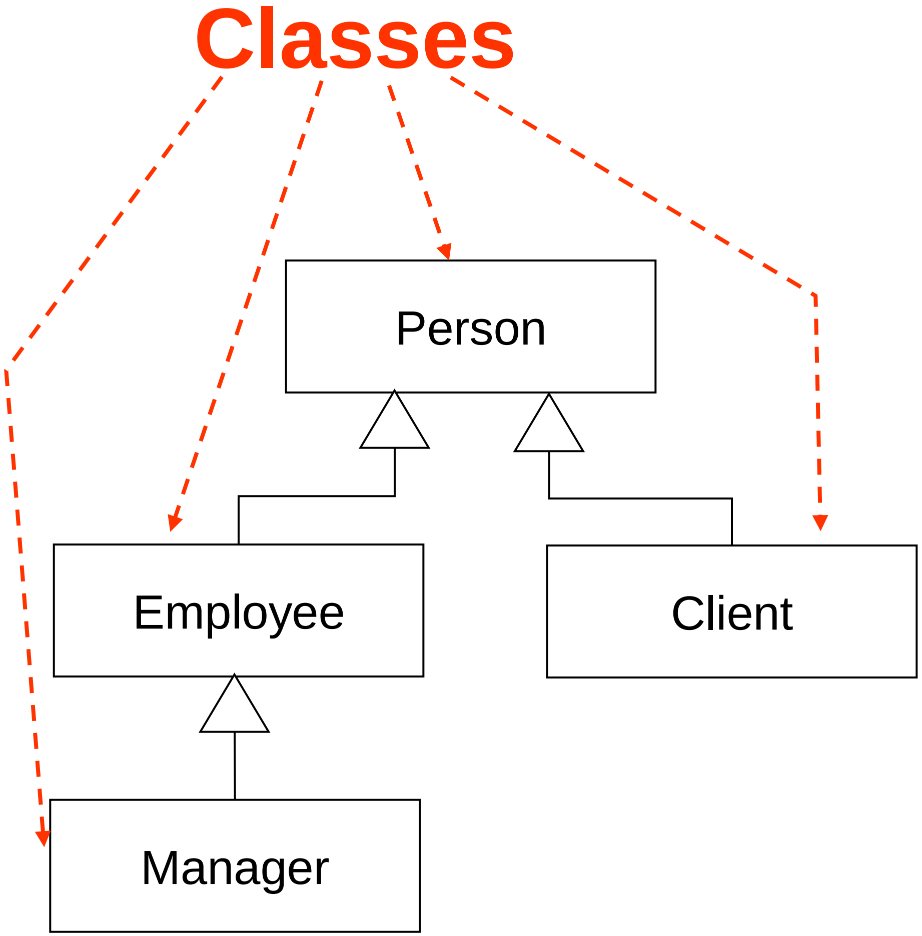


Example

Designing the classes

- Identify relevant classes in the domain (nouns)

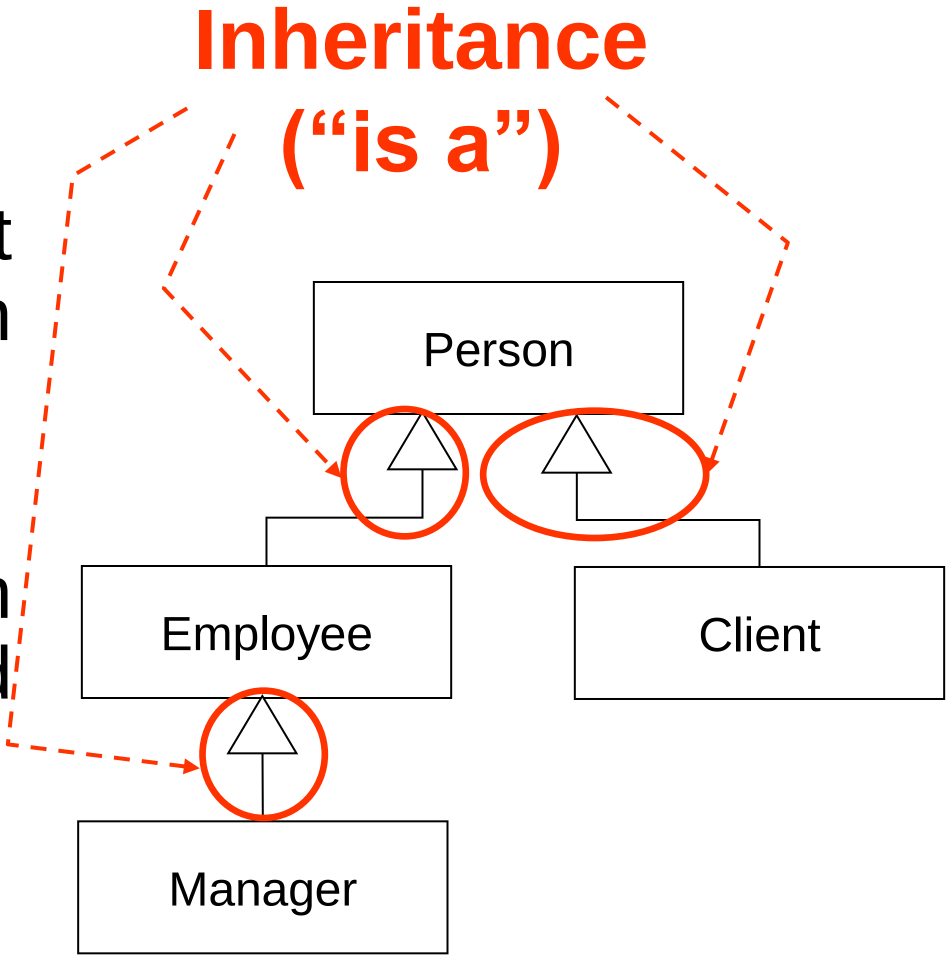
- Group classes with shared data and behaviour
  - Inheritance hierarchy



Example

Designing the classes

- Identify relevant classes in the domain (nouns)
- Group classes with shared data and behaviour
  - Inheritance hierarchy



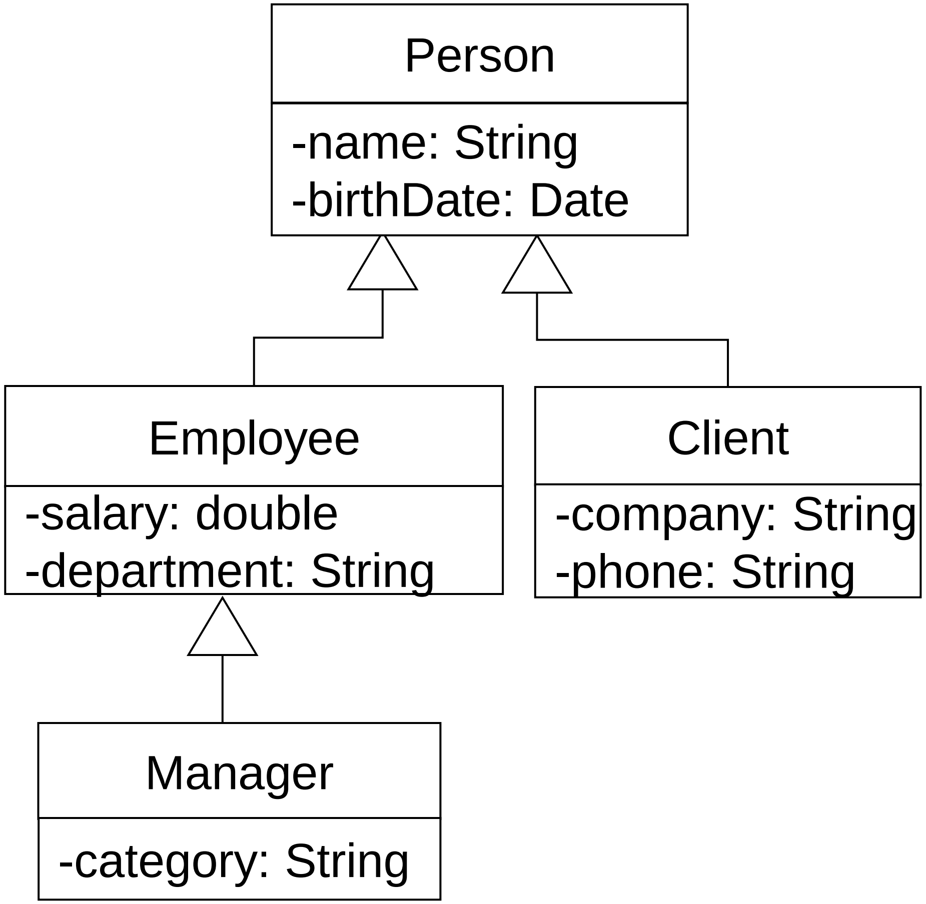
Example

Designing the classes

- Common data

- Data of the parent class are inherited by the child class (also called super/subclass)

- The child class can add additional data



Person
-name: String
-birthDate: Date

Employee
-salary: double
-department: String

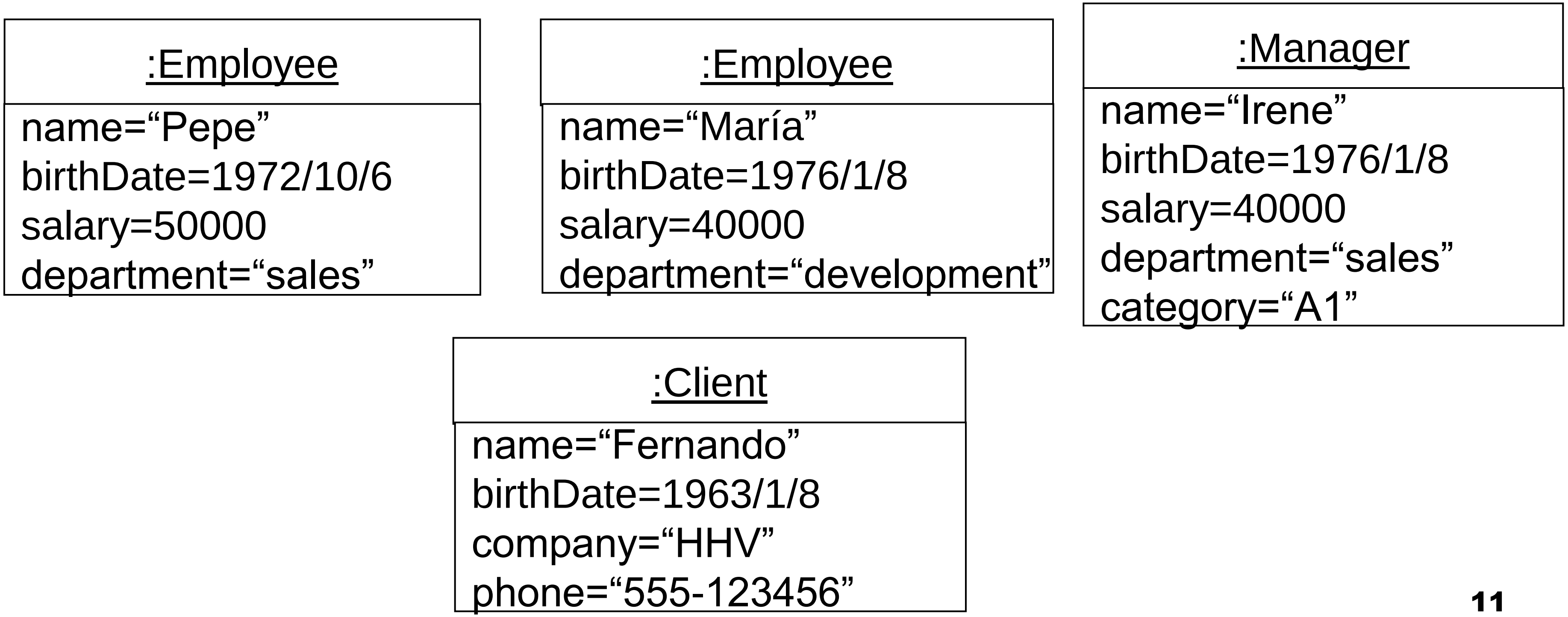
Client
-company: String
-phone: String

Manager
-category: String

Example

Objects

- Instances of classes at run-time.
- Classes are used as templates to create objects.



:Employee

name="Pepe"
birthDate=1972/10/6
salary=50000
department="sales"

:Employee

name="María"
birthDate=1976/1/8
salary=40000
department="development"

:Manager

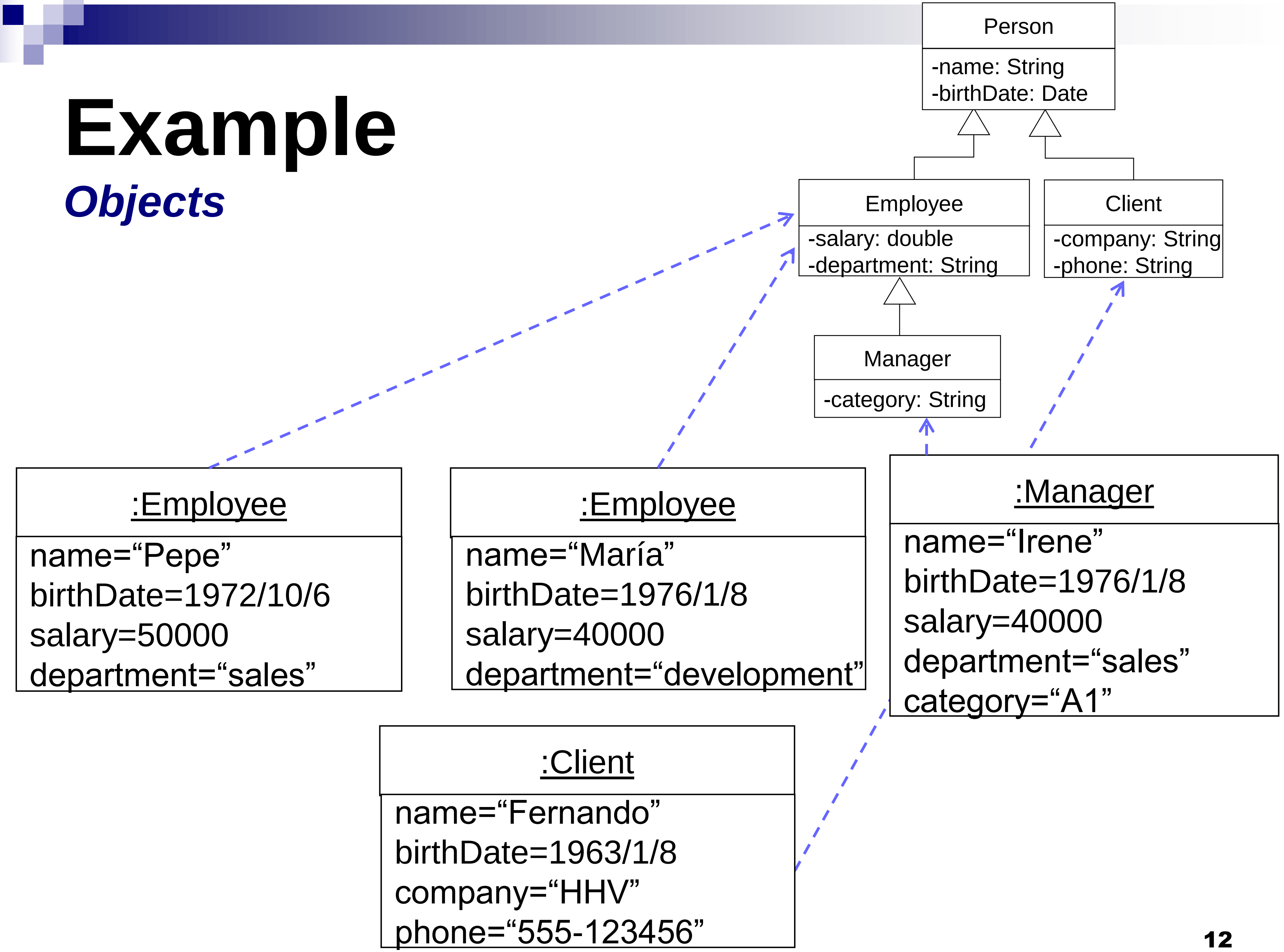
name="Irene"
birthDate=1976/1/8
salary=40000
department="sales"
category="A1"

:Client

name="Fernando"
birthDate=1963/1/8
company="HHV"
phone="555-123456"

Example

Objects



Person
-name: String
-birthDate: Date

Employee
-salary: double
-department: String

Client
-company: String
-phone: String

Manager
-category: String

:Employee
name="Pepe"
birthDate=1972/10/6
salary=50000
department="sales"

:Employee
name="María"
birthDate=1976/1/8
salary=40000
department="development"

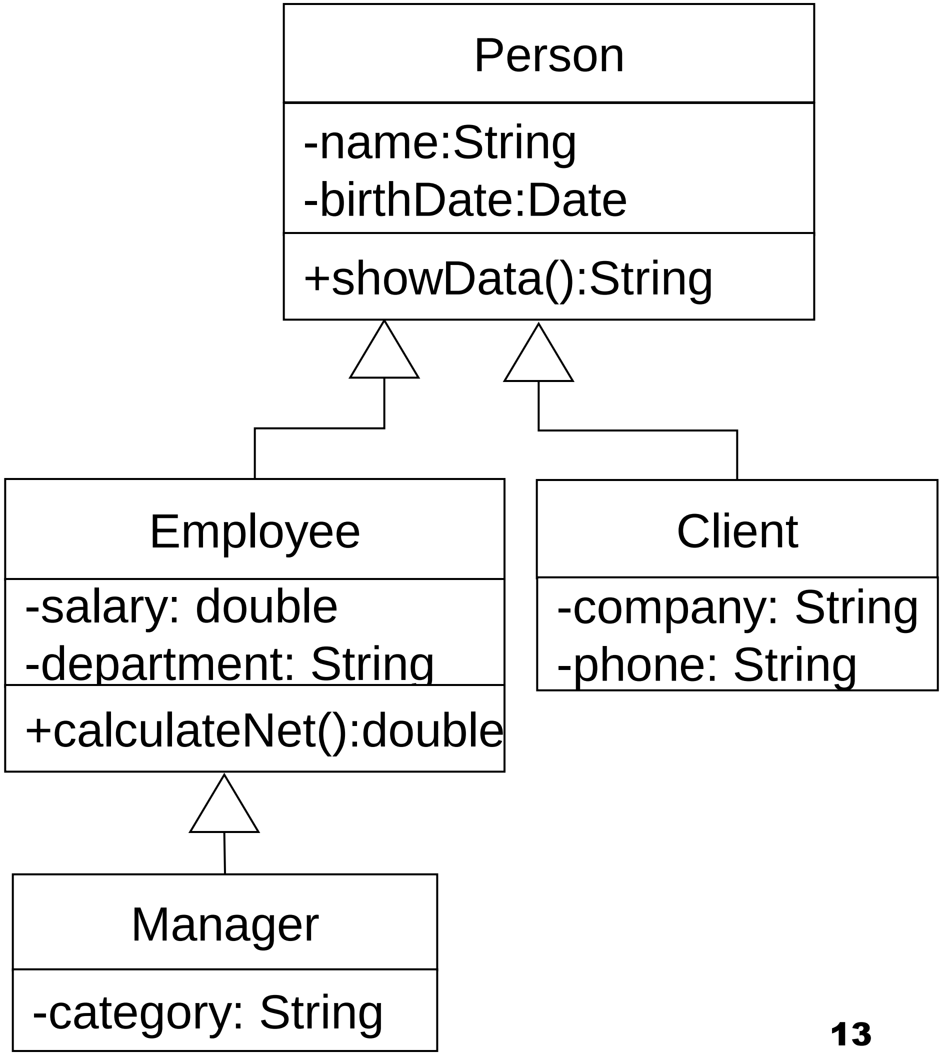
:Manager
name="Irene"
birthDate=1976/1/8
salary=40000
department="sales"
category="A1"

:Client
name="Fernando"
birthDate=1963/1/8
company="HHV"
phone="555-123456"

Example

Designing the classes: Behaviour

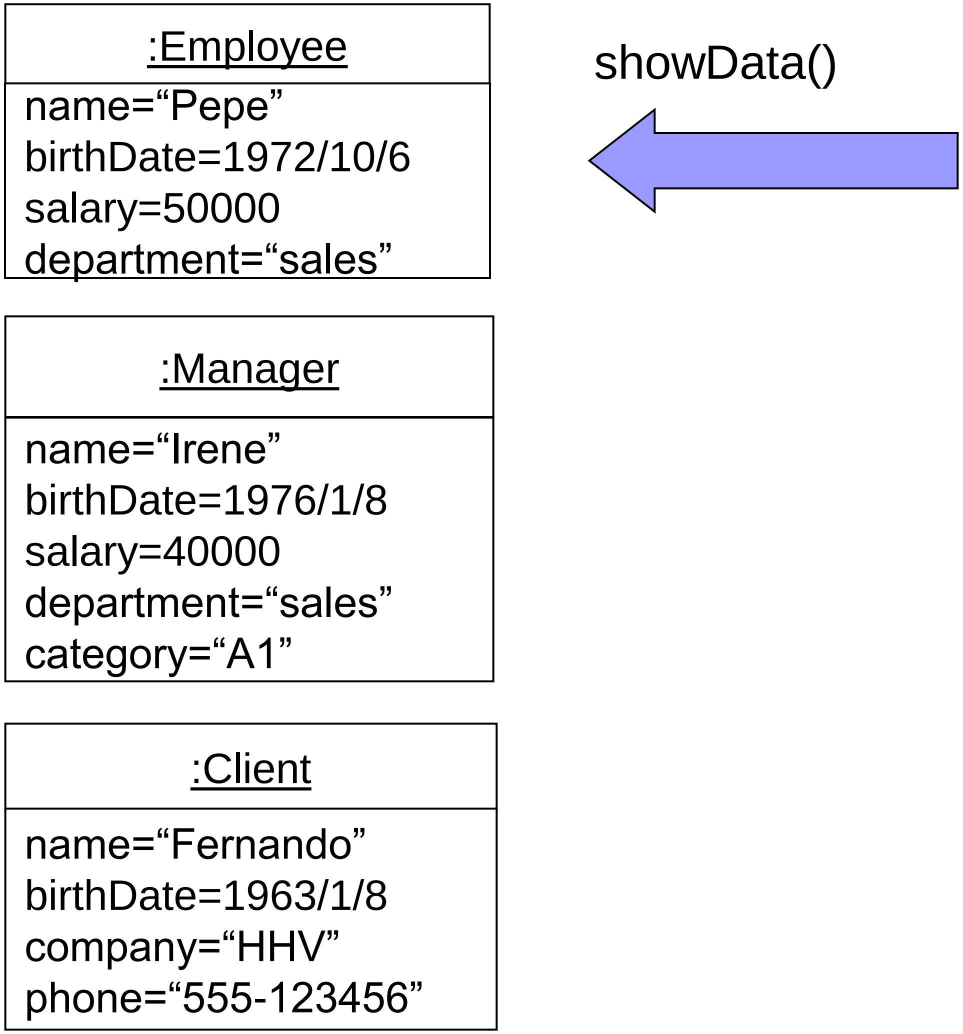
- Behaviour is encapsulated in methods (functions or procedures in the scope of a class)
- Methods of the superclass are inherited by the subclass
- The subclass can add additional methods



Example

Behaviour execution

[click-me]



:Employee

name=“Pepe”
birthDate=1972/10/6
salary=50000
department=“sales”

showData()

:Manager

name=“Irene”
birthDate=1976/1/8
salary=40000
department=“sales”
category=“A1”

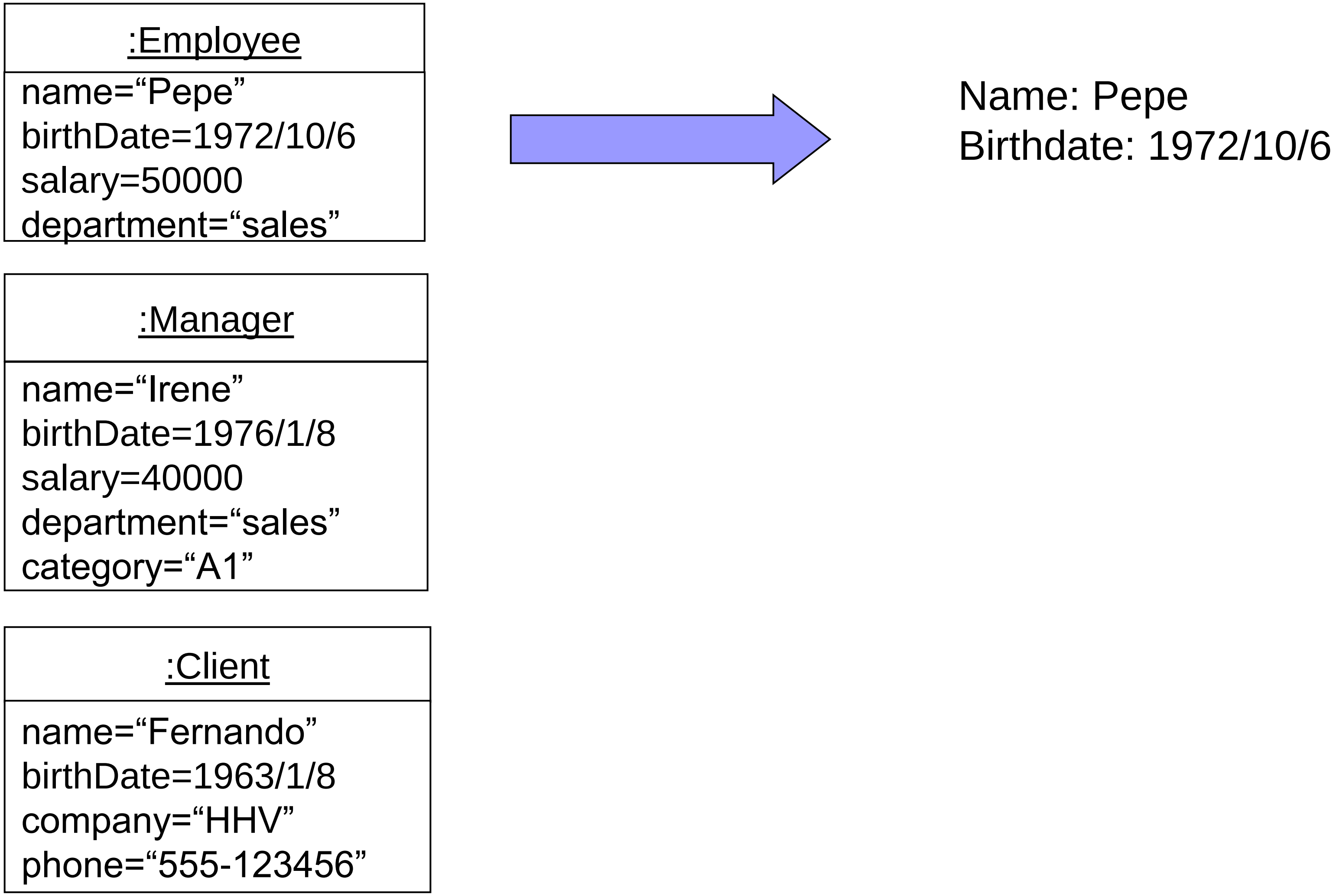
:Client

name=“Fernando”
birthDate=1963/1/8
company=“HHV”
phone=“555-123456”

Example

Behaviour execution

[click-me]



:Employee

name=“Pepe”
birthDate=1972/10/6
salary=50000
department=“sales”

:Manager

name=“Irene”
birthDate=1976/1/8
salary=40000
department=“sales”
category=“A1”

:Client

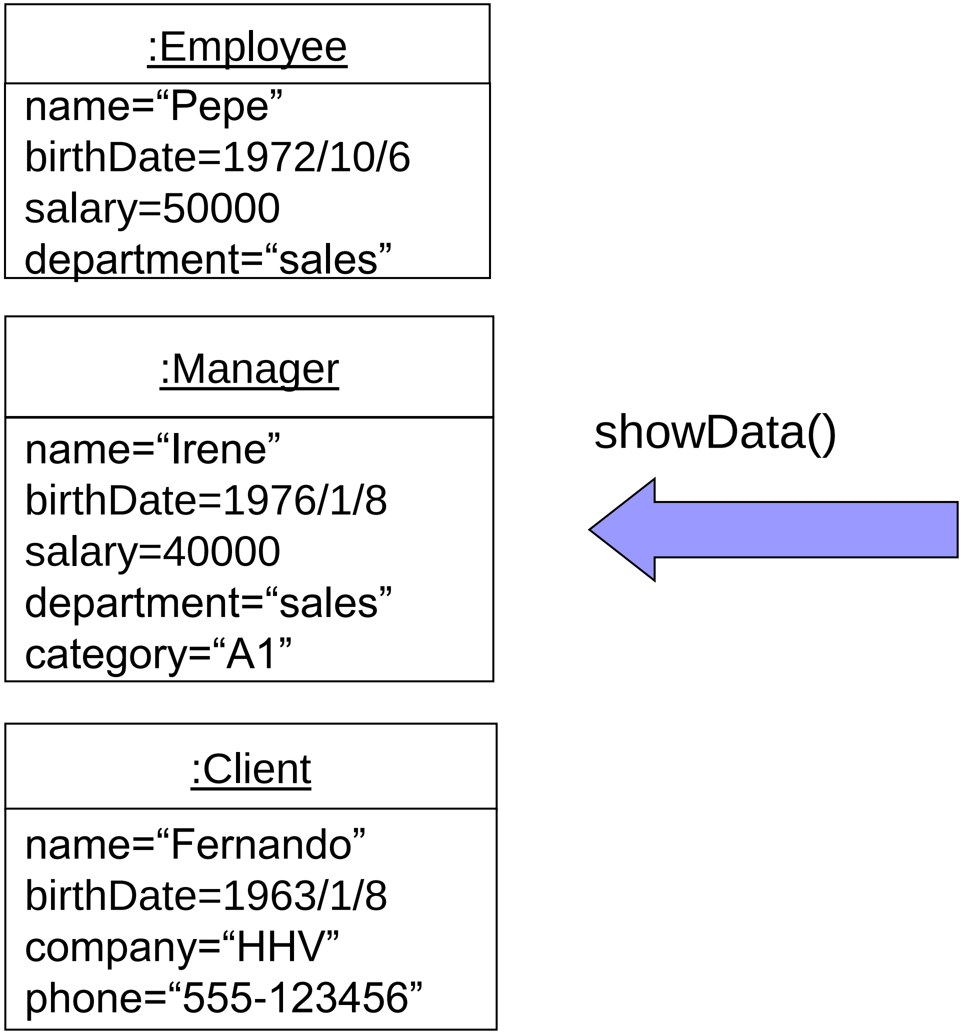
name=“Fernando”
birthDate=1963/1/8
company=“HHV”
phone=“555-123456”

Name: Pepe
Birthdate: 1972/10/6

[click-me]

Example

Behaviour execution



:Employee

name="Pepe"
birthDate=1972/10/6
salary=50000
department="sales"

:Manager

name="Irene"
birthDate=1976/1/8
salary=40000
department="sales"
category="A1"

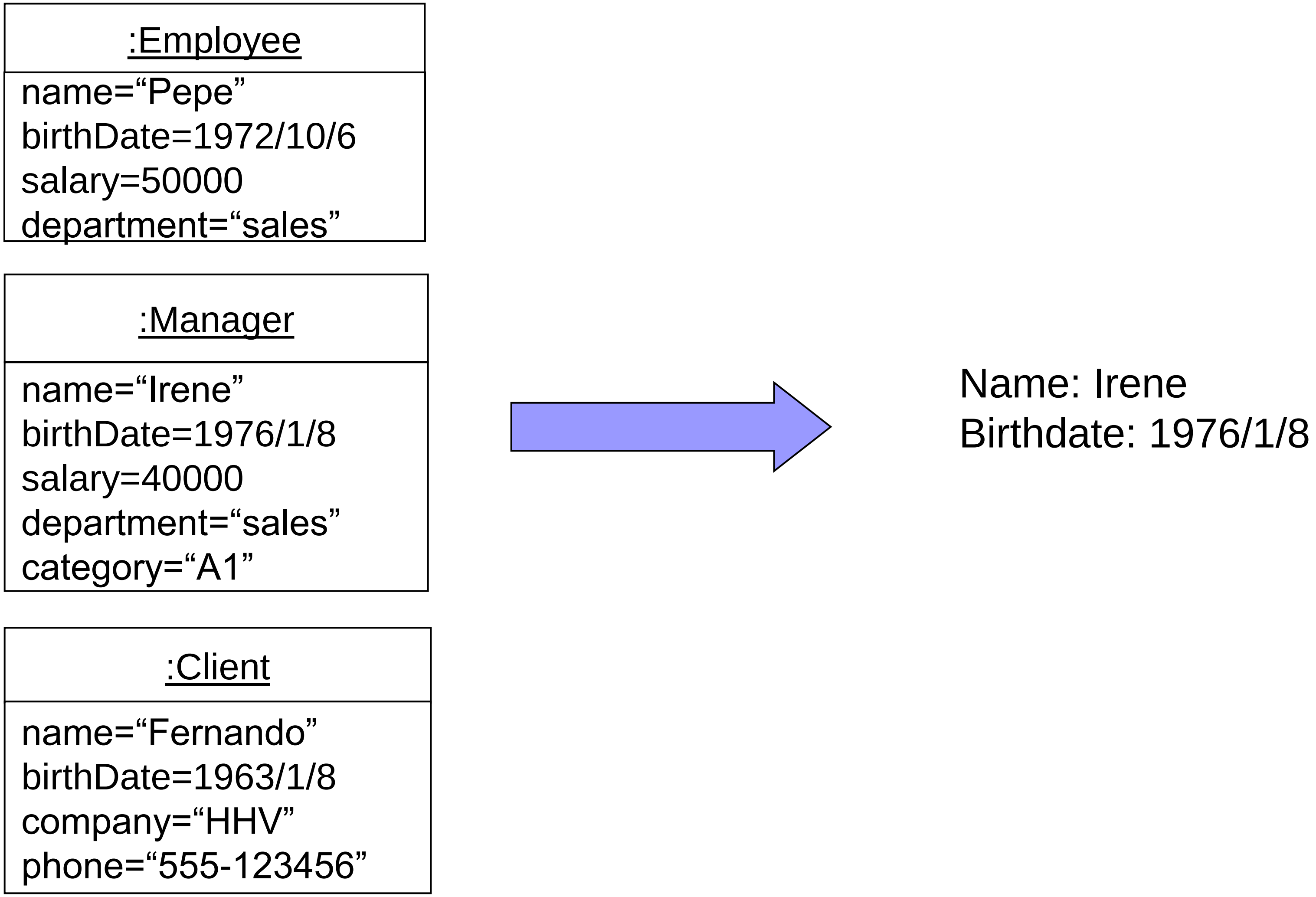
:Client

name="Fernando"
birthDate=1963/1/8
company="HHV"
phone="555-123456"

Example

Behaviour execution

[click-me]



:Employee

name="Pepe"
birthDate=1972/10/6
salary=50000
department="sales"

:Manager

name="Irene"
birthDate=1976/1/8
salary=40000
department="sales"
category="A1"

:Client

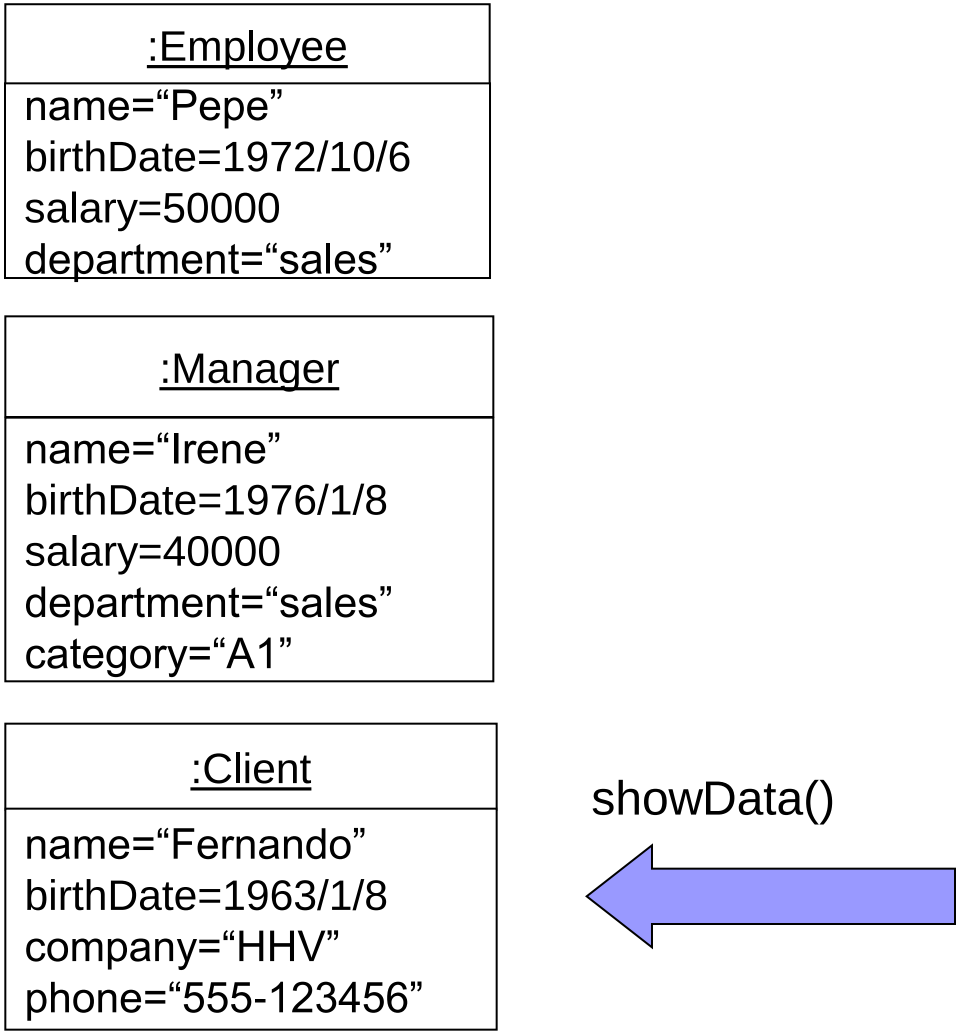
name="Fernando"
birthDate=1963/1/8
company="HHV"
phone="555-123456"

Name: Irene
Birthdate: 1976/1/8

[click-me]

Example

Behaviour execution



:Employee

name="Pepe"
birthDate=1972/10/6
salary=50000
department="sales"

:Manager

name="Irene"
birthDate=1976/1/8
salary=40000
department="sales"
category="A1"

:Client

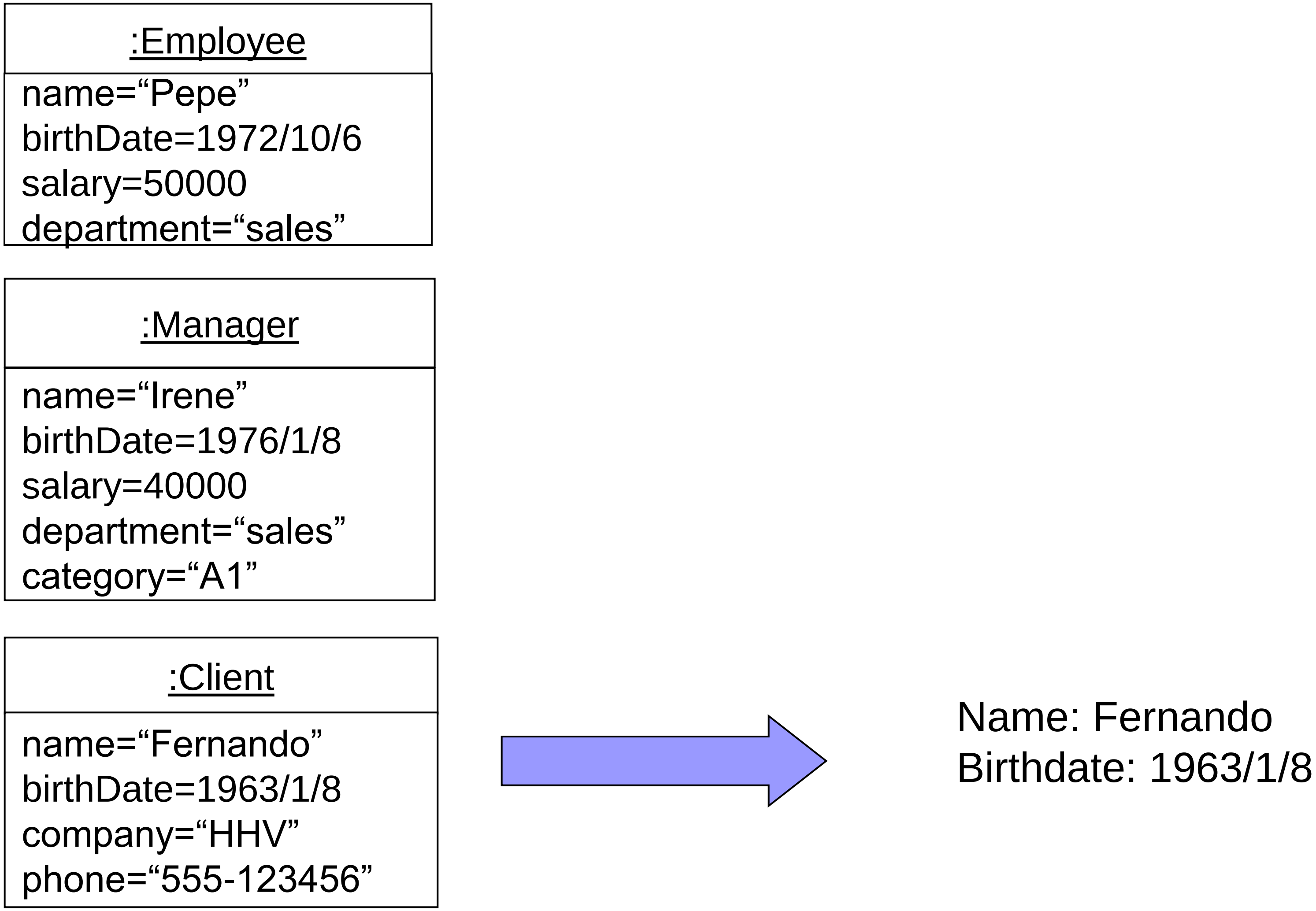
name="Fernando"
birthDate=1963/1/8
company="HHV"
phone="555-123456"

showData()

Example

Behaviour execution

[click-me]



:Employee

name="Pepe"
birthDate=1972/10/6
salary=50000
department="sales"

:Manager

name="Irene"
birthDate=1976/1/8
salary=40000
department="sales"
category="A1"

:Client

name="Fernando"
birthDate=1963/1/8
company="HHV"
phone="555-123456"

Name: Fernando
Birthdate: 1963/1/8

Example

Behaviour specialization

- Method specialization (overriding). Additional actions:
  - For a person, we need to show the name and birth date
  - For an employee, in addition, we need to show the salary and department
  - For a manager, the category
- We do not need to specialize calculateNet(): the procedure is the same for employees and managers

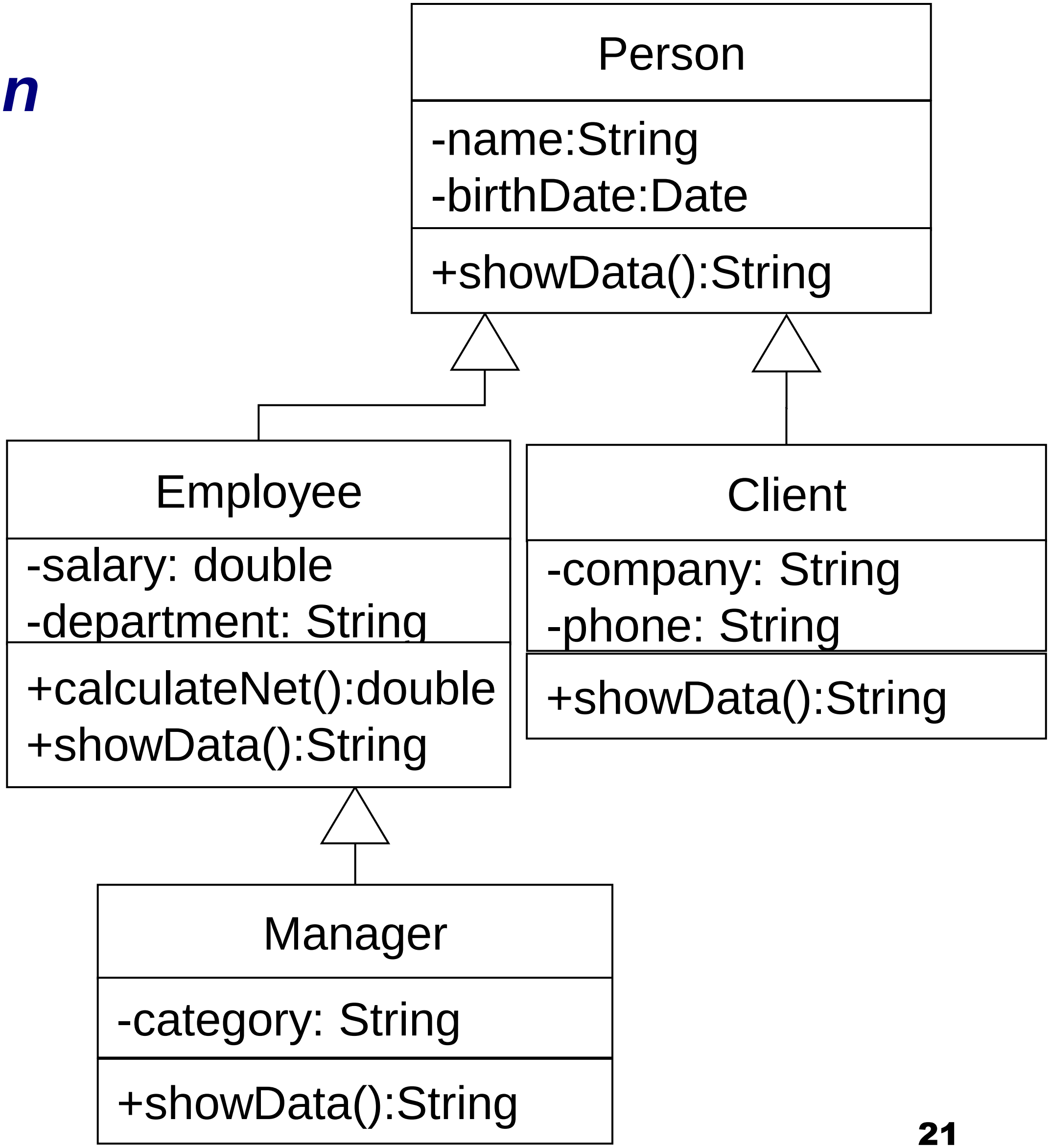
Example

Behaviour specialization

- showData() shows in addition:
  - Employee’s salary and department.
  - Client’s company and phone
  - Manager’s category

- Modify the behaviour of the parent class

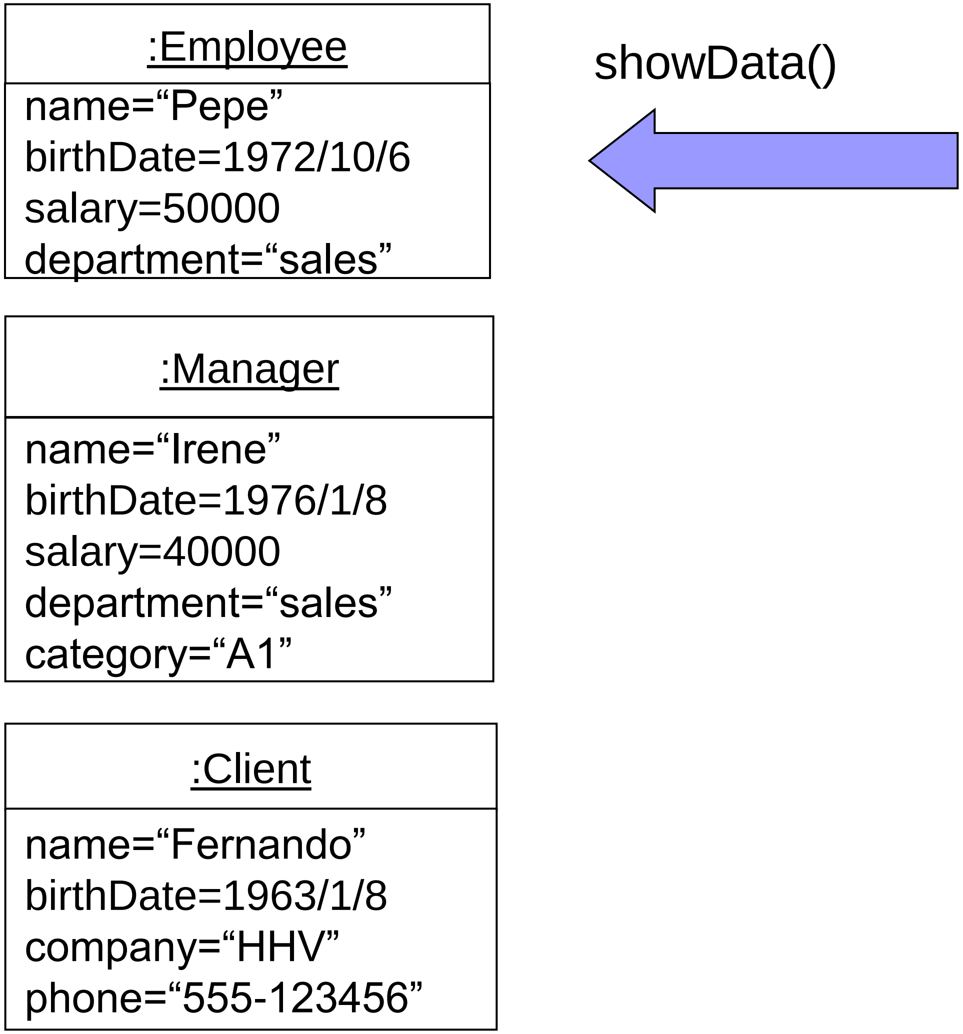
- Can call the original method of the parent class



[click-me]

Example

Behaviour execution



:Employee

name="Pepe"
birthDate=1972/10/6
salary=50000
department="sales"

showData()

:Manager

name="Irene"
birthDate=1976/1/8
salary=40000
department="sales"
category="A1"

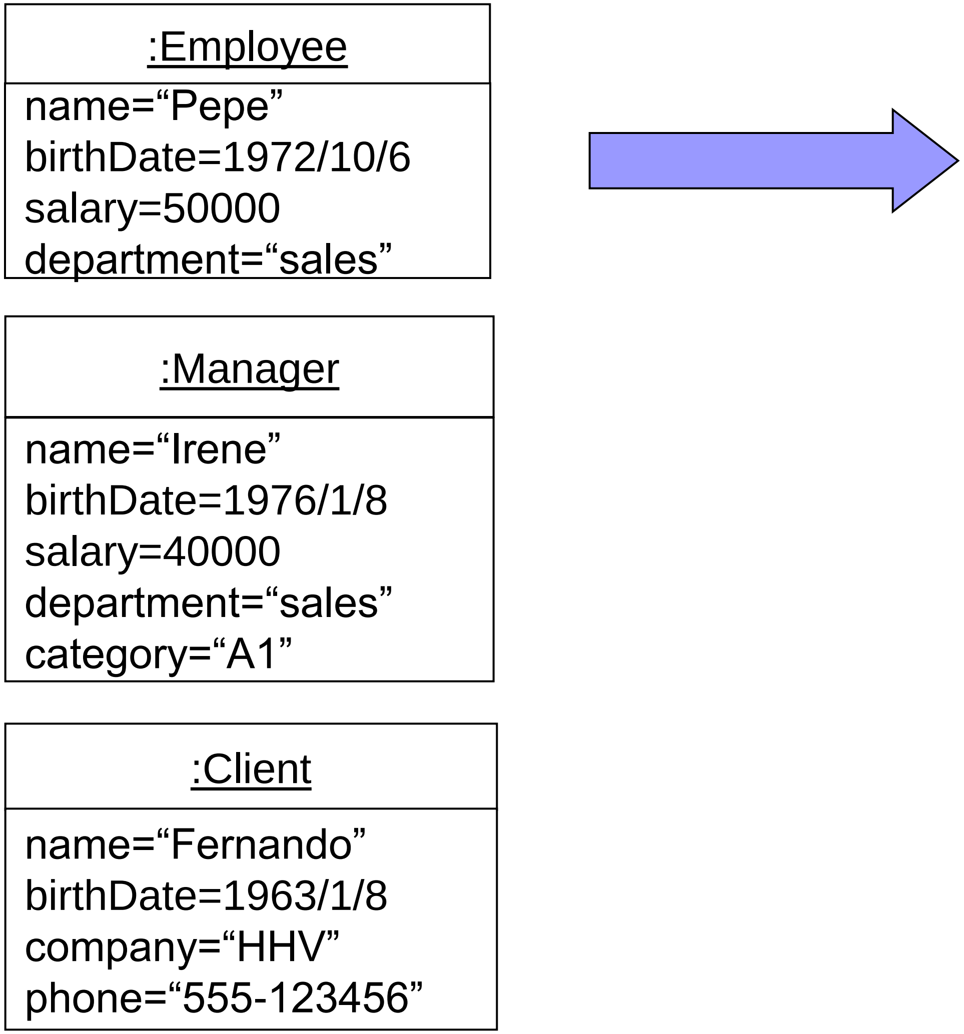
:Client

name="Fernando"
birthDate=1963/1/8
company="HHV"
phone="555-123456"

[click-me]

Example

Behaviour execution



:Employee

name="Pepe"
birthDate=1972/10/6
salary=50000
department="sales"

:Manager

name="Irene"
birthDate=1976/1/8
salary=40000
department="sales"
category="A1"

:Client

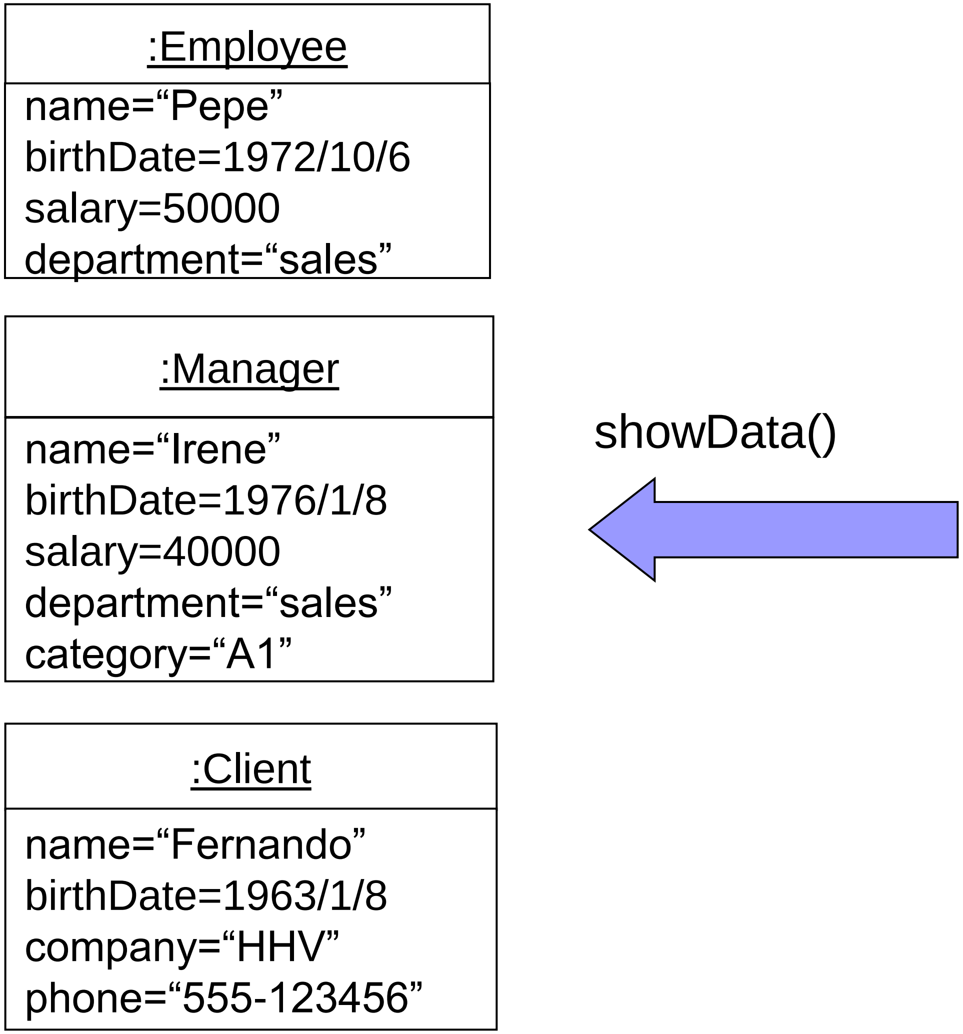
name="Fernando"
birthDate=1963/1/8
company="HHV"
phone="555-123456"

Name: Pepe
Birthdate: 6/10/72
Salary: 50000€
Department: sales

[click-me]

Example

Behaviour execution



:Employee

name=“Pepe”
birthDate=1972/10/6
salary=50000
department=“sales”

:Manager

name=“Irene”
birthDate=1976/1/8
salary=40000
department=“sales”
category=“A1”

:Client

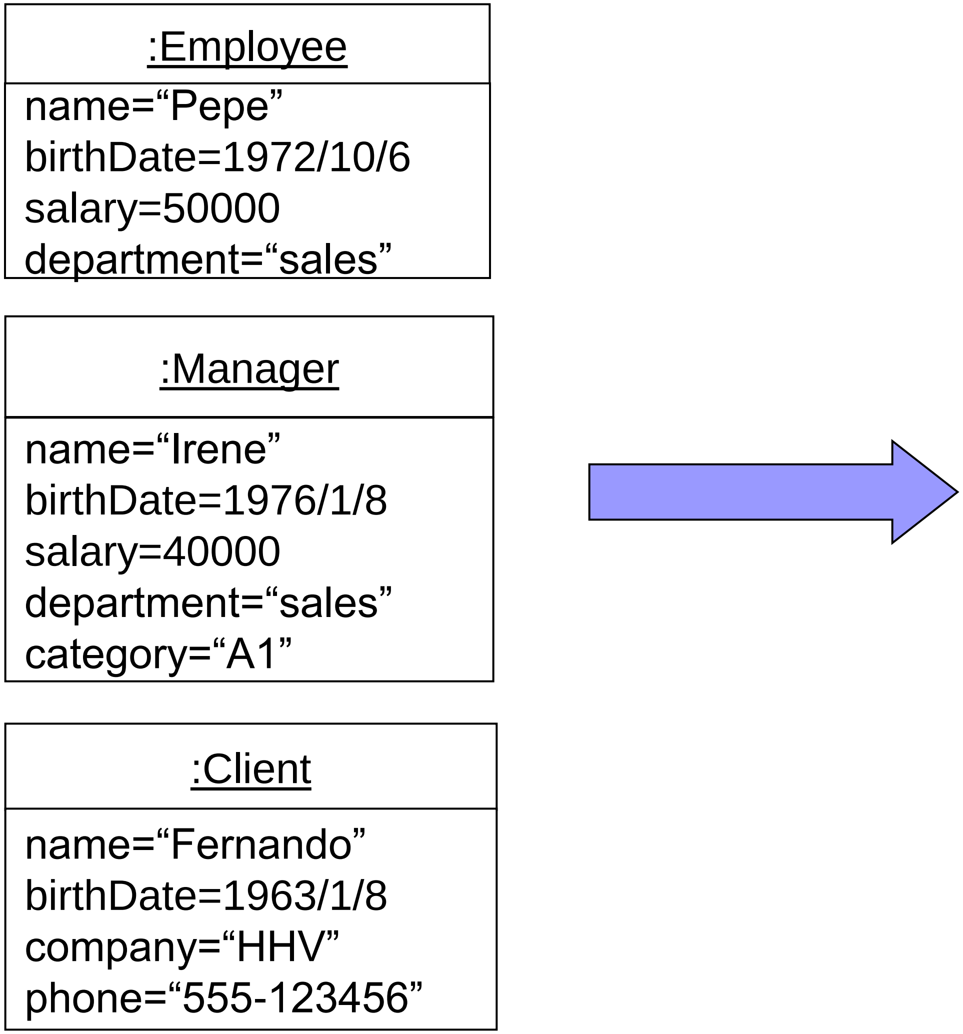
name=“Fernando”
birthDate=1963/1/8
company=“HHV”
phone=“555-123456”

showData()

[click-me]

Example

Behaviour execution



:Employee

name="Pepe"
birthDate=1972/10/6
salary=50000
department="sales"

:Manager

name="Irene"
birthDate=1976/1/8
salary=40000
department="sales"
category="A1"

:Client

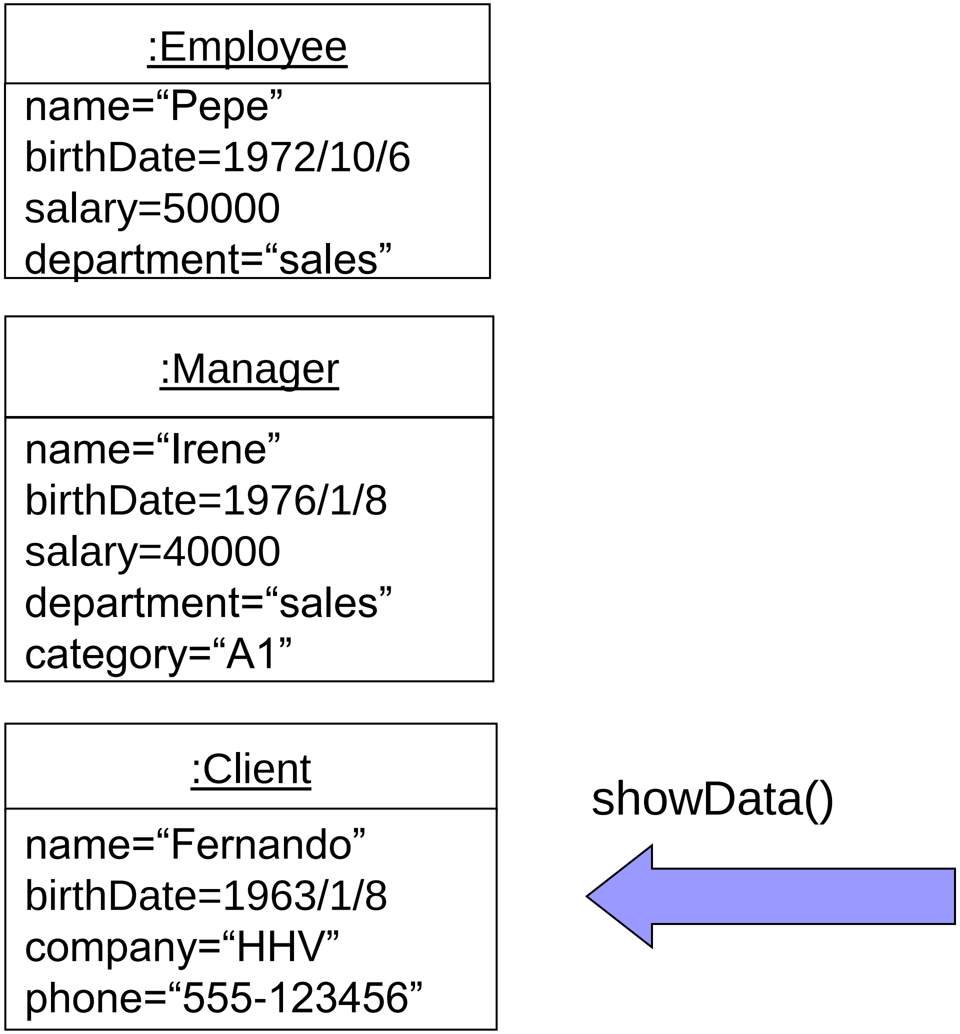
name="Fernando"
birthDate=1963/1/8
company="HHV"
phone="555-123456"

Name: Irene
Birthdate: 8/01/76
Salary: 40000€
Department: ventas
Category: A1

[click-me]

Example

Behaviour execution



:Employee

name="Pepe"
birthDate=1972/10/6
salary=50000
department="sales"

:Manager

name="Irene"
birthDate=1976/1/8
salary=40000
department="sales"
category="A1"

:Client

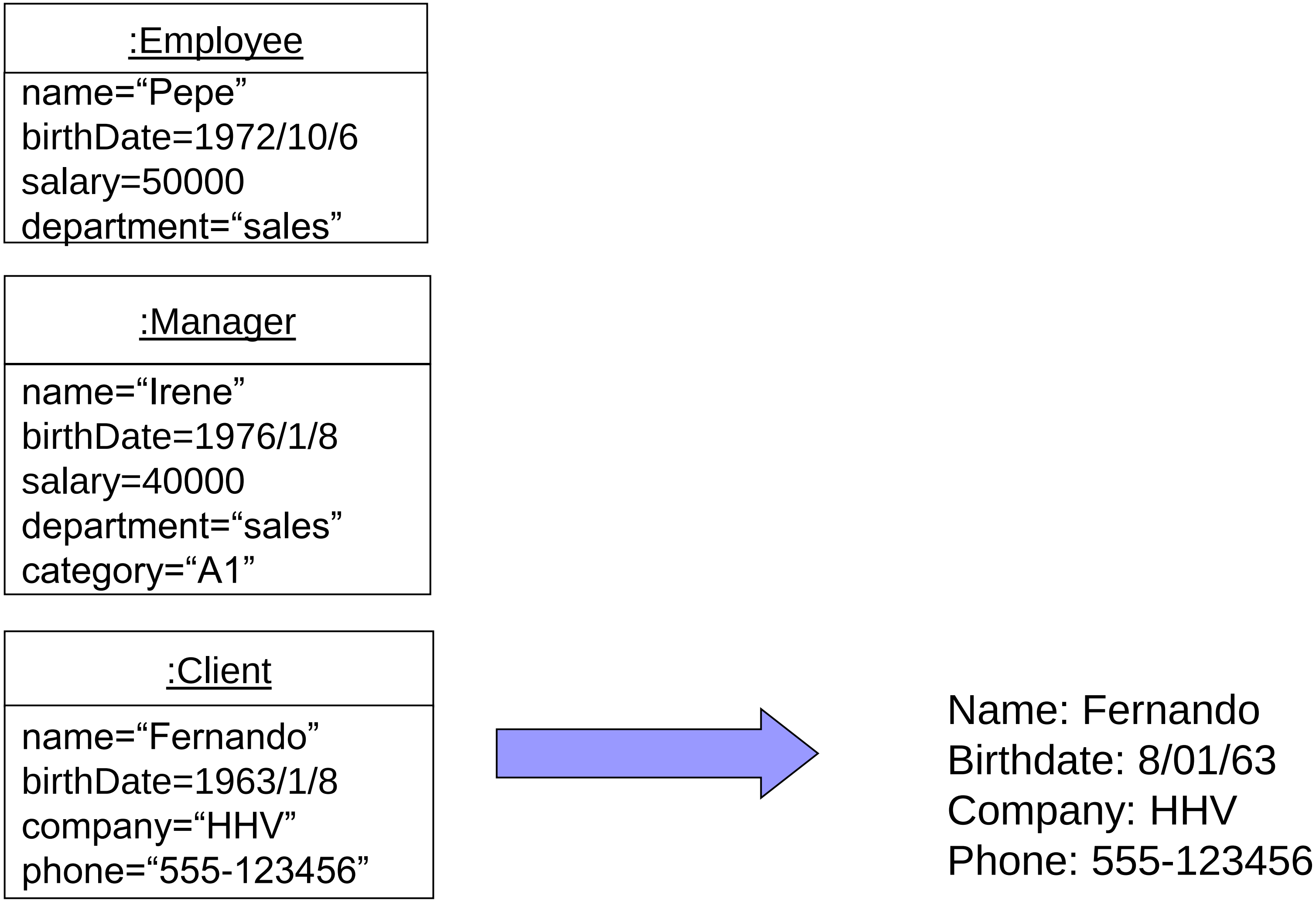
name="Fernando"
birthDate=1963/1/8
company="HHV"
phone="555-123456"

showData()

[click-me]

Example

Behaviour execution



:Employee

name="Pepe"
birthDate=1972/10/6
salary=50000
department="sales"

:Manager

name="Irene"
birthDate=1976/1/8
salary=40000
department="sales"
category="A1"

:Client

name="Fernando"
birthDate=1963/1/8
company="HHV"
phone="555-123456"

Name: Fernando
Birthdate: 8/01/63
Company: HHV
Phone: 555-123456

Object Orientation

Advantages

- Models real-life concepts in a natural way

- Design extensibility
  - By means of inheritance: add new classes, extend method behaviour
  - By means of encapsulation: the user of a class does not see or deals with unnecessary details

- Promotes reuse

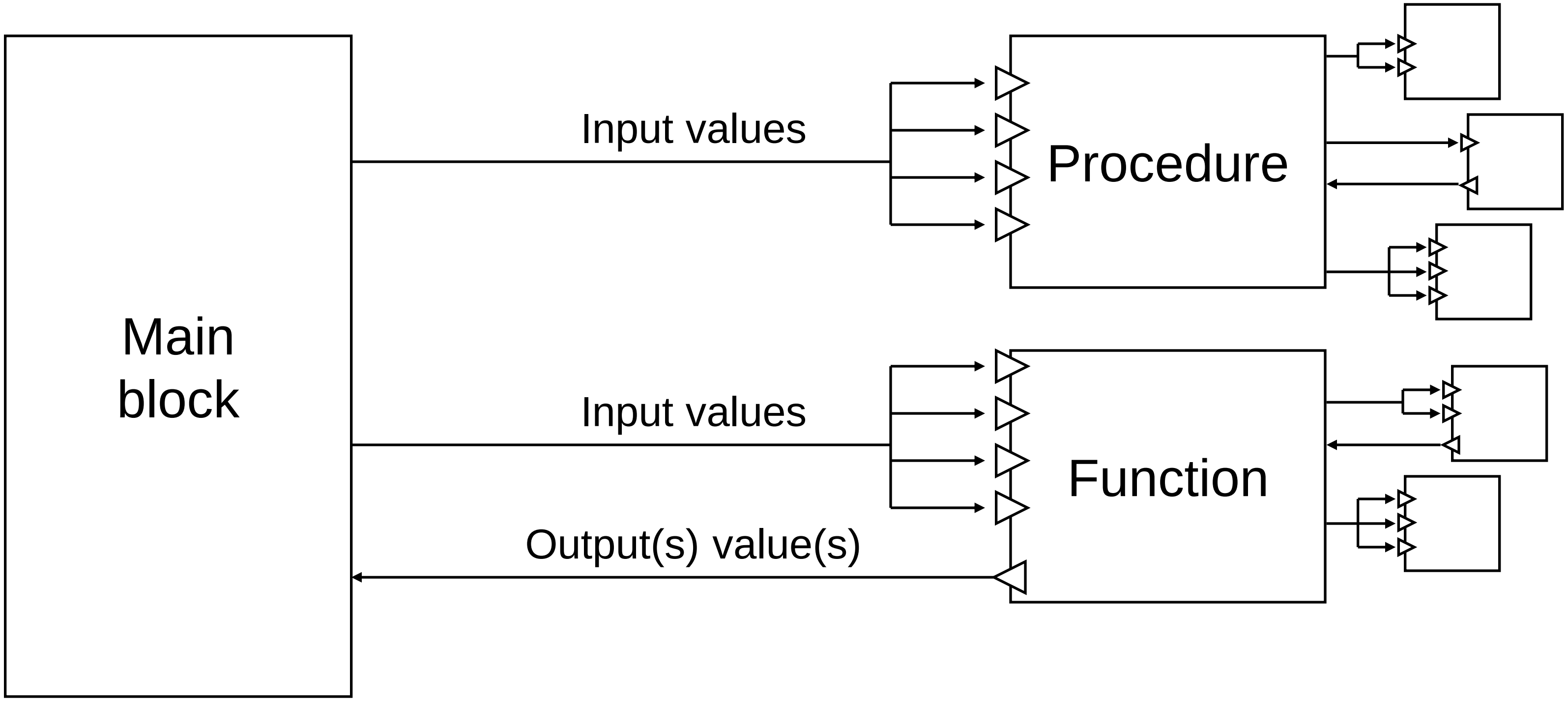
Index

- **Object oriented design concepts**
    - **Comparison with structured programming**
- Objects and classes
- Encapsulation
- Inheritance and polymorphism
- Summary and conclusions

Structured programming

- **structured program theorem:** “*sequencing, selection, and iteration—are sufficient to express any computable function*”
- Functions/procedures
  - Abstractions to organize, and reuse code
  - Featured by languages like Pascal or C
- Separation of algorithms and data structures
- Program: made of procedures and invocations among them.
  - “Top-down” design

Structured design



Structured Programming

Operations as abstractions

- Structure of a module:
  - □ Interface
    - Input data
    - Output data
    - Description of functionality
- Language syntax
  - □ Organization of the code in instruction blocks

    Definition of functions and procedures
  - □ Extension of the program “*vocabulary*” with new operations

    Call to new functions and procedures

Structured Programming

Advantages

- Facilitates software development
  - Avoids code repetition
  - Splits programming tasks in independent modules
    - Can be designed separately, by (independent) developers
  - Top-down design: “divide-and-conquer”
- Facilitates maintenance
  - Code clarity
  - Module independence
- Promotes reuse

Structured programming

Example in C

```
void main ()
{
  double u1, u2, m;
  u1 = 4;
  u2 = -2;
  m = sqrt (u1*u1 + u2*u2);
  printf (”%lf”, m);
}
```

```
double module (double u1, double u2)
{
  double m;
  m = sqrt (u1*u1 + u2*u2);
  return m;
}

void main ()
{
 printf (”%lf”, module (4, -2));
}
```

Abstract data types

Data and operation abstractions

- An abstract data type (ADT) is made of
  - **Data structure** storing information to represent a given concept
  - **Functionality** set of operations performed over those data

- Language syntax
  - Modules associated to data types
  - It does not necessarily introduce variations over modular programming

Abstract data types

Example

```
struct vector {
  double x;
  double y;
};

void make (vector *u, double u1, double u2)
{
  u->x = u1;
  u->y = u2;
}

double modulus (vector u)
{
  double m;
  m = sqrt (u.x*u.x + u.y*u.y);
  return m;
}
```

```
void main ()
{
  vector u;
  make (&u, 4, -2);
  printf (”%lf”, modulus (u));
}
```

Abstract data types

Extensibility

```
...

double product (vector u, vector v)
{
  return u.x * v.x + u.y * v.y;
}

void main ()
{
  vector u, v;
  make (&u, 4, -2);
  make (&v, 1, 5);
  printf (”%lf”, product (u, v));
}
```

Abstract data types

Advantages

- Domain concepts reflected in the code
- Encapsulation: hiding internal complexity, operation and data details
- Specification vs implementation: use of the data type independently of its internal programming details
- Better modularity: also applies to data
- Facilitates maintenance and reuse

Object oriented programming

Object oriented programming

=

syntactic support for abstract data types

+

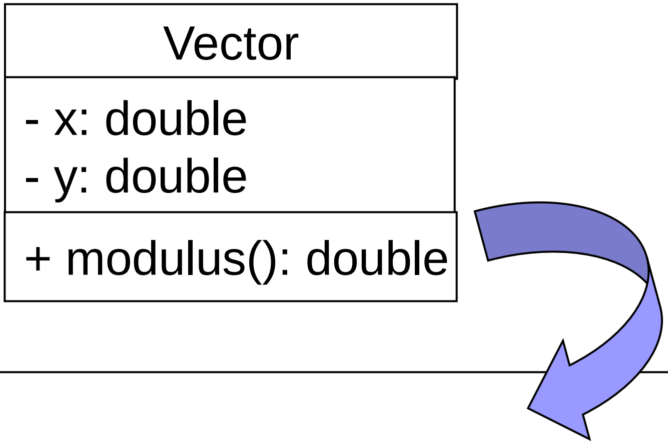
features associated to class hierarchies

+

change of perspective

Object oriented programming

Example



Vector

- x: double
- y: double

+ modulus(): double

[click-me]

```
class Vector {
  private double x;
  private double y;
  public Vector (double u1, double u2) { x = u1; y = u2; }
  public double modulus () { return Math.sqrt (x*x + y*y); }
}

class MainClass {
  public static void main (String args []) {
    Vector u = new Vector (4, -2);
    System.out.println (u.modulus ());
  }
}
```

Index

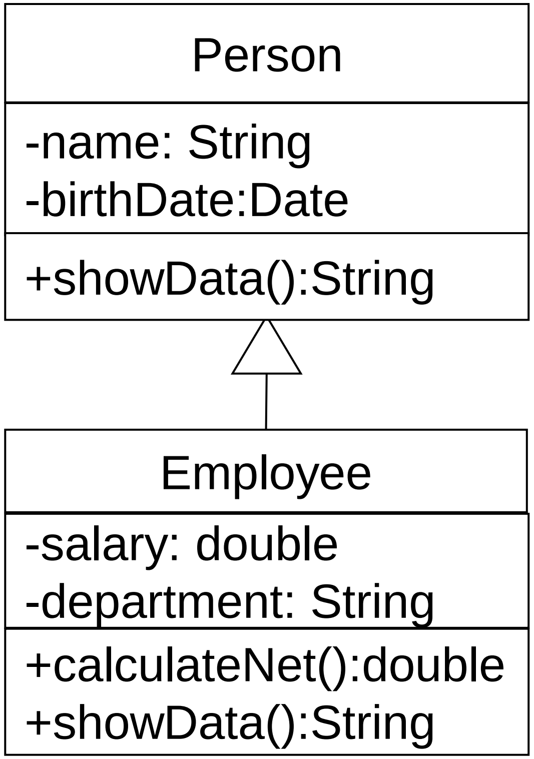
- Object oriented design concepts
- **Objects and classes**
- Encapsulation
- Inheritance and polymorphism
- Summary and conclusions

Elements of Object-Oriented Programming

- **Objets**: attributes (slots) + methods
- **Methods**: operations within objects
- **Classes**: object categories with common properties and operations
- **Inheritance**: class hierarchies
- **Relations** between objects. Composite objects

Structure of classes and objects

- **Class**: template to create objects
- **Object**: has values for the attributes defined in the class. Reacts to the methods defined in the class



Person

-name: String
-birthDate:Date

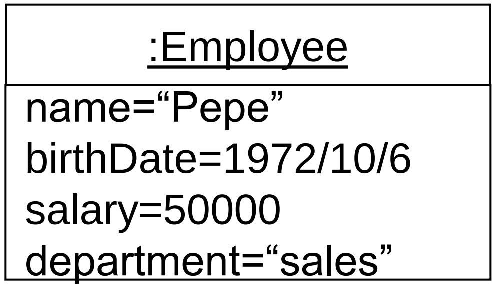
+showData():String

Employee

-salary: double
-department: String

+calculateNet():double
+showData():String

Class



:Employee

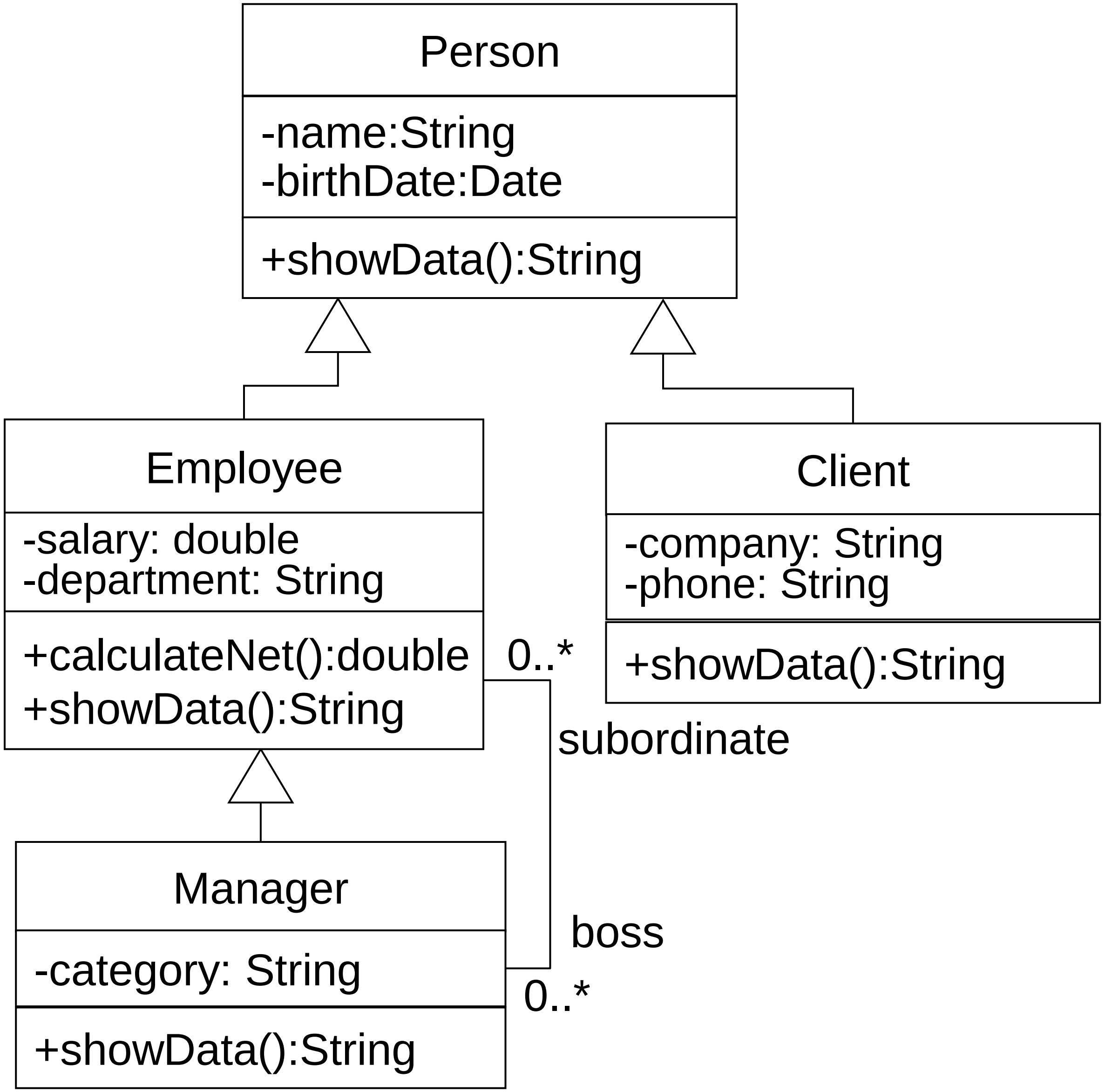
name="Pepe"
birthDate=1972/10/6
salary=50000
department="sales"

Object

Object life-cycle

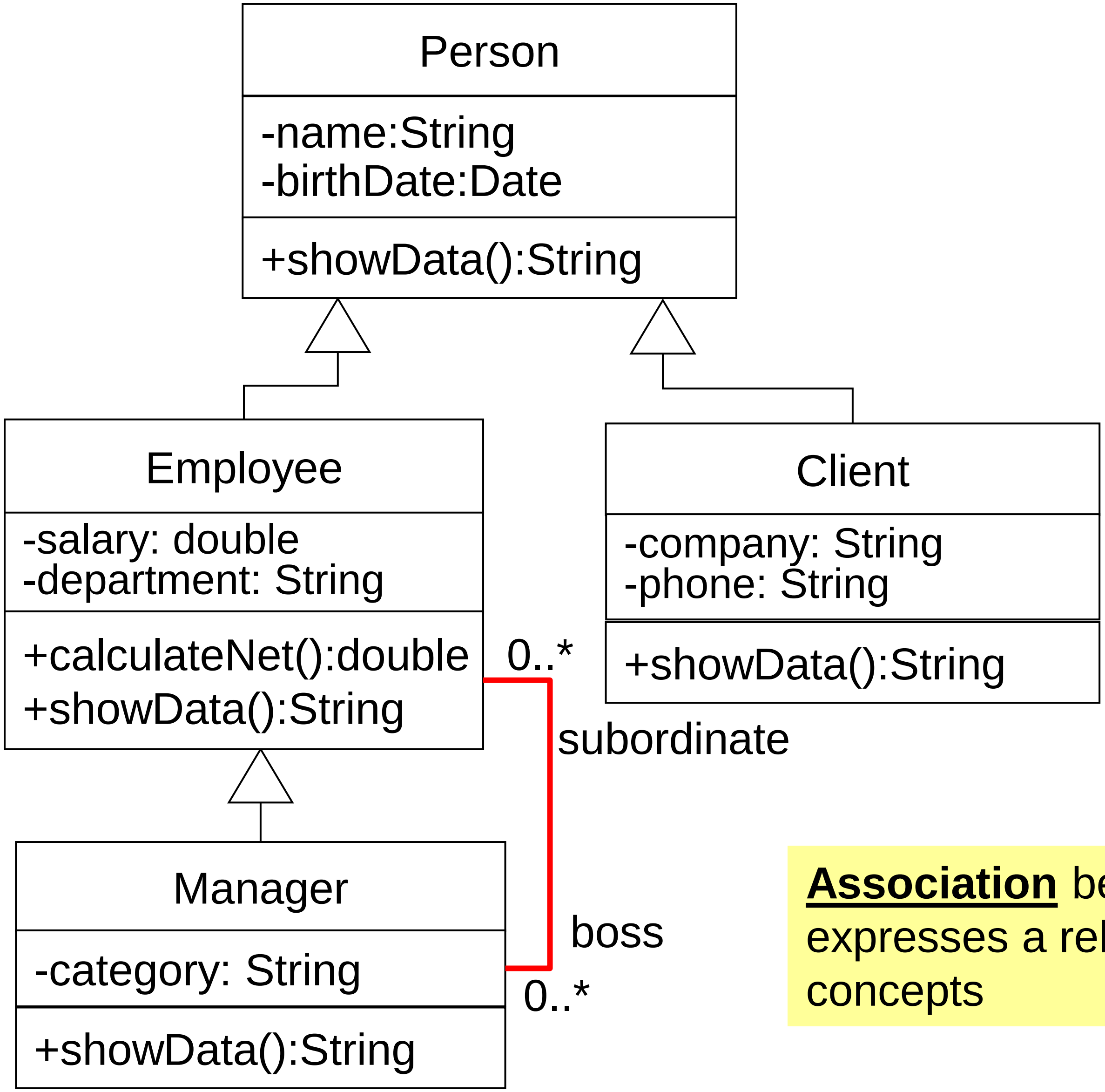
- **Creation**
  - □ Memory allocation: Employee x = new Employee (···)
  - □ Attribute initialization
    - Called “***constructor***”.
- **Manipulation**
  - □ Access to attributes: x.name
  - □ Method invocation: x.calculateNet ( )
- **Destruction**
  - □ Free memory
  - □ Destroy internal parts, if any
  - □ Eliminate references of the destroyed object (e.g. boss)
    - Called “***destructor***”.
    - Depending on the language, the destructor call can be implicit (e.g.: Java, local objects in C++, JavaScript)

Structure of classes and objects



- Relations with other objects
  - Association
  - Aggregation
  - Containment

Structure of classes and objects

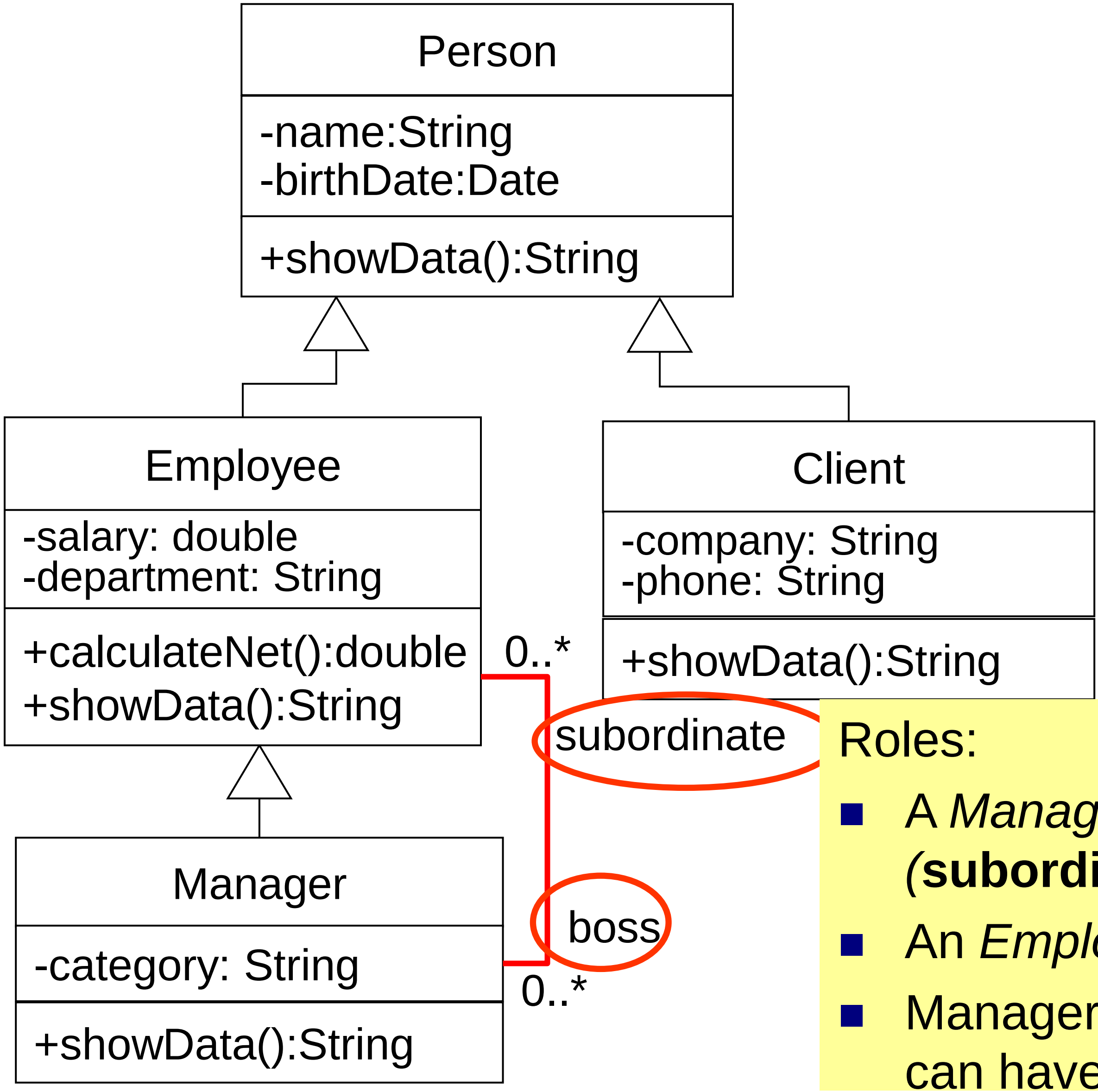


- Relations with other objects
  - □ **Association**
  - □ Aggregation
  - □ Containment

Association between two classes: expresses a relationship between to concepts

Structure of classes and objects

- Relations with other objects
  - □ **Association**
  - □ Aggregation
  - □ Containment



Person

-name:String
-birthDate:Date

+showData():String

Employee

-salary: double
-department: String

+calculateNet():double
+showData():String

Client

-company: String
-phone: String

+showData():String

Manager

-category: String

+showData():String

0..*

subordinate

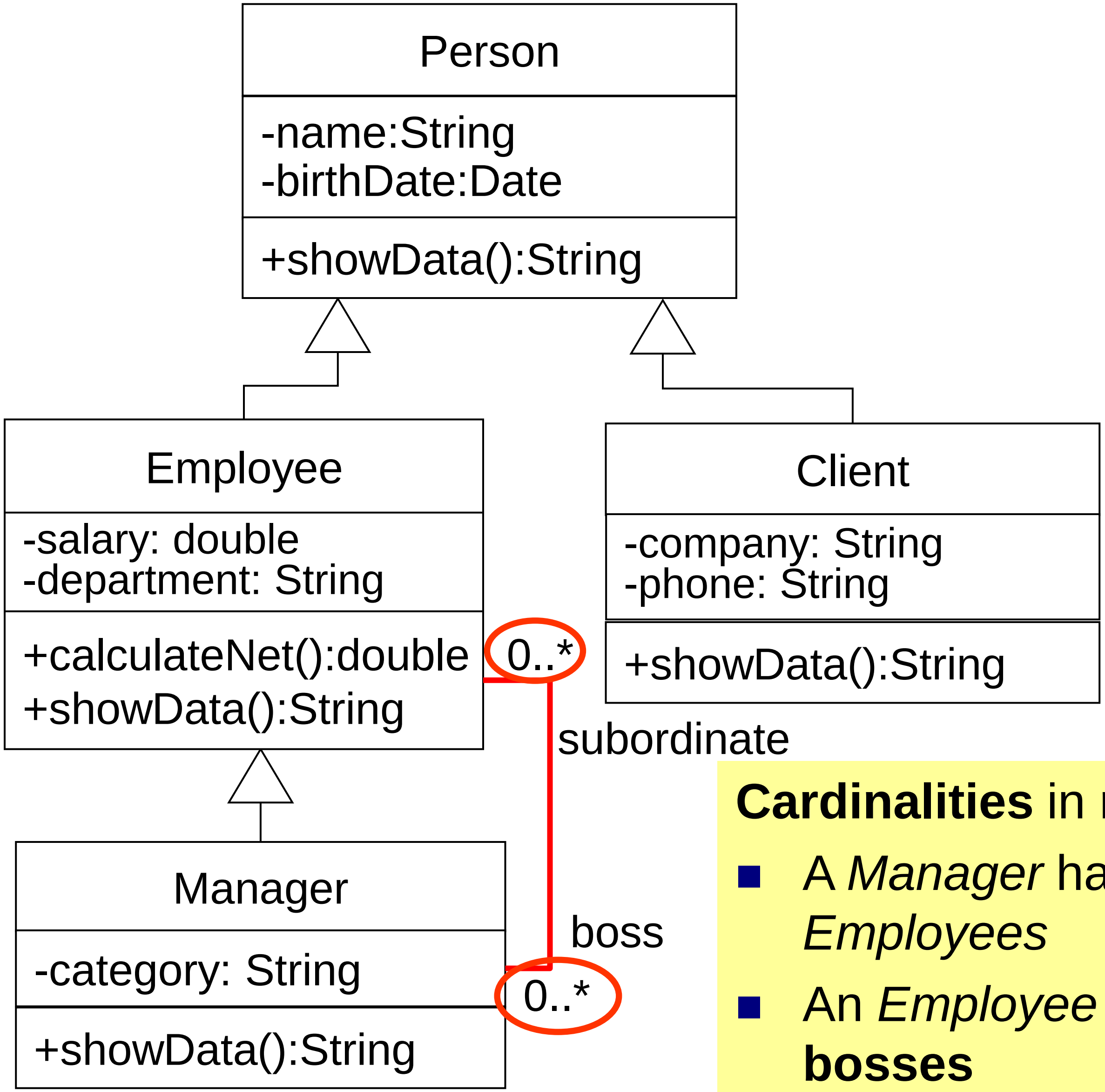
boss

0..*

Roles:

- A *Manager* has *Employees* **(subordinates)**
- An *Employee* has *Manager* **bosses**.
- Managers are Employees, and they can have bosses

Structure of classes and objects



Person

-name:String
-birthDate:Date

+showData():String

Employee

-salary: double
-department: String

+calculateNet():double
+showData():String

Client

-company: String
-phone: String

+showData():String

Manager

-category: String

+showData():String

0..*

subordinate

boss

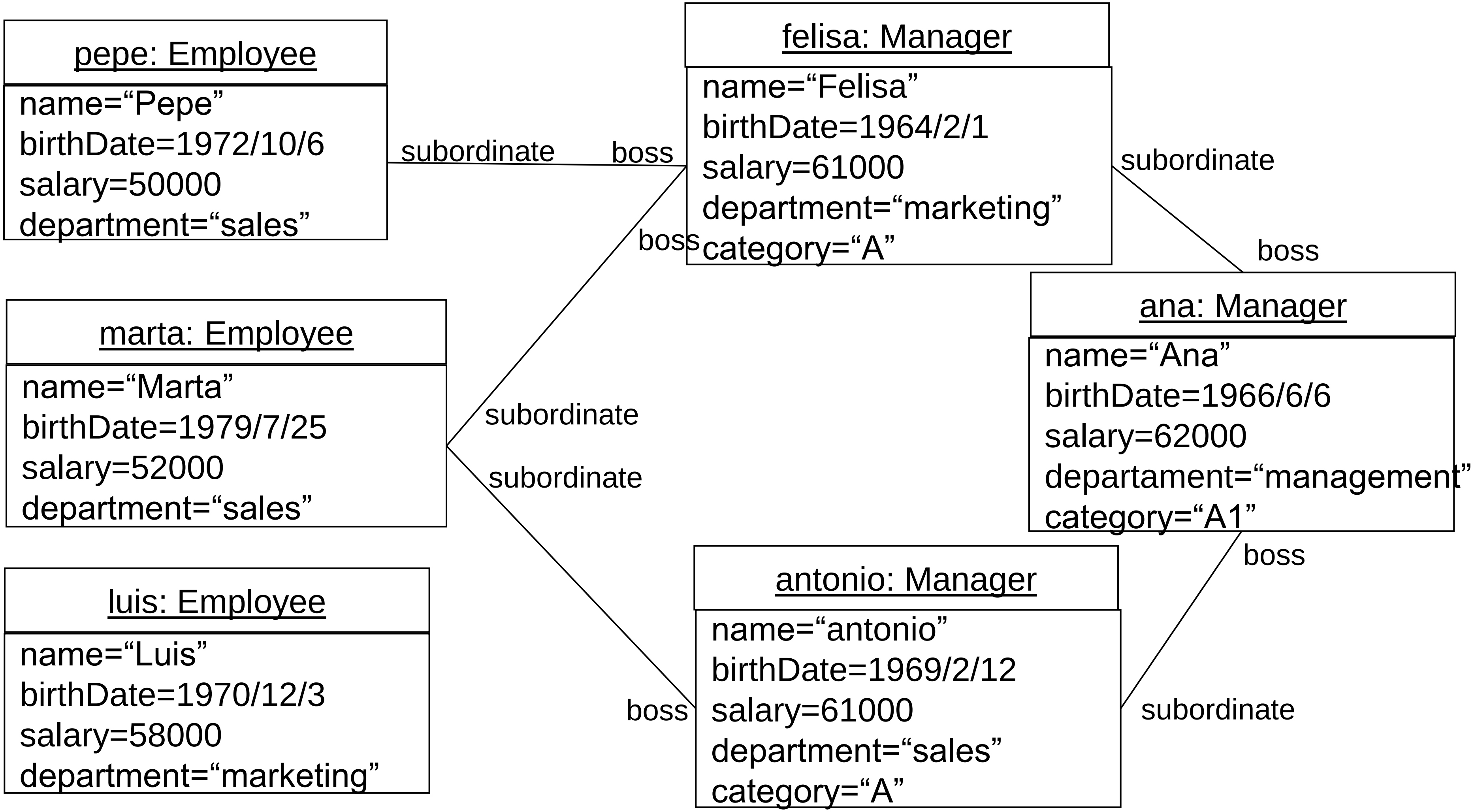
0..*

- Relations with other objects
  - □ **Association**
  - □ Aggregation
  - □ Containment

Cardinalities in roles:

- A *Manager* has zero or more **subordinate** *Employees*
- An *Employee* has zero or more *Manager* **bosses**

Structure of classes and objects



pepe: Employee

name="Pepe"
birthDate=1972/10/6
salary=50000
department="sales"

felisa: Manager

name="Felisa"
birthDate=1964/2/1
salary=61000
department="marketing"
category="A"

marta: Employee

name="Marta"
birthDate=1979/7/25
salary=52000
department="sales"

ana: Manager

name="Ana"
birthDate=1966/6/6
salary=62000
departament="management"
category="A1"

luis: Employee

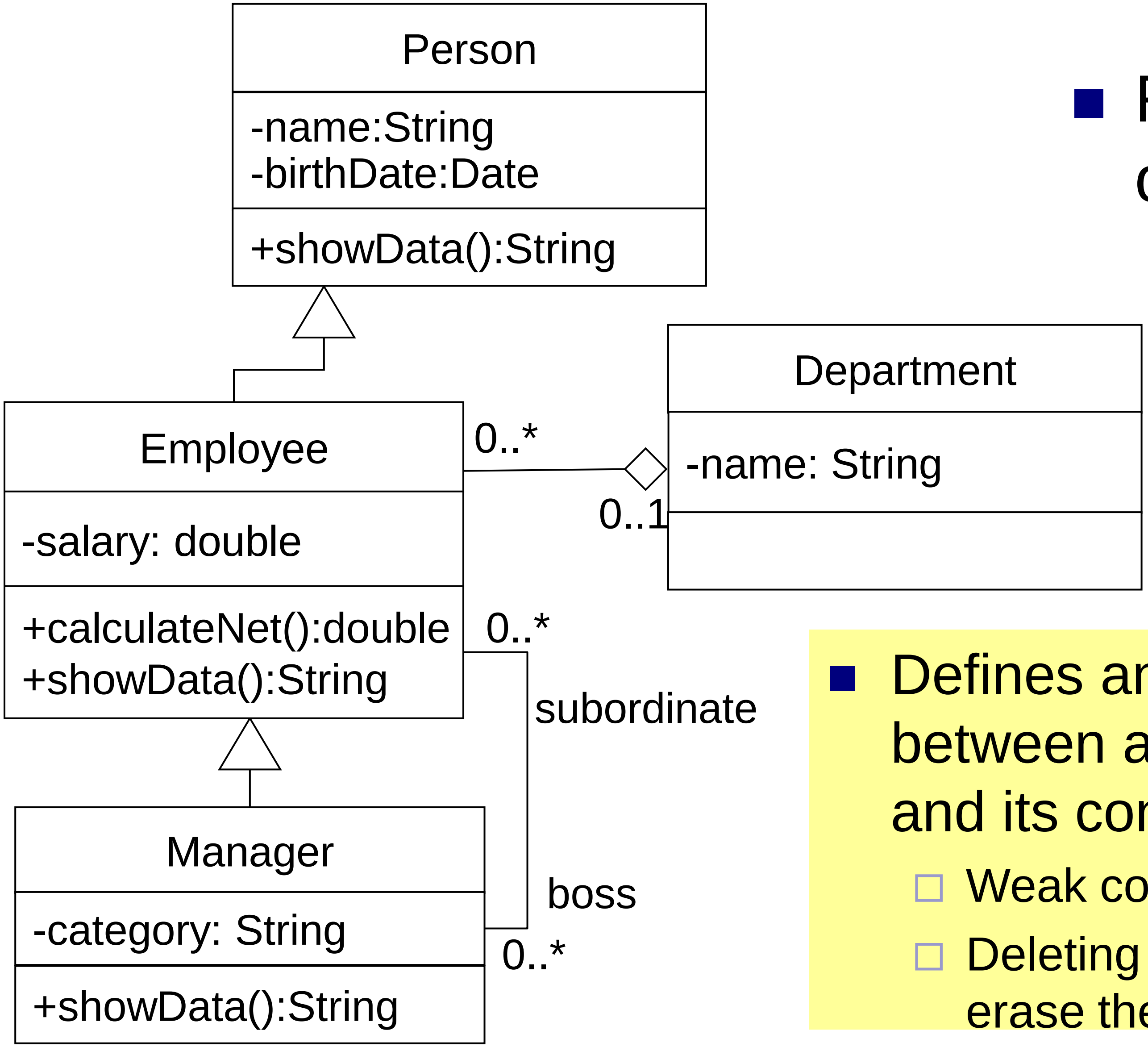
name="Luis"
birthDate=1970/12/3
salary=58000
department="marketing"

antonio: Manager

name="antonio"
birthDate=1969/2/12
salary=61000
department="sales"
category="A"

Do cardinalities hold?

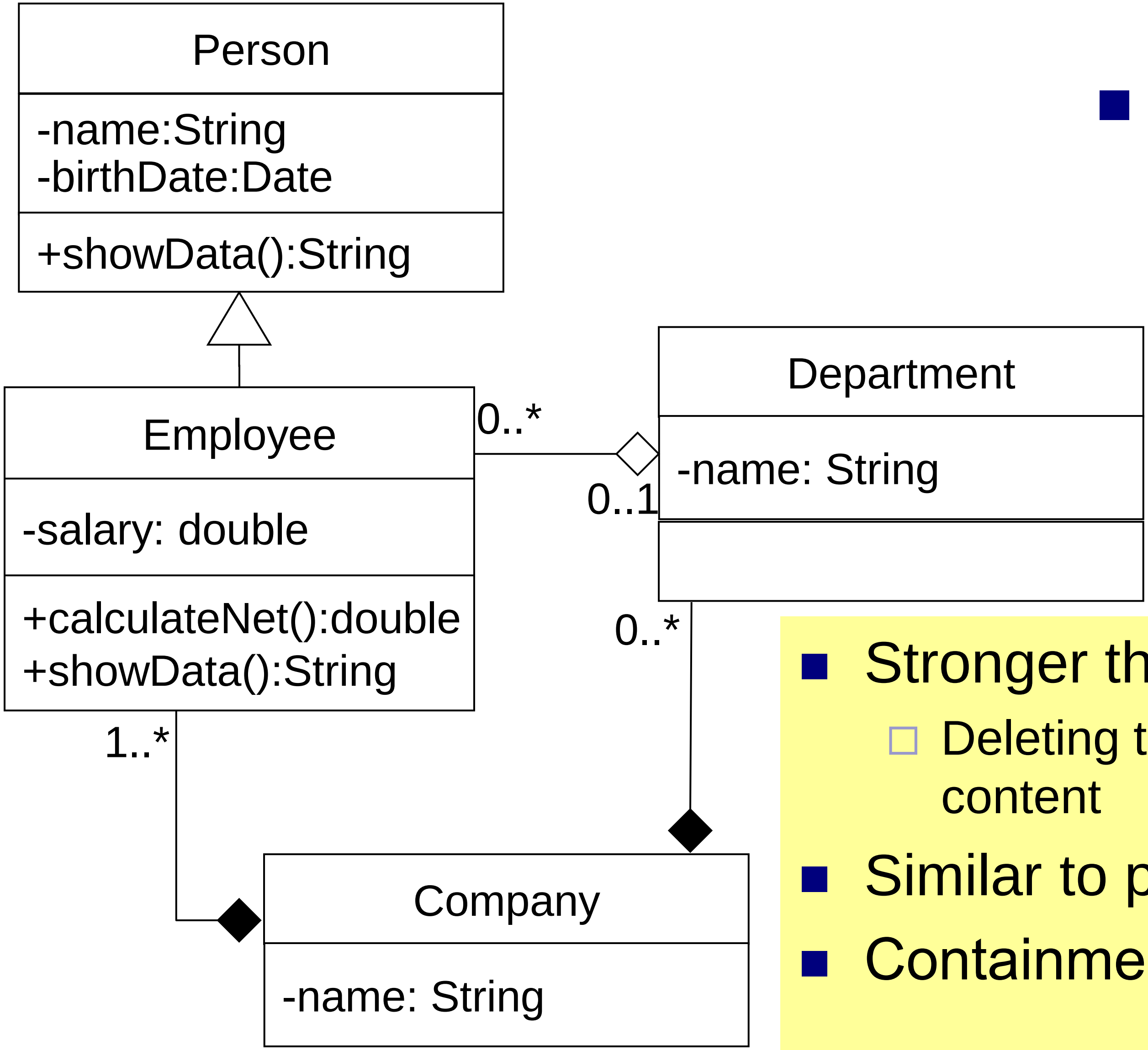
Structure of classes and objects



- Relations with other objects
  - Association
  - **Aggregation**
  - Containment

- Defines an association between a container object and its contained objects
  - Weak containment
  - Deleting the container does not erase the containee objects

Structure of classes and objects

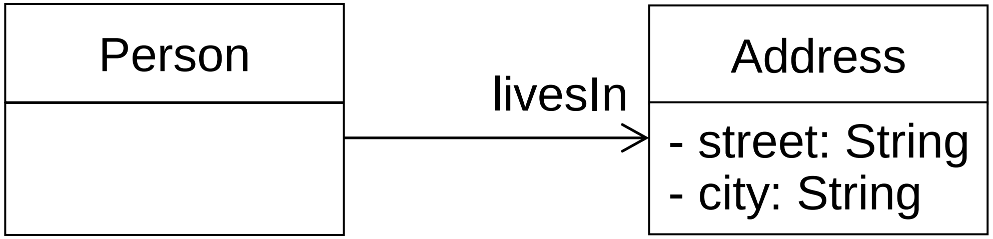


- Relations with other objects.
  - □ Association
  - □ Aggregation
  - □ **Containment**

- Stronger than aggregation
  - □ Deleting the container removes the content
- Similar to physical containment
- Containment = “*is made of*”.

Navigation

- Indicates whether from a class we can navigate to the other via the association
- Displayed with an arrow

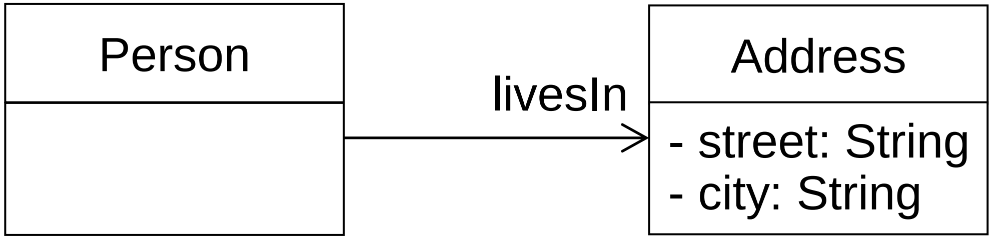


```
class Person {
  private Address livesIn;
}
```

```
class Address {
  private String street;
  private String city;
}
```

Navigation

- Indicates whether from a class we can navigate to the other via the association
- Displayed with an arrow



Person

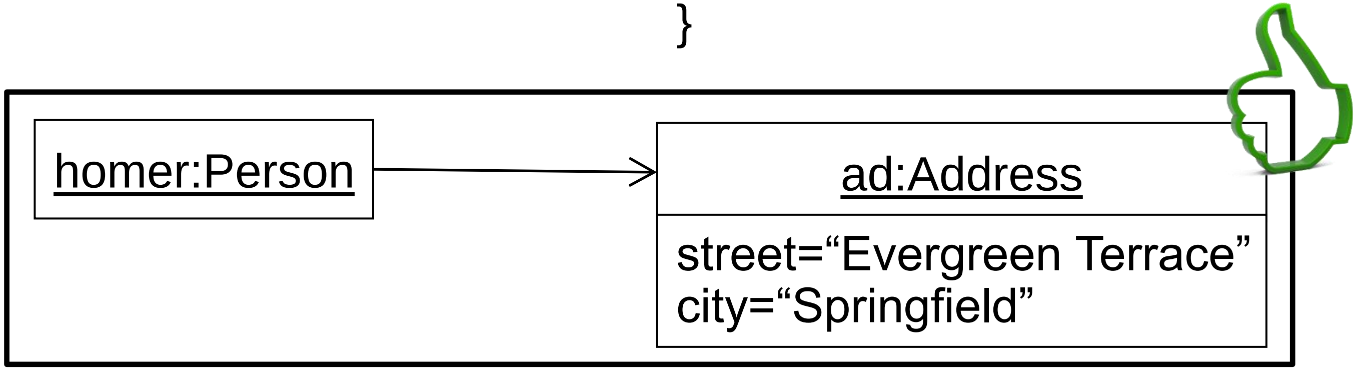
livesIn

Address

- street: String
- city: String

```
class Person {
  private Address livesIn;
}
```

```
class Address {
  private String street;
  private String city;
}
```



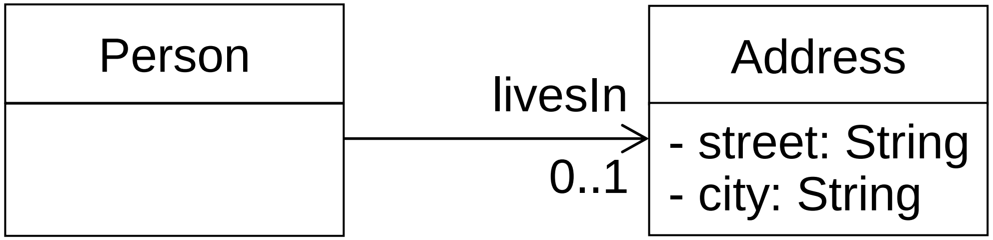
homer:Person

ad:Address

street=“Evergreen Terrace”
city=“Springfield”

Navigation

- Indicates whether from a class we can navigate to the other via the association
- Displayed with an arrow



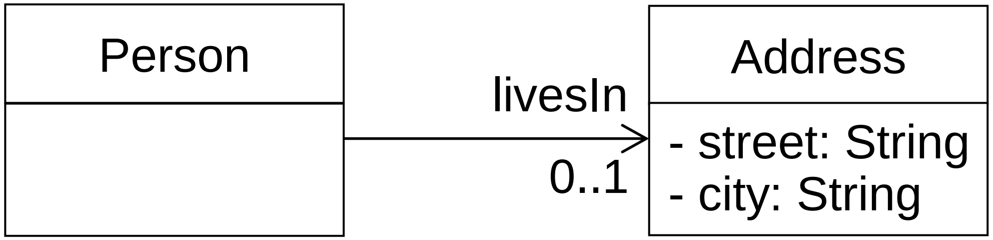
```
class Person {
  private Address livesIn;
}
```

```
class Address {
  private String street;
  private String city;
}
```

But now livesIn can be null

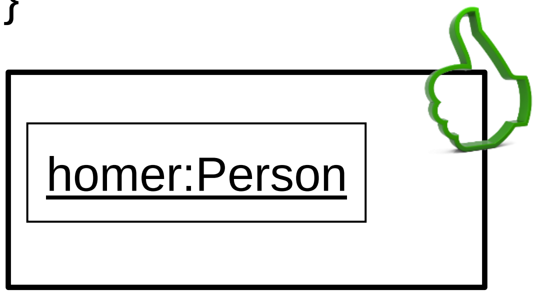
Navigation

- Indicates whether from a class we can navigate to the other via the association
- Displayed with an arrow



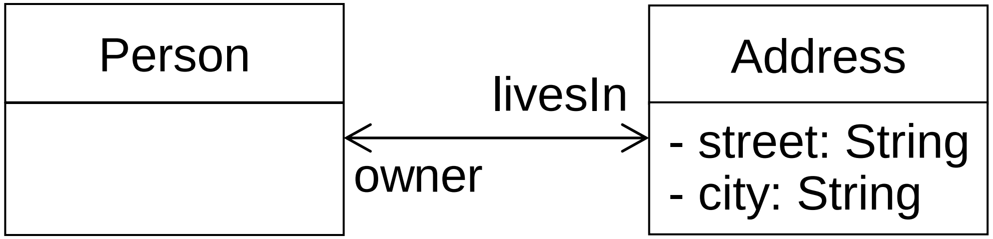
```
class Person {
  private Address livesIn;
}
```

```
class Address {
  private String street;
  private String city;
}
```



Bidirectional Navigation

- Navigation between both classes
- Equivalent to ommiting both arrows
  - □ But this may also mean that such design decision has not been made yet

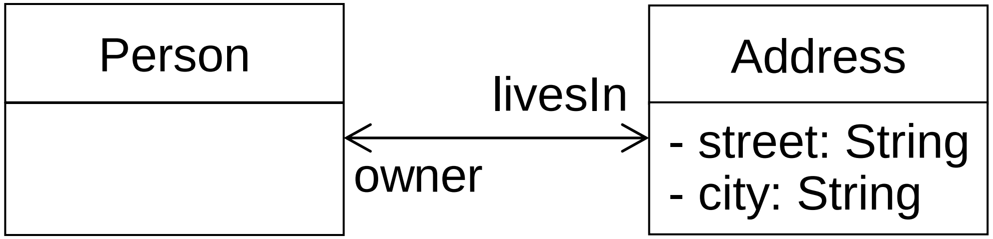


```
class Person {
  private Address livesIn;
}
```

```
class Address {
  private String street;
  private String city;
  private Person owner;
}
```

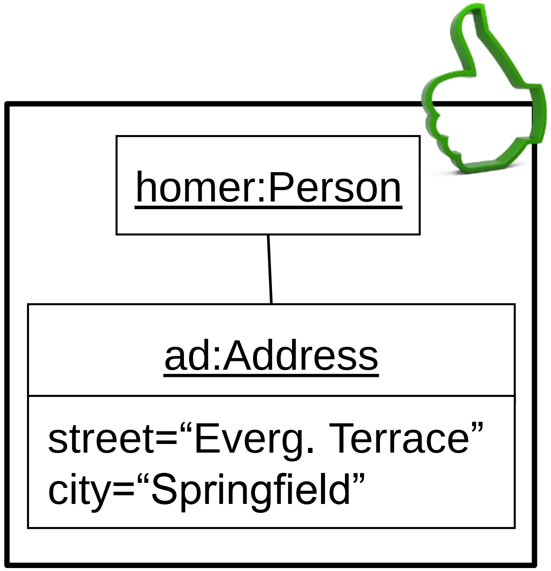
Bidirectional Navigation

- Navigation between both classes
- Equivalent to ommiting both arrows
  - But this may also mean that such design decision has not been made yet



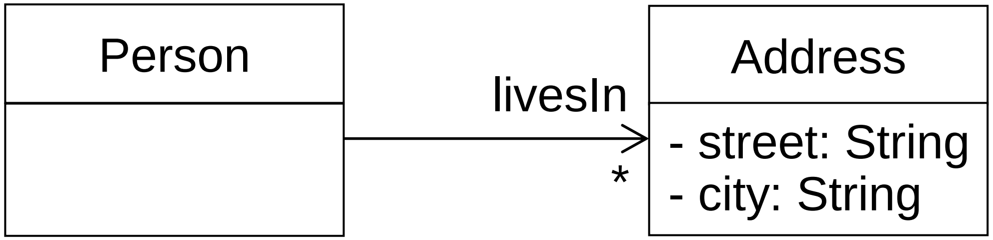
```
class Person {
  private Address livesIn;
}
```

```
class Address {
  private String street;
  private String city;
  private Person owner;
}
```



Navigation 1-to-many

- A collection on one end
- Many-to-many bi-directional navigation would create collections on both ends



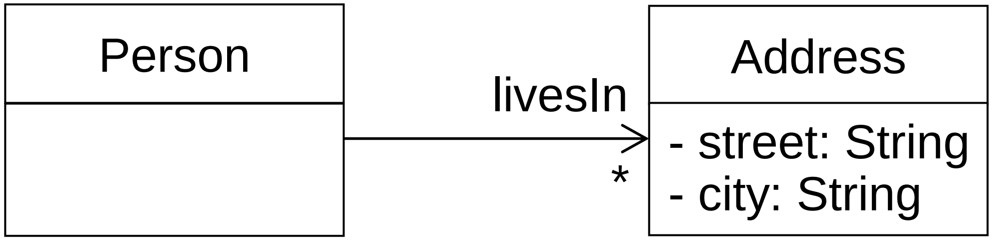
```
class Person {
  private Address[] livesIn;
}
```

```
class Address {
  private String street;
  private String city;
}
```

Or better: some of the Java Collection types (List, Set, etc)

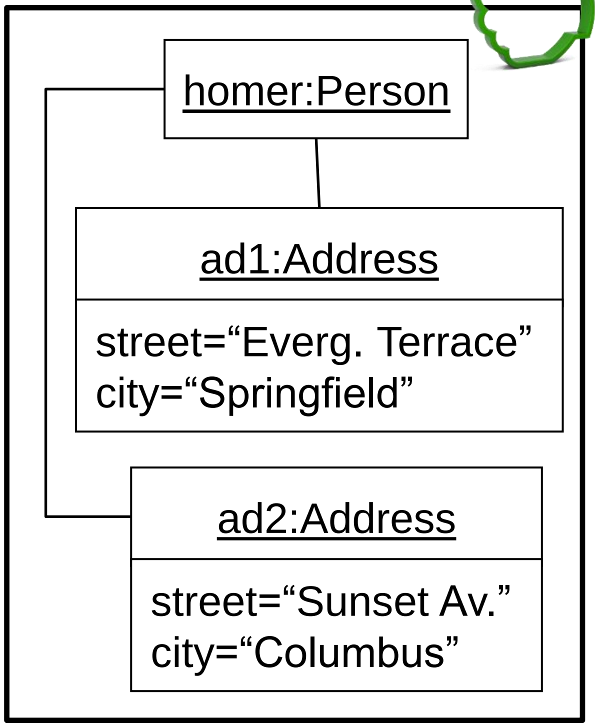
Navigation 1-to-many

- A collection on one end
- Many-to-many bi-directional navigation would create collections on both ends



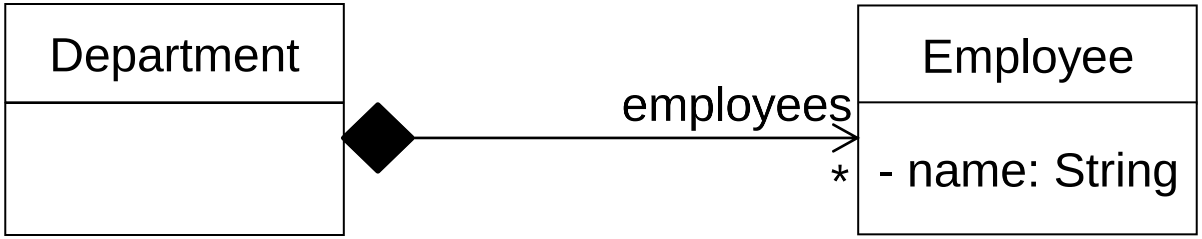
```
class Person {
  private Address[] livesIn;
}
```

```
class Address {
  private String street;
  private String city;
}
```



Composition+uni-dir navigation

- A collection on one end
- The code in the classes should take care of the composition semantics:
  - Containee does not belong to more than one container
  - Deleting the container deletes the containee

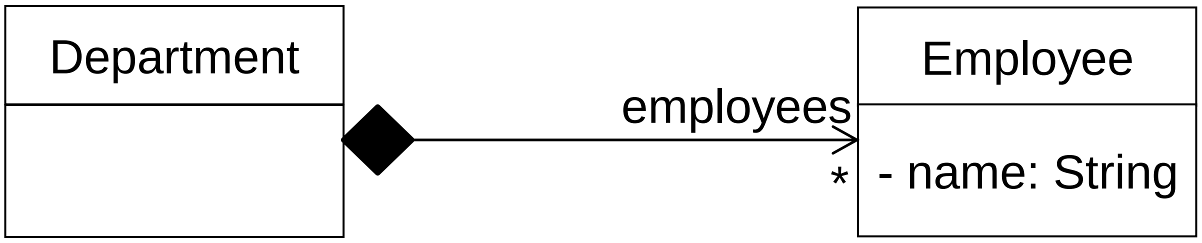


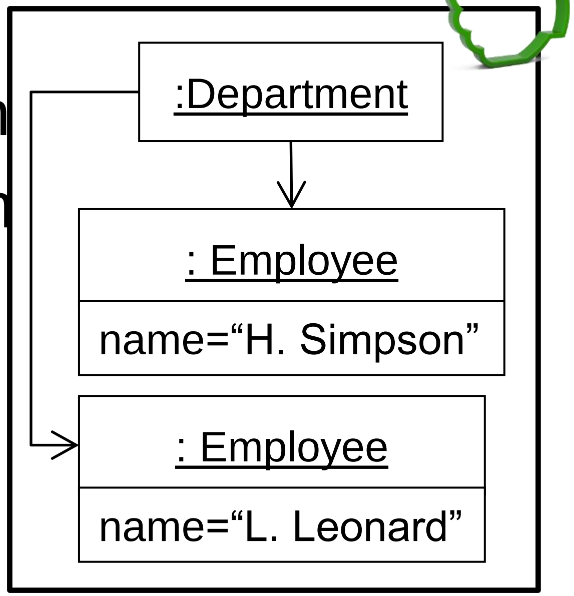
```
class Department {
  private Employee[] employees;
}
```

```
class Employee {
  private String name;
}
```

Composition+uni-dir navigation

- A collection on one end
- The code in the classes should take care of the composition semantics:
  - □ Containee does not belong to more th
  - □ Deleting the container deletes the con



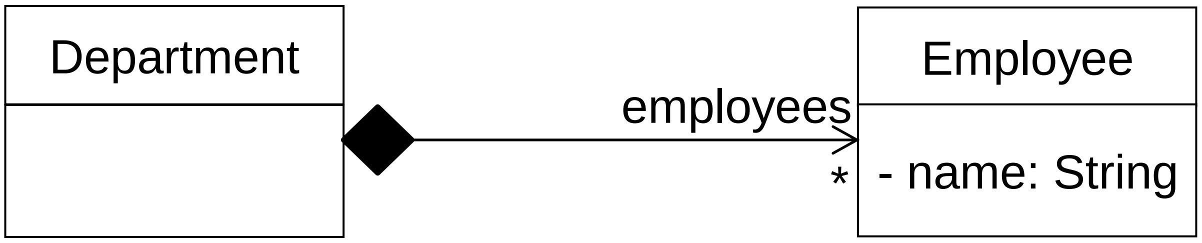


```
class Department {
  private Employee[] employees;
}
```

```
class Employee {
  private String name;
}
```

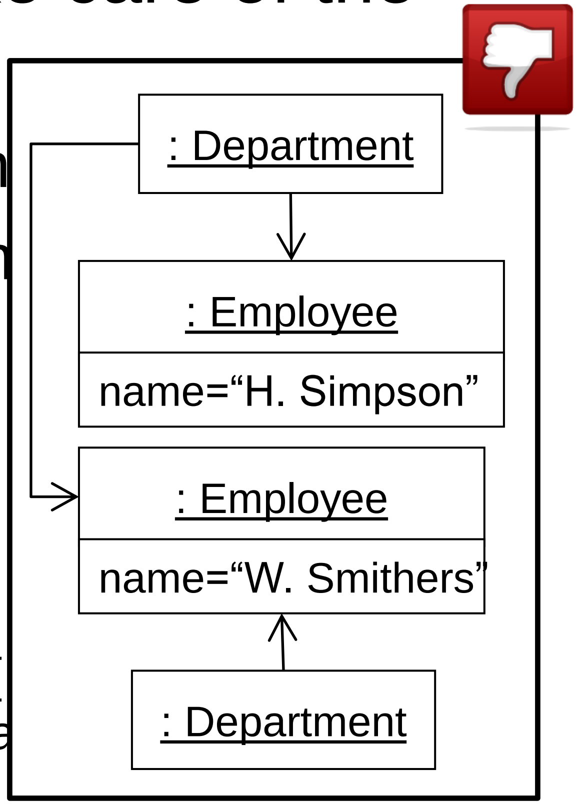
Composition+uni-dir navigation

- A collection on one end
- The code in the classes should take care of the composition semantics:
  - □ Containee does not belong to more th
  - □ Deleting the container deletes the con



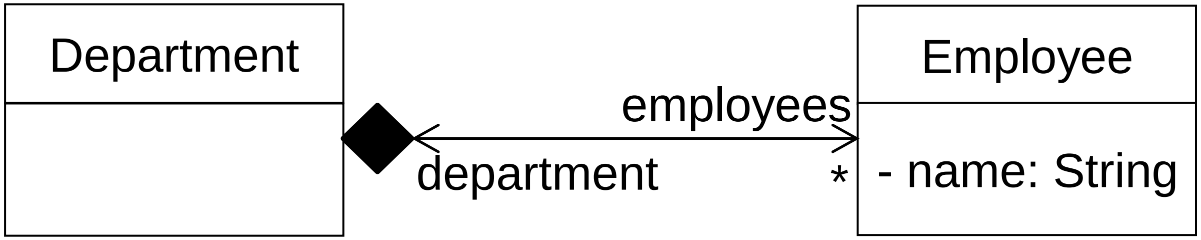
```
class Department {
  private Employee[] employees;
}
```

```
class Employee {
  private String na
}
```



Composition+bi-dir navigation

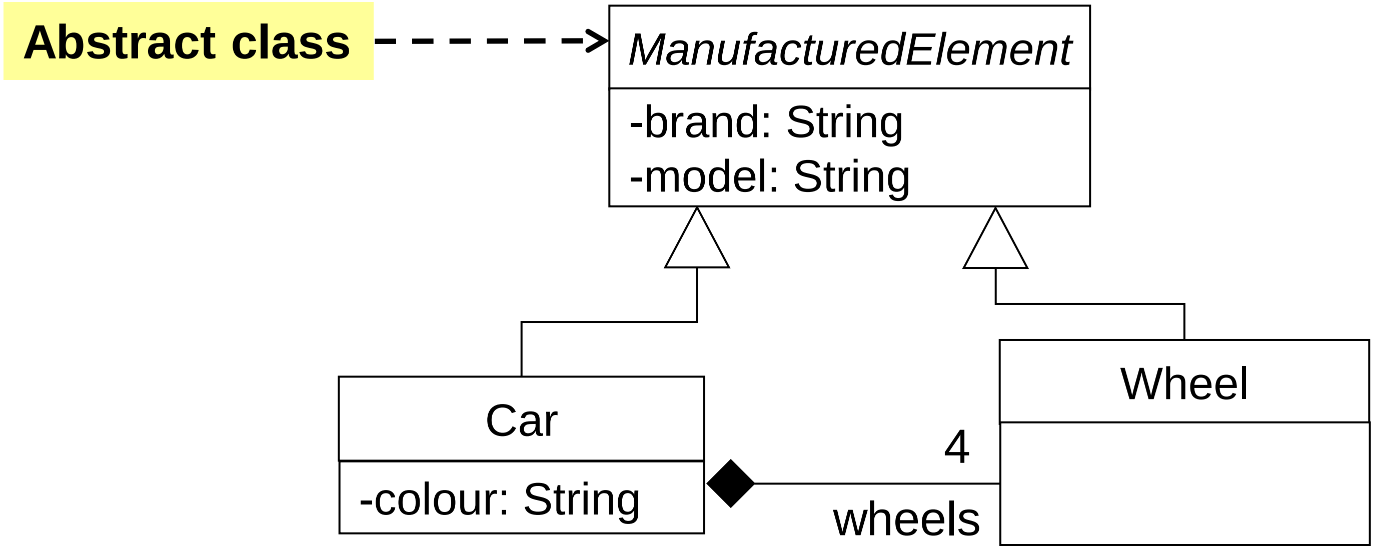
- A collection on one end, and a reference on the other



```
class Department {
  private Employee[] employees;
}
```

```
class Employee {
  private String name;
  private Department department;
}
```

Structure of classes and objects



- Abstract class: We cannot instantiate it (we cannot create objects)

- Used to specify data and behaviour common to several subclasses

Index

- **Object oriented design concepts**
- Objects and classes
- **Encapsulation**
- Inheritance and polymorphism
- Summary and conclusions

Encapsulation

- Control the external access to methods and attributes
  - **Private elements (-)**: not accessible or visible from outside
    - A private method cannot be invoked from an object of a different type
  - **Public elements (+)**: accessible from outside
    - A public method can be invoked from the same and other objects
  - **Protected elements (#)**: accessible from the class and subclasses

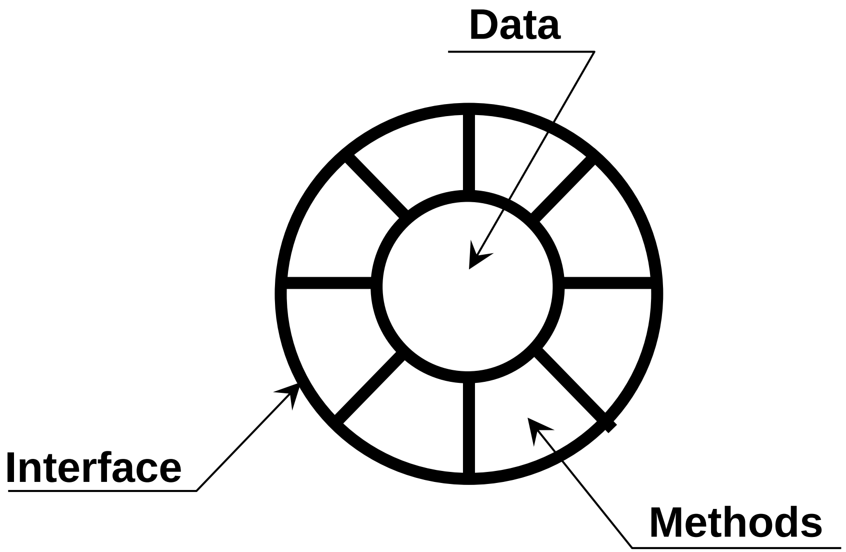
- **Encapsulation**: we only expose the relevant interface to the rest of the system

- **Information hiding:** facilitates design, making it simpler and extensible

Encapsulation

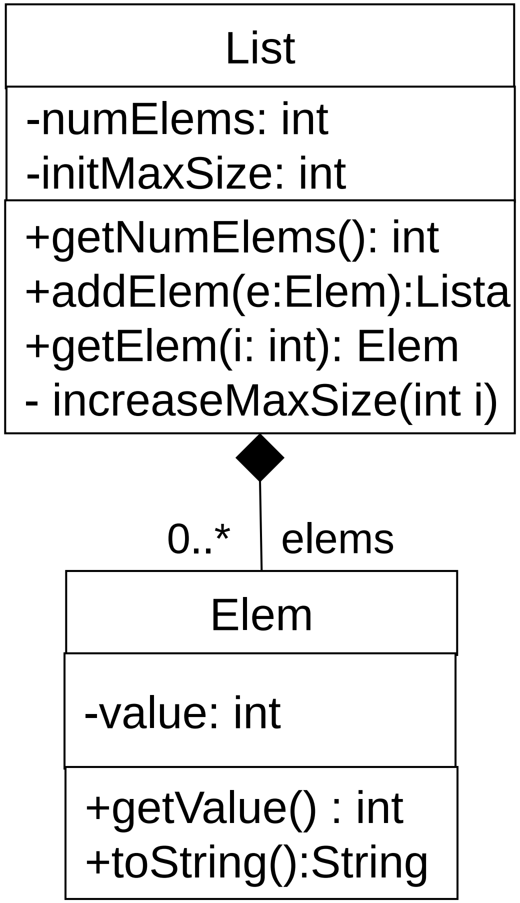
- Normally **all** attributes of a class are declared **private**
- Constructors initialize the class attributes
- Accessor (*getters*) and mutator (*setters*) methods are declared **if needed**
  - Not all attributes require getters and setters
  - Some getters may rely on computation (“*derived attributes*”)
- A public attribute would be equivalent to a global variable, regarding its access level
  - Bad design, complicates understanding who accesses and changes the variable
  - Complicates testing and debugging

Encapsulation



- Data are protected (not visible)
- The access to the object state is done via the methods in the interface

Encapsulation



[click-me]

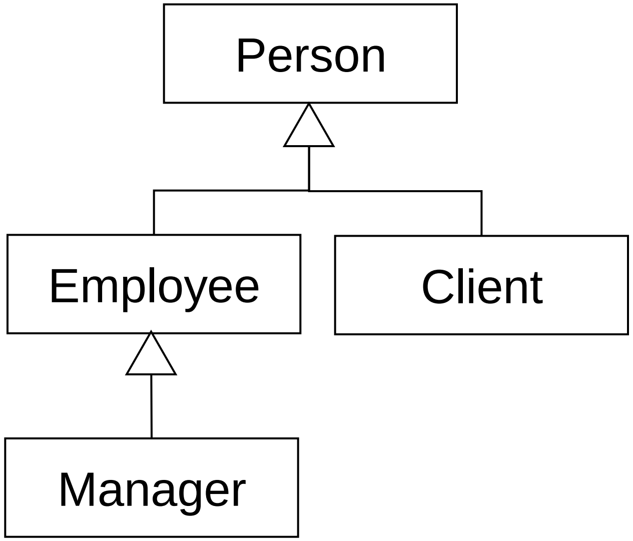
```
class Elem{
    private int value;
    public Elem(int v){
         value=v;
    }
    public int getValue(){
        return value;
    }
    public String toString(){
        return ""+value;
    }

}
```

Index

- **Object oriented design concepts**
- Objects and classes
- Encapsulation
- **Inheritance and polymorphism**
- Summary and conclusions

Inheritance and polymorphism

- Relations, attributes and methods in the parent are available in (direct and indirect) children

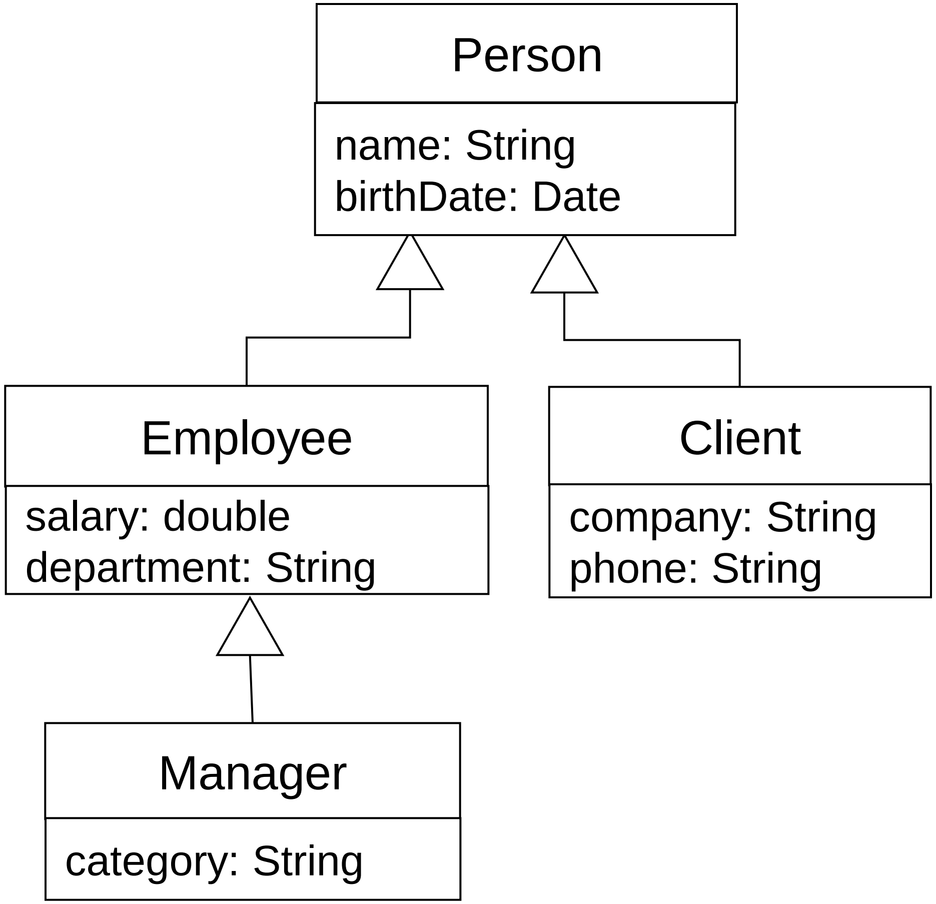


- Type hierarchy
  - Replacement of objects of the parent type by objects of the child type

```
Person x;
Employee y = new Employee();
Manager   z = new Manager();
x = y;
x = z;
```

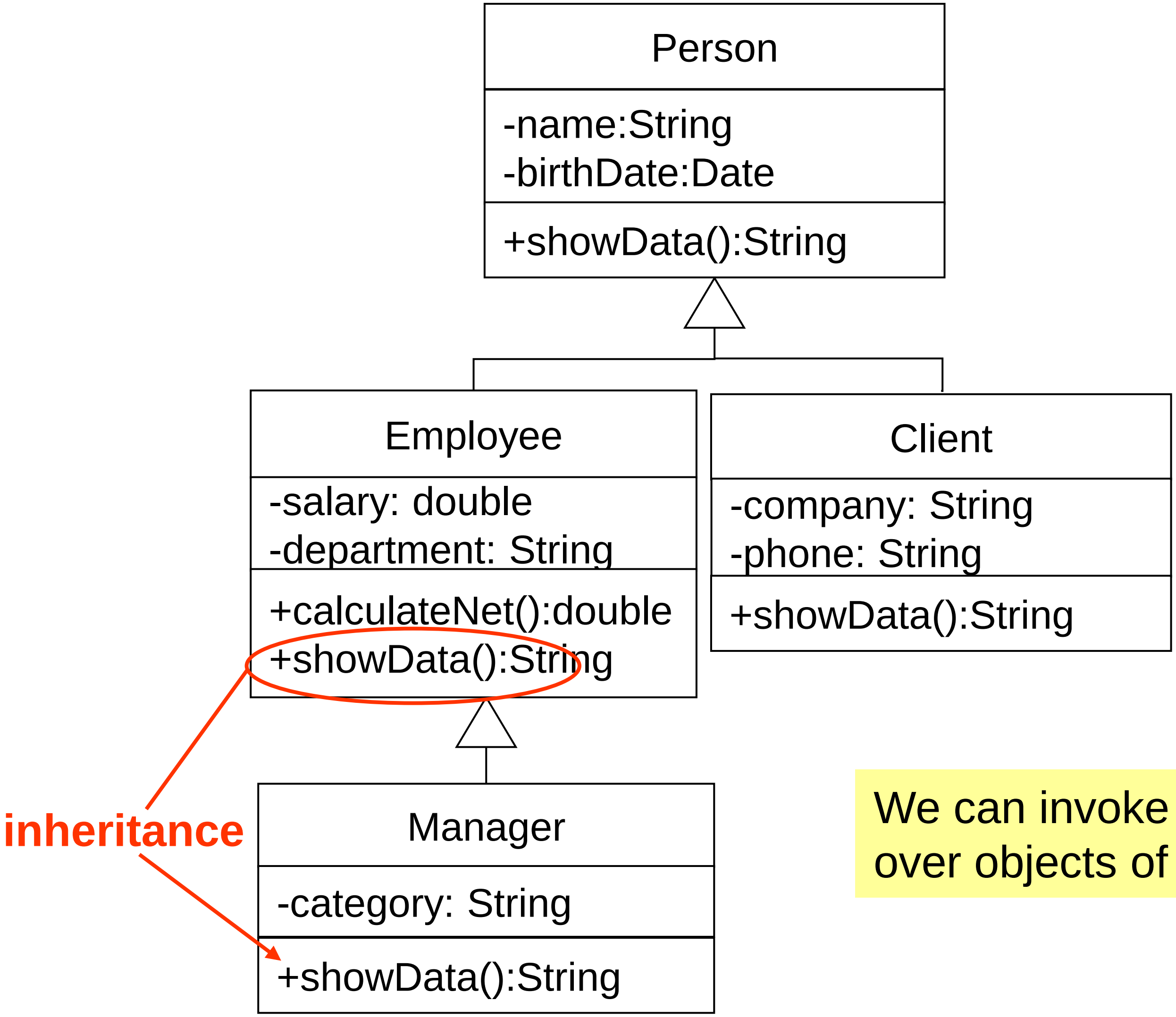
Structure inheritance

```
Employee y = new Employee();
Manager z = new Manager();
// attribute declared in the parent
y.name = “Pedro”;
// attribute declared in Employee
y.salary  = 50000;
// attribute declared in Employee
z.salary = 60000;
```



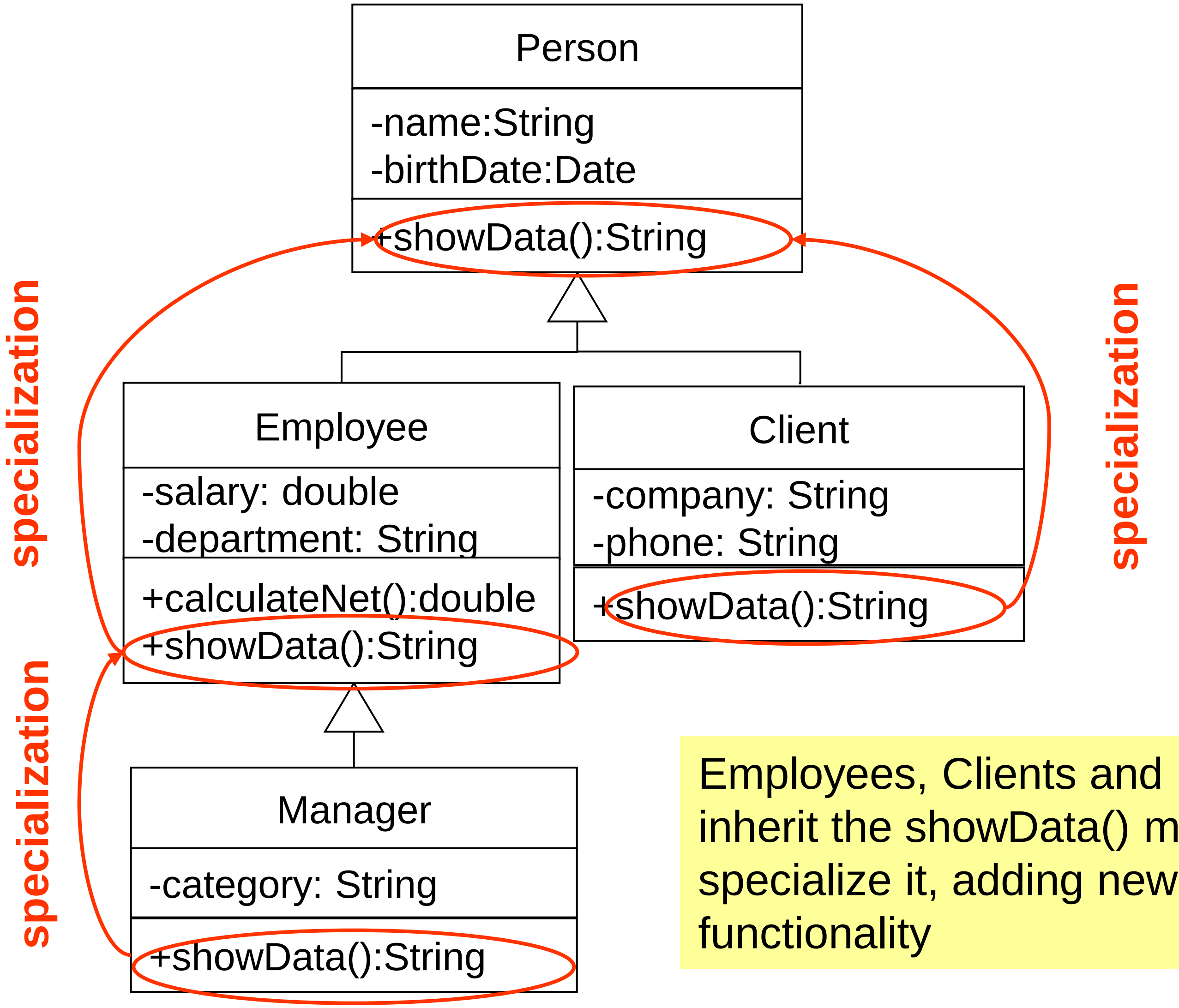
Note: Allowing external access to attributes is a design error. Normally they are only accessed from other classes through public or protected methods

Inheritance of Functionality



We can invoke *calculateNet*() over objects of type *Manager*

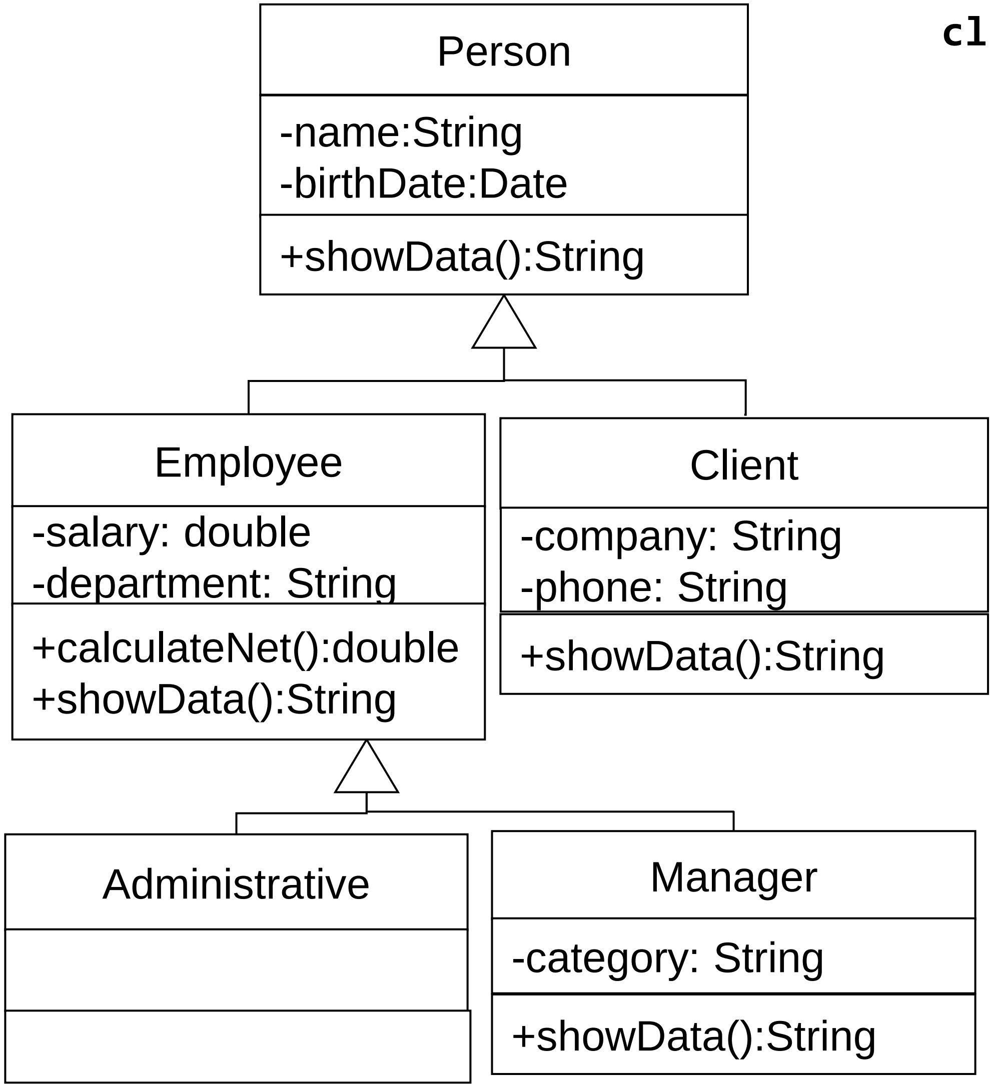
Inheritance of Functionality



Employees, Clients and Managers inherit the showData() method, but specialize it, adding new functionality

[click-me]

Design extensibility



Person

-name:String
-birthDate:Date

+showData():String

Employee

-salary: double
-department: String

+calculateNet():double
+showData():String

Client

-company: String
-phone: String

+showData():String

Administrative

Manager

-category: String

+showData():String

```
class Administrative extends Employee{
     public Administrative(
       String name,
       LocalDate birthDate,
       double salary,
       String department){
        super(name, birthDate,
              salary, department);
    }
   }
```

Reuse and Modularity

Adding a new type of Employee is easy, since the new class inherits all data and functionality of Employee:

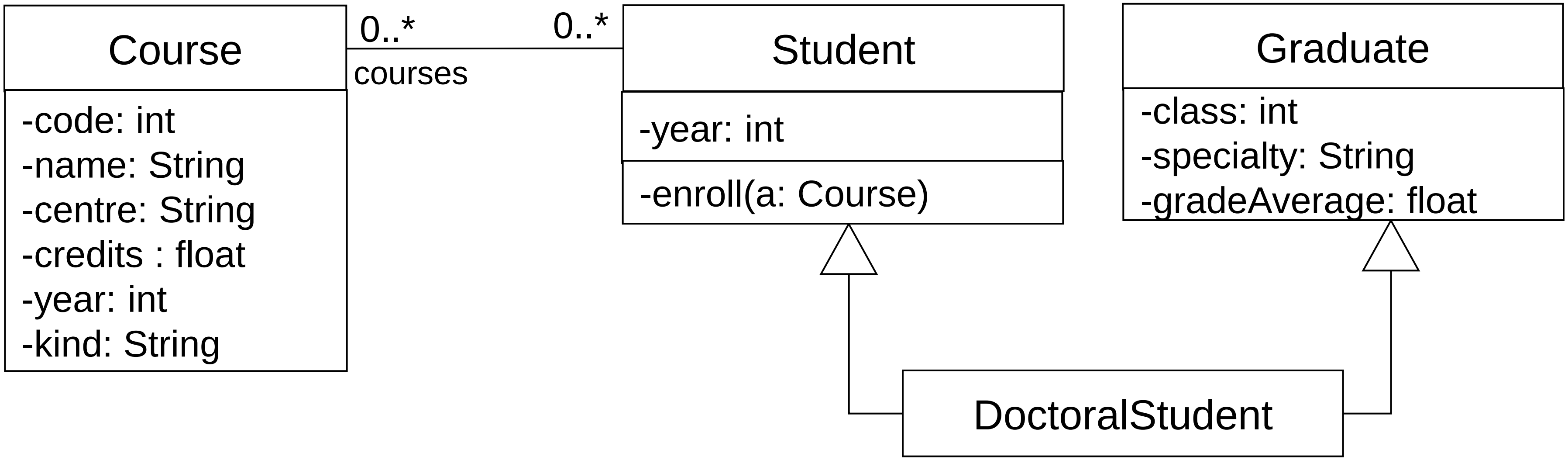
Administrative a = new Administrative(...

….

a.showData();

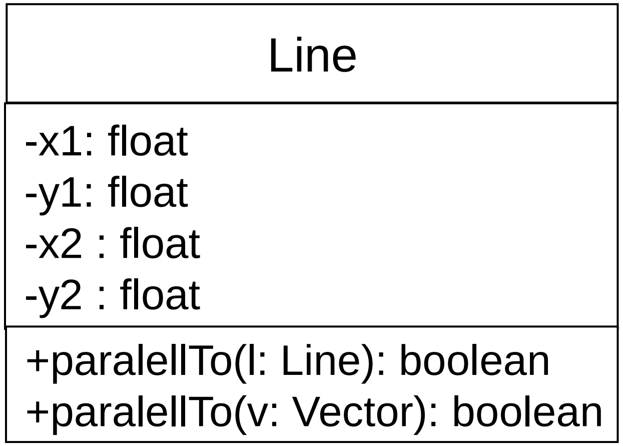
Multiple inheritance

- A class may have several superclasses
- The class inherits attributes and methods from all of them



Polimorphism

- Method overloading



Line

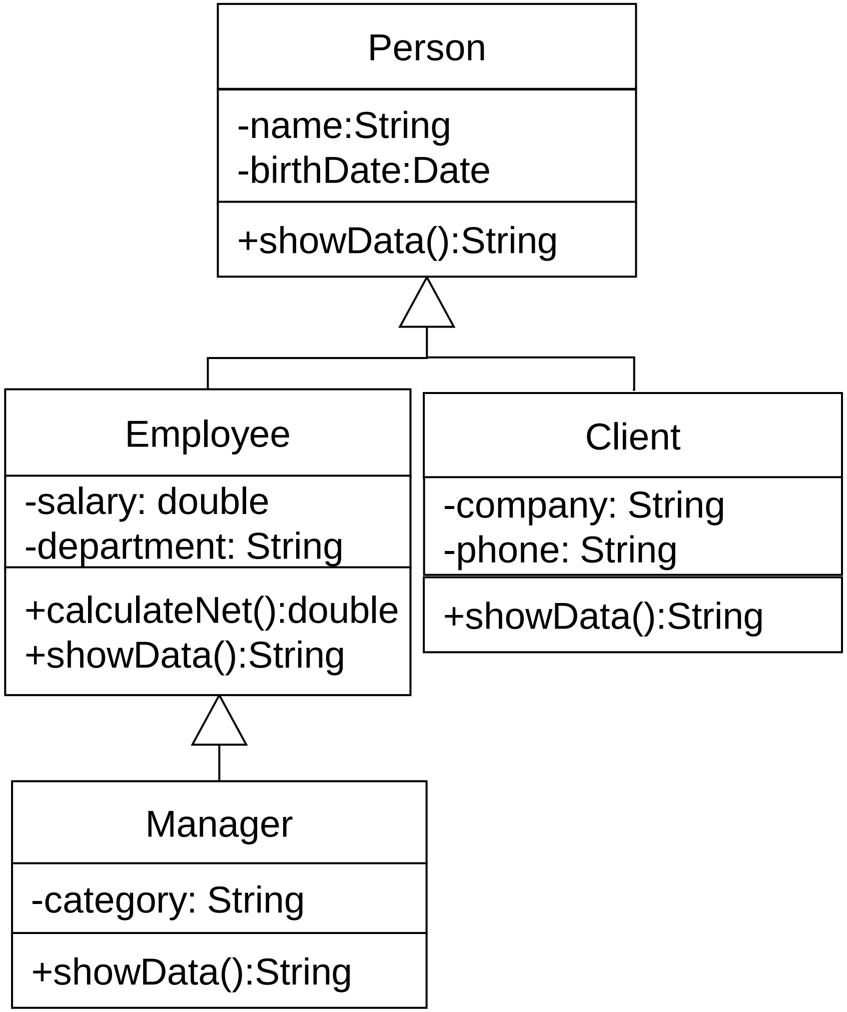
-x1: float
-y1: float
-x2 : float
-y2 : float

+paralellTo(l: Line): boolean
+paralellTo(v: Vector): boolean

```
Line r1 = new Line();
Line r2 = new Line();
Vector v = new Vector();

r1.parallelTo(r2);
r1.parallelTo(v);
```

- Same method name, different arguments

Dynamic Binding



Person

-name:String
-birthDate:Date

+showData():String

Employee

-salary: double
-department: String

+calculateNet():double
+showData():String

Client

-company: String
-phone: String

+showData():String

Manager

-category: String

+showData():String

```
Person x;
Employee y = new Employee();
x = y;
x.showData ( ) // (1)?
y.showData ( )
```

- Which method body is executed?
- In C++: the one of Person
  - **Static binding**
  - To make it dynamic, methods should be declared “virtual”.
- In Java: the one of Employee.
  - **Dynamic binding**
- Due to the inheritance hierarchy, the compiler does not know until run-time which method body is to be executed

Index

- **Object oriented design concepts**
- Objects and classes
- Encapsulation
- Inheritance and polymorphism
- **Summary and conclusions**

Summary

- Object orientation: Application as a set of interacting objects
- Concepts:
  - Classes, Objects, Encapsulation, Polymorphism and Inheritance
- Advantages:
  - Extensibility, reuse
  - Models the real world in a more natural way

Bibliography

- Object-Oriented and Classical Software Engineering, Eight edition. Stephen Schach. McGraw-Hill.
- Object Oriented Software Construction. Betrand Meyer. Prentice Hall. INF/681.3.06/MEY.
- The Unified Modeling Language Reference Manual. Rumbaugh, James. Pearson Addison Wesley. 2007. INF/681.3.062-U/RUM.