Design and Analysis of Information Systems (MAS)

Mariusz Trzaska, Ph. D.
mtrzaska@pjwstk.edu.pl

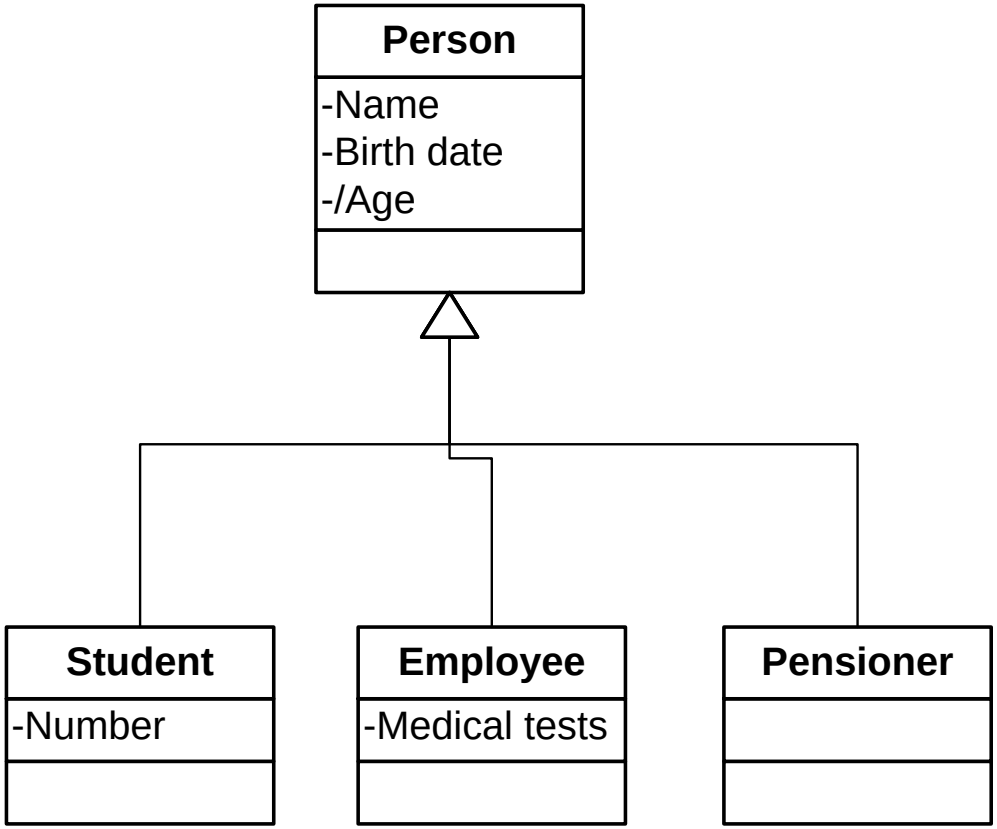
Lecture 07

Inheritance in Object-Oriented Programming Languages

Outline

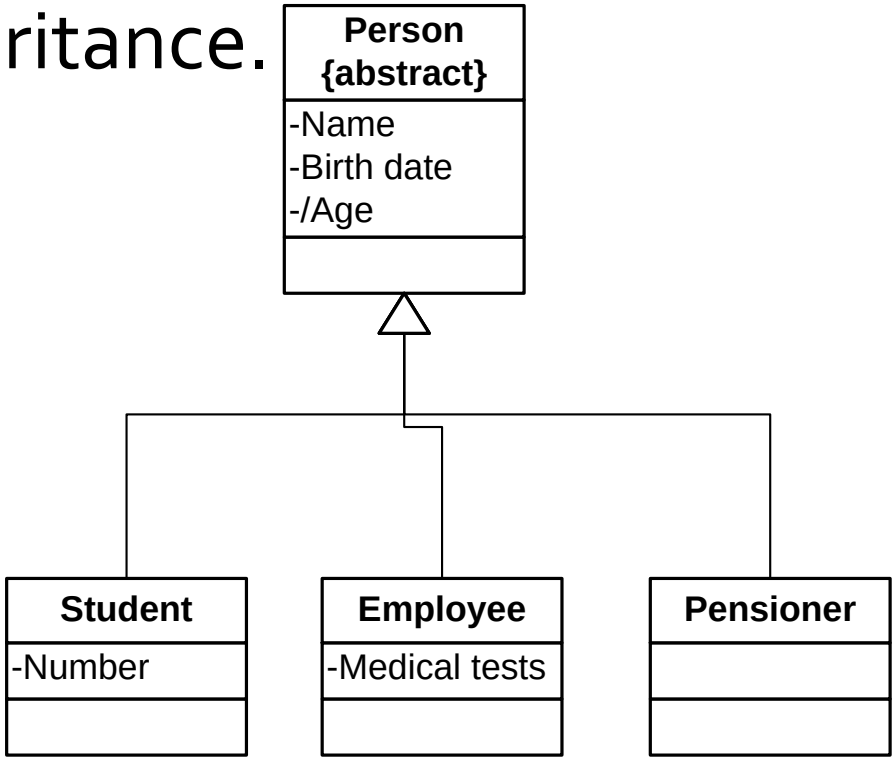
- Discussion of different types of inheritance, abstract classes and polymorphic methods calls.
- Implementation of the basic inheritance.
- Utilization of the abstract classes and polymorphic methods calls.
- Implementation of the various inheritance types:
  - overlapping,
  - complete, incomplete,
  - multi-inheritance,
  - multi-aspect,
  - dynamic.
- Pros and cons of different approaches.
- Summary.

The *Disjoint* Inheritance



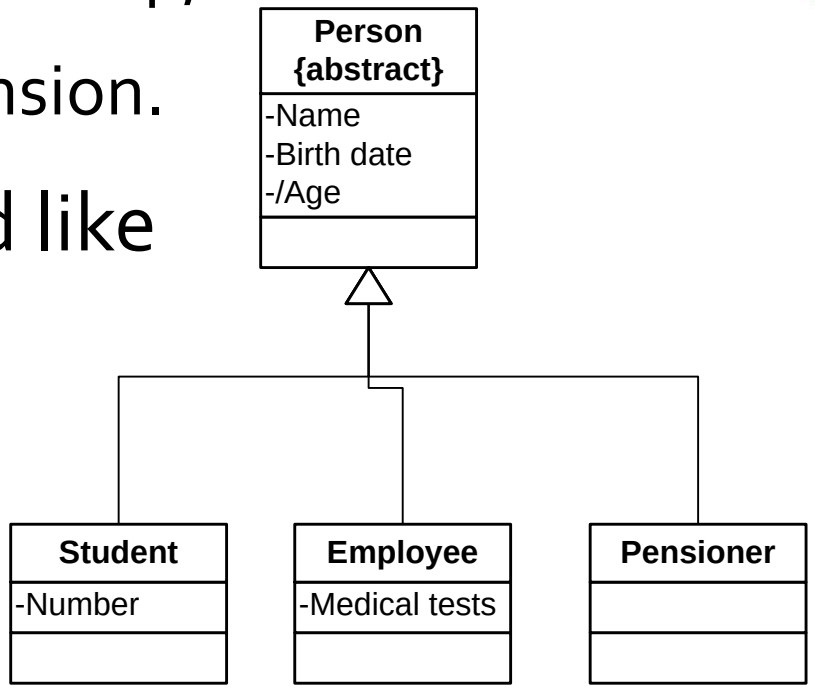
The Abstract Class

- A class which cannot have direct occurrences.
- Used to create a hierarchy of the inheritance.



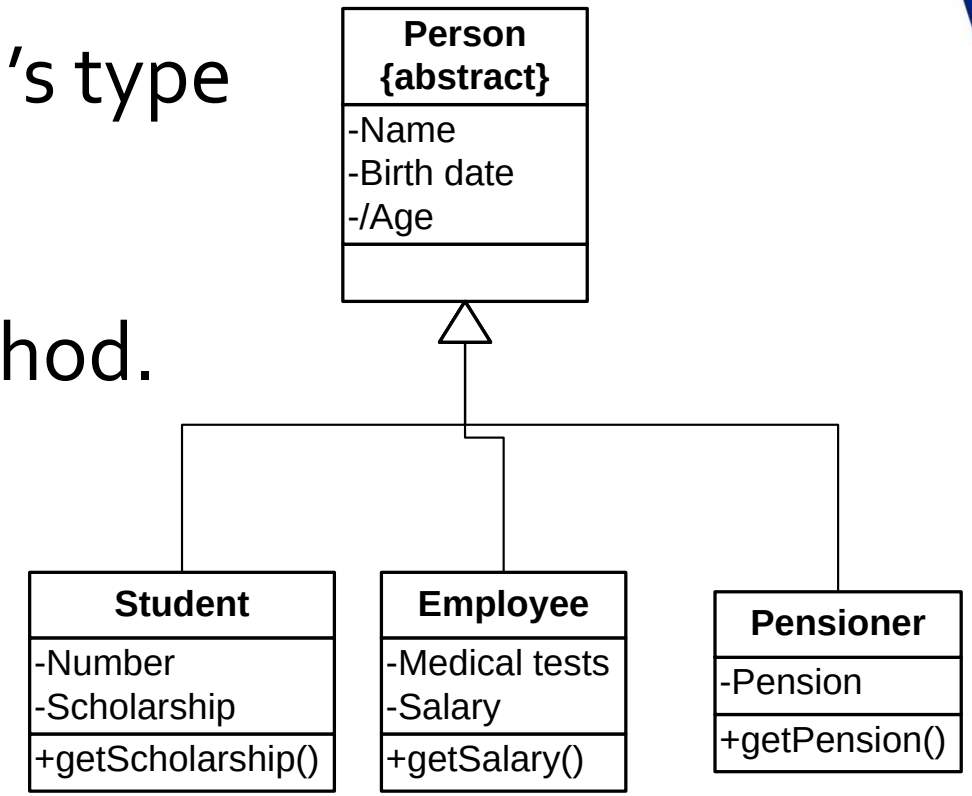
A Business Case

- Let's assume that the people from the diagram have some incomes:
  - The employee has a salary,
  - The student has a scholarship,
  - The pensioner has a pension.
- And of course we would like to have a way of asking for the income.



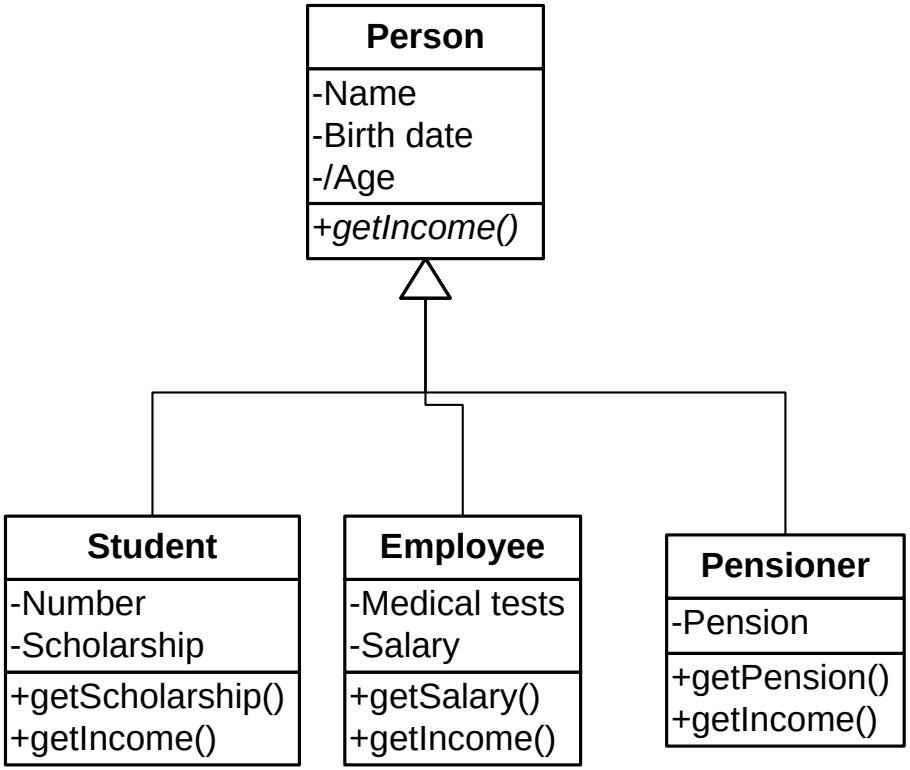
A Business Case (2)

- The simplest way is to put appropriate attributes in particular classes and add setters and getters.
- Based on a person's type we will call the particular method.
- Is there a better way?



The Polymorphic Method Calls

- Uses overriding.
- Allows performing an operation without „manually" checking particular type of the object.

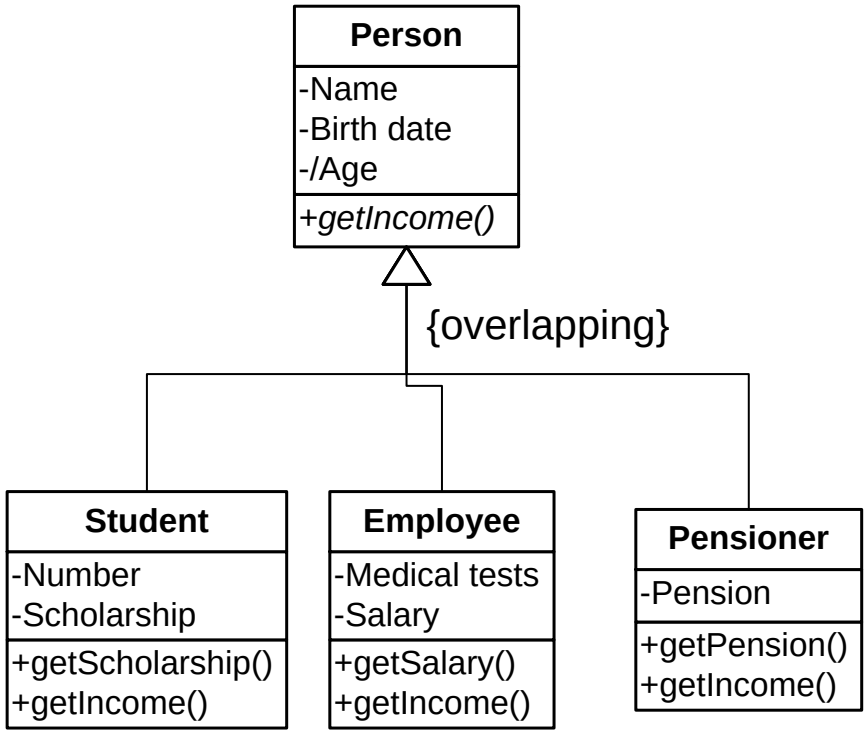


Abstract Methods

- What kind of source code will be placed in the `getIncome()` method of the Person class?
- The person does not have any incomes.
- The solution: Let's mark the method as an **abstract method**.
- An abstract method:
  - Does not have a body,
  - Has to be implemented in subclasses,
  - Can be located only in an abstract class.

Remaining types of an inheritance

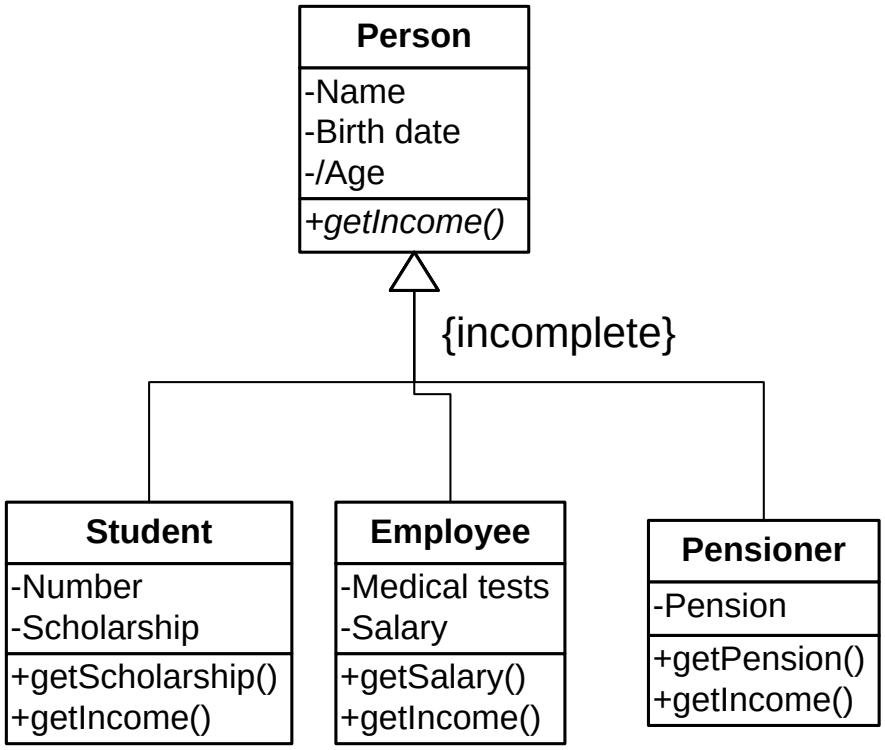
- The *overlapping* inheritance



How about overriding and polymorphic method calls?

Remaining types of an inheritance (2)

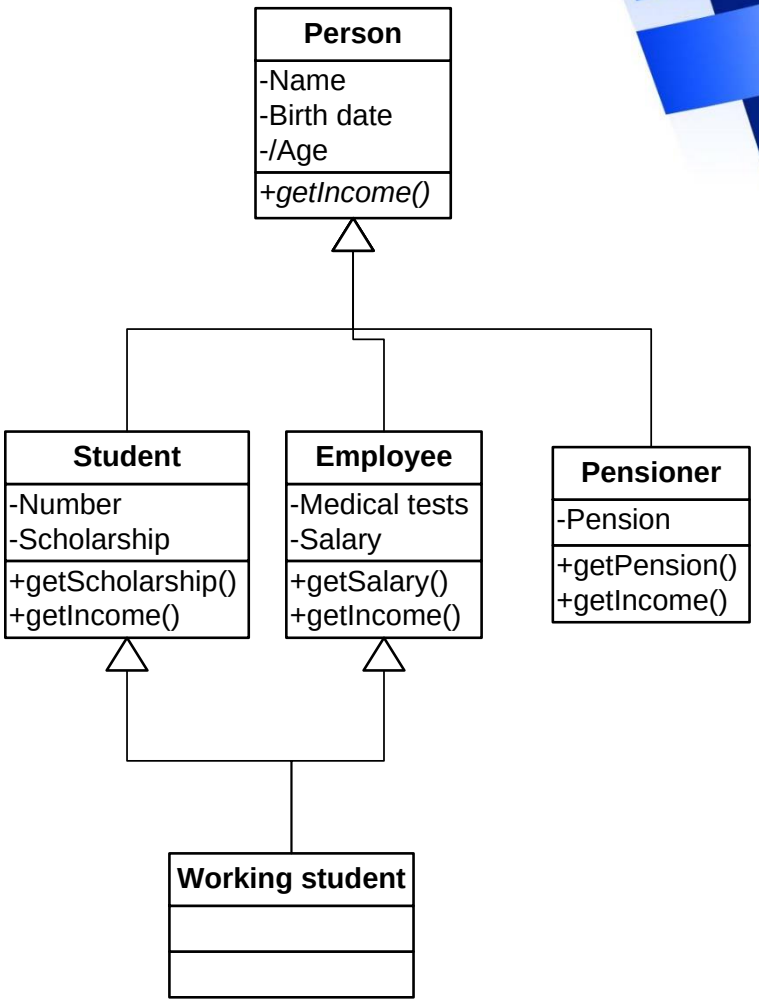
- *Incomplete, complete* inheritance



- Ellipsis (three dots)

Remaining types of an inheritance (3)

- *Multi-inheritance*
  - on contrary to the overlapping inheritance, we can add additional attributes in the subclass.

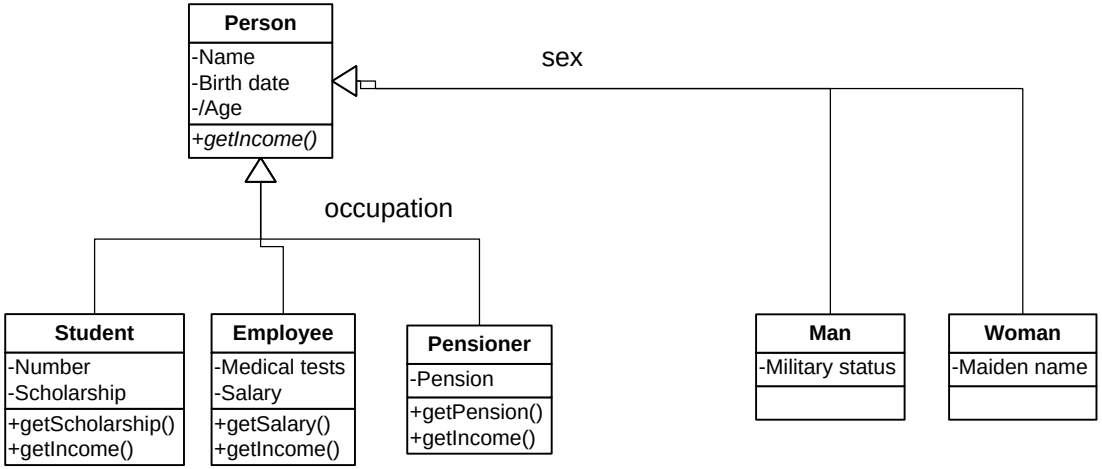


Remaining types of an inheritance (4)

- Multi-inheritance – *cont*.:
  - Problems,
  - A perfect solution?
  - How about overriding and polymorphic method calls?

Remaining types of an inheritance (5)

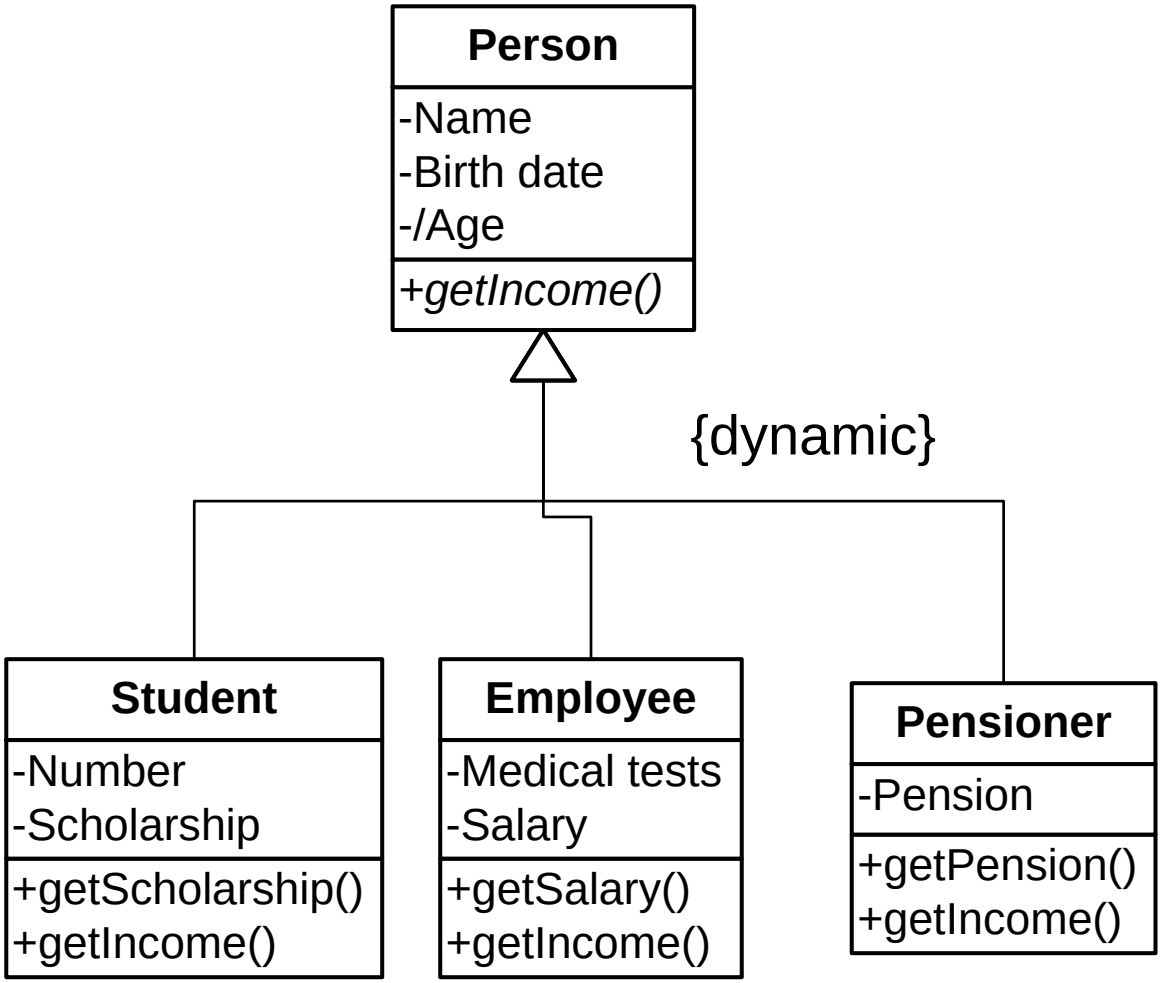
- The *multi-aspect* inheritance



How about overriding and polymorphic method calls?

Remaining types of an inheritance (6)

- The *dynamic* inheritance



Inheritance and object-oriented languages

- In most cases popular programming languages have only the simplest inheritance:
  - Disjoint,
  - Multi-inheritance (C++ only).
- How about the remaining types of inheritances?
  - Different ways of “faking”,
  - Implementation.
- How about abstract classes and methods?
- How about polymorphic method calls?

Implementation of the *Disjoint* Inheritance

- This type of inheritance exists directly in popular programming languages.

```
public abstract class Person {
   private String firstName;
   private String lastName;
   private LocalDate birthDate;
}
```

```
public class Employee extends Person {
   private boolean medicalTest;
}
```

```
public class Student extends Person {
   private int number;
}
```

```
public class Pensioner extends Person {
   private float pension;
}
```

Utilization of the Polymorphic Methods Calls

- In the Java language:
  - Abstract classes,
  - Abstract methods,
  - Polymorphic method calls

  exist directly.
- In the C++ language the above terms also exist, but utilization is a bit different. They have to be declared with the `virtual` keyword.

Utilization of the Polymorphic Methods Calls (2)

- Utilization is similar to the *disjoint* inheritance.

```
public abstract class Person {
   private String firstName;
   private String lastName;
   private LocalDate birthDate;

   public Person(String firstName, String lastName, LocalDate birthDate) {
      super();
      this.firstName = firstName;
      this.lastName = lastName;
      this.birthDate = birthDate;
   }

   public abstract float getIncome();
}
```

```
public class Employee extends Person {
   // [...]
   @Override
   public float getIncome() {
      return getSalary();
   }

   public float getSalary() { return salary; }
}
```

- The remaining classes are implemented in the same way.

Utilization of the Polymorphic Methods Calls (3)

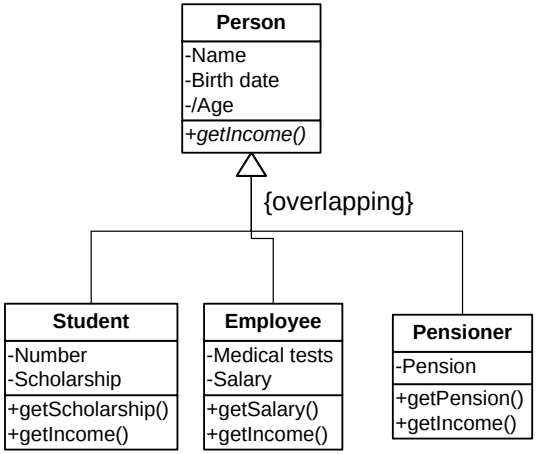
- Create two objects:
  - An employee,
  - A student.
- We treat them just as persons (a reference to the Person class)
- Each of them we ask about an income (without checking real kind of the object).
- Thanks to the polymorphic call we got the right answers (method calls) regardless of the real type of the object (employee or student).

```
Person p1 = new Employee("John", "Smith", LocalDate.of(1990, 12, 20), true,
4000.0f);
Person p2 = new Student("Adam", "Black", LocalDate.of(1995, 10, 18), 1212,
2000.0f);

System.out.println(p1.getIncome());
System.out.println(p2.getIncome());
```

```
4000.0
2000.0
```

Implementation of the *overlapping* Inheritance

- This kind of inheritance does not exist directly in popular programming languages.
- Faking:
  - Replacing entire hierarchy of the inheritance with a single class (flattening the hierarchy),
  - Utilization of the aggregation or composition,
  - Mixed solution.



Implementation of the *overlapping* Inheritance (2)

- Replacing entire hierarchy of the inheritance with a single class
  - All invariants are placed in the single super class,
  - Add a discriminator which will tell us the kind of object we use (we need to use `EnumSet` because we need a way to store information about a couple of types in the same time).

```
enum PersonType {Person, Employee, Student, Pensioner};

public class Person {
   private String firstName;
   private String lastName;
   private LocalDate birthDate;

   private boolean medicalTest;
   private int number;

   // We need to use EnumSet rather then PersonType because we would like
   // to have a possibility of storing combinations of the Person, e.g. Employee
   //                + Student
   private EnumSet<PersonType> personKind =  EnumSet.of(PersonType.Person);

   // [...]
}
```

Implementation of the *overlapping* Inheritance (3)

- Replacing entire hierarchy of the inheritance with a single class – cont.
  - A code checking the kind of the person inside the methods, e.g.

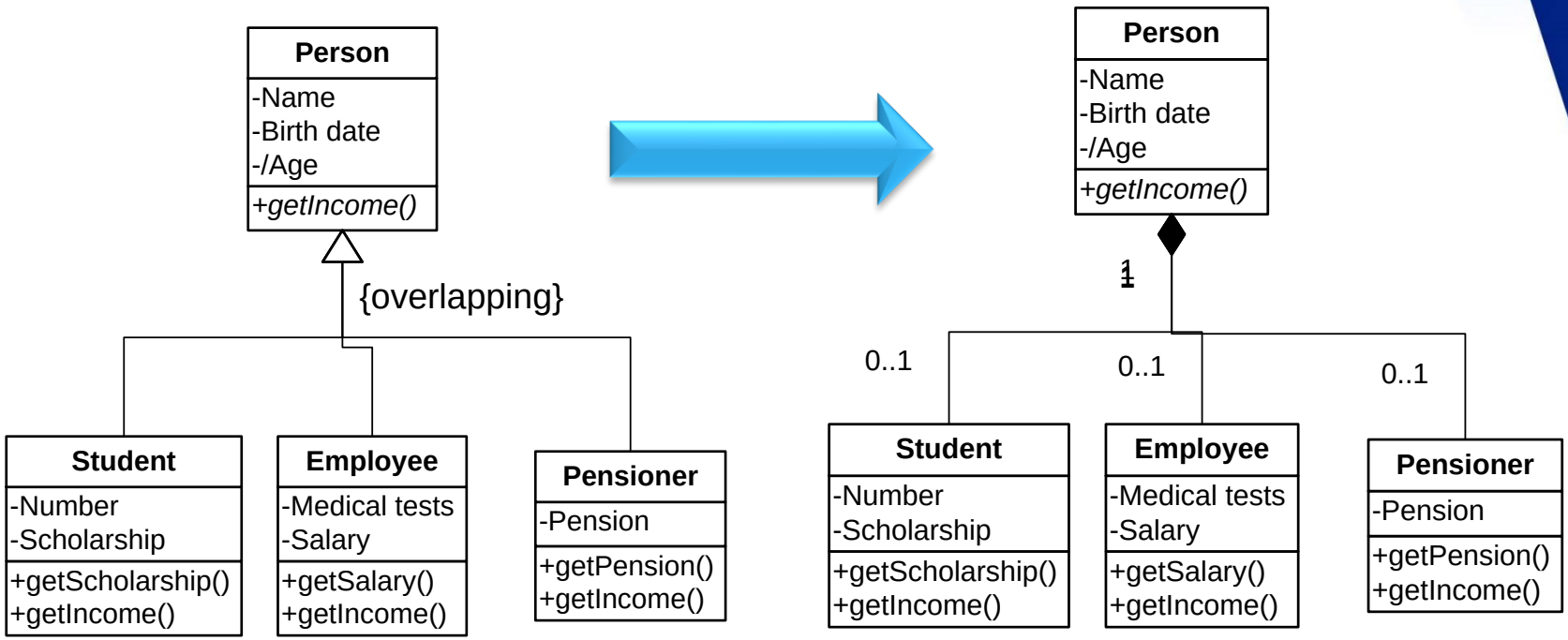
```
public class Person {
   private boolean medicalTest;

   // [...]

   private EnumSet<PersonType> personKind =  EnumSet.of(PersonType.Person);

   public boolean hasMedicalTest() throws Exception {
      if(personKind.contains(PersonType.Employee)) {
         return medicalTest;
      }

      throw new Exception("The person is not an employee!");
   }
   public void setMedicalTest(boolean medicalTest) throws Exception {
      if(personKind.contains(PersonType.Employee)) {
         this.medicalTest = medicalTest;
      }
      else {
         throw new Exception("The person is not an employee!");
      }
   }
}
```

Implementation of the *overlapping* Inheritance (4)

- Replacing entire hierarchy of the inheritance with a single class – *cont.*
  - Pros
    - Easy implementation
    - Easy utilization
  - Cons
    - No inheritance goodies (i.e. overriding methods, polymorphic calls),
    - Wasting of place for unused attributes.

Implementation of the *overlapping* Inheritance (5)

- Utilization of the aggregation or composition



Implementation of the overlapping Inheritance (6)

- Utilization of the aggregation or composition – cont.
  - Associations from sub classes points to:
    - The whole. Implementations of the associations have to be moved from sub classes to the super class.
    - A part. In such an approach object-parts cannot be hidden. They have to be directly reachable (not through the whole-object).
  - Aggregation or composition implemented using one of the already discussed approaches.
  - Utilization of the ObjectPlusPlus can save us a lot of work.

Implementation of the overlapping Inheritance (7)

- Utilization of the aggregation or composition – cont.
  - Additional methods:
    - Giving access to target objects,
    - Giving access to links stored inside of the target objects.

Implementation of the *overlapping* Inheritance (8)

```
public class Person extends ObjectPlusPlus {
   private String firstName;
   private String lastName;
   private LocalDate birthDate;

   public Person(String firstName, String lastName, LocalDate birthDate) {
      super();   // Required by the ObjectPlusPlus

      this.firstName = firstName;
      this.lastName = lastName;
      this.birthDate = birthDate;
   }

   /**
    * Creates a person as an employee.
    */
   public Person(String firstName, String lastName, LocalDate birthDate, boolean
medicalTest) {
      super();   // Required by the ObjectPlusPlus

      this.firstName = firstName;
      this.lastName = lastName;
      this.birthDate = birthDate;

      // "Changes" a person into an employee
      addEmployee(medicalTest);
   }

   //[...]
```

Implementation of the *overlapping* Inheritance (9)

```
public class Person extends ObjectPlusPlus {

   // [...]

   public void addEmployee(boolean medicalTest) {
      // Creation of the employee part
      Employee p = new Employee(medicalTest);

      // Adding an employee as a link
      // We use a method provided by the ObjectPlusPlus
      this.addLink(roleNameEmployee, roleNameGeneralization, p);
   }

   public void addStudent(int number) throws Exception {
      // Creation of the student part
      Student s = new Student(number);

      // Adding a student as a link
      // We use a method provided by the ObjectPlusPlus
      this.addLink(roleNameStudent, roleNameGeneralization, s);
   }

   private static String roleNameEmployee = "specializationEmployee";
   private static String roleNamePensioner = "specializationPensioner";
```

Implementation of the *overlapping* Inheritance (10)

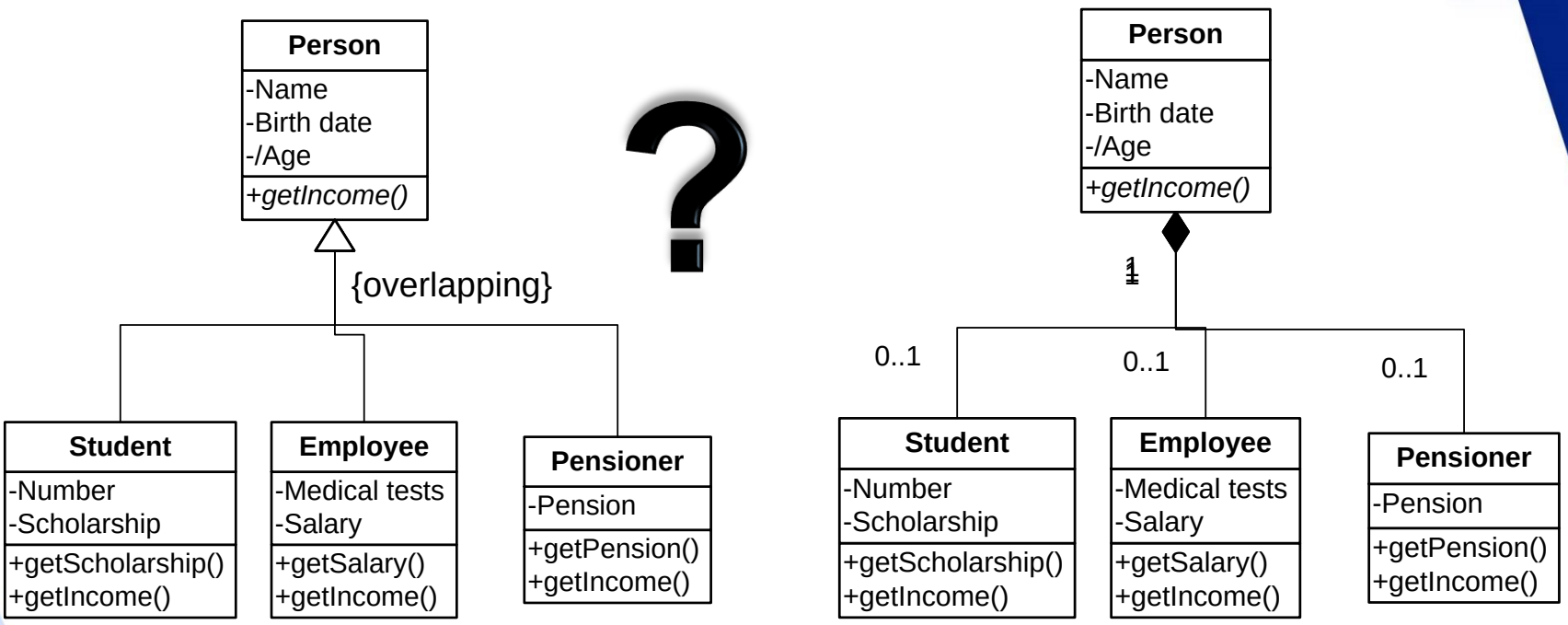
```
public class Person extends ObjectPlusPlus {
   // [...]

   public boolean hasMedicalTest() throws Exception {
      // get an employee object
      try {
         ObjectPlusPlus[] obj = this.getLinks(roleNameEmployee);
         return ((Employee) obj[0]).isMedicalTest();
      } catch (Exception e) {
         // Probably this is an exception telling that this is not an employee
         // (we should introduce different exception classes)
         throw new Exception("The object is not an employee!");
      }
   }

   public int getNumber() throws Exception {
      // get a student object
      try {
         ObjectPlusPlus[] obj = this.getLinks(roleNameStudent);
         return ((Student) obj[0]).getNumber();
      } catch (Exception e) {
         // Probably this is an exception telling that this is not a student
         // (we should introduce different exception classes)
         throw new Exception("The object is not a Student!");
      }
   }
```

Implementation of the *overlapping* Inheritance (11)

- Utilization of the aggregation or composition – *cont*.
  - Pros
    - Easy to use (with additional methods)
    - We use only necessary invariants.
  - Cons
    - No inheritance goodies (i.e. overriding methods, polymorphic calls),

Polymorphic in the *overlapping* Inheritance

- Which version of the method should be executed?



- Probably none of the above…

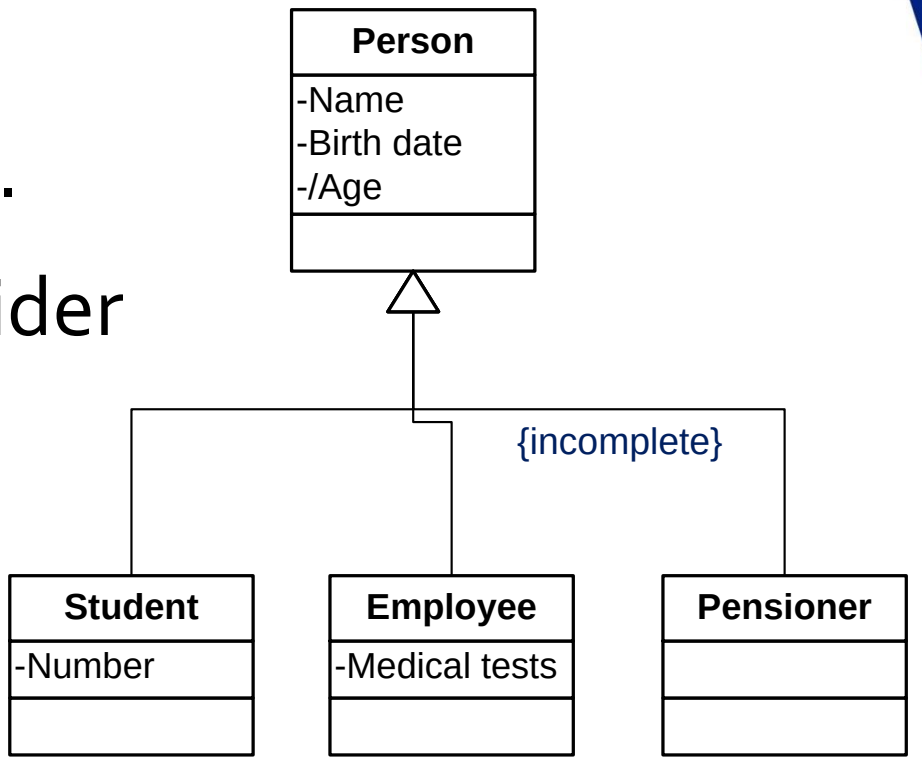
Polymorphic in the *overlapping* Inheritance (2)

- We need to create a new method, which will perform special calculations:

```
public float getIncome() throws Exception {
   float income = 0.0f;

   if(this.anyLink(roleNameEmployee)) {
      // Employee
      ObjectPlusPlus[] obj = this.getLinks(roleNameEmployee);

      // ==> add employee's income
      income += ((Employee) obj[0]).getIncome();
   }

   if(this.anyLink(roleNameStudent)) {
      // Student
      ObjectPlusPlus[] obj = this.getLinks(roleNameStudent);

      // ==> add Student's income
      income += ((Student) obj[0]).getIncome();
   }

   if(this.anyLink(roleNamePensioner)) {
      // Pensioner
      ObjectPlusPlus[] obj = this.getLinks(roleNamePensioner);

      // ==> ==> add pensioner's income
      income += ((Pensioner) obj[0]).getIncome();
   }

   return income;
}
```

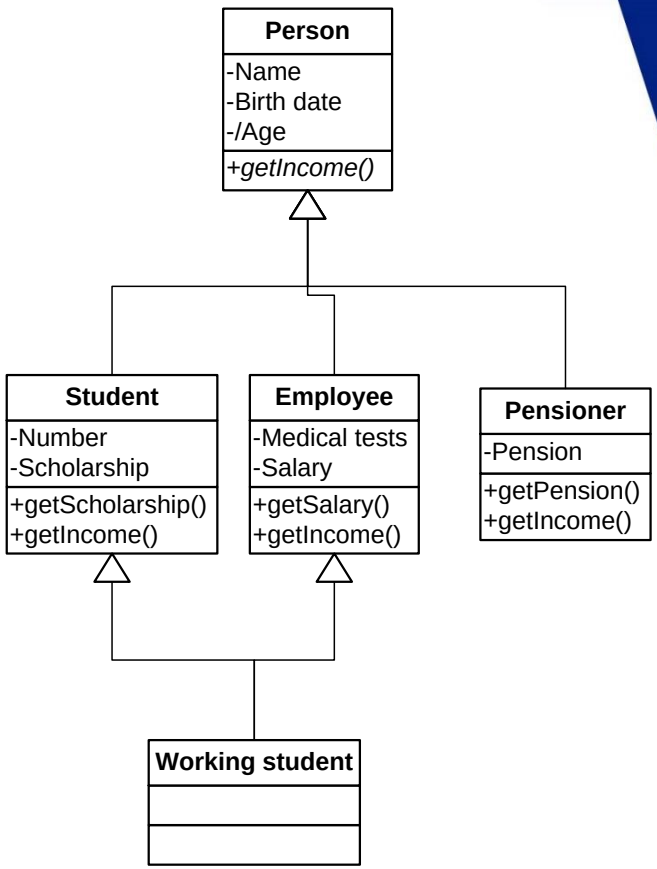
The *Complete* and *Incomplete* Inheritance

- What those types of inheritance mean for the diagram?
- Do they mean something for the implementation?
- Usually, they do not.
- Optionally, we consider the future implementation.



Implementation of the multi-inheritance

- It does exist in the C++
  - In case of a naming conflict we need to use the scope operator.
- **It does not exist in** Java nor MS C#.
- Hence, how we can implement it?



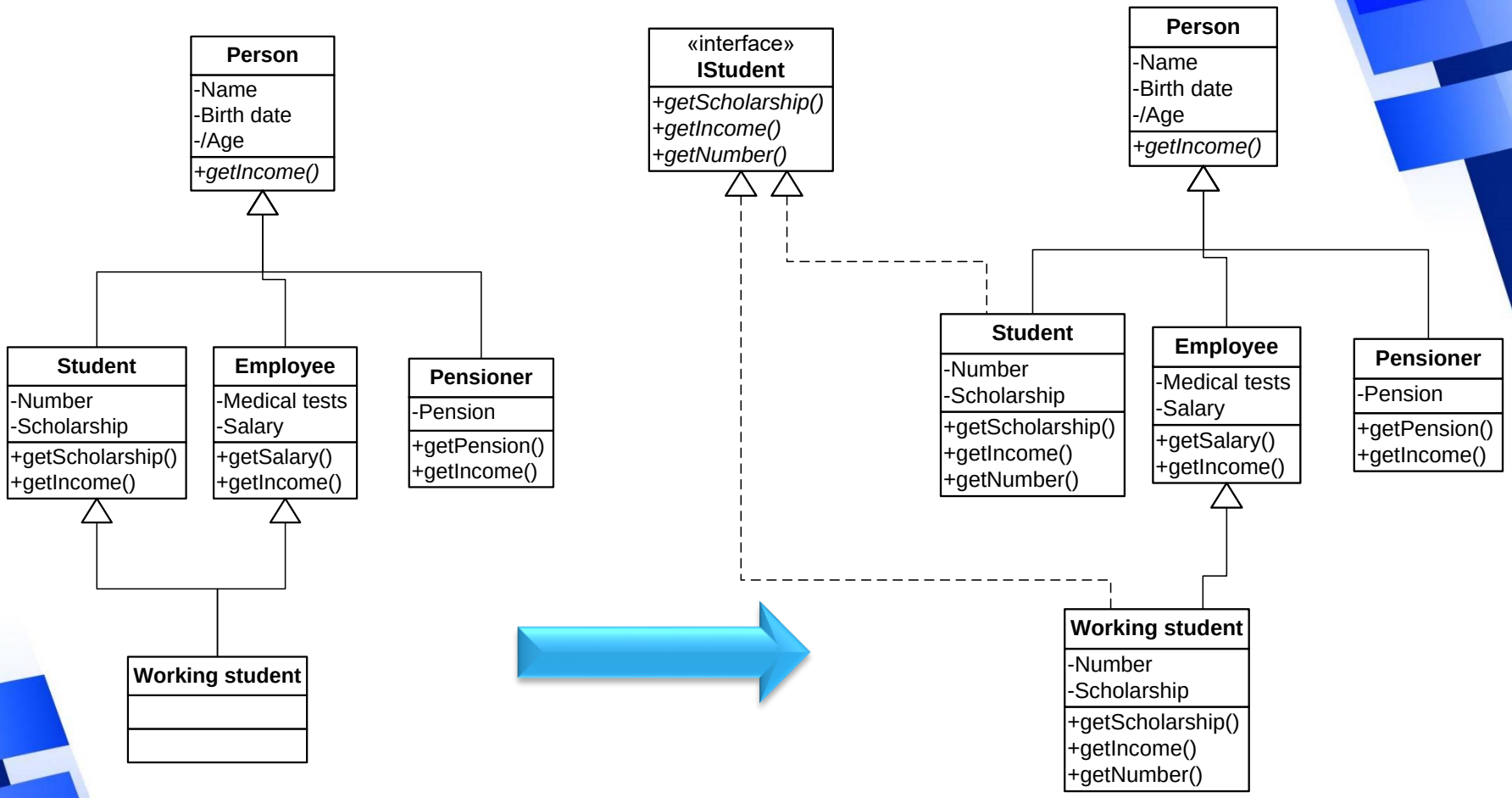
Implementation of the multi-inheritance (2)

- We can use of the approaches described in case of the overlapping inheritance:
  - A single class (flattening the hierarchy),
  - An aggregation or composition.
- We can also use interfaces.

Implementation of the Multi-inheritance Using Interfaces

- A class can implement any number of interfaces.
- Because of the interfaces' limitation we can use only methods (no attributes).
- The above problem could be partially solved using getters/setters approach.
- Sometimes there is a necessity of implementation of the same methods in the same way.
- Java 8+: `default` and `static` methods.

Implementation of the Multi-inheritance Using Interfaces (2)



Implementation of the Multi-inheritance Using Interfaces (3)

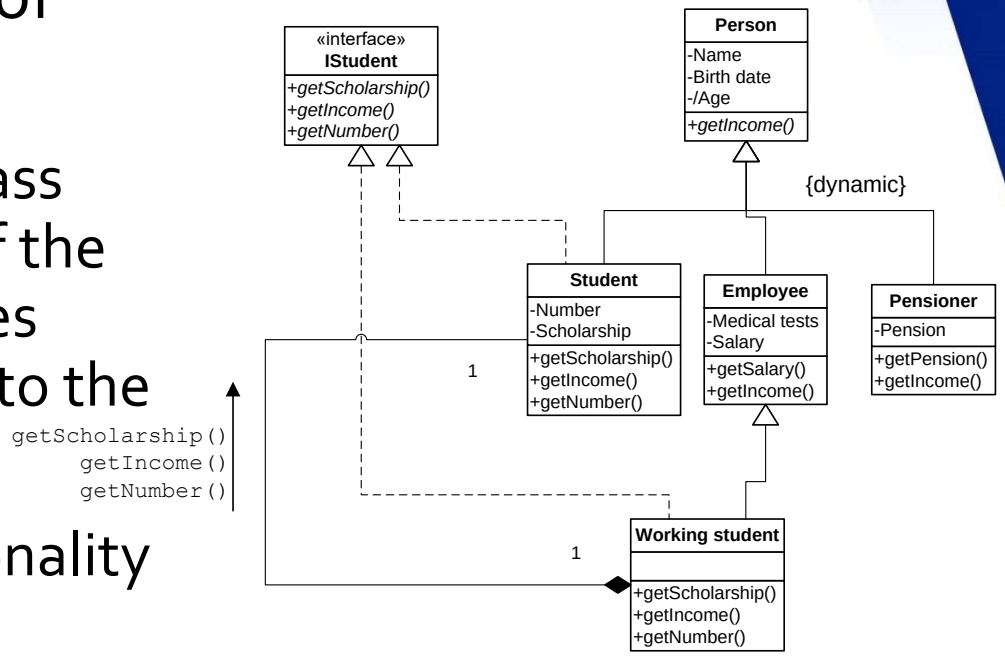
```
public interface IStudent {
   float getIncome();
   float getScholarship();
   void setScholarship(float scholarship);
   int getNumber();
}
```

```
public class WorkingStudent extends Employee implements IStudent {
   private int number;
   private float scholarship;

   public WorkingStudent(String firstName, String lastName, LocalDate birthDate, int number,
float
                         scholarship, boolean medicalTest, float salary) {
      super(firstName, lastName, birthDate, medicalTest, salary);
      this.number = number;
      this.scholarship = scholarship;
   }

   @Override
   public float getScholarship() {
      return scholarship;
   }
   @Override
   public float getIncome() {
      return super.getIncome() + getScholarship();
   }

   // [...]
}
```

As you can see some methods (i.e. `getScholarship()`) have to be implemented more than once and in the same way.

Implementation of the Multi-inheritance Using Interfaces (4)

- Partial solution of the mentioned problem (the same implementations of the same methods).
  - The *WorkingStudent* class inherits functionality of the employee and delegates student's functionality to the *Student* class.
  - The class wraps functionality of the *Student* class.
  - Optionally one can use *static* and/or *default* methods in interfaces (Java 8+).



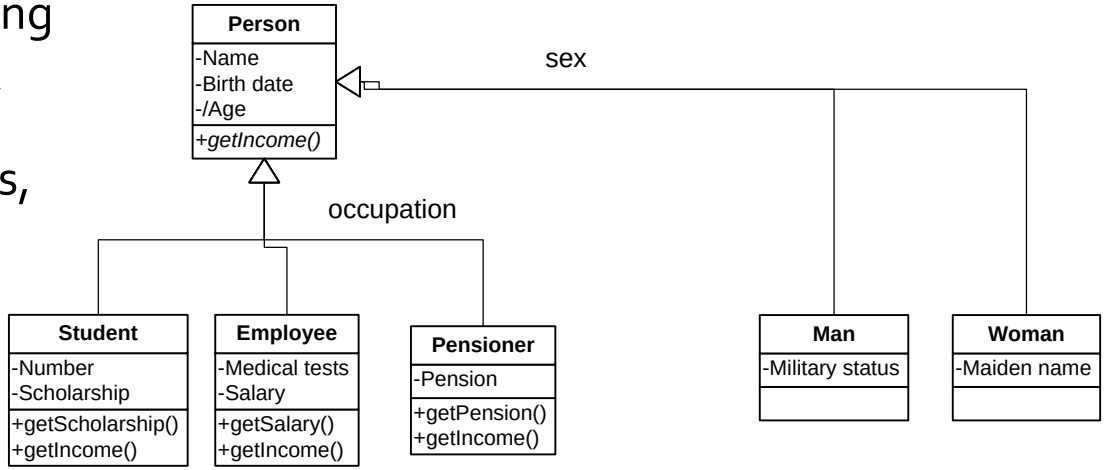
Implementation of the Multi-inheritance Using Interfaces (5)

- Some concern could be related to the way of storing student properties. They could be stored two times: in the *WorkingStudent* class and in the *Student* class.
  - Modyfication of the *Student* class,
  - Passing *nulls* to the instance of the *Student* class.

```
public class WorkingStudent extends Employee implements IStudent {
   Student student;

   public WorkingStudent(String firstName, String lastName, LocalDate birthDate,
int number, float scholarship, boolean medicalTest, float salary) {
      super(firstName, lastName, birthDate, medicalTest, salary);
      student = new Student(null, null, null, number, scholarship);
   }

   @Override public float getIncome() {
      return super.getIncome() + getScholarship();
   }
   @Override public int getNumber() {
      return student.getNumber();
   }
   @Override public float getScholarship() {
      return student.getScholarship();
   }
   @Override public void setScholarship(float scholarship) {
      student.setScholarship(scholarship);
   }
}
```

Implementation of the Multi-aspect Inheritance

- Does not exist directly in popular programming languages (Java, C#, C++).
- It has to be implemented:
  - One of the aspects is inherited using embedded inheritance of the language.
  - Remaining aspects:
    - Are implemented using one of the previously discussed approaches,



    - Are removed using e.g. additional flag in the base class

Implementation of the Multi-aspect Inheritance (2)

- Which aspect should be inherited?
  - The one where is method overriding, polymorphism, etc.,
  - The one where is greater attributes diversity in sub classes.
  - Other words: the one with more complex/complicated hierarchy.

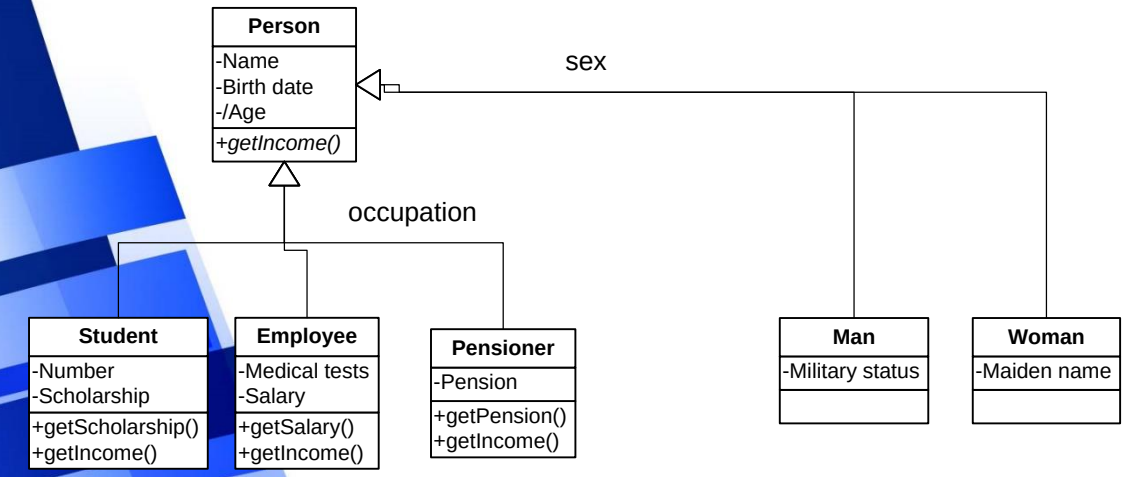
Implementation of the Multi-aspect Inheritance (3)

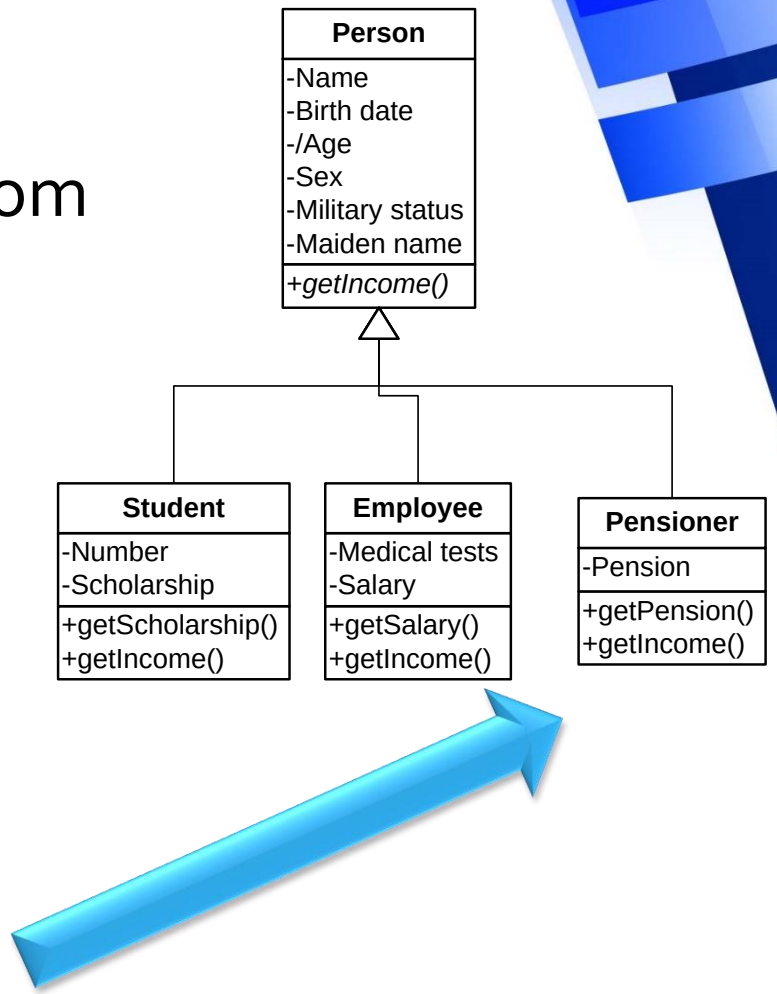
- In some situations when:
  - There is no aspect specific information (and only information about aspect) we can use a simple flag in a super class rather than inheritance.
  - Aspect specific information is stored only in a few attributes we can also put them into a super class and abandon one of the aspect.

Implementation of the Multi-aspect Inheritance (4)

- Sample solution no 1

  Attributes and methods from the removed aspect are placed in the super class.





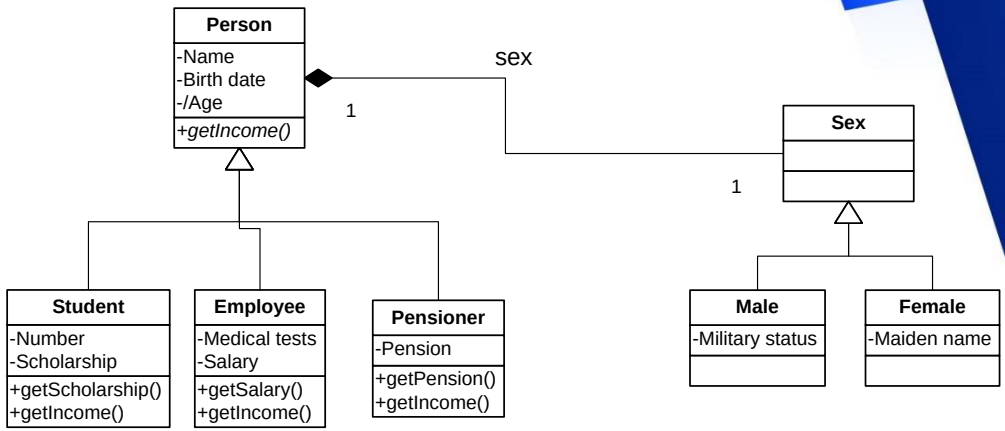
Implementation of the Multi-aspect Inheritance (5)

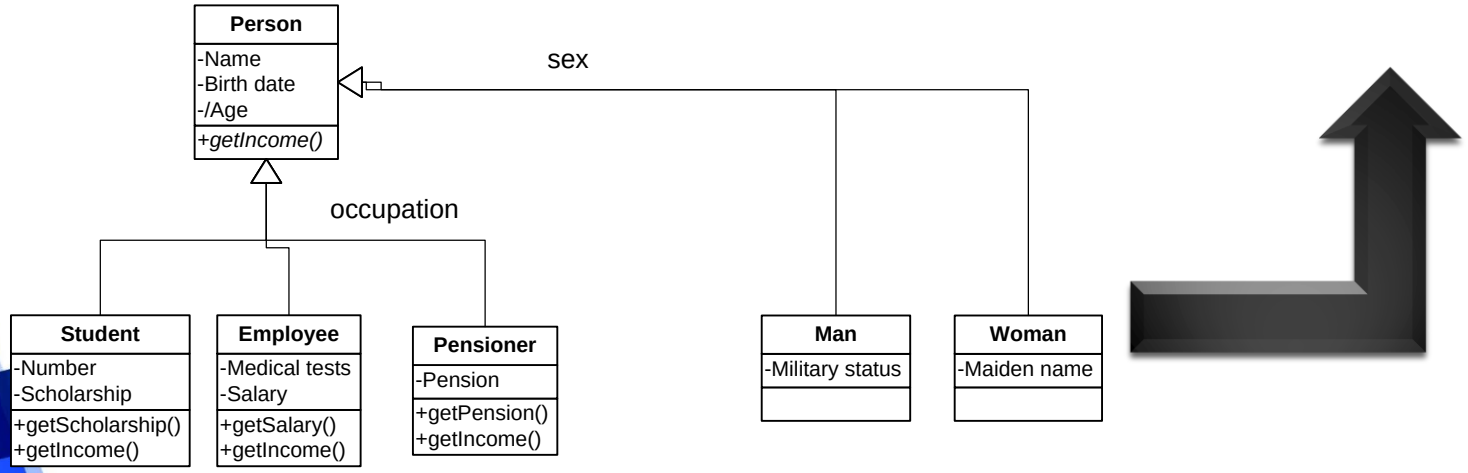
- Sample solution no 2

  One of the hierarchies

  is replaced

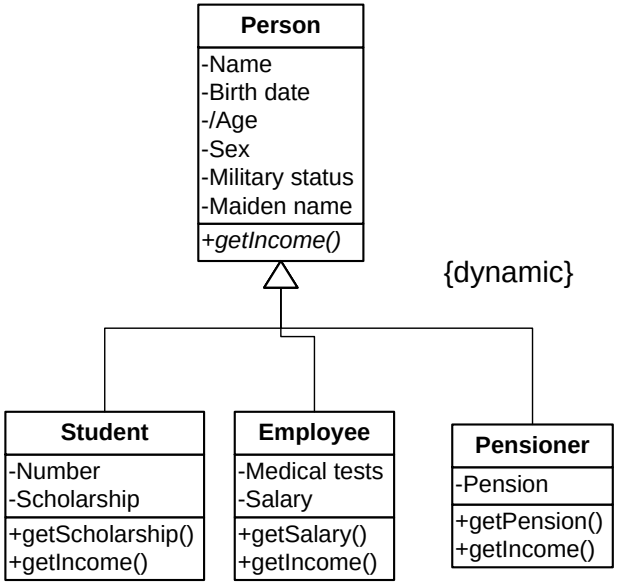
  with a composition.



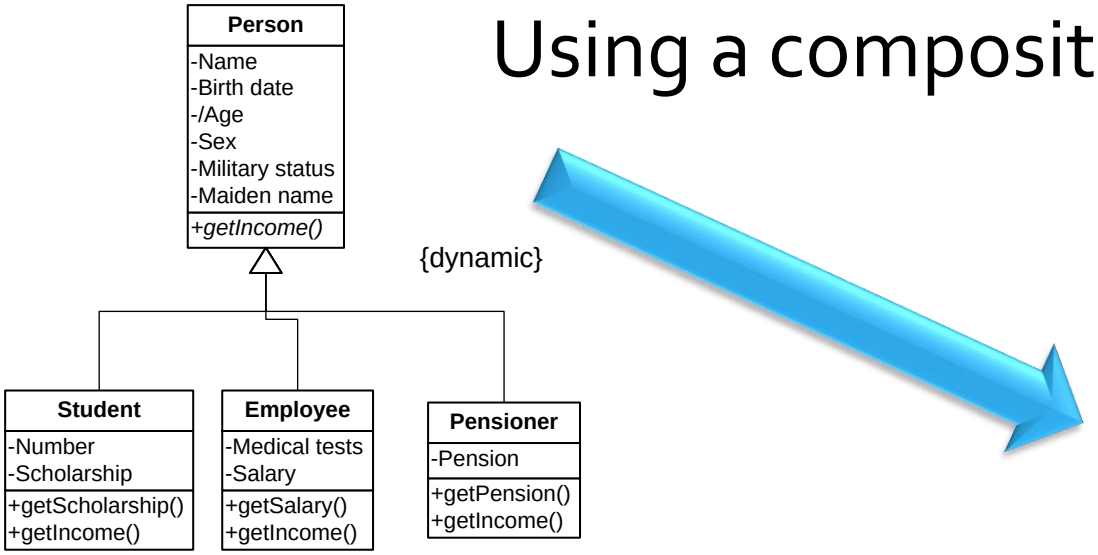


Implementation of the Dynamic Inheritance

- Does not exist directly in popular programming languages (Java, C#, C++).
- It could be implemented:
  - Using an aggregation/composition with the {xor} constraint.
  - By putting all invariants in the super class (with a discriminator),
  - As Smart Objects Copying.

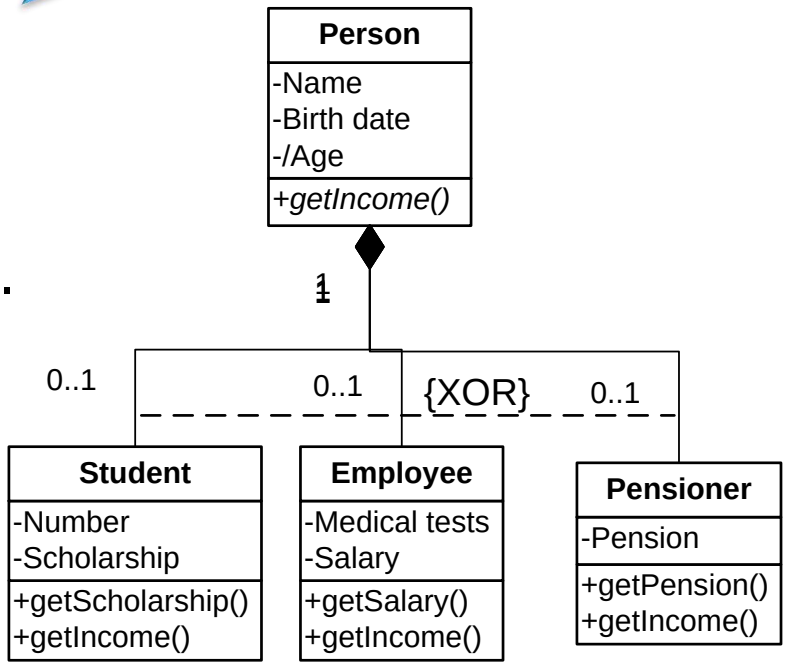


Implementation of the Dynamic Inheritance (2)



Using a composition

- Reusing the code written for the overlapping inheritance.
- Additionally we can add methods for „changing the class”.



Implementation of the Dynamic Inheritance (3)

- Smart Object Copying
  - The idea is based on replacing the old object with a new one. To do this, in each sub class we add additional constructors,
  - Each of them takes a parameter being a reference to the super class object (plus additional business data),
  - The common information is copied from the old object to the new one.

Implementation of the Dynamic Inheritance (4)

- Smart Object Copying – cont.
  - The potential problem is related to updating some of the references pointing to the old object in such a way that they will point to the new one.
    - „Some" means those which are common for the old a new class.
    - The remaining references are lost – just like the attributes' values.
    - In case of ObjectPlusPlus class the solution is quite easy because the library already „knows" objects „pointing at us" – all links in ObjectPlusPlus are bidirectional!
  - Remember also about removing the „old" instance from the extent!

Implementation of the Dynamic Inheritance (5)

- Smart Object Copying – *cont*.

```
public abstract class Person {
   protected String firstName;
   protected String lastName;
   protected LocalDate birthDate;

   // [...]

   @Override
   public String toString() {
      return this.getClass().getSimpleName() + ": " + firstName + " " + lastName;
   }
}
```

```
public class Employee extends Person {
   private boolean medicalTest;
   private float salary;

   public Employee(Person prevPerson, boolean medicalTest, float salary) {
      // Copy the old data
      super(prevPerson.firstName, prevPerson.lastName, prevPerson.birthDate);

      // Remember the new one
      this.medicalTest = medicalTest;
      this.salary = salary;
   }

   // [...]
}
```

Implementation of the Dynamic Inheritance (6)

- Smart Object Copying – *cont*.

```
public static void testInheritanceDynamic_Copying() {
    // Create a student
    Person p1  = new Student("John", "Smith", LocalDate.of(1994, 9, 14), 1212,
2000.0f);
    System.out.println(p1);

    // Create an Employee based on the Student
    p1 = new Employee(p1, true, 4000.0f);
    System.out.println(p1);

    // Create a Pensioner based on the Employee
    p1 = new Pensioner(p1, 3000.0f);
    System.out.println(p1);
}
```

```
Student: John Smith
Employee: John Smith
Pensioner: John Smith
```

Implementation of the Dynamic Inheritance (7)

- Smart Object Copying – *cont*.
  - We need to take care of:
    - Updating the references pointing at our new object,
    - Removing from the extent the old object.
  - In case of `ObjectPlusPlus` required information is stored within the library. There is only a necessity of adding a couple of method which perform the operation.

Homework for volunteers?

Pros and Cons of the Presented Approaches

All kinds of inheritance could be faked using one (or many) of the following techniques:

- Replacing the hierarchy with a single class
  - Easy to implement. Sometimes delusive, e.g. replacement of methods overriding with many casees or ifs.
  - Easy to use.
  - Wasting of resources.

Pros and Cons of the Presented Approaches (2)

- Utilization of the aggregation/composition
  - Optimal resource management,
  - Time-consuming implementation (i.e. „wrapper" methods), but not complicated.
- Interface utilization
  - Time-consuming,
  - The above problem could be minimized using a delegation design pattern.
  - Wide possibilities.

The Summary

- There is only a simplest inheritance in the popular programming languages.
- The rest of them has to be implemented manually using different approaches.
- On contrary to the associations there is no perfect solution. Each case has to be analyzed individually.
- All discussed solutions are some kinds of tricks and cannot be treated as an inheritance itself.
- **Hence wherever possible the „real" inheritance has to be used rather then its substitutes.**

Source files

Download source files for all MAS lectures

http://www.mtrzaska.com/plik/mas/mas-source-files-lectures