POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Modelowanie i Analiza Systemów informacyjnych (MAS)

dr inż. Mariusz Trzaska
mtrzaska@pjwstk.edu.pl

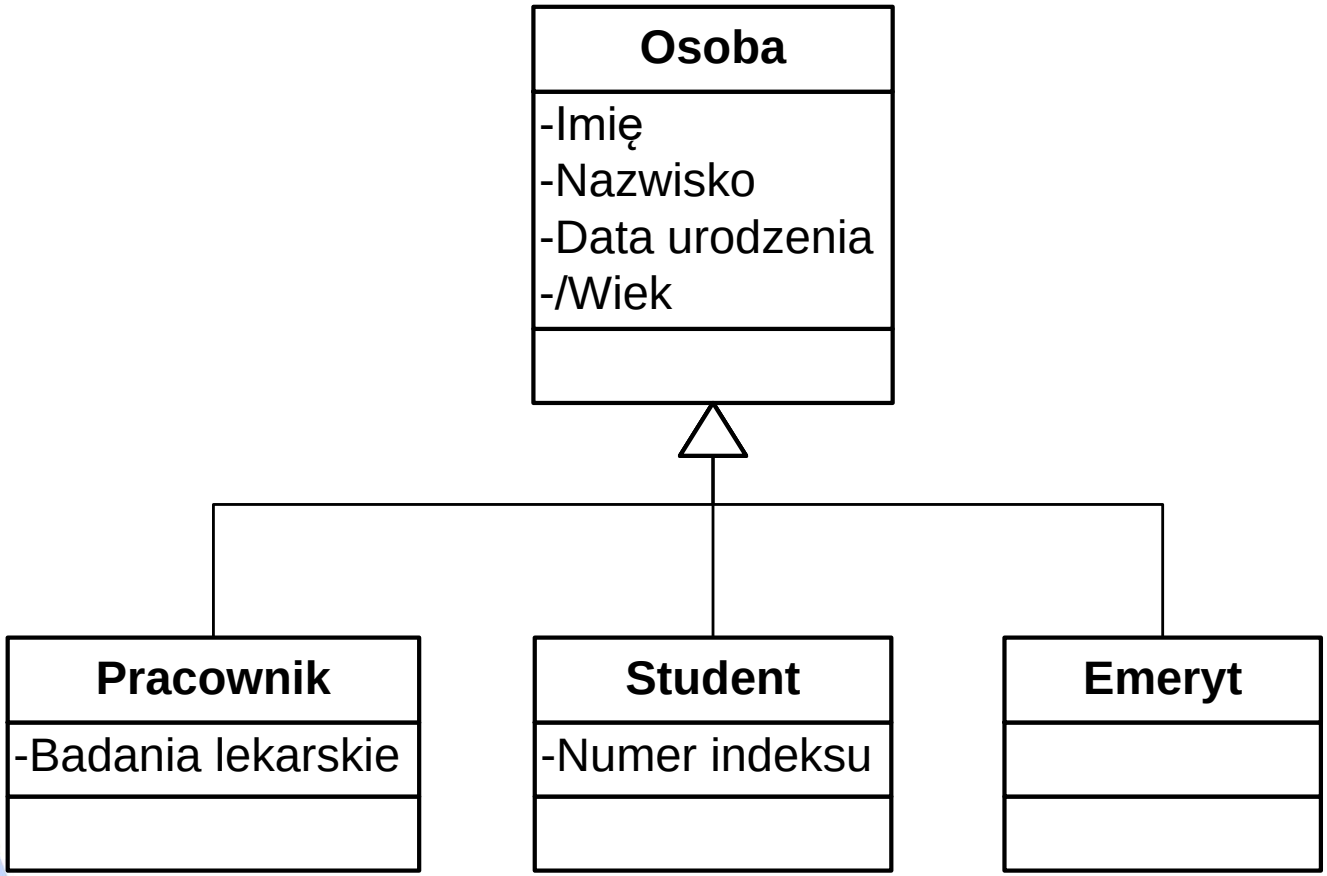
Wykład 7

Realizacja różnych modeli dziedziczenia w obiektowych językach programowania

Zagadnienia

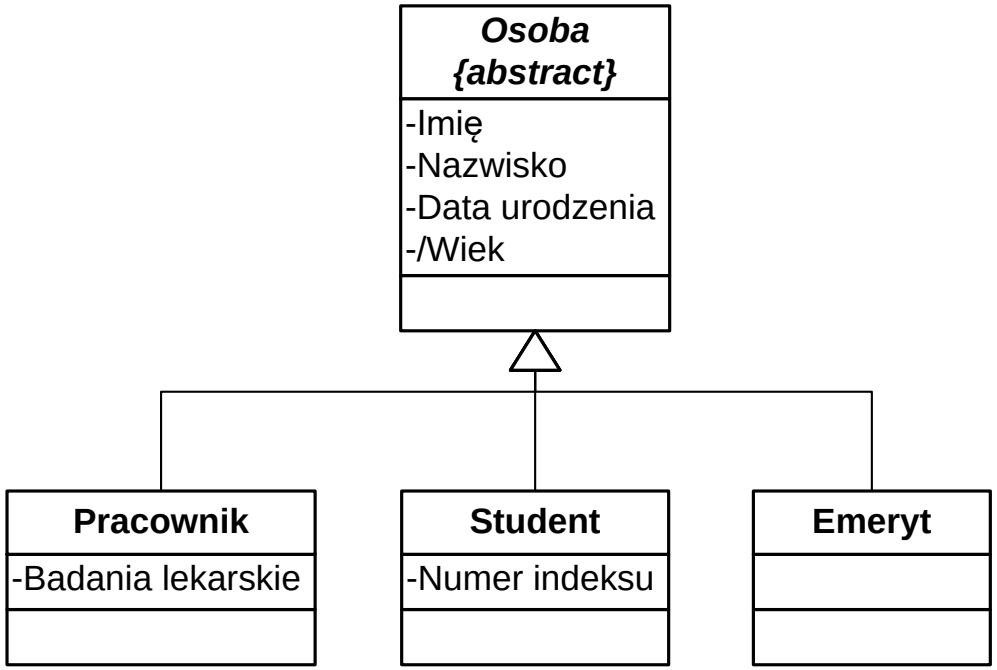
- Omówienie różnych rodzajów dziedziczenia, klas abstrakcyjnych oraz polimorficznego wołania metod.
- Realizacja podstawowego dziedziczenia
- Wykorzystanie klas abstrakcyjnych oraz polimorficznego wołania metod.
- Implementacja pozostałych rodzajów dziedziczenia:
  - overlapping,
  - complete, incomplete,
  - multi-inheritance,
  - multi-aspect,
  - dynamic.
- Wady i zalety poszczególnych rozwiązań.
- Podsumowanie

Dziedziczenie rozłączne (*disjoint)*



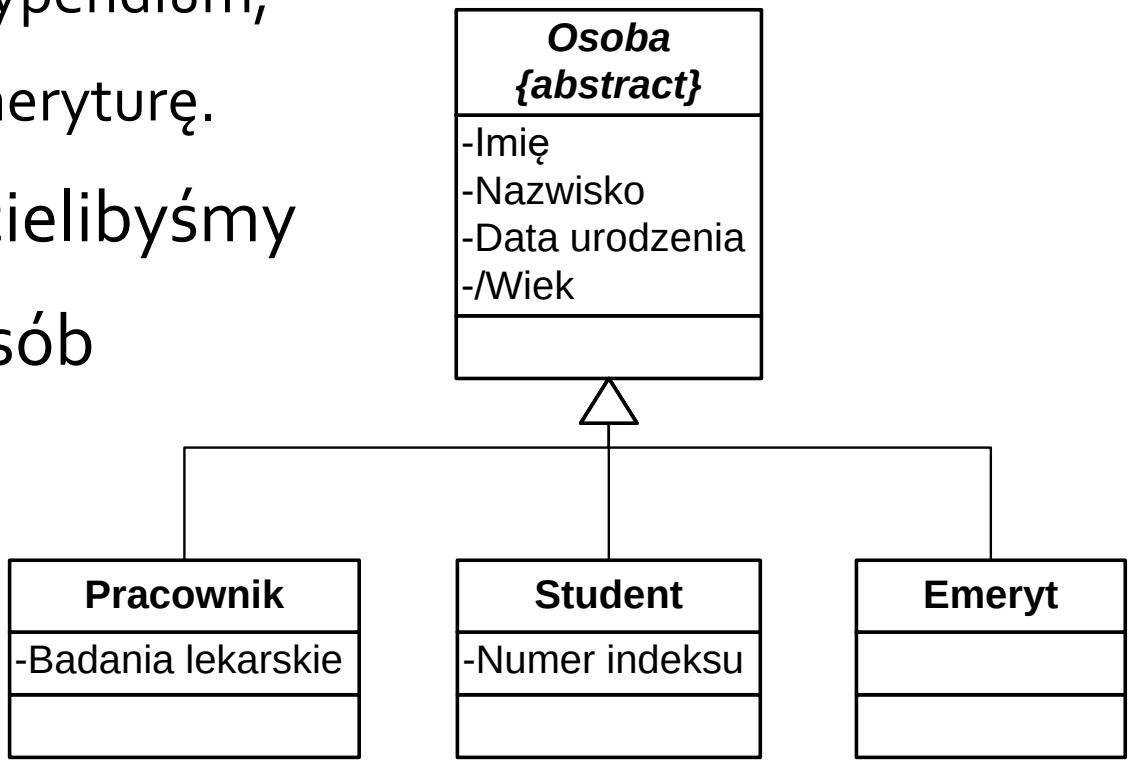
Klasa abstrakcyjna

- Klasa, która nie może mieć bezpośrednich wystąpień.
- Wykorzystywana do tworzenia hierarchii dziedziczenia.



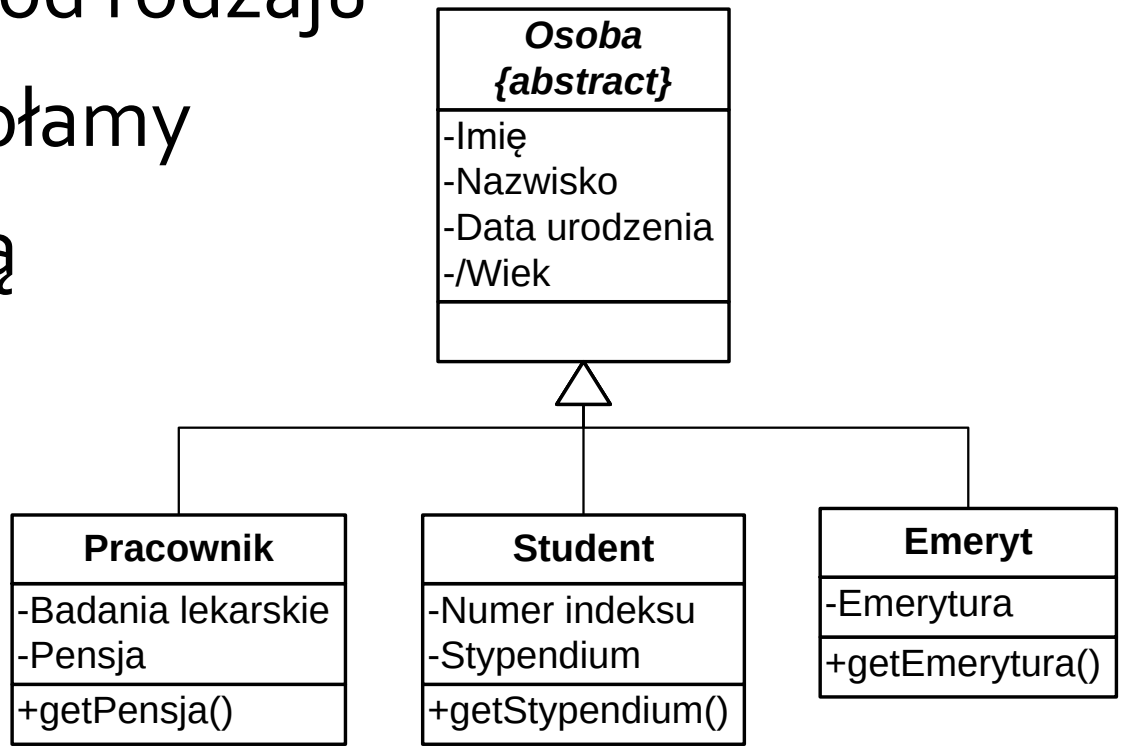
Problem biznesowy

- Załóżmy, że osoby z diagramu mają jakieś dochody:
  - Pracownik ma pensję,
  - Student ma stypendium,
  - Emeryt ma emeryturę.
- I oczywiście chcielibyśmy mieć jakiś sposób zapytania o te dochody.



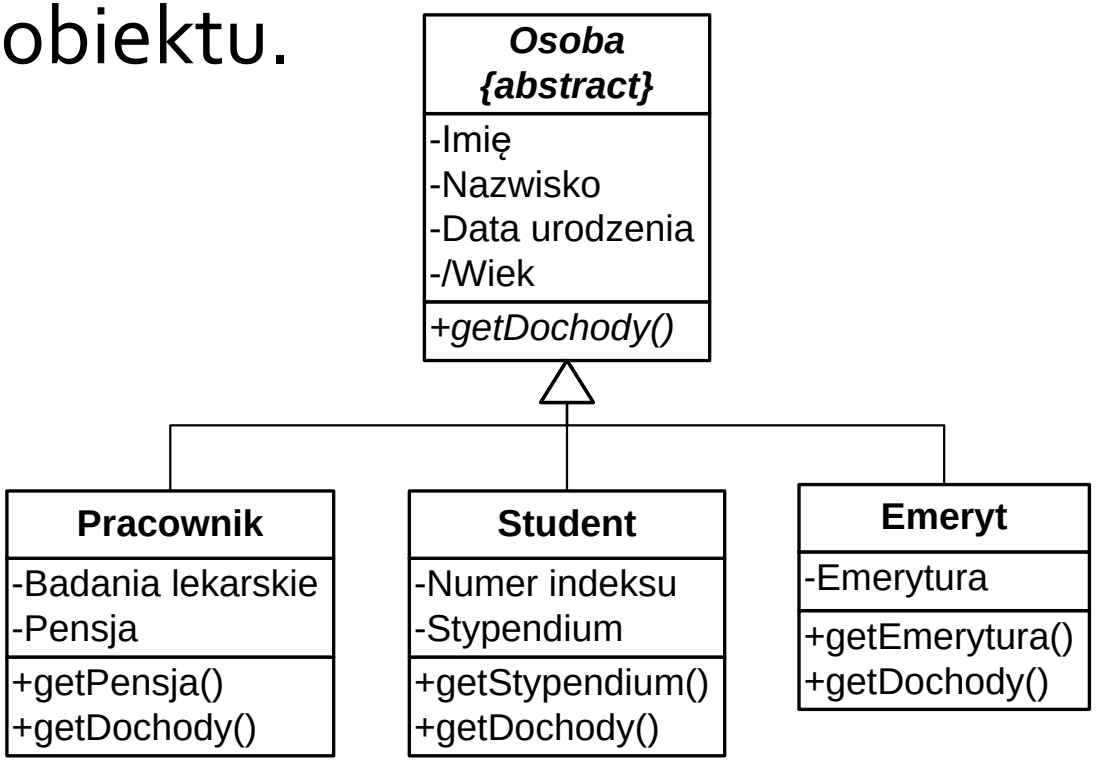
Problem biznesowy (2)

- Najprostszym sposobem wydaje się umieszczenie atrybutów w poszczególnych klasach i dodanie odpowiednich metod
- W zależności od rodzaju osoby, wywołamy odpowiednią metodę.
- Jakiś lepszy sposób?



Polimorficzne wołanie metod

- Wykorzystuje przesłanianie.
- Umożliwia wykonywanie operacji bez „ręcznego” sprawdzania konkretnego rodzaju obiektu.

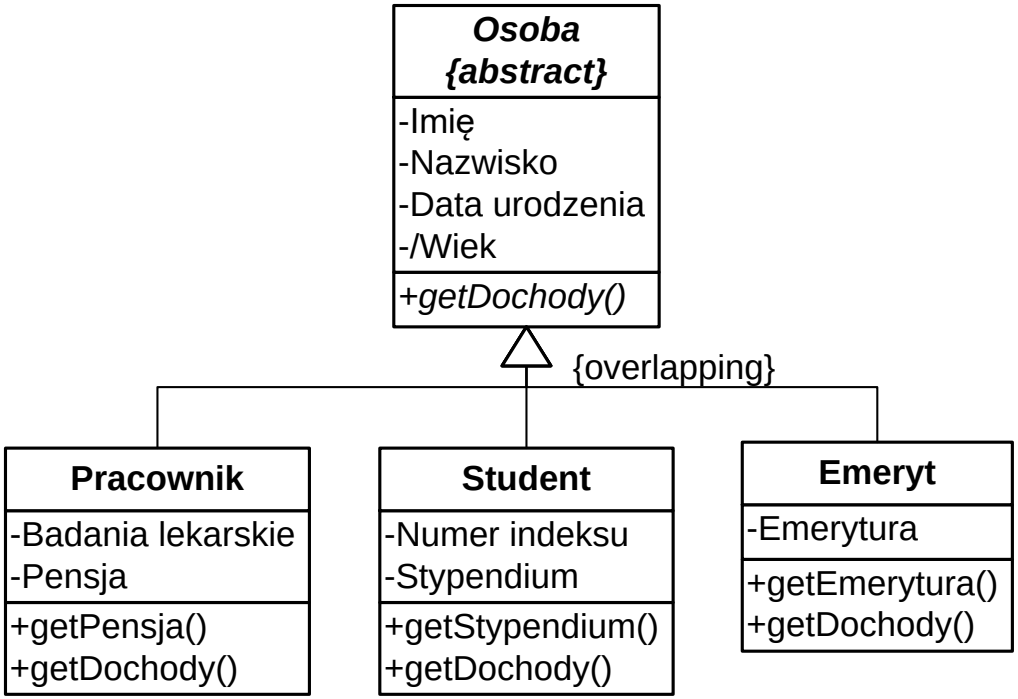


Metody abstrakcyjne

- Jaki kod będzie znajdował się w metodzie `getDochody()` w klasie `Osoba`?
- Przecież osoba jako taka nie ma dochodów (mają je tylko jej specjalizacje).
- Rozwiązanie: Oznaczmy ją jako metodę abstrakcyjną.
- Metoda abstrakcyjna:
  - nie ma ciała,
  - musi zostać zaimplementowana w podklasach,
  - może być tylko w klasie abstrakcyjnej.

Pozostałe rodzaje dziedziczenia

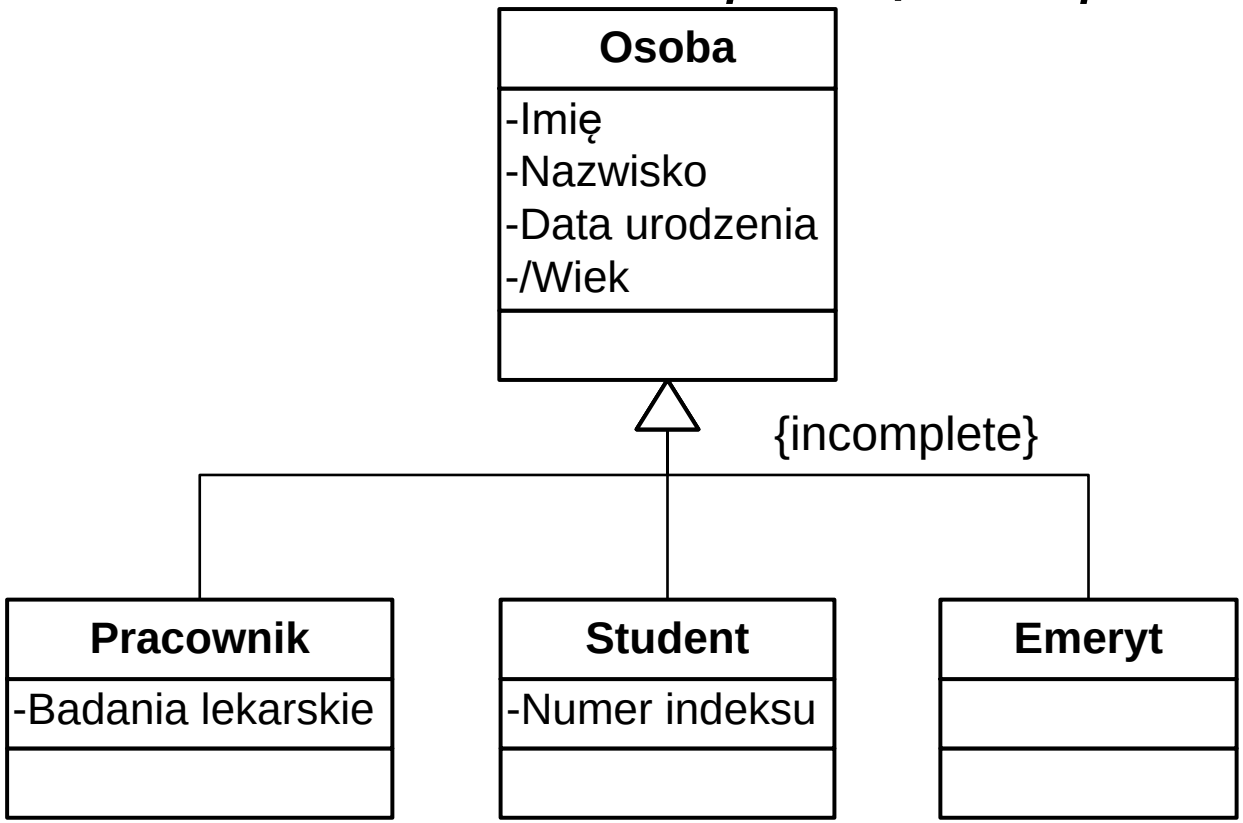
- Dziedziczenie łączne (*overlapping)*



- Co z przesłanianiem i polimorficznym wołaniem metod?

Pozostałe rodzaje dziedziczenia (2)

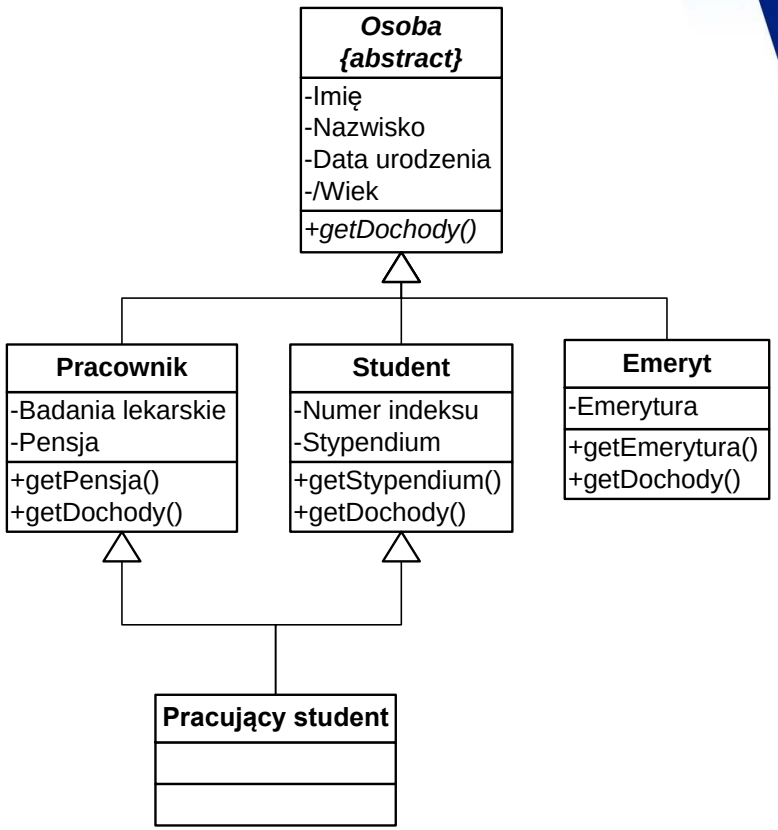
- Dziedziczenie *incomplete, complete*



- Dziedziczenie *ellipsis* (*trzy kropki*)

Pozostałe rodzaje dziedziczenia (3)

- Wielodziedziczenie (dziedziczenie wielokrotne, *multi-inheritance*)
  - w przeciwieństwie do dz. łącznego (overlapping), mamy możliwość dodania atrybutów w podklasie.

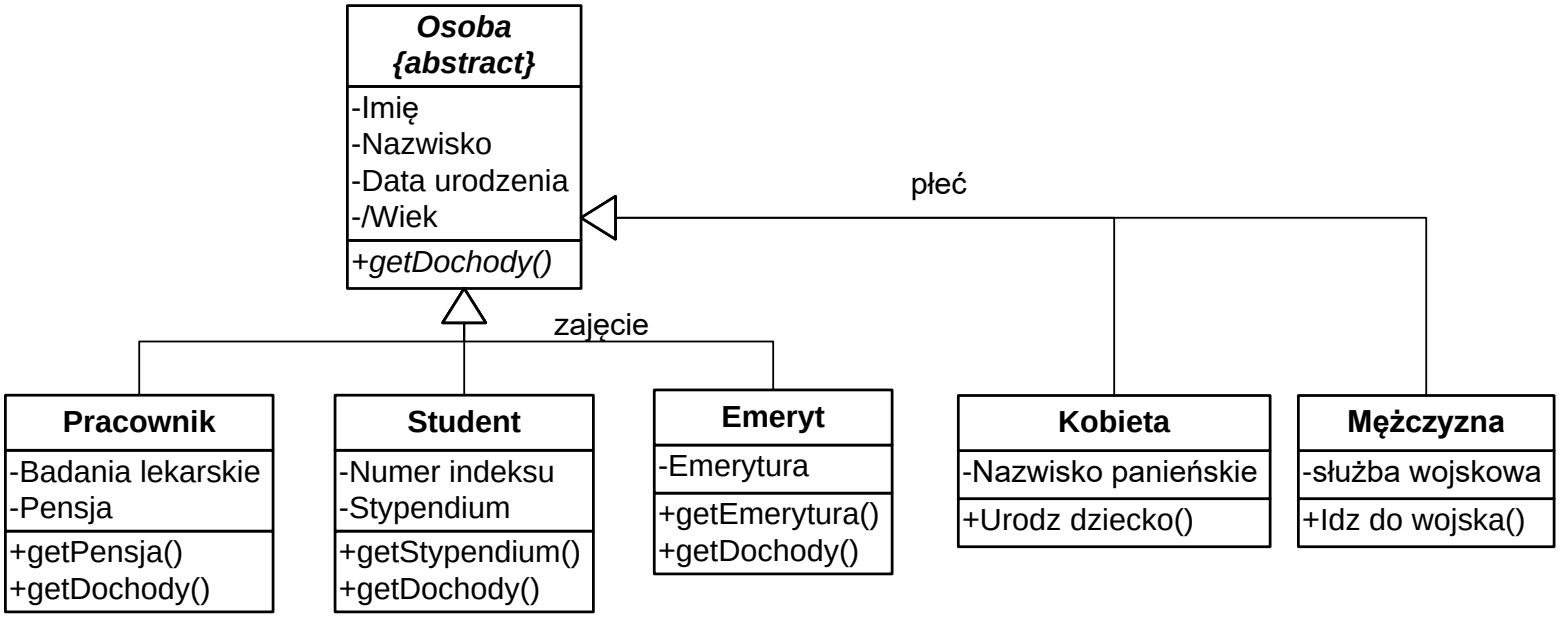


Pozostałe rodzaje dziedziczenia (4)

- Wielodziedziczenie (dziedziczenie wielokrotne, *multi-inheritance*) – c. d.:
  - Problemy,
  - Idealne rozwiązanie?
  - Co z przesłanianiem i polimorficznym wołaniem metod?

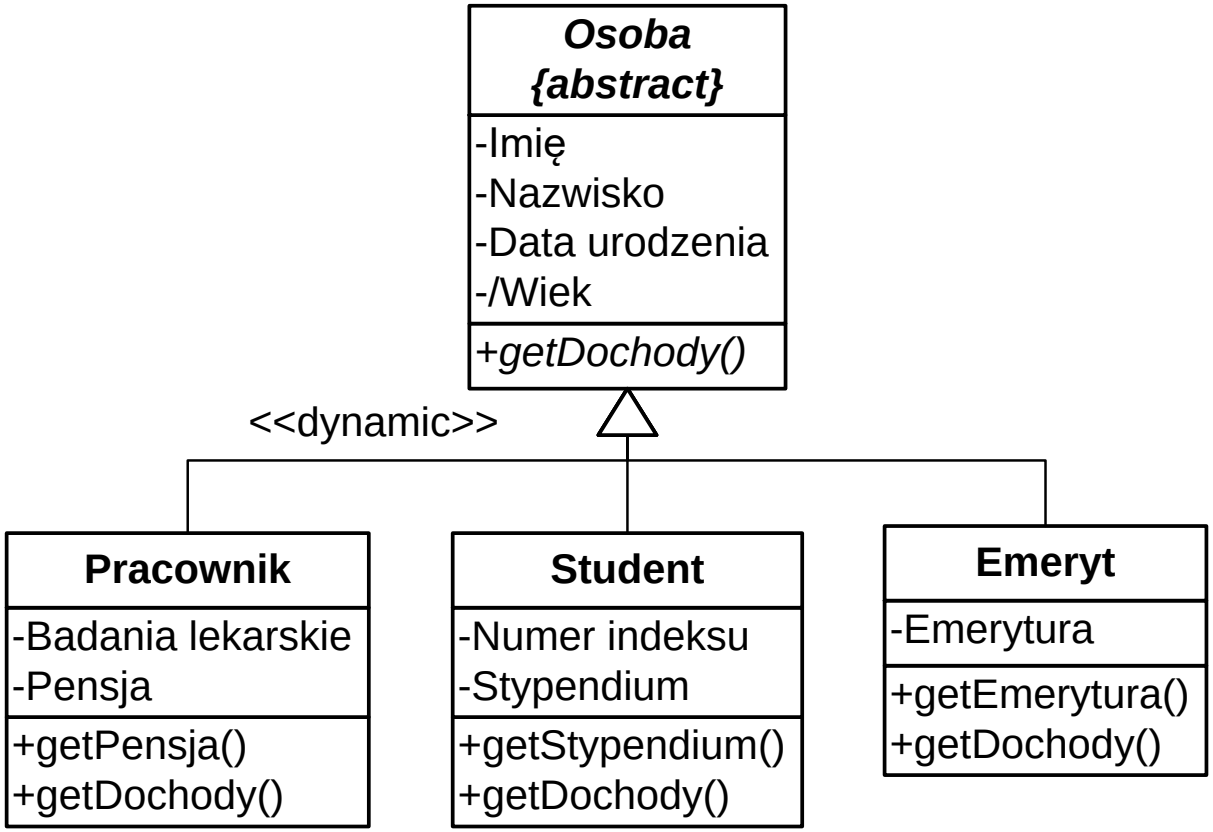
Pozostałe rodzaje dziedziczenia (5)

- Dziedziczenie wieloaspektowe (*multi-aspect*)
  - Co z przesłanianiem i polimorficznym wołaniem metod?



Pozostałe rodzaje dziedziczenia (6)

- Dziedziczenie dynamiczne (*dynamic*)



Dziedziczenie, a obiektowe języki programowania

- W większości przypadków, popularne języki programowania posiadają najprostszy rodzaj dziedziczenia:
  - Rozłączne (*disjoint*),
  - Wielokrotne (tylko C++).
- Co z pozostałymi rodzajami?
  - Różne metody obejścia,
  - Implementacja.
- Co z klasami i metodami abstrakcyjnymi?
- Co z polimorficznym wołaniem metod?

Realizacja dziedziczenia disjoint

- Ten typ dziedziczenia występuje bezpośrednio w popularnych językach programowania.

```
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;
}
```

Wykorzystanie polimorficznego wołania metod

- W języku Java:
  - klasy abstrakcyjne,
  - metody abstrakcyjne,
  - polimorficzne wołanie metod

  występują bezpośrednio.
- W języku C++ powyższe pojęcia również występują, z tym, że chęć korzystania z polimorficznego wołania metod należy zadeklarować za pomocą słowa kluczowego `virtual`.

Wykorzystanie polimorficznego wołania metod (2)

- Sposób wykorzystania podobny do klasycznego dziedziczenia *disjoint*.

```
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; }
}
```

- Pozostałe klasy są zaimplementowane analogicznie do powyższych.

Wykorzystanie polimorficznego wołania metod (3)

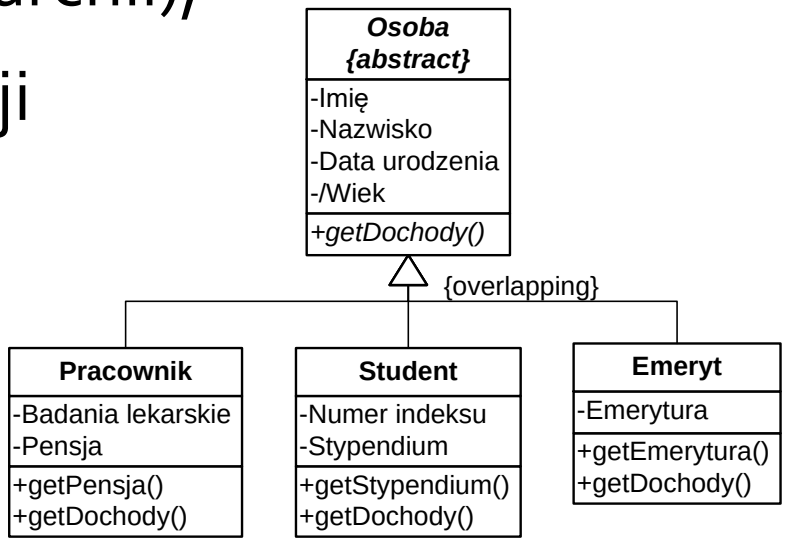
- Tworzymy dwa obiekty:
  - Pracownika,
  - Studenta.
- Traktujemy je po prostu jako osoby (referencja do typu osoba)
- Każdą z nich pytamy o dochody (bez sprawdzania z jaką klasą mamy do czynienia).
- Dzięki polimorficznemu wołaniu metody, dostajemy odpowiedzi właściwe dla poszczególnych rodzajów osób.

```
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
```

Realizacja dziedziczenia overlapping

- Ten typ dziedziczenia nie występuje bezpośrednio w popularnych językach programowania.
- Sposoby obejścia:
  - Zastąpienie całej hierarchii dziedziczenia jedną klasą (spłaszczenie hierarchii),
  - Wykorzystanie agregacji lub kompozycji,
  - Rozwiązania łączące powyższe metody.



Realizacja dziedziczenia overlapping (2)

- Zastąpienie całej hierarchii dziedziczenia jedną klasą
  - Wszystkie inwarianty umieszczamy w jednej nadklasie,
  - Dodajemy dyskryminator, który informuje nas o rodzaju obiektu (używamy `EnumSet` ponieważ chcemy przechowywać informacje o kilku rodzajach na raz).

```
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);

   // [...]
}
```

Realizacja dziedziczenia overlapping (3)

- Zastąpienie całej hierarchii dziedziczenia jedną klasą – c. d.
  - Kod sprawdzający rodzaj osoby w metodach,

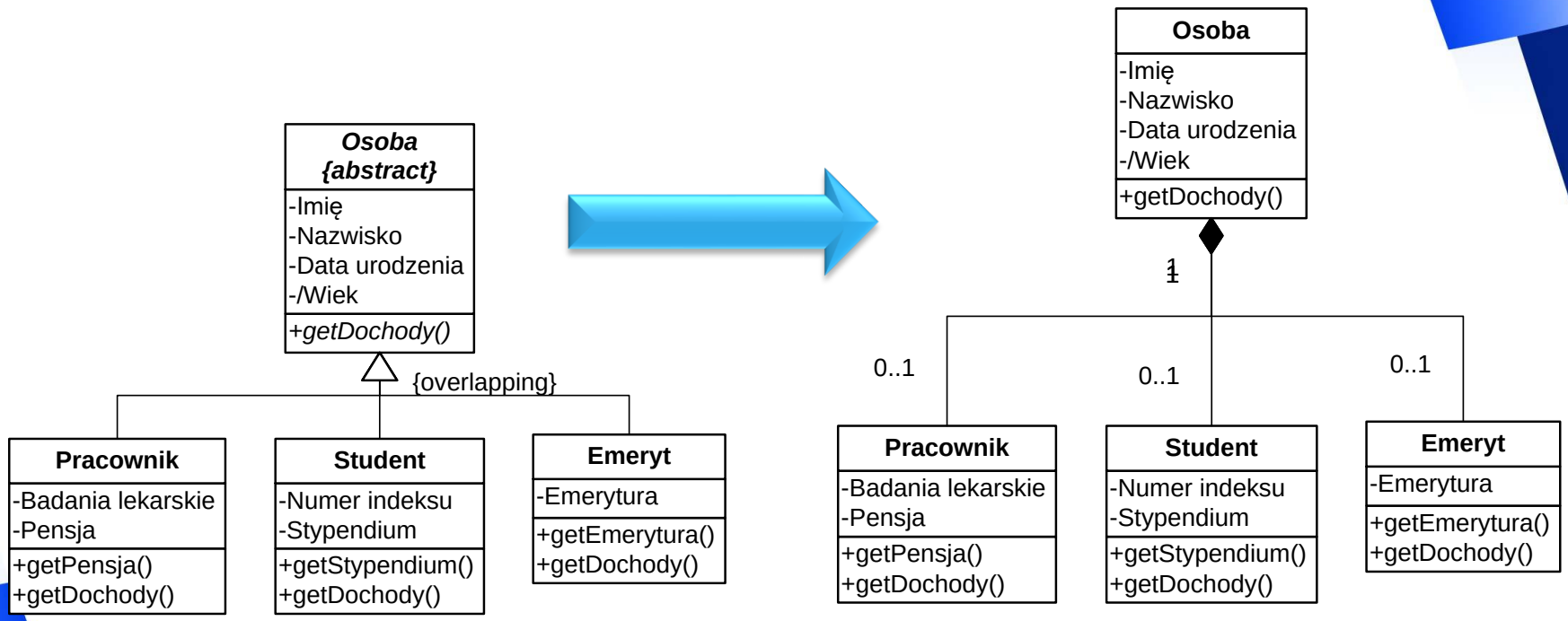
```
public class Person {
   private boolean medicalTest;

   // [...]

   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!");
      }
   }
}
```

Realizacja dziedziczenia overlapping (4)

- Zastąpienie całej hierarchii dziedziczenia jedną klasą – c. d.
  - Zalety
    - Prostota realizacji
    - Łatwość używania
  - Wady
    - Brak możliwości korzystania z konstrukcji związanych z dziedziczeniem, np. przesłanianie metod, polimorficzne wołanie metod, itd.
    - Niewykorzystywanie inwariantów należących do innej specjalizacji (mimo tego, że zajmują miejsce).

Realizacja dziedziczenia overlapping (5)

- Wykorzystanie agregacji lub kompozycji



Realizacja dziedziczenia overlapping (6)

- Wykorzystanie agregacji lub kompozycji – c.d.
  - Asocjacje z podklas pokazują na:
    - Całość. Trzeba też zmodyfikować połączenia asocjacji (z podklas przenieść do nadklasy).
    - Część. W takiej sytuacji, obiekty-części nie mogą być ukryte. Musi być do nich dostęp bezpośredni (nie przez obiekt-całość).
  - Agregacja lub kompozycja implementowane na jeden ze wcześniej poznanych sposobów.
  - Wykorzystanie klasy *ObjectPlusPlus* zaoszczędzi nam sporo pracy.

Realizacja dziedziczenia overlapping (7)

- Wykorzystanie agregacji lub kompozycji – c.d.
  - Dodatkowe metody:
    - Dające dostęp do atrybutów znajdujących się w obiektach „po drugiej stronie" agregacji,
    - Dające dostęp do powiązań znajdujących się w obiektach „po drugiej stronie" agregacji.

Realizacja dziedziczenia overlapping (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);
   }

   //[...]
```

Realizacja dziedziczenia overlapping (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 final static String roleNameEmployee = "specializationEmployee";
   private final static String roleNamePensioner = "specializationPensioner";
```

Realizacja dziedziczenia overlapping (10)

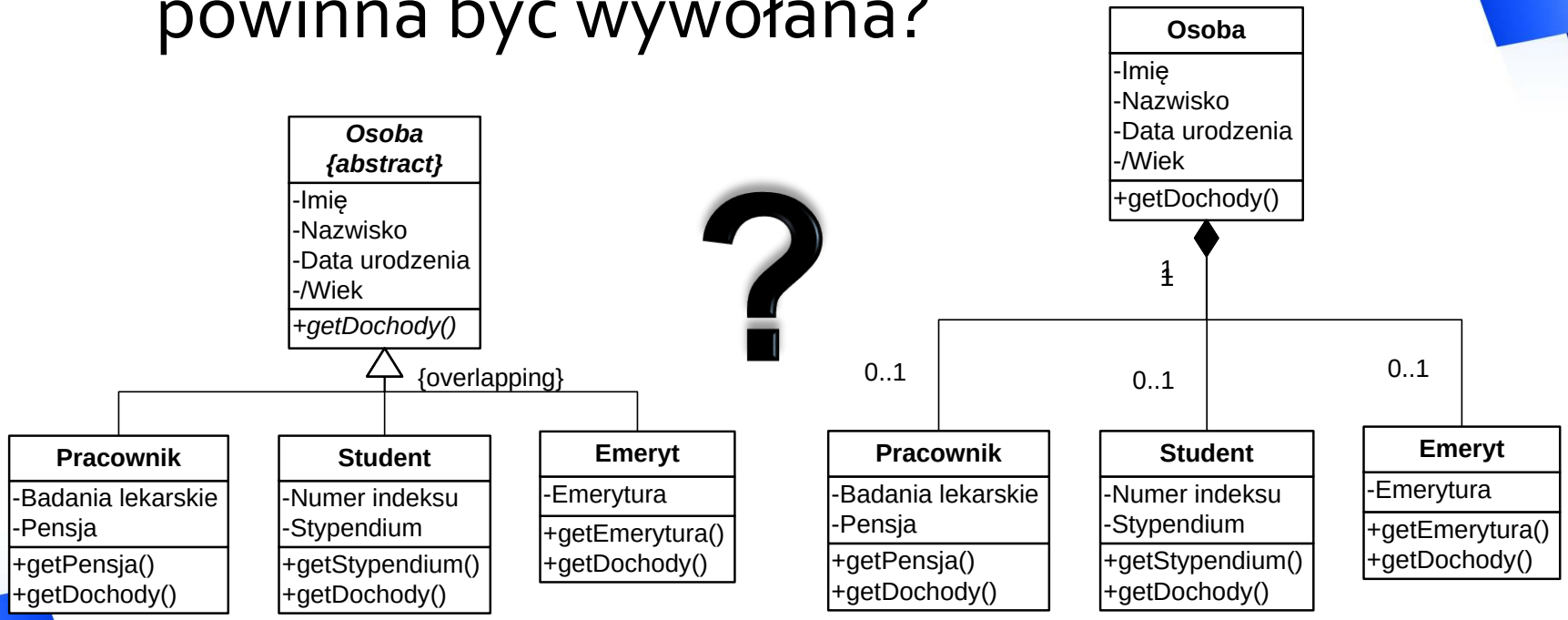
```
public class Person extends ObjectPlusPlus {
   // [...]

   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!");
      }
   }
```

Realizacja dziedziczenia overlapping (11)

- Wykorzystanie agregacji lub kompozycji – c.d.
  - Zalety
    - Łatwość używania (gdy dodamy odpowiednie metody)
    - Korzystamy tylko z tych inwariantów, których rzeczywiście potrzebujemy.
  - Wady
    - Brak możliwości korzystania z konstrukcji związanych z dziedziczeniem, np. przesłanianie metod, polimorficzne wołanie metod, itd. Można to tylko symulować tworząc specjalne, dodatkowe metody.

Polimorfizm w dziedziczeniu overlapping

- Która wersja metody (z której klasy) powinna być wywołana?



- Chyba żadna...

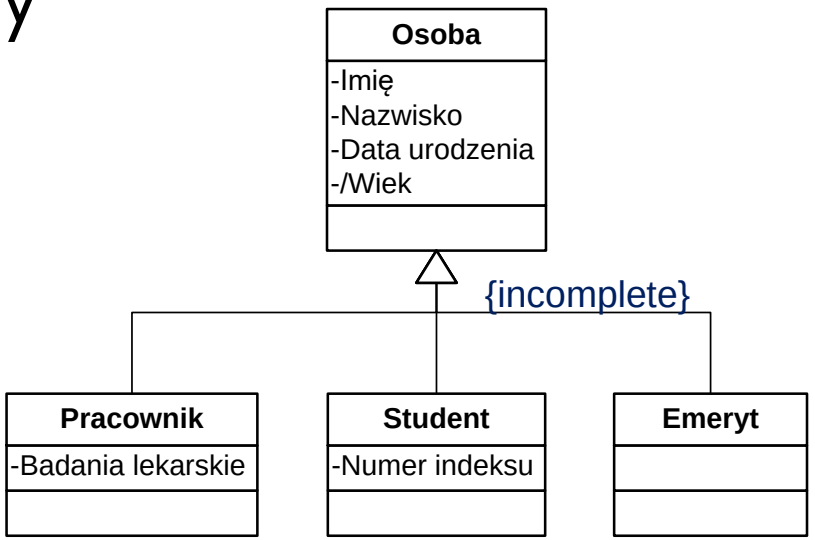
Polimorfizm w dziedziczeniu overlapping (2)

- Trzeba stworzyć nową metodę, która w zależności od rodzajów(!) obiektów, uwzględni odpowiednie dochody(!)

```
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;
}
```

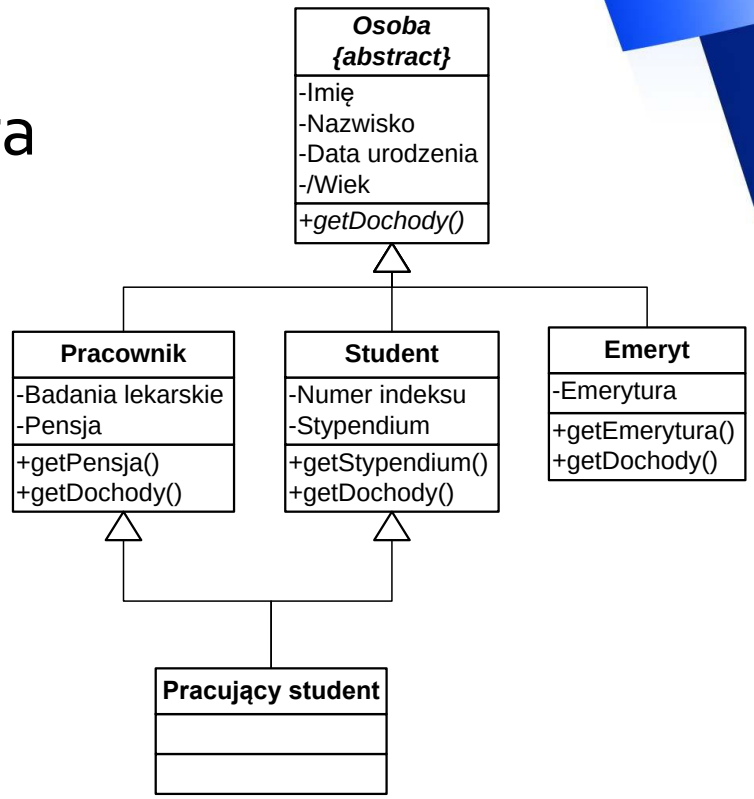
Dziedziczenie *complete* oraz *incomplete*

- Co te rodzaje dziedziczenia znaczą dla diagramu?
- Czy coś znaczą dla implementacji?
- Przeważnie nie.
- Ewentualnie rozważamy ich wpływ na późniejszą implementację.



Implementacja wielodziedziczenia

- Występuje w języku C++
  - W przypadku konfliktu nazw używamy operatora zakresu.
- Nie występuje w języku Java ani w MS C#.
- W związku z tym, w jaki sposób możemy je zaimplementować?



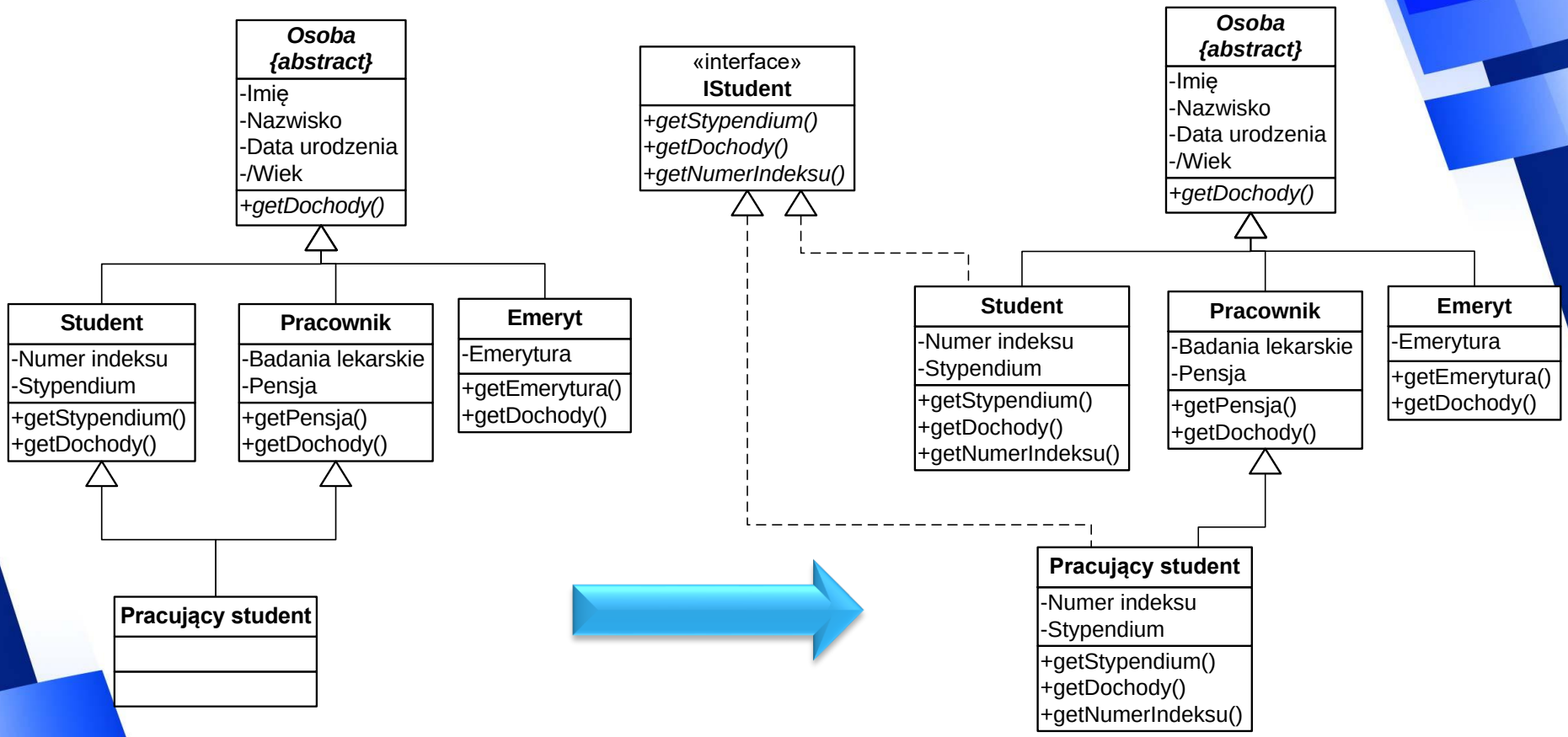
Implementacja wielodziedziczenia (2)

- Implementujemy je korzystając ze sposobów podanych przy okazji dziedziczenia overlapping:
  - Jedna klasa (spłaszczenie hierarchii),
  - Agregacja, kompozycja
- Możemy także wykorzystać interfejsy.

Implementacja wielodziedziczenia z wykorzystaniem interfejsów

- Klasa może implementować dowolną liczbę interfejsów.
- Ze względu na ograniczenia interfejsów, korzystamy tylko z metod (brak atrybutów).
- Powyższy problem możemy częściowo rozwiązać używając get/set.
- Czasami występuje konieczność wielokrotnego implementowania takich samych metod i do tego w ten sam sposób.
- Java 8+: metody `default` oraz `static`.

Implementacja wielodziedziczenia z wykorzystaniem interfejsów (2)



Implementacja wielodziedziczenia z wykorzystaniem interfejsów (3)

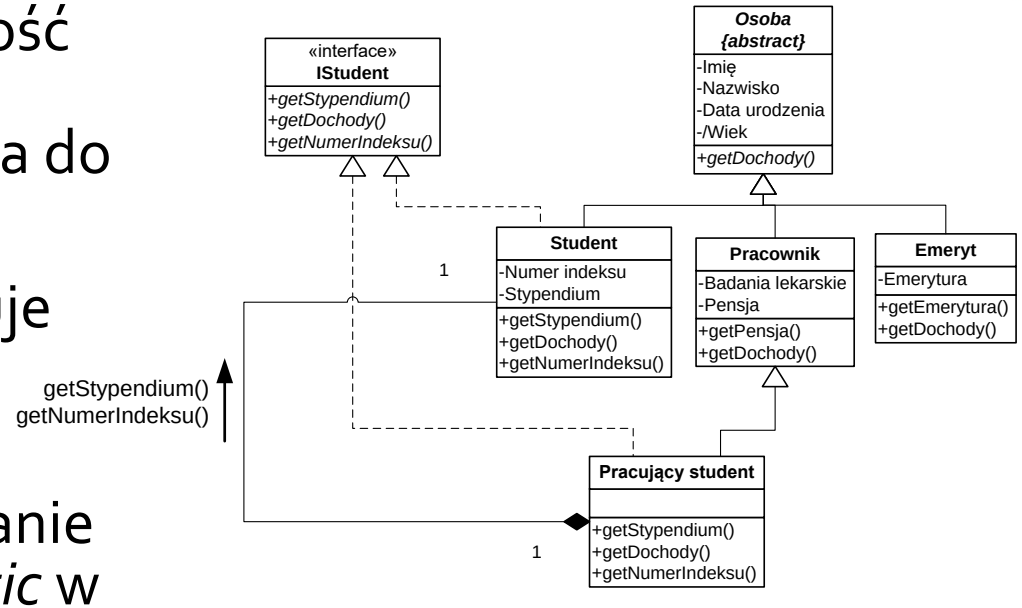
```
public interface IStudent {
   float getIncome();
   float getScholarship();
   void setScholarship(float scholarship);
   int getNumber();
}
```

```
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();
   }

   // [...]
}
```

Jak widać musieliśmy pewne metody (np. `getScholarship()`) implementować kilka razy (i do tego tak samo).

Implementacja wielodziedziczenia z wykorzystaniem interfejsów (4)

- Częściowe rozwiązanie problemu wielokrotnej implementacji tych samych metod.
  - Klasa `PracującyStudent` dziedziczy funkcjonalność pracownika i deleguje funkcjonalność studenta do podłączonego obiektu.
  - Innymi słowy: opakowuje funkcjonalność klasy `Student`.
  - Ewentualne wykorzystanie metod *default* i/lub *static* w interfejsach (Java 8+).



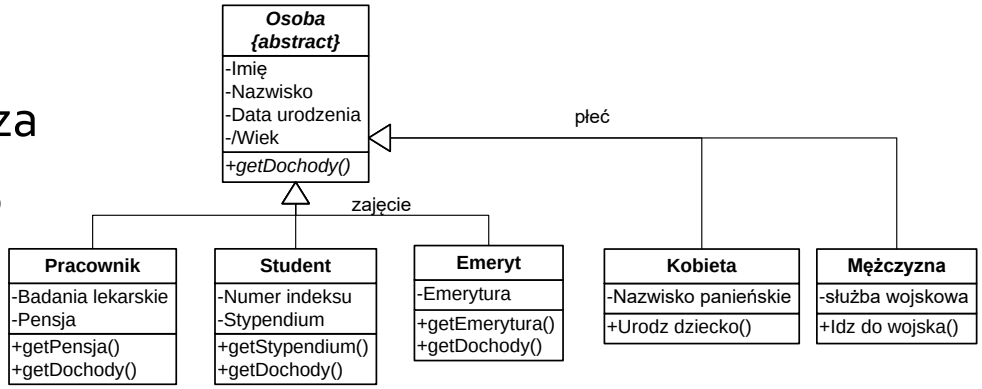
Implementacja wielodziedziczenia z wykorzystaniem interfejsów (5)

- Pewien niepokój może budzić pamiętanie niektórych atrybutów dwa razy (np. imię, nazwisko): raz w klasie `PracujacyStudent`, a drugi raz w podłączonym obiekcie klasy `Student`.
  - Modyfikacja klasy `Student`,
  - Przekazanie `null`'i do obiektu klasy `Student`.

```
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);
   }
}
```

Implementacja dziedziczenia wieloaspektowego

- Nie występuje bezpośrednio w żadnym popularnym języku programowania (Java, C#, C++).
- Trzeba je zaimplementować:
  - Jeden aspekt dziedziczymy używając wbudowanych prostych mechanizmów dziedziczenia danego języka programowania.
  - Pozostałe aspekty:
    - Implementujemy za pomocą jednego z wcześniej omawianych sposobów,
    - Usuwamy, dodając np. flagi do głównej klasy.



Implementacja dziedziczenia wieloaspektowego (2)

- Który aspekt powinniśmy dziedziczyć?
  - Tam gdzie występuje przesłanianie metod, polimorficzne wołanie,
  - Tam gdzie jest większe zróżnicowanie atrybutów w poszczególnych podklasach.
  - Innymi słowy – najbardziej skomplikowaną/rozbudowaną hierarchię.

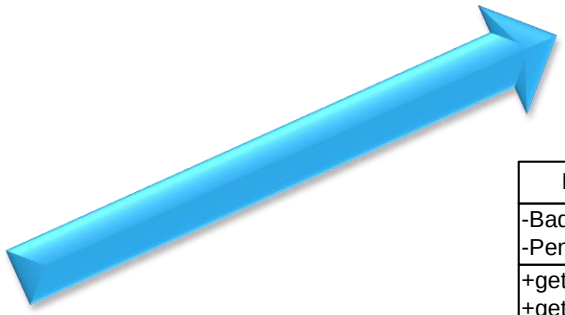
Implementacja dziedziczenia wieloaspektowego (3)

- W niektórych sytuacjach, gdy:
  - nie przechowujemy informacji specyficznych dla danego aspektu, a tylko informację o rodzaju obiektu, możemy dziedziczenie zastąpić np. flagą umieszczoną w nadklasie.
  - specyficznych informacji jest mało, możemy je umieścić w nadklasie i również całkowicie zrezygnować z jednego aspektu dziedziczenia.

Implementacja dziedziczenia wieloaspektowego (4)

- Przykładowe rozwiązanie nr 1

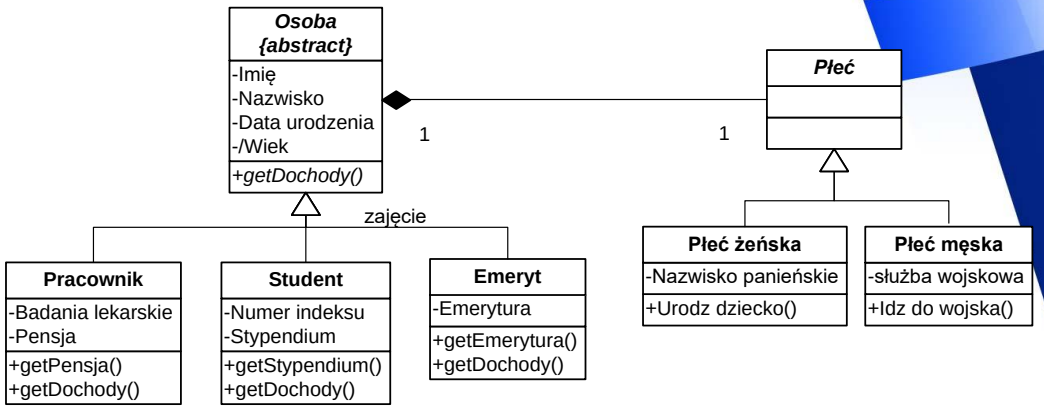
  Atrybuty i metody ze zlikwidowanego aspektu umieszczamy w nadklasie.

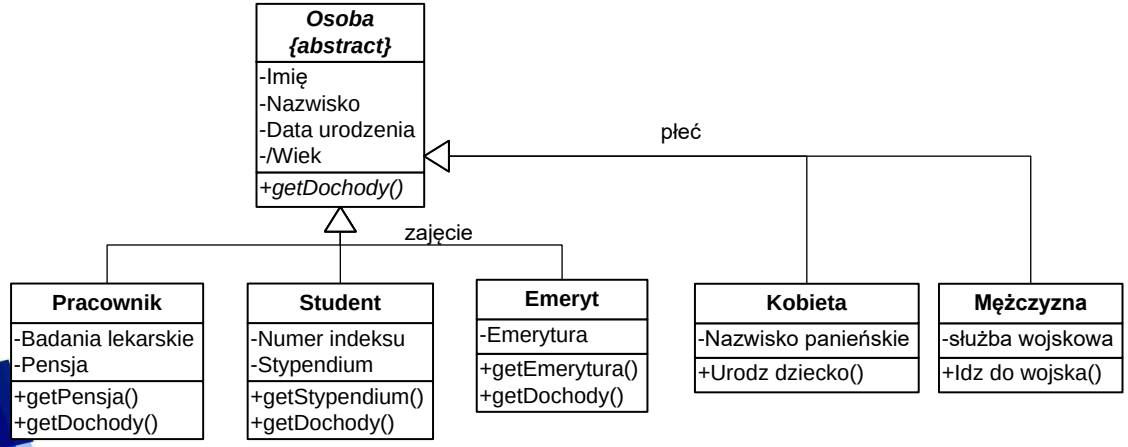


Implementacja dziedziczenia wieloaspektowego (5)

- Przykładowe rozwiązanie nr 2

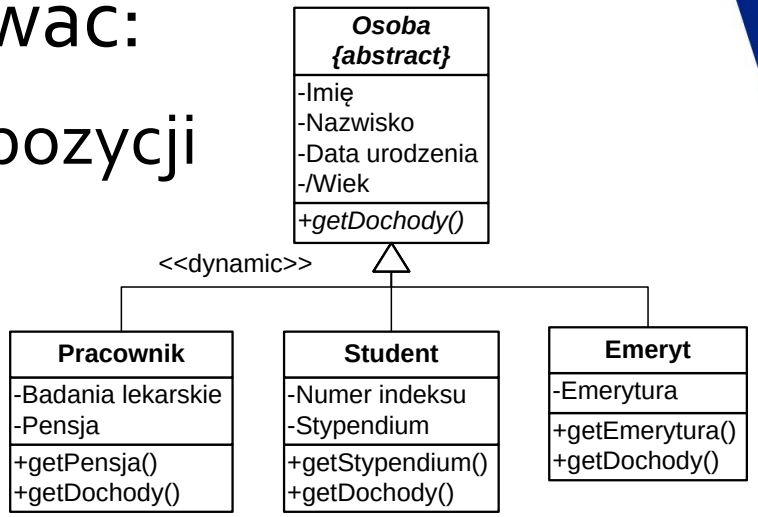
Jedną z hierarchii zastępujemy kompozycją.





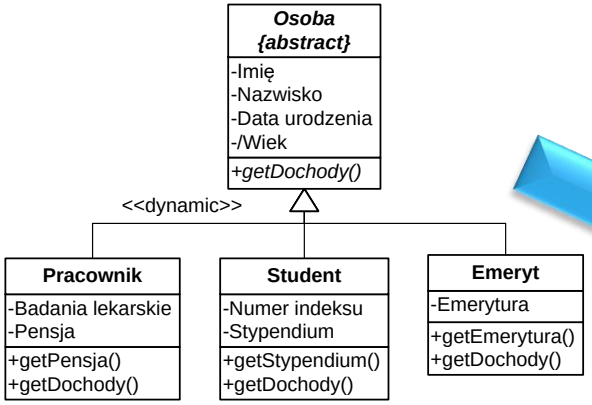
Implementacja dziedziczenia dynamicznego

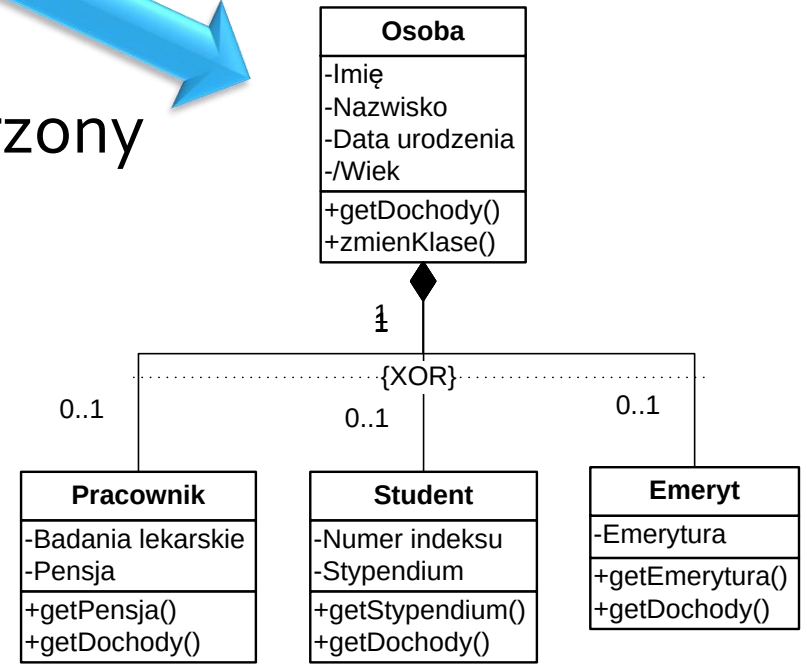
- Nie występuje bezpośrednio w żadnym popularnym języku programowania (Java, C#, C++).
- Trzeba je zaimplementować:
  - Używając agregacji/kompozycji z ograniczeniem {xor},
  - Umieszczając wszystkie inwarianty w nadklasie i dodając dyskryminator,
  - „Sprytnie” kopiując obiekty.



Implementacja dziedziczenia dynamicznego (2)

- Wykorzystanie kompozycji





- Wykorzystujemy kod stworzony przy okazji dziedziczenia overlapping.
- Dodatkowo umieszczamy metody ułatwiające „zmianę klasy”.

Implementacja dziedziczenia dynamicznego (3)

- „Sprytne" kopiowanie obiektów
  - Pomysł polega na zastąpieniu starego obiektu, nowym. W tym celu, w każdej z podklas tworzymy dodatkowe konstruktory,
  - Każdy z nich przyjmuje jako parametr referencję do obiektu nadklasy (plus ewentualnie dodatkowe informacje specyficzne dla określonej klasy),
  - Informacje wspólne dla wszystkich obiektów znajdujących się na danym poziomie hierarchii są kopiowane z wnętrza otrzymanego obiektu do wnętrza nowotworzonego obiektu.

Implementacja dziedziczenia dynamicznego (4)

- „Sprytne” kopiowanie obiektów – c. d.
  - Problemem może być uaktualnienie odpowiednich referencji prowadzących do „starego” obiektu tak, aby pokazywały na nowy obiekt.
    - „Odpowiednie” referencje oznaczają te, które są wspólne dla „starej” i „nowej” klasy.
    - Pozostałe referencje (te specyficzne dla „starej” klasy) „przepadają” – podobnie jak wartości atrybutów.
    - W przypadku korzystania z *ObjectPlusPlus*, rozwiązanie tego problemu jest dużo łatwiejsze. Jest tak dlatego, że posiadamy informacje o obiektach, które na „nas” pokazują – bo wszystkie powiązania w *ObjectPlusPlus* są dwustronne!
  - Trzeba również pamiętać o zadbaniu o ekstensję!

Implementacja dziedziczenia dynamicznego (5)

- „Sprytne” kopiowanie obiektów – c. d.

```
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;
   }

   // [...]
}
```

Implementacja dziedziczenia dynamicznego (6)

- „Sprytne” kopiowanie obiektów – c. d.

```
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
```

Implementacja dziedziczenia dynamicznego (7)

- „Sprytne" kopiowanie obiektów – c. d.
  - Trzeba jeszcze zadbać o:
    - zamianę referencji pokazujących na nasz nowy obiekt,
    - usunięcie starego obiektu z ekstensji.
  - W przypadku korzystania z *ObjectPlusPlus* wymagane informacje są już przechowywane w systemie. Trzeba tylko dodać kilka metod, które całą operację zautomatyzują.
  - **Praca domowa** dla chętnych?

Zalety i wady poszczególnych rozwiązań

- Wszystkie rodzaje dziedziczenia, można obejść za pomocą jednej lub kilku poniższych technik:
  - Zastąpienie hierarchii za pomocą jednej klasy
    - Łatwość implementacji. Czasami pozorna, np. trzeba zastąpić przesłanianie metod za pomocą np. *case*'ów lub *if*'ów.
    - Względna łatwość użycia.
    - Nieoptymalne wykorzystanie zasobów.

Zalety i wady poszczególnych rozwiązań (2)

- Wykorzystanie agregacji/kompozycji
  - Optymalne wykorzystanie zasobów,
  - Dość pracochłonna implementacja (m. in. metody „opakowujące”), chociaż niezbyt trudna.
- Zastosowanie interfejsów
  - Dość duża pracochłonność,
  - Można ją zmniejszyć używając agregacji i/oraz propagacji operacji.
  - Duże możliwości.

Podsumowanie

- W popularnych językach programowania występuje tylko najprostszy rodzaj dziedziczenia.
- Wszystkie inne trzeba zaimplementować korzystając z różnych konstrukcji.
- W przeciwieństwie do asocjacji, nie ma jednego idealnego rozwiązania. Każdy przypadek powinien być traktowany indywidualnie.
- Wszystkie omówione sposoby są obejściem dziedziczenia, a nie konstrukcjami równoważnymi.
- W związku z powyższym, tam gdzie się tylko da, należy korzystać z dziedziczenia, a nie jego substytutów.

Pliki źródłowe

- Pobierz pliki źródłowe do wszystkich wykładów MAS

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