Modelowanie Systemów Informacyjnych (MSI)

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

Wykład 10

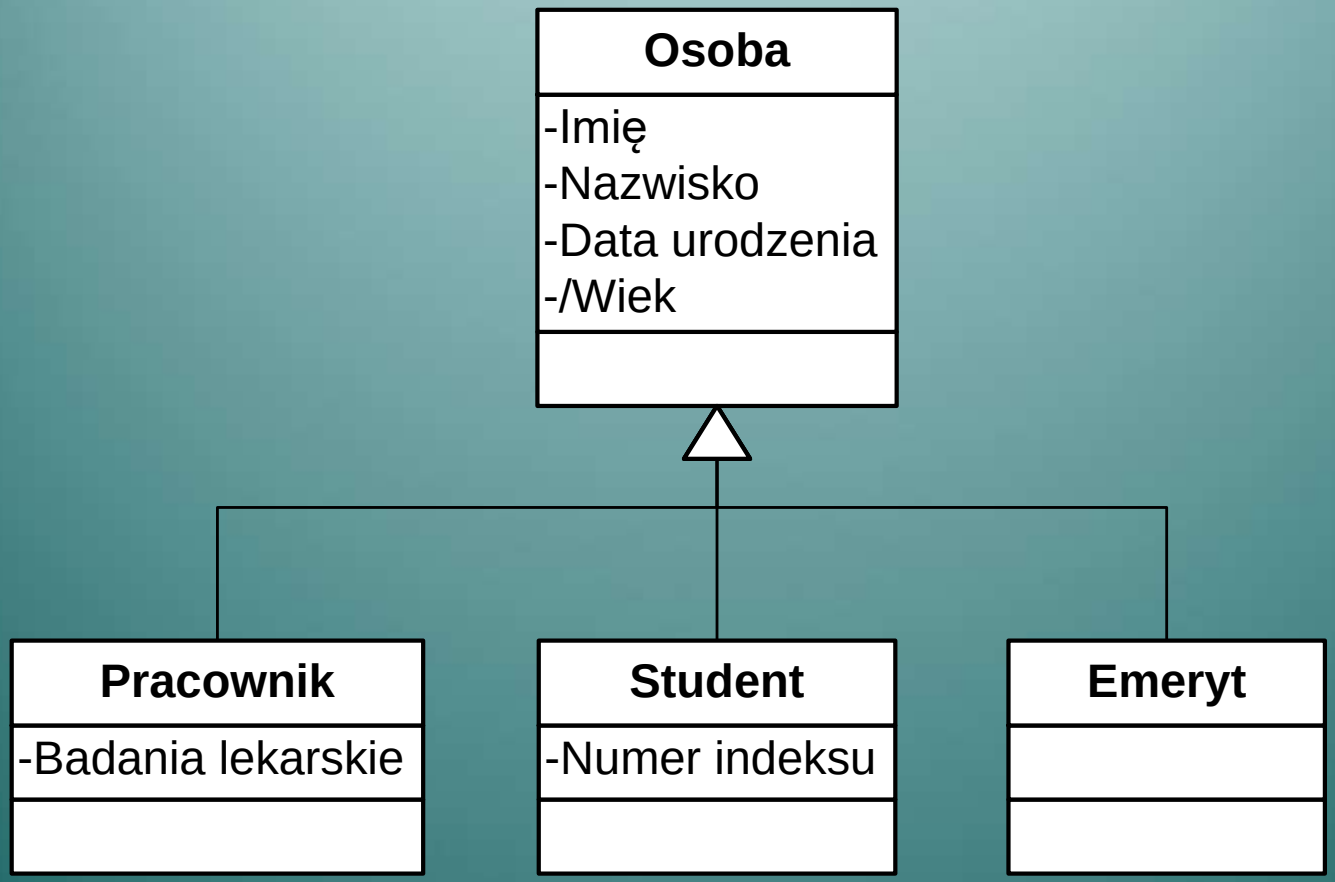
Realizacja różnych modeli dziedziczenia w obiektowych językach programowania

http://www.mtrzaska.com

Zagadnienia

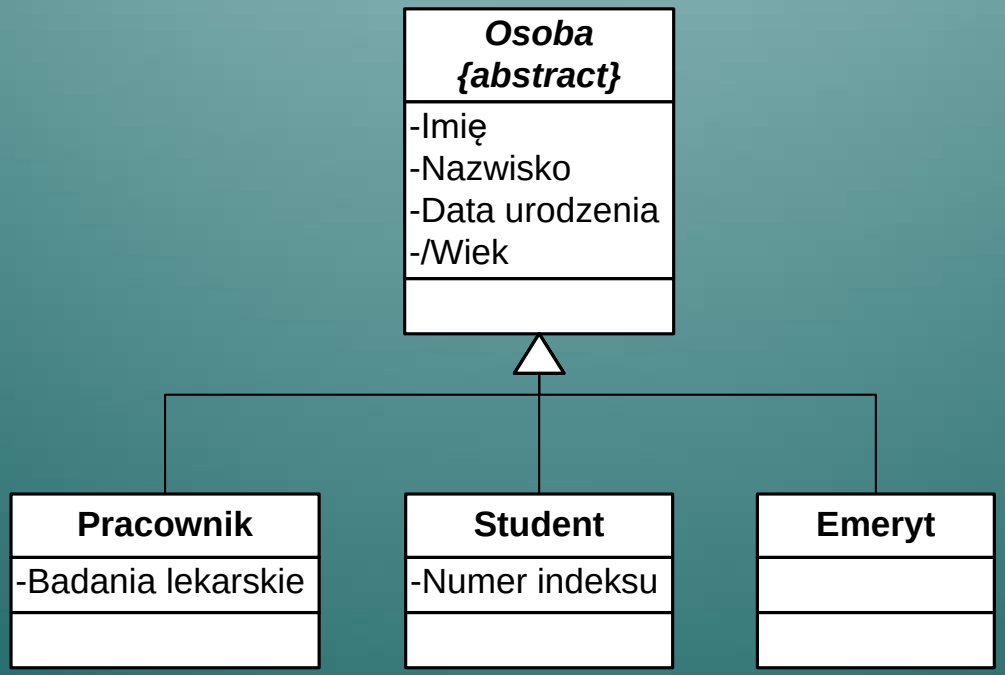
- Omówienie różnych rodzajów dziedziczenia, klas abstrakcyjnych oraz polimorficznego wołania metod.
- Realizacja podstawowego dziedziczenia
- Wykorzystanie klas abstrakcyjnych oraz polimorficznego wołania metod.
- Implementacja pozostałych rodzajów dziedziczenia:
  - *overlapping,*
  - *complete, incomplete,*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

Dziedziczenie *disjoint*



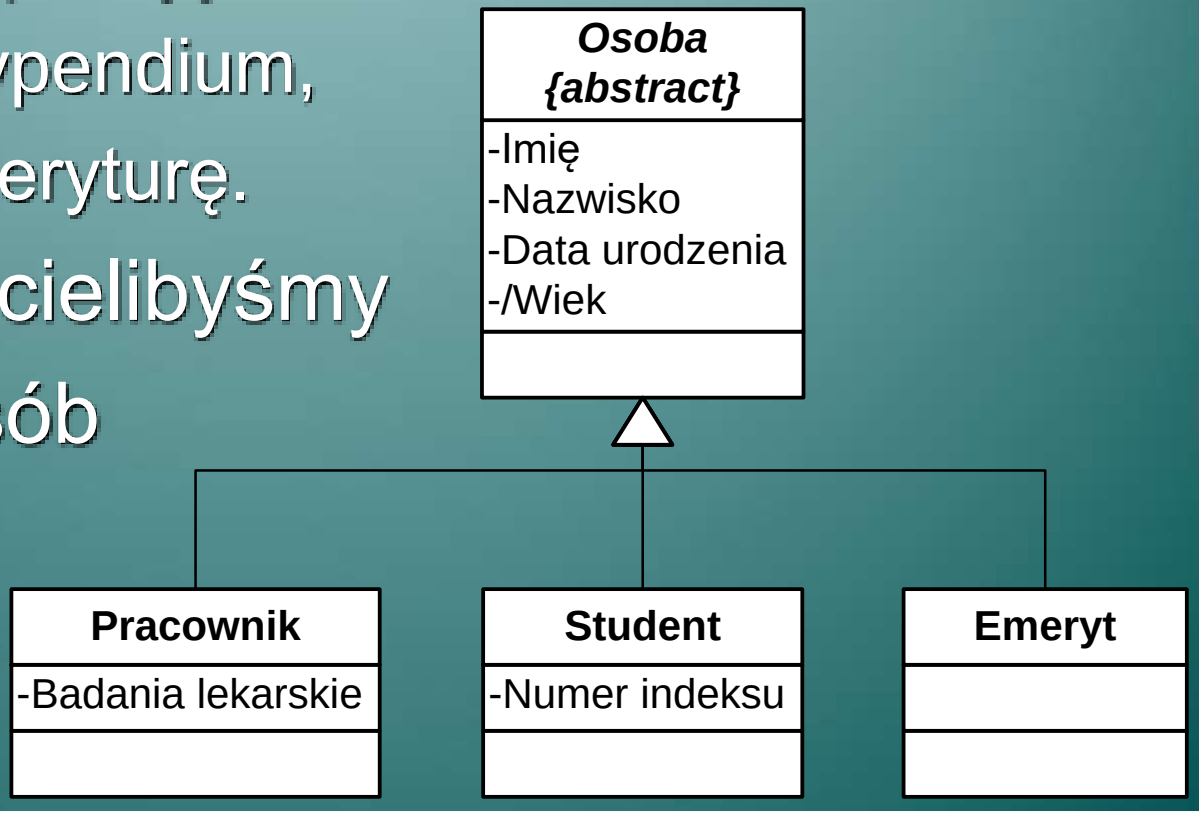
Klasa abstrakcyjna

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



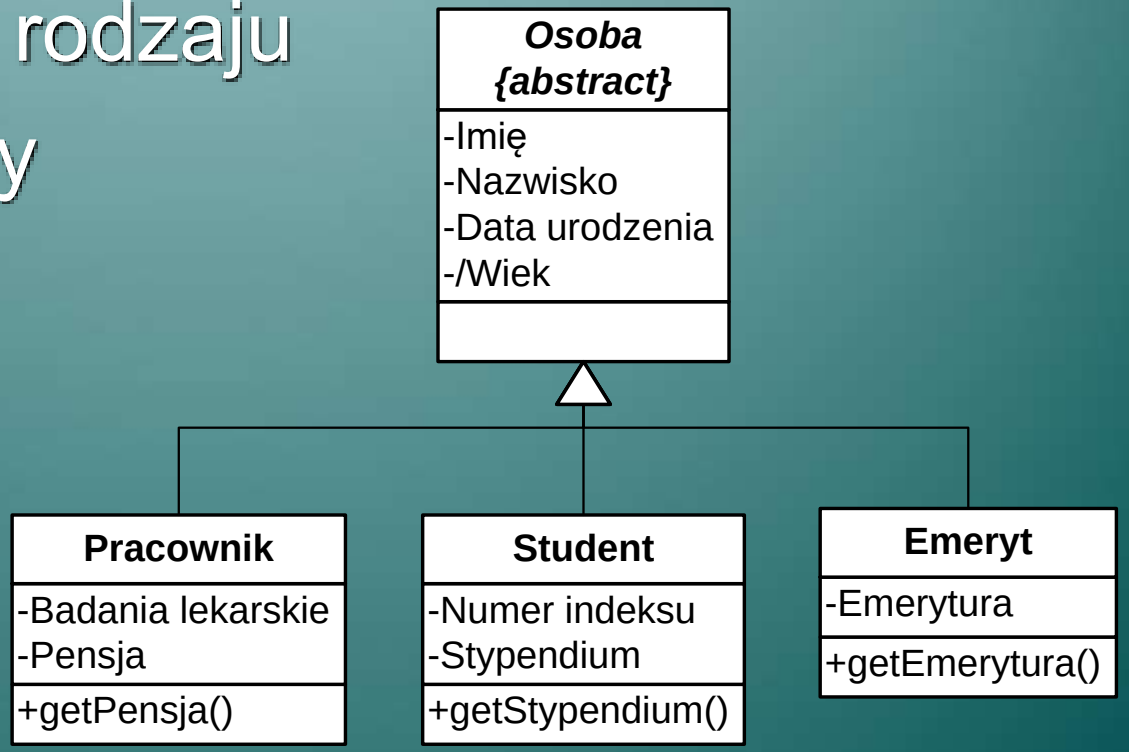
Problem biznesowy

- 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.



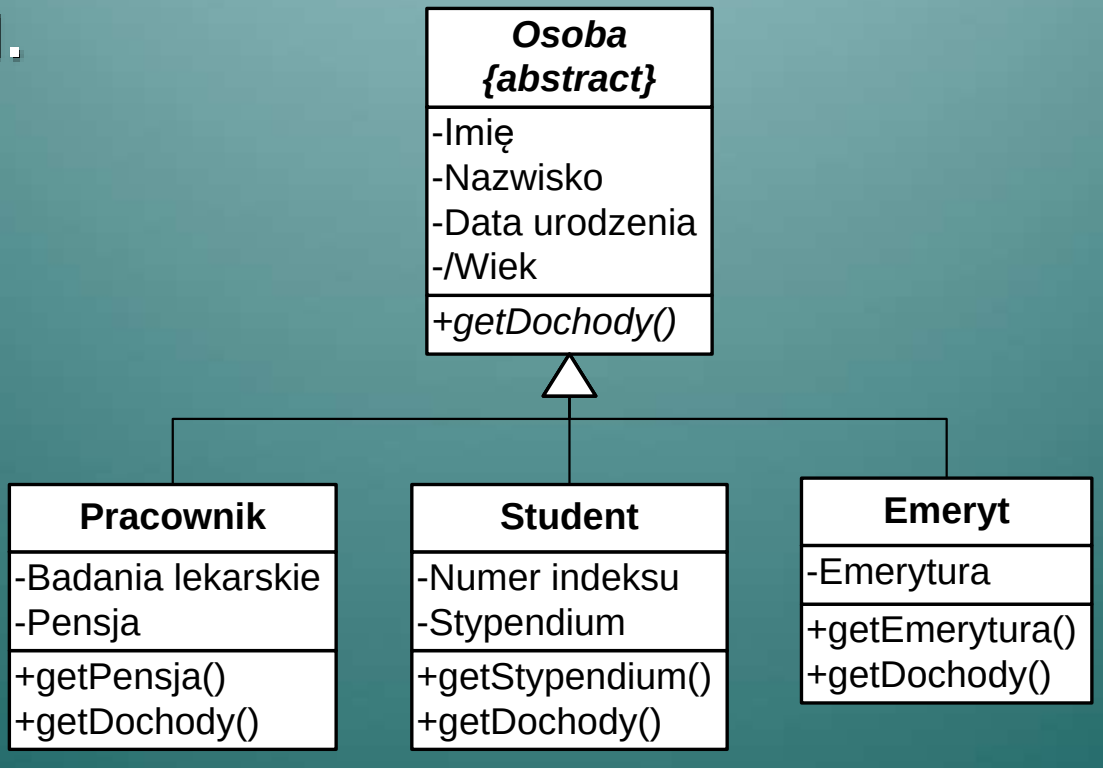
Problem biznesowy (2)

- 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.

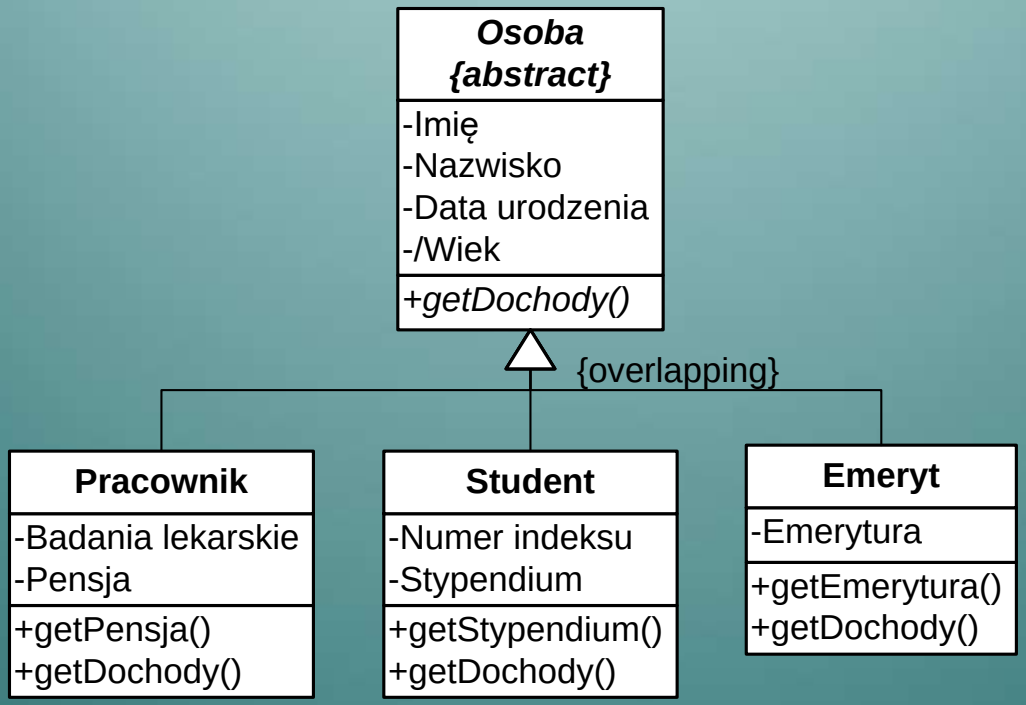


Metody abstrakcyjne

- 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

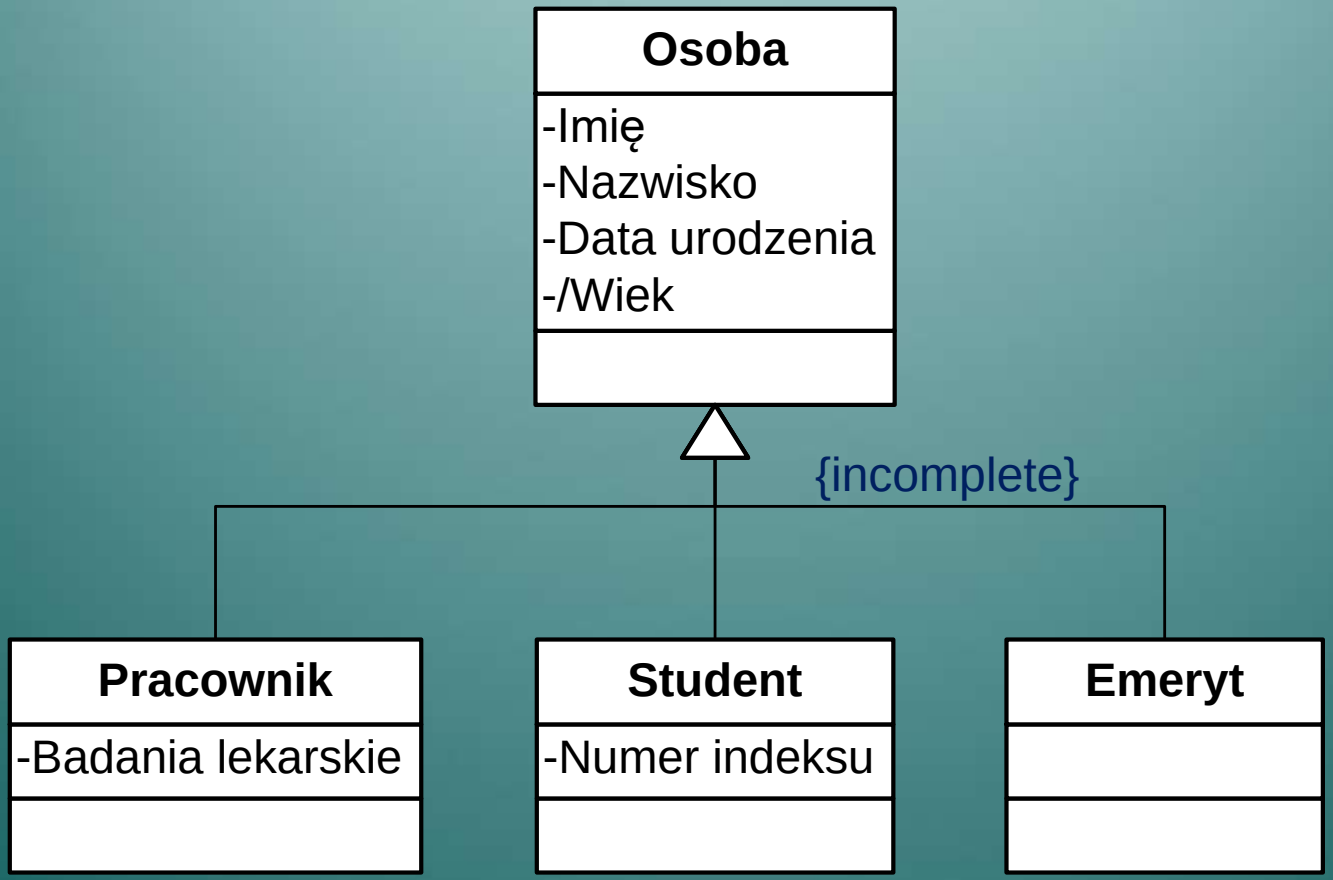
- Dziedziczenie *overlapping*



Co z przesłanianiem i polimorficznym wołaniem metod?

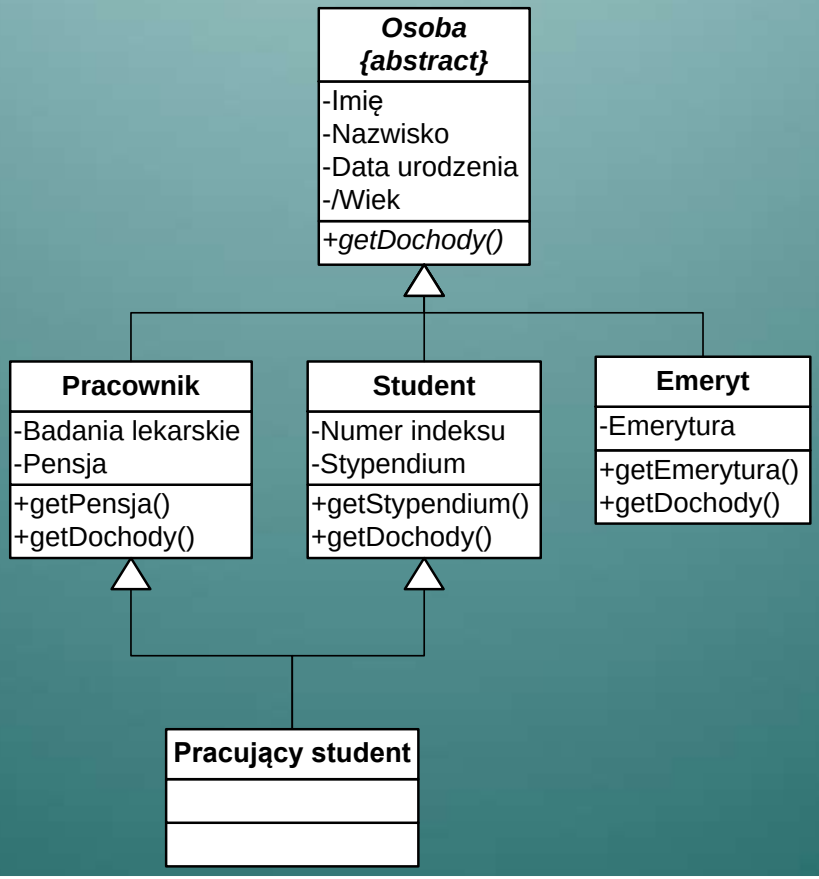
Pozostałe rodzaje dziedziczenia (2)

- Dziedziczenie *incomplete, complete*



Pozostałe rodzaje dziedziczenia (3)

- Wielodziedziczenie (*dziedziczenie wielokrotne, multi-inheritance*)

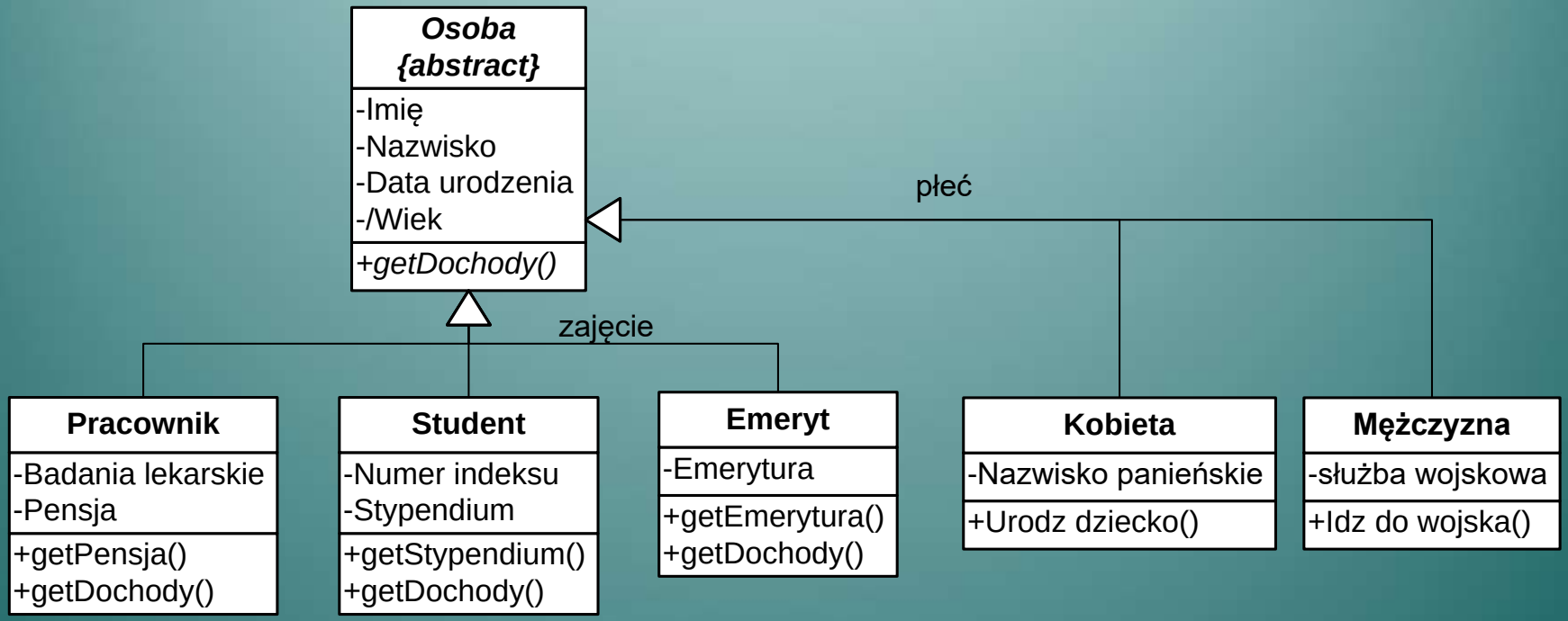


Pozostałe rodzaje dziedziczenia (4)

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

Pozostałe rodzaje dziedziczenia (5)

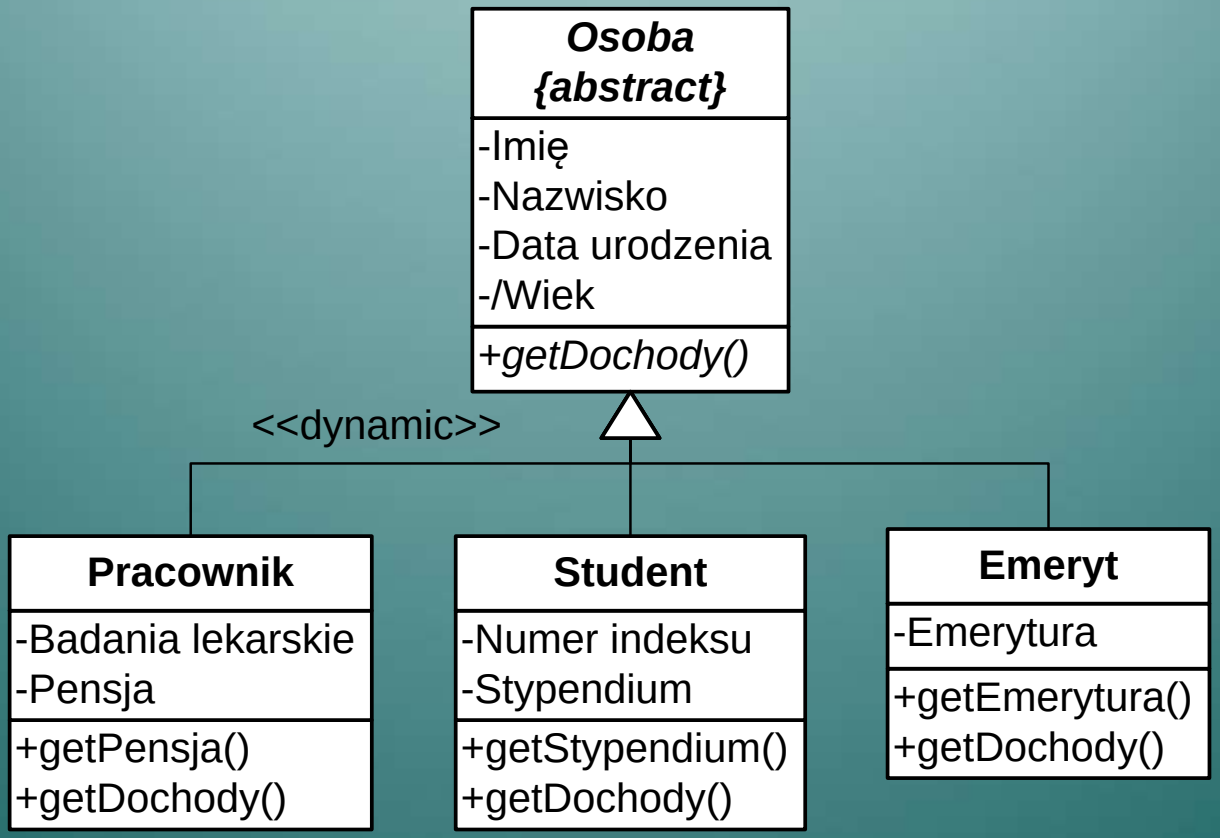
- Dziedziczenie wieloaspektowe (*multi-aspect*)



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:
  - *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 class Osoba {
     private String imie;
     private String nazwisko;
     private Date dataUrodzenia;
}
```

```
public class Pracownik extends Osoba {
     private boolean badaniaLekarskie;
}
```

```
public class Student extends Osoba {
     private int numerIndeksu;
}
```

```
public class Emeryt extends Osoba {

}
```

Wykorzystanie polimorficznego wołania metod

- W językach Java oraz C#:
  - 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 Osoba {
      // [...]
      public Osoba(String imie, String nazwisko, Date dataUrodzenia) {
            super();
            this.imie = imie;
            this.nazwisko = nazwisko;
            this.dataUrodzenia = dataUrodzenia;
      }

      public abstract float getDochody();
}
```

```
public class Pracownik extends Osoba {
      // [...]
      public float getDochody() {
            return getPensja();
      }
      public float getPensja() {
            return pensja;
      }
}
```

- Pozostałe klasy są zaimplementowane analogicznie do powyższych.

Wykorzystanie polimorficznego wołania metod (3)

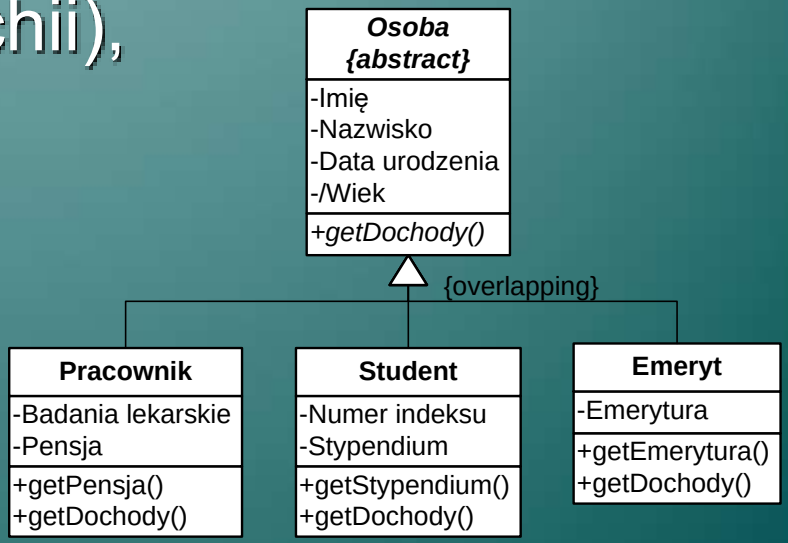
- Tworzymy dwa obiekty:
  - Pracownika,
  - Studenta.
- Traktujemy je po prostu jako osoby (referencja do typu osoba)
- Każdą z nich pytamy o dochody (bez sprawdzania z jaką klasą mamy do czynienia).
- Dzięki polimorficznemu wołaniu metody, dostajemy odpowiedzi właściwe dla poszczególnych rodzajów osób.

```
Osoba o1 = new Pracownik("Jan", "Kowalski", new Date(), true, 4000.0f);
Osoba o2 = new Student("Adam", "Abacki", new Date(), 1212, 2000.0f);

System.out.println(o1.getDochody());
System.out.println(o2.getDochody());
```

```
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).

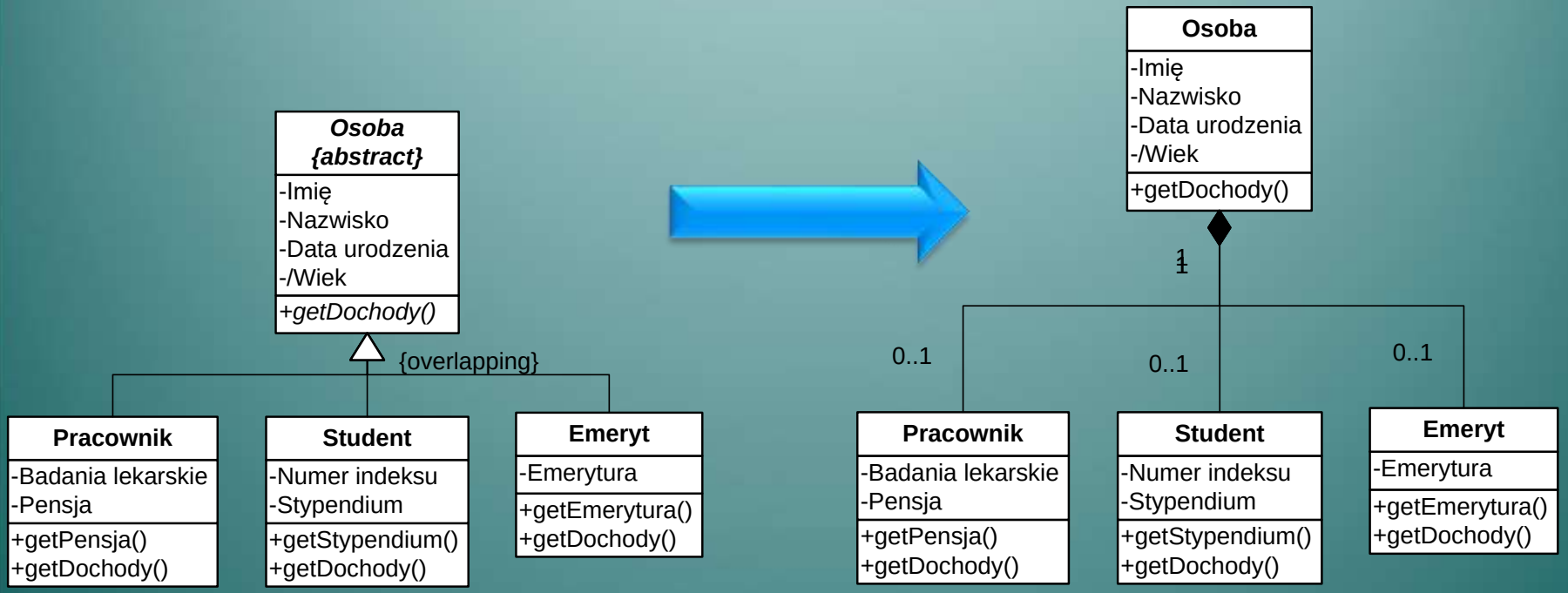
```
enum OsobaRodzaj {Osoba, Pracownik, Student, Emeryt};

public class Osoba {
     private String imie;
     private String nazwisko;
     private Date dataUrodzenia;

     private boolean badaniaLekarskie;

     private int numerIndeksu;

     // Musimy uzyc EnumSet zamiast rodzajOsoby poniewaz chcemy miec
     // mozliwosc przechowywania kombinacji osob, np. Pracownik + Student
     private EnumSet<OsobaRodzaj> rodzajOsoby =  EnumSet.<OsobaRodzaj>of(OsobaRodzaj.Osoba);
}
```

Realizacja dziedziczenia *overlapping* (3)

- 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* (4)

- Wykorzystanie agregacji lub kompozycji



Realizacja dziedziczenia *overlapping* (5)

- 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* (6)

- 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* (7)

```
public class Osoba extends ObjectPlusPlus {
     private String imie;
     private String nazwisko;
     private Date dataUrodzenia;

     public Osoba(String imie, String nazwisko, Date dataUrodzenia) {
          super();// Wymagane przez ObjectPlusPlus

          this.imie = imie;
          this.nazwisko = nazwisko;
          this.dataUrodzenia = dataUrodzenia;
     }

     public Osoba(String imie, String nazwisko, Date dataUrodzenia, boolean badaniaLekarskie)
     {
          super();// Wymagane przez ObjectPlusPlus

          this.imie = imie;
          this.nazwisko = nazwisko;
          this.dataUrodzenia = dataUrodzenia;

          // "Zmienia" osobe w pracownika
          dodajPracownika(badaniaLekarskie);
     }

     // […]
```

Realizacja dziedziczenia *overlapping* (8)

```
public class Osoba extends ObjectPlusPlus {

    // […]

    public void dodajPracownika(boolean badaniaLekarskie) {
        // Tworzymy czesc opisujaca Pracownika
        Pracownik p = new Pracownik(badaniaLekarskie);

        // Dodanie pracownika jako powiazania
        // (nie korzystamy z dodawania jako czesci w agregacji aby uniknac wyjatku)
        // Korzystamy z metody dostarczanej przez ObjectPlusPlus
        this.dodajPowiazanie(nazwaRoliPracownik, "generalizacja", p);
    }

    public void dodajEmeryta() throws Exception {
        // Tworzymy czesc opisujaca Pracownika
        Emeryt e = new Emeryt();

        // Dodanie emeryta jako powiazania
        // (nie korzystamy z dodawania jako czesci w agregacji aby uniknac wyjatku)
        // Korzystamy z metody dostarczanej przez ObjectPlusPlus
        this.dodajPowiazanie(nazwaRoliEmeryt, "generalizacja", e);
    }

    private static String nazwaRoliPracownik = "specjalizacjaPracownik";
    private static String nazwaRoliEmeryt = "specjalizacjaEmeryt";

    // […]
```

Realizacja dziedziczenia *overlapping* (9)

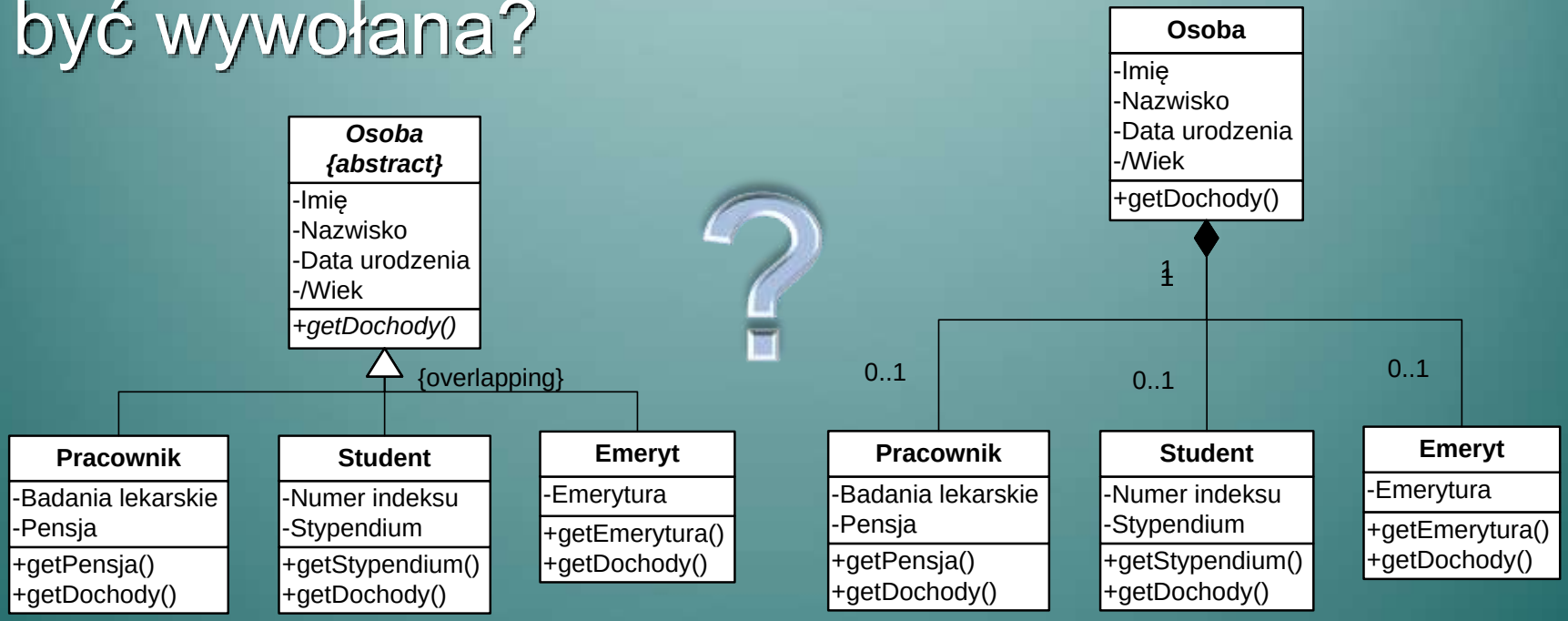
```
public class Osoba extends ObjectPlusPlus {
      // […]

      public boolean czyMaBadaniaLekarskie() throws Exception {
            // daj obiekt opisujacy pracownika
            try {
                  ObjectPlusPlus[] obj = this.dajPowiazania(nazwaRoliPracownik);
                  return ((Pracownik) obj[0]).isBadaniaLekarskie();
            } catch (Exception e) {
                  // Prawdopodobnie dostalismy wyjatek mowiacy, ze taka rola nie istnieje
                  // (docelowo powinny to byc rozne klasy wyjatkow)
                  throw new Exception("Obiekt nie jest Pracownikiem!");
            }
      }

      public int dajNumerIndeksu() throws Exception {
            // daj obiekt opisujacy pracownika
            try {
                  ObjectPlusPlus[] obj = this.dajPowiazania(nazwaRoliStudent);
                  return ((Student) obj[0]).getNumerIndeksu();
            } catch (Exception e) {
                  // Prawdopodobnie dostalismy wyjatek mowiacy, ze taka rola nie istnieje
                  // (docelowo powinny to byc rozne klasy wyjatkow)
                  throw new Exception("Obiekt nie jest Studentem!");
            }
      }
}
```

Realizacja dziedziczenia *overlapping* (10)

- 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(!)

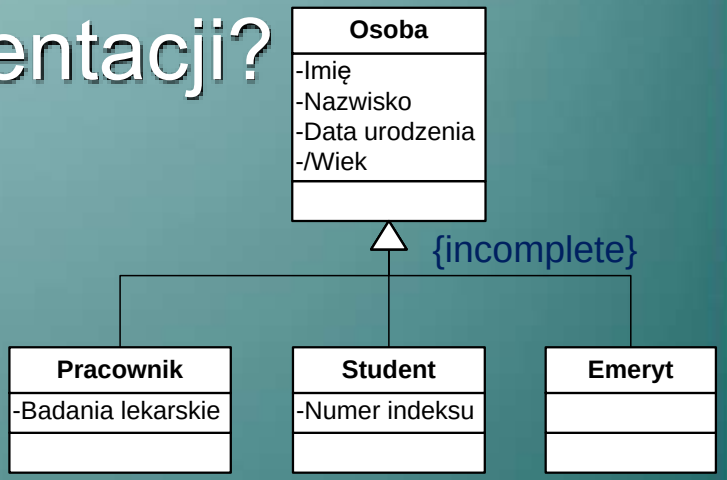
```
public float getDochody() throws Exception {
     float dochody = 0.0f;

     if(this.czySaPowiazania(nazwaRoliPracownik)) {
          // Jest pracownikiem. Znajdz obiekt opisujacy pracownika.
          ObjectPlusPlus[] obj = this.dajPowiazania(nazwaRoliPracownik);
          // ==> dolicz dochody pracownika
          dochody += ((Pracownik) obj[0]).getDochody();
     }

     if(this.czySaPowiazania(nazwaRoliStudent)) {
          // Jest studentem. Znajdz obiekt opisujacy studenta.
          ObjectPlusPlus[] obj = this.dajPowiazania(nazwaRoliStudent);
          // ==> dolicz dochody studenta
          dochody += ((Student) obj[0]).getDochody();
     }

     if(this.czySaPowiazania(nazwaRoliEmeryt)) {
          // Jest emerytem. Znajdz obiekt opisujacy emeryta.
          // [...]
     }
     return dochody;
}
```

Dziedziczenie *complete* oraz *incomplete*

- Co te rodzaje dziedziczenia znaczą dla diagramu?
- Czy coś znaczą dla implementacji?
- **Nie**!
- W związku z tym,

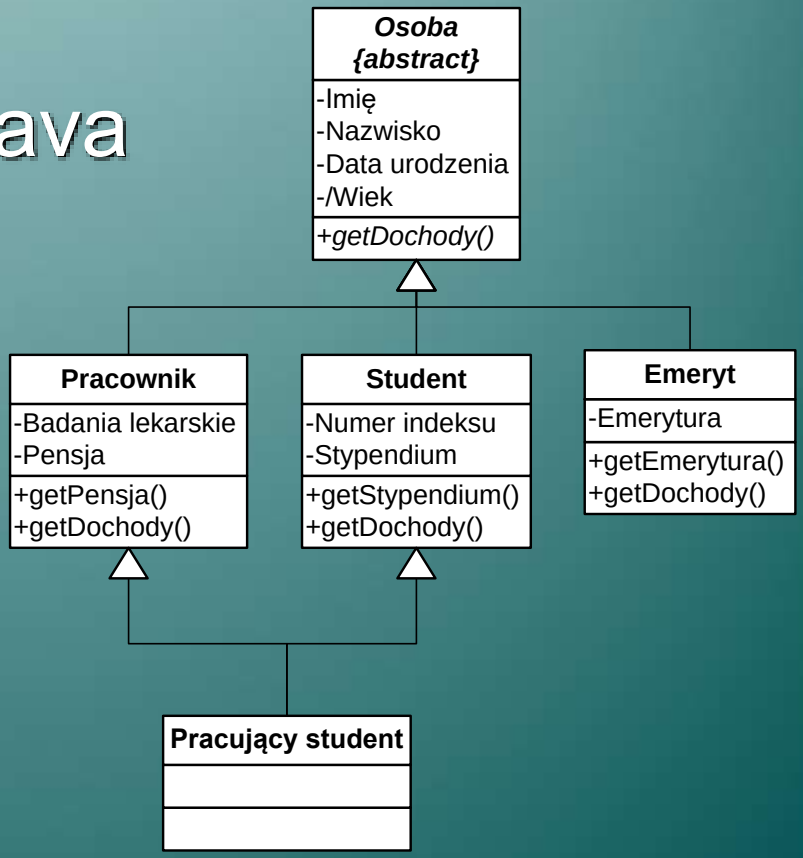
  ignorujemy te oznaczenia.

  Ewentualnie bierzemy je pod uwagę wybierając konkretny rodzaj implementacji.



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?



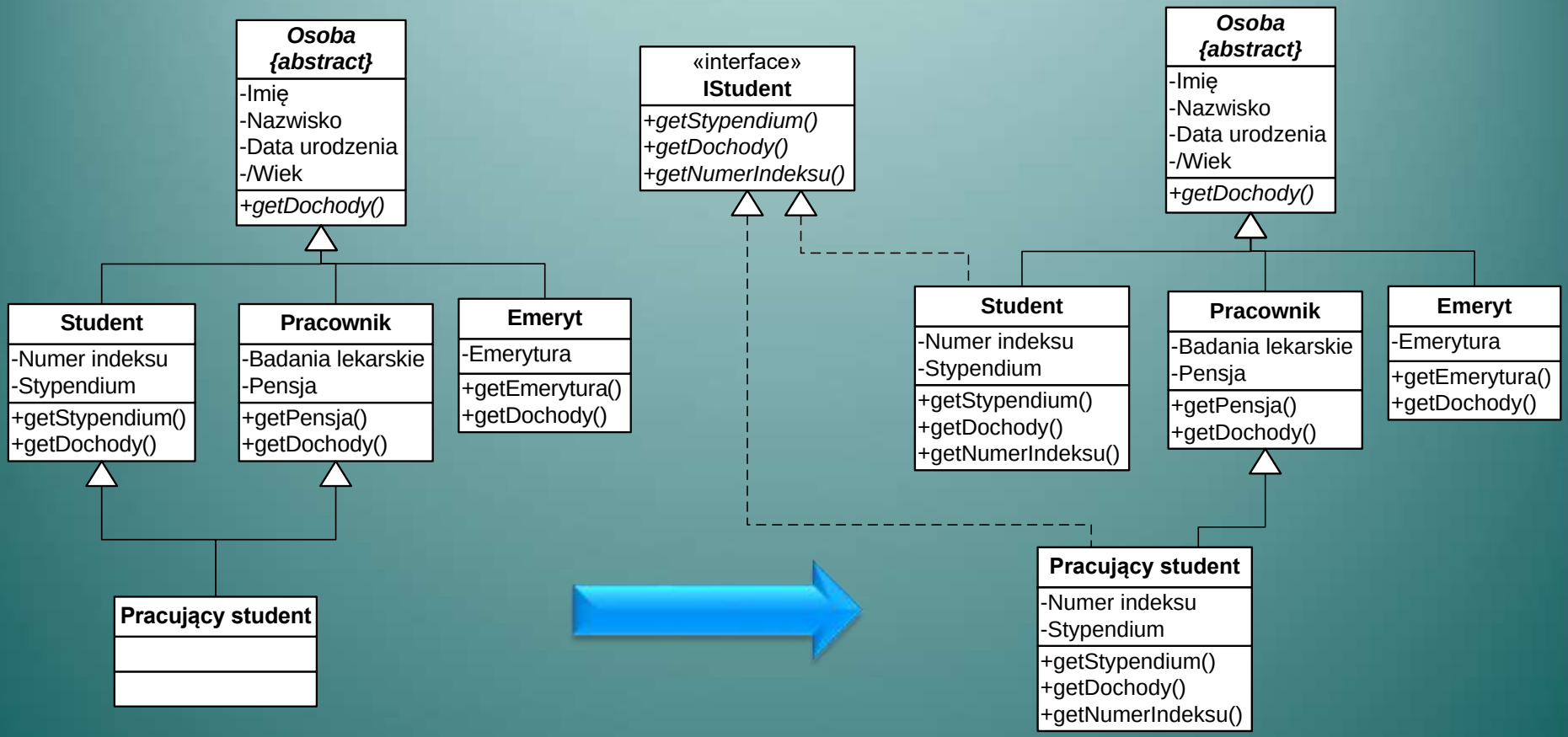
Implementacja wielodziedziczenia (2)

- Implementujemy je korzystając ze sposobów podanych przy okazji dziedziczenia *overlapping*:
  - Jedna klasa,
  - 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.

Implementacja wielodziedziczenia z wykorzystaniem interfejsów (2)



Implementacja wielodziedziczenia z wykorzystaniem interfejsów (3)

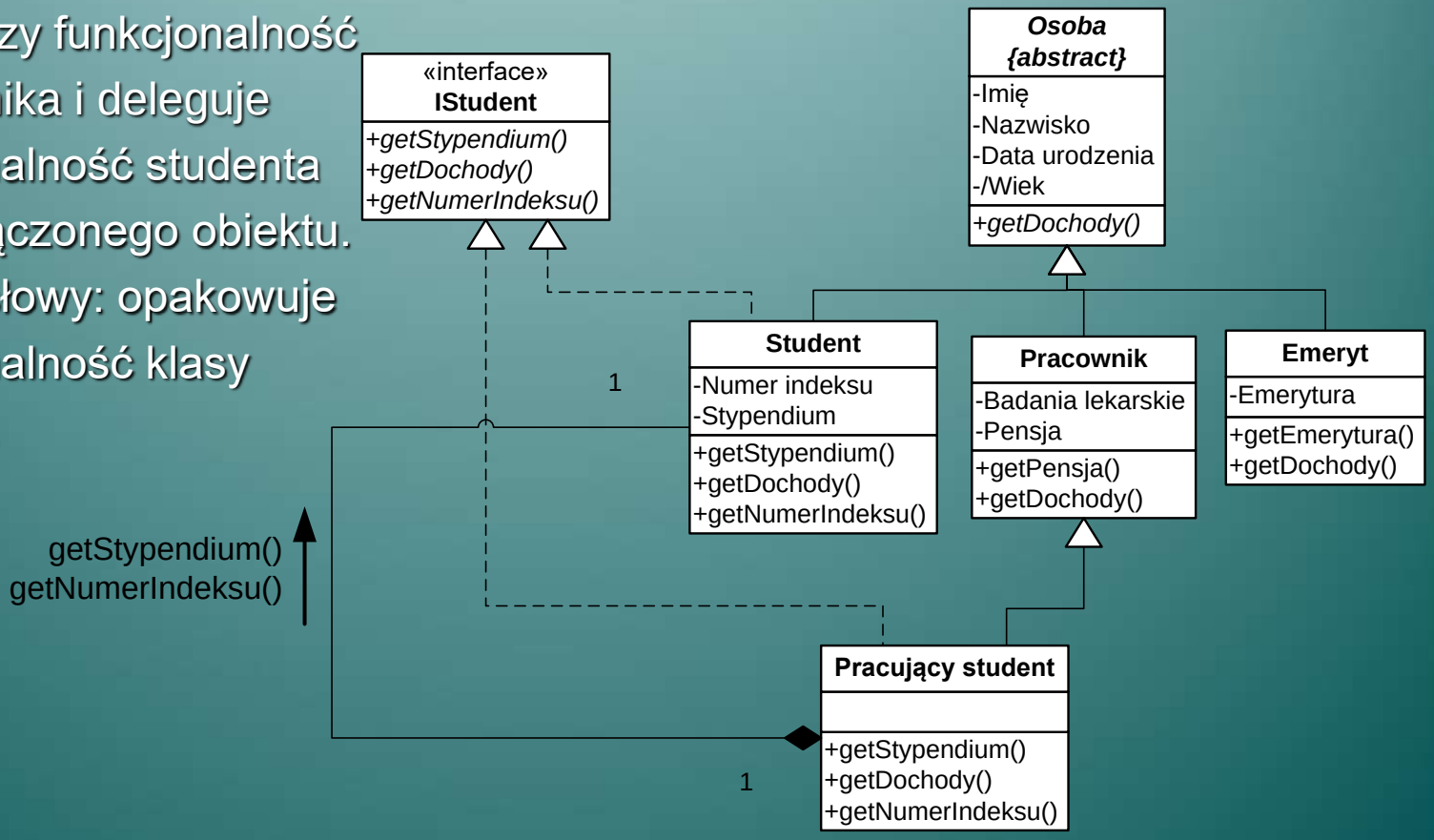
```
public interface IStudent {
	public abstract float getDochody();
	public abstract float getStypendium();
	public abstract void setStypendium(float stypendium);
	public abstract int getNumerIndeksu();
}
```

```
public class PracujacyStudent extends Pracownik implements IStudent {
	private int numerIndeksu;
	private float stypendium;

	public PracujacyStudent(String imie, String nazwisko, Date dataUrodzenia, boolean
	badaniaLekarskie, float pensja, int numerIndeksu, float stypendium) {
		super(imie, nazwisko, dataUrodzenia, badaniaLekarskie, pensja);

		this.numerIndeksu = numerIndeksu;
		this.stypendium = stypendium;
	}
	public float getStypendium() {
		return stypendium;
	}
	public float getDochody() {
		return super.getDochody() + getStypendium();
	}
	public int getNumerIndeksu() {
		return numerIndeksu;
	}
}
```

Jak widać musieliśmy pewne metody (np. `getStypendium()`) 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*.



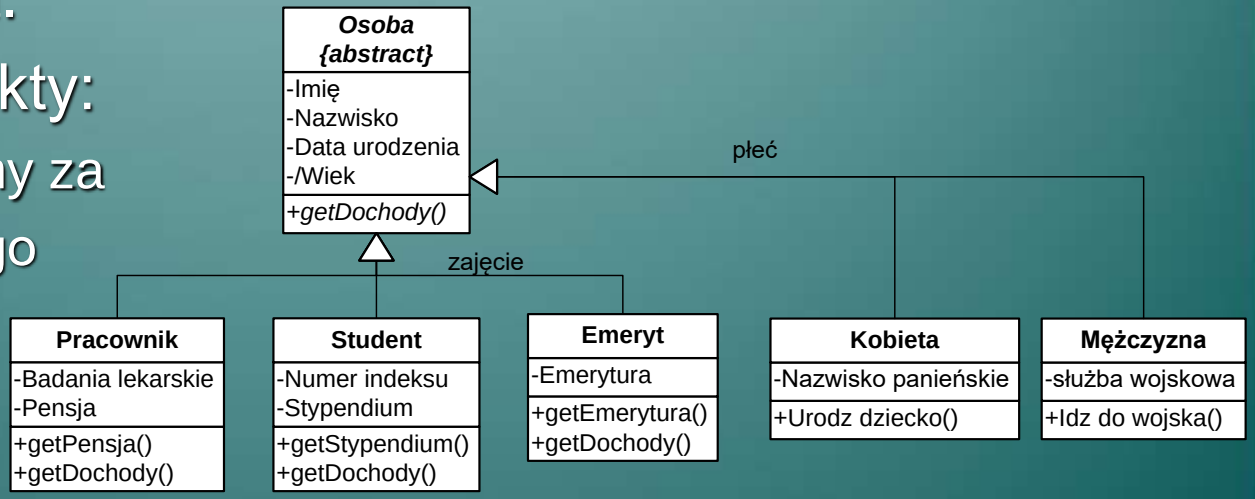
Implementacja wielodziedziczenia z wykorzystaniem interfejsów (5)

```
public class PracujacyStudent extends Pracownik implements IStudent {
      Student student;
      public PracujacyStudent(String imie, String nazwisko, Date dataUrodzenia, boolean
      badaniaLekarskie, float pensja, int numerIndeksu, float stypendium) {
            super(imie, nazwisko, dataUrodzenia, badaniaLekarskie, pensja);
            student = new Student(imie, nazwisko, dataUrodzenia, numerIndeksu, stypendium);
      }
      public float getStypendium() {
            return student.getStypendium();
      }
      public void setStypendium(float stypendium) {
            student.setStypendium(stypendium);
      }
      public float getDochody() {
            return super.getDochody() + getStypendium();
      }
      public int getNumerIndeksu() {
            return student.getNumerIndeksu();
      }
}
```

- 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`.

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)

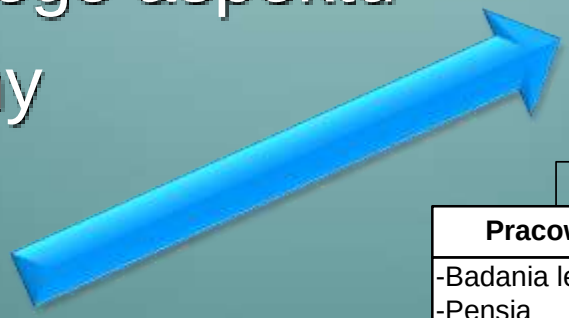
- Który aspekt powinniśmy dziedziczyć?
  - Tam gdzie występuje przesłanianie metod, polimorficzne wołanie,
  - Tam gdzie jest większe zróżnicowanie atrybutów w poszczególnych podklasach.
  - Innymi słowy – najbardziej skomplikowaną/rozbudowaną hierarchię.

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.

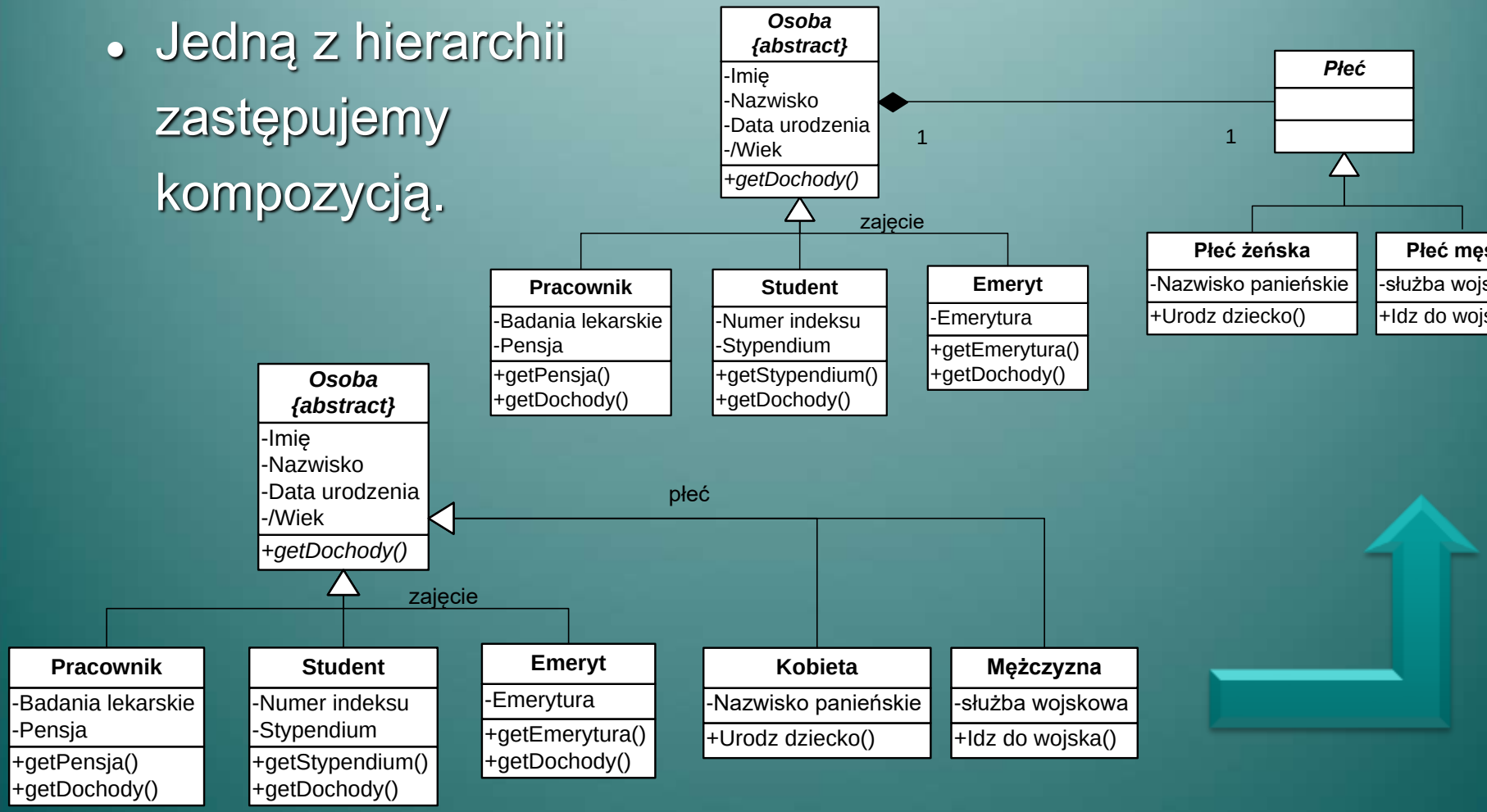
Implementacja dziedziczenia wieloaspektowego (4)

- Przykładowe rozwiązanie nr 1
  - Atrybuty i metody ze zlikwidowanego aspektu umieszczamy w nadklasie.



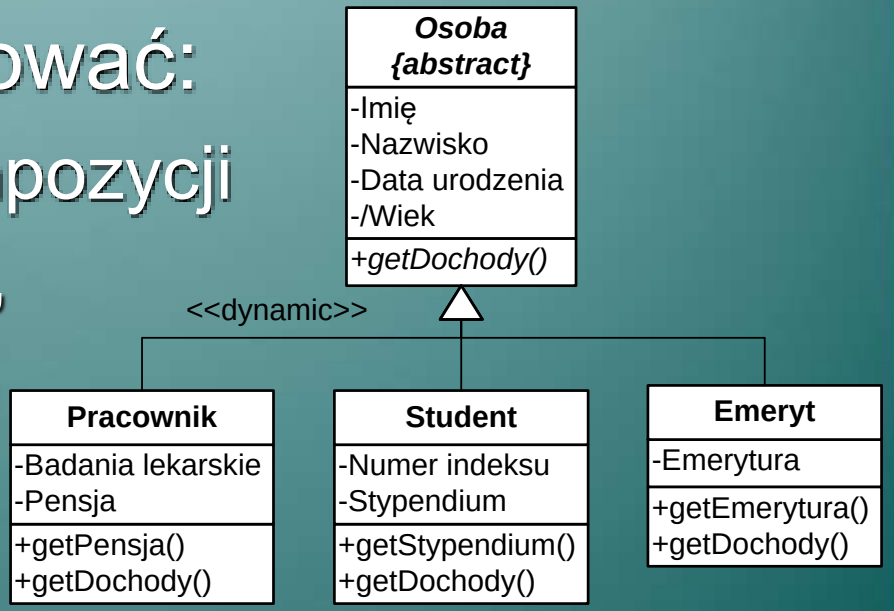
Implementacja dziedziczenia wieloaspektowego (5)

- Przykładowe rozwiązanie nr 2
  - Jedną z hierarchii zastępujemy kompozycją.

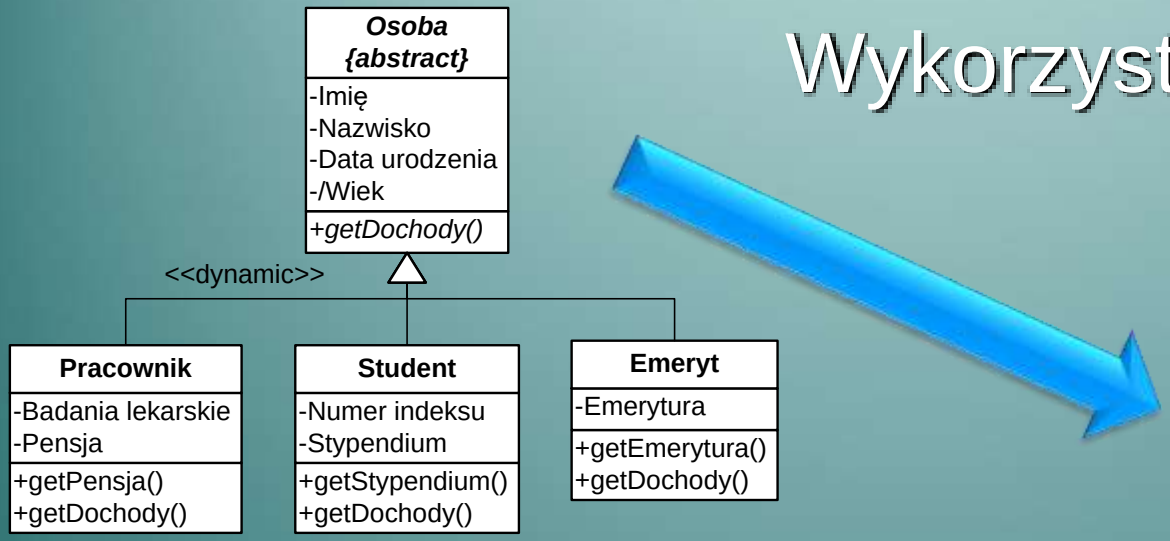


Implementacja dziedziczenia dynamicznego

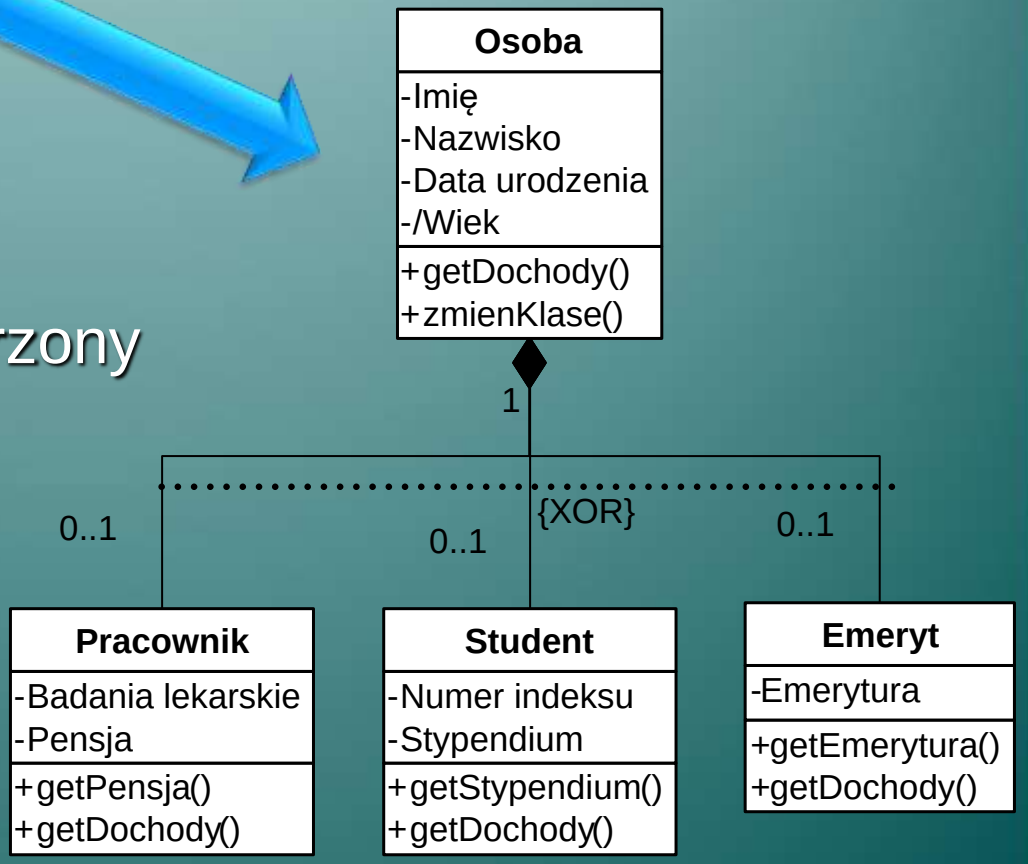
- Nie występuje bezpośrednio w żadnym popularnym języku programowania (Java, C#, C++).
- Trzeba je zaimplementować:
  - Używając agregacji/kompozycji z ograniczeniem `{xor}`,
  - Umieszczając wszystkie inwarianty w nadklasie i dodając dyskryminator,
  - „Sprytnie” kopiując obiekty.



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 Osoba {
     protected String imie;
     protected String nazwisko;
     protected Date dataUrodzenia;

     // […]
     public String toString() {
          return this.getClass().getSimpleName() + ": " + imie + " " + nazwisko;
     }
}
```

```
public class Pracownik extends Osoba {
     private boolean badaniaLekarskie;
     private float pensja;
     // […]

     public Pracownik(Osoba poprzedniaOsoba, boolean badaniaLekarskie, float pensja) {
          // Skopiowanie "starych" danych
          super(poprzedniaOsoba.imie, poprzedniaOsoba.nazwisko,
                    poprzedniaOsoba.dataUrodzenia);

          // Zapamietanie nowych
          this.badaniaLekarskie = badaniaLekarskie;
          this.pensja = pensja;
     }
```

Implementacja dziedziczenia dynamicznego (6)

- „Sprytne” kopiowanie obiektów – c. d.

```
// Tworzymy studenta
Osoba o1  = new Student("Jan", "Kowalski", new Date(), 1212, 2000.0f);
System.out.println(o1);

// Tworzymy pracownika na podstawie studenta
o1 = new Pracownik(o1, true, 4000.0f);
System.out.println(o1);

// Tworzymy emeryta na podstawie pracownika
o1 = new Emeryt(o1, 3000.0f);
System.out.println(o1);
```

```
Student: Jan Kowalski
Pracownik: Jan Kowalski
Emeryt: Jan Kowalski
```

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 (spłaszczenie hierarchii, ang. *flattening*)
  - 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.**