Język JAVA
podstawy programowania

Jacek Rumiński

Wykład 4, część 1

Plan wykładu:

1. Podstawy modelowania obiektowego
2. Konstruktory
3. Dziedziczenie, związki pomiędzy klasami, UML
4. Polimorfizm
5. Klasy abstrakcyjne i interfejsy
6. Adaptery, klasy anonimowe
7. Tablice, kolekcje i typ wyliczeniowy

Klasy i obiekty

Klasa – typ obiektów,
Obiekt – instancja, wystąpienie, realizacja klasy.

Jak stworzyć klasę?

Obserwujemy rzeczywiste obiekty i zbieramy informacje o ich cechach oraz zachowaniu. Jeśli współdzielą grupę cech i funkcji to jest to ich wspólny typ (kategoria, forma, itd.).
Ilość cech i rodzaj funkcji jakie zdefiniujemy jest taka, jaka jest potrzebna ze względu na cel tworzenia oprogramowania.
Przykładowo – liczba włosów na głowie jest cechą klasy Człowiek, ale czy w praktyce jest to po coś potrzebne zbierając informacje o osobach?

Tworzenie obiektów

Utworzenie obiektu polega na wywołaniu:
-jednego z konstruktorów danej klasy, albo
-jednej z specjalnie zaprojektowanych funkcji statycznych (jeśli istnieje).

Wywołanie konstruktora:

```
Rycerz luke = new Rycerz(); //m. w. wcześniej ładuje kod klasy
```

oznacza, że tworzony jest nowy obiekt typu Rycerz, do którego przywiązana jest referencja "luke". Tworzenie obiektu danej klasy bardzo dobrze ilustruje kod obsługujący jawne ładowanie klas:

```
Class c = Class.forName("Rycerz");
Rycerz luke = c.newInstance(); //konstruktor bez argumentów
```

Plan wykładu:

1. Podstawy modelowania obiektowego
2. Konstruktory
3. Dziedziczenie, związki pomiędzy klasami, UML
4. Polimorfizm
5. Klasy abstrakcyjne i interfejsy
6. Adaptery, klasy anonimowe
7. Tablice, kolekcje i typ wyliczeniowy

Konstruktor klasy

- Konstruktor klasy to specjalna metoda, która zwraca referencję do obiektu danej klasy. Oznacza to, że nie deklaruje się typu danych wartości zwracanej.
- Nazwa konstruktora musi być taka sama jak nazwa klasy (zgodność wielkości liter !!!).
- Dla jednej klasy można zdefiniować wiele konstruktorów, każdy tak samo się nazywa, lecz musi mieć różną liczbę argumentów lub różne typy argumentów.

Nazwa metody wraz z liczą argumentów i typami danych argumentów określana jest sygnaturą metody (unikalność!). Listę sygnatur metod dla skompilowanej klasy (plik cos.class) można uzyskać wykonując w linii komend (terminalu) polecenie

c:\> javap –s cos

Konstruktor klasy

Konstruktor klasy może wywołać konstruktor klasy nadrzędnej (w przypadku gdy dana klasa dziedziczy po innej) oraz może wywołać inny konstruktor tej samej klasy.
Kolejność wołania konstruktorów w kodzie danego konstruktora jest następująca:

```
NazwaKlasy(argumety){
  this(argumenty1); //wywołanie innego konstruktora tej samej klasy
  super(argumenty1); //wywołanie konstruktora klasy bazowej
  kod;
}
```

Kod programu: KonstruktoryJedi.java

```
public class KonstruktoryJedi{

        int typ;
        /**wykonaj konstruktor bez argumentów, domyślne wartości*/
        public KonstruktoryJedi(){
                this(1); //wykonaj konstruktor z 1 argumentem
        }
        /**wykonaj konstruktor z 1 argumentem*/
        public KonstruktoryJedi(int typ){
                this.typ=typ;
        }
        //Wykonać raz z jednym, raz z drugim konstruktorem
        public static void main(String a[]){
                KonstruktoryJedi kj = new KonstruktoryJedi();
                //KonstruktoryJedi kj = new KonstruktoryJedi(50);
                System.out.println("Numer typu Jedi to: "+kj.typ);
        }//koniec main()

}//koniec public class KonstruktoryJedi
```

Kod programu: SuperKonstruktoryJedi.java

```
class WzorzecJedi{
        int typ;
        public WzorzecJedi(){       this(1);       }
        public WzorzecJedi(int typ){   this.typ=typ;          }
}
public class SuperKonstruktoryJedi extends WzorzecJedi{
        public SuperKonstruktoryJedi(){
                //domyślnie wykonaj super() -> WzorzecJedi()
        }
        public SuperKonstruktoryJedi(int typ){
                super(typ); //wykonaj -> WzorzecJedi(typ)
        }
        public static void main(String a[]){
                SuperKonstruktoryJedi kj = new SuperKonstruktoryJedi();
                //SuperKonstruktoryJedi kj = new SuperKonstruktoryJedi(50);
                System.out.println("Numer typu Jedi to: "+kj.typ);
        }//koniec main()
}//koniec public class SuperKonstruktoryJedi
```

Konstruktor klasy

Jeśli nie zdefiniujemy jawnie ŻADNEGO konstruktora, automatycznie tworzony jest domyślny, pusty (bez argumentów, bez instrukcji) konstruktor:

```
NazwaKlasy(){
}
```

Jeśli jednak napisany został jakikolwiek konstruktor – konstruktor domyślny nie będzie dostępny.
Zawsze możemy sprawdzić jakie są konstruktory w dokumentacji klasy.

Po utworzeniu obiektu, mając referencję do obiektu możemy sprawdzić jakiej klasy jest dany obiekt (operator instanceof):

```
luke instanceof Rycerz
```

Plan wykładu:

1. Podstawy modelowania obiektowego
2. Konstruktory
3. Dziedziczenie, związki pomiędzy klasami, UML
4. Polimorfizm
5. Klasy abstrakcyjne i interfejsy
6. Adaptery, klasy anonimowe
7. Tablice, kolekcje i typ wyliczeniowy

Ponownie o dziedziczeniu

Dziedziczenie - proces ewolucyjny, w którym potomkowie posiadają pewne cechy rodziców.

Dziedziczenie w Javie – klasa dziedzicząca po innej klasie przejmuje jej wszystkie cechy i metody (z pewnymi wyjątkami).

W Javie określenie dziedziczenia odbywa się poprzez użycie słowa kluczowego extends (rozszerza). Przykładowa deklaracja:

```
class Kobieta extends Rycerz{
(...)
}
```

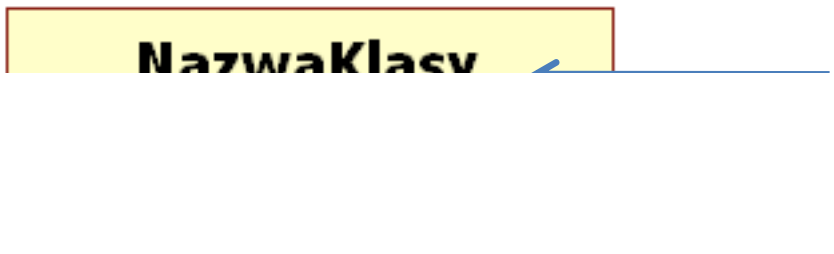
Projektując zestaw klas warto przemyśleć problem zależności pomiędzy klasami. Utworzenie klasy bazowej w jak najbardziej uniwersalny sposób umożliwia dziedziczenie po niej, czyli tworzenia bardziej szczegółowych klas dla danych zastosowań (REUSE !).

Ponownie o dziedziczeniu

W Javie możliwe jest tylko dziedziczenie typu **jeden-do-jednego**, co oznacza, że klasa może dziedziczyć tylko po jednej klasie nadrzędnej. Ponieważ klasa nadrzędna może również dziedziczyć po jednej klasie dla niej nadrzędnej otrzymuje się specyficzne drzewo dziedziczenia w Javie.
Nadrzędną klasą dla wszystkich klas w Javie jest klasa **Object**.

Projektowanie klas i związków pomiędzy nimi (dziedziczenie i inne) ułatwia specjalna notacja w formie diagramów wprowadzona przez Ujednolicony Język Modelowania (UML – Unified Modelling Language, www.omg.org).

Wprowadźmy kilka podstawowych zasad budowania diagramów klas.

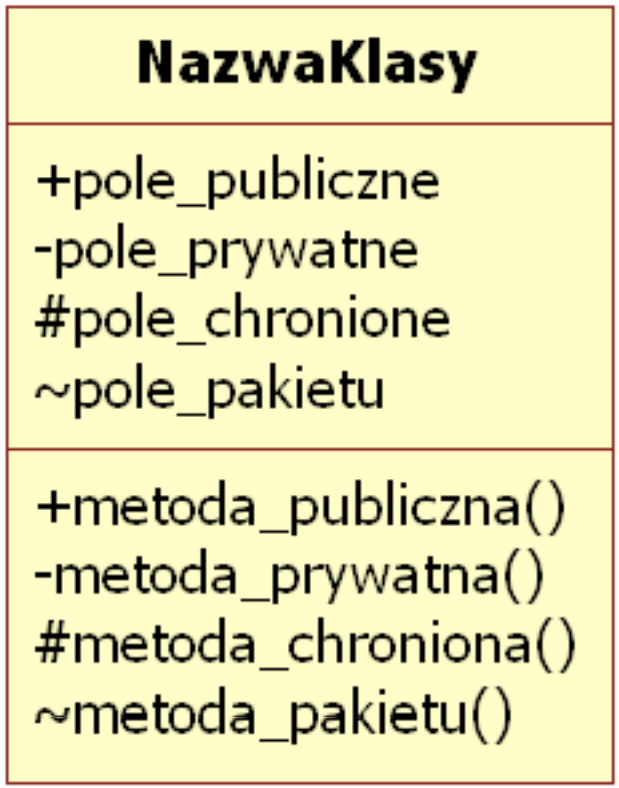


Nazwa ' ' y

Operacje - metody

Budując diagram klasy można pozostawić pusty zbiór zarówno pól

Dostęp do pól i metod



Specyfikatory dostępu:

+ inaczej public,

-Inaczej private,

inaczej protected,

~ inaczej package (w Javie nic)

Co oznaczają te specyfikatory ->

Specyfikatory dostępu:

+ inaczej public, oznacza możliwy dostęp do oznaczonego elementu z dowolnego kodu wewnątrz klasy oraz poza kodem tej klasy (np. utworzony obiekt w innej klasie może mieć dostęp do takiego elementu),

- inaczej private, oznacza możliwy dostęp do oznaczonego elementu TYLKO w obrębie kodu danej klasy (na zewnątrz dany element jest niewidoczny i niedostępny),

inaczej protected, oznacza możliwy dostęp do oznaczonego elementu w obrębie kodu danej klasy, w obrębie klasy, która dziedziczy po tej klasie oraz w obrębie tego samego pakietu klas,

~ inaczej package (w Javie nic), dostęp z kodu klas tego samego pakietu

Język JAVA - dziedziczenie

```
class Jedi{
        public String nazwa="Luke";
        private int moc=12;
        protected int liczbaUczniow=3;
        String kolorMiecza="zielony";          }//koniec class Jedi
class MasterJedi extends Jedi{
        public void opis(){
                System.out.println("MJ - Nazwa: "+nazwa);
                //System.out.println("MJ - Moc: "+moc); //dostęp do pola private - błąd
                System.out.println("MJ - Liczba uczniów: "+liczbaUczniow);
                System.out.println("MJ - Kolor miecza: "+kolorMiecza);
        } //koniec opis()       }//koniec class MasterJedi
public class PublicJedi{
        public static void main(String []a){
                Jedi j = new Jedi();
                MasterJedi mj = new MasterJedi();
                mj.opis();
                System.out.println("PJ - Nazwa: "+j.nazwa);
                //System.out.println("PJ - Moc: "+j.moc); //dostęp do pola private - błąd
                System.out.println("PJ - Liczba uczniów: "+j.liczbaUczniow);
                System.out.println("PJ - Kolor miecza: "+j.kolorMiecza);
        }//koniec main()     }//koniec class PublicJedi
```

Zilustrujmy również dostęp dla specyfikatorów:
-protected
-package (nieoznaczany).
W tym celu potrzebne nam dwa pakiety: wrog i nasi.

Kod klasy wrog.Sith.java

```
package wrog;
public class Sith{
        public String nazwa="Darth Vader";
        private int moc=16;
        protected int liczbaUczniow=0;
        String kolorMiecza="czerwony";
}//koniec class Sith
```

cdn.

Język JAVA - dziedziczenie

```
package nasi;
import wrog.Sith;
class MasterSith extends Sith{
        public void opis(){
          System.out.println("MS - Nazwa: "+nazwa); //public - OK
          //System.out.println("MS - Moc: "+moc); //dostęp do pola private - błąd
          System.out.println("MS - Liczba uczniów: "+liczbaUczniow);//protected - OK
          //System.out.println("MS - Kolor miecza: "+kolorMiecza);
          //dostęp do kolorMiecza tylko w ramach tego samego pakietu - błąd
        }//koniec opis()                    }//koniec class MasterSith
public class Jedi{
        public static void main(String []a){
                Sith s = new Sith();
                MasterSith ms = new MasterSith();
                ms.opis();
                System.out.println("J - Nazwa: "+s.nazwa); //public - OK
                //System.out.println("J - Moc: "+s.moc); //dostęp do pola private - błąd
                //System.out.println("J - Liczba uczniów: "+s.liczbaUczniow);
                //Jedi nie dziedziczy po Sith i jest w innym pakiecie - błąd
                //System.out.println("J - Kolor miecza: "+s.kolorMiecza);
                //dostęp do kolorMiecza tylko w ramach tego samego pakietu - błąd
        }//koniec main()                    }//koniec class Jedi
```

Poznaliśmy podstawowe specyfikatory dostępu. Kiedy je stosować?

Jedno z głównych założeń paradygmatu (podstawowej zasady, teorii) modelu obiektowego nazywane jest **enkapsulacją**.

Enkapsulacja (**hermetyzacja**) – ukrywanie implementacji, ***udostępnianie obiektom tylko tego, co jest im niezbędne do zamierzonego działania, najczęściej za pośrednictwem metod.***

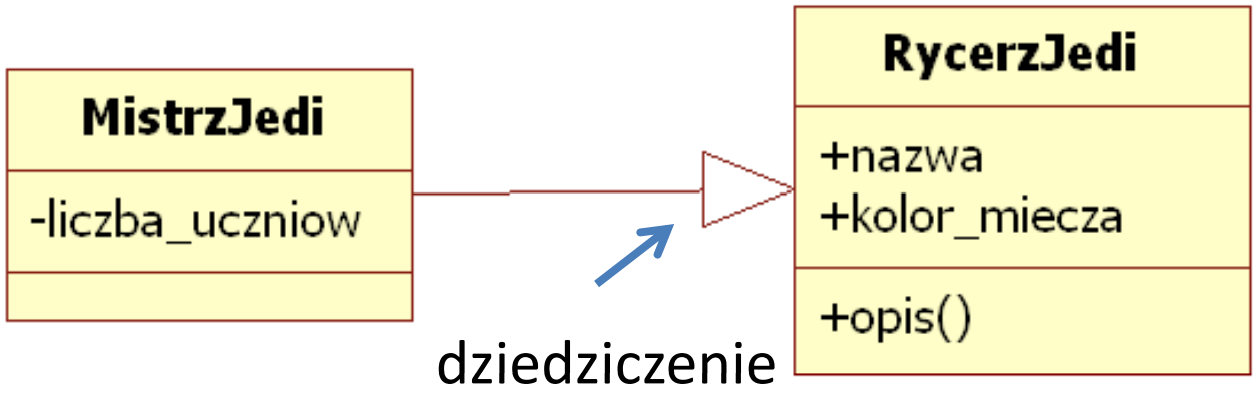
Wniosek – co się da oznaczamy private. ***Dostęp do pól poprzez metody typu set*** (ustaw wartość pola prywatnego) ***i get*** (pobierz wartość pola prywatnego). Np. pole tylko do odczytu = pole private oraz metoda typu get. ***Zestaw dostępnych metod*** (dla obiektów poza kodem klasy) ***stanowi niejako interfejs, jaki jest udostępniany na zewnątrz***!

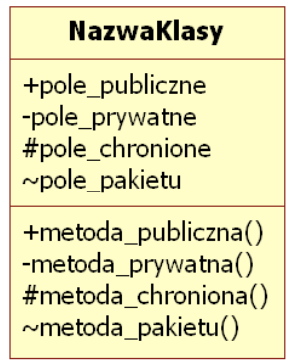
Jeśli z jakiś przyczyn chcemy mieć bezpośredni dostęp do pól i metod to zmieniamy ich specyfikator na package->protected->public.

Język JAVA – UML – diagramy klas

Wróćmy do diagramów UML opisujących związki pomiędzy klasami:

1. **dziedziczenie**



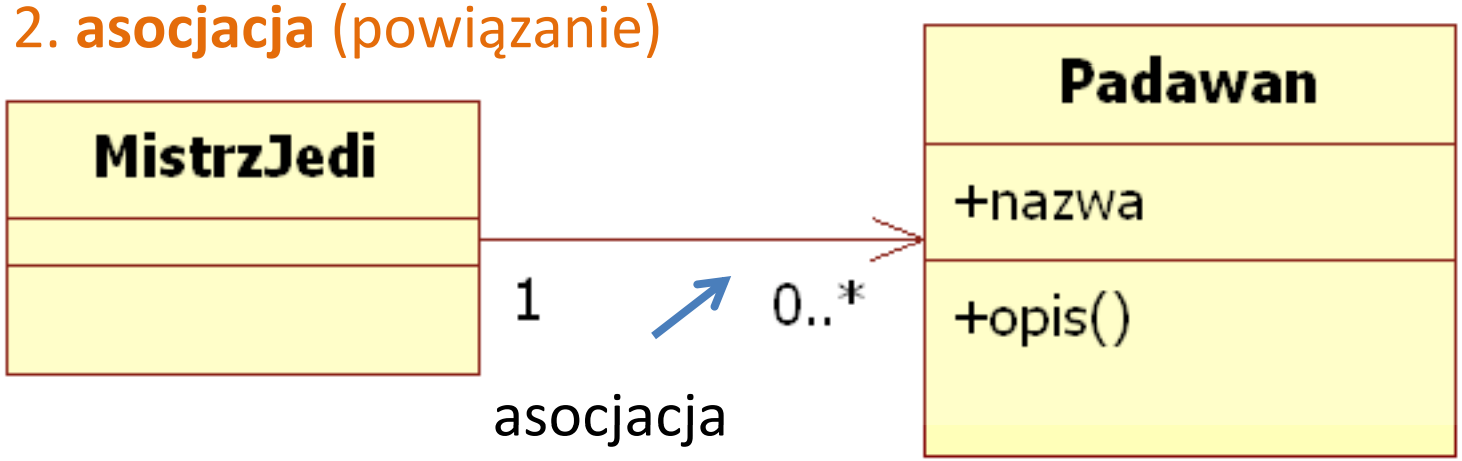


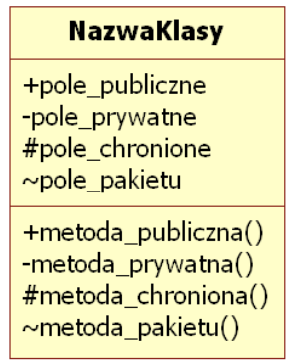
Równoważna realizacja diagramu klas w Javie:

```
class RycerzJedi{
        public String nazwa;
        public String kolor_miecza;
        public void opis(){  /* instrukcje metody*/   }
}//koniec class RyceczeJedi

class MistrzJedi extends RycerzJedi{
        private int liczba_uczniow=0;
}//koniec class MistrzJedi
```

Wróćmy do diagramów UML opisujących związki pomiędzy klasami:

2. **asocjacja** (powiązanie)





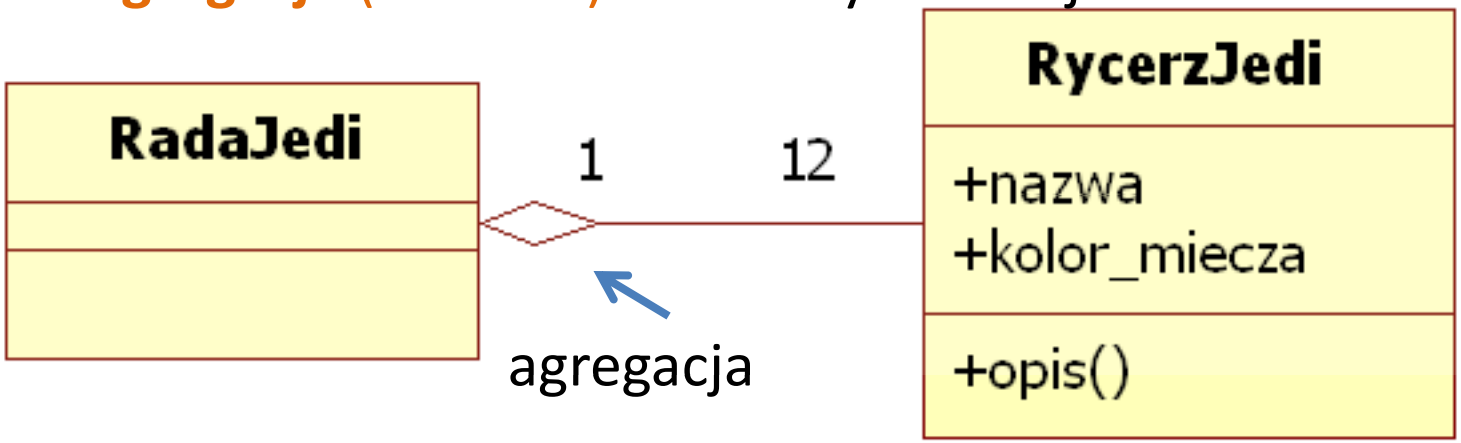
Równoważna realizacja diagramu klas w Javie:

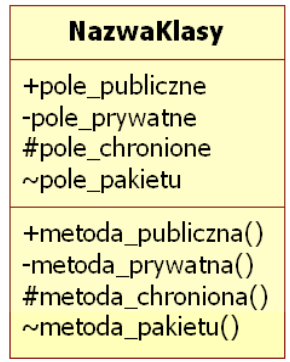
```
class Padawan{
        public String nazwa;
        public void opis(){  /* instrukcje metody*/   }
}//koniec class Padawan

class MistrzJedi {
        private Padawan[] uczniowie; //tablica i jej wartości  ustawiane przez metody
}//koniec class MistrzJedi
```

Język JAVA – UML – diagramy klas

Wróćmy do diagramów UML opisujących związki pomiędzy klasami:

3. **agregacja** (złożenie) – w nowych wersjach UML nieużywane



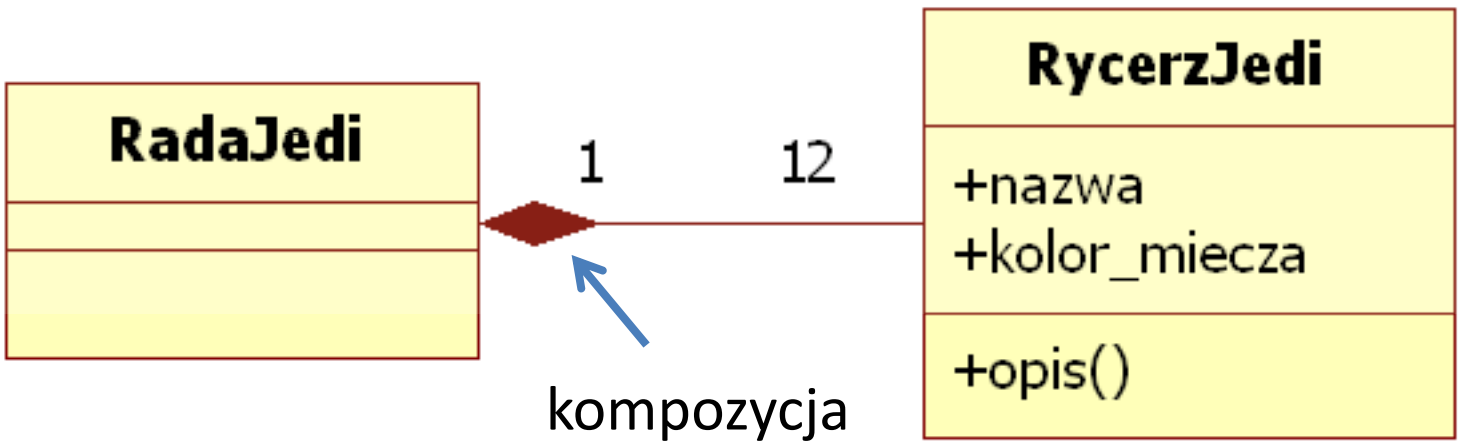


Równoważna realizacja diagramu klas w Javie (w Javie tak samo jak asocjacja):

```
class RycerzJedi{
        public String nazwa;
        public String kolor_miecza;
        public void opis(){  /* instrukcje metody*/   }
}//koniec class RyceczeJedi

class MistrzJedi {
        private RycerzJedi [12] rycerze; //tablica i jej wartości  ustawiane przez metody
}//koniec class RadaJedi
```

Wróćmy do diagramów UML opisujących związki pomiędzy klasami:

4. **kompozycja** (złożenie) – jak asocjacja, tylko odnosi się do związku zawierania: obiekt jednej klasy zawiera na wyłączność obiekty innej klasy. Jeśli "ginie" obiekt zawierający "giną" również obiekty zawarte (np. poprzez zastosowanie destruktora – w Javie brak!).



Równoważna realizacja diagramu klas w Javie (w Javie tak samo jak asocjacja, brak destruktorów!):

KOD IDENTYCZNY JAK DLA AGREGACJI !!!

Modelowanie z zastosowaniem diagramu klas ułatwia czasem możliwość zobaczenia całego problemu jaki się rozważa wytwarzając oprogramowanie.

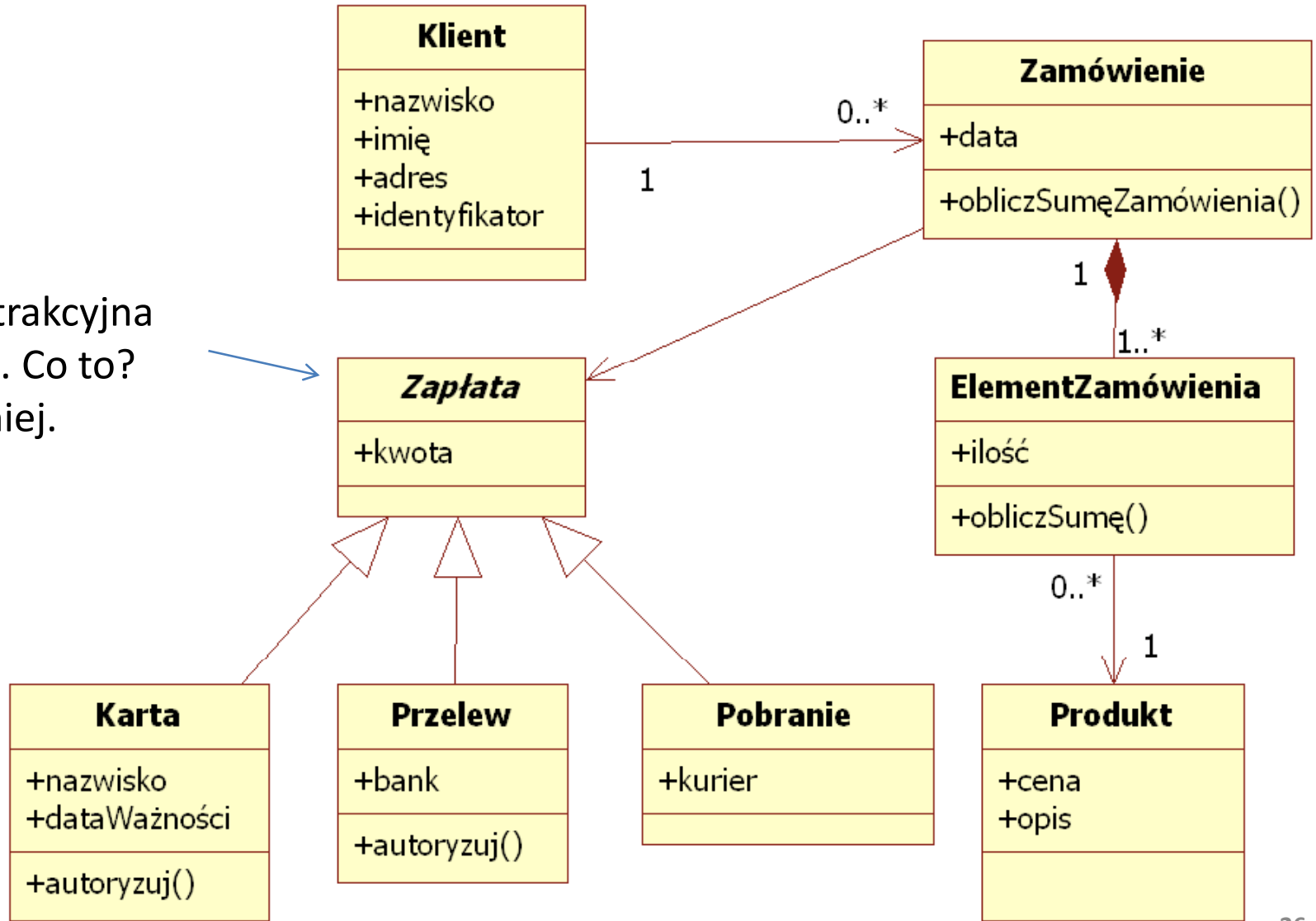
CASE (Computer Assisted Software/System Engineering)

Istnieją pakiety oprogramowania i moduły umożliwiające utworzenie diagramu klas, a później automatyczną generację kodu (engineering – wytwarzanie). Te same pakiety dają często możliwość wytwarzania wstecznego (reverse engineering), czyli mając kod źródłowy budowany jest diagram klas.

Diagram klas może być również wykorzystany jako model danych (projektując schemat bazy danych), a później odwzorowany (ang. mapping) na schemat bazy relacyjnej/hybrydowej.

Przykładowy diagram klas

Klasa abstrakcyjna (kursywa). Co to? Opis później.



Specyfikatory dostępu:

Oprócz 4 poznanych specyfikatorów dostępu (public, private, protected, pusty czyli package) istnieją jeszcze inne, specjalne oznaczenia.
Takim przykładowym oznaczeniem jest final.
Oznaczenie pola (zmiennej) specyfikatorem final oznacza, że jest to ostateczna definicja zmiennej, czyli jest to stała!.

Co ma final do dziedziczenia?
Otóż jeśli oznaczymy klasę jako final wówczas jest to ostateczna definicja klasy i nie można jej rozszerzać, czyli nie można po takiej klasie dziedziczyć!

Co do tej pory wiemy o dziedziczeniu w Javie:

- Jest to operacja umożliwiająca tworzenie nowych typów danych (klas) poprzez uszczegółowienie (specjalizacja) innych.
- Można dziedziczyć tylko po jednej klasie na raz. Możliwe jest dziedziczenie kaskadowe (lista). Każda klasa (nawet jeśli tego jawnie nie zapiszemy) dziedziczy po klasie Object (praojciec wszystkich typów; typ wszystkich tworzonych obiektów).
- W procesie dziedziczenia można ograniczyć dostęp do pól i metod posługując się specyfikatorami dostępu (m.in. private).
- Możemy zabezpieczyć się przed dziedziczeniem oznaczając klasę jako final.

Istnieją również inne własności dziedziczenia, niektóre zostaną omówione w kolejnej części wykładu !

Plan wykładu:

1. Podstawy modelowania obiektowego
2. Konstruktory
3. Dziedziczenie, związki pomiędzy klasami, UML
4. Polimorfizm
5. Klasy abstrakcyjne i interfejsy
6. Adaptery, klasy anonimowe
7. Tablice, kolekcje i typ wyliczeniowy