POLITECHNIKA GDAŃSKA

Język Java podstawy

Jacek Rumiński

Język Java podstawy

Jacek Rumiński

Katedra Inżynierii Biomedycznej,
Wydział Elektroniki, Telekomunikacji i Informatyki
Politechnika Gdańska

1. Wprowadzenie do modelowania obiektowego

2. Klasy i konstruktory

3. Klasy i dziedziczenie

Dziedziczenie

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{
(...)
}
```

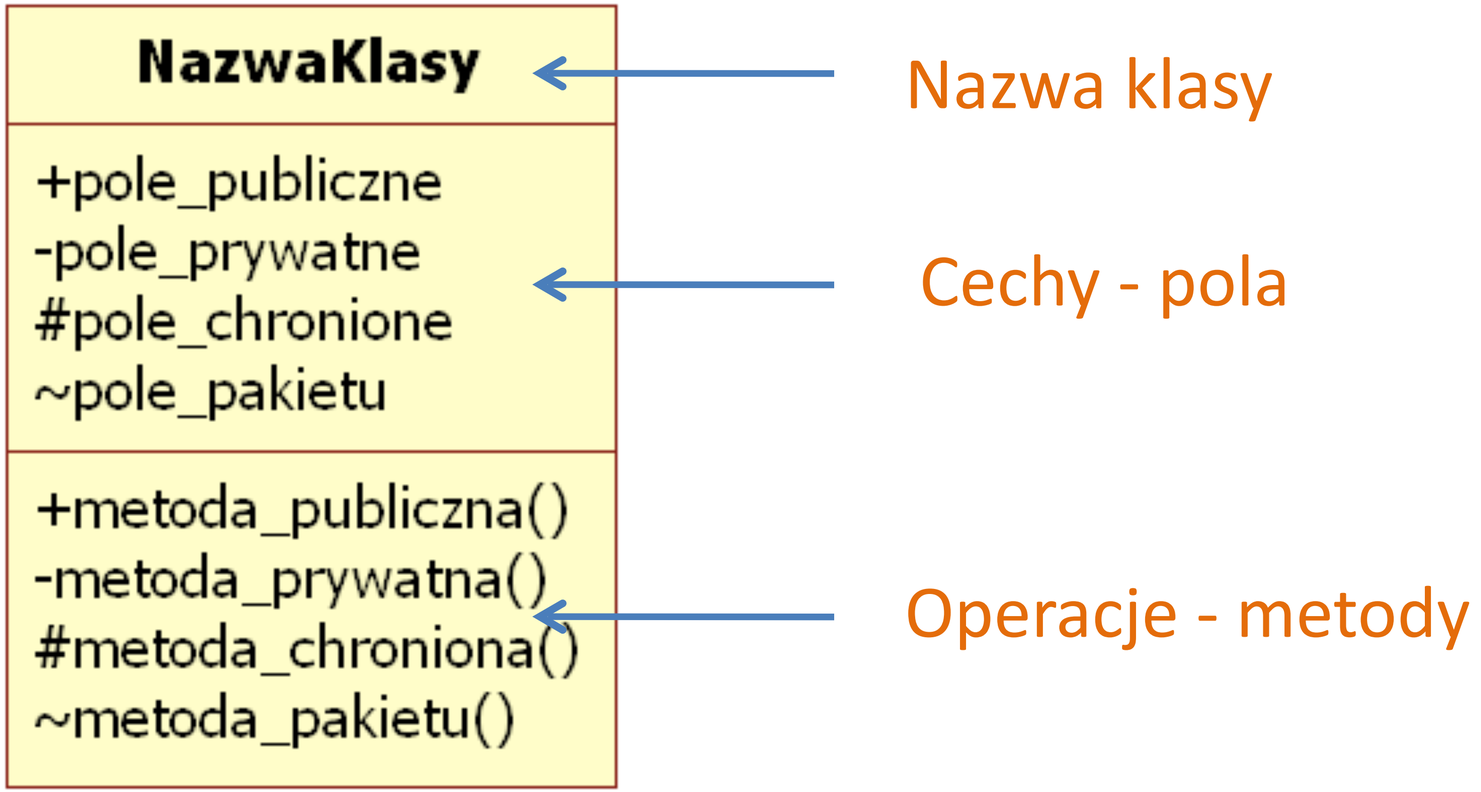
Projektując zestaw klas warto przemyśleć problem zależności pomiędzy klasami.
Utworzenie klasy bazowej w jak najbardziej uniwersalny sposób umożliwia dziedziczenie po niej, czyli tworzenia bardziej szczegółowych klas dla danych zastosowań (REUSE !).

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.

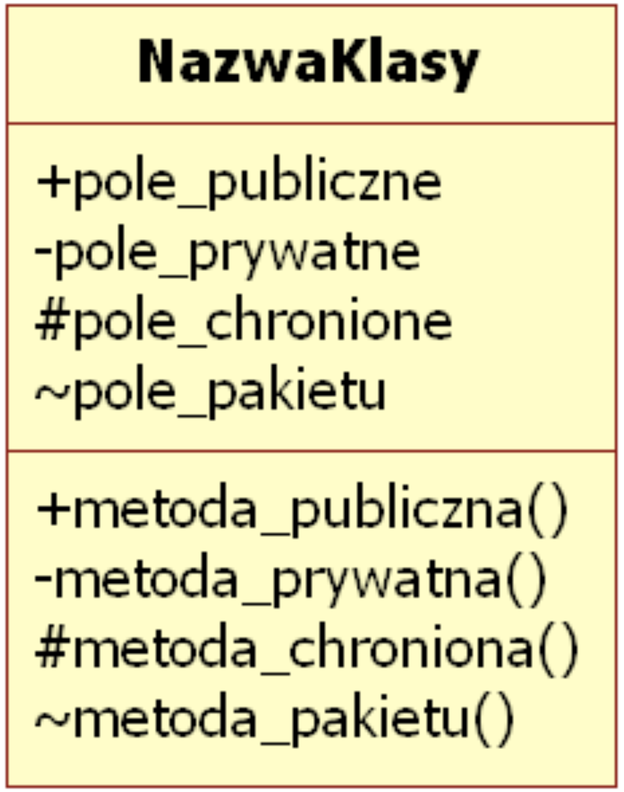
Projektowanie klas i związków pomiędzy nimi (dziedziczenie i inne) ułatwia specjalna notacja w formie diagramów wprowadzona przez Ujednolicony Język Modelowania (UML – Unified Modelling Language, www.omg.org).

Wprowadźmy kilka podstawowych zasad budowania diagramów klas.



Budując diagram klasy można pozostawić pusty zbiór zarówno pól jak i metod.

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

```
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.

```
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?

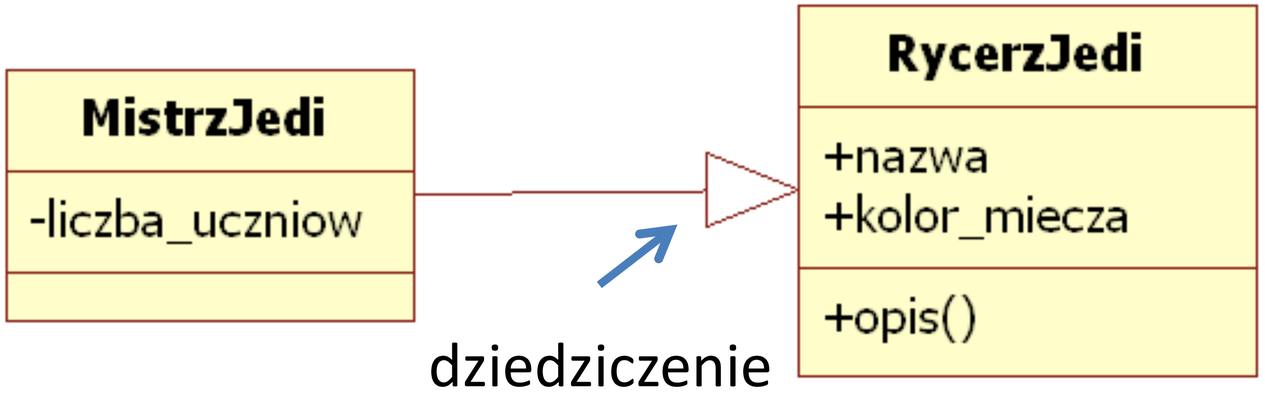
Jedno z głównych założeń paradygmatu (podstawowej zasady, teorii) modelu obiektowego nazywane jest hermetyzacją (enkapsulacją).

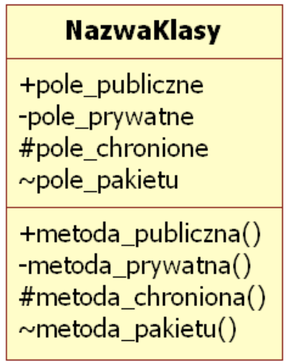
Hermetyzacja – ukrywanie implementacji, udostępnianie obiektom tylko tego, co jest im niezbędne do zamierzonego działania, najczęściej za pośrednictwem metod.

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.

Wróćmy do diagramów UML opisujących związki pomiędzy klasami:

1. **dziedziczenie**





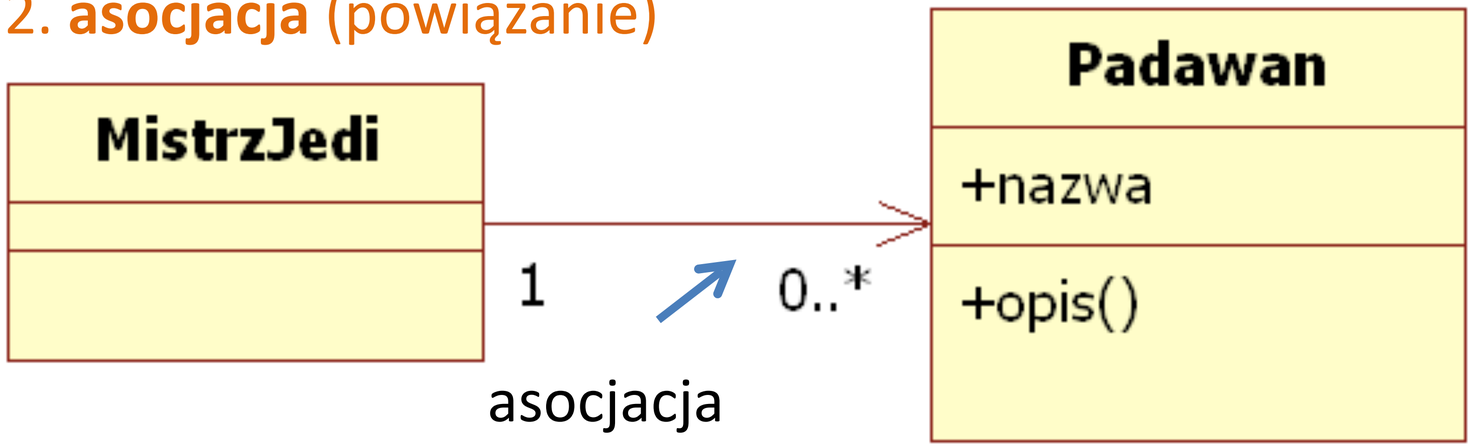
Równoważna realizacja diagramu klas w Javie:

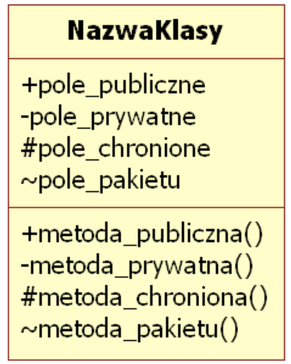
```
class RycerzJedi{
        public String nazwa;
        public String kolor_miecza;
        public void opis(){  /* instrukcje metody*/   }
}//koniec class RyceczeJedi

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

Wróćmy do diagramów UML opisujących związki pomiędzy klasami:

2. **asocjacja** (powiązanie)





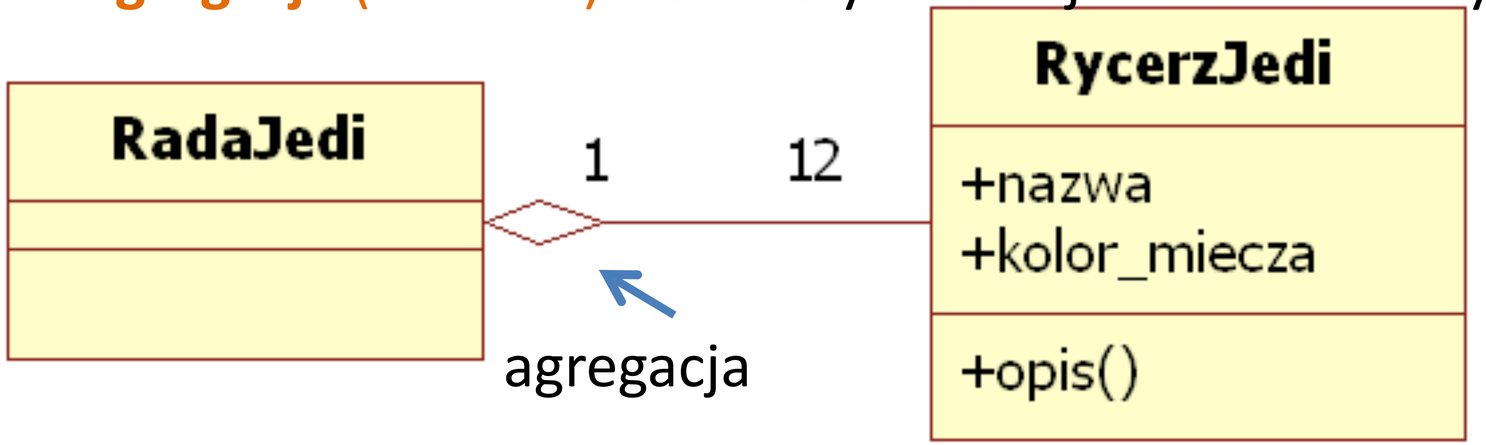
Równoważna realizacja diagramu klas w Javie:

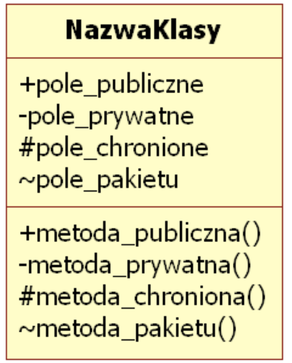
```
class Padawan{
        public String nazwa;
        public void opis(){  /* instrukcje metody*/   }
}//koniec class Padawan

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

Wróćmy do diagramów UML opisujących związki pomiędzy klasami:

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





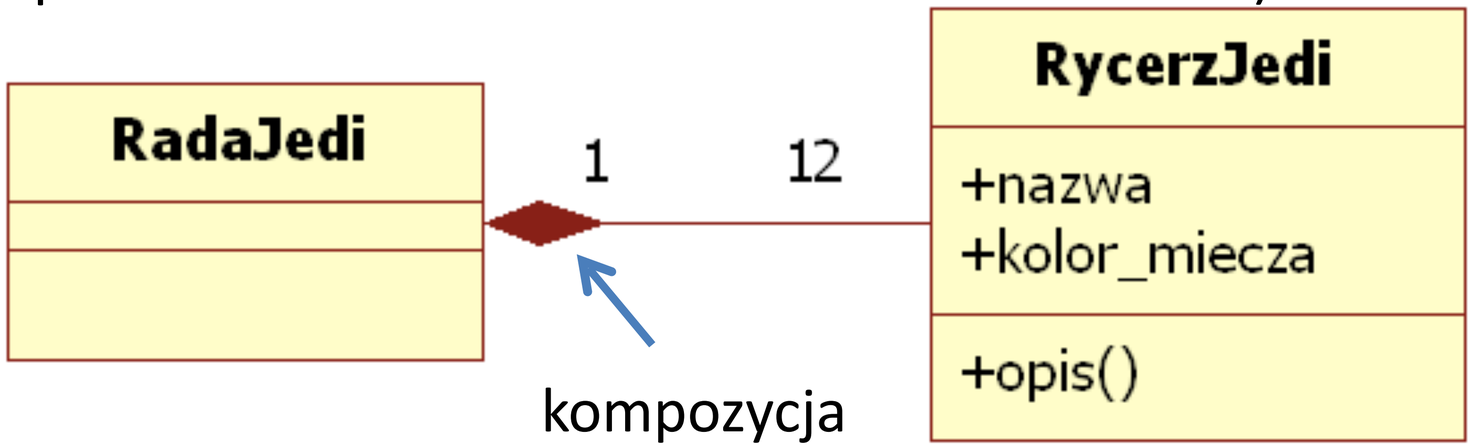
Równoważna realizacja diagramu klas w Javie (w Javie tak samo jak asocjacja):

```
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.

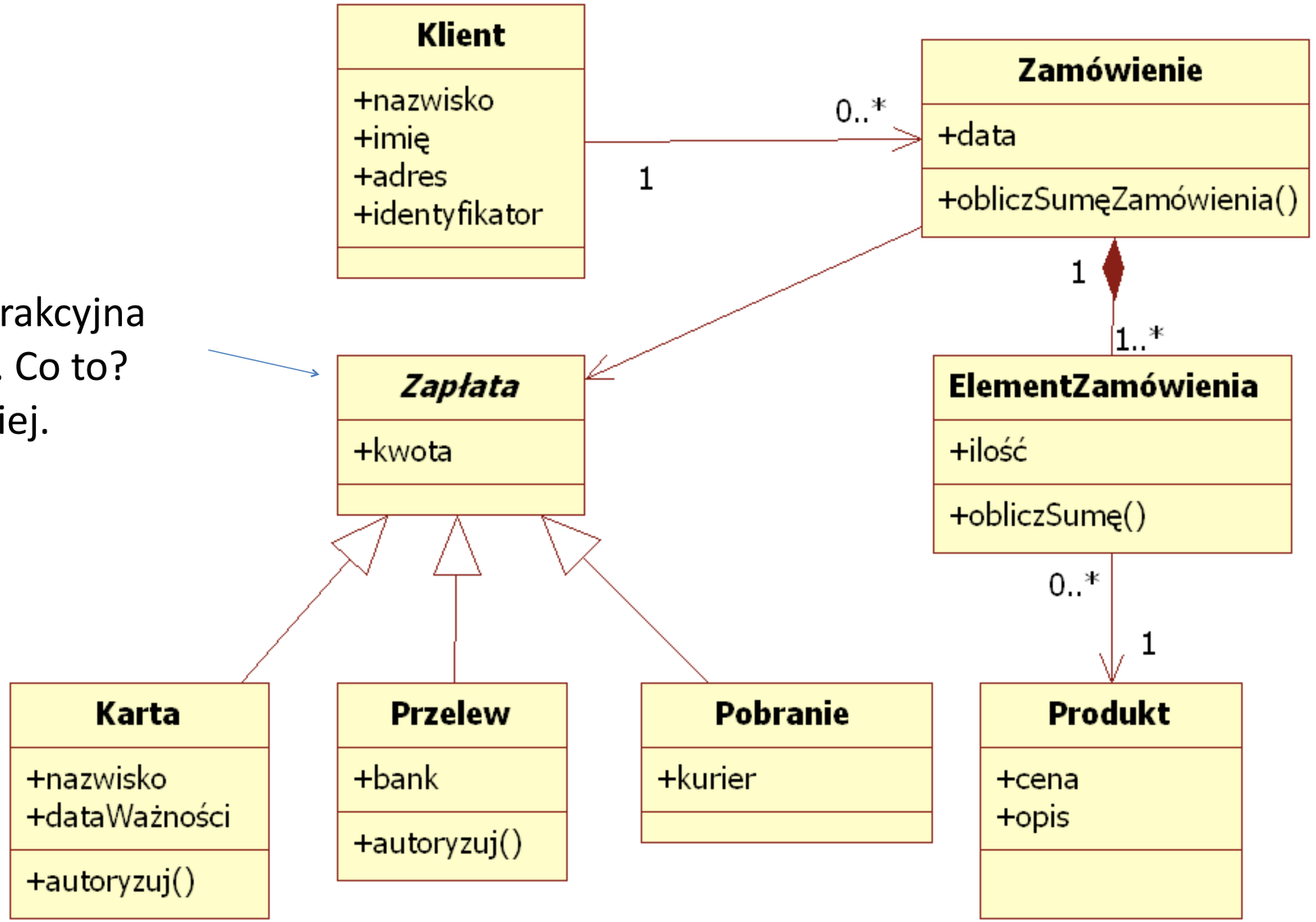
CASE (Computer Assisted Software/System Engineering)

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 zajęć !

Polimorfizm – wielopostaciowość .

Jedno z głównych założeń paradygmatu obiektowego (1. abstrakcja 2. dziedziczenie, 3. hermetyzacja, 4. polimorfizm).

Istnieją różne rodzaje polimorfizmu, ale w zasadzie własność ta oznacza:
ZDOLNOŚĆ do prezentacji takiego samego interfejsu dla różnych form (typów danych)

Inaczej jest to również
WZORZEC, zgodnie z którym klasy (metody) mogą mieć różną funkcjonalność współdzieląc taki sam interfejs.

Co to jest ten interfejs?

Polimorfizm

Istnieją różne rodzaje i definicje polimorfizmu. Dla potrzeb naszych zajęć rozpatrzymy tylko dwie postaci polimorfizmu:

- polimorfizm ad-hoc,
- polimorfizm przez przesłanianie.

Polimorfizm ad-hoc oznacza wykorzystanie skończonej liczby typów danych w różnych konfiguracjach metody o tej samej nazwie, np. suma(int a, int b) oraz suma(double a, double b).

Polimorfizm przez przesłanianie oznacza nadpisywanie metod w procesie dziedziczenia -ta sama metoda wykonywać będzie co innego.

Polimorfizm przez przesłanianie (method overriding)

Polimorfizm przez przesłanianie oznacza nadpisywanie metod w procesie dziedziczenia -ta sama metoda wykonywać będzie co innego.

```
class RycerzykJedi{
        public void walczyMieczem(int moc){
                System.out.println("Walczę z siłą : "+moc);
        } //koniec walczyMieczem()
} //koniec class RycerzykJedi
public class  SuperMistrzJedi extends RycerzykJedi{
        public void walczyMieczem(int moc){//ta sama deklaracja/sygnatura metody
                moc=moc+10; //nadpisujemy treść metody z klasy RycerzykJedi
                System.out.println("Walczę z siłą : "+moc);
        }//koniec walczyMieczem()
        public static void main(String a[]){
                SuperMistrzJedi mj = new SuperMistrzJedi();
                mj.walczyMieczem(20);
        }//koniec main()
}//koniec public class SuperMistrzJedi
```

Kod programu: SuperMistrzJedi.java

```
class RycerzykJedi{
        int wiek=30;
        //eksperyment: dodaj final po public i skompiluj. Co się stanie i dlaczego?
        public void walczyMieczem(int moc){
                System.out.println("Mam "+wiek+" lat.Walczę z siłą : "+moc);
        } //koniec walczyMieczem()
} //koniec class RycerzykJedi

public class  SuperMistrzJedi extends RycerzykJedi{
        int wiek=40;//tylko po co na nowo definiować ten sam identyfikator, lepiej wiek=40
        public void walczyMieczem(int moc){//ta sama deklaracja/sygnatura metody
                moc=moc+10; //nadpisujemy treść metody z klasy RycerzykJedi
                System.out.println("Mam "+wiek+" lat.Walczę z siłą : "+moc);
        }//koniec walczyMieczem()
        public static void main(String a[]){
                SuperMistrzJedi mj = new SuperMistrzJedi();
                mj.walczyMieczem(20);
        }//koniec main()
}//koniec public class SuperMistrzJedi
```

Polimorfizm ad-hoc

Wiele metod (konstruktorów), operatorów o tej samej nazwie lecz z argumentami różnych typów. Przeciążanie metod i operatorów!
W Javie nie można przeciążać operatorów (tylko "+" jest przeciążony").

```
public class  MalyJedi {
        public String sumujMoc(int a, String b){
                return (a+b); //przeciążenie operatora dodawania (dodanie liczby i tekstu)
        }
        public void walczyMieczem(int moc){//argument typu int
                System.out.println("Walczę z siłą : "+moc);
        }//koniec walczyMieczem()
        public void walczyMieczem(String moc){//argument typu String
                System.out.println("Walczę z siłą : "+moc);
        }//koniec walczyMieczem()
        public static void main(String a[]){
                MalyJedi mj = new MalyJedi();
                mj.walczyMieczem(20); mj.walczyMieczem("trzydzieści");
        }//koniec main()
}//koniec public class MalyJedi
```

Polimorfizm ad-hoc

Stosowanie wielu konstruktorów oraz wielu metod o tej samej nazwie jest dobrą praktyką umożliwiającą wykonanie danej operacji bez względu na rodzaj typu danych (metody powinny zrealizować ewentualną konwersję).

Poniżej pokazano fragment dokumentacji klasy **String**, opisujący przeciążoną funkcję **indexOf**, zwracającą pozycję wystąpienia znaku lub ciąg znaków w danym łańcuchu znaków, będącym wartością obiektu klasy **String**.

int	**indexOf**(int ch) Returns the index within this string of the first occurrence of the specified character.
int	**indexOf**(int ch, int fromIndex) Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
int	**indexOf**(String str) Returns the index within this string of the first occurrence of the specified substring.
int	**indexOf**(String str, int fromIndex) Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

Polimorfizm ad-hoc

Poniżej pokazano fragment dokumentacji klasy String, opisujący listę konstruktorów (specjalnych metod o tej samej nazwie).

Constructor Summary
String() Initializes a newly created String object so that it represents an empty character sequence.
String(byte[] bytes) Constructs a new String by decoding the specified array of bytes using the platform's default charset.
String(byte[] bytes, Charset charset) Constructs a new String by decoding the specified array of bytes using the specified charset.
String(byte[] ascii, int hibyte) **Deprecated**. *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a Charset, charset name, or that use the platform's default charset.*
String(byte[] bytes, int offset, int length) Constructs a new String by decoding the specified subarray of bytes using the platform's default charset.
String(byte[] bytes, int offset, int length, Charset charset) Constructs a new String by decoding the specified subarray of bytes using the specified charset.
String(byte[] ascii, int hibyte, int offset, int count) **Deprecated**. *This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a Charset, charset name, or that use the platform's default charset.*
String(byte[] bytes, int offset, int length, String charsetName) Constructs a new String by decoding the specified subarray of bytes using the specified charset.
String(byte[] bytes, String charsetName) Constructs a new String by decoding the specified array of bytes using the specified charset.

Na koniec kilka eksperymentów...

Zapraszamy na kolejne zajęcia