



Gdy zaczniesz programować obiektowo możesz już nigdy więcej nie chcieć wrócić do programowania strukturalnego !

PROGRAMOWANIE ZORIENTOWANE OBIEKTOWO

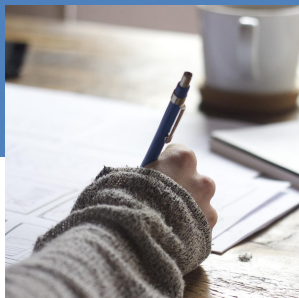
Przemysław Grzesiowski
23/24 czerwca 2018

OBJECT ORIENTED PROGRAMMING

Agenda

1. strukturalnie vs obiektowo
2. obiekt a klasa
3. jak żyć z klasą?
4. przeciążanie (overload)
5. definicje
6. konstruktor
7. najważniejsze cechy OOP
8. dziedziczenie
9. super()
10. final i abstract
11. przesłanianie metod (override)
12. polimorfizm
13. modyfikatory dostępu
14. enkapsulacja
15. abstract vs concrete
16. interfejsy

Cześć



Jestem Przemek, jestem programistą.

Agenda

1. strukturalnie vs obiektowo
2. obiekt a klasa
3. jak żyć z klasą?
4. przeciążanie (overload)
5. definicje
6. konstruktor
7. najważniejsze cechy OOP
8. dziedziczenie
9. super()
10. final i abstract
11. przesłanianie metod (override)
12. polimorfizm
13. modyfikatory dostępu
14. enkapsulacja
15. abstract vs concrete
16. interfejsy

1. strukturalnie vs obiektoowo

Programowanie strukturalne

wykorzystuje funkcje, te funkcje pracują na pewnych zmiennych i oddają wynik. Jednak nie komunikują się między sobą.

Programowanie obiektowe grupuje zmienne i metody w jedną całość (obiekt). Wywołanie metody powoduje zmianę stanu obiektu (jego atrybutów). Zmienne powiązane są logicznie ze sobą tak jak w rzeczywistości. Zmiana jednego atrybutu może zmienić wartość innego.

Anatomia

MyApp.java

```
public class MyApp {
```



```
}
```

Source file - (.java)



Class



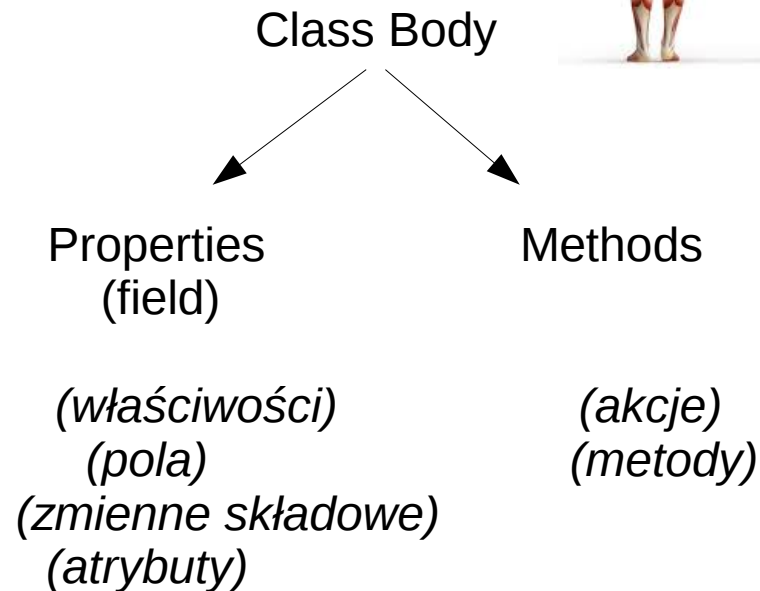
Class body

*Nazwa pliku musi być koniecznie taka sama jak nazwa klasy !!
Jedna klasa publiczna → jeden plik.*

Anatomia Klasy w Javie



```
public class MyApp {  
    /* Properties */  
  
    /* Methods */  
}
```



Przepis na klasę

- informacje które obiekt zna / o których wie (POLA),
- czynności które obiekt wykonuje (METODY).

**zmienne
składowe**
(stan)

metody
(działanie)

Piosenka
tytuł wykonawca
określTytuł() określWykonawce() odtwórz()

wie

wykonuje

Przykłady

Koszyk
zawartosc
dodajDoKoszyka() usunZKoszyka() sprawdz()

wie

wykonuje

Przycisk
etykieta kolor
okreslKolor() okreslEtykiete() zwolnij() wcisnij()

wie

wykonuje

Alarm
alarmCzas alarmTryb
okreslCzasAlarmu() pobierzCzasAlarmu() czyAlarmUstawiony() wstrzymaj()

Ćwiczenie



Zaostrz ołówek

Poniżej wpisz, co obiekt Telewizor powinien wiedzieć i robić.



Telewizor

**zmienne
składowe**

metody

Ćwiczenie

Wykonaj to samo ćwiczenie dla obiektów:

*Pies, Kot, Radio, Zmywarka, Kuchenka,
Piekarnik, Samochód, Motocykl, Samolot,
Drukarka, Autobus, Krzesło, Wiertarka,
Budzik, Książka, Telefon, Wentylator*

Classname (Identifier)	Student	Circle
Data Member (Static attributes)	name grade	radius color
Member Functions (Dynamic Operations)	getName() printGrade()	getRadius() getArea()

SoccerPlayer	Car
name number xLocation yLocation	plateNumber xLocation yLocation speed
run() jump() kickBall()	move() park() accelerate()

Examples of classes

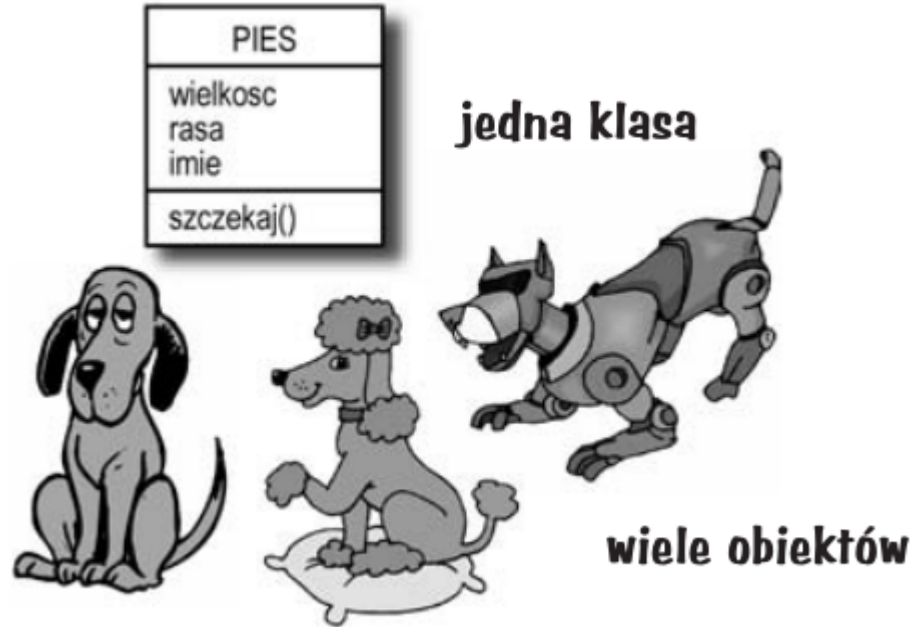
```
[modyfikator dostępu] [abstract] class NazwaKlasy {  
    // definicja pól i metod  
}
```


2.

Obiekt a klasa

Klasa a obiekt

Klasa vs obiekt



Klasa vs obiekt

Klasa - matryca/szablon do tworzenia obiektów.

Obiekt = obiekt danej klasy

Klasa vs obiekt

Klasa - matryca/szablon do tworzenia obiektów.
Obiekt = konkretny obiekt danej klasy (instancja)

```
Pies p1 = new Pies("Azor", "kundel", 10);  
Pies p2 = new Pies("Bysio", "labrador", 40);  
Pies p3 = new Pies("Juras", "jamnik", 12);
```

Classname	<u>paul:Student</u>	<u>peter:Student</u>
Data Members	name="Paul Lee" grade=3.5	name="Peter Tan" grade=3.9
Member Functions	getName() printGrade()	getName() printGrade()

Two instances of the **Student** class

KLASA



Abstrakcyjny byt określający
zbiór Obiektów o takich
samyh właściwościach.

3.

Jak żyć z klasą ?

Przepis na klasę

1. Wybierz nazwę klasy (z wielkiej litery !!)
2. Stwórz plik (np. MojaKlasa.java)
3. Dodaj szczyptę **pól**
4. Dodaj odrobinę zachowań (**metod**)
5. Jeśli chcesz dodaj **konstruktor**.
6. Całość wymieszaj (skompiluj)

Klasa gotowa, jeść z operatorem new(), metody odpalać przez kropkę.

METODA



```
public class Pies {
```

```
    public void metoda() {
```

```
        //logika metody (ciało)
```

```
    }
```

```
}
```

modyfikatory
np. modyfikator dostępu: private

nazwa metody

co zwraca po wykonaniu?
(tutaj: void – nic nie zwraca)

METODA



```
public class Pies {
```

```
    public void metoda(String input) {  
        //logika metody (ciało)  
    }
```

Parametry wejściowe
(argumenty)

```
}
```

METODA



```
public class Pies {
```

```
    public String metoda(String input) {
```

```
        //logika metody (ciało)
```

```
        return x;
```

```
    }
```

```
}
```

zwracamy coś na
wyjściu



Metoda -

Ćwiczenie 3.1

App

```
+main(args:String[])  
+hello(imie:String): String
```

- otwórz projekt “emptyProject”
- stwórz metodę *hello* w klasie App
- metoda przyjmuje jeden argument wejściowy typu String
- metoda zwraca jeden argument typu String
- metoda ma dwa modyfikatory - jest publiczna i statyczna (public static)
- Metoda zwraca: **Witaj [imie]**
(gdzie [imie] to parametr wejściowy)
- Odpal metodę *hello* z poziomu metody *main*, zweryfikuj działanie

POLE



```
public class Pies {  
    public String imie;  
    Integer waga;  
}
```

Pole w klasie

```
[modyfikator] typPole nazwaPola;
```

element	opis	Wymagany?
[modyfikator]	określa tryb dostępu (np. Private) oraz właściwości (np. static)	nie
typPola	typ przechowywanych danych (np. String, Integer)	tak
nazwaPola	nazwa	tak

KONSTRUKTOR



```
public class Pies {  
    public Pies() {  
        //"instrukcja" jak tworzyć obiekt  
    }  
}
```


KONSTRUKTOR



```
public class Pies {  
    public Pies() {  
        //"instrukcja" jak stworzyć obiekt  
    }  
}
```

Konstruktor jest “specjalną metodą”:

- ma taką samą nazwę jak nazwa klasy,
- jest odpalany automatycznie przy tworzeniu nowego obiektu klasy (poprzez operator new).

KONSTRUKTOR



```
public class Pies {  
    String imie;  
  
    public Pies(String imie) {  
        this.imie = imie;  
    }  
}
```

Konstruktor z parametrem

→ this daje dostęp do pól i metod

Ćwiczenie 3.2

Pies

rasa:String
waga:Integer
imie:String

szczekaj()

- Stwórz klasę Pies.java
 - konstruktor ma mieć 3 argumenty wejściowe
 - metoda *szczekaj()* ma wyrzucić na ekran następujący tekst: “[*rasa*] [*imie*] o wadze [*waga*] kg zaszczekał”
- Stwórz klasę Main.java, która utworzy 3 obiekty klasy Pies oraz wywoła metodę *szczekaj()* na każdym z nich
- Odpal program sprawdzając, czy wszystko się zgadza
- [*] dodaj pole: wzrost oraz metodę wyliczającą wskaźnik BMI

4. Przeciążanie (overload)

- * przeciążanie metod
- * przeciążanie konstruktorów

antyprzykład

```
public class FileSaver {  
    public void saveObject1(Object1 obj){  
        //preparing object1  
        //saving  
    }  
    public void saveObject2(Object2 obj){  
        //preparing object2 - step1  
        //preparing object2 - step2  
        //saving  
    }  
}
```

```
public class FileSaver {  
    public static void save(Object1 obj){  
        //preparing object1  
        //saving:  
        save(obj, "/obj/obj1.csv");  
    }  
    public static void save(Object2 obj){  
        //preparing object2 - step 1  
        //preparing object1 - step 2  
        //saving:  
        save(obj, "/obj/data/obj2.txt");  
    }  
    private static void save(Object obj, String path){  
        //saving object into given path  
    }  
}
```

Constructor overloading

```
public class TimeOfDay {  
    public int hour;  
    public int minute;  
  
    public TimeOfDay(int h, int m) {  
        hour    = h;  
        minute  = m;  
    }  
  
    public TimeOfDay(int h) {  
        hour    = h;  
        minute  = 0;  
    }  
}
```

Konstruktor nr.1

Konstruktor nr. 2

Constructor overloading

```
public class TimeOfDay {
    public int hour;
    public int minute;

    public TimeOfDay(int h, int m) {
        hour    = h;
        minute  = m;
    }

    public TimeOfDay(int h) {
        // wywołanie pierwszego konstruktora przy użyciu this(...)
        this(h, 0);
    }
}
```


Constructor overloading

```
public class TimeOfDay {  
    public int hour;  
    public int minute;  
  
    public TimeOfDay(int h, int m) {  
        hour    = h;  
        minute  = m;  
    }  
  
    public TimeOfDay(int h) {  
        // wywołanie pierwszego konstruktora przy użyciu this(...)  
        this(h, 0);  
    }  
}
```

Jak zainicjujesz obiekt przedstawiający aktualną godzinę?

Constructor overloading

```
public class Prostokat {  
    public int a;  
    public int b;  
  
    public Prostokat(int a, int b) {  
        this.a = a;  
        this.b = b;  
    }  
  
    public Prostokat(int a) {  
        this(a,a);  
    }  
}
```

Ćwiczenie 4.1 - zmieniamy klasę Pies

<u>Pies</u>
rasa:String waga:Integer imie:String
szczekaj()

- przeciąż konstruktor klasy Pies, dodając drugi konstruktor który ma 4 argumenty (rasa, waga, imie, wzrost); gdy użyty jest pierwszy konstruktor to wzrost=0
- dodaj konstruktor bezargumentowy, który zainicjuje “defaultowego” psa: jamnik Szarik o masie 100 kg i wzroście 180 cm :-)
- zmień metodę “szczekaj()” - dodając informację o wzroście: **“[rasa] [imie] o wadze [waga] kg, wzroście [wzrost] cm zrobił chał hał hau”**
- Upewnij się, że z poziomu klasy Main wykorzystujesz wszystkie 3 konstruktory, odpal program i zweryfikuj.

Przeciążanie (overload)

Ta sama nazwa, ale inne parametry

5. Definicje

OOP - Wikipedia

Programowanie obiektowe (ang. object-oriented programming) – paradygmat programowania, w którym programy definiuje się za pomocą obiektów – elementów łączących **stan** (czyli dane, nazywane najczęściej polami) i **zachowanie** (czyli procedury, tu: metody).

OOP - Wikipedia

Programowanie obiektowe (ang. object-oriented programming) – paradygmat programowania, w którym programy definiuje się za pomocą obiektów – elementów łączących **stan** (czyli dane, nazywane najczęściej polami) i **zachowanie** (czyli procedury, tu: metody). Obiektowy program komputerowy wyrażony jest jako zbiór takich obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań.

OOP - Wikipedia

Programowanie obiektowe (ang. object-oriented programming) – paradygmat programowania, w którym programy definiuje się za pomocą obiektów – elementów łączących **stan** (czyli dane, nazywane najczęściej polami) i **zachowanie** (czyli procedury, tu: metody). Obiektowy program komputerowy wyrażony jest jako zbiór takich obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań.

Podejście to różni się od tradycyjnego programowania proceduralnego, gdzie dane i procedury nie są ze sobą bezpośrednio związane. Programowanie obiektowe ma ułatwić pisanie, konserwację i wielokrotne użycie programów lub ich fragmentów.

OOP - Wikipedia

Programowanie obiektowe (ang. object-oriented programming) – paradygmat programowania, w którym programy definiuje się za pomocą obiektów – elementów łączących **stan** (czyli dane, nazywane najczęściej polami) i **zachowanie** (czyli procedury, tu: metody). Obiektowy program komputerowy wyrażony jest jako zbiór takich obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań.

Podejście to różni się od tradycyjnego programowania proceduralnego, gdzie dane i procedury nie są ze sobą bezpośrednio związane. Programowanie obiektowe ma ułatwić pisanie, konserwację i wielokrotne użycie programów lub ich fragmentów.

Największym atutem programowania, projektowania oraz analizy obiektowej jest zgodność takiego podejścia z rzeczywistością – mózg ludzki jest w naturalny sposób najlepiej przystosowany do takiego podejścia przy przetwarzaniu informacji.

6. Konstruktor raz jeszcze

Co można zrobić w konstruktorze?

- wywołać inny konstruktor
- zainicjować pola wartościami przekazanymi jako argumenty do konstruktora
- zainicjować pola wartościami domyślnymi
- wywołać inną metodę

7. Najważniejsze cechy OOP

ABSTRAKCJA



HERMETYZACJA

DZIEDZICZENIE



POLIMORFIZM

8. Dziedziczenie - pierwsza krew

Po co nam dziedziczenie?

- dla uniknięcia powtarzania tego samego kodu
- grupowanie podobnych zachowań
- ujednolicenie
- łatwość zmian

Po co nam dziedziczenie?



Po co nam dziedziczenie?

- kobieta jest człowiekiem
- mężczyzna jest człowiekiem



Po co nam dziedziczenie?

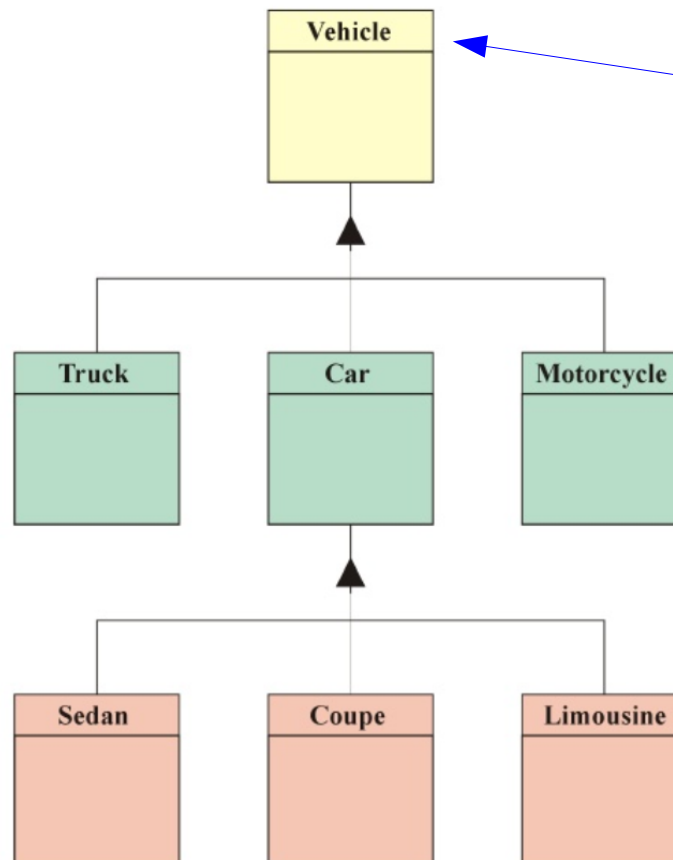
- kobieta jest człowiekiem
- mężczyzna jest człowiekiem
- człowiek jest ssakiem

Po co nam dziedziczenie?

- kobieta jest człowiekiem
- mężczyzna jest człowiekiem
- człowiek jest ssakiem

Wynika z tego, że:

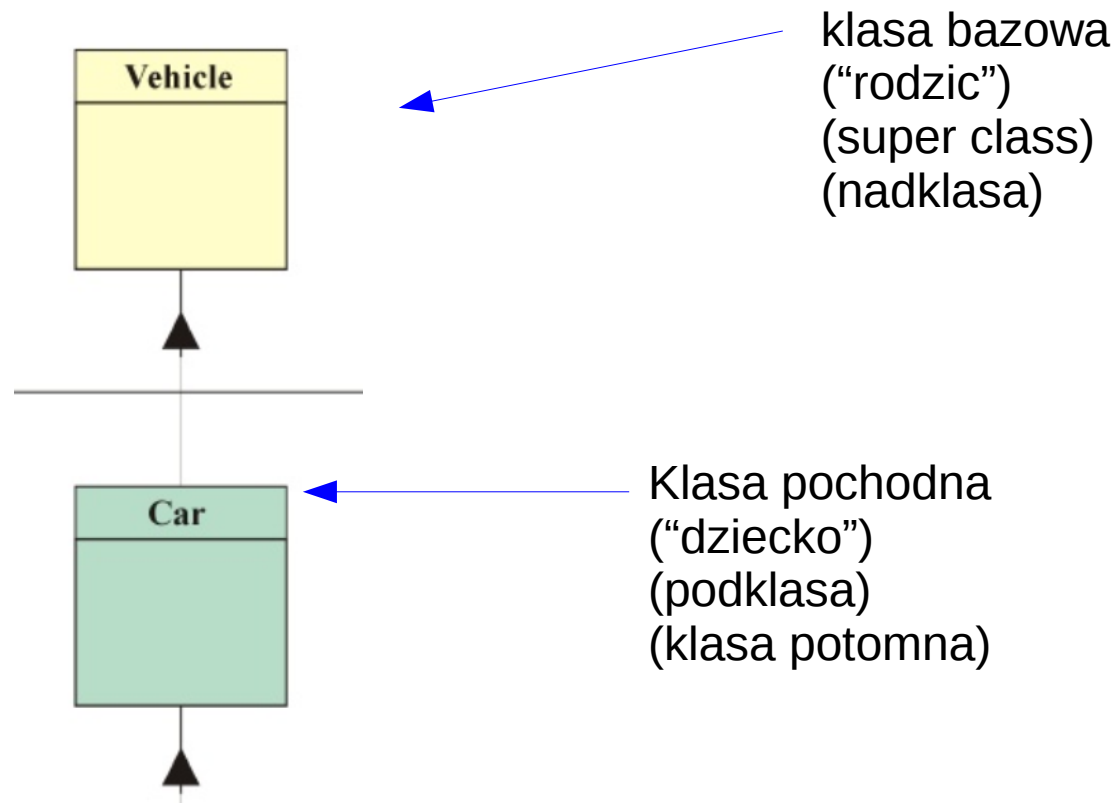
- Kobieta jest ssakiem
 - Mężczyzna jest ssakiem
-
- z ang. - relacja **IS-A** (np. man is-a mammal)



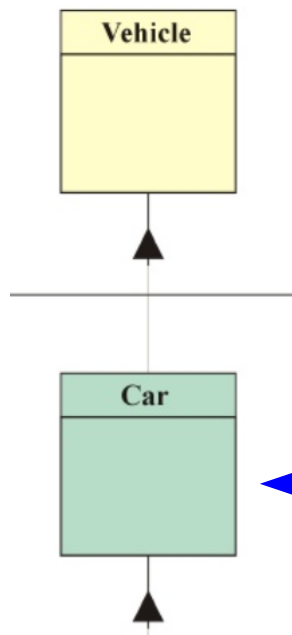
klasa bazowa(rodzic)

Car is-a vehicle
Sedan is-a car
Sedan is-a vehicle

PARENT – CHILD relationship



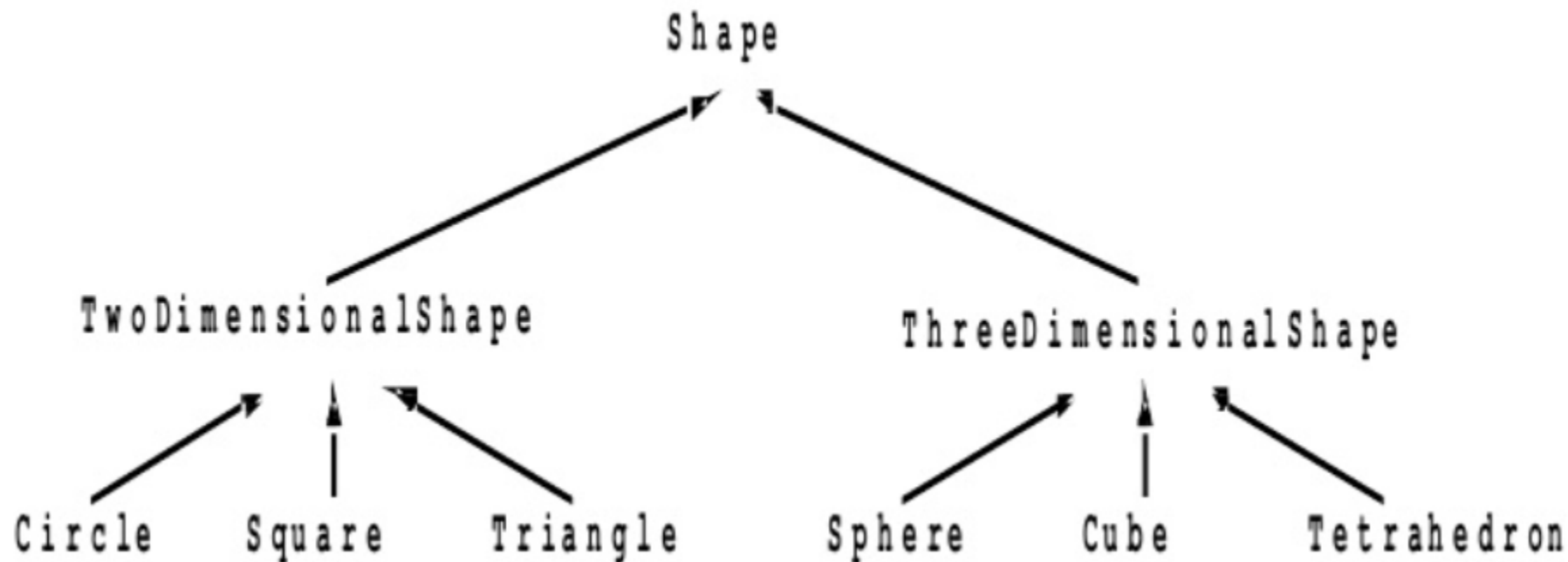
PARENT – CHILD relation



klasa bazowa (rodzic)
(super class)

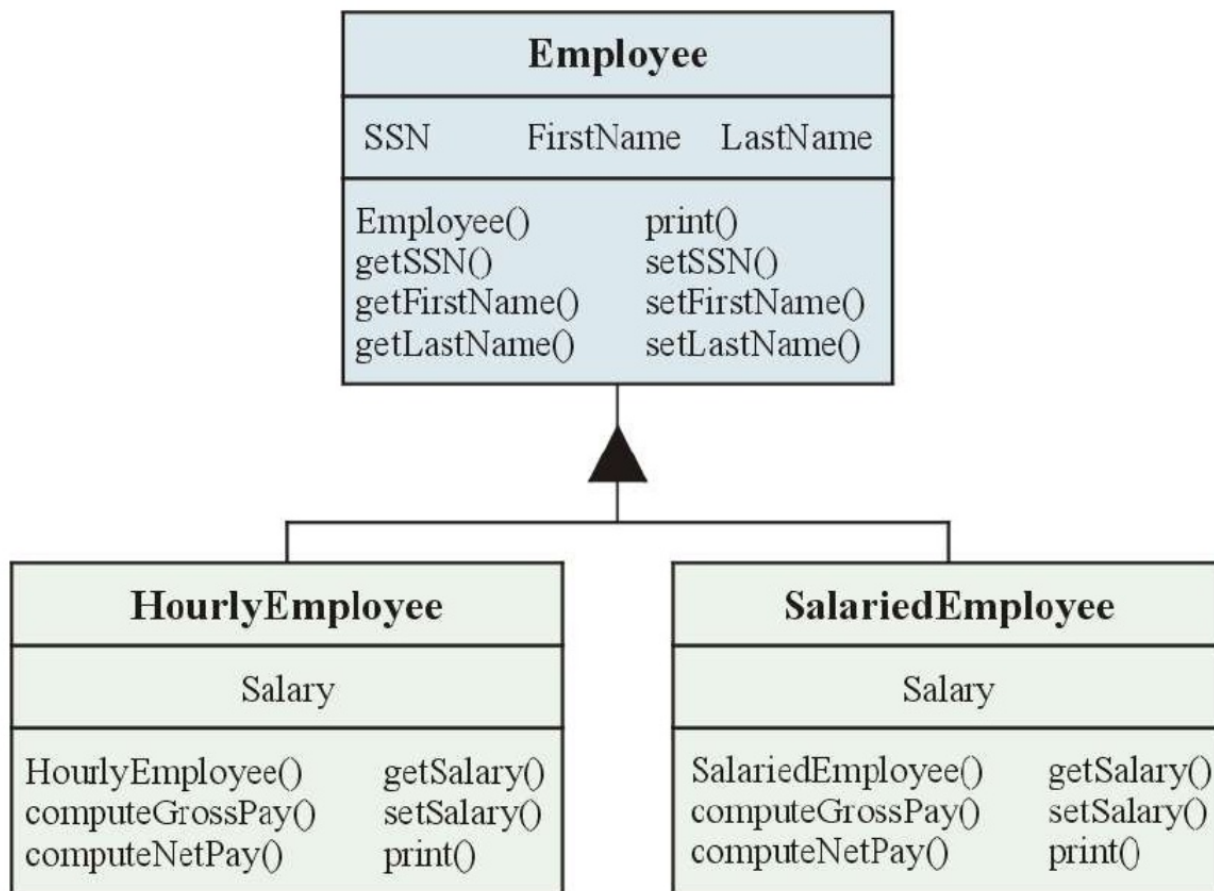
Klasa pochodna
(dziecko)

```
public class Car extends Vehicle { /* ... */ }
```



Pojazd ← Samochód ← Fiat 126p

Dziedziczenie polega na “przejęciu”
pewnych zachowań(metod) i pól.



HourlyEmployee

Salary

HourlyEmployee()
getSalary()
print()
putSalary()
computeGrossPay()
computeNetPay()

Employee

SSN	FirstName	LastName
-----	-----------	----------

Employee()	print()
getSSN()	setSSN()
getFirstName()	setFirstName()
getLastName()	setLastName()

9.

**jest super() więc
o co Ci chodzi?**

super()

Employee

HourlyEmployee

```
class HourlyEmployee extends Employee {  
    private float salary;
```

```
    public HourlyEmployee(int ssn, String firstName, String lastName, float salary){  
        super(ssn, firstName, lastName);  
        this.salary = salary;  
    }  
}
```

```
}
```

super(...) = konstruktor rodzica



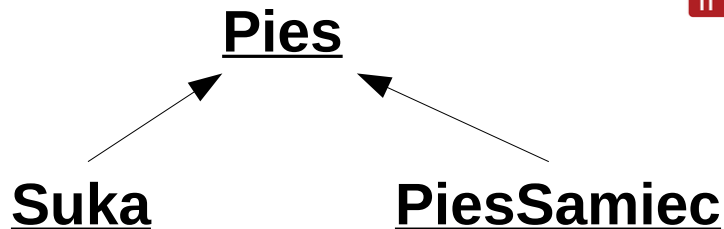
Jak nie jest `super(...)` to i tak jest `super()`



Jak nie jest `super(...)` to i tak jest `super()`

Kompilator wywoła konstruktor domyślny rodzica automatycznie, jeżeli w konstruktorze klasy pochodnej (“dziecka”) nie ma odwołania `super(...)`

Ćwiczenie 9.1 - dziedziczymy po Psie



- stwórz klasy Suka oraz PiesSamiec, które dziedziczą po napisanej wcześniej klasie Pies
- zmodyfikuj działanie metody szczekaj() dla suki – ma wyświetlać:
“suka [imie] rasy [rasa] o wadze [waga] kg, wzroście [wzrost] cm zrobiła chał hał hau”
- zainicjuj obiekt klasy Suka oraz klasy PiesSamiec z poziomu klasy MainApp i wywołaj na nich metodę szczekaj() weryfikując działanie programu.
- dodaj metodę “kupa()” wyświetlającą (***“[imie] zrobił kupkę”***) dla obu klas (uwaga – zakaz duplikowania kodu !)
- (*) dodaj super klasę Zwierze oraz klasę Kot

text = ?

```
class A {
    String text = "X";

    public A() {
        text += " chał ";
    }

}

class B extends A {
    public B() {
        text += " miał ";
    }

}
```

text = ?

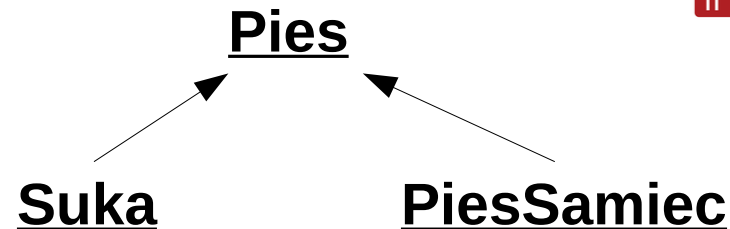
```
class Aa {
    String text = "a";

    public Aa() {
        text += "x";
    }

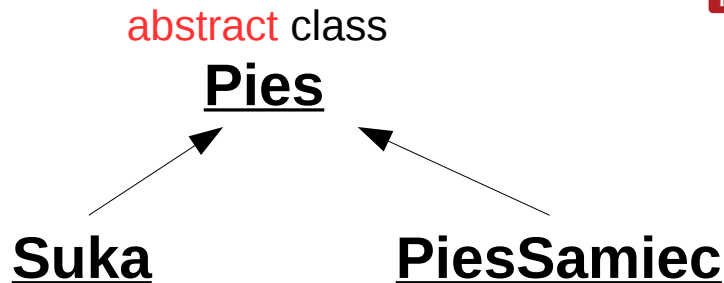
    public Aa(String s) {
        text += s;
    }
}

class Bb extends Aa {
    public Bb() {
        super("Boo");
        text += "Foo";
    }
}
```

10. **final i abstract**



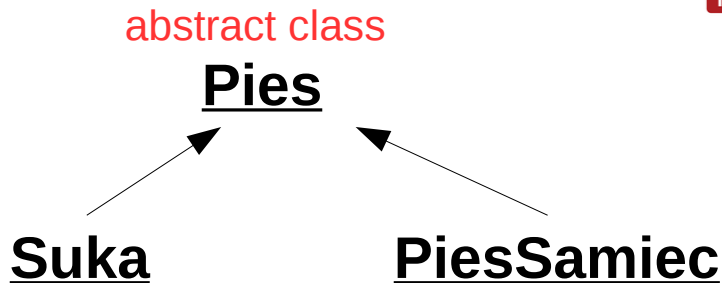
```
Pies p = new Pies("Azor", "dalmatynczyk", 30);  
Suka s1 = new Suka("Luka", "malamut", 50);  
PiesSamiec p1= new PiesSamiec("Maciej","jamnik", 25);
```



```
Pies p = new Pies("Azor", "dalmatynczyk", 30);  
Suka s1 = new Suka("Luka", "malamut", 50);  
PiesSamiec p1 = new PiesSamiec("Maciej", "jamnik", 25);
```


Ćwiczenie 10.1

- abstract/final class



- niech klasa Pies będzie abstrakcyjną
 - odpal istniejący kod
- CZĘŚĆ II
- niech klasa Pies będzie final (final class Pies {})
 - co zauważyłeś?
 - (*) usuń słówko “final” z klasy Pies, do konstruktora każdej z 3 klas dodaj wyświetlanie na konsolę “Konstruktor [nazwaKlasy]”, odpal program. W jakiej kolejności odpalane są konstruktory przy tworzeniu np. obiektu klasy Suka?

abstract **final**

- abstract class cannot be instantiated
- final class cannot be extended

11. przesłanianie metod (override)

Method override

- Baba robi siusiu()
 - Chłop robi siusiu()
-
- Prostokąt ← Kwadrat, metoda polePowierzchni()

```
class Animal {
    public void move() {
        System.out.println("Animals can move");
    }
}
class Dog extends Animal {
    @Override
    public void move() {
        System.out.println("Dogs can walk and run");
    }
}
public class TestDog {
    public static void main(String args[]) {
        Animal a = new Animal();    // Animal reference and object
        Animal b = new Dog();       // Animal reference but Dog object
        a.move();    // runs the method in Animal class
        b.move();    // runs the method in Dog class
    }
}
```

Method override

- rules

- The argument list should be exactly the same as that of the overridden method.

Method override

- rules

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.

Method override

- rules

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level. (For example: If the superclass method is declared public then the overriding method in the sub class cannot be either private or protected.)

Method override

- rules

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level. (For example: If the superclass method is declared public then the overriding method in the sub class cannot be either private or protected.)
- A method declared final cannot be overridden.

Method override

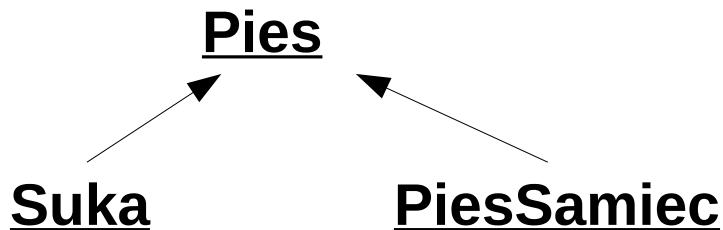
- rules

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level. (For example: If the superclass method is declared public then the overriding method in the sub class cannot be either private or protected.)
- A method declared final cannot be overridden.
- Good practice – always use @Override annotation

12. polimorfizm

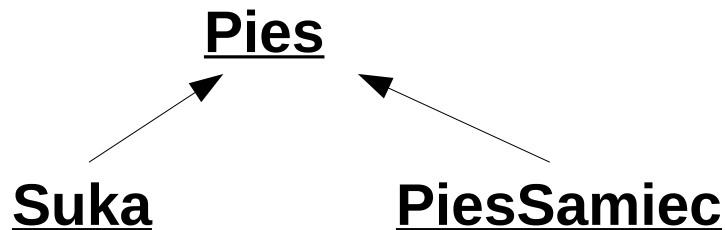
Ćwiczenie 12.1

- polimorizm



- stwórz klasę Wataha
- dodaj pole *czlonkowie* - przechowujące listę obiektów klasy Pies wchodzących w skład watahy
- wykorzystaj konstruktor domyślny
- zaimplementuj metodę dodającą nowego psa do watahy
- z poziomu klasy stwórz obiekty: wataha oraz pies, suka, samiec i dodaj psy do watahy
- w klasie Wataha zaimplementuj metodę “ktoWStadzie()” listującą skład stada poprzez zwrócenie Stringa z imionami psów (po przecinku)
- (*) dodaj konstruktor z jednym parametrem – (Pies osobnikAlfa), przechowuj go w odrębnym polu.
- (*)niech metoda ktoWStadzie() zwraca psy w kolejności alfabetycznej (wg rasy)

Polimorizm



```
public class Wataha {  
    [...]  
    public void add(Pies p) {...}  
    [...]  
}
```

```
wataha.add(pies);  
wataha.add(suka);  
wataha.add(samiec);
```

```
public abstract class FiguraPłaska
{
    protected String atrNazwa;
    FiguraPłaska()
    {
        this.atrNazwa = "Nieznana";
    }
    public double obwód()
    {
        return 0.0;
    }
}
```

```
public class Koło extends FiguraPłaska
{
    public double obwód()
    {
        return 12.8;
    }
}
public class Kwadrat extends FiguraPłaska
{
    public double obwód()
    {
        return 4.9;
    }
}
```

```
public class Wyświetl
{
    public static void wyświetlInf(FiguraPłaska parFigura)
    {
        System.out.println(parFigura.atrNazwa);
        System.out.println(parFigura.obwód());
    }
}

public class Program
{
    public static void main (String args[])
    {
        Koło koło = new Koło();
        Kwadrat kwadrat = new Kwadrat();
        Wyświetl.wyświetlInf(koło);
        Wyświetl.wyświetlInf(kwadrat);
    }
}
```

13. Modyfikatory dostępu

modyfikatory dostępu

```
[modyfikator] typPole nazwaPola;  
[modyfikator] class nazwaKlasy;
```

modyfikator	przykład	dotyczy
public	public class MyClass {...} public String lastName;	klas, metod, pól
protected	protected String getName(){ ...}	metod, pól
empty (package) (default)	class Beverage { ... } String name;	klas, metod, pól
private	private String imie;	metod, pól

modyfikatory dostępu

(w kontekście metod i pól)

modyfikator	klasa	pakiet	podklasa	poza pakietem	opis
public	✓	✓	✓	✓	dostęp dla wszystkich
protected	✓	✓	✓	✗	jak wyżej, za wyjątkiem klas z innych pakietów
brak (=package)	✓	✓	✗	✗	możliwe użycie tylko w tym samym pakiecie
private	✓	✗	✗	✗	dostęp jedynie w ramach własnej klasy (metoda do użytku wewnętrznego)

Ćwiczenie 13.1

- modyfikatory dostępu

- otwórz projekt “accessModifiers”
- przeanalizuj klasy PublicClass, DefaultClass
- następnie zerknij na TestInsidePackageAccess oraz TestOutsidePackageAccess
- przeanalizuj również TestInsidePackageInheritance, TestOutsidePackageInheritance,

14.

Ukryjmy się- Enkapsulacja (hermetyzacja)



enkapsulacja

```
public class Person {  
    public String name;  
}
```

dostęp do pola "name" przez **operator kropki**. (umożliwia również edycję zawartości !!)

| Ćwiczenie 14.1

- enkapsulacja

- otwórz projekt “enkapsulacja”, klasa: Circle.java
- postępuj zgodnie z instrukcją “todo. 1”

enkapsulacja

```
class Person {  
    private String name;  
    public String getName() {  
        return name;  
    }  
}
```

Ćwiczenie 14.1

- enkapsulacja

- otwórz projekt “enkapsulacja”, klasa: Circle.java
- postępuj zgodnie z instrukcją “todo. 1”
- postępuj zgodnie z instrukcją “todo. 2”
-
- (*) wróć do projektu z psami. We wszystkich klasach (np. PiesSamiec, Suka itp.) dokonaj ukrycia dostępu do pól poprzez zastosowanie getterów.

Po co enkapsulacja?

- zabezpieczenie przed modyfikacjami
- zabezpiecza utratę synchronizacji danych
- programy są bezpieczniejsze

Po co enkapsulacja?

- można dodać logikę do gettera/settera
- obiekt pilnuje swoich zmiennych

Prawidłowa enkapsulacja gwarantuje, że jedynym obiektem odpowiedzialnym za zmianę stanu jest nasz obiekt i nie ma możliwości nieuprawnionej modyfikacji tego stanu z zewnątrz.

Po co enkapsulacja? (wikipedia)

Głównym zadaniem wyodrębnienia interfejsu, a tym samym enkapsulacji, jest ukrycie przed użytkownikiem sposobu w jaki klasa wewnętrznie realizuje swoje zadanie. Metody i pola znajdujące się w sekcji publicznej stanowią interfejs, tj. jedyny dopuszczalny zbiór elementów klasy, którymi inne klasy mogą oddziaływać z daną klasą. Posiadanie wyodrębnionego interfejsu powoduje, że użytkownik danej klasy ma pewność, że korzystając z tych metod jest bezpieczny, tj. nie dojdzie do sytuacji w której klasa zostanie uszkodzona.

Po co enkapsulacja? (wikipedia)

Głównym zadaniem wyodrębnienia interfejsu, a tym samym enkapsulacji, jest ukrycie przed użytkownikiem sposobu w jaki klasa wewnętrznie realizuje swoje zadanie. Metody i pola znajdujące się w sekcji publicznej stanowią interfejs, tj. jedyny dopuszczalny zbiór elementów klasy, którymi inne klasy mogą oddziaływać z daną klasą. Posiadanie wyodrębnionego interfejsu powoduje, że użytkownik danej klasy ma pewność, że korzystając z tych metod jest bezpieczny, tj. nie dojdzie do sytuacji w której klasa zostanie uszkodzona.

Metody interfejsowe, tj. ujawnione użytkownikowi klasy, w założeniu są absolutnie bezpieczne i korzystając tylko z nich nie można doprowadzić do nieprawidłowego stanu klasy.

| Enkapsulacja - strategie

1.
wszystkie pola private,
gettery,
brak setterów,
obiekt inicjujemy poprzez konstruktor i potem nie ma potrzeby go zmieniać

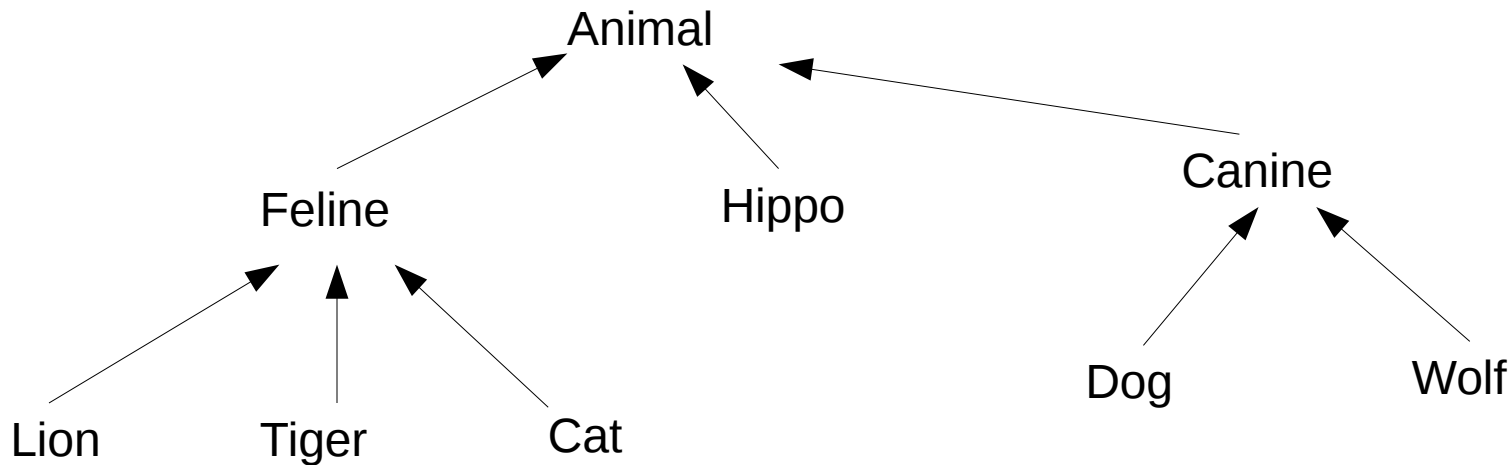
| Enkapsulacja - strategie

1.
wszystkie pola private,
gettery,
brak setterów,
obiekt inicjujemy poprzez konstruktor i potem nie ma potrzeby go zmieniać

2.
wszystkie pola private,
gettery,
settery,
settery pilnują, żeby stan obiektu był pod kontrolą, nie wykroczył poza warunki brzegowe itp.

15. abstract vs concrete

Interfejs



Jak wygląda obiekt `new Animal()` ??

**What does a new `Animal()` object
look like?**



scary objects



```
abstract public class Canine extends Animal  
{  
    public void roam() { }  
}
```

```
public class MakeCanine {
```

```
    public void go() {
```

```
        Canine c;
```

```
        c = new Dog();
```

```
        c = new Canine();
```

```
        c.roam();
```

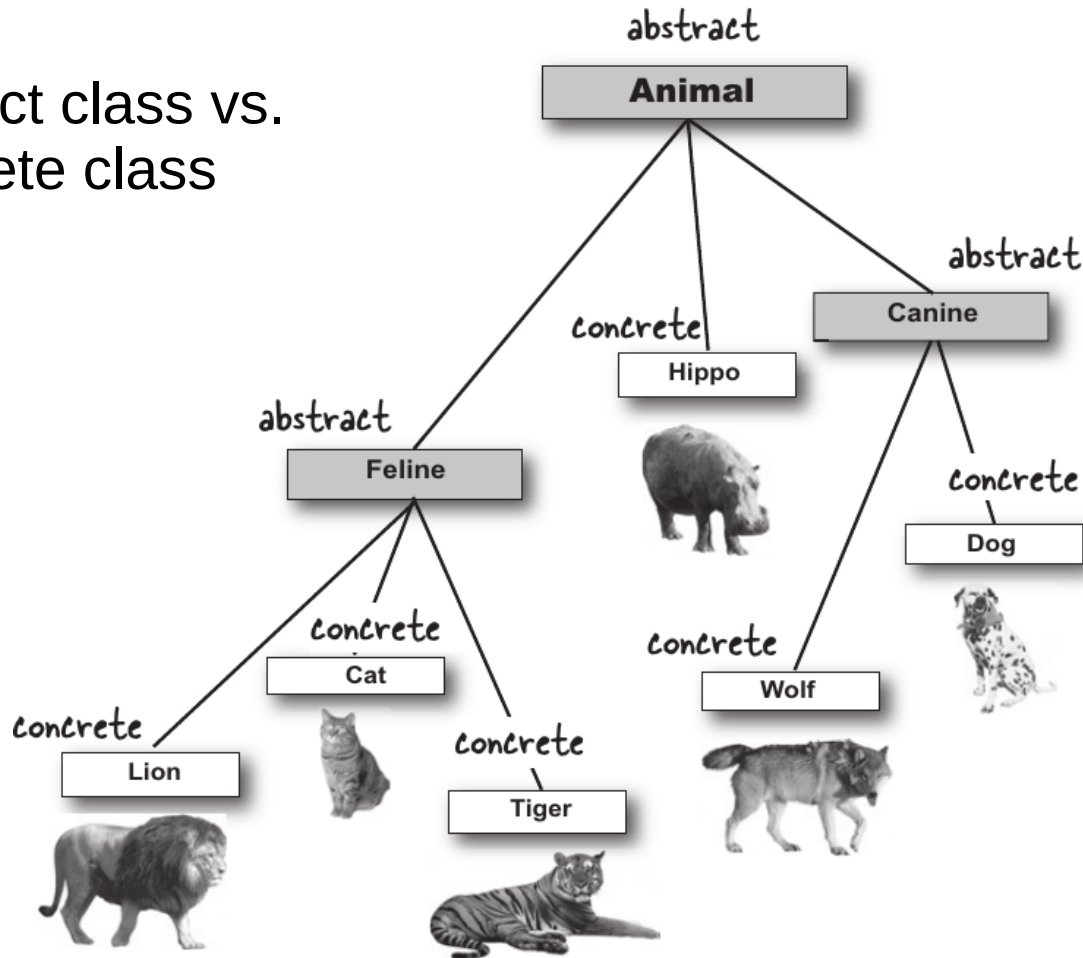
```
    }
```

```
}
```

← This is OK, because you can always assign a subclass object to a superclass reference, even if the superclass is abstract.

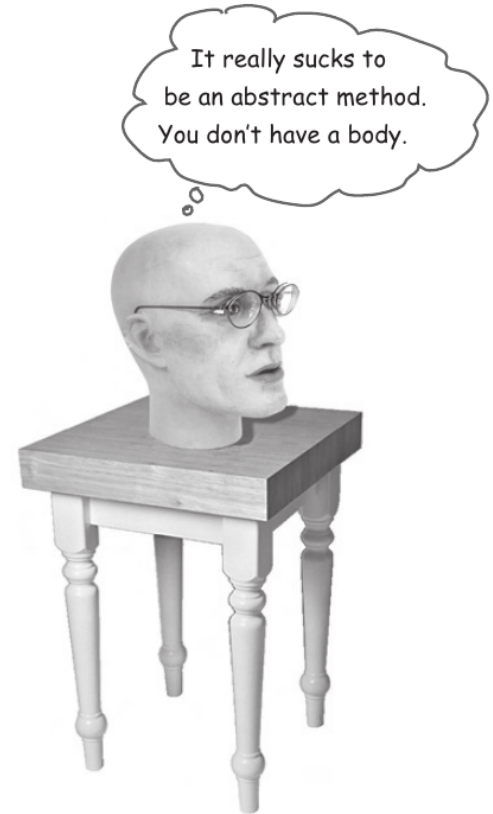
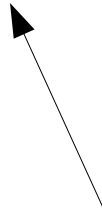
← new Canine() class Canine is marked abstract, so the compiler will NOT let you do this.

Abstract class vs. Concrete class



Abstract methods

```
public abstract void someMeth();
```



Abstract methods

```
public abstract void someMeth();
```

If you declare an abstract method, you **MUST** mark the class abstract as well. You can't have an abstract method in a non-abstract class.



```
abstract class MyAbstractClass
{
    public abstract void showMe();
}
```

```
class MyConcreteClass extends MyAbstractClass
{
    @Override
    public void showMe()
    {
        System.out.println("I am from concrete class:");
        System.out.println("I am supplying the method body for showMe()");
    }
}
```

Implementing an abstract method
is just like overriding a method

Abstract methods

- don't have body, exists for polymorphizm
- first concrete class in inheritance tree must implement all abstract methods

16. interfejs

= 100% abstract class

Interfejs

```
public interface Printable{  
    public void print();  
}
```

Interfejs

Ograniczenia dziedziczenia w javie:

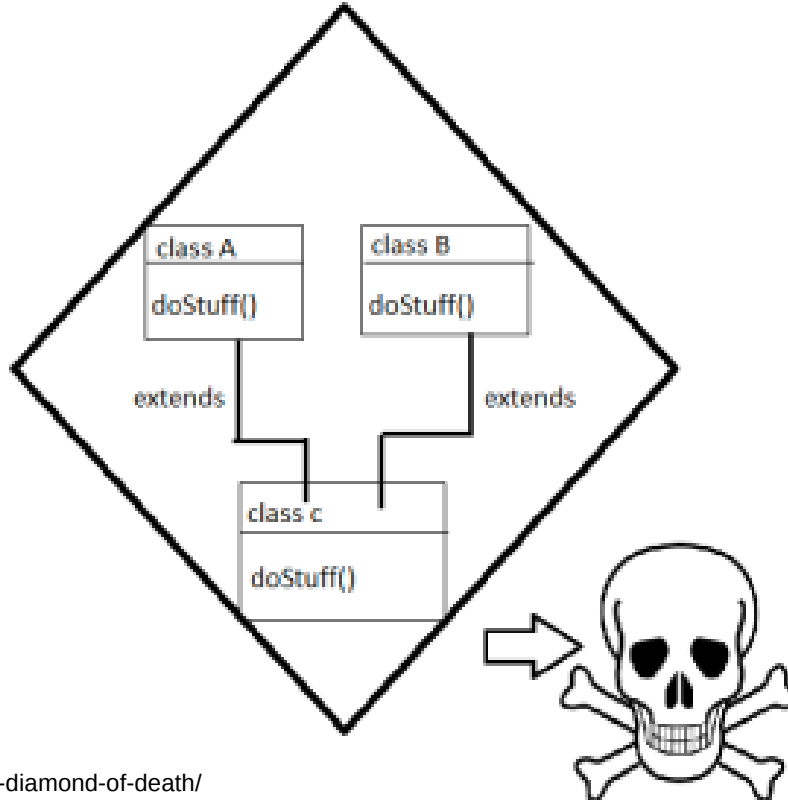
- **można dziedziczyć tylko po jednej klasie !!**

public class MyClass extends A, B - zabronione

public class MyClass implements A, B - dozwolone

Interfejsy umożliwiają szersze użycie polimorfizmu w Javie.

"diamond problem"



Interfejs

```
public interface HelloInterface {  
    public void sayHello();  
}
```

```
public class HelloInterfaceImpl implements HelloInterface {  
    @Override  
    public void sayHello() {  
        System.out.println("Hello");  
    }  
}
```

→ @Override -zalecane

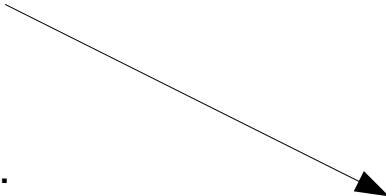
implementacja

```
public class MyInterfaceImpl  
    implements HelloInterface, GoodByeInterface {
```

```
    @Override  
    public void sayHello() {  
        System.out.println("Hello");  
    }
```

```
    @Override  
    public void sayGoodbye() {  
        System.out.println("Goodbye");  
    }
```

```
}
```



po przecinku kolejne
interfejsy

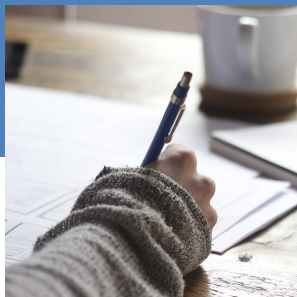
| Ćwiczenie 16.1

- interfejsy

- otwórz projekt “interface”
- istniejące klasy: Circle, Rectangle, Square
- istniejące interfejsy: Printable, Movable
- zaimplementuj oba interfejsy dla wszystkich 3 klas
- przetestuj

Dzięki

Za uwagę !!



Literatura

Head First Java; Kathy Sierra, Bert Bates

https://www.ntu.edu.sg/home/ehchua/programming/cpp/cp3_OOP.html

https://pl.wikipedia.org/wiki/Programowanie_obiektowe

Interactive Object Oriented Programming in Java; Vaskaran Sarcar

<http://slideplayer.pl/slide/2267818/>

<https://www.slideshare.net/tareq1988/java-inheritance-2397559>