



Wprowadzenie do programowania w Javie

Autor: *Piotr Dubiela*



Programowanie obiektowe



Co to jest programowanie obiektowe?

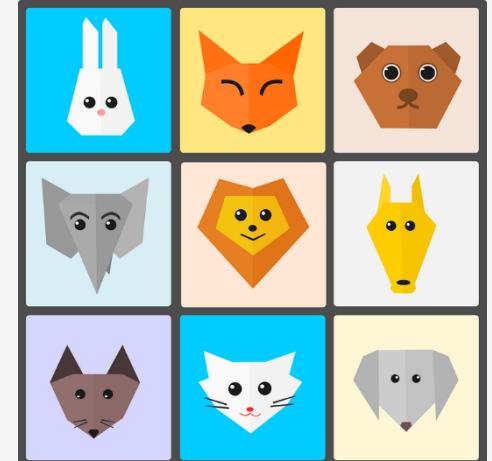
Programowanie obiektowe – paradymat programowania, w którym programy definiujemy za pomocą **obiektów** – elementów łączących **stan** (czyli dane, nazywane najczęściej *polami*) i **zachowanie** (czyli metody). Obiektowy program komputerowy jest przedstawiony jako zbiór takich obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań.



Programowanie obiektowe

Co to jest obiekt?

- Obiekt jest to instancja dowolnej klasy
- Posiada trzy cechy :
 - tożsamość – coś co odróżnia go na tle innych obiektów
 - stan – aktualny stan w jakim się znajduje
 - zachowanie – zestaw metod wykonujących operacje na danych, które posiada





Programowanie obiektowe

Co to jest obiekt?

- Obiekt jest to instancja dowolnej klasy
- Posiada trzy cechy :
 - tożsamość – coś co odróżnia go na tle innych obiektów
 - stan – aktualny stan w jakim się znajduje
 - zachowanie – zestaw metod wykonujących operacje na danych, które posiada





Programowanie obiektowe

Co to jest obiekt?

- Obiekt jest to instancja dowolnej klasy
- Posiada trzy cechy :
 - tożsamość – coś co odróżnia go na tle innych obiektów
 - stan – aktualny stan w jakim się znajduje
 - zachowanie – zestaw metod wykonujących operacje na danych, które posiada





Programowanie obiektowe

Co to jest obiekt?

- Obiekt jest to instancja dowolnej klasy
- Posiada trzy cechy :
 - tożsamość – coś co odróżnia go na tle innych obiektów
 - stan – aktualny stan w jakim się znajduje
 - zachowanie – zestaw metod wykonujących operacje na danych, które posiada

Pies burek





Programowanie obiektowe

Co to jest obiekt?

- Obiekt jest to instancja dowolnej klasy
- Posiada trzy cechy :
 - tożsamość – coś co odróżnia go na tle innych obiektów
 - stan – aktualny stan w jakim się znajduje
 - zachowanie – zestaw metod wykonujących operacje na danych, które posiada

Pies burek

- imię: „Burek”
- wiek: „2 lata”
- kolor: „brązowy”





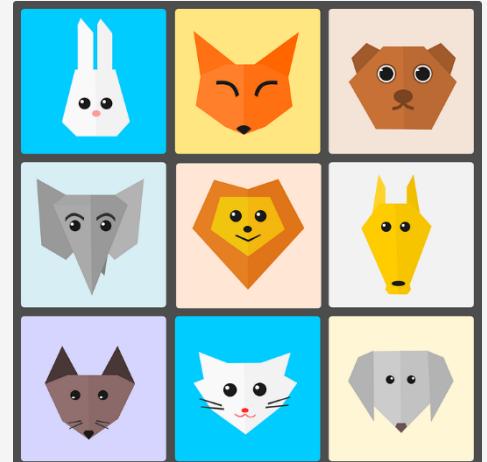
Programowanie obiektowe

Co to jest obiekt?

- Obiekt jest to instancja dowolnej klasy
- Posiada trzy cechy :
 - tożsamość – coś co odróżnia go na tle innych obiektów
 - stan – aktualny stan w jakim się znajduje
 - zachowanie – zestaw metod wykonujących operacje na danych, które posiada

Pies burek

- imię: „Burek”
- wiek: „2 lata”
- kolor: „brązowy”





Programowanie obiektowe

Tworzenie kolejnych obiektów

- W programowaniu obiektowym, deklarujemy cechy pozwalające nam na nadawanie cech rozpoznawczych obiektów danej klasy





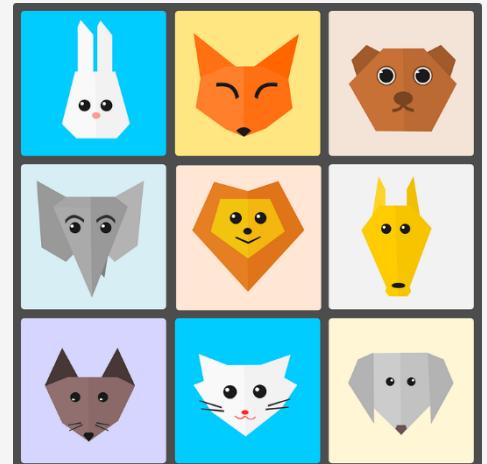
Programowanie obiektowe

Tworzenie kolejnych obiektów

- W programowaniu obiektowym, deklarujemy cechy pozwalające nam na nadawanie cech rozpoznawczych obiektów danej klasy

Pies

- imię
- rok urodzenia
- kolor umaszczenia





Programowanie obiektowe

Tworzenie kolejnych obiektów

- W programowaniu obiektowym, deklarujemy cechy pozwalające nam na nadawanie cech rozpoznawczych obiektów danej klasy

Pies

- imię
- rok urodzenia
- kolor umaszczenia



burek

- „Burek”
- „2015-01-05”
- „brązowy”





Programowanie obiektowe

Tworzenie kolejnych obiektów

- W programowaniu obiektowym, deklarujemy cechy pozwalające nam na nadawanie cech rozpoznawczych obiektów danej klasy

Pies

- imię
- rok urodzenia
- kolor umaszczenia



burek

- „Burek”
- „2015-01-05”
- „brązowy”



corgi

- „Mietek”
- „2016-11-18”
- „rudawy”





Programowanie obiektowe

Tworzenie kolejnych obiektów

- W programowaniu obiektowym, deklarujemy cechy pozwalające nam na nadawanie cech rozpoznawczych obiektów danej klasy

Pies

- imię
- rok urodzenia
- kolor umaszczenia
- daj głos
- podaj łapę
- aportuj
- turlaj się



burek

- „Burek”
- „2015-01-05”
- „brązowy”



corgi

- „Mietek”
- „2016-11-18”
- „rudawy”





Programowanie obiektowe

Tworzenie kolejnych obiektów

- W programowaniu obiektowym, deklarujemy cechy pozwalające nam na nadawanie cech rozpoznawczych obiektów danej klasy

Pies

- imię
- rok urodzenia
- kolor umaszczenia
- daj głos
- podaj łapę
- aportuj
- turlaj się



burek

- „Burek”
- „2015-01-05”
- „brązowy”



corgi

- „Mietek”
- „2016-11-18”
- „rudawy”





Tworzenie obiektów – zadanie

1. *Utworzenie nowego pakietu*
2. *Utworzenie klasy Osoba*
3. *Utworzenie pól :*
 - *imie:String*
 - *rok urodzenia:int*
4. *Utworzenie metody:*
 - przedstawSie():void*
5. *Implementacja metody*
6. *Utworzenie 3 obiektów:*
 1. *Ania lat 25*
 2. *Andrzej lat 54*
 3. *Mariola lat 68*
7. *Uruchomienie metody przedstawSie na każdym z obiektów*

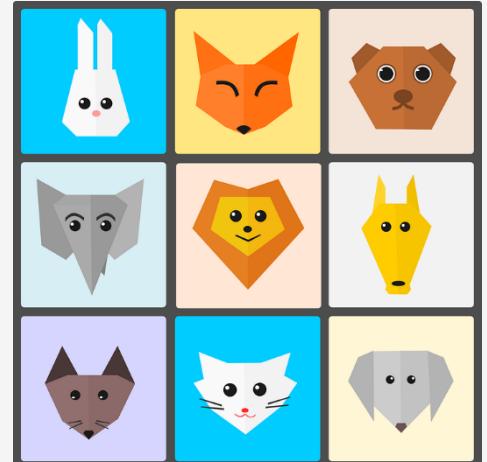




Programowanie obiektowe

Wszystko jest obiektem

- Programowanie obiektowe zakłada, że cały otaczający nas świat można przedstawić w postaci obiektów, które będą reprezentować uproszczony model świata rzeczywistego





Programowanie obiektowe

Wszystko jest obiektem

- Programowanie obiektowe zakłada, że cały otaczający nas świat można przedstawić w postaci obiektów, które będą reprezentować uproszczony model świata rzeczywistego





Programowanie obiektowe

Wszystko jest obiektem

- Programowanie obiektowe zakłada, że cały otaczający nas świat można przedstawić w postaci obiektów, które będą reprezentować uproszczony model świata rzeczywistego

Sala

- ilość m²
- liczba krzeseł
- liczba biurek
- liczba lamp
- liczba okien
- czyJestKlimatyzacja
- czyJestRzutnik
- ...

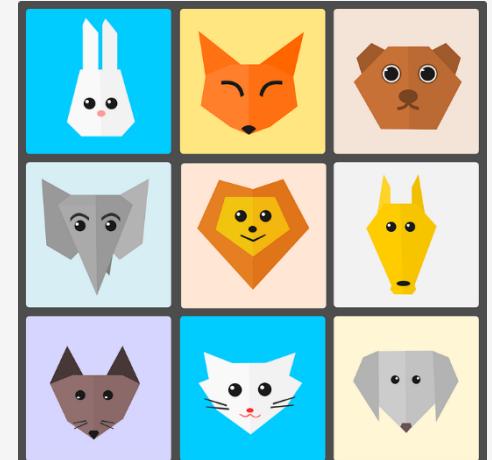




Programowanie obiektowe

Jak bardzo szczegółowy powinien być nasz model?

- Celem tworzenia modelu obiektowego jest rozwiązywanie jakiegoś problemu w świecie rzeczywistym
- Model powinien odzwierciedlać to co jest istotne dla ‚biznesu’ i nic więcej





Programowanie obiektowe

Przykład

Jan zajmuje się wynajmem sali pod różnego rodzaju szkolenia i konferencje. Najemcy pomieszczeń zwracają uwagę na wielkość sali, możliwość wyświetlania obrazu na ekranie oraz liczebność miejsc dla uczestników.

Jan jest zainteresowany programem, który pozwoli mu lepiej zarządzać salami w budynku.





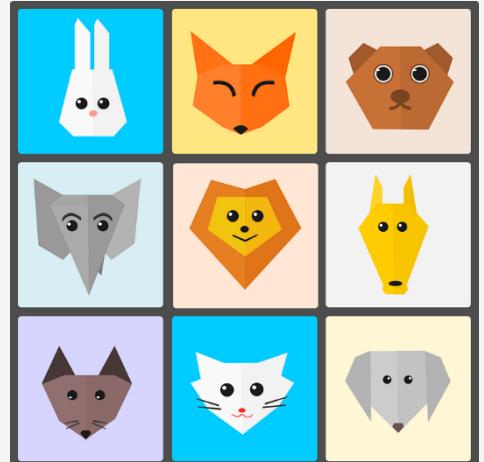
Przykład

Jan zajmuje się wynajmem sali pod różnego rodzaju szkolenia i konferencje. Najemcy pomieszczeń zwracają uwagę na wielkość sali, możliwość wyświetlania obrazu na ekranie oraz liczebność miejsc dla uczestników.

Jan jest zainteresowany programem, który pozwoli mu lepiej zarządzać salami w budynku.

Sala

- nazwa(identyfikator)
- ilość m²
- liczba stanowisk
- czyJestRzutnik
- czyJestWolna





Przykład

Jan zajmuje się wynajmem sali pod różnego rodzaju szkolenia i konferencje. Najemcy pomieszczeń zwracają uwagę na wielkość sali, możliwość wyświetlania obrazu na ekranie oraz liczebność miejsc dla uczestników.

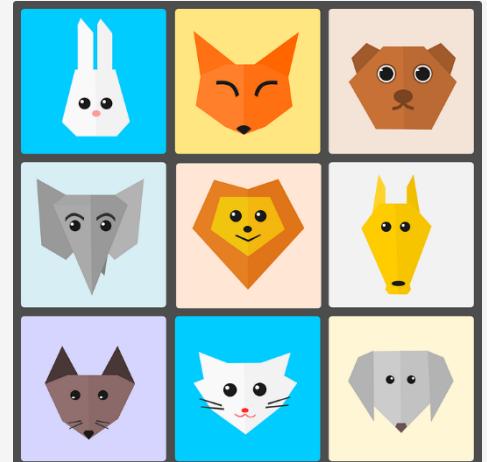
Jan jest zainteresowany programem, który pozwoli mu lepiej zarządzać salami w budynku.

Sala

- nazwa(identyfikator)
- ilość m²
- liczba stanowisk
- czyJestRzutnik
- czyJestWolna

Menadżer

- imię
- zarządzaneSale





Przykład

Jan zajmuje się wynajmem sali pod różnego rodzaju szkolenia i konferencje. Najemcy pomieszczeń zwracają uwagę na wielkość sali, możliwość wyświetlania obrazu na ekranie oraz liczebność miejsc dla uczestników.

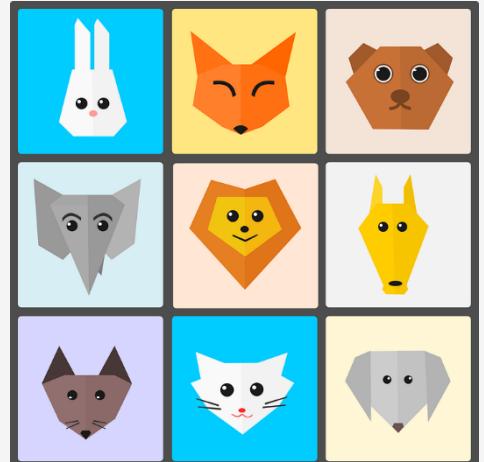
Jan jest zainteresowany programem, który pozwoli mu lepiej zarządzać salami w budynku.

Sala

- nazwa(identyfikator)
- ilość m2
- liczba stanowisk
- czyJestRzutnik
- czyJestWolna

Menadżer

- imię
- zarządzaneSale
- wyświetlDostepneSale
- zarezerwujSale(nazwaSali)





Przykład

Jan zajmuje się wynajmem sali pod różnego rodzaju szkolenia i konferencje. Najemcy pomieszczeń zwracają uwagę na wielkość sali, możliwość wyświetlania obrazu na ekranie oraz liczebność miejsc dla uczestników.

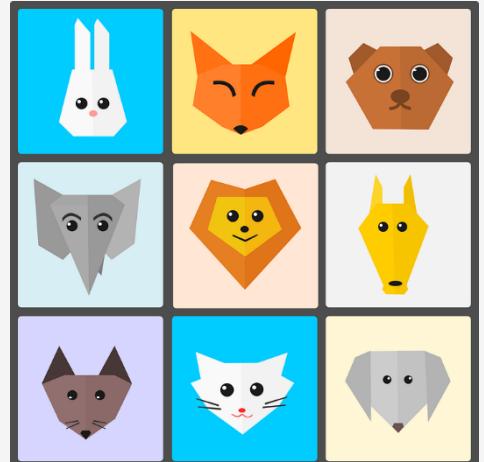
Jan jest zainteresowany programem, który pozwoli mu lepiej zarządzać salami w budynku.

Sala

- nazwa(identyfikator)
- ilość m2
- liczba stanowisk
- czyJestRzutnik
- czyJestWolna
- wyświetlOpisSali()

Menadżer

- imię
- zarządzaneSale
- wyświetlDostepneSale()
- zarezerwujSale(nazwaSali)

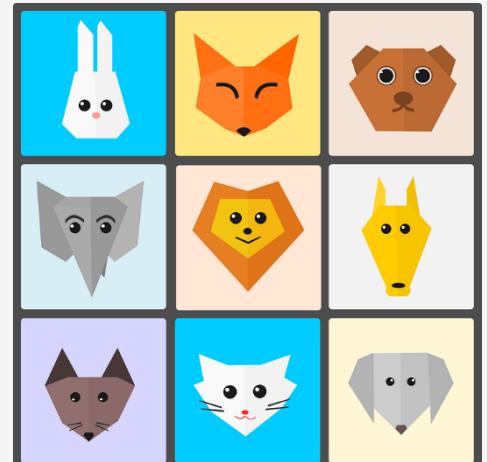




Programowanie obiektowe

Tworzenie obiektów

- Każdorazowe uzupełnianie pól klasy przy inicjalizacji obiektów jest żmudne i łamie zasadę *DRY(Don't repeat yourself)*
- Dlatego Java definiuje specjalną metodę, która służy do tworzenia nowych instancji klasy
- Metodę tę nazywamy konstruktorem klasy





Konstruktor

- Specjalna metoda do tworzenia nowych obiektów danej klasy
- Nie posiada typu zwracanego
- Podobnie jak każda metoda w Javie może przyjmować od 1 do n argumentów różnego typu
- W przypadku niezadeklarowania jawnie metody konstruktora, Java korzysta z konstruktora domyślnego (bezargumentowego)





Programowanie obiektowe

Konstruktor

```
public class Osoba {  
    public String imie;  
    public int rokUrodzenia;  
  
    public Osoba(String imie, int rokUrodzenia) {  
        this.imie = imie;  
        this.rokUrodzenia = rokUrodzenia;  
    }  
}
```





Programowanie obiektowe

Konstruktor

```
public class Osoba {  
    public String imie;  
    public int rokUrodzenia;  
  
    public Osoba(String imie, int rokUrodzenia) {  
        this.imie = imie;  
        this.rokUrodzenia = rokUrodzenia;  
    }  
}
```

Pola klasy



Programowanie obiektowe



Konstruktor

```
public class Osoba {  
    public String imie;  
    public int rokUrodzenia;  
  
    public Osoba(String imie, int rokUrodzenia) {  
        this.imie = imie;  
        this.rokUrodzenia = rokUrodzenia;  
    }  
}
```

Brak typu
zwracanego/nazwy metody





Programowanie obiektowe

Konstruktor

Przypisanie wartościom pól wartości przekazanych w parametrach konstruktora:

```
public class Osoba {  
    public String imie;  
    public int rokUrodzenia;  
  
    public Osoba(String imie, int rokUrodzenia) {  
        this.imie = imie;  
        this.rokUrodzenia = rokUrodzenia;  
    }  
}
```





Programowanie obiektowe

Konstruktor

Przypisanie wartościom pól wartości przekazanych w parametrach konstruktora:

```
public class Osoba {  
    public String imie;  
    public int rokUrodzenia;  
  
    public Osoba(String imie, int rokUrodzenia) {  
        this.imie = imie;  
        this.rokUrodzenia = rokUrodzenia;  
    }  
}
```





Programowanie obiektowe

Konstruktor

Przypisanie wartościom pól wartości przekazanych w parametrach konstruktora:

```
public class Osoba {  
    public String imie;  
    public int rokUrodzenia;  
  
    public Osoba(String imie, int rokUrodzenia) {  
        this.imie = imie;  
        this.rokUrodzenia = rokUrodzenia;  
    }  
}
```





Użycie konstruktora – zadanie

1. Aktualizacja klasy *Osoba*
2. Dodanie konstruktora
3. Utworzenie dodatkowych 3 obiektów typu *osoba* za pomocą konstruktora
4. Umieszczenie wszystkich obiektów *Osoba* do tablicy
5. Wyświetlenie wszystkich osób z tablicy w pętli
6. * Wyświetlenie tylko pań
7. * Wyświetlenie tylko panów





Zadanie kolejne

1. Utwórz klasę *KontoBankowe*
2. Nadaj pola publiczne :
 - *numerKonta : long*
 - *stanKonta: int*
3. Utwórz metody:
 - *wyswietlStanKonta():void*
 - *wplacSrodki(int):void*
 - *pobierzSrodki(int):int*
4. Utwórz 2 obiekty:
 1. *kontoAndrzej(123L, 1000)*
 2. *kontoBeaty(555L, 2000)*
 3. Przetestuj przesył pieniędzy pomiędzy kontami





TEST WIEDZY



Powtórzenie wiedzy z poprzednich zajęć

Wykonaj poniższe zadania:

1. Utwórz nowy projekt o nazwie ‚Wyscig’
2. Utwórz klasę o nazwie Zawodnik
3. Zawodnik powinien posiadać takie pola jak:
 1. Imię
 2. Identyfikator (numer startowy)
 3. Predkosc minimalna
 4. Predkosc maksymalna
 5. Pokonana odległość
4. Oraz metody:
 1. przedstawSie():void // wyświetla dane o zawodniku np. „Nazywam się Robert, mam numer 4#, biegam z predkoscia od 10km/h do 20 km/h”
 2. biegij():void – pokonuje odległość w Random(min, max)* lub max+min/2
5. W metodzie psvm utwórz 3 zawodników
6. Przeprowadź symulację zawodów - wywołuj metodę biegij() na każdym z zawodników dopóki nie wyłonisz zwycięzcy (przebiegnięcie 50 km)

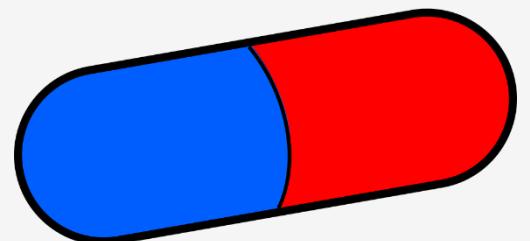




Enkapsulacja

Enkapsulacja, inaczej *Hermetyzacja*

- Jedno z podstawowych założeń programowania obiektowego
- Polega na ukrywaniu pól oraz wybranych metod przed niezamierzonym użyciem
- Dzięki takiemu podejściu klasa staje się mniej zależna od innych klas
- Metody i ukryte pola są wykorzystywane w celu realizacji działań metod widocznych na zewnątrz klasy np. metod publicznych

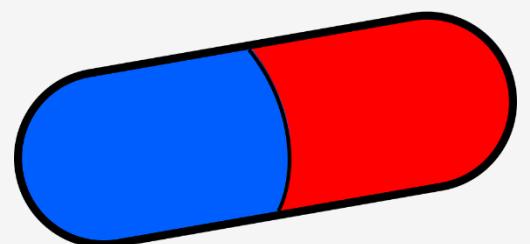




Enkapsulacja

Jak osiągnąć hermetyzację klasy?

- Ukrycie pól zmiennych przez użycie odpowiedniego akcesora jak np. ‚private’
- Wystawienie metod zwracających wartość pól w postaci tzw. ‚getterów’
- Oraz utworzenie metod nadających wartości polom, jeśli to konieczne w formie tzw. ‚setterów’



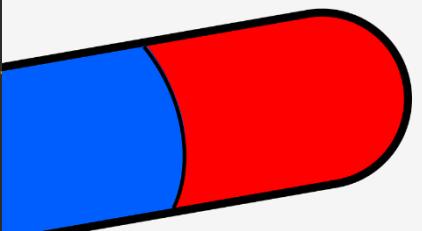


Enkapsulacja

Przykład:

```
public class Student {  
    private long indeks;  
    private String imie;  
    private String nazwisko;  
    private double ocenaZStudiow;  
  
    public double getOcenaZStudiow() {  
        return ocenaZStudiow;  
    }  
  
    public void setOcenaZStudiow(double ocenaZStudiow) {  
        this.ocenaZStudiow = ocenaZStudiow;  
    }  
  
    public Student(long indeks, String imie, String nazwisko) {  
        this.indeks = indeks;  
        this.imie = imie;  
        this.nazwisko = nazwisko;  
    }  
  
    public void wypiszDaneOStudencie(){  
        System.out.printf("Student %s %s o indeksie %s ma ocene %.2f", indeks, imie, nazwisko, ocenaZStudiow);  
    }  
}
```

Zmienna, której wartość nie jest nadawana w czasie tworzenia obiektu



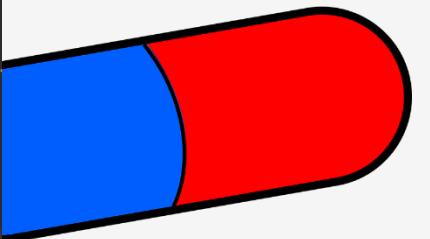


Enkapsulacja

Przykład:

```
public class Student {  
    private long indeks;  
    private String imie;  
    private String nazwisko;  
    private double ocenaZStudiow;  
  
    public double getOcenaZStudiow() {  
        return ocenaZStudiow;  
    }  
  
    public void setOcenaZStudiow(double ocenaZStudiow) {  
        this.ocenaZStudiow = ocenaZStudiow;  
    }  
  
    public Student(long indeks, String imie, String nazwisko) {  
        this.indeks = indeks;  
        this.imie = imie;  
        this.nazwisko = nazwisko;  
    }  
  
    public void wypiszDaneOStudencie(){  
        System.out.printf("Student %s %s o indeksie %s ma ocene %.2f", indeks, imie, nazwisko, ocenaZStudiow);  
    }  
}
```

Tzw. „getter” czyli metoda zwracająca wartość zmiennej prywatnej obiektu



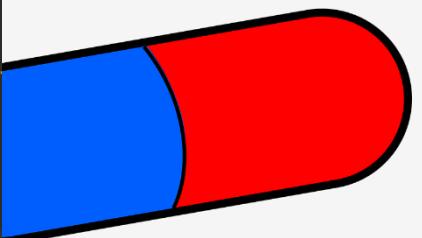


Enkapsulacja

Przykład:

```
public class Student {  
    private long indeks;  
    private String imie;  
    private String nazwisko;  
    private double ocenaZStudiow;  
  
    public double getOcenaZStudiow() {  
        return ocenaZStudiow;  
    }  
  
    public void setOcenaZStudiow(double ocenaZStudiow) {  
        this.ocenaZStudiow = ocenaZStudiow;  
    }  
  
    public Student(long indeks, String imie, String nazwisko) {  
        this.indeks = indeks;  
        this.imie = imie;  
        this.nazwisko = nazwisko;  
    }  
  
    public void wypiszDaneOStudencie(){  
        System.out.printf("Student %s %s o indeksie %s ma ocene %.2f", indeks, imie, nazwisko, ocenaZStudiow);  
    }  
}
```

Tzw. „setter” czyli metoda pozwalająca nadać wartość zmiennej prywatnej obiektu





Enkapsulacja – zadanie

1. Uniemożliw cwaniaczkowi generowanie wirtualnych pieniędzy
2. Zmień akcesory dla pól klasy KontoBankowe
3. Zobacz jak teraz zachowa się kod cwaniaczka





Modyfikatory dostępu

Modyfikatory dostępu

- Elementy w Javie takie jak klasy, pola, metody itp. mają określoną właściwość, która określa ich widoczność na przestrzeni całego projektu
- Wyróżniamy następujące modyfikatory dostępu:
 - ❖ *public* – widoczny w całym projekcie
 - ❖ *private* – widoczny jedynie wewnątrz klasy
 - ❖ *brak/pusty* – tzw. domyślny modyfikator, działa w zależności od kontekstu (np. w klasie pozwala na dostęp jedynie dla klas wewnątrz tego samego pakietu)
 - ❖ *protected* – widoczny dla klas znajdujących się wewnątrz tego samego pakietu oraz klas dziedziczących





Modyfikatory dostępu

Modyfikatory dostępu dla zmiennych – przykład:

```
public class Sklep {  
    private double gotowka;          zmiennej gotowka możemy użyć w  
    public String nazwaSklepu;  
    String imieKierownika;  
  
    public double zaplacZaZakupy(double kwota, double gotowka) {  
        this.gotowka += kwota;  
        double reszta = gotowka-kwota;  
        return reszta-0.01d;  
    }  
}
```





Modyfikatory dostępu

Modyfikatory dostępu dla zmiennych – przykład:

```
public class Sklep {  
    private double gotowka;  
    public string nazwaSklepu;      nazwaSklepu jest publiczna, dlatego  
    String imieKierownika;        jest dostępna również spoza klasy  
                                Sklep  
  
    public double zaplacZaZakupy(double kwota, double gotowka) {  
        this.gotowka += kwota;  
        double reszta = gotowka-kwota;  
        return reszta-0.01d;  
    }  
}
```





Modyfikatory dostępu

Modyfikatory dostępu dla zmiennych – przykład:

```
public class Sklep {  
    private double gotowka;  
    public String nazwaSklepu;  
    String imieKierownika;  
  
    public double zaplacZaZakupy(double kwota, double gotowka) {  
        this.gotowka += kwota;  
        double reszta = gotowka-kwota;  
        return reszta-0.01d;  
    }  
}
```

imieKierownika jest polem klasy z „default”owym modyfikatorem, w związku z czym możemy się do niego odwołać z dowolnego miejsca w klasie + z klas z tego samego pakietu





Modyfikatory dostępu

Modyfikatory dostępu dla zmiennych – przykład:

```
public class Sklep {  
    private double gotowka;  
    public String nazwaSklepu;  
    String imieKierownika;  
  
    public double zaplacZaZakupy(double kwota, double gotowka) {  
        this.gotowka += kwota;  
        double reszta = gotowka-kwota;  
        return reszta-0.01d;  
    }  
}
```

reszta ma również defaultowy modyfikator, ale nie jest polem klasy co oznacza, że możemy się do niej odwołać jedynie wewnątrz pętli metody, w której się znajduje





Modyfikatory dostępu

Modyfikatory dostępu dla zmiennych – przykład:

```
public class Sklep {  
    private double gotowka;  
    public String nazwaSklepu;  
    String imieKierownika;  
  
    public double zaplacZaZakupy(double kwota, double gotowka) {  
        this.gotowka += kwota;  
        double reszta = gotowka-kwota;  
        return reszta-0.01d;  
    }  
}
```

reszta ma również defaultowy modyfikator, ale nie jest polem klasy co oznacza, że możemy się do niej odwołać jedynie wewnątrz pętli metody, w której się znajduje





Modyfikatory dostępu

Modyfikatory dostępu dla metod – przykład:

```
public class Cukiernia {  
    private Przepis tajnyPrzepis;  
  
    public Paczek zrobPaczka() {  
        przygotujCiasto();  
        Przepis przepis = rozszyfrujTajemnaFormule();  
        return upieczWedlugPrzepisu(przepis);  
    }  
  
    Paczek upieczWedlugPrzepisu(Przepis przepis) {  
        System.out.println("Pieke wg przepisu");  
        return new Paczek();  
    }  
  
    private Przepis rozszyfrujTajemnaFormule() {  
        System.out.println("Rozszyfrowuje przepis");  
        return tajnyPrzepis;  
    }  
  
    protected void przygotujCiasto() {  
        System.out.println("Przygotowuje ciasto!");  
    }  
}
```

widoczna dla całego projektu





Modyfikatory dostępu

Modyfikatory dostępu dla metod – przykład:

```
public class Cukiernia {  
    private Przepis tajnyPrzepis;  
  
    public Paczek zrobPaczka(){  
        przygotujCiasto();  
        Przepis przepis = rozszyfrujTajemnaFormule();  
        return upieczWedlugPrzepisu(przepis);  
    }  
  
    Paczek upieczWedlugPrzepisu(Przepis przepis) {  
        System.out.println("Pieke wg przepisu");  
        return new Paczek();  
    }  
  
    private Przepis rozszyfrujTajemnaFormule() {  
        System.out.println("Rozszyfrowuje przepis");  
        return tajnyPrzepis;  
    }  
  
    protected void przygotujCiasto() {  
        System.out.println("Przygotowuje ciasto!");  
    }  
}
```

widoczna tylko dla klas
wewnętrz tego samego pakietu





Modyfikatory dostępu

Modyfikatory dostępu dla metod – przykład:

```
public class Cukiernia {  
    private Przepis tajnyPrzepis;  
  
    public Paczek zrobPaczka(){  
        przygotujCiasto();  
        Przepis przepis = rozszyfrujTajemnaFormule();  
        return upiecZWedlugPrzepisu(przepis);  
    }  
  
    Paczek upiecZWedlugPrzepisu(Przepis przepis) {  
        System.out.println("Pieke wg przepisu");  
        return new Paczek();  
    }  
  
    private Przepis rozszyfrujTajemnaFormule() {  
        System.out.println("Rozszyfrowuje przepis");  
        return tajnyPrzepis;  
    }  
  
    protected void przygotujCiasto() {  
        System.out.println("Przygotowuje ciasto!");  
    }  
}
```

widoczna tylko dla klasy
Cukiernia





Modyfikatory dostępu

Modyfikatory dostępu dla metod – przykład:

```
public class Cukiernia {  
    private Przepis tajnyPrzepis;  
  
    public Paczek zrobPaczka(){  
        przygotujCiasto();  
        Przepis przepis = rozszyfrujTajemnaFormule();  
        return upieczWedlugPrzepisu(przepis);  
    }  
  
    Paczek upieczWedlugPrzepisu(Przepis przepis) {  
        System.out.println("Pieke wg przepisu");  
        return new Paczek();  
    }  
  
    private Przepis rozszyfrujTajemnaFormule() {  
        System.out.println("Rozszyfrowuje przepis");  
        return tajnyPrzepis;  
    }  
  
    protected void przygotujCiasto() {  
        System.out.println("Przygotowuje ciasto!");  
    }  
}
```

widoczna tylko dla klas z tego samego pakietu oraz klas dziedziczących po Cukierni

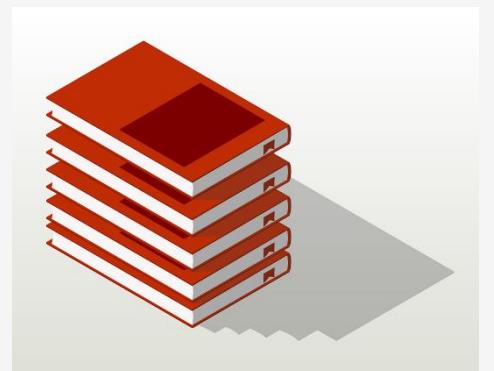




Jak przechowywane są zmienne w Javie?

Stos i sterta

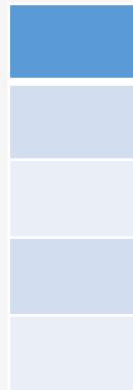
- Oba elementy zajmują pamięć RAM maszyny
- Sterta służy do przechowywania obiektów
- Stos służy do przechowywania referencji oraz typów prymitywnych
- W stercie dodatkowo utworzony jest ‚String pool’, który służy do przechowywania literałów dla typu String, dlatego:
 - String str = „Ala”, String str2 = „Ala”
 - str==str2 → zwróci prawdę, natomiast dla :
 - String str = new String(„Ala”), String str2 = new String(„Ala”)
 - str==str2 → zwróci fałsz



Jak przechowywane są zmienne w Javie?



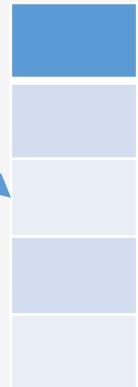
Stos i sterta





Jak przechowywane są zmienne w Javie?

Stos i sterta

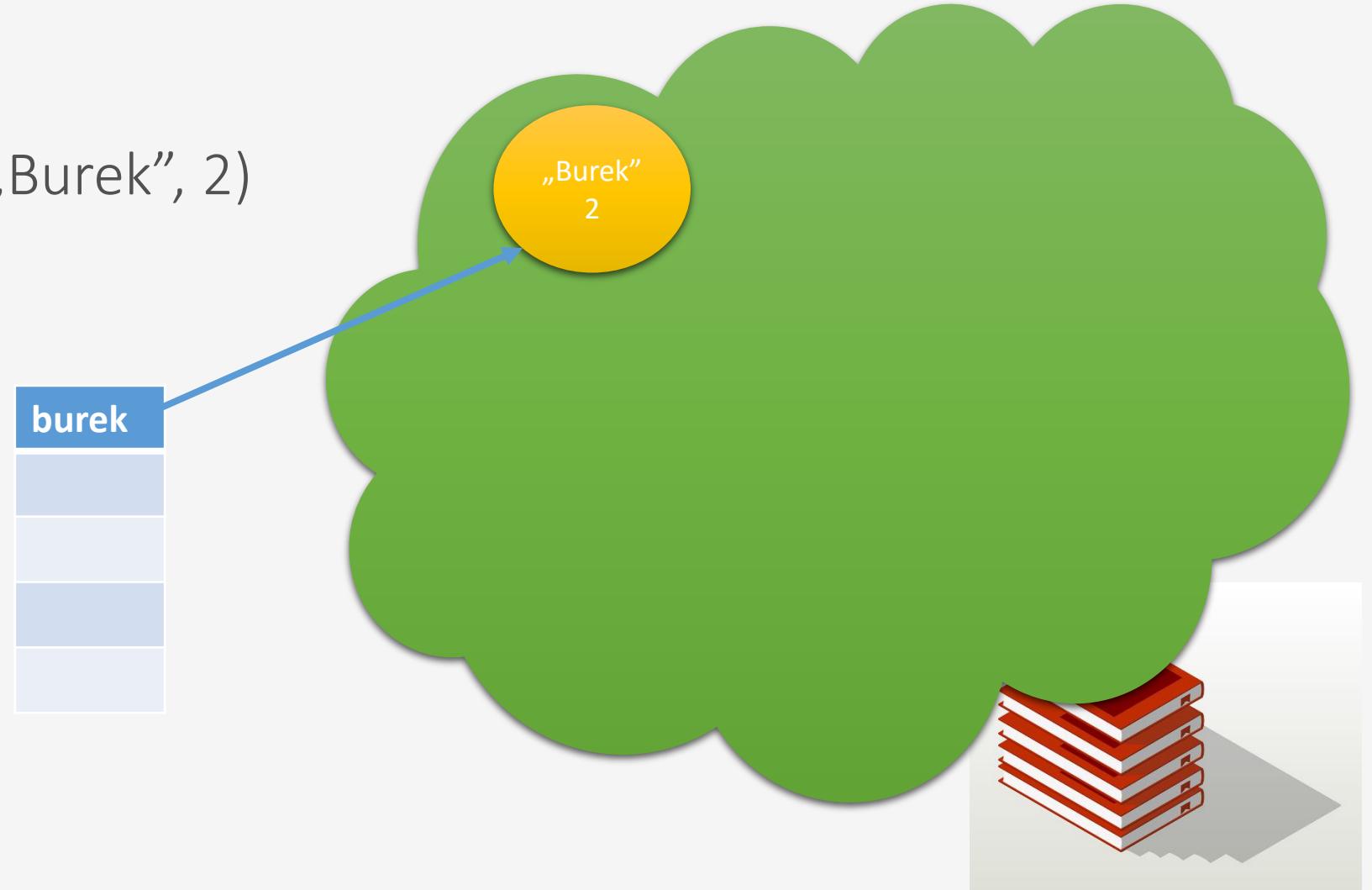




Jak przechowywane są zmienne w Javie?

Stos i sterta

Pies burek = new Burek(„Burek”, 2)



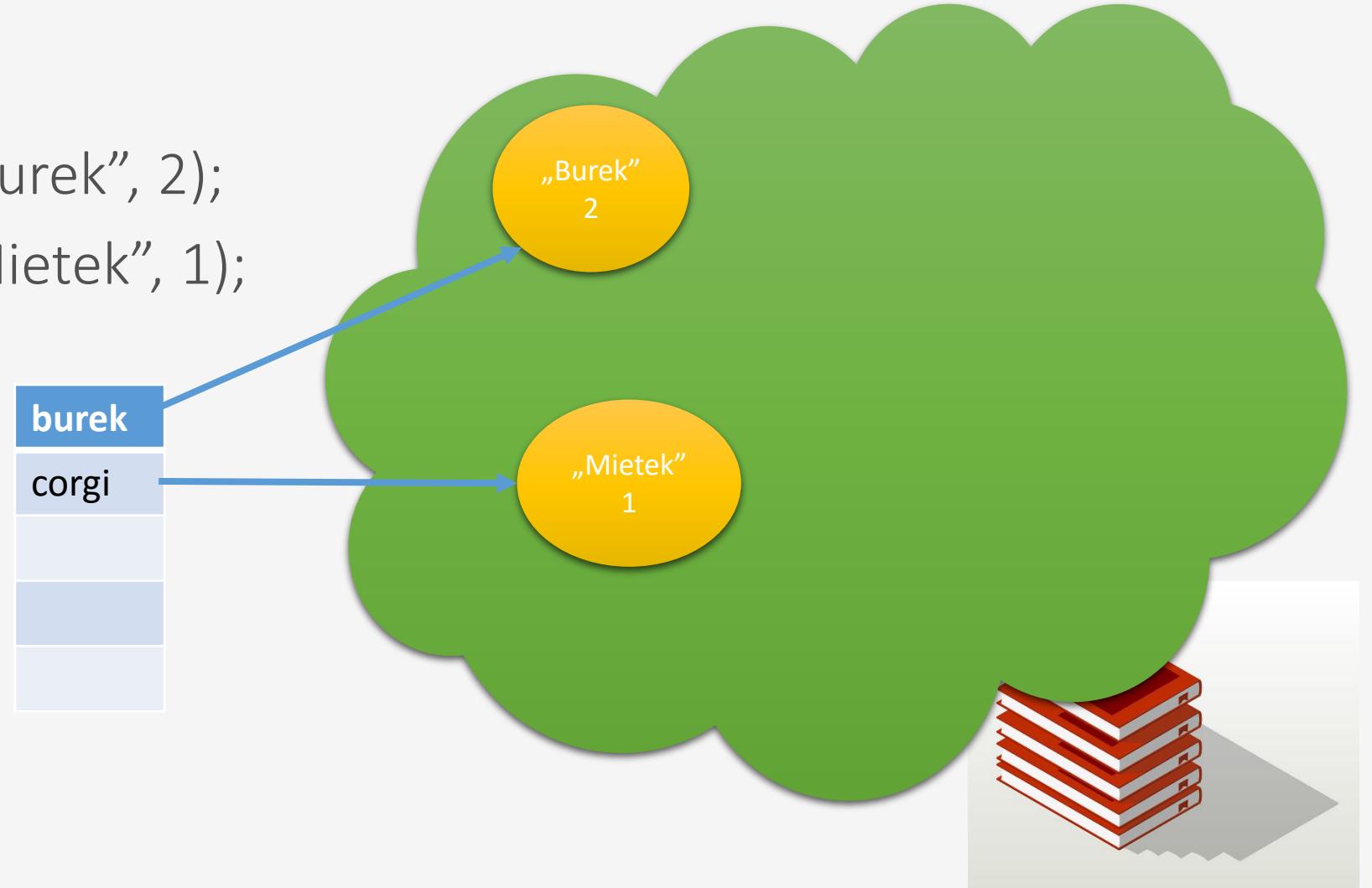


Jak przechowywane są zmienne w Javie?

Stos i sterta

```
Pies burek = new Pies(„Burek”, 2);
```

```
Pies corgi = new Pies(„Mietek”, 1);
```





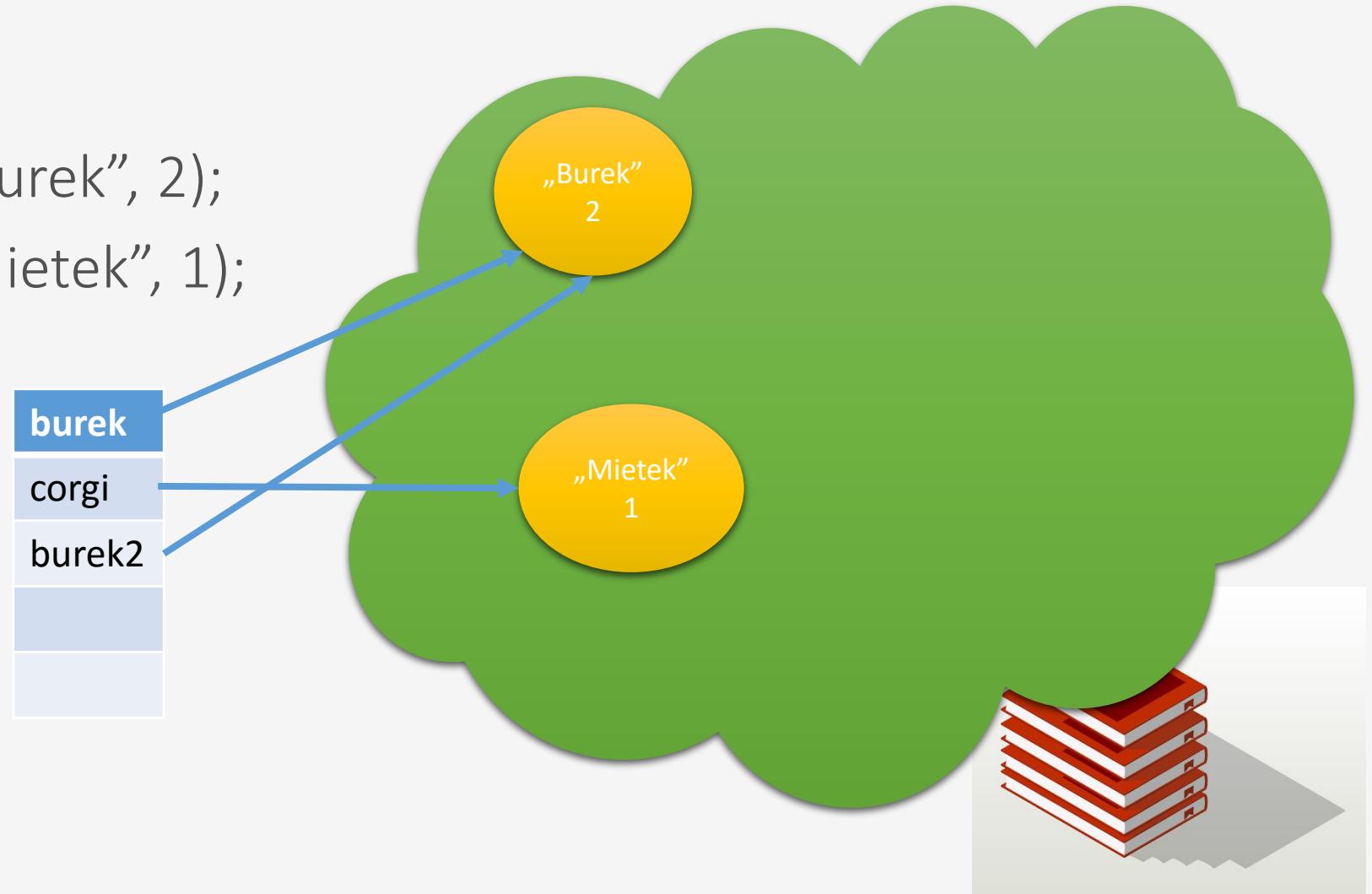
Jak przechowywane są zmienne w Javie?

Stos i sterta

```
Pies burek = new Pies(„Burek”, 2);
```

```
Pies corgi = new Pies(„Mietek”, 1);
```

```
Pies burek2 = burek;
```





Jak przechowywane są zmienne w Javie?

Stos i sterta

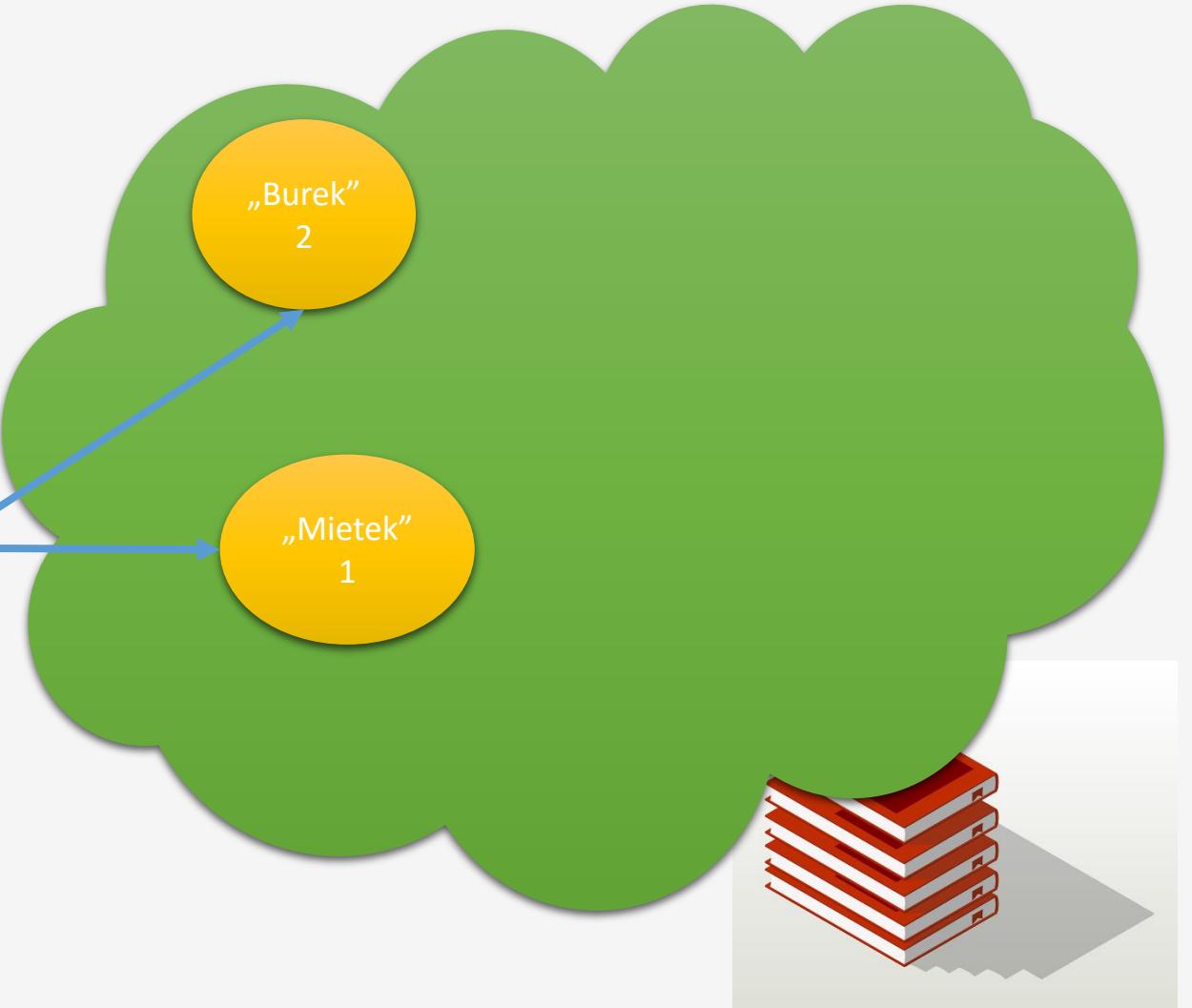
```
Pies burek = new Pies(„Burek”, 2);
```

```
Pies corgi = new Pies(„Mietek”, 1);
```

```
Pies burek2 = burek;
```

```
burek = null;
```

burek
corgi
burek2





Jak przechowywane są zmienne w Javie?

Stos i sterta

```
Pies burek = new Pies(„Burek”, 2);
```

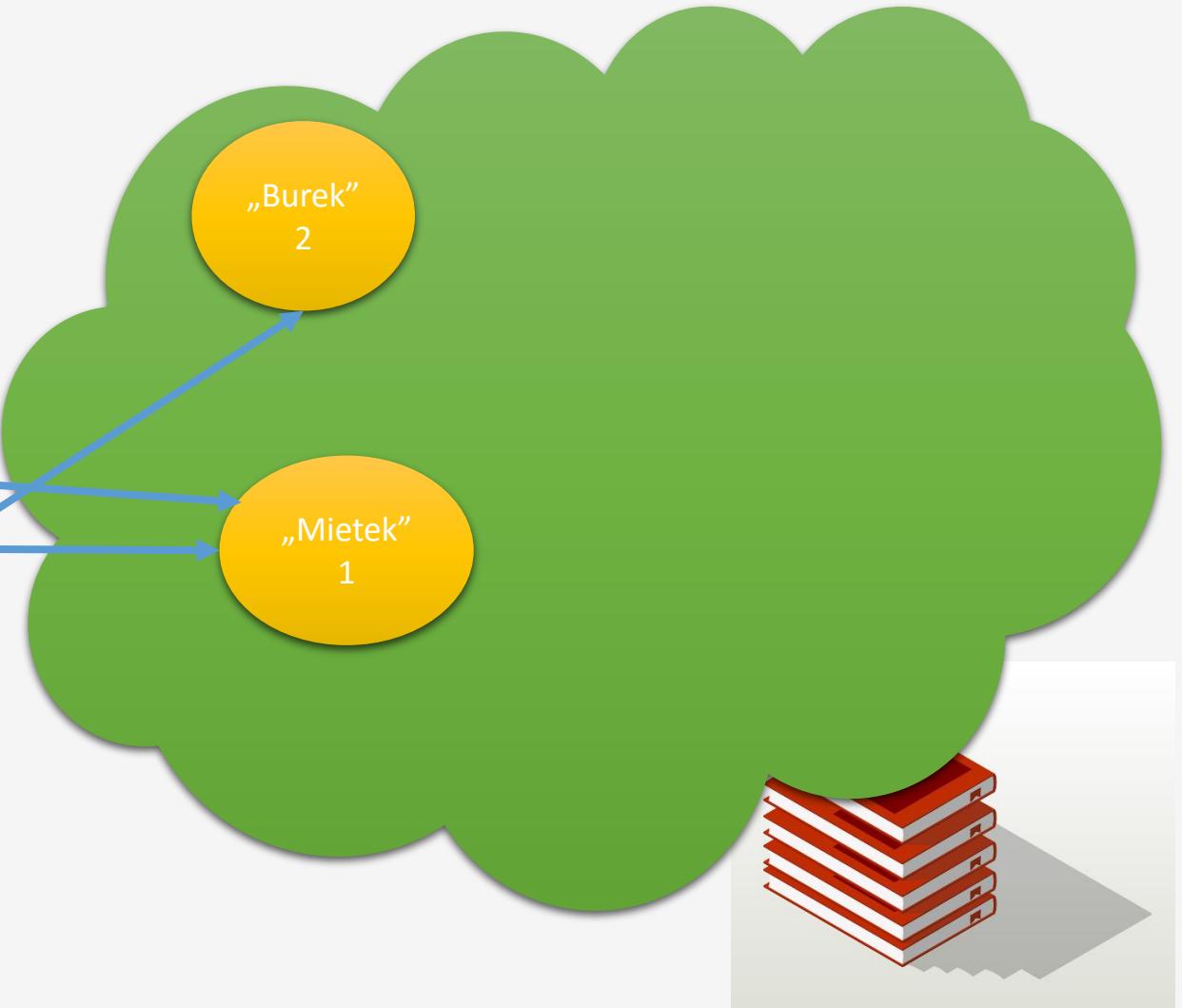
```
Pies corgi = new Pies(„Mietek”, 1);
```

```
Pies burek2 = burek;
```

```
burek = null;
```

```
burek = corgi;
```

burek
corgi
burek2





Jak przechowywane są zmienne w Javie?

Stos i sterta

```
Pies burek = new Pies(„Burek”, 2);
```

```
Pies corgi = new Pies(„Mietek”, 1);
```

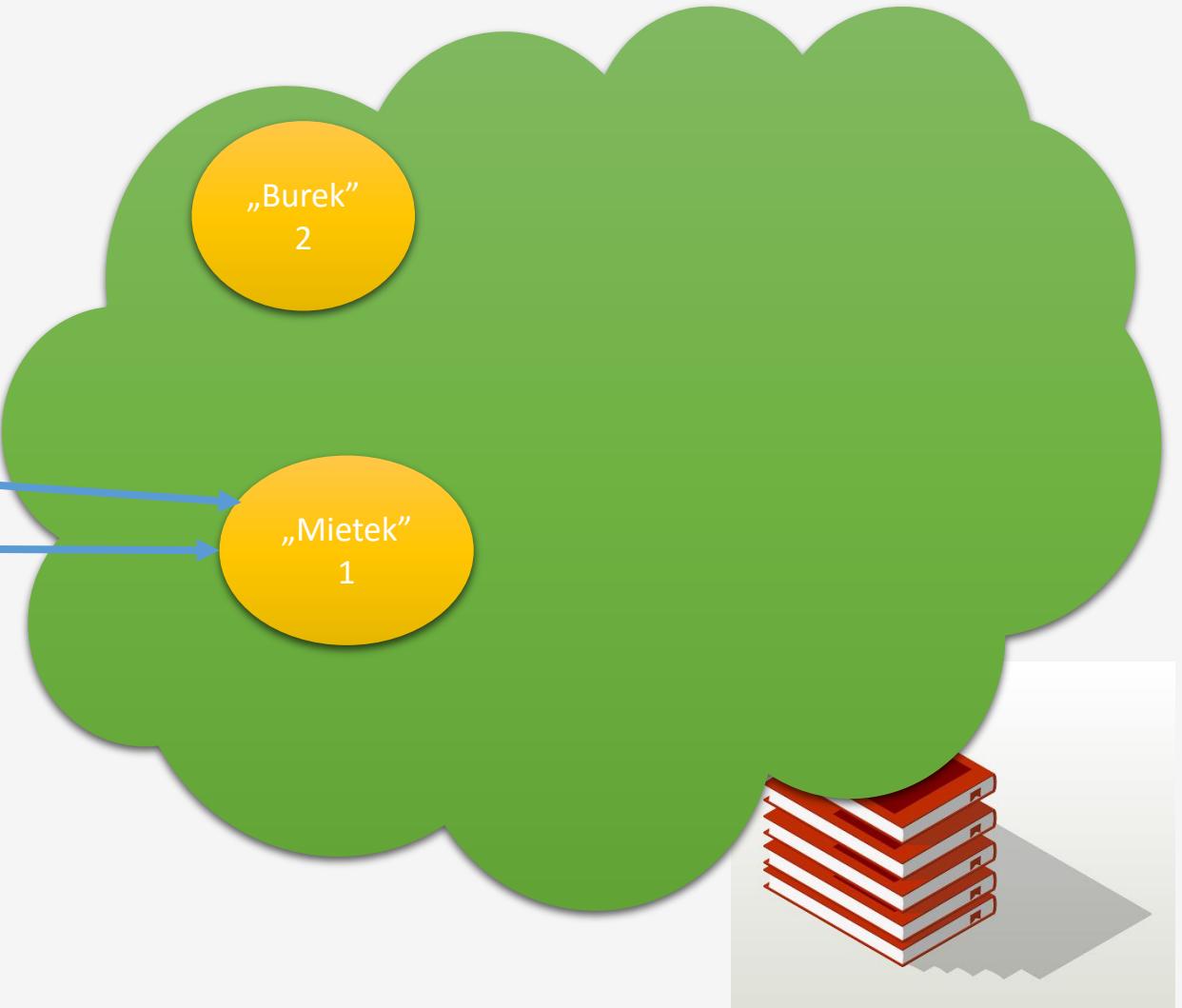
```
Pies burek2 = burek;
```

```
burek = null;
```

```
burek = corgi;
```

```
burek2 = null;
```

burek
corgi
burek2





Jak przechowywane są zmienne w Javie?

Stos i sterta

```
Pies burek = new Pies(„Burek”, 2);
```

```
Pies corgi = new Pies(„Mietek”, 1);
```

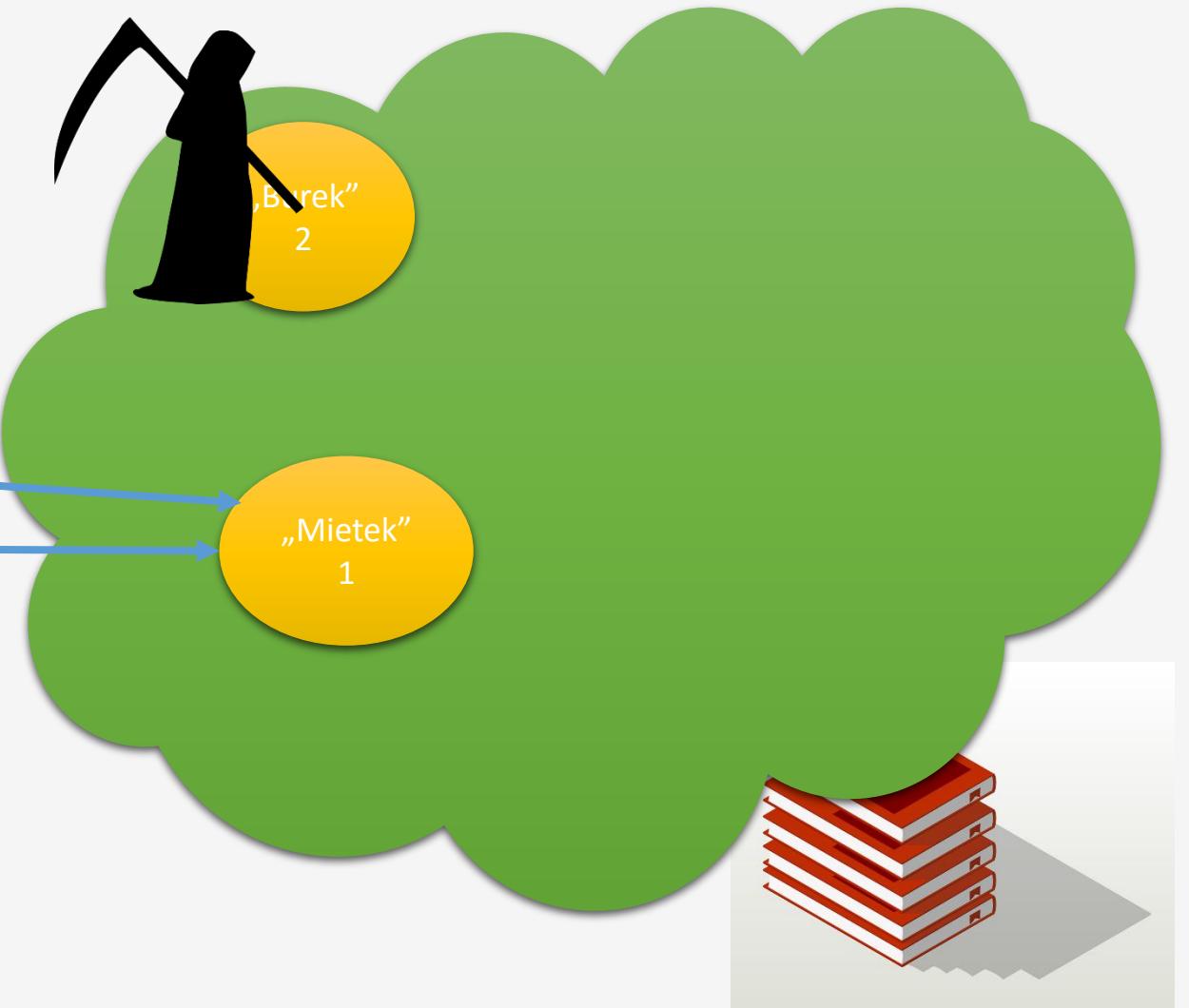
```
Pies burek2 = burek;
```

```
burek = null;
```

```
burek = corgi;
```

```
burek2 = null;
```

burek
corgi
burek2





Jak przechowywane są zmienne w Javie?

Stos i sterta

```
Pies burek = new Pies(„Burek”, 2);
```

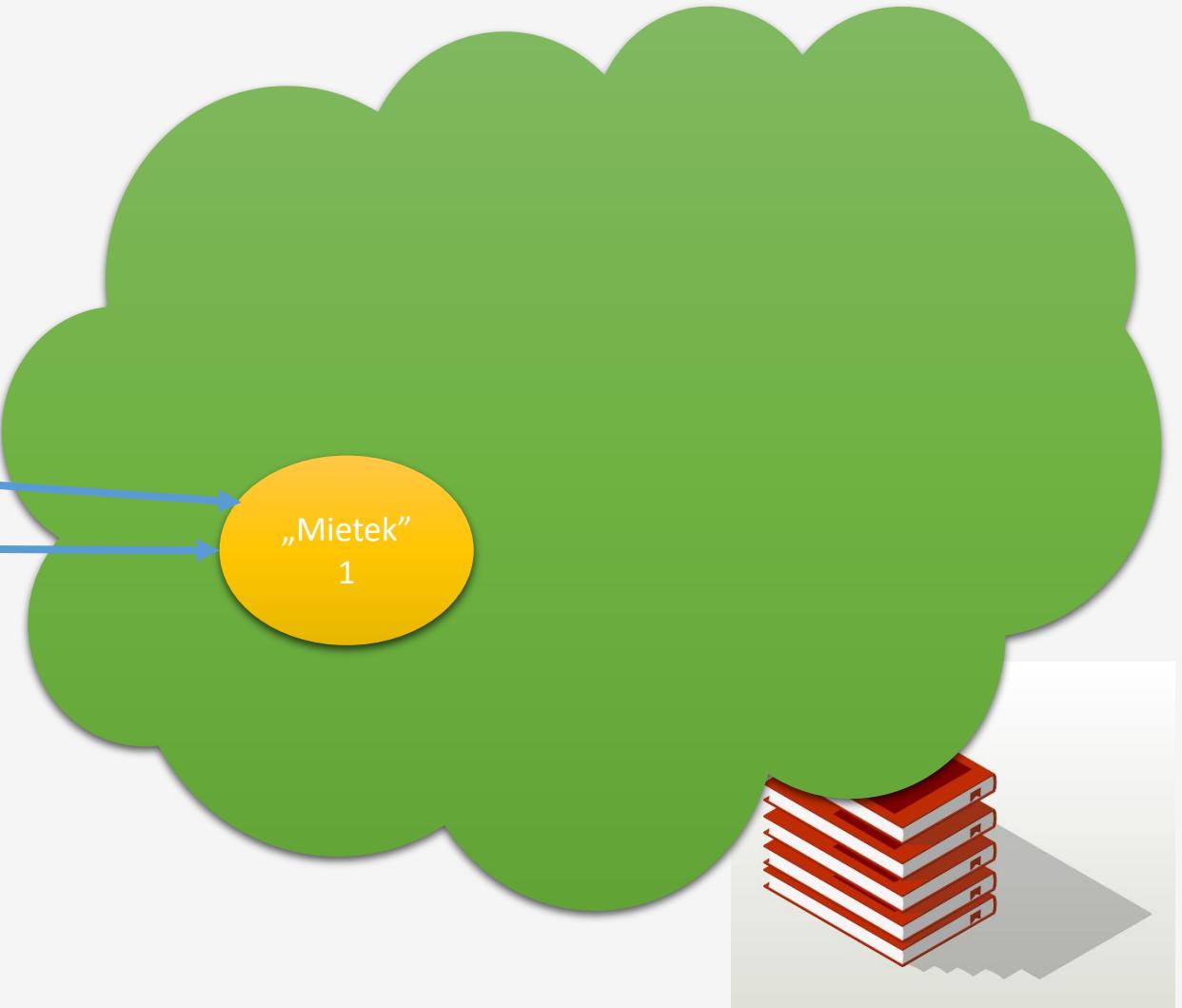
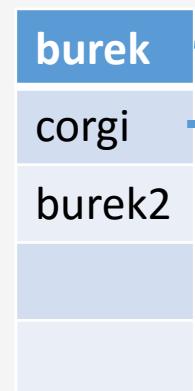
```
Pies corgi = new Pies(„Mietek”, 1);
```

```
Pies burek2 = burek;
```

```
burek = null;
```

```
burek = corgi;
```

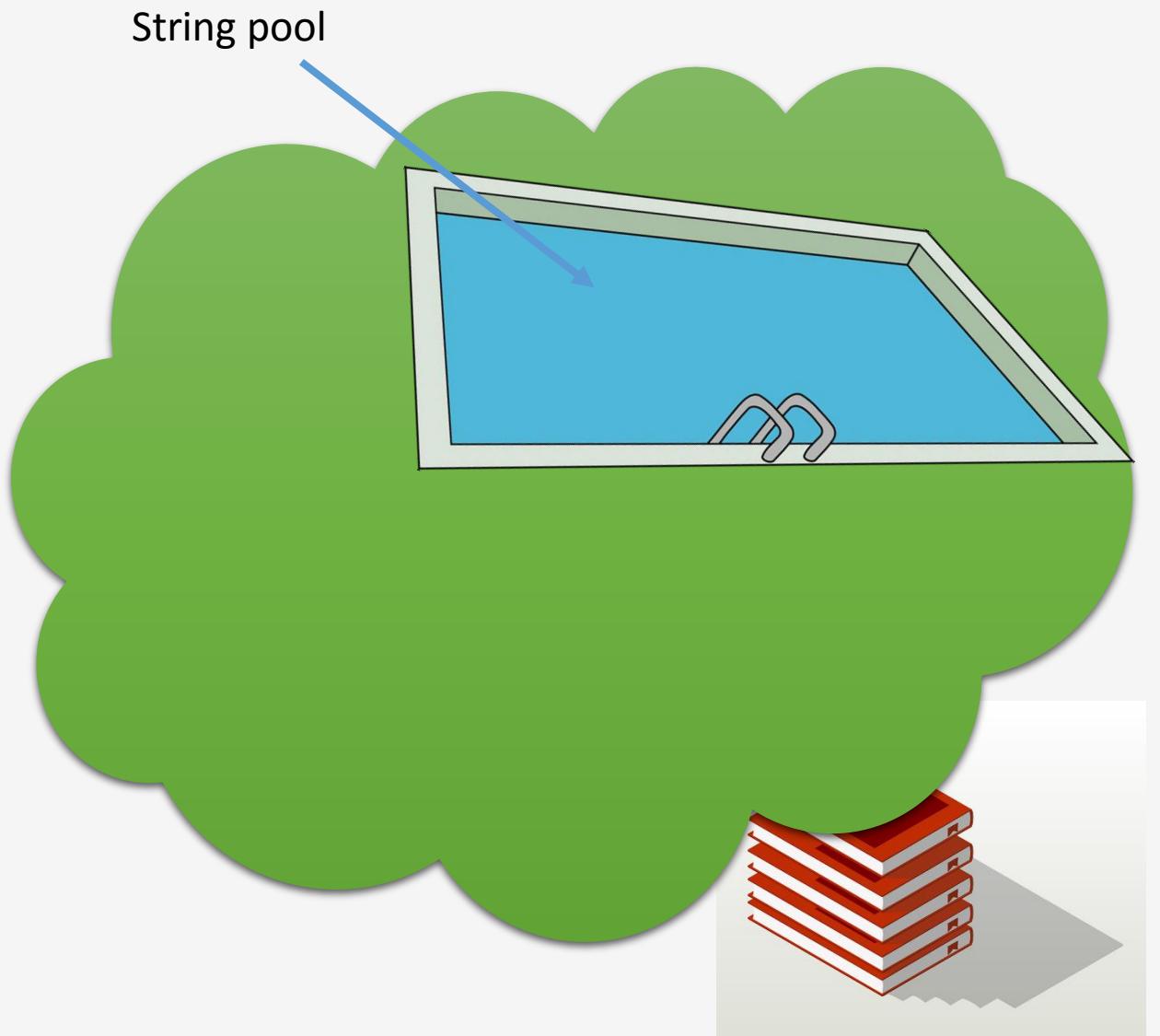
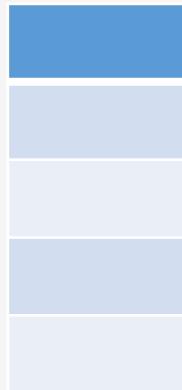
```
burek2 = null;
```





Jak przechowywane są zmienne w Javie?

Stos i sterta – przykład z String

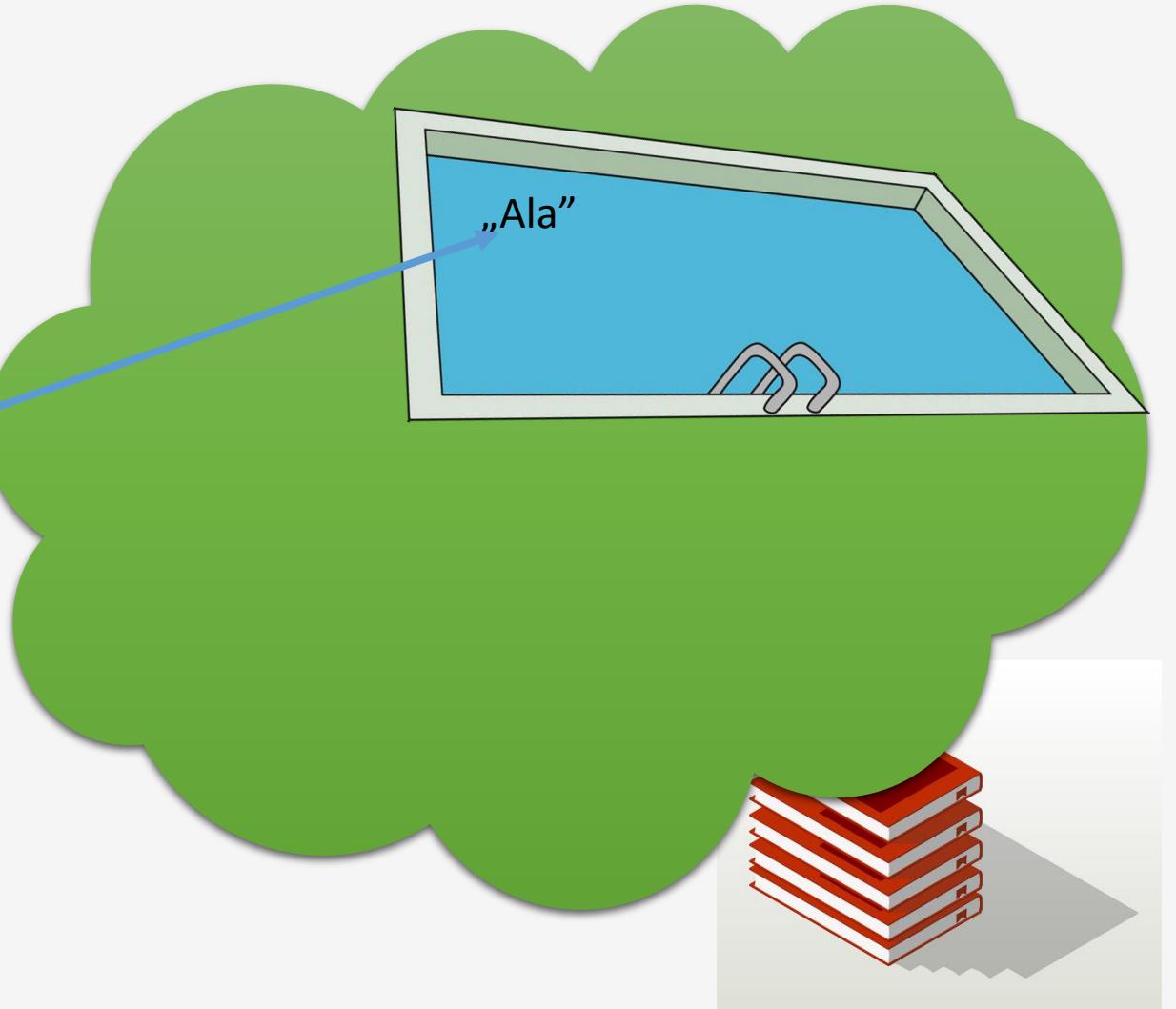
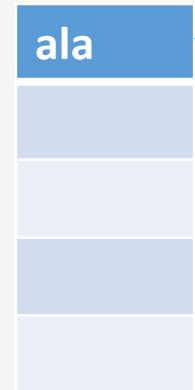




Jak przechowywane są zmienne w Javie?

Stos i sterta – przykład z String

String ala = „Ala”



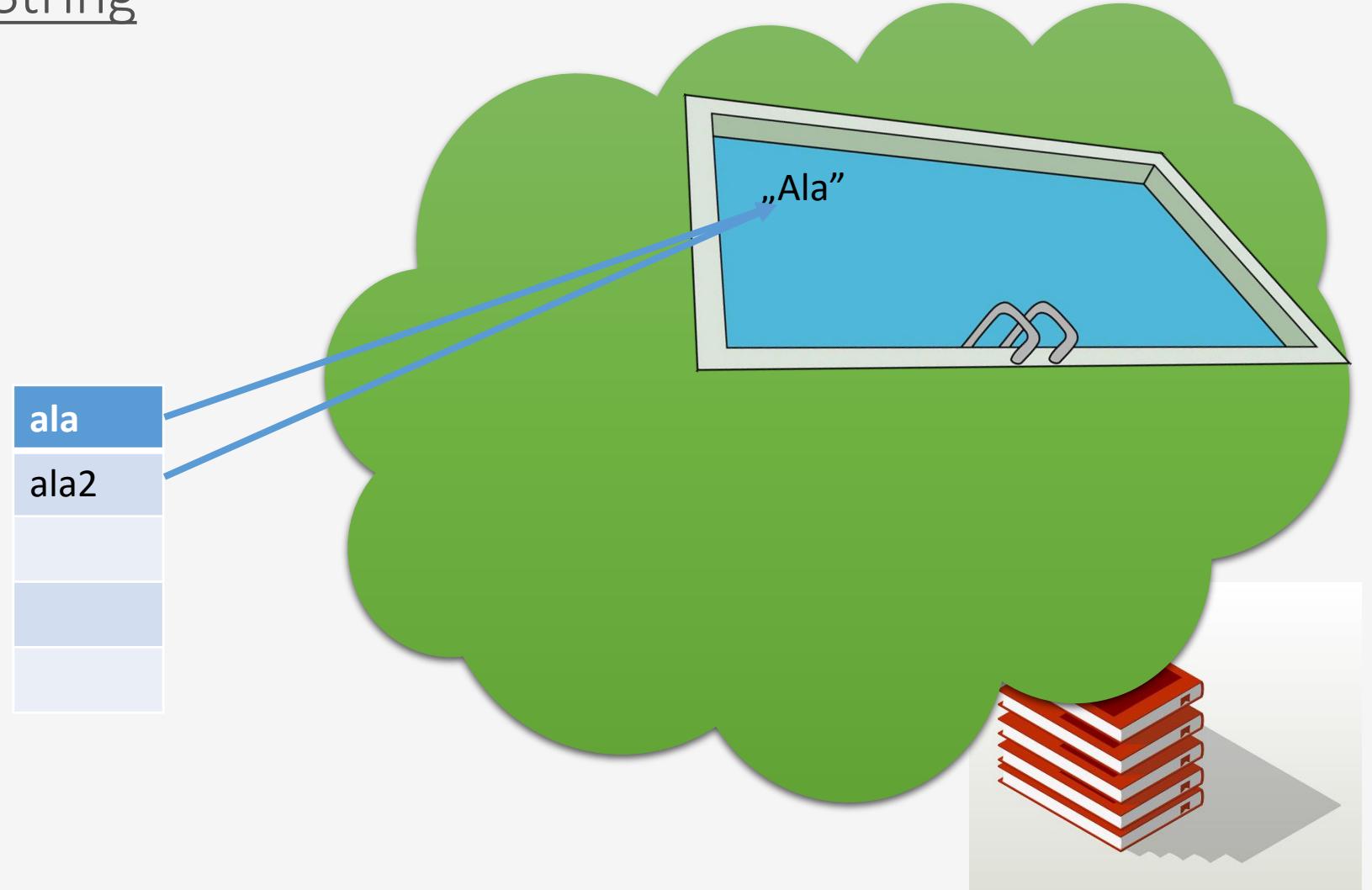


Jak przechowywane są zmienne w Javie?

Stos i sterta – przykład z String

String ala = „Ala”

String ala2 = „Ala”





Jak przechowywane są zmienne w Javie?

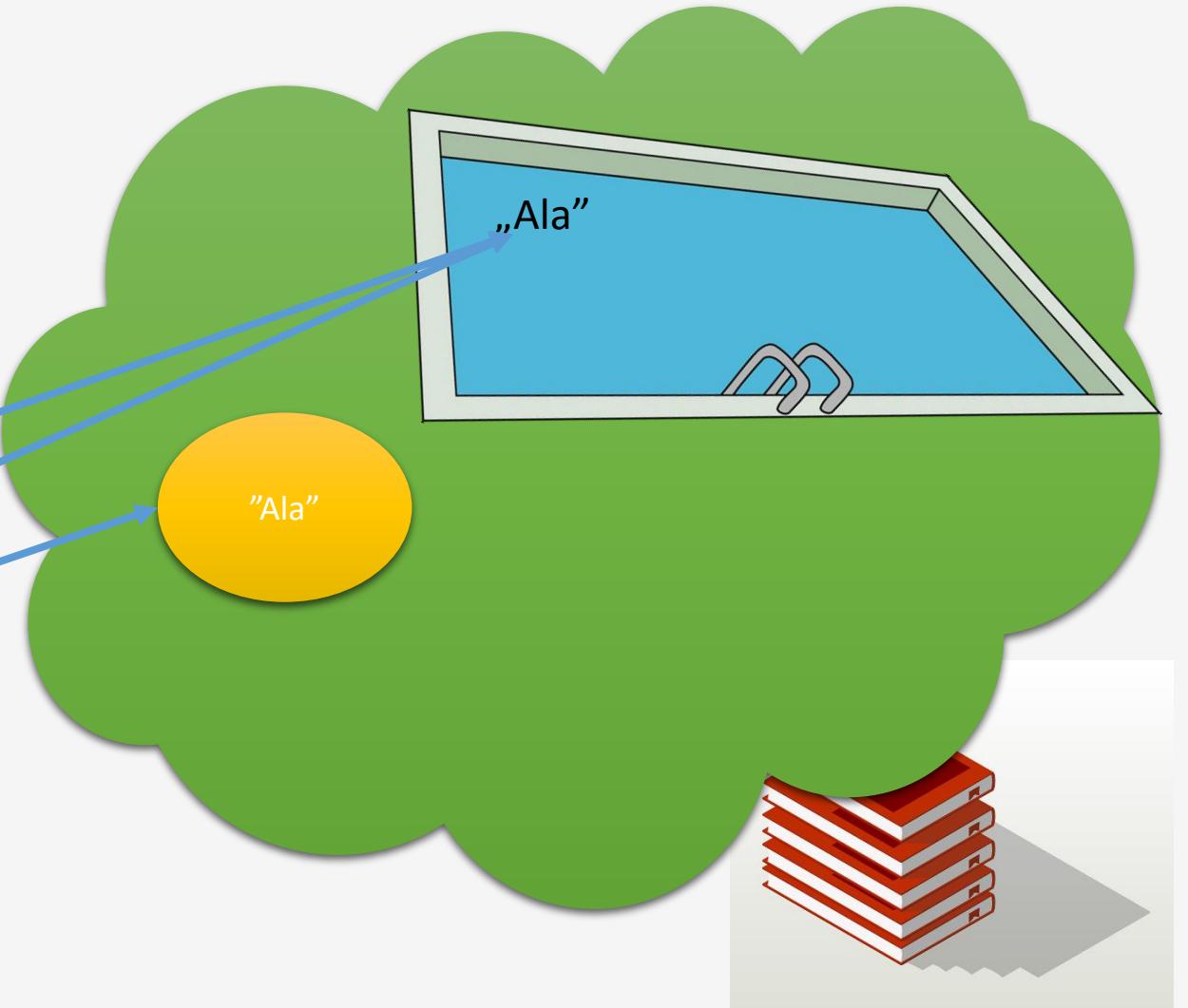
Stos i sterta – przykład z String

String ala = „Ala”

String ala2 = „Ala”

String ala3 = new String("Ala")

ala
ala2
ala3





Jak przechowywane są zmienne w Javie?

Stos i sterta – przykład z String

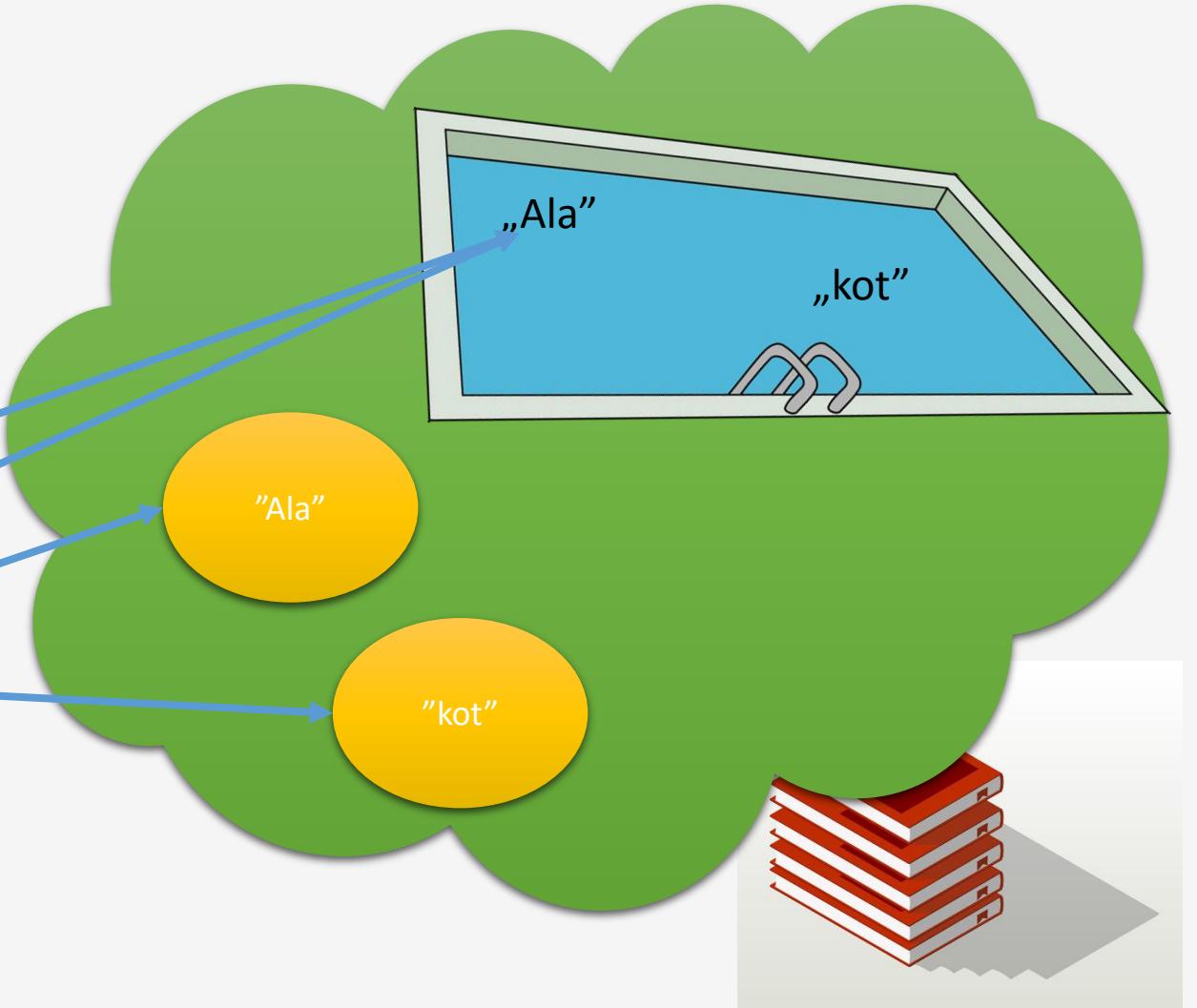
String ala = „Ala”

String ala2 = „Ala”

String ala3 = new String("Ala")

String kot = new String(„kot”)

ala
ala2
ala3
kot





Stos i sterta – Zadanie

1. Utwórz klasę „*SuperBohater*”
2. Dodaj 2 pola tekstowe : *nazwa, supermoc*
3. Utwórz 3 bohaterów
4. Przećwicz zachowanie obiektów na przykładach:
 - a. *bohater1 = bohater2; bohater1=null; sout(bohater2==null)*
 - b. *bohater1=null; bohater2=bohater1; bohater1=bohater3; sout przyrównanie do null na każdym z bohaterów*
5. *Przećwicz zachowanie się *Stringów* poprzez tworzenie literałów i nowych obiektów typu *String*
 - a. Sprawdź zachowanie metody *equals()*
 - b. Sprawdź zachowanie przyrównania *==*





Programowanie obiektowe – zadania

1. Zgadnij hasło – utwórz program obiektowy, który pozwala użytkownikowi zagrać w grę na poniższych zasadach:
 - a. Program losuje numer w zakresie od 1 do 100
 - b. Program pyta się użytkownika o numer
 - c. Jeśli użytkownik zgadnie numer – wypisuje ‘Gratulacje, wygrałeś!’
 - d. Jeśli nie – wypisuje użytkownikowi czy numer jest większy lub mniejszy od podanej przez użytkownika liczby
 - e. * Użytkownik sam określa zakres
 - f. * Po 5 nieudanych próbach program wypisuje ‘Niestety, przegrałeś’
2. Utwórz klasę Ulamek, reprezentującą ułamek zwykły. Klasa ma udostępniać operację dodawania, odejmowania, mnożenia, dzielenia oraz wyświetlanie ułamków (w formie licznik/mianownik np. 4/3). Ułamki powinny mieć liczbę całkowitą zarówno w liczniku jak i mianowniku. Przetestuj swoje rozwiązanie





Programowanie obiektowe

Typy wyliczeniowe

- Pozwala na utworzenie nowego typu danych ograniczonego do określonych wartości
- Nazwy wartości piszemy wielką literą
- Enumy mogą posiadać konstruktory, metody i pola (ale nie można bezpośrednio wywołać konstruktora)
- Można wykorzystać w instrukcji switch, oraz porównać przez `,==,`





Programowanie obiektowe

Typy wyliczeniowe

```
public enum Bilet {  
    ULGOWY,  
    NORMALNY,  
    RODZINNY;  
  
    public static void main(String[] args) {  
        Bilet ulgowy = Bilet.ULGOWY;  
        Bilet normalny = Bilet.NORMALNY;  
  
        System.out.println(ulgowy);  
        System.out.println(normalny);  
    }  
}
```





Programowanie obiektowe

Typy wyliczeniowe

```
public enum Bilet {  
    ULGOWY,  
    NORMALNY,  
    RODZINNY;  
  
    public static void main(String[] args) {  
        Bilet ulgowy = Bilet.ULGOWY;  
        Bilet normalny = Bilet.NORMALNY;  
  
        System.out.println(ulgowy);  
        System.out.println(normalny);  
    }  
}
```





Programowanie obiektowe

Typy wyliczeniowe

```
public enum Bilet {  
    ULGOWY,  
    NORMALNY,  
    RODZINNY;  
  
    public static void main(String[] args) {  
        Bilet ulgowy = Bilet.ULGOWY;  
        Bilet normalny = Bilet.NORMALNY;  
  
        System.out.println(ulgowy);  
        System.out.println(normalny);  
    }  
}
```

ograniczona pula wartości jakie może przyjąć





Programowanie obiektowe

Typy wyliczeniowe

```
public enum Bilet {  
    ULGOWY,  
    NORMALNY,  
    RODZINNY;  
  
    public static void main(String[] args) {  
        Bilet ulgowy = Bilet.ULGOWY;  
        Bilet normalny = Bilet.NORMALNY;  
  
        System.out.println(ulgowy);  
        System.out.println(normalny);  
    }  
}
```

odwołujemy się przez przyjmowaną wartość





Programowanie obiektowe

Typy wyliczeniowe

```
public enum Bilet {  
    ULGOWY,  
    NORMALNY,  
    RODZINNY;  
  
    public static void main(String[] args) {  
        Bilet ulgowy = Bilet.ULGOWY;  
        Bilet normalny = Bilet.NORMALNY;  
  
        System.out.println(ulgowy);  
        System.out.println(normalny);  
    }  
}
```

domyślnie zwróci nam wartość,
czyli „ULGOWY” i „NORMALNY”





Programowanie obiektowe

Typy wyliczeniowe – przykład z użyciem konstruktora

```
public enum Bilet {  
    ULGOWY(1.60d),  
    NORMALNY(3.20d),  
    RODZINNY(2.00d);  
  
    private double cena;  
  
    Bilet(double cena) {  
        this.cena = cena;  
    }  
  
    public double pobierzCene() {  
        return this.cena;  
    }  
  
    public static void main(String[] args) {  
        Bilet ulgowy = Bilet.ULGOWY;  
  
        System.out.println(ulgowy.pobierzCene());  
        System.out.println(Bilet.NORMALNY.pobierzCene());  
    }  
}
```

zdefiniowany konstruktor nie
może być używany jawnie





Programowanie obiektowe

Typy wyliczeniowe – przykład z użyciem konstruktora

```
public enum Bilet {  
    ULGOWY(1.60d),  
    NORMALNY(3.20d),  
    RODZINNY(2.00d);  
  
    private double cena;  
  
    Bilet(double cena) {  
        this.cena = cena;  
    }  
  
    public double pobierzCene() {  
        return this.cena;  
    }  
  
    public static void main(String[] args) {  
        Bilet ulgowy = Bilet.ULGOWY;  
  
        System.out.println(ulgowy.pobierzCene());  
        System.out.println(Bilet.NORMALNY.pobierzCene());  
    }  
}
```

w nawiasie podajemy
argumenty odpowiadające
konstruktorowi





Programowanie obiektowe

Typy wyliczeniowe – przykład z użyciem konstruktora

```
public enum Bilet {  
    ULGOWY(1.60d),  
    NORMALNY(3.20d),  
    RODZINNY(2.00d);  
  
    private double cena;  
  
    Bilet(double cena) {  
        this.cena = cena;  
    }  
  
    public double pobierzCene() {  
        return this.cena;  
    }  
  
    public static void main(String[] args) {  
        Bilet ulgowy = Bilet.ULGOWY;  
  
        System.out.println(ulgowy.pobierzCene());  
        System.out.println(Bilet.NORMALNY.pobierzCene());  
    }  
}
```

enum może posiadać własne pola, które powinny być prywatne, aby zapobiec nieprawidłowemu działaniu





Programowanie obiektowe

Typy wyliczeniowe – przykład z użyciem konstruktora

```
public enum Bilet {  
    ULGOWY(1.60d),  
    NORMALNY(3.20d),  
    RODZINNY(2.00d);  
  
    private double cena;  
  
    Bilet(double cena) {  
        this.cena = cena;  
    }  
  
    public double pobierzCene() {  
        return this.cena;  
    }  
  
    public static void main(String[] args) {  
        Bilet ulgowy = Bilet.ULGOWY;  
  
        System.out.println(ulgowy.pobierzCene());  
        System.out.println(Bilet.NORMALNY.pobierzCene());  
    }  
}
```

enum może posiadać własne metody usprawniające jego wykorzystanie





Programowanie obiektowe

Typy wyliczeniowe – przykład z użyciem konstruktora

```
public enum Bilet {  
    ULGOWY(1.60d),  
    NORMALNY(3.20d),  
    RODZINNY(2.00d);  
  
    private double cena;  
  
    Bilet(double cena) {  
        this.cena = cena;  
    }  
  
    public double pobierzCene() {  
        return this.cena;  
    }  
  
    public static void main(String[] args) {  
        Bilet ulgowy = Bilet.ULGOWY;  
  
        System.out.println(ulgowy.pobierzCene());  
        System.out.println(Bilet.NORMALNY.pobierzCene());  
    }  
}
```

mamy dwa sposoby na wywołanie metody – bezpośrednie odwołanie do wartości, lub przez odwołanie do zmiennej





Typy wyliczeniowe – zadanie

1. Zamodelować sytuację zakupu biletu
2. Utworzyć enum *Bilet*
3. Nadać następujące wartości:
 - a. *ULGOWY_GODZINNY*
 - b. *ULGOWY_CALODNIOWY*
 - c. *NORMALNY_GODZINNY*
 - d. *NORMALNY_CALODNIOWY*
 - e. *BRAK_BILETU*
4. Dodać konstruktor przyjmujący 2 parametry:
 - a. *cena (double)*
 - b. *czasJazdy w minutach (int)*
5. Nadać odpowiednie wartości startowe
6. Utworzyć metodę *pobierzCeneBiletu():int*
7. Utworzyć metodę *pobierzCzasJazdy():int*
8. Utworzyć metodę *wyswietlDaneOBilecie():void* np. „*Bilet ulgowy 1-godzinny*”
9. Utwórz kilka biletów, wywołaj metody i przetestuj działanie
10. * Utwórz metodę statyczną przyjmującą wiek osoby kupującej bilet, czas jazdy w minutach oraz kwotę – zwróć odpowiedni bilet (w przypadku niewystarczających środków zwróć *BRAK_BILETU*)





Dziedziczenie

Dziedziczenie

- Klasy mogą po sobie dziedziczyć poprzez dodanie słowa `extends`
- Klasa dziedzicząca (podklasa) dziedziczy pola i metody klasy nadzędnej (nadklasy)
- Klasa `Object` jest klasą ogólną po której dziedziczy (niejawnie) każda inna klasa
- Podklasa może dziedziczyć tylko po 1 klasie
- Ale każda podklasa może mieć kolejną podkласę ☺





Dziedziczenie – zadanie

1. Utwórz klasę *Samochód* o metodach:

- `przyspiesz():void` - metoda zwiększa aktualną predkość samochodu o 10 km/h.

Ale auto nie może jechać więcej niż 120km/h. Wyświetl tekst „Przyspieszam do xxx km/h”

- `wlaczSwiatla():void`
- `czySwiatlaWlaczone():boolean`

2. Utwórz klasę *Kabriolet* dziedziczącą po klasie *Samochód*

3. Dodaj dodatkową metody:

- `schowajDach():void`
- `czyDachSchowany():boolean`





Dziedziczenie

Dziedziczenie – co jeśli nie jesteśmy zadowoleni z „spadku” ?

- Metody „odziedziczone” możemy nadpisywać
- Stosujemy specjalną anotację `@Override` dla podkreślenia, że metoda w klasie podzielnej zachowuje swoją niezależność w sposobie jej wykonania





Dziedziczenie – zadanie

1. Nadpisz metodę `przyspiesz()` w `Kabriolecie`, tak aby samochód mógł jechać max 180 km/h
2. Gdy dach jest schowany wyświetl napis „Dach jest już schowany”





Dziedziczenie

Dziedziczenie – istotne metody dziedziczone po Object

- *toString():String* – zwraca reprezentację naszego obiektu w formie ciągu znaków
- *equals(Obj):boolean* – zwraca true jeśli obiekty są sobie równe





Dziedziczenie po Object – zadanie

1. Zmodyfikuj konstruktor klasy *Samochód* tak, aby pobierała 3 parametry: kolor, markę i rocznik
2. Zaktualizuj konstruktor klasy *Kabriolet*
3. Nadpisz metodę `'toString()'` klasy *Samochod*, tak aby wyświetlała opis „*{kolor} samochód marki {marka} rocznik {rocznik}*”
4. Nadpisz metodę `,toString()'` klasy *Kabriolet*, aby wyświetlała następujący opis:
„*{kolor} samochód marki {marka} rocznik {rocznik} z rozsuwanym dachem*”
5. Nadpisz metodę `equals()` klasy *Samochod*
6. * Zmodyfikuj metodę w klasie *Kabriolet*, aby jedynie „dorzucała” ostatnie 3 słowa od siebie (z rozsuwanym dachem)





Dziedziczenie

Co jeśli nie chcemy, aby po nas dziedziczone?

- Słowo kluczowe *final*
- Przyjmuje różne zastosowania w zależności od kontekstu:
 - a. *final class* – klasa, po której nie można dziedziczyć
 - b. *final method* – metoda, której nie można nadpisać
 - c. *final zmienna* – zmienna, która nie może zmienić referencji





Dziedziczenie

Użycie *final* w zależności od kontekstu:

```
public final class KlasaFinalna {  
    public final int finalnyPrymityw = 10;  
    public final Osoba finalnaReferencja = new Osoba(imie: "Anna", rokUrodzenia: 1995);  
  
    public final void metodaFinalna(){  
        System.out.println("Nie można mnie nadpisać!");  
    }  
  
    public static void main(String[] args) {  
        KlasaFinalna test = new KlasaFinalna();  
        test.finalnyPrymityw = 10;  
        test.finalnaReferencja.imie = "Beata";  
        System.out.println(test.finalnaReferencja);  
    }  
}
```

Klasa po której nie możemy dziedziczyć





Dziedziczenie

Użycie *final* w zależności od kontekstu:

```
public final class KlasaFinalna {  
    public final int finalnyPrymityw = 10;  
    public final Osoba finalnaReferencja = new Osoba(imie: "Anna", rokUrodzenia: 1995);  
  
    public final void metodaFinalna() {  
        System.out.println("Nie można mnie nadpisać!");  
    }  
  
    public static void main(String[] args) {  
        KlasaFinalna test = new KlasaFinalna();  
        test.finalnyPrymityw = 10;  
        test.finalnaReferencja.imie = "Beata";  
        System.out.println(test.finalnaReferencja);  
    }  
}
```

Ostateczna wartość zmiennej,
której nie możemy nadpisać





Dziedziczenie

Użycie *final* w zależności od kontekstu:

```
public final class KlasaFinalna {  
    public final int finalnyPrymityw = 10;  
    public final Osoba finalnaReferencja = new Osoba(imie: "Anna", rokUrodzenia: 1995);  
  
    public final void metodaFinalna() {  
        System.out.println("Nie można mnie nadpisać!");  
    }  
  
    public static void main(String[] args) {  
        KlasaFinalna test = new KlasaFinalna();  
        test.finalnyPrymityw = 10;  
        test.finalnaReferencja.imie = "Beata";  
        System.out.println(test.finalnaReferencja);  
    }  
}
```

Próba nadpisania = błąd kompilacji





Dziedziczenie

Użycie *final* w zależności od kontekstu:

```
public final class KlasaFinalna {  
    public final int finalnyPrymityw = 10;  
    public final Osoba finalnaReferencja = new Osoba(imie: "Anna", rokUrodzenia: 1995);  
  
    public final void metodaFinalna() {  
        System.out.println("Nie można mnie nadpisać!");  
    }  
  
    public static void main(String[] args) {  
        KlasaFinalna test = new KlasaFinalna();  
        test.finalnyPrymityw = 10;  
        test.finalnaReferencja.imie = "Beata";  
        System.out.println(test.finalnaReferencja);  
    }  
}
```

Finalna referencja, której nie
możemy zmienić





Dziedziczenie

Użycie *final* w zależności od kontekstu:

```
public final class KlasaFinalna {  
    public final int finalnyPrymityw = 10;  
    public final Osoba finalnaReferencja = new Osoba(imie: "Anna", rokUrodzenia: 1995);  
  
    public final void metodaFinalna() {  
        System.out.println("Nie można mnie nadpisać!");  
    }  
  
    public static void main(String[] args) {  
        KlasaFinalna test = new KlasaFinalna();  
        test.finalnyPrymityw = 10;  
        test.finalnaReferencja.imie = "Beata";  
        System.out.println(test.finalnaReferencja);  
    }  
}
```

Ale możemy zmodyfikować obiekt, na który wskazuje ☺





Dziedziczenie

Użycie *final* w zależności od kontekstu:

```
public final class KlasaFinalna {  
    public final int finalnyPrymityw = 10;  
    public final Osoba finalnaReferencja = new Osoba(imie: "Anna", rokUrodzenia: 1995);  
  
    public final void metodaFinalna() {  
        System.out.println("Nie można mnie nadpisać!");  
    }  
  
    public static void main(String[] args) {  
        KlasaFinalna test = new KlasaFinalna();  
        test.finalnyPrymityw = 10;  
        test.finalnaReferencja.imie = "Beata";  
        System.out.println(test.finalnaReferencja);  
    }  
}
```

Analogicznie do klasy, metoda finalna, której nie możemy nadpisać w podklasie





Dziedziczenie

To po co używać *final* ?

- Przy oznaczaniu pól wymuszamy, aby metoda konstruktora nadawała wartości
- Przechowywanie stałych jak np. liczba PI
- Metoda jest skończona i chcemy aby zawsze była wykonywana w ten sam sposób

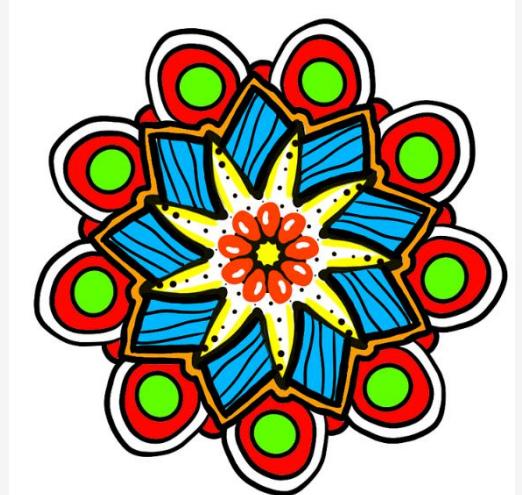




Abstrakcja

Klasy Abstrakcyjne

- Służą do lepszego zamodelowania świata rzeczywistego
- Upraszczają rzeczywistość
- Brak możliwości zainicjowania obiektu
- Każda klasa jest abstrakcyjna jeśli posiada chociaż jedną metodę abstrakcyjną

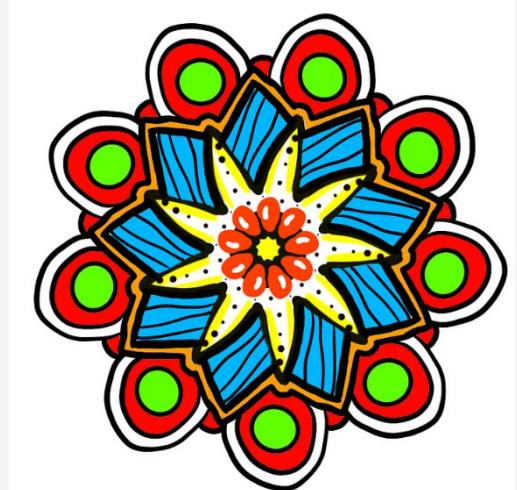




Abstrakcja

Metoda abstrakcyjna

- Metoda która posiada następujące cechy:
 - nazwa
 - typ zwracany
 - przyjmowane argumenty
 - modyfikator dostępu
- Ale, nie posiada ciała metody, czyli robi nic



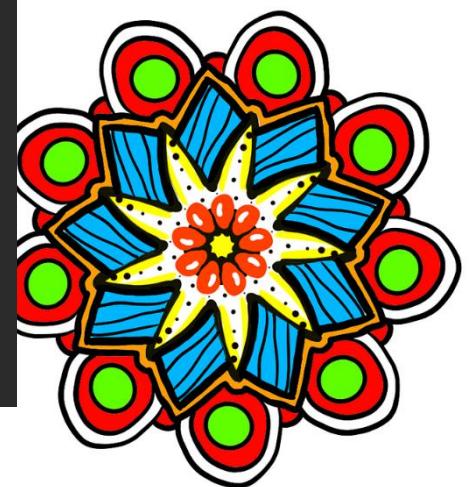


Abstrakcja

Abstrakcja na przykładzie

```
public abstract class KlasaAbstrakcyjna {  
    public abstract int zrobCos(int xRazy);  
    public void toJuzRobie(){  
        System.out.println("Cos robi!");  
    }  
}  
  
class KlasaJuzNieAbstrakcyjna extends KlasaAbstrakcyjna{  
  
    @Override  
    public int zrobCos(int xRazy) {  
        System.out.println("Robie cos!");  
        return xRazy*xRazy;  
    }  
}
```

Deklaracja klasy abstrakcyjnej



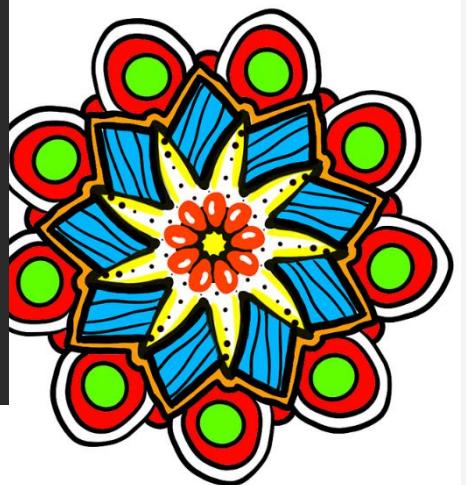


Abstrakcja

Abstrakcja na przykładzie

```
public abstract class KlasaAbstrakcyjna {  
    public abstract int zrobCos(int xRazy);  
    public void toJuzRobie(){  
        System.out.println("Cos robi!");  
    }  
}  
  
class KlasaJuzNieAbstrakcyjna extends KlasaAbstrakcyjna{  
  
    @Override  
    public int zrobCos(int xRazy) {  
        System.out.println("Robie cos!");  
        return xRazy*xRazy;  
    }  
}
```

Deklaracja metody abstrakcyjnej



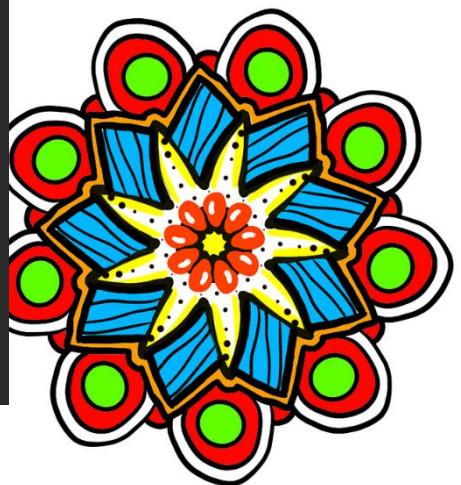


Abstrakcja

Abstrakcja na przykładzie

```
public abstract class KlasaAbstrakcyjna {  
    public abstract int zrobCos(int xRazy);  
    public void toJuzRobie(){  
        System.out.println("Cos robi!");  
    }  
}  
  
class KlasaJuzNieAbstrakcyjna extends KlasaAbstrakcyjna{  
  
    @Override  
    public int zrobCos(int xRazy) {  
        System.out.println("Robie cos!");  
        return xRazy*xRazy;  
    }  
}
```

Implementacja metody abstrakcyjnej

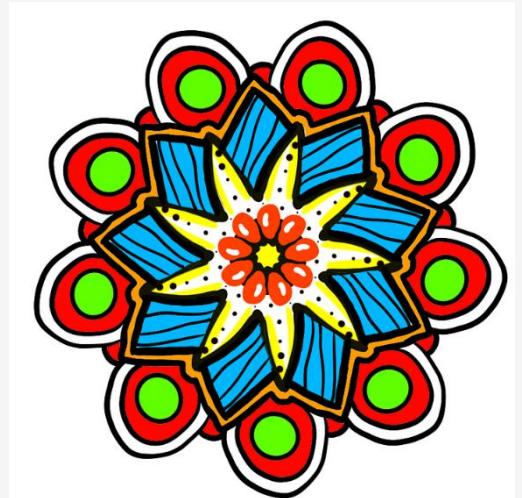




Abstrakcja

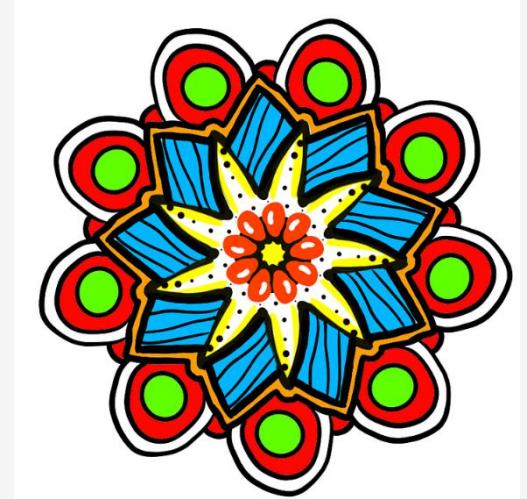
Od ogółu do szczegółu

- Modelowanie świata rzeczywistego zaczynamy od ogółu i kontynuujemy do bardziej szczegółowych postaci





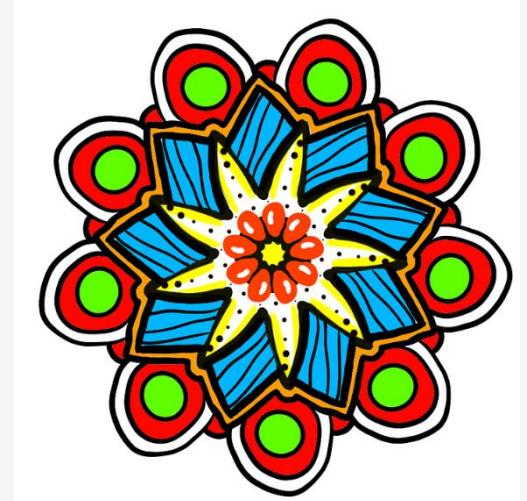
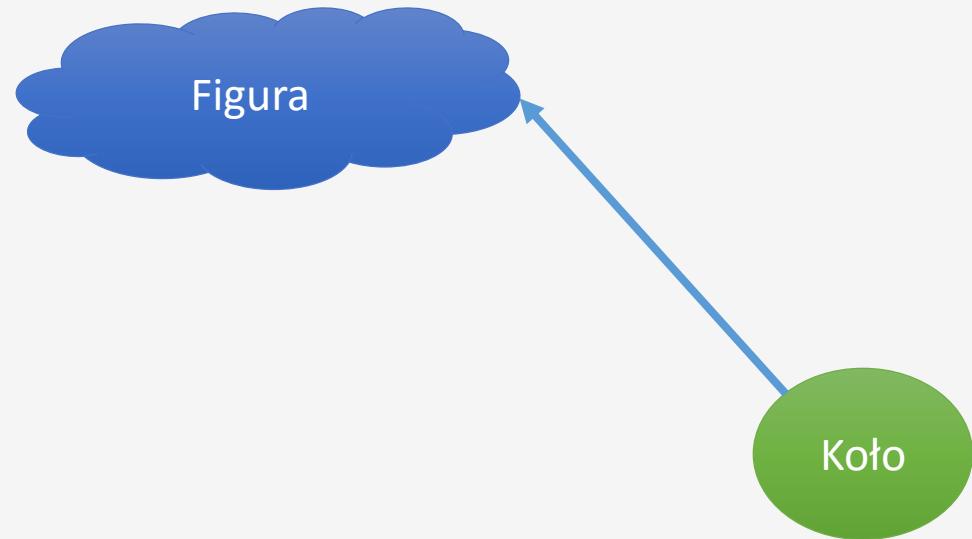
Od ogółu do szczegółu





Abstrakcja

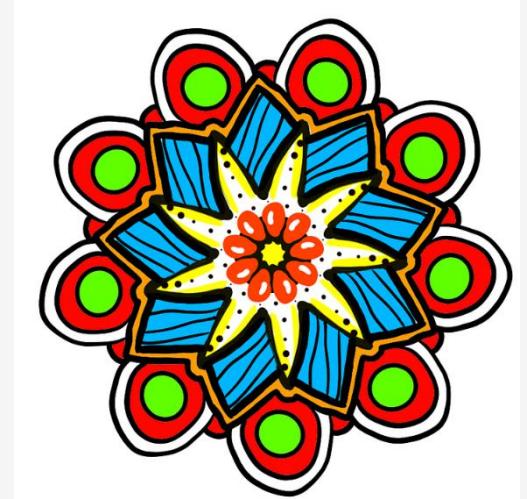
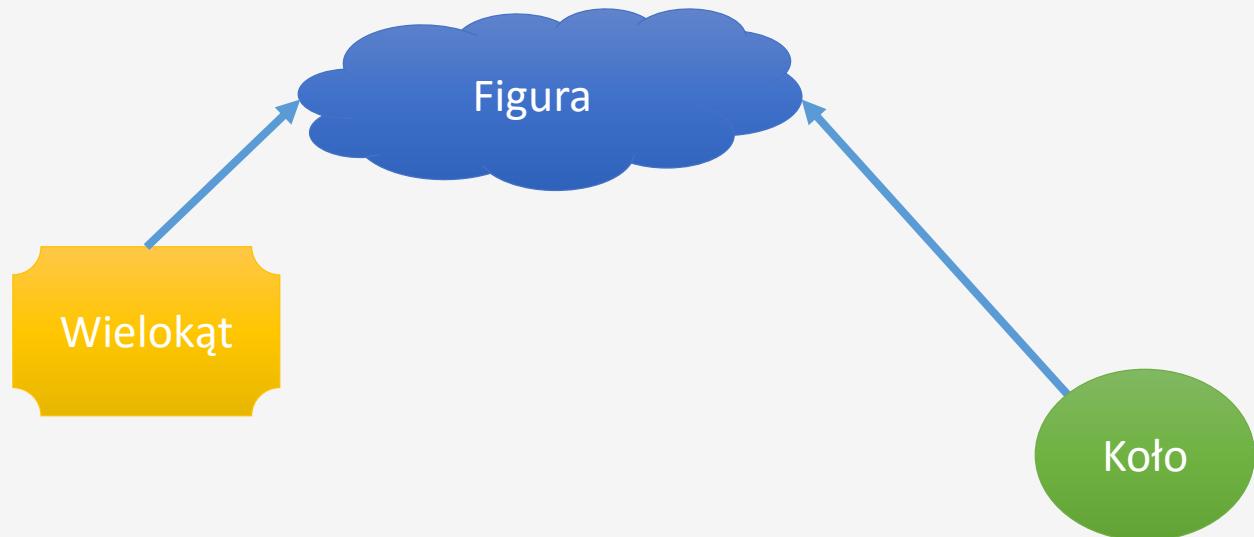
Od ogółu do szczegółu



Abstrakcja



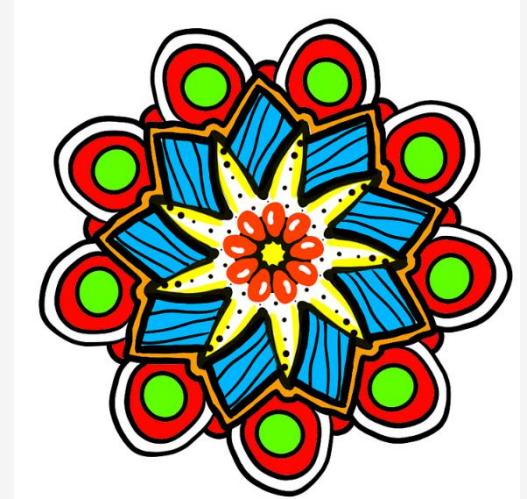
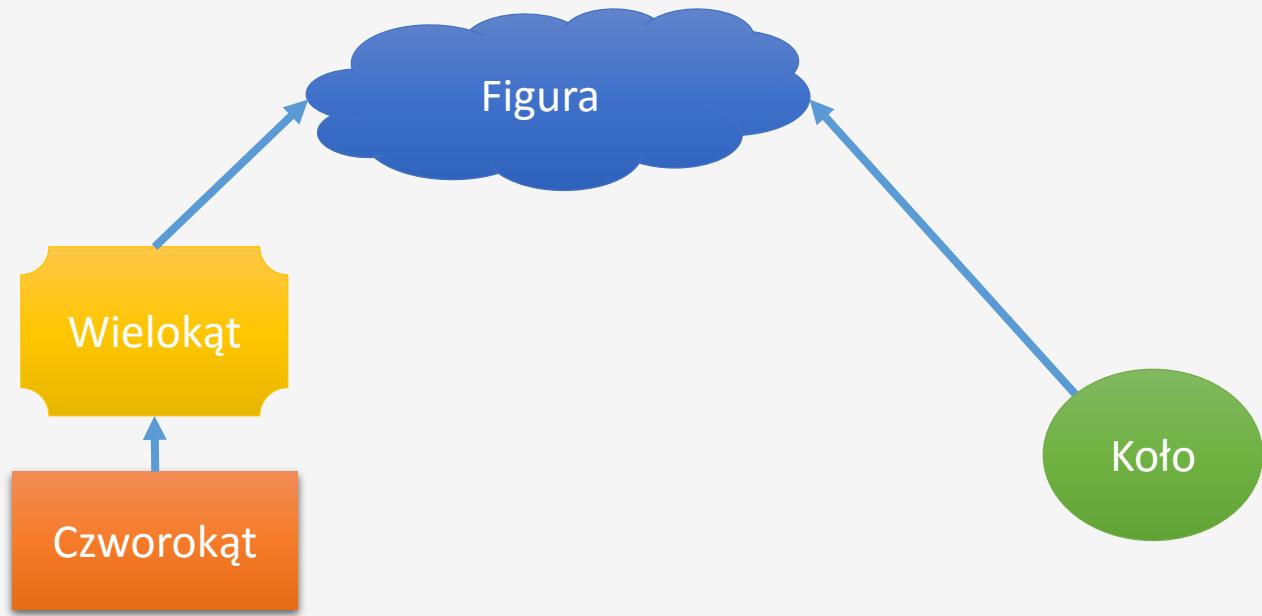
Od ogólnego do szczegółu



Abstrakcja



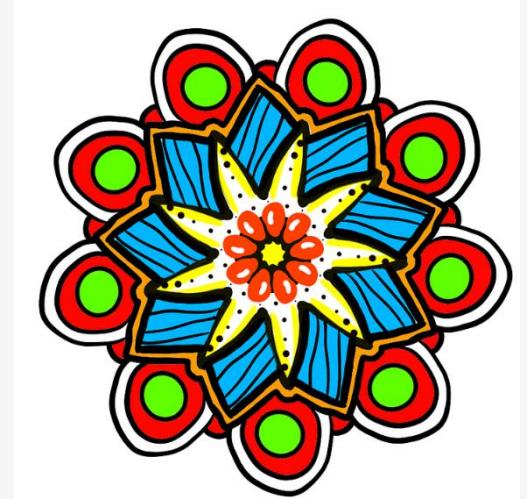
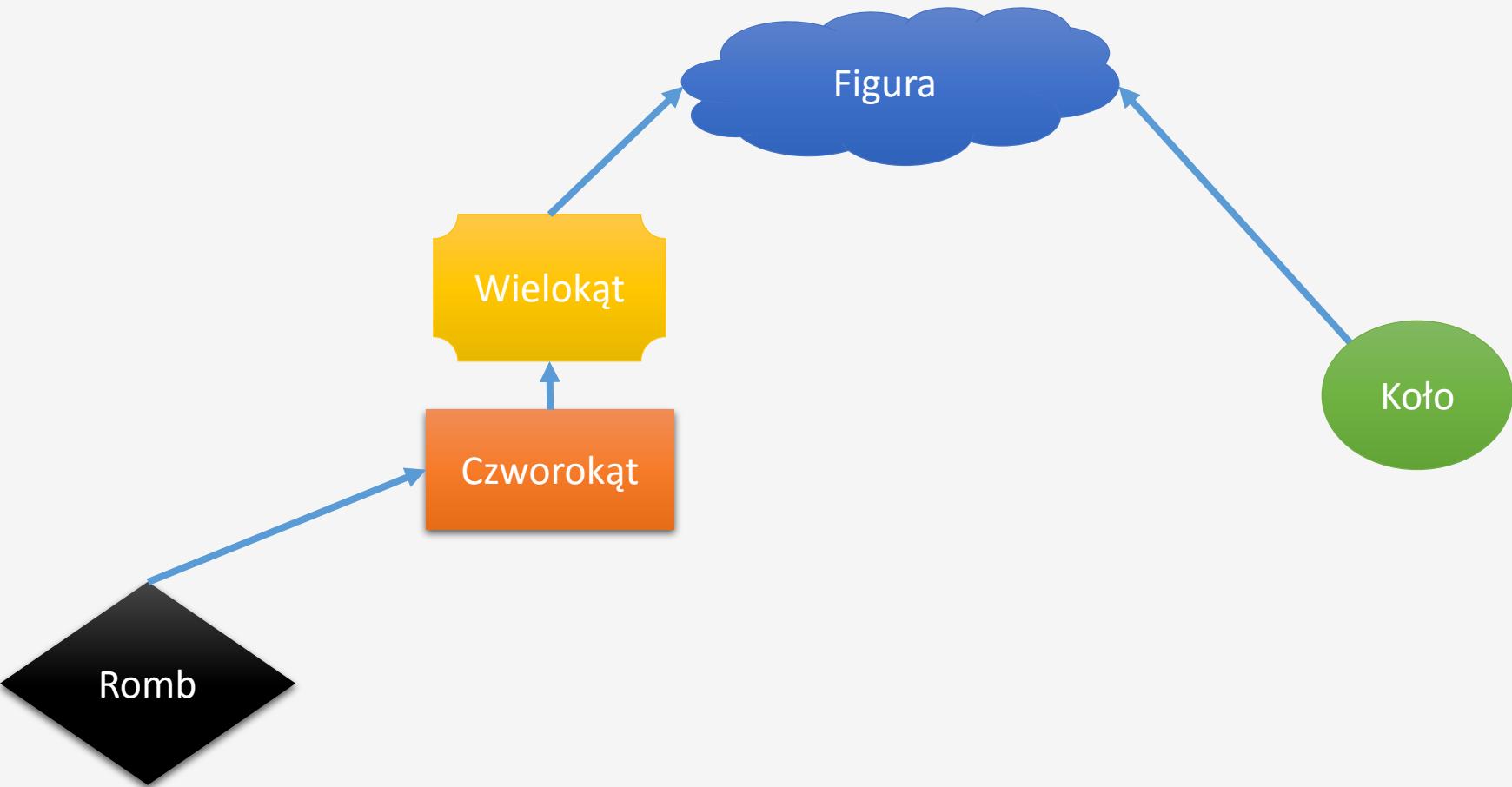
Od ogólnego do szczegółu



Abstrakcja



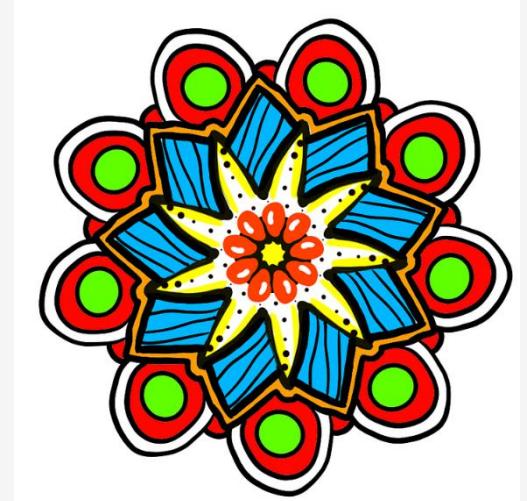
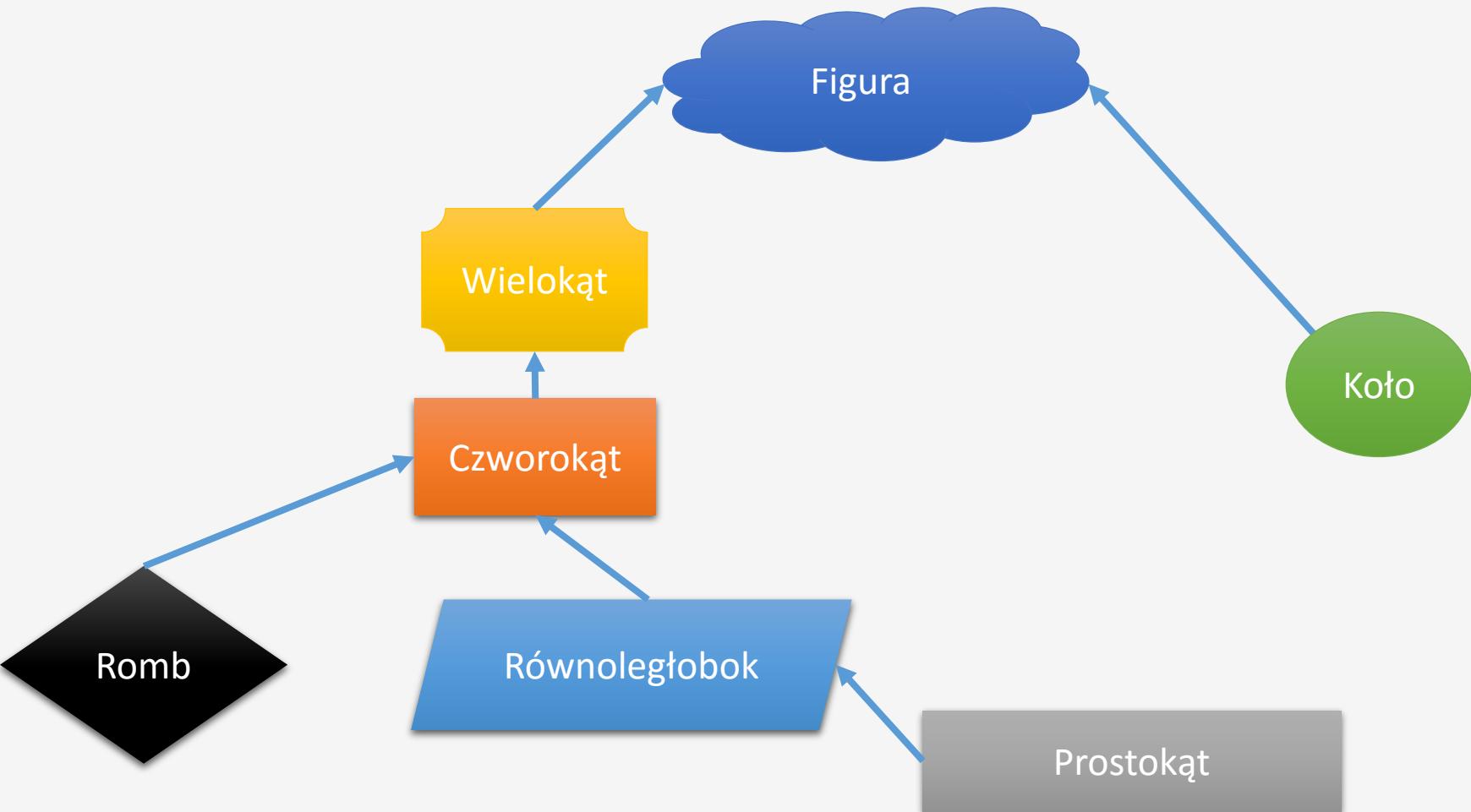
Od ogółu do szczegółu



Abstrakcja



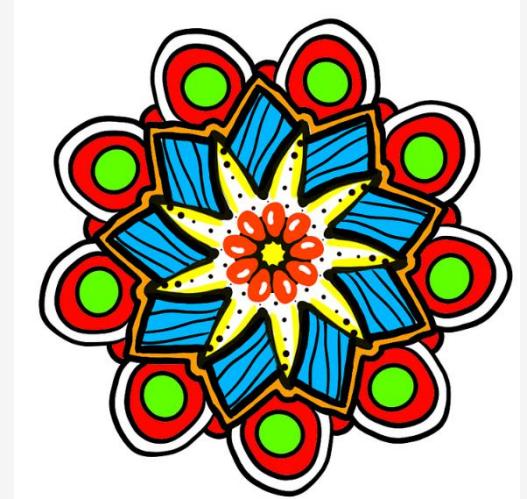
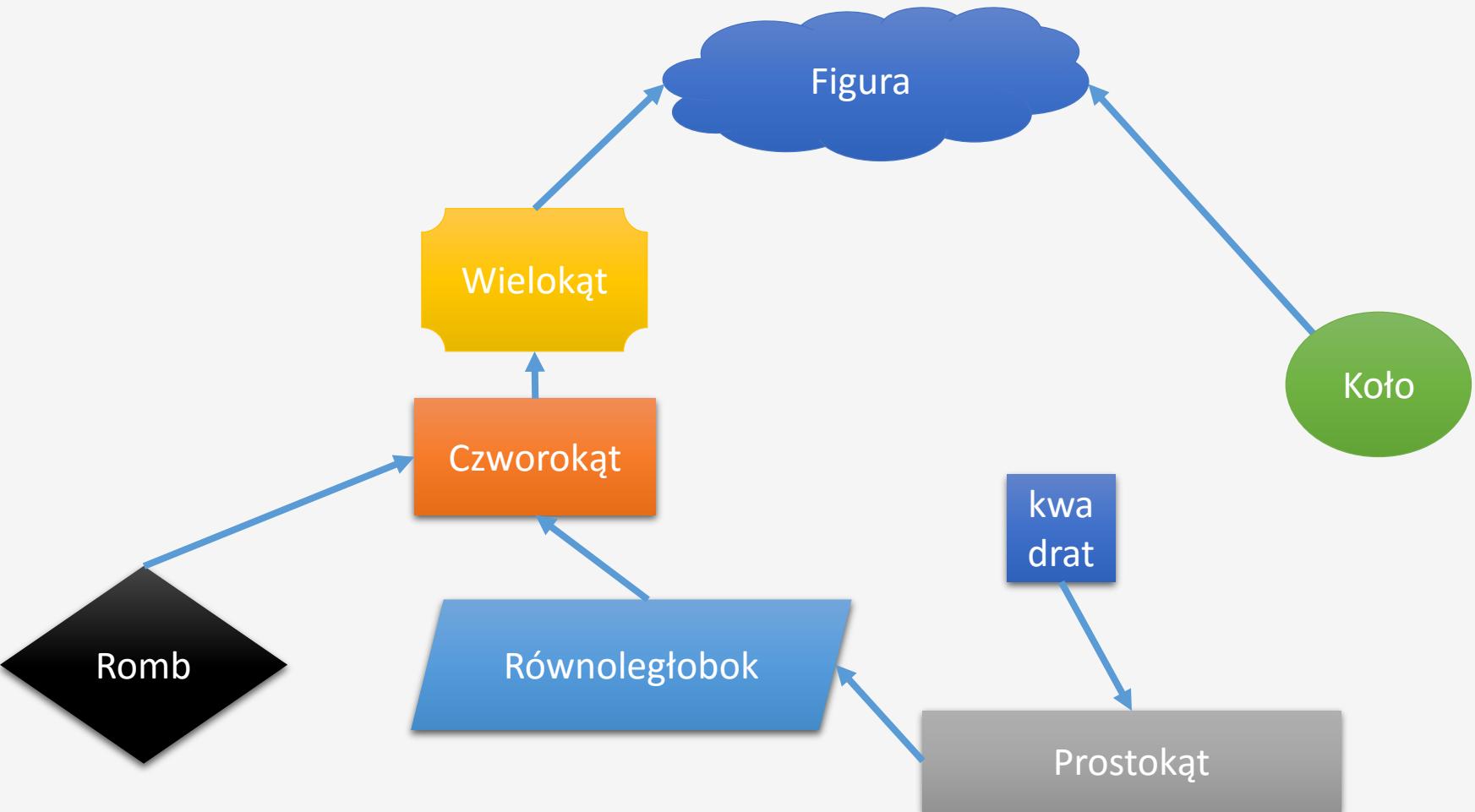
Od ogółu do szczegółu



Abstrakcja



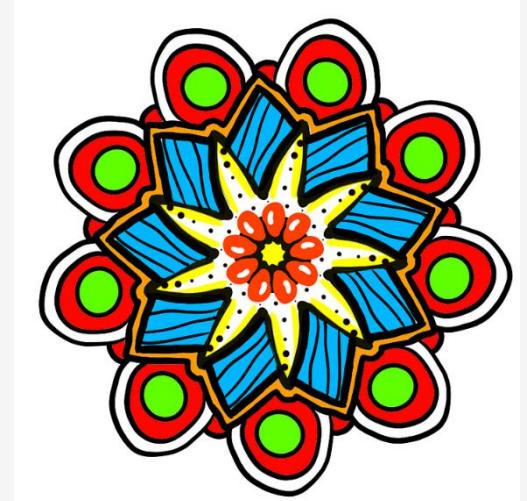
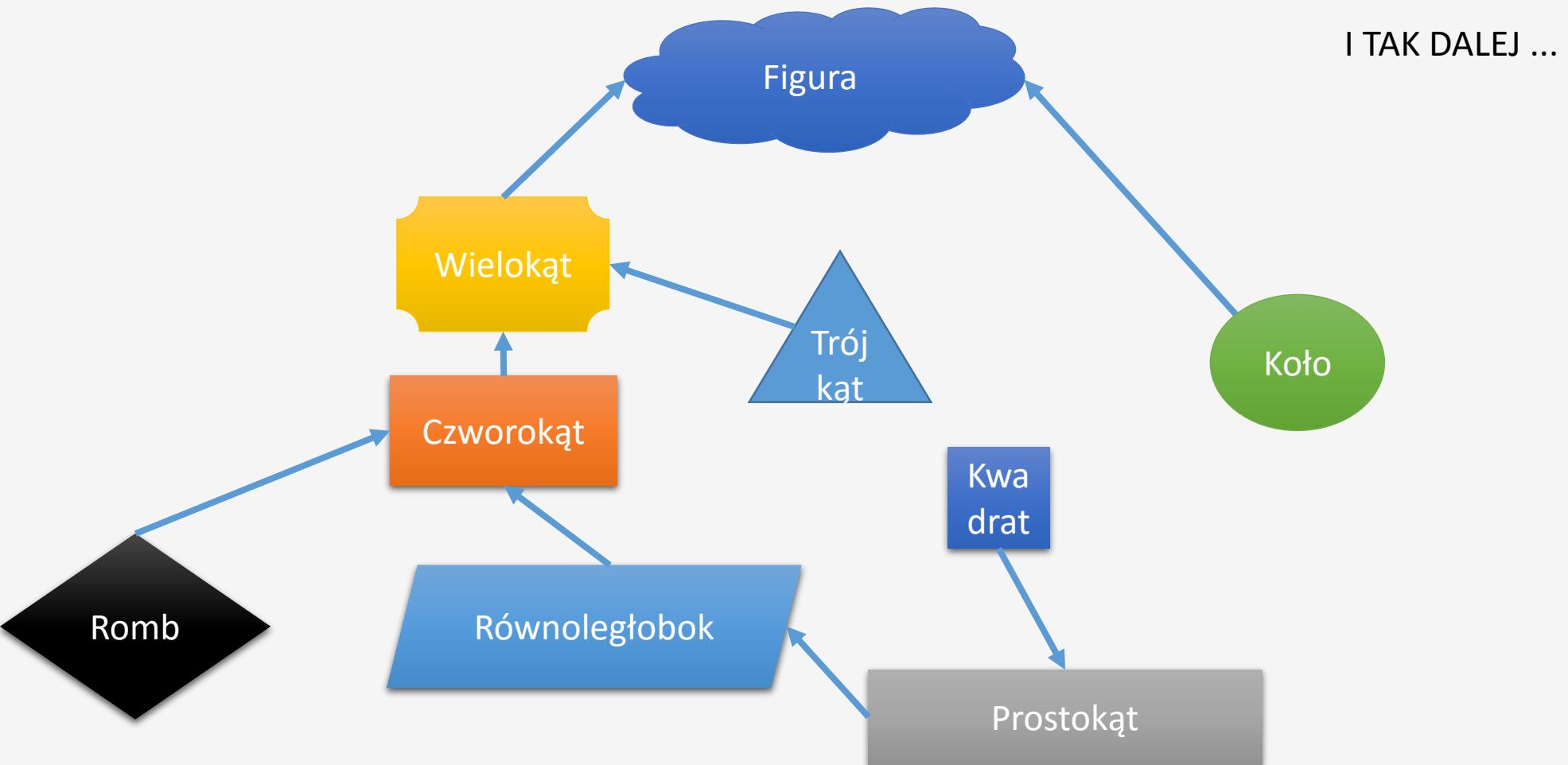
Od ogółu do szczegółu





Abstrakcja

Od ogółu do szczegółu

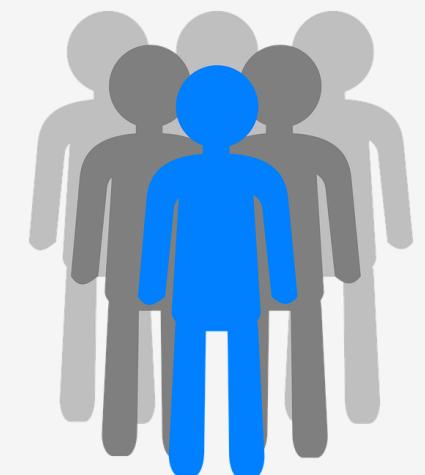




Polimorfizm

Polimorfizm (wielopostaciowość)

- Dana referencja może mieć dostęp do wielu różnych form
- Jeśli kwadrat jest prostokątem, a prostokąt jest figurą, to:
 - Kwadrat kwadrat = new Kwadrat(2); // mamy dostęp do wszystkich metod klasy Kwadrat
 - Prostokąt prostokąt = kwadrat; // mamy dostęp jedynie do wspólnych metod dla klasy Kwadrat i Prostokąt
 - Figura figura = prostokąt; // mamy dostęp jedynie do metod klasy Figura, które kwadrat odziedziczył

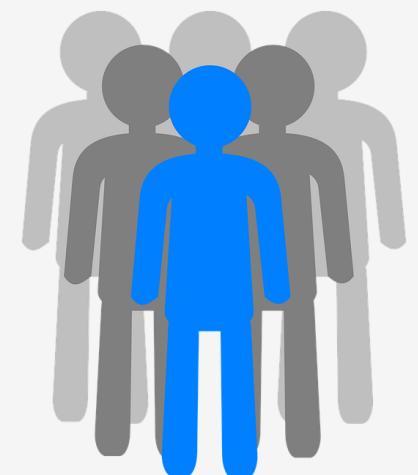




Polimorfizm

Polimorfizm (wielopostaciowość)

- W przypadku uruchomienia metody nadpisywanej przez podkласę, to rodzaj obiektu zadecyduje czy wywołana zostanie bardziej ogólna (Ogólny obiekt) czy bardziej szczegółowa wersja metody (Obiekt podklasy)



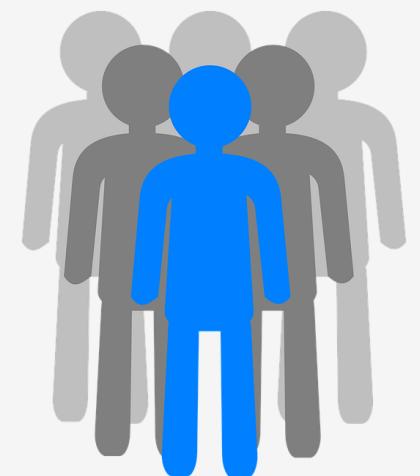


Polimorfizm

Polimorfizm (wielopostaciowość)

```
public static void main(String[] args) {  
    Osoba andrzej = new Student(imie: "Andrzej", rokUrodzenia: 1999, numerAlbumu: 55521321L);  
  
    andrzej.przedstawSie();  
}
```

Typ referencji



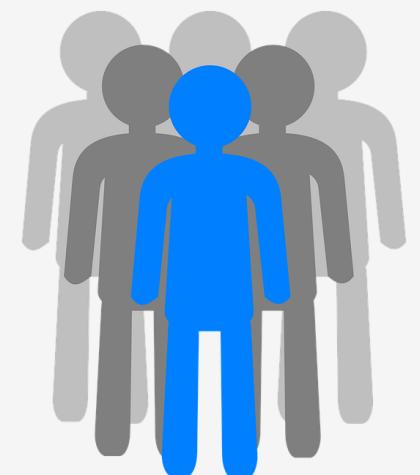


Polimorfizm

Polimorfizm (wielopostaciowość)

```
public static void main(String[] args) {  
    Osoba andrzej = new Student(imie: "Andrzej", rokUrodzenia: 1999, numerAlbumu: 55521321L);  
  
    andrzej.przedstawSie();  
}
```

Zmienna referencyjna



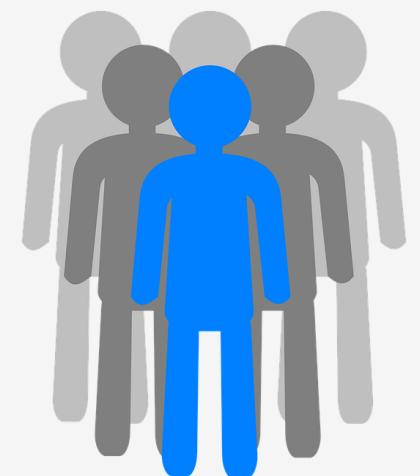


Polimorfizm

Polimorfizm (wielopostaciowość)

```
public static void main(String[] args) {  
    Osoba andrzej = new Student(imie: "Andrzej", rokUrodzenia: 1999, numerAlbumu: 55521321L);  
  
    andrzej.przedstawSie();  
}
```

Typ obiektu



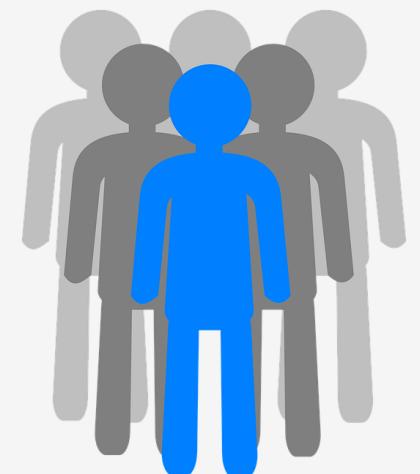


Polimorfizm

Polimorfizm (wielopostaciowość)

```
public static void main(String[] args) {  
    Osoba andrzej = new Student(imie: "Andrzej", rokUrodzenia: 1999, numerAlbumu: 55521321L);  
  
    andrzej.przedstawSie();  
}
```

Zostanie wykonana metoda zaimplementowana w
Typie obiektu, czyli Studencie





Polimorfizm – zadanie

Pan Roman prowadzi firmę malarską zajmującą się malowaniem nietypowych powierzchni. Firma notuje duże straty w związku z zakupem nieadekwatnej ilości farb w stosunku do powierzchni malowania. Pomóż firmie Romana przewidzieć realne zapotrzebowanie na farby, wiedząc, że tzw. „nietypowe” powierzchnie stanowią figury geometryczne deklarowane przez klientów jak np. koło, kwadrat, trapez itp.





Polimorfizm – zadanie

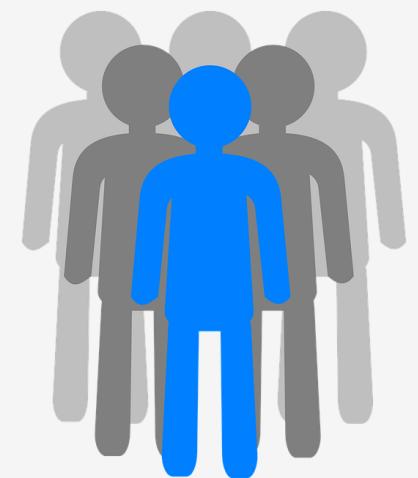
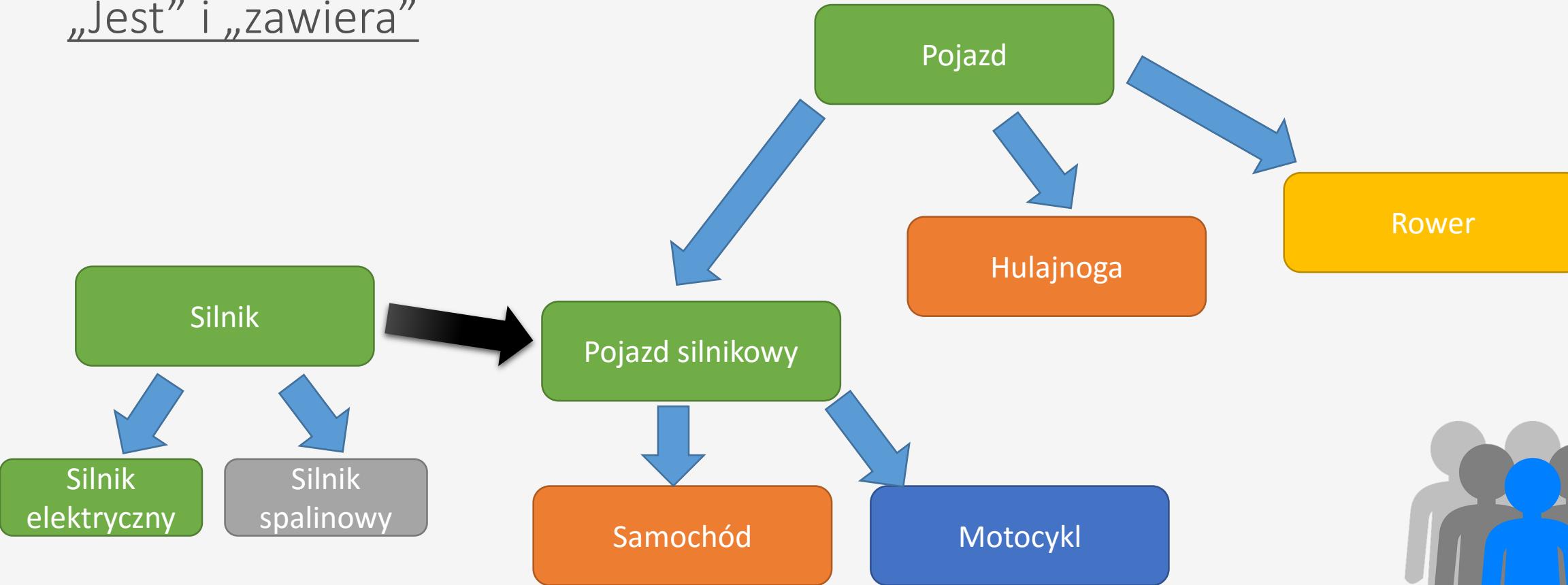
1. Utwórz klasę *SymulatorFarby*
2. Dodaj metodę publiczną statyczną :
 1. *obliczZapotrzebowanieNaFarbe():int*
 2. Metoda pobiera tablicę elementów typu *Figura* oraz wielkość pojemnika na farbę w *double*
 3. Metoda oblicza powierzchnię a następnie zakładamy, że jeśli np. powierzchnia do malowania = 200.5 a pojemność pojemnika wynosi 50, to potrzebujemy 5 pojemników aby pomalować całą powierzchnię
3. Dodaj metodę *psvm*
4. Przeprowadź symulację:
 1. Utwórz kilka obiektów typu *Kwadrat*, *Koło*, *Trapez*
 2. Wrzuć je do pojedynczej tablicy *Figura[]*
 3. Prześlij dane do *SymulatoraFarby*
 4. Sprawdź wynik





Związki pomiędzy klasami

„Jest” i „zawiera”

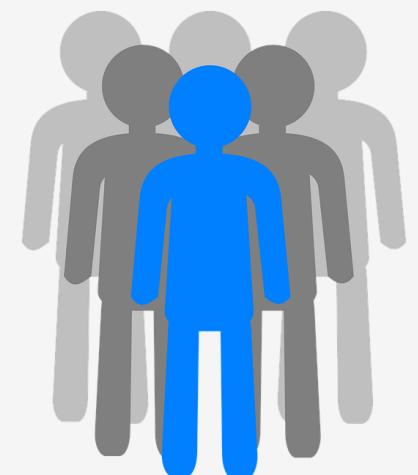
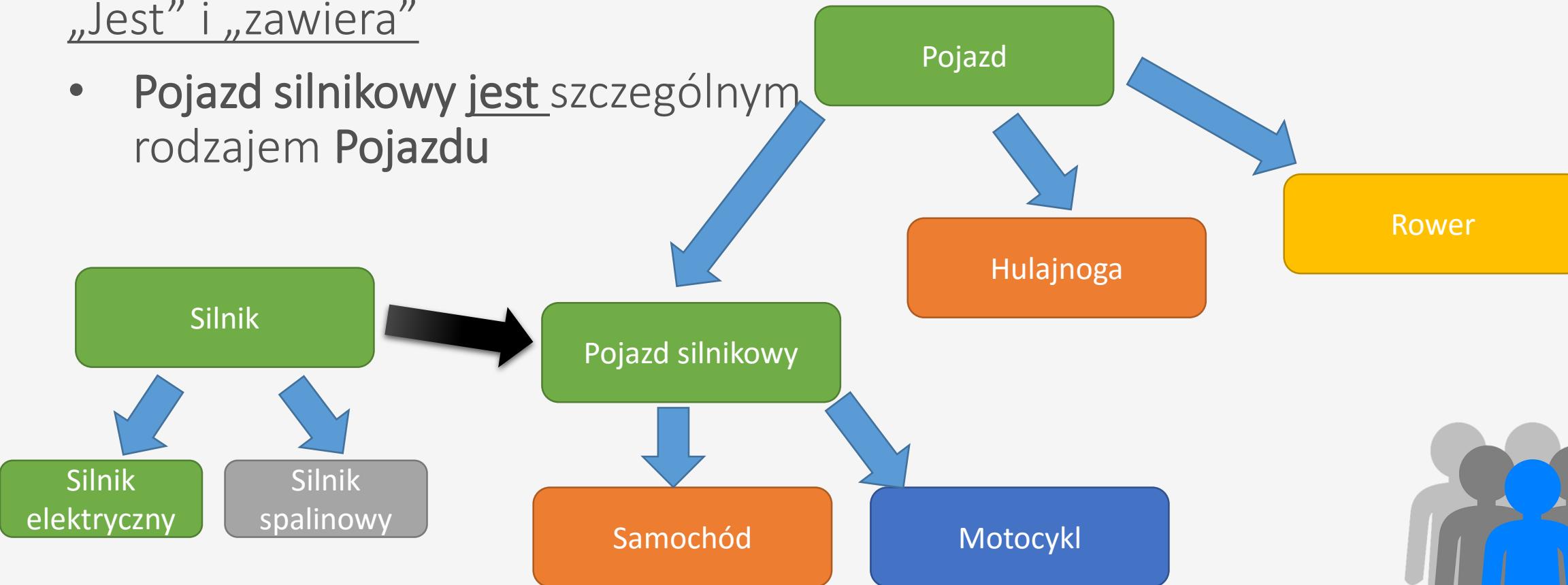




Związki pomiędzy klasami

„Jest” i „zawiera”

- Pojazd silnikowy jest szczególnym rodzajem Pojazdu

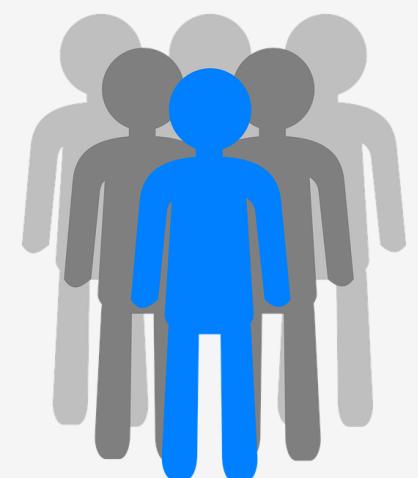
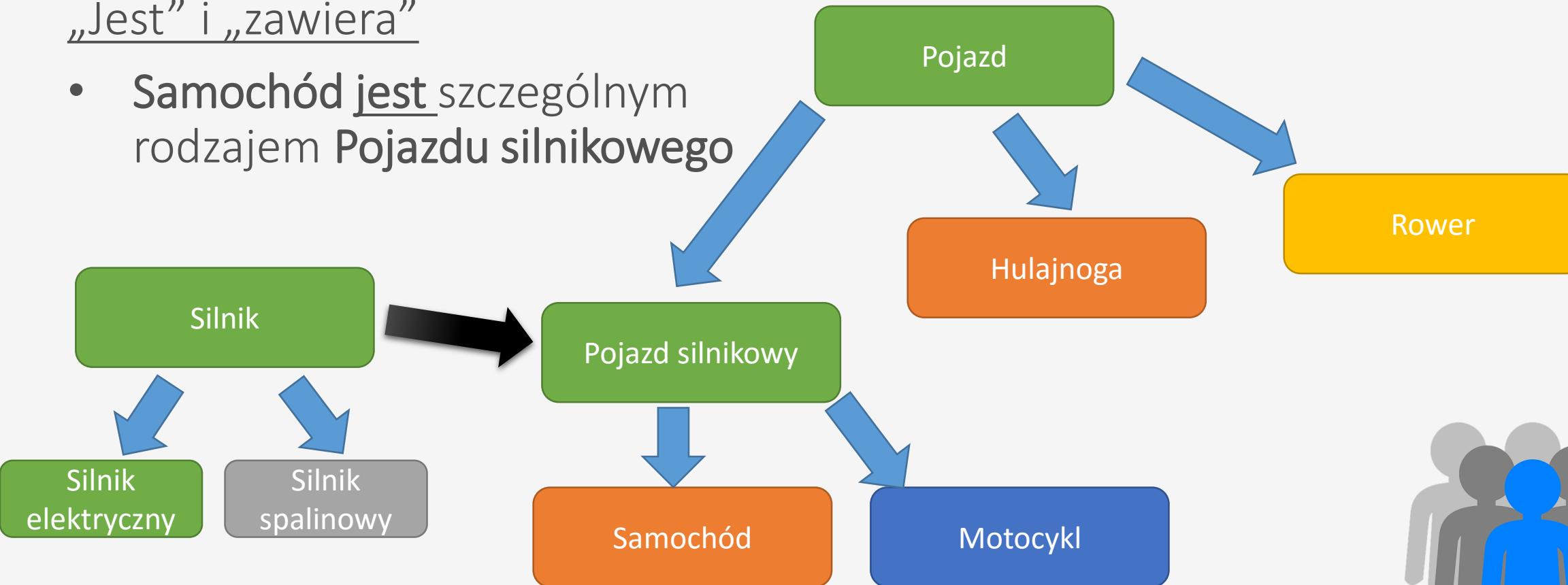




Związki pomiędzy klasami

„Jest” i „zawiera”

- Samochód jest szczególnym rodzajem Pojazdu silnikowego

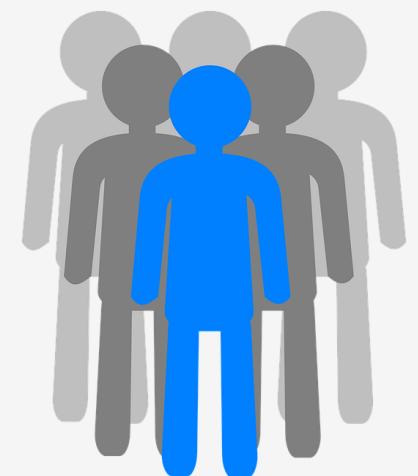
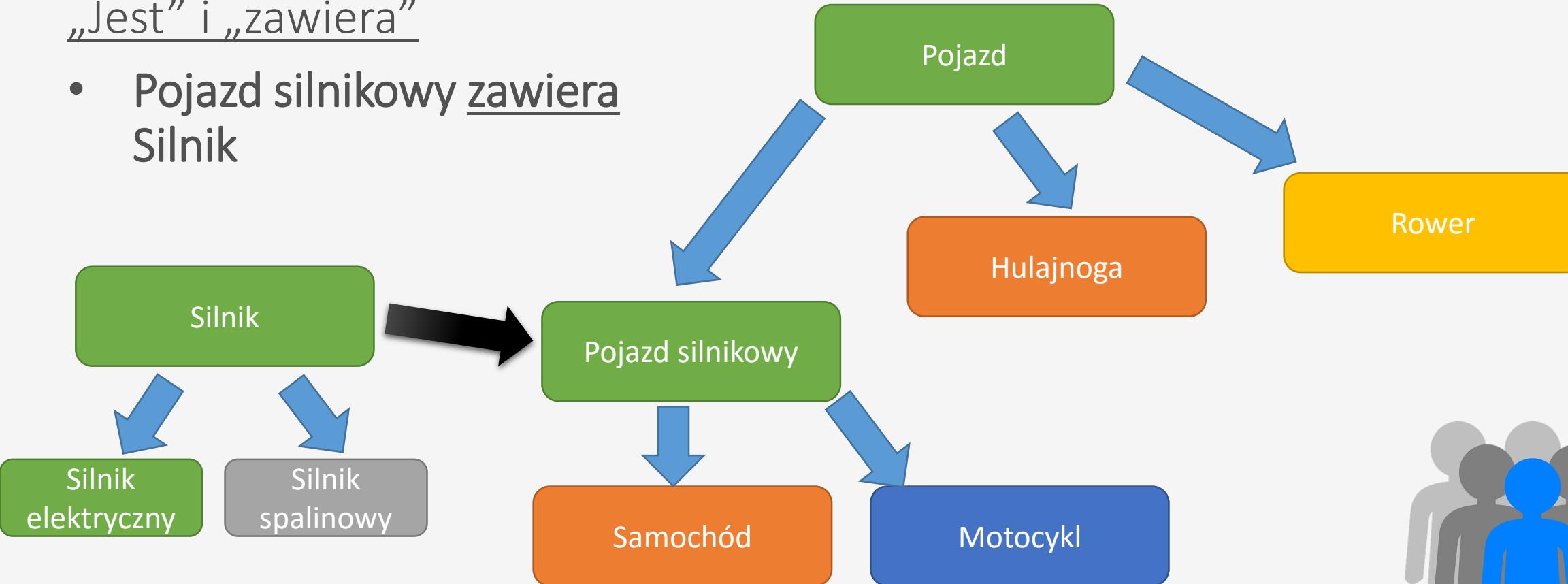




Związki pomiędzy klasami

„Jest” i „zawiera”

- Pojazd silnikowy zawiera Silnik

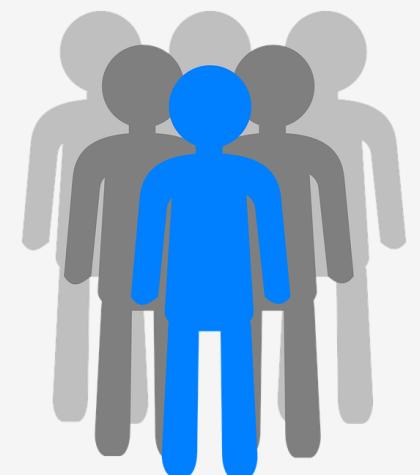




Związki pomiędzy klasami

Kompozycja

- Kompozycja reprezentuje związek „HAS-A”
- Uzyskujemy poprzez definiowanie w nowej klasie pól, które są obiektami istniejących klas
- np.
 - Ksiazka ksiazka = new Ksiazka(new Osoba(„John”, „Tolkien”), „Hobbit”);

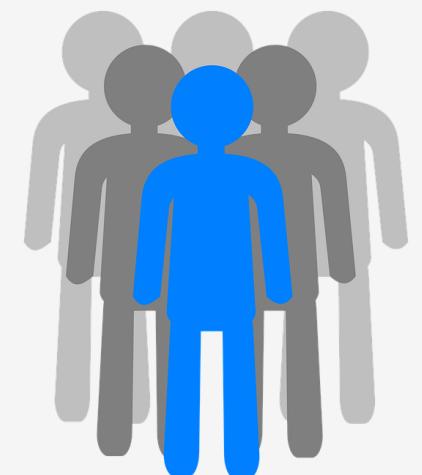




Związki pomiędzy klasami

Dziedziczenie

- Dziedziczenie reprezentuje związek „IS-A”
- Polega na przejęciu pól i metod obiektów innej klasy i ewentualnej ich modyfikacji, tak aby były bardziej szczegółowe





Bibliografia

1. <https://pixabay.com/pl/młotek-narzędzia-metalowe-celuj-w-33617/> (dostęp 07.09.2017)
2. <https://pixabay.com/pl/zwierzęta-ładny-płaski-1485100/> (dostęp 07.09.2017)
3. https://pl.wikipedia.org/wiki/Programowanie_obięktowe (dostęp 07.09.2017)
4. <https://pixabay.com/pl/pies-kość-brown-twarz-ładny-35553/> (dostęp 07.09.2017)
5. <https://pixabay.com/pl/corgi-pies-szczeniak-2026347/> (dostęp 07.09.2017)
6. <https://pixabay.com/pl/spotkanie-biznesmenów-osobowych-1219530/> (dostęp 07.09.2017)
7. <https://pixabay.com/pl/budowniczy-pracownik-budowlany-147524/> (dostęp 07.09.2017)
8. <https://pixabay.com/pl/piątka-kapsułka-czerwony-niebieski-311237/> (dostęp 07.09.2017)
9. <https://pixabay.com/pl/tarcza-rozeta-trójkątna-tarcza-31869/> (dostęp 07.09.2017)
10. <https://pixabay.com/pl/książki-stos-książek-1690753/> (dostęp 08.09.2017)
11. <https://www.journaldev.com/4098/java-heap-space-vs-stack-memory> (dostęp 08.09.2017)
12. <https://www.journaldev.com/797/what-is-java-string-pool> (dostęp 08.09.2017)
13. <https://pixabay.com/pl/śmierć-grim-reaper-reaper-kosa-2024663/> (08.09.2017)
14. <https://pixabay.com/pl/basen-pula-drabina-lato-149632/> (08.09.2017)
15. <https://pixabay.com/pl/armia-żołnierz-wojskowe-jednolite-160087/> (08.09.2017)
16. Podstawy Java (Rafał Roppel)
17. <https://pixabay.com/pl/leonardo-da-vinci-rzeźbiarz-153911/> (dostęp 08.09.2017)
18. <https://pixabay.com/pl/mandala-mandale-design-kwiat-1959687/> (dostęp 08.09.2017)
19. http://skinderowicz.pl/static/pp/zad_cw_7.pdf (dostęp 08.09.2017)
20. <https://pixabay.com/pl/ludzie-grupa-sylwetka-zespół/> (09.09.2017)
21. http://pawel.rogalinski.staff.iiar.pwr.wroc.pl/dydaktyka/INE2018L_JP3_Java/Kompozycja_i_dziedziczenie.pdf (09.09.2017)