

Stefan Sokołowski

PROGRAMOWANIE OBIEKTOWE II

wykład 1

Inst. Informatyki Stosowanej, ANS

Elbląg, 2024/2025

Nie jestem biegły w Jawie...



Nie jestem biegły w Jawie...



**Proszę nie mówić „nie wiem”
proszę szukać!**

Nie jestem biegły w Jawie...



**Proszę nie mówić „nie wiem”
proszę szukać!**

Co robię, kiedy nie wiem?

- **internet** — na przykład:

<https://docs.oracle.com/en/java/javase/11/docs/api/>

Nie jestem biegły w Jawie...



**Proszę nie mówić „nie wiem”
proszę szukać!**

Co robię, kiedy nie wiem?

- **internet** — na przykład:

<https://docs.oracle.com/en/java/javase/11/docs/api/>

(literatury na temat Jawy jest w sieci mnóstwo)

Nie jestem biegły w Jawie...



**Proszę nie mówić „nie wiem”
proszę szukać!**

Co robię, kiedy nie wiem?

- **internet** — na przykład:
<https://docs.oracle.com/en/java/javase/11/docs/api/>
(literatury na temat Jawy jest w sieci mnóstwo)
- **eksperymenty** z kompilatorem

Nie jestem biegły w Jawie...

Przykład: (jak znaleźć informacje o funkcjach **matematycznych**)

Nie jestem biegły w Jawie...

Przykład: (jak znaleźć informacje o funkcjach **matematycznych**)

Google: `java 11 math`

Nie jestem biegły w Jawie...

Przykład: (jak znaleźć informacje o funkcjach **matematycznych**)

Google: **java 11 math** *to prowadzi do:*

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Math.html>

Nie jestem biegły w Jawie...

Przykład: (jak znaleźć informacje o funkcjach matematycznych)

Google: **java 11 math** to prowadzi do:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Math.html>

Class Math

java.lang.Math

...

Fields

static double *E*

...

static double *PI*

...

Methods

static double *abs*(double *a*)

...

static float *abs*(float *a*)

...

Nie jestem biegły w Jawie...

Przykład: (jak znaleźć informacje o funkcjach matematycznych)

Google: **java 11 math** to prowadzi do:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Math.html>

Class Math

java.lang.Math

...

Fields

static double *E*

...

static double *PI*

...

Methods

static double *abs*(double *a*)

...

static float *abs*(float *a*)

...

nazwa klasy

Nie jestem biegły w Jawie...

Przykład: (jak znaleźć informacje o funkcjach matematycznych)

Google: **java 11 math** to prowadzi do:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Math.html>

Class Math

java.lang.Math

nazwa klasy

co importować

...

Fields

static double *E*

...

static double *PI*

...

Methods

static double *abs*(double a)

...

static float *abs*(float a)

...

Nie jestem biegły w Jawie...

Przykład: (jak znaleźć informacje o funkcjach matematycznych)

Google: **java 11 math** to prowadzi do:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Math.html>

Class Math

java.lang.Math

...

Fields

static double *E*

...

static double *PI*

...

Methods

static double *abs*(double a)

...

static float *abs*(float a)

...

nazwa klasy

co importować

OPIS KLASY

Nie jestem biegły w Jawie...

Przykład: (jak znaleźć informacje o funkcjach matematycznych)

Google: **java 11 math** to prowadzi do:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Math.html>

Class Math

java.lang.Math

...

Fields

static double *E*

...

static double *PI*

...

Methods

static double *abs*(double a)

...

static float *abs*(float a)

...

nazwa klasy

co importować

OPIS KLASY

Spis własności

Nie jestem biegły w Jawie...

Przykład: (jak znaleźć informacje o funkcjach **matematycznych**)

Google: **java 11 math** to prowadzi do:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Math.html>

Class Math	← nazwa klasy
java.lang.Math	← co importować
...	← OPIS KLASY
Fields	← Spis własności
static double <i>E</i>	
...	
static double <i>PI</i>	
...	
Methods	← Spis metod
static double <i>abs</i> (double a)	
...	
static float <i>abs</i> (float a)	
...	

Uwaga: metody o tej samej nazwie ale różnych (nieidentycznych) listach parametrów są **różne**.

Nie jestem biegły w Jawie...

Przykład: (jak znaleźć informacje o wczytywaniu z **wejścia**)

Nie jestem biegły w Jawie...

Przykład: (jak znaleźć informacje o wczytywaniu z wejścia)

Google: `java 11 scanner` to prowadzi do:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Scanner.html>

Nie jestem biegły w Jawie...

Przykład: (jak znaleźć informacje o wczytywaniu z **wejścia**)

Google: **java 11 scanner** to prowadzi do:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Scanner.html>

Class Scanner

java.util.Scanner

...

Constructors

Scanner (File source)

...

Scanner (File source, String charsetName)

...

Methods

boolean *hasNext()*

...

boolean *hasNextInt()*

...

Nie jestem biegły w Jawie...

Przykład: (jak znaleźć informacje o wczytywaniu z **wejścia**)

Google: **java 11 scanner** to prowadzi do:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Scanner.html>

Class Scanner

java.util.Scanner

...

Constructors

Scanner (File source)

...

Scanner (File source, String charsetName)

...

Methods

boolean **hasNext()**

...

boolean **hasNextInt()**

...

nazwa klasy

co importować

OPIS KLASY

Nie jestem biegły w Jawie...

Przykład: (jak znaleźć informacje o wczytywaniu z **wejścia**)

Google: **java 11 scanner** to prowadzi do:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Scanner.html>

Class Scanner

java.util.Scanner

...

Constructors

Scanner (File source)

...

Scanner (File source, String charsetName)

...

Methods

boolean **hasNext()**

...

boolean **hasNextInt()**

...

nazwa klasy

co importować

OPIS KLASY

Spis konstruktorów

Uwaga: konstruktory o różnych (nieidentycznych) listach parametrów są **różne**.

Nie jestem biegły w Jawie...

Przykład: (jak znaleźć informacje o wczytywaniu z **wejścia**)

Google: **java 11 scanner** to prowadzi do:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Scanner.html>

Class Scanner

java.util.Scanner

...

Constructors

Scanner (File source)

...

Scanner (File source, String charsetName)

...

Methods

boolean **hasNext()**

...

boolean **hasNextInt()**

...

nazwa klasy

co importować

OPIS KLASY

Spis konstruktorów

Uwaga: konstruktory o różnych (nieidentycznych) listach parametrów są **różne**.

Spis metod

Uwaga: metody o tej samej nazwie ale różnych (nieidentycznych) listach parametrów są **różne**.

Przypominajka o klasach i obiektach

Obiekt — zestaw wartości (t.zw. *własności*) i funkcji (t.zw. *metod*)

Przypominajka o klasach i obiektach

Obiekt — zestaw wartości (t.zw. *własności*) i funkcji (t.zw. *metod*)

Klasa — opis tego, co powinno być w obiekcie
zbiór obiektów tej samej natury

Przypominajka o klasach i obiektach

Obiekt — zestaw wartości (t.zw. *własności*) i funkcji (t.zw. *metod*)

Klasa — opis tego, co powinno być w obiekcie
zbiór obiektów tej samej natury

obiekt tak się ma do **klasy**
jak **element** do **zbioru**

Przypominajka o klasach i obiektach

Obiekt — zestaw wartości (t.zw. *własności*) i funkcji (t.zw. *metod*)

Klasa — opis tego, co powinno być w obiekcie
zbiór obiektów tej samej natury

obiekt tak się ma do **klasy**
jak **element** do **zbioru**

Przykład:

Klasa:

```
class Zespolona {  
    double czesc_rzeczywista;  
    double czesc_urojona;  
}
```

Przypominajka o klasach i obiektach

Obiekt — zestaw wartości (t.zw. *własności*) i funkcji (t.zw. *metod*)

Klasa — opis tego, co powinno być w obiekcie
zbiór obiektów tej samej natury

obiekt tak się ma do **klasy**
jak **element** do **zbioru**

Przykład:

Klasa:

```
class Zespolona {  
    double czesc_rzeczywista;  
    double czesc_urojona;  
}
```

obiekt: $1 + i$

Przypominajka o klasach i obiektach

Obiekt — zestaw wartości (t.zw. *własności*) i funkcji (t.zw. *metod*)

Klasa — opis tego, co powinno być w obiekcie
zbiór obiektów tej samej natury

obiekt tak się ma do **klasy**
jak **element** do **zbioru**

Przykład:

Klasa:

```
class Zespolona {  
    double czesc_rzeczywista;  
    double czesc_urojona;  
}
```

obiekt: $1 + i$

obiekt: $2.31 + 3.14 \cdot i$

Przypominajka o klasach i obiektach

Obiekt — zestaw wartości (t.zw. *własności*) i funkcji (t.zw. *metod*)

Klasa — opis tego, co powinno być w obiekcie
zbiór obiektów tej samej natury

obiekt tak się ma do **klasy**
jak **element** do **zbioru**

Przykład:

Klasa:

```
class Zespolona {  
    double czesc_rzeczywista;  
    double czesc_urojona;  
}
```

obiekt: $1 + i$

obiekt: $2.31 + 3.14 \cdot i$

obiekt: $35.0 - 0.72 \cdot i$

Analogia anatomiczna: publiczne a prywatne

pola klasy
(własności i metody)



części funkcjonalne
organizmu

Analogia anatomiczna: publiczne a prywatne

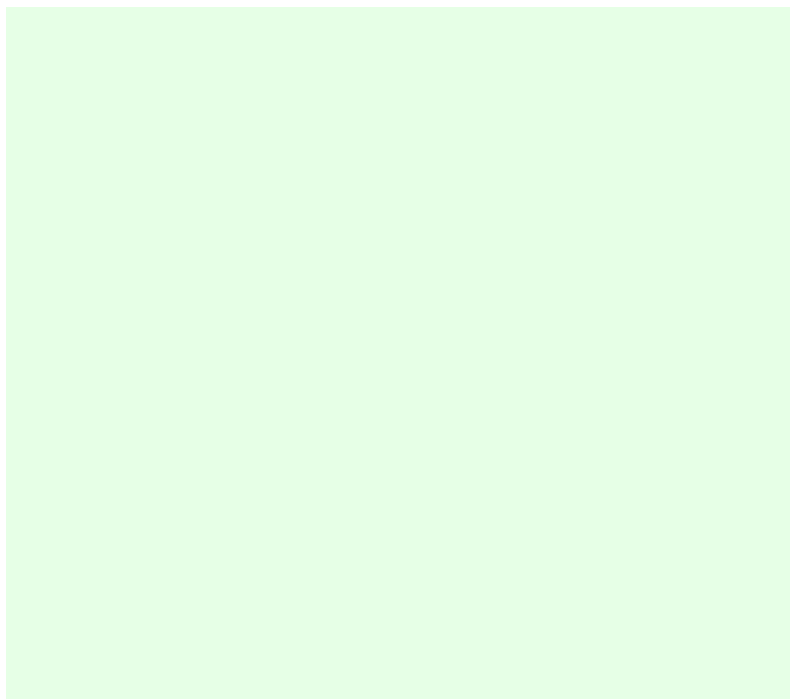
pola klasy
(własności i metody)



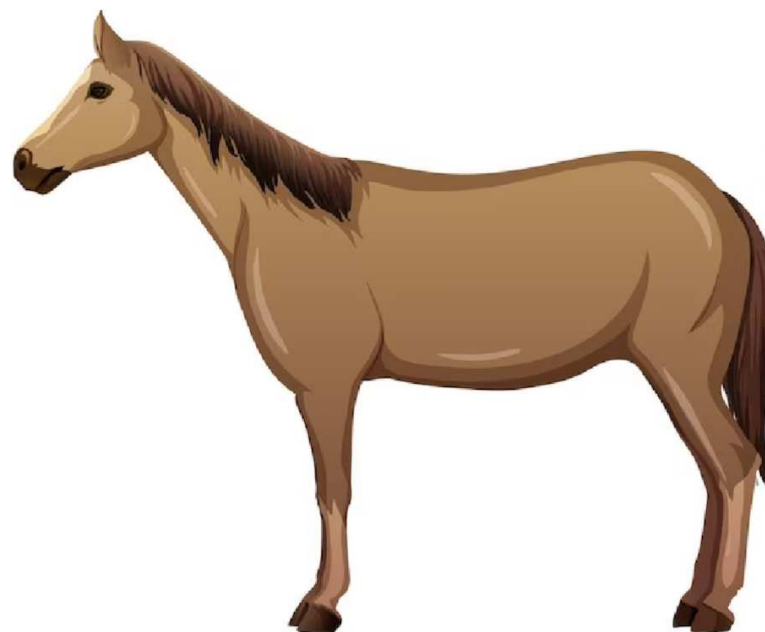
części funkcjonalne
organizmu

Przykład: (koń)

Klasa:



Obiekt:



Analogia anatomiczna: publiczne a prywatne

pola klasy
(własności i metody)



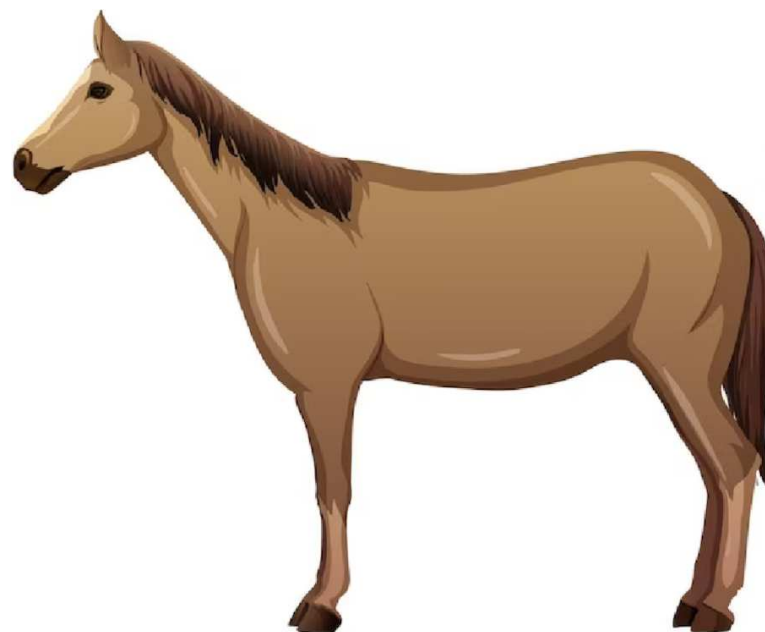
części funkcjonalne
organizmu

Przykład: (koń)

Klasa:

własność: nogi

Obiekt:



Analogia anatomiczna: publiczne a prywatne

pola klasy
(własności i metody)



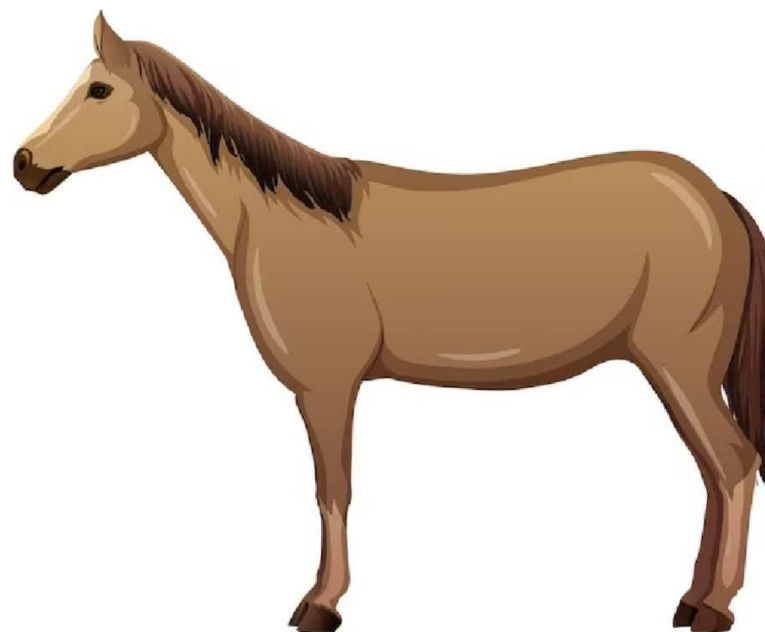
części funkcjonalne
organizmu

Przykład: (koń)

Klasa:

własność: *nogi*
własność: *kopyta*

Obiekt:



Analogia anatomiczna: publiczne a prywatne

pola klasy
(własności i metody)



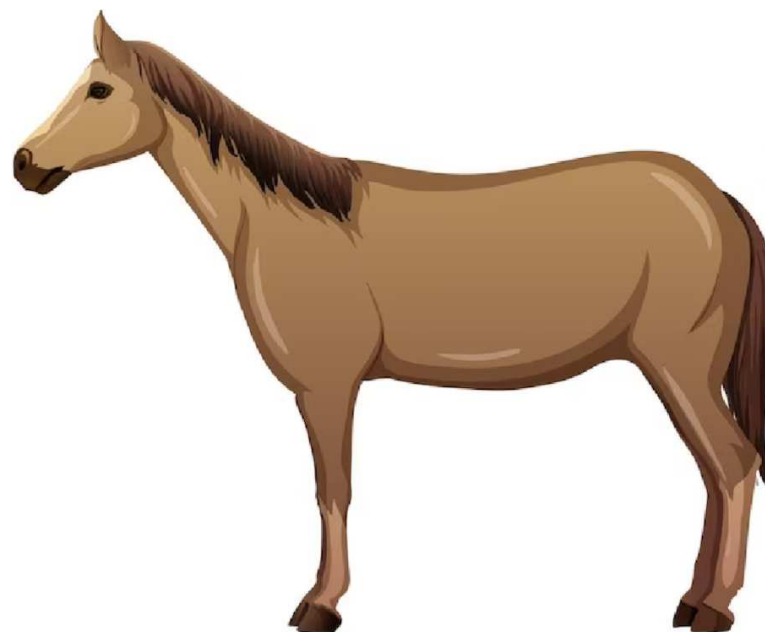
części funkcjonalne
organizmu

Przykład: (koń)

Klasa:

własność: ***nogi***
własność: ***kopyta***
własność: ***ogon***

Obiekt:



Analogia anatomiczna: publiczne a prywatne

pola klasy
(własności i metody)



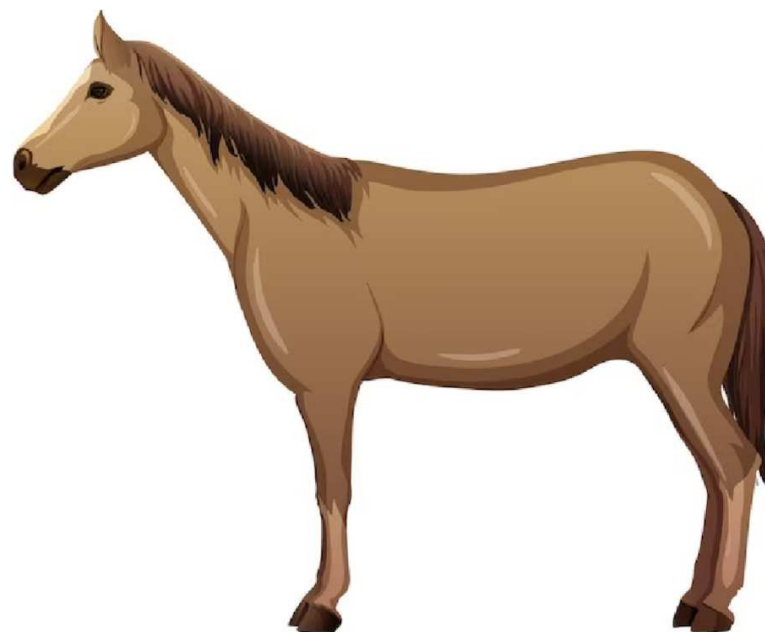
części funkcjonalne
organizmu

Przykład: (koń)

Klasa:

własność: ***nogi***
własność: ***kopyta***
własność: ***ogon***
metoda: ***galop***

Obiekt:



Analogia anatomiczna: publiczne a prywatne

pola klasy
(własności i metody)



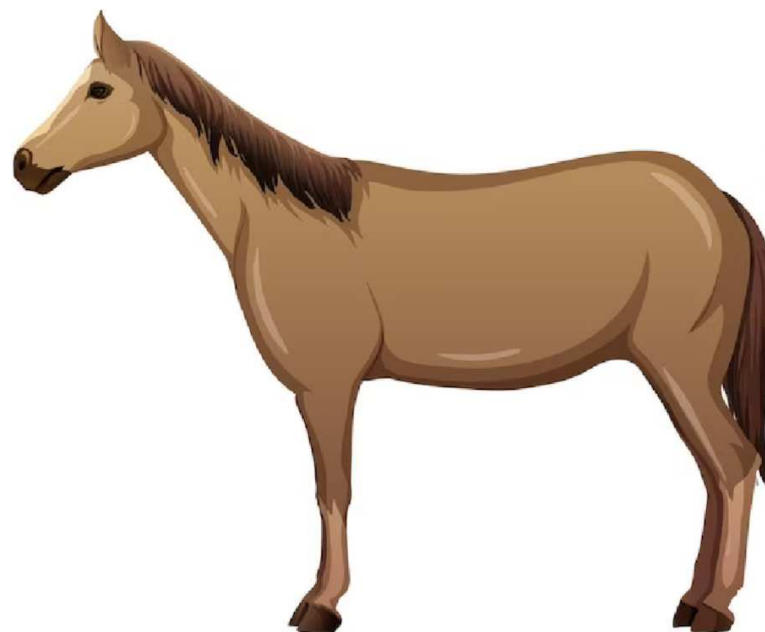
części funkcjonalne
organizmu

Przykład: (koń)

Klasa:

własność: ***nogi***
własność: ***kopyta***
własność: ***ogon***
metoda: ***galop***
metoda: ***stęp***

Obiekt:



Analogia anatomiczna: publiczne a prywatne

pola klasy
(własności i metody)



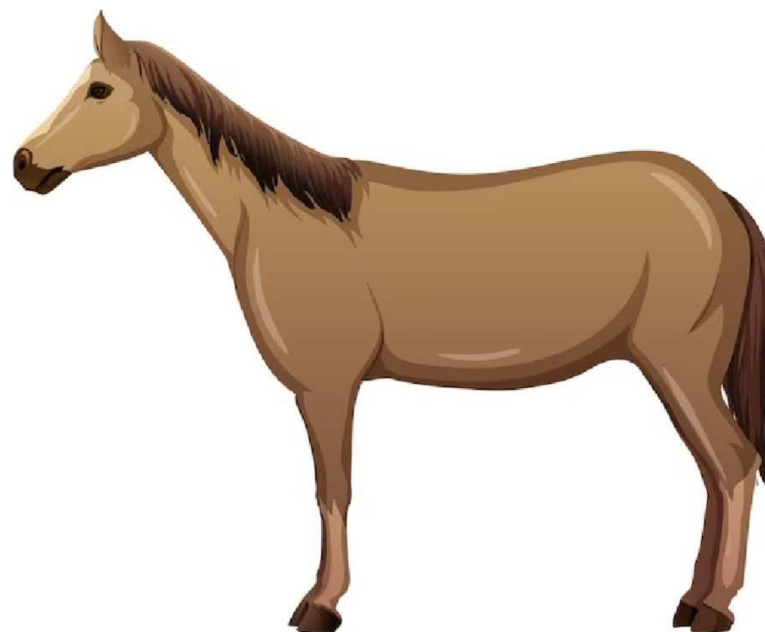
części funkcjonalne
organizmu

Przykład: (koń)

Klasa:

własność: ***nogi***
własność: ***kopyta***
własność: ***ogon***
metoda: ***galop***
metoda: ***stęp***
metoda: ***skubanie trawy***

Obiekt:



Analogia anatomiczna: publiczne a prywatne

pola klasy
(własności i metody)



części funkcjonalne
organizmu

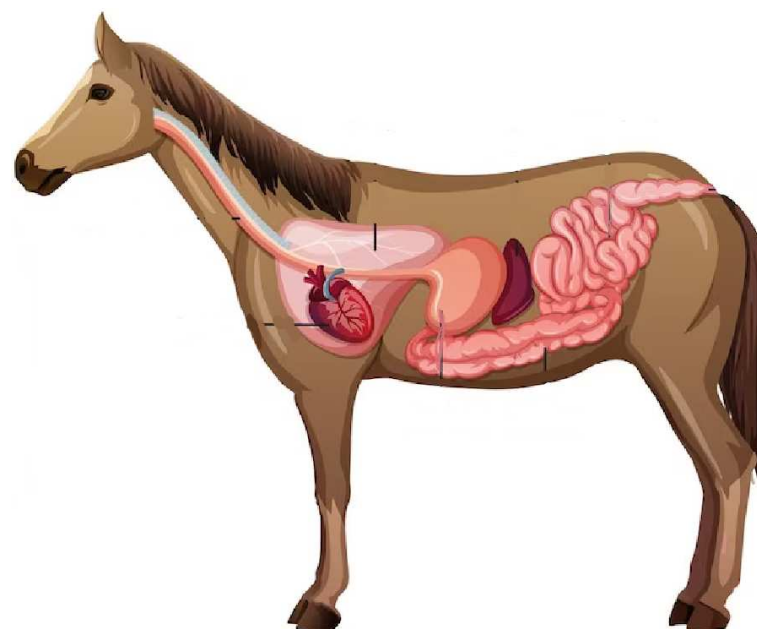
Przykład: (koń)

Klasa:

Obiekt:

publiczne

własność: *nogi*
własność: *kopyta*
własność: *ogon*
metoda: *galop*
metoda: *stęp*
metoda: *skubanie trawy*
własność: *przełyk*



Analogia anatomiczna: publiczne a prywatne

pola klasy
(własności i metody)



części funkcjonalne
organizmu

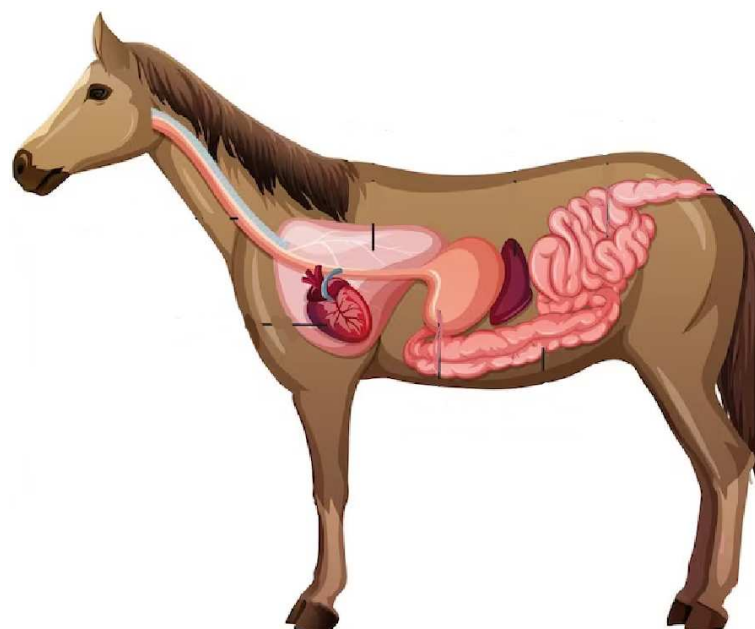
Przykład: (koń)

Klasa:

Obiekt:

publiczne

własność: *nogi*
własność: *kopyta*
własność: *ogon*
metoda: *galop*
metoda: *stęp*
metoda: *skubanie trawy*
własność: *przełyk*
własność: *serce*



Analogia anatomiczna: publiczne a prywatne

pola klasy
(własności i metody)



części funkcjonalne
organizmu

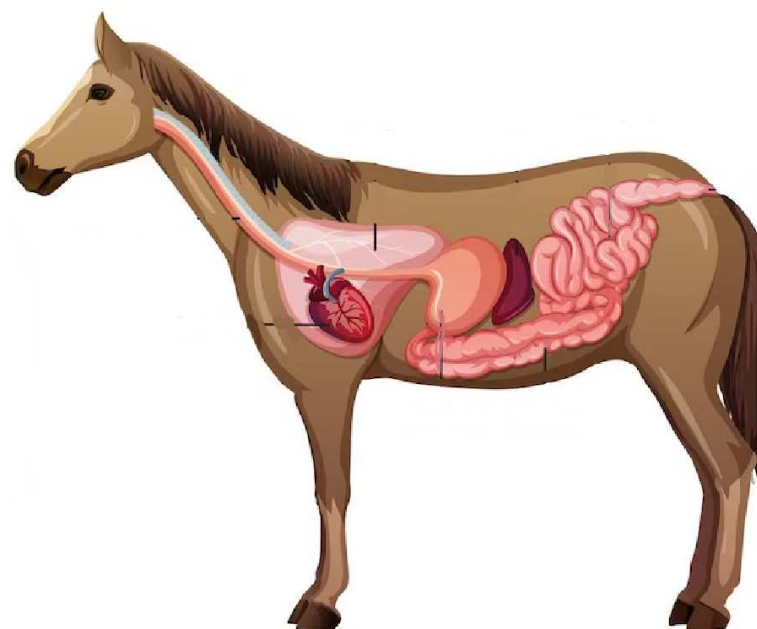
Przykład: (koń)

Klasa:

Obiekt:

publiczne

własność: *nogi*
własność: *kopyta*
własność: *ogon*
metoda: *galop*
metoda: *stęp*
metoda: *skubanie trawy*
własność: *przełyk*
własność: *serce*
własność: *żołądek*



Analogia anatomiczna: publiczne a prywatne

pola klasy
(własności i metody)



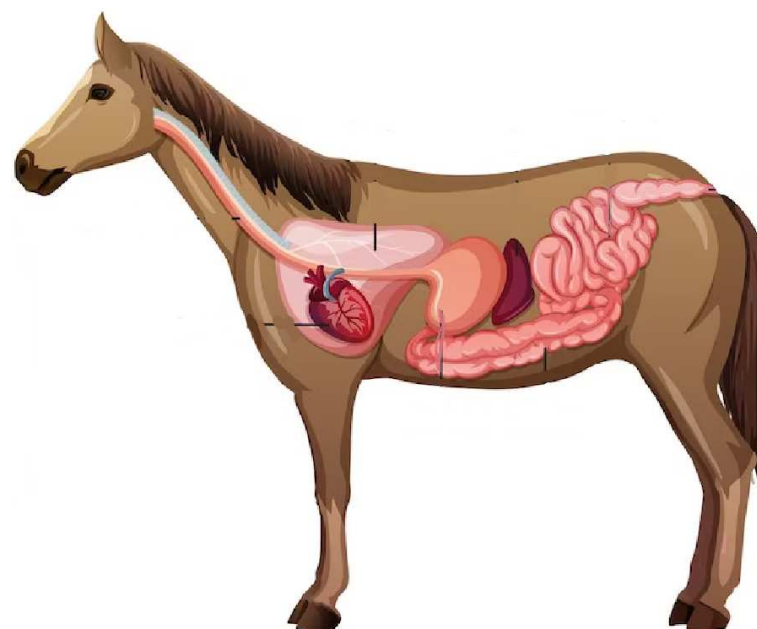
części funkcjonalne
organizmu

Przykład: (koń)

Klasa:

Obiekt:

publiczne	własność: <i>nogi</i>
	własność: <i>kopyta</i>
	własność: <i>ogon</i>
	metoda: <i>galop</i>
	metoda: <i>stęp</i>
prywatne	metoda: <i>skubanie trawy</i>
	własność: <i>przełyk</i>
	własność: <i>serce</i>
	własność: <i>żołądek</i>



Publiczne a prywatne i konstruktor

Klasa:

```
class Zespolona {  
    private double cz_rzecz, cz_uroj;  
  
    public double re() { return cz_rzecz; }  
    public double im() { return cz_uroj; }  
}
```

Publiczne a prywatne i konstruktor

Klasa:

```
class Zespolona {  
    private double cz_rzecz, cz_uroj;
```

każdy obiekt ma mieć takie „wnętrznosci”

```
    public double re() { return cz_rzecz; }  
    public double im() { return cz_uroj; }  
}
```

ich działanie na zewnątrz obiektu

Publiczne a prywatne i konstruktor

Klasa:

```
class Zespolona {  
    private double cz_rzecz, cz_uroj;
```

każdy obiekt ma mieć takie „wnętrznosci”

```
    public double re() { return cz_rzecz; }  
    public double im() { return cz_uroj; }  
}
```

ich działanie na zewnątrz obiektu

Obiekty:

```
Zespolona z1 = ...  
Zespolona z2 = ...  
.....  
double z1_absval =  
    Math.sqrt(z1.re()*z1.re() + z1.im()*z1.im());  
double z2_absval =  
    Math.sqrt(z2.re()*z2.re() + z2.im()*z2.im());
```

Publiczne a prywatne i konstruktor

Klasa:

każdy obiekt ma mieć takie „wnętrza”

```
class Zespolona {
    private double cz_rzecz, cz_uroj;
    public Zespolona(double x, double y) {
        cz_rzecz = x;  cz_uroj = y;
    }
    public double re() { return cz_rzecz; }
    public double im() { return cz_uroj; }
}
```

konstruktor

ich działanie na zewnątrz obiektu

Obiekty:

```
Zespolona z1 = ...
Zespolona z2 = ...
.....
double z1_absval =
    Math.sqrt(z1.re()*z1.re() + z1.im()*z1.im());
double z2_absval =
    Math.sqrt(z2.re()*z2.re() + z2.im()*z2.im());
```

Publiczne a prywatne i konstruktor

Klasa:

każdy obiekt ma mieć takie „wnętrza”

```
class Zespolona {
    private double cz_rzecz, cz_uroj;
    public Zespolona(double x, double y) {
        cz_rzecz = x;  cz_uroj = y;
    }
    public double re() { return cz_rzecz; }
    public double im() { return cz_uroj; }
}
```

konstruktor

ich działanie na zewnątrz obiektu

Obiekty:

```
Zespolona z1 = new Zespolona(3.14, -11.5);
Zespolona z2 = new Zespolona(-15.1, 22.3);
.....
double z1_absval =
    Math.sqrt(z1.re()*z1.re() + z1.im()*z1.im());
double z2_absval =
    Math.sqrt(z2.re()*z2.re() + z2.im()*z2.im());
```

$3.14 - 11.5 \cdot i$

$-15.1 + 22.3 \cdot i$

Przypominajka o klasach i obiektach

W programie w Jawie:

- **Klasa** jest wzorcem — opisem wspólnych cech zbioru obiektów.

Przypominajka o klasach i obiektach

W programie w Jawie:

- **Klasa** jest wzorcem — opisem wspólnych cech zbioru obiektów.
- Nowy obiekt klasy tworzy się jej **konstruktorem**:
`new nazwa_klasy(parametry)`

Przypominajka o klasach i obiektach

W programie w Jawie:

- **Klasa** jest wzorcem — opisem wspólnych cech zbioru obiektów.
- Nowy obiekt klasy tworzy się jej **konstruktorem**:

`new nazwa_klasy(parametry)`

Klasa może mieć wiele konstruktorów — o tej samej nazwie, różniących się listą parametrów.

Przypominajka o klasach i obiektach

W programie w Jawie:

- **Klasa** jest wzorcem — opisem wspólnych cech zbioru obiektów.
- Nowy obiekt klasy tworzy się jej **konstruktorem**:

```
new nazwa_klasy(parametry)
```

Klasa może mieć wiele konstruktorów — o tej samej nazwie, różniących się listą parametrów.

Jeśli w klasie nie napiszemy żadnego konstruktora, to rozumie się, że jest w niej konstruktor bezparametrowy

```
public nazwa_klasy() { }
```

Przypominajka o klasach i obiektach

W programie w Jawie:

- **Klasa** jest wzorcem — opisem wspólnych cech zbioru obiektów.

- Nowy obiekt klasy tworzy się jej **konstruktorem**:

```
new nazwa_klasy(parametry)
```

Klasa może mieć wiele konstruktorów — o tej samej nazwie, różniących się listą parametrów.

Jeśli w klasie nie napiszemy żadnego konstruktora, to rozumie się, że jest w niej konstruktor bezparametrowy

```
public nazwa_klasy() { }
```

- Do pól (własności i metod) obiektu, odpowiadającym polom **prywatnym** klasy, nie ma dostępu.

Przypominajka o klasach i obiektach

- Do pól (własności i metod) obiektu, odpowiadającym polom **publicznym** klasy, mamy dostęp przez notację kropkową

Przypominajka o klasach i obiektach

- Do pól (własności i metod) obiektu, odpowiadającym polom **publicznym** klasy, mamy dostęp przez notację kropkową:

nazwa_obiektu . nazwa_pola

Przypominajka o klasach i obiektach

- Do pól (własności i metod) obiektu, odpowiadającym polom **publicznym** klasy, mamy dostęp przez notację kropkową:
`nazwa_obiektu . nazwa_pola`
- Wymagana jest zgodność *typu* konkretnej własności obiektu z *typem* wymienionym w opisie klasy.

Przypominajka o klasach i obiektach

- Do pól (własności i metod) obiektu, odpowiadającym polom **publicznym** klasy, mamy dostęp przez notację kropkową:

`nazwa_obiektu . nazwa_pola`

- Wymagana jest zgodność *typu* konkretnej własności obiektu z *typem* wymienionym w opisie klasy. Np.:

`int x = tu musi stać liczba całkowita;`

Przypominajka o klasach i obiektach

- Do pól (własności i metod) obiektu, odpowiadającym polom **publicznym** klasy, mamy dostęp przez notację kropkową:

`nazwa_obiektu . nazwa_pola`

- Wymagana jest zgodność *typu* konkretnej własności obiektu z *typem* wymienionym w opisie klasy. Np.:

`int x =` tu musi stać liczba całkowita;
`char x =` tu musi stać znak;

Przypominajka o klasach i obiektach

- Do pól (własności i metod) obiektu, odpowiadającym polom **publicznym** klasy, mamy dostęp przez notację kropkową:

`nazwa_obiektu . nazwa_pola`

- Wymagana jest zgodność *typu* konkretnej własności obiektu z *typem* wymienionym w opisie klasy. Np.:

`int x =` *tu musi stać liczba całkowita* ;

`char x =` *tu musi stać znak* ;

`Zespolona z =` *tu musi stać liczba zespolona* ;

Przypominajka o klasach i obiektach

- Do pól (własności i metod) obiektu, odpowiadającym polom **publicznym** klasy, mamy dostęp przez notację kropkową:

`nazwa_obiektu . nazwa_pola`

- Wymagana jest zgodność *typu* konkretnej własności obiektu z *typem* wymienionym w opisie klasy. Np.:

`int x =` tu musi stać liczba całkowita;

`char x =` tu musi stać znak;

`Zespolona z =` tu musi stać liczba zespolona;

`z.czesc_urojona =` tu musi stać liczba rzeczywista;

Przypominajka o klasach i obiektach

- Do pól (własności i metod) obiektu, odpowiadającym polom **publicznym** klasy, mamy dostęp przez notację kropkową:

`nazwa_obiektu . nazwa_pola`

- Wymagana jest zgodność *typu* konkretnej własności obiektu z *typem* wymienionym w opisie klasy. Np.:

`int x =` tu musi stać liczba całkowita;

`char x =` tu musi stać znak;

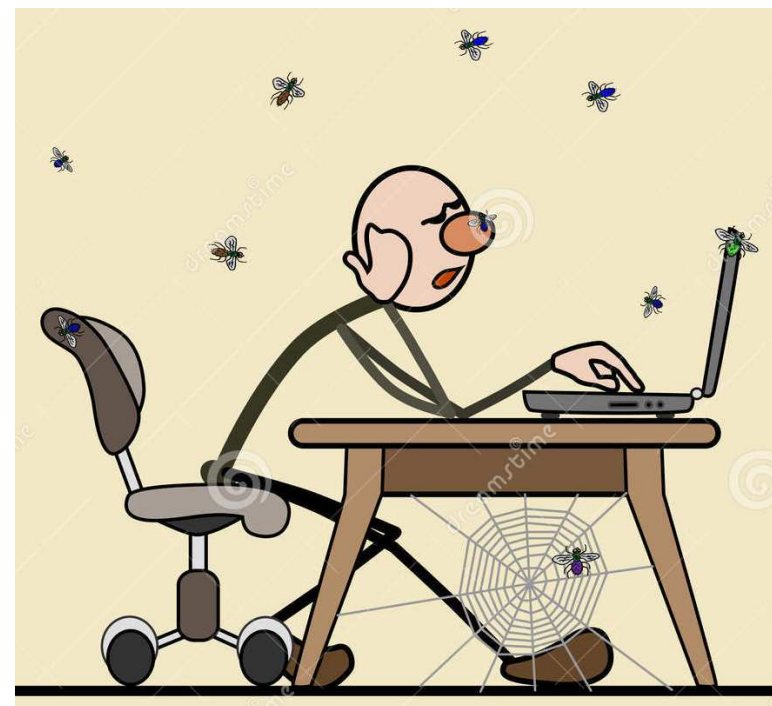
`Zespolona z =` tu musi stać liczba zespolona;

`z.czesc_urojona =` tu musi stać liczba rzeczywista;

`Kon k =` tu musi stać koń;

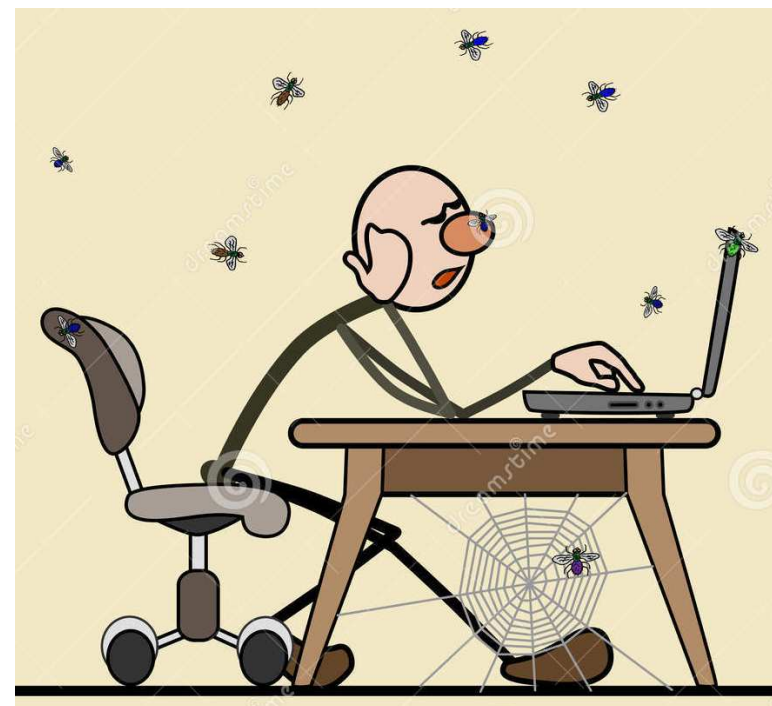
itp.

Programowanie obiektowe a strukturalne



Programowanie obiektowe a strukturalne

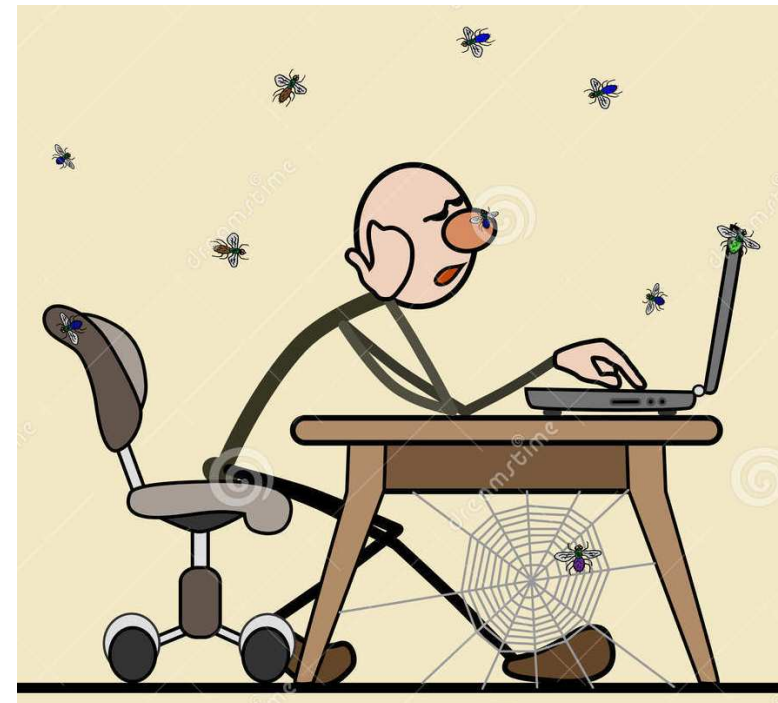
Odpowiednikami obiektów w C są *struktury*...



Programowanie obiektowe a strukturalne

Odpowiednikami obiektów w **C** są *struktury*...

Więc czym różni się programowanie obiektowe od programowania strukturalnego, takiego jak w **C**?



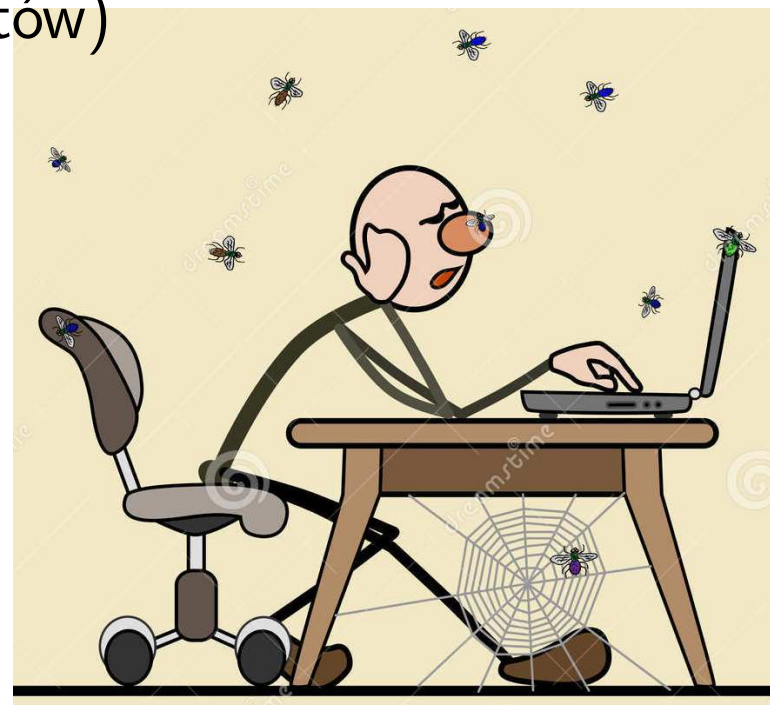
Programowanie obiektowe a strukturalne

Odpowiednikami obiektów w **C** są *struktury*...

Więc czym różni się programowanie obiektowe od programowania strukturalnego, takiego jak w **C**?

Najważniejsza różnica:

polami klasy (i w konsekwencji cechami obiektów) mogą być **funkcje** na własnościach obiektu — czyli *metody*.



Program w Jawie, ale jeszcze strukturalny

```
class Punkt { double wsp_x, wsp_y; }

class Punkty {
    static Punkt nowy_pkt(double x, double y) {
        Punkt p = new Punkt( );
        p.wsp_x = x;  p.wsp_y = y;
        return p;
    }

    static double odl(Punkt pt1, Punkt pt2) {
        double odl_x = pt1.wsp_x - pt2.wsp_x,
               odl_y = pt1.wsp_y - pt2.wsp_y;
        return Math.sqrt(odl_x*odl_x + odl_y*odl_y);
    }

    public static void main(String[ ] param) {
        Punkt pkt0 = nowy_pkt(0,0),
               pkt1 = nowy_pkt(1,0),
               pkt2 = nowy_pkt(0,1);
        double obwod = odl(pkt0,pkt1)+odl(pkt1,pkt2)+odl(pkt2,pkt0);
        System.out.println("\n  Obwod: " + obwod + "\n");
    }
}
```

Program w Jawie, ale jeszcze strukturalny

```
class Punkt { double wsp_x, wsp_y; }
```

← 2 pola rzeczywiste

```
class Punkty {
```

```
    static Punkt nowy_pkt(double x, double y) {  
        Punkt p = new Punkt( );  
        p.wsp_x = x;  p.wsp_y = y;  
        return p;  
    }
```

```
    static double odl(Punkt pt1, Punkt pt2) {  
        double odl_x = pt1.wsp_x - pt2.wsp_x,  
               odl_y = pt1.wsp_y - pt2.wsp_y;  
        return Math.sqrt(odl_x*odl_x + odl_y*odl_y);  
    }
```

```
    public static void main(String[ ] param) {  
        Punkt pkt0 = nowy_pkt(0,0),  
               pkt1 = nowy_pkt(1,0),  
               pkt2 = nowy_pkt(0,1);  
        double obwod = odl(pkt0,pkt1)+odl(pkt1,pkt2)+odl(pkt2,pkt0);  
        System.out.println("\n  Obwod: " + obwod + "\n");  
    }
```

```
}
```

Program w Jawie, ale jeszcze strukturalny

```
class Punkt { double wsp_x, wsp_y; }
```

← 2 pola rzeczywiste

```
class Punkty {
```

```
    static Punkt nowy_pkt(double x, double y) {  
        Punkt p = new Punkt( );  
        p.wsp_x = x;  p.wsp_y = y;  
        return p;  
    }
```

} tworzenie punktu

```
    static double odl(Punkt pt1, Punkt pt2) {  
        double odl_x = pt1.wsp_x - pt2.wsp_x,  
               odl_y = pt1.wsp_y - pt2.wsp_y;  
        return Math.sqrt(odl_x*odl_x + odl_y*odl_y);  
    }
```

```
    public static void main(String[ ] param) {  
        Punkt pkt0 = nowy_pkt(0,0),  
              pkt1 = nowy_pkt(1,0),  
              pkt2 = nowy_pkt(0,1);  
        double obwod = odl(pkt0,pkt1)+odl(pkt1,pkt2)+odl(pkt2,pkt0);  
        System.out.println("\n  Obwod: " + obwod + "\n");  
    }
```

```
}
```

Program w Jawie, ale jeszcze strukturalny

```
class Punkt { double wsp_x, wsp_y; }
```

← 2 pola rzeczywiste

```
class Punkty {
```

```
    static Punkt nowy_pkt(double x, double y) {  
        Punkt p = new Punkt( );  
        p.wsp_x = x;  p.wsp_y = y;  
        return p;  
    }
```

} tworzenie punktu

```
    static double odl(Punkt pt1, Punkt pt2) {  
        double odl_x = pt1.wsp_x - pt2.wsp_x,  
              odl_y = pt1.wsp_y - pt2.wsp_y;  
        return Math.sqrt(odl_x*odl_x + odl_y*odl_y);  
    }
```

} odległość punktów

```
    public static void main(String[ ] param) {  
        Punkt pkt0 = nowy_pkt(0,0),  
              pkt1 = nowy_pkt(1,0),  
              pkt2 = nowy_pkt(0,1);  
        double obwod = odl(pkt0,pkt1)+odl(pkt1,pkt2)+odl(pkt2,pkt0);  
        System.out.println("\n  Obwod: " + obwod + "\n");  
    }
```

```
}
```

Program w Jawie, ale jeszcze strukturalny

```
class Punkt { double wsp_x, wsp_y; }
```

← 2 pola rzeczywiste

```
class Punkty {
```

```
    static Punkt nowy_pkt(double x, double y) {  
        Punkt p = new Punkt( );  
        p.wsp_x = x;  p.wsp_y = y;  
        return p;  
    }
```

} tworzenie punktu

```
    static double odl(Punkt pt1, Punkt pt2) {  
        double odl_x = pt1.wsp_x - pt2.wsp_x,  
               odl_y = pt1.wsp_y - pt2.wsp_y;  
        return Math.sqrt(odl_x*odl_x + odl_y*odl_y);  
    }
```

} odległość punktów

```
    public static void main(String[ ] param) {  
        Punkt pkt0 = nowy_pkt(0,0),  
               pkt1 = nowy_pkt(1,0),  
               pkt2 = nowy_pkt(0,1);  
        double obwod = odl(pkt0,pkt1)+odl(pkt1,pkt2)+odl(pkt2,pkt0);  
        System.out.println("\n  Obwod: " + obwod + "\n");  
    }
```

} GŁÓWNY

```
}
```

Program w Jawie, ale jeszcze strukturalny

```
class Punkt { double wsp_x, wsp_y; }
```

← 2 pola rzeczywiste

```
class Punkty {
```

```
    static Punkt nowy_pkt(double x, double y) {  
        Punkt p = new Punkt( );  
        p.wsp_x = x;  p.wsp_y = y;  
        return p;  
    }
```

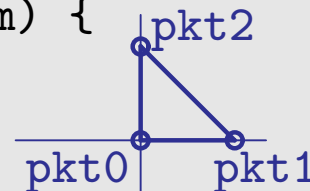
} tworzenie punktu

```
    static double odl(Punkt pt1, Punkt pt2) {  
        double odl_x = pt1.wsp_x - pt2.wsp_x,  
              odl_y = pt1.wsp_y - pt2.wsp_y;  
        return Math.sqrt(odl_x*odl_x + odl_y*odl_y);  
    }
```

} odległość punktów

```
    public static void main(String[ ] param) {  
        Punkt pkt0 = nowy_pkt(0,0),  
              pkt1 = nowy_pkt(1,0),  
              pkt2 = nowy_pkt(0,1);  
        double obwod = odl(pkt0,pkt1)+odl(pkt1,pkt2)+odl(pkt2,pkt0);  
        System.out.println("\n  Obwod: " + obwod + "\n");  
    }
```

} GŁÓWNY



Program w Jawie, ale jeszcze strukturalny

Program z poprzedniego slajdu

- dałby się łatwo przepisać na język nieobiektowy (taki jak C)

Program w Jawie, ale jeszcze strukturalny

Program z poprzedniego slajdu

- dałby się łatwo przepisać na język nieobiektowy (taki jak C);
- w nim **Punkt** jest tylko parą liczb bez żadnych metod

Program w Jawie, ale jeszcze strukturalny

Program z poprzedniego slajdu

- dałby się łatwo przepisać na język nieobiektowy (taki jak C);
- w nim `Punkt` jest tylko parą liczb bez żadnych metod;
- funkcje działające na punktach (`nowy_pkt()`, `odl()`) są dla klasy `Punkt` całkiem zewnętrzne.

Program w Jawie, ale jeszcze strukturalny

Program z poprzedniego slajdu

- dałby się łatwo przepisać na język nieobiektowy (taki jak C);
- w nim `Punkt` jest tylko parą liczb bez żadnych metod;
- funkcje działające na punktach (`nowy_pkt()`, `odl()`) są dla klasy `Punkt` całkiem zewnętrzne.

Jak smakuje obiektowość?

Program w Jawie, ale jeszcze strukturalny

Program z poprzedniego slajdu

- dałby się łatwo przepisać na język nieobiektowy (taki jak C);
- w nim `Punkt` jest tylko parą liczb bez żadnych metod;
- funkcje działające na punktach (`nowy_pkt()`, `odl()`) są dla klasy `Punkt` całkiem zewnętrzne.

Jak smakuje obiektowość?

Tworzenie punktu powinno stać się wewnętrznym konstruktorem.

Program w Jawie, ale jeszcze strukturalny

Program z poprzedniego slajdu

- dałby się łatwo przepisać na język nieobiektowy (taki jak C);
- w nim `Punkt` jest tylko parą liczb bez żadnych metod;
- funkcje działające na punktach (`nowy_pkt()`, `odl()`) są dla klasy `Punkt` całkiem zewnętrzne.

Jak smakuje obiektowość?

Tworzenie punktu powinno stać się wewnętrznym *konstruktorem*.

Odległość powinna stać się wewnętrzną metodą liczącą, jak daleko dany punkt jest od innego punktu.

Jak smakuje obiektowość?

```
class Punkt {  
    double wsp_x, wsp_y;  
    Punkt(double x, double y) {  
        wsp_x = x; wsp_y = y;  
    }  
    double odl(Punkt pt) {  
        double odl_x = pt.wsp_x - this.wsp_x,  
            odl_y = pt.wsp_y - this.wsp_y;  
        return Math.sqrt(odl_x*odl_x + odl_y*odl_y);  
    }  
}  
  
class Punkty {  
    public static void main(String[] param) {  
        Punkt pkt0 = new Punkt(0,0),  
            pkt1 = new Punkt(1,0),  
            pkt2 = new Punkt(0,1);  
        double obwod = pkt0.odl(pkt1)+pkt1.odl(pkt2)+pkt2.odl(pkt0);  
        System.out.println("\n  Obwod: " + obwod + "\n");  
    }  
}
```

Jak smakuje obiektowość?

```
class Punkt {  
    double wsp_x, wsp_y;  
    Punkt(double x, double y) {  
        wsp_x = x; wsp_y = y;  
    }  
    double odl(Punkt pt) {  
        double odl_x = pt.wsp_x - this.wsp_x,  
            odl_y = pt.wsp_y - this.wsp_y;  
        return Math.sqrt(odl_x*odl_x + odl_y*odl_y);  
    }  
}  
  
class Punkty {  
    public static void main(String[] param) {  
        Punkt pkt0 = new Punkt(0,0),  
            pkt1 = new Punkt(1,0),  
            pkt2 = new Punkt(0,1);  
        double obwod = pkt0.odl(pkt1)+pkt1.odl(pkt2)+pkt2.odl(pkt0);  
        System.out.println("\n  Obwod: " + obwod + "\n");  
    }  
}
```

← własności

Jak smakuje obiektowość?

```
class Punkt {  
    double wsp_x, wsp_y;  
    Punkt(double x, double y) {  
        wsp_x = x; wsp_y = y;  
    }  
    double odl(Punkt pt) {  
        double odl_x = pt.wsp_x - this.wsp_x,  
            odl_y = pt.wsp_y - this.wsp_y;  
        return Math.sqrt(odl_x*odl_x + odl_y*odl_y);  
    }  
}
```

← własności

} konstruktor

```
class Punkty {  
    public static void main(String[] param) {  
        Punkt pkt0 = new Punkt(0,0),  
            pkt1 = new Punkt(1,0),  
            pkt2 = new Punkt(0,1);  
        double obwod = pkt0.odl(pkt1)+pkt1.odl(pkt2)+pkt2.odl(pkt0);  
        System.out.println("\n  Obwod: " + obwod + "\n");  
    }  
}
```

Jak smakuje obiektowość?

```
class Punkt {  
    double wsp_x, wsp_y;  
    Punkt(double x, double y) {  
        wsp_x = x; wsp_y = y;  
    }  
    double odl(Punkt pt) {  
        double odl_x = pt.wsp_x - this.wsp_x,  
            odl_y = pt.wsp_y - this.wsp_y;  
        return Math.sqrt(odl_x*odl_x + odl_y*odl_y);  
    }  
}
```

← własności

} konstruktor

} metoda

```
class Punkty {  
    public static void main(String[] param) {  
        Punkt pkt0 = new Punkt(0,0),  
            pkt1 = new Punkt(1,0),  
            pkt2 = new Punkt(0,1);  
        double obwod = pkt0.odl(pkt1)+pkt1.odl(pkt2)+pkt2.odl(pkt0);  
        System.out.println("\n Obwod: " + obwod + "\n");  
    }  
}
```

Jak smakuje obiektowość?

```
class Punkt {  
    double wsp_x, wsp_y;  
    Punkt(double x, double y) {  
        wsp_x = x; wsp_y = y;  
    }  
    double odl(Punkt pt) {  
        double odl_x = pt.wsp_x - this.wsp_x,  
            odl_y = pt.wsp_y - this.wsp_y;  
        return Math.sqrt(odl_x*odl_x + odl_y*odl_y);  
    }  
}
```

← własności

} konstruktor

} metoda


```
class Punkty {  
    public static void main(String[ ] param) {  
        Punkt pkt0 = new Punkt(0,0),  
            pkt1 = new Punkt(1,0),  
            pkt2 = new Punkt(0,1);  
        double obwod = pkt0.odl(pkt1)+pkt1.odl(pkt2)+pkt2.odl(pkt0);  
        System.out.println("\n  Obwod: " + obwod + "\n");  
    }  
}
```

} użycie konstruktora

Program już obiektowy, ale...

Program z ostatniego slajdu

- nie da się łatwo przepisać na C

Program już obiektowy, ale...

Program z ostatniego slajdu

- nie da się łatwo przepisać na C;
- oprócz własności, każdy **Punkt** zawiera konstruktor i metodę liczenia odległości od innego punktu.

Program już obiektowy, ale...

Program z ostatniego slajdu

- nie da się łatwo przepisać na C;
- oprócz własności, każdy **Punkt** zawiera konstruktor i metodę liczenia odległości od innego punktu.

... są wady:

```
poczatek = new Punkt(0,0);  
poczatek.wsp_x = poczatek.wsp_y = 1;
```

Program już obiektowy, ale...

Program z ostatniego slajdu

- nie da się łatwo przepisać na C;
- oprócz własności, każdy **Punkt** zawiera konstruktor i metodę liczenia odległości od innego punktu.

... są wady:

```
poczatek = new Punkt(0,0);  
poczatek.wsp_x = poczatek.wsp_y = 1;
```

W wyniku tej komendy już ustalony początek układu współrzędnych przestaje być początkiem układu.

Program już obiektowy, ale...

Program z ostatniego slajdu

- nie da się łatwo przepisać na C;
- oprócz własności, każdy **Punkt** zawiera konstruktor i metodę liczenia odległości od innego punktu.

... są wady:

```
poczatek = new Punkt(0,0);  
poczatek.wsp_x = poczatek.wsp_y = 1;
```

W wyniku tej komendy już ustalony początek układu współrzędnych przestaje być początkiem układu.

Żeby temu zapobiec, własności `wsp_x` i `wsp_y` powinny być **prywatne**.

Program już obiektowy, ale...

Program z ostatniego slajdu

- nie da się łatwo przepisać na C;
- oprócz własności, każdy **Punkt** zawiera konstruktor i metodę liczenia odległości od innego punktu.

... są wady:

```
poczatek = new Punkt(0,0);  
poczatek.wsp_x = poczatek.wsp_y = 1;
```



W wyniku tej komendy już ustalony początek układu współrzędnych przestaje być początkiem układu.

Żeby temu zapobiec, własności `wsp_x` i `wsp_y` powinny być **prywatne**.

iis.ans-elblag.pl/~stefan/Dydaktyka/Prog0biII/Prog01/Punkt.java

iis.ans-elblag.pl/~stefan/Dydaktyka/Prog0biII/Prog01/Punkty.java

Program w pełni obiektowy

ENKAPSULACJA — zamknięcie w szczelną kapsułkę

HERMETYZACJA — mniej więcej to samo

Program w pełni obiektowy

ENKAPSULACJA — zamknięcie w szczelną kapsułkę

HERMETYZACJA — mniej więcej to samo

— uniemożliwienie zmiany własności obiektów w sposób niezgodny z intencją programisty, który stworzył klasę.

Program w pełni obiektowy

ENKAPSULACJA — zamknięcie w szczelną kapsułkę

HERMETYZACJA — mniej więcej to samo



— uniemożliwienie zmiany własności obiektów w sposób niezgodny z intencją programisty, który stworzył klasę.

Jest to szczególnie istotne, kiedy program składa się z wielu klas, każda pochodzi od innego autora, każdy zna szczegółowo tylko klasy stworzone przez siebie. Wtedy ma sens uniemożliwienie innym dokonywania działań „nielegalnych”.

Program w pełni obiektowy

ENKAPSULACJA — zamknięcie w szczelną kapsułkę

HERMETYZACJA — mniej więcej to samo



— uniemożliwienie zmiany własności obiektów w sposób niezgodny z intencją programisty, który stworzył klasę.

Jest to szczególnie istotne, kiedy program składa się z wielu klas, każda pochodzi od innego autora, każdy zna szczegółowo tylko klasy stworzone przez siebie. Wtedy ma sens uniemożliwienie innym dokonywania działań „nielegalnych”.

Nawet, jeśli wszystkie klasy pochodzą od tego samego programisty, ma sens utrudnienie sobie samemu popełniania niesygnalizowanych błędów.

Prywatne własności

Własności obiektu *w zasadzie* powinny być `private`:

```
class Pojazd {  
    private int liczba_miejsc;  
    private double czas_jazdy=0, przebieg=0;  
    ...  
}
```

Prywatne własności

Własności obiektu *w zasadzie* powinny być **private**:

```
class Pojazd {  
    private int liczba_miejsc;  
    private double czas_jazdy=0, przebieg=0;  
    ...  
}
```

Dostęp do nich z zewnątrz przez specjalne metody, t.zw. *selektory* (*getter*):

```
class Pojazd {  
    ...  
    int liczba_miejsc() { return liczba_miejsc; };  
    double czas_jazdy() { return czas_jazdy; };  
    double przebieg() { return przebieg; };  
    ...  
}
```

Prywatne własności

Własności obiektu w zasadzie powinny być *private*:

```
class Pojazd {  
    private int liczba_miejsc;  
    private double czas_jazdy=0, przebieg=0;  
    ...  
}
```

Dostęp do nich z zewnątrz przez specjalne metody, t.zw. *selektory* (*getter*):

```
class Pojazd {  
    ...  
    int liczba_miejsc() { return liczba_miejsc; };  
    double czas_jazdy() { return czas_jazdy; };  
    double przebieg() { return przebieg; };  
    ...  
}
```

Jeśli nie dostarczymy metod obsługujących jakąś własność prywatną, to jest ona całkiem *ukryta* — niedostępna z zewnątrz ani do zapisu ani do odczytu.

Inicjalizacja prywaty

Inicjalizacja własności prywatnych:

- albo **przy deklaracji**: `double czas_jazdy=0;`

Inicjalizacja prywatny

Inicjalizacja własności prywatnych:

- albo **przy deklaracji**: `double czas_jazdy=0;`
- albo **przez *konstruktor*** klasy:

```
class Pojazd {  
    ...  
    Pojazd(int licz_mc) { liczba_miejsc = licz_mc; }  
    ...  
}
```

Inicjalizacja prywaty

Inicjalizacja własności prywatnych:

- albo **przy deklaracji**: `double czas_jazdy=0;`
- albo **przez *konstruktor*** klasy:

```
class Pojazd {  
    ...  
    Pojazd(int licz_mc) { liczba_miejsc = licz_mc; }  
    ...  
}
```

Zewnętrzne wywołanie:

```
Pojazd samochod = new Pojazd(5);
```



Inicjalizacja prywaty

Inicjalizacja własności prywatnych:

- albo **przy deklaracji**: `double czas_jazdy=0;`
- albo **przez *konstruktor*** klasy:

```
class Pojazd {  
    ...  
    Pojazd(int licz_mc) { liczba_miejsc = licz_mc; }  
    ...  
}
```

Zewnętrzne wywołanie:

```
Pojazd samochod = new Pojazd(5);  
Pojazd rower = new Pojazd(1);
```



Inicjalizacja prywaty

Inicjalizacja własności prywatnych:

- albo **przy deklaracji**: `double czas_jazdy=0;`
- albo **przez *konstruktor*** klasy:

```
class Pojazd {  
    ...  
    Pojazd(int licz_mc) { liczba_miejsc = licz_mc; }  
    ...  
}
```

Zewnętrzne wywołanie:

```
Pojazd samochod = new Pojazd(5);  
Pojazd rower = new Pojazd(1);
```



Ponieważ pole `liczba_miejsc` klasy `Pojazd` jest `private`, tak utworzonemu samochodowi czy rowerowi **nie da się** już zmienić liczby miejsc.

I BARDZO DOBRZE!