

Stefan Sokołowski

PROGRAMOWANIE OBIEKTOWE II

wykład 3

Inst. Informatyki Stosowanej, ANS
Elbląg, 2024/2025

Wykład 3, str. 1

BMI dla figur na płaszczyźnie

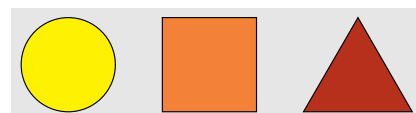
BMI — body-mass index
 $\frac{\text{masa (kg)}}{(\text{wzrost (m)})^2}$

Koło jest *pękatę*, trójkąt jest *szczuplejszy*.

Czy figury geometryczne (**nie tylko wielokąty!**) da się porównywać ze względu na ich „*pękatę*”?

Przez *pękatę* figury płaskiej będę rozumiał stosunek jej powierzchni do kwadratu jej obwodu:

$$\text{pękatę} \stackrel{\text{def}}{=} \frac{\text{pole}}{\text{obwód}^2}$$



Np.:

figura	pękatę (BMI)
koło o prom. r	$\frac{\pi r^2}{(2\pi r)^2} = \frac{1}{4\pi} \approx 0.0796$ (niezależnie od r)
kwadrat o boku a	$\frac{a^2}{(4a)^2} = \frac{1}{16} = 0.0625$ (niezależnie od a)
3-kąt równobocz. o boku a	$\frac{\frac{a^2\sqrt{3}}{4}}{(3a)^2} = \frac{\sqrt{3}}{9 \cdot 4} \approx 0.0481$ (niezależnie od a)

BMI dla figur na płaszczyźnie

```
class BMI_Figur {
    public static void main (String[] args) {
        System.out.println(
            "\n BMI kola o promieniu 101.6 == " +
            new Kolo(101.6).bmi()
        );
        System.out.println(
            "\n BMI kwadratu o boku 11.3 == " +
            new Kwadrat(11.3).bmi()
        );
        System.out.println(
            "\n BMI trojk. rownobocz. o boku 25.7 == " +
            new TrojkForm(25.7).bmi()
        );
        System.out.println();
    }
}
```

Musimy zdefiniować klasy

Kolo

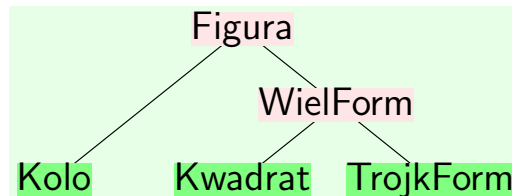
Kwadrat

TrojckForm

a w każdej własną metodę `bmi()`.

Żeby nie definiować tego samego po kilka razy, te klasy powinny **dziedziczyć** (czyli być rozszerzeniami) tej samej klasy nadrzędnej.

Koło jest figurą, ale nie jest wielokątem foremnym

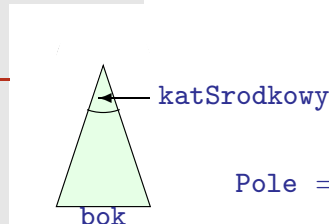


Klasa **wielokątów foremnych** zawiera m.in. klasy **kwadratów** i **trójkątów równobocznych**

BMI dla figur na płaszczyźnie

```
class WielForm extends Figura {
    private int ileWierzch;
    private double bok, katSrodkowy;
    public WielForm(int ileWierzch, double bok) {
        this.ileWierzch = ileWierzch; this.bok = bok;
        katSrodkowy = 2*Math.PI / ileWierzch;
    }
    public double pole() {
        return ileWierzch * ...
    }
    public double obwod() {
        return ileWierzch * bok;
    }
}
```

Klasa **WielForm** dziedziczy metodę `bmi()` z klasy **Figura**, ale musi jej dostarczyć własny sposób obliczania pola i obwodu.



$$\text{Pole} = \frac{b^2}{4 \cdot \text{tg}(\text{katSrodkowy})}$$

```
class Kwadrat extends WielForm {
    public Kwadrat(double bok) {
        super(4, bok);
    }
}
```

```
class TrojkForm extends WielForm {
    public TrojkForm(double bok) {
        super(3, bok);
    }
}
```

Klasy **Kwadrat** i **TrojckForm** dziedziczą metodę `bmi()` z klasy **WielForm**.

BMI dla figur na płaszczyźnie

```
abstract class Figura {
    abstract double pole();
    abstract double obwod();
    public double bmi() {
        return this.pole() /
            (this.obwod()*this.obwod());
    }
}
```

Metody `pole()` i `obwod()` wykorzystywane w metodzie `bmi()` zostaną różnie zdefiniowane w różnych rozszerzeniach tej klasy. Na razie są „**abstrakcyjne**”, co musi zostać przy nich zaznaczone.

Klasa zawierająca własności lub metody abstrakcyjne, musi być oznaczona jako „**abstrakcyjna**”. Klasa abstrakcyjna nie ma własnych obiektów

~~`Figura fg = new Figura();`~~

```
class Kolo extends Figura {
    private double r; // promien
    public Kolo(double r) { this.r = r; }
    public double pole() { return Math.PI*r*r; }
    public double obwod() { return 2*Math.PI*r; }
}

class WielForm extends Figura {
    private int ileWierzch;
    private double bok, katSrodkowy;
    public WielForm(int ileWierzch, double bok) {
        this.ileWierzch = ileWierzch; this.bok = bok;
        katSrodkowy = 2*Math.PI / ileWierzch;
    }
    public double pole() {
        double poleWycinka =
            bok*bok/Math.tan(katSrodkowy/2)/4;
        return ileWierzch*poleWycinka;
    }
    public double obwod() {
        return ileWierzch*bok;
    }
}
```

O abstrakcjach

Klasa **abstrakcyjna** — wzorec dla konstrukcji podklas.

Klasa **abstrakcyjna** może wymagać od klas dziedziczących zdefiniowania metod, z których będzie korzystał np. dla wspólnego zdefiniowania własnych metod lub własności.

<pre>abstract class Figura { abstract double pole(); abstract double obwod(); public double bmi() { return this.pole() / (this.obwod()*this.obwod()); } }</pre>	} — to mają zdefiniować podklasy } — to klasa abstrakcyjna dostarczy podklasom
---	--

Klasa anonimowa

```
abstract class A {
    abstract void aa(String s);
}
.....
new A().aa(" OK.");
```

A is abstract;
cannot be instantiated

```
abstract class A {
    abstract void aa(String s);
}
.....
new A(){
    void aa(String s) {
        System.out.println(s);
    }
}.aa(" OK.");
```

drukuje OK.

iis.ans-elblag.pl/~stefan/Dydaktyka/ProgObiII/Prog03/P2_Anonim/

```
abstract class A {
    abstract void aa(String s);
}
abstract class B extends A {
    void aa(String s) {
        System.out.println(s);
    }
}
.....
new B().aa(" OK.");
```

drukuje OK.

Konstruktor klasy A „poprawionej” przez zdefiniowanie brakującej metody `aa`. Ta klasa nie ma nazwy, ale już nie jest abstrakcyjna, pozwala na tworzenie obiektów.

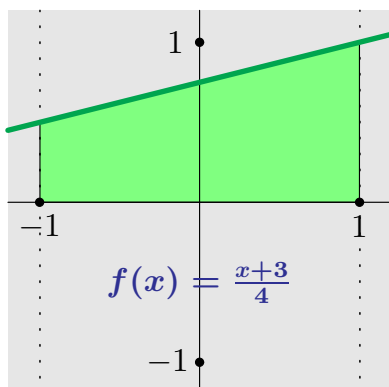
Funkcja jako argument innej funkcji

Przykład:

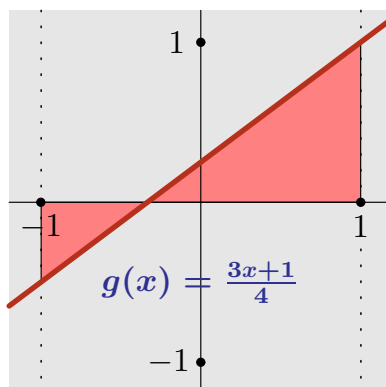
Napisać funkcję *calka*, która dla *dowolnej* funkcji

`double fcja(double x)`

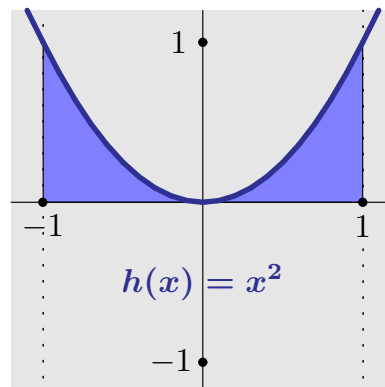
oblicza **pole pod wykresem** funkcji *fcja* nad odcinkiem $[-1 \dots 1]$:



$$\text{calka}(f) = 1.5$$



$$\text{calka}(g) = 0.5$$

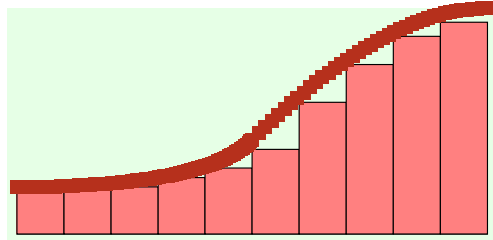


$$\text{calka}(h) = 0.66666\dots$$

Funkcja jako argument innej funkcji

W C funkcja **może być** argumentem innej funkcji:

```
double calka(double fcja(double)) {
    double wynik = 0.0;
    for (double x=dol; x<gora; x+=krok)
        wynik += krok*fcja(x);
    return wynik;
}
```



```
double f(double x) { return (x+3)/4; }
double g(double x) { return (3*x+1)/4; }
double h(double x) { return x*x; }
. . . . .
printf(" calka z f == %lf\n", calka(f));
printf(" calka z g == %lf\n", calka(g));
printf(" calka z h == %lf\n", calka(h));
```

iis.ans-elblag.pl/~stefan/Dydaktyka/Prog0biII/Prog03/P3_FcjaFcji/fcja_fcji.c

Funkcja jako argument innej funkcji

W Javie funkcja **nie może być** argumentem innej funkcji.
Ale argumentem funkcji może być obiekt.

Opakować typ funkcyjny w klasę!

```
abstract class Fcja {
    abstract double fcja(double x);
}
```

— „typ” parametru funkcyjnego

Parametr `Fcja fun` oznacza, że odpowiadający mu argument wywołania jest obiektem posiadającym metodę `double fcja(double x)`.

Funkcja licząca całki powinna pobierać obiekt tej klasy z dospecyfikowaną metodą `fcja`:

```
calka(
    new Fcja(){
        public double fcja(double x) { return 1; }
    }
)
```

iis.ans-elblag.pl/~stefan/Dydaktyka/Prog0biII/Prog03/P3_FcjaFcji/Abstrakcja.java

Funkcja jako argument innej funkcji

To samo inaczej:

Opakować typ funkcyjny w interface!

```
interface Fcja {
    public double fcja(double x);
}
```

— „typ” parametru funkcyjnego

Interface — zbiór wymagań od klas, które mają go **implementować**: mają zawierać wyszczególnione własności i metody.

Nie ma żadnego `abstract` : w interfejsie **wszystko** (prawie...) jest abstrakcyjne.

iis.ans-elblag.pl/~stefan/Dydaktyka/ProgObiII/Prog03/P3_FcjaFcji/Interfejs.java

Klasy abstrakcyjne a interface'y

KLASY ABSTRAKCYJNE

- `abstract class`
- **niektóre** metody wymagają konkretyzacji przed wygenerowaniem obiektu (metody `abstract`)
- przejście do konkretyzacji przez `extends`

INTERFACE'Y

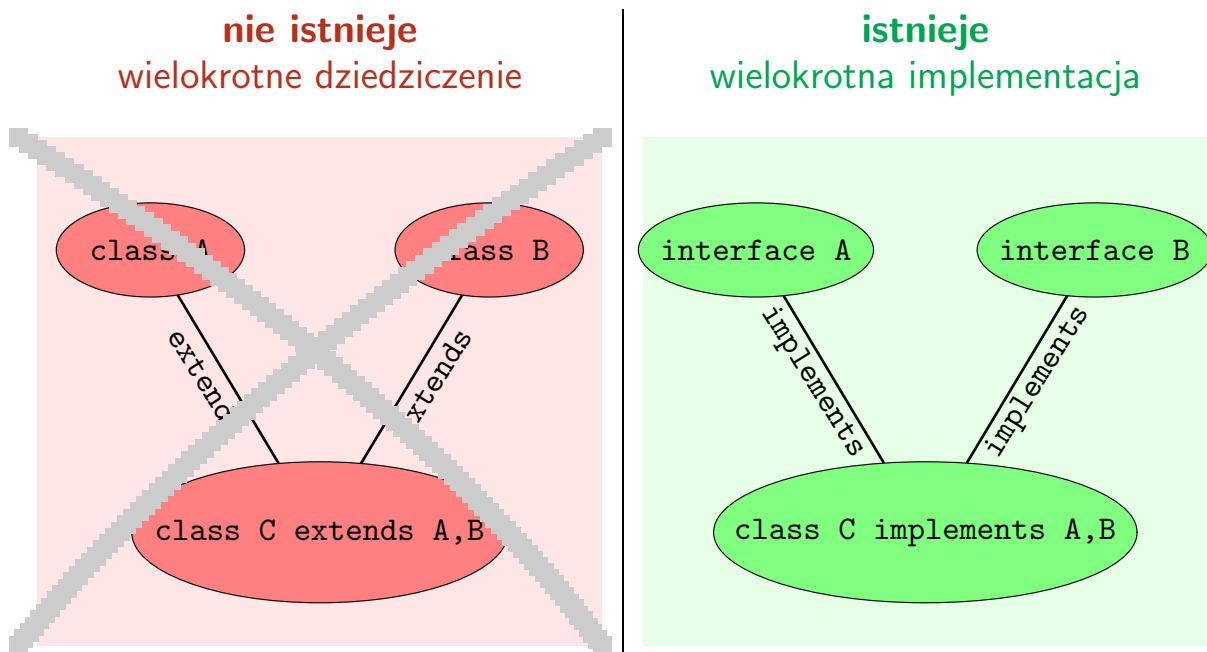
- `interface`
- **wszystkie** metody wymagają konkretyzacji przed wygenerowaniem obiektu
- przejście do konkretyzacji przez `implements`

Po co są interface'y? Przecież zawsze można zamienić

- `interface` na `abstract class` i jej metody poprzedzić kwalifikatorem `abstract`
- zmienić wszystkie `implements` na `extends`

No więc **co nowego** dają interface'y?

Klasy abstrakcyjne a interface'y



Klasy abstrakcyjne a interface'y

Kiedy używać którego?

klasy abstrakcyjne	interfejsy
• kilka podobnych klas ma mieć wspólny fragment kodu • wspólny kod ma nie być public • wspólne właściwości mają nie być static	• klasy implementujące mają mieć ze sobą mało wspólnego • potrzebne jest dziedziczenie wielokrotne

Dziedziczenie wielokrotne bywa potrzebne wtedy, gdy chcemy rozszerzyć klasę (np. Punkt do PktBarwny), a jednocześnie potrzebujemy w niej narzędzi bibliotecznych.

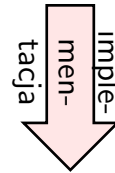
Jeszcze o *interface*'ach

```
interface Fcja {
    public double fcja(double x);
}
```

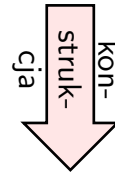
```
class Kwad implements Fcja {
    public double fcja(double x) {
        return (x*x-1)/2;
    }
}
```

```
...calka(new Kwad())...
```

interface



klasa



obiekt i operacja na nim

Klasy anonimowe

interfejs $\xrightarrow{\text{implementacja}}$ klasa $\xrightarrow{\text{konstrukcja}}$ obiekt

Wprowadzanie klasy pośredniczącej między interface-em a obiektem często wydaje się nadmiarowe. Np.

```
class Kwad implements Fcja {
    public double fcja(double x) {
        return (x*x-1)/2;
    }
}
...calka(new Kwad())...
```

„Różne” obiekty klasy Kwad
niczym się od siebie nie różnią.
Więc po co taka klasa?

Czy można połączyć implementację i konstrukcję?

interfejs $\xrightarrow[\text{klasa}]{\text{implementacja i new}}$ obiekt

Klasy anonimowe

Już to robiliśmy...

```
class Kwad implements Fcja {
    public double fcja(double x) {
        return (x*x-1)/2;
    }
    ...
    ...calka(new Kwad( ))...
```

```
...calka( new
    Fcja() {
        public double fcja(double x) {
            return (x*x-1)/2;
        }
    }
)...
```

utworzenie obiektu
klasy nienazwanej

nienazwana
„implementacja”
interfejsu Fcja

Konstruktor klasy nienazwanej:

```
new      nazwa      ( ) {      implementacje
         interface'u          metod
                                interface'u }
```

Wyrażenie Lambda

Jeśli interface składa się **tylko z jednej metody**, to można jeszcze prościej:

```
...calka(
    new Fcja() {
        public double fcja(double x) {
            return (x*x-1)/2;
        }
    }
)...
```

```
...calka(
    x -> { return (x*x-1)/2; }
)...
```

$x \rightarrow \dots$ — wyrażenie Lambda: funkcja x -owi przypisująca $\dots$

z λ -rachunku (Alonzo Church, 1903-1995); oryginalna notacja: $\lambda x. \dots$

$\lambda x. x + 1$
 $x \rightarrow \{ \text{return } x+1; \}$ — następnik

$\lambda x. x \cdot x$
 $x \rightarrow \{ \text{return } x*x; \}$ — kwadrat

$\lambda(x,y).x$
 $(x,y) \rightarrow \{ \text{return } x; \}$ — rzutowanie pary
na pierwszą
współrzędną

Wyrażenie Lambda

```
...calka(  
  new Fcja() {  
    public double fcja(double x) {  
      return (x*x-1)/2;  
    }  
  }  
)...
```



```
...calka(  
  x -> { return (x*x-1)/2; }  
)...
```

czyli

```
new Fcja() {  
  public double fcja(double x) {  
    return (x*x-1)/2;  
  }  
}
```



```
x -> { return (x*x-1)/2; }
```

Czy kompilatorowi nie brakuje informacji
takich jak `double` lub `public` ?

Wyrażenie Lambda

Java jest silnie **utypowana** — każda wartość/wyrażenie/obiekt/metoda ma przypisany jeden typ, do którego należy. Tam, gdzie kompilator nie może ustalić typu, sygnalizuje błąd.

KOD	TYP
<pre>new Fcja() { public double fcja(double x) { return (x*x-1)/2; } }</pre>	konstruktor obiektu klasy implementującej interface <code>Fcja</code> przez zdefiniowanie metody działającej z <code>double</code> do <code>double</code>
<pre>x -> { return (x*x-1)/2; }</pre>	???

λ -rachunek Churcha (skąd pochodzą λ -wyrażenia) jest beztypowy.

Czy to się da pogodzić?

Przykład:

skąd kompilator wie, jaki ma być typ zmiennej `x` w wyrażeniu

```
calka( x -> { return (x*x-1)/2; } ); ?
```

Wyrażenie Lambda

skąd kompilator wie, jaki ma być typ zmiennej `x` w wyrażeniu

```
calka( x -> { return (x*x-1)/2; } ); ?
```

Kontekst:

deklaracja metody `calka`: `static double calka(Fcja f)`

więc

```
x -> { return (x*x-1)/2; }
```

musi być obiektem jakiejś implementacji *interface'u* `Fcja`

którego jedyna metoda ma parametr typu `double`

więc zmienna `x` jest typu `double`

Wyrażenie Lambda zamiast pętli

```
interface Warunek { public Boolean warunek(); }
```

```
interface Akcja { public void akcja(); }
```

dwa interface'y

```
class While {
```

```
    public static void dopoki(Warunek war, Akcja akc) {
```

```
        if (war.warunek()) {
```

```
            akc.aksja(); dopoki(war,akc);
```

```
        }
```

```
    }
```

„definicja”
pętli **while**

```
}
```

```
class Powtorka {
```

```
    static String napis; static int ileRazy;
```

```
    public static void main(String[] args) {
```

```
        napis = args[0]; ileRazy = Integer.parseInt(args[1]);
```

```
        While.dopoki (
```

```
            () -> ileRazy>0 ,
```

```
            () -> { System.out.println(napis); ileRazy--; }
```

```
        );
```

```
    }
```

```
}
```