# Języki i Paradygmaty Programowania

## Programowanie obiektowe i Smalltalk

Marcin Benke

MIM UW

23 maja 2016

# Paradygmat obiektowy

- Świat może być opisany w kategoriach *obiektów*.
- Obiekt ma dwa rodzaje charakterystyk:
  - statyczne: atrybuty (np. kolor, rozmiar)
  - dynamiczne: zachowanie

Inne odczytanie charakterystyki obiektu:

- Co wie? (atrybuty)
- Co potrafi zrobić (metody)

**Jeden mechanizm sterowania: przesyłanie komunikatów.**

*I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages* — Alan Kay

# Hermetyzacja (encapsulation, kapsułkowanie)

```
account.balance -= amount
```

# Hermetyzacja (encapsulation, kapsułkowanie)

```
account.balance -= amount
```

problematyczne... może lepiej

```
account withdraw: amount
```

# Hermetyzacja (encapsulation, kapsułkowanie)

```
account.balance -= amount
```

problematyczne... może lepiej

```
account withdraw: amount
```

Wewnętrzny stan obiektu nie jest dostępny z zewnątrz (zwłaszcza do modyfikacji)

**Tylko przesyłanie komunikatów**

# Języki obiektowe

Dwa główne podejścia

- Klasowe
  - Obiekty mające te same charakterystyki tworzą klasę.
  - Klasyfikacja może mieć wiele poziomów (jak np. systematyka roślin i zwierząt)
  - Każdy obiekt jest egzemplarzem jakiejś klasy; tworzenie obiektów za pośrednictwem klas
  - Simula (1966), Smalltalk (1970), Python (1991?), Ruby (1993), Java (1995?)
- Bezklasowe (prototypy)
  - Brak sztywnej klasyfikacji
  - tworzenie obiektów przez klonowanie istniejących
  - Self (1986), LPmud (1989), Javascript (1995)

Tu: języki klasowe, głównie Smalltalk

# Podstawy paradygmatu obiektowego

1. Wszystko jest obiektem.
2. Obiekty realizują obliczenie przesyłając między sobą komunikaty.
3. Obiekt ma swoją pamięć zawierającą inne obiekty (odwołania do nich).
4. Każdy obiekt jest egzemplarzem klasy (która też jest obiektem).
   (w językach bezklasowych obiekt jest klonem pewnego prototypu)
5. Klasa jest wzorcem zachowania obiektu.
6. Klasy są zorganizowane w hierarchię dziedziczenia.

[Alan Kay]

# Dziedziczenie

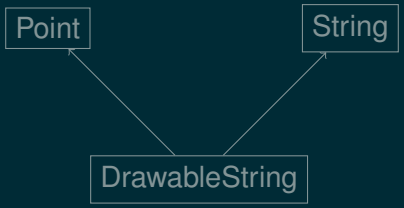
- Klasy obiektów można kojarzyć w hierarchie klas (prowadzi to do drzew lub grafów dziedziczenia).
- Atrubuty i zachowanie klas-przodków są dostępne w klasach potomków (pośrednio lub bezpośrednio).
- Potomek może mieć odmienne zachowanie niż przodek.
- Nadklasy i podklasy (klasy bazowe i pochodne).
- Zasada podstawialności: zawsze powinno być możliwe podstawienie obiektów podklas w miejsce obiektów nadklas.
- Obiekt podklasy musi realizować komunikaty realizowane przez nadklasę
- Jesli podklasa nie definiuje reakcji na dany komunikat, używana jest realizacja z nadklasy (wyszukiwanie metod w hierarchii dziedziczenia).

# Wielodziedziczenie

Wielodziedziczenie *(multiple inheritance)* oznacza, że klasa dziedziczy po 2 lub więcej niezwiązanych ze sobą klasach.

Dziedziczenie po “babce” i “matce” nie jest jeszcze wielodziedziczeniem.

**Przykład**




DrawableString dziedziczy po klasach *Point* i *String*.

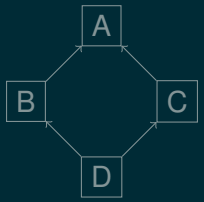
# Wielodziedziczenie — problemy

*“Multiple inheritance is good, but there is no good way to do it.”*

Wielodziedziczenie stwarza liczne problemy, stąd niektóre języki (w tym Smalltalk) unikają go.

Co zrobić gdy więcej niż jedna z nadklas implementuje reakcję na komunikat?

Co zrobić gdy nadklasy mają wspólnego przodka (Diamond of Death)




Jaka powinna być kolejność wyszukiwania metod w tej sytuacji?

W niektórych językach ograniczone wielodziedziczenie: mixins (Lisp/Flavors), interfaces (Java), traits,. . .

# Smalltalk

- Stworzony w latach 70-tych w firmie Xerox.
- Język czysto obiektowy: "wszystko jest obiektem".
- Sterowanie przez wysyłanie komunikatów.
- Pierwszy język dla którego stworzono IDE; prekursor dzisiejszych języków “wizualnych”
- Programy są tworzone przez dodawanie nowych klas do środowiska.
- Stan środowiska razem z kodem programu są zapisywane do pliku obrazu (*image file*)

# Charakterystyczne cechy Smalltalku

- wszystko jest obiektem
- także klasy są obiektami
- program = biblioteka klas + obiekty
  - zadeklarowanie nowej klasy oznacza dodanie jej do biblioteki klas
  - nie ma rozróżnienia pomiędzy klasami “standardowymi” a klasami użytkownika, wszystko można modyfikować
- brak typów zmiennych (dokładniej jeden typ: obiekt)
  - wartością zmiennej może być dowolny obiekt,
  - nie ma kontroli w czasie kompilacji,
  - jeśli obiekt nie rozumie komunikatu — błąd wykonania

# Obiekty i klasy

- Wszystko jest obiektem
- Przykłady:
  - `3`
  - `'ala ma kota'`
  - `nil`
- Klasa — wzorzec dla obiektów
- Przykłady:
  - Integer
  - String
  - UndefinedObject
- Każda klasa jest (być może pośrednio) podklasą klasy `Object`
- Klasa też jest obiektem,
  np. klasa `Integer` jest obiektem klasy `Integer class`
- Klasę, której obiekty są klasami nazywamy *metaklasą*

# Komunikaty i metody

Jedyną formą sterowania jest wysłanie komunikatu, np.

- `2 + 3` (obiekt `2` otrzymuje komunikat `+` z argumentem `3`)
- `counter increment` (obiekt `counter` otrzymuje komunikat `increment`)
- `b ifTrue: [...]`
  (obiekt `b` otrzymuje komunikat `ifTrue:`, którego argumentem jest blok)

- Obiekt odbiera komunikaty i reaguje na nie wykonując akcje zdefiniowane w odpowiednich metodach.
- Wybór metody dla komunikatu po stronie odbiorcy;
  Nadawca nie ma na to wpływu (chyba, ze jest jednocześnie odbiorcą)
- Klasa też jest obiektem.
- Nowy obiekt powstaje przez wysłanie komunikatu `new` do klasy, lub jest literałem (np. `3`, `'ala ma kota'`)

```
IntegerCount new
```

# Rodzaje komunikatów (1)

Komunikaty unarne (zero parametrów)

```
odbiorca nazwa_komunikatu
licznik reset
3 negated
'abcde' size
5 factorial
```

Komunikaty binarne (1 parametr)

```
odbiorca operator parametr
x == 2
1 = 4
2 + 3
'abc' , 'de'
```

# Rodzaje komunikatów (2)

Komunikaty złożone (*keyword messages*, wiele parametrów)

```
odbiorca nazwa1: par1 nazwa2: par2 ...
tab at: 1 put: 3
licznik asBase: 8
```

Kaskada (wysłanie wielu komunikatów do tego samego odbiorcy)

```
tab at: 1 put: 'ala';
    at: 2 put: 'ma';
    at: 3 put: 'asa'
```

wsszystkie komunikaty sa kolejno wysyłane do obiektu `tab`.

# Kolejność wykonania

- Odbiorca (bądź argument) komunikatu może być wyrażeniem złożonym z innych komunikatów
- Kolejność obliczeń jest następująca:
  - wyrażenia w nawiasach
  - komunikaty unarne, od lewej do prawej
  - komunikaty binarne, od lewej do prawej
  - komunikaty złożone
- Nie ma priorytetów operatorów
- $*$ nie wiąże mocniej niż $+$.

```
self tab at: 1+2*3 put: 1+4 factorial
```

oznacza

```
(self tab) at: ((1+2)*3) put: (1+(4 factorial))
```

# Klasy i egzemplarze

- *Klasa* opisuje implementację kategorii obiektów jednego rodzaju.
- Pojedyncze obiekty należące do klasy nazywane są egzemplarzami (instancjami).
- Klasa określa postać pamięci obiektów oraz sposób ich zachowania.
- Egzemplarze klasy sa podobne zarówno w swoich publicznych, jak i prywatnych aspektach.
- Publicznymi aspektami obiektu są komunikaty tworzące jego protokół.
- Prywatnymi aspektami obiektu są jego zmienne indywiduowe oraz metody tworzące implementację protokołu.
- Zmienne indywiduowe ani metody nie są bezpośrednio dostępne dla innych obiektów.
- Klasa moze utworzyć egzemplarz w odpowiedzi na komunikat (np. `new`).

# Dziedziczenie

- W Smalltalku wyłącznie dziedziczenie pojedyncze
- Korzeniem hierarchii jest klasa **Object** (każdy obiekt jest tej klasy)
- Każda klasa z wyjatkiem **Object** ma dokładnie jedną nadklasę.
- Podklasa “widzi” atrybuty nadklasy; nie może deklarować własnych atrybutów o tej samej nazwie.
- Podklasa moze swobodnie definiować metody dla dowolnych komunikatów; może odwołać się do implementacji z nadklasy (szczegóły za chwilę).

# Metody

Metoda: funkcja opisująca reakcje obiektu na komunikat

- metoda zeroargumentowa:

```
nazwa
treść
```

  treść — zmienne tymczasowe i ciąg wyrażeń oddzielonych kropkami

- metoda jednoargumentowa (operator dwuargumentowy)

```
znak_specjalny argument
treść
```

- metoda wieloargumentowa

```
nazwa1: argument1 nazwa2: argument2 ...
treść
```

# Wyszukiwanie metod

- Wyszukiwanie metod odbywa się w górę łańcucha dziedziczenia, poczynając od odbiorcy komunikatu (wyjatek: komunikaty do `super`)
- Jeśli w klasie nie ma metody pasujacej do selektora komunikatu, kontynuujemy poszukiwanie w nadklasie.
- Jesli metoda nie została znaleziona, system wysyła do odbiorcy komunikat `doesNotUnderstand: msg` (metoda dla niego zdefiniowana jest co najmniej w klasie `Object`, ale klasa może mieć własną implementację).

# Komunikaty do samego siebie — `self`, `super`

- Obiekt może wysłać komunikat do samego siebie.
- Podstawową formą wskazania siebie jako odbiorcy jest użycie słowa `self` — wtedy wyszukiwanie metody zaczyna się (jak zwykle) od klasy rozważanego obiektu.
- Alternatywną formą jest użycie słowa `super`, wtedy wyszukiwanie metody zaczyna się od nadklasy klasy, w której zdefiniowana jest metoda wysyłajaca komunikat (ale odbiorca jest ten sam co przy `self`).
- `self` może być ponadto wynikiem metody (ale `super` nie może).
- Inaczej niz w Javie czy C++ nie można pomijać `self`; nie mozna zamiast `self f: x` napisać `f: x`

```
Object subclass: #One
test
  ^1
result1
  ^self test

One subclass: #Two
test
  ^2

x1 := One new. x2 := Two new.

x1 test                                    1
x1 result1                                 1
x2 test                                    2
x2 result1                                 2
```

```
Two subclass: #Three
result2
  ^self result1
result3
  ^super test

Three subclass: #Four
test
  ^4

x3 := Three new. x4 := Four new.

x4 result1                                    4
x3 result2                                    2
x4 result2                                    4
x3 result3                                    2
x4 result3                                    2
```

# Rodzaje metod

- klasowe
  — wykonywane są w klasie, mogą działać na zmiennych globalnych i na zmiennych klasowych
- kategorie
  - inicjalizacja zmiennych klasowych
  - tworzenie nowych obiektów
- egzemplarzowe (obiektowe)
  — wykonywane w obiektach, mogą działać na zmiennych egzemplarzowych, klasowych i globalnych
- kategorie
  - dostęp do atrybutów
  - porównywanie obiektów
  - kopiowanie
  - wyświetlanie
  - inicjalizacja obiektu
  - pomocnicze (prywatne)
  - ...

# Definiowanie klasy

```
NazwaNadklasy subclass: #NazwaKlasy
instanceVariableNames: 'zmienna1 zmienna2... '
classVariableNames: 'Zmienna1 Zmienna2 ... '
poolDictionaries: 'Pool1 Pool2 ... '
```

- Zmienne egzemplarzowe (indywidualne, obiektowe)
- Zmienne klasowe
  — atrybuty wspólne dla wszystkich obiektów danej klasy
- pule (słowniki)
  — pule zmiennych dzielonych, wspólne dla obiektów klas, w których są wymienione
- "zmienne globalne" — słownik systemowy **Smalltalk** wspólny dla wszystkich obiektów (np. nazwy klas).

**Nie ma innych zmiennych globalnych**

# Licznik

```
Object subclass: #Count
   instanceVariableNames: 'value resetValue'
   classVariableNames: ''

" Count methodsFor: 'initialize-release'"
initialize
        self reset

" Count methodsFor: 'accessing' "
resetValue
        ^resetValue
resetValue: aValue
        resetValue := aValue.
```

```
value
        ^value

value: aValue
        value := aValue.

" Count methodsFor: 'counting' "
decrement
        " Musi być zaimplementowane w podklasie"

increment
        " Musi być zaimplementowane w podklasie"

reset
        value := resetValue
```

# Metody klasowe

```
" Count class methodsFor: 'instance-creation' "

new
        ^super new initialize
```

- Klasa (tu: `Count`) też jest obiektem pewnej klasy (tu: `Count class`)
- Metody klasowe formalnie są metodami tej ostatniej klasy
- W praktyce IDE umożliwiają ich deklarację za pośrednictwem klasy `Count`

# Przykład

```
Count subclass: #IntegerCount
        instanceVariableNames: ''
        classVariableNames: ''

initialize
        resetValue := 0.
        super initialize

decrement
        value := value - 1

increment
        value := value + 1
```

# Wyrażenia

- literał
- pseudozmienne `nil true false`
- komunikat

  `odbiorca nazwa_komunikatu_z_argumentami`

  np.

  ```
  x + 3
  x wypisz: ekran
  Count new
  kolekcja add: x after: y
  ```
- przypisanie (tylko na własne zmienne obiektu), np.

  `value := value + 1`
- wyrażenie przekazujące wynik z metody:

  `^wyrażenie`
- blok np. `[ :x | x+1 ]`

# Literały

- `42` — unikalny obiekt 42
- `18.5` — liczba zmiennoprzecinkowa 18.5 (niekoniecznie unikalna)
- `'napis'` napis (nieunikalny)
- `#request` — symbol o nazwie `request`;
  dwa wystapienia symbolu o tej samej nazwie reprezentują ten sam obiekt.
- `$r` pojedynczy znak 'r' (znaki specjalne przez np. `Character cr`)
- `#(3,2.7, $x, 'a string')` tablica obiektów.

# Symbole, napisy, unikalność

Symbole są używane jako klucze w słownikach, selektory metod itp.
Różnice między symbolami a napisami:

- Symbole są unikalne, napisy nie są.
- Symbole są niemodyfikowalne, napisy — modyfikowalne.

```
#homer == #homer  true
'homer' == 'homer' false
42 == 42 true
1.0 == 1.0 false
```

# Bloki

- Bloki reprezentują anonimowe funkcje

  ```
  [:arg1 :arg2 | S1. S2. ... Sn ]
  ```

- bloki są obiektami klasy `BlockContext` (niektóre dialekty: `BlockClosure`).
- mogą być przypisywane na zmienne, przekazywane jako argumenty i dawane w wyniku (jak wszystkie obiekty).
- w miejscu wystąpienia blok nie jest wykonywany; tworzony jest tylko egzemplarz klasy BlockContext.
- Blok może być wykonany przez wysłanie doń komunikatu `value`. Daje to w wyniku wartość ostatniego wyrażenia wykonanego w bloku.

  ```
  [i:=2+2] value
  [:x :y | x+y*3] value: 2 value: 5
  ```

- Blok może mieć zmienne lokalne:

  ```
  [:arg| |locals| stmts]
  ```

- Blok może odwoływac się do zmiennych nielokalnych (otaczającego go bloku, czy metody); są one wiązane statycznie.

# Konstrukcje warunkowe

```
cond ifTrue: aBlock
cond ifFalse: aBlock
cond ifTrue: aBlock1 ifFalse: aBlock2
cond ifFalse: aBlock ifTrue: aBlock2
```

`cond` musi być obiektem klasy `True` lub `False`

```
self = 0
  ifTrue: [^self error: 'zero has no reciprocal']
  ifFalse: [^1 / self]
```

# Konstrukcje warunkowe

Formalnie `ifTrue:` etc. są zwykłymi komunikatami; zachowują się, jakby klasa `True` miała metody

```
ifTrue: trueBlock ifFalse: falseBlock
  ^trueBlock value
ifFalse: aBlock
  ^nil
etc...
```

W praktyce zwykle implementacje Smalltalka dokonują pewnych optymalizacji.

# Pętle

```
condBlock whileTrue: aBlock
condBlock whileFalse: aBlock
```

`condBlock` powinien być bezparametrowym blokiem dającym w wyniku `true` lub `false`.

```
sum := 0. num := 4.
[num <= 100] whileTrue:
                  [sum := sum + num.
                   num := num + 1]
```

Podobnie jak wyrażenia warunkowe, pętle są często optymalizowane przez kompilator.

# Pętle á la “for”

```
number to: final by: step do: aBlock
number to: final do: aBlock
```

Pętla z poprzedniego przykładu moze byc wyrażona jako

```
sum := 0.
4 to: 100 do: [:num | sum := sum + num]
```

# Powrót z bloku

Wyrażenie `^expr` powoduje powrót z metody w której blok wystepuje tekstowo (a nie z metody, która ten blok oblicza)

Klasa `Integer` mogłaby mieć metodę

```
isPrime
  |candidate divisor|
  candidate := self.
  candidate <= 1 ifTrue: [^false].
  candidate > 1 & (candidate  <= 3)
    ifTrue: [^true].
  (candidate \\ 2) = 0 ifTrue: [^false].
  divisor := 3.
  [divisor squared <= candidate] whileTrue:
    [(candidate\\divisor)=0
       ifTrue: [^false]
       ifFalse: [divisor := divisor + 2]
    ]
  ^true
```