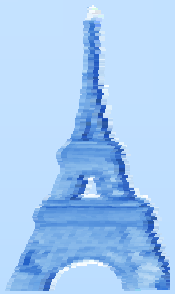
Język programowania Eiffel

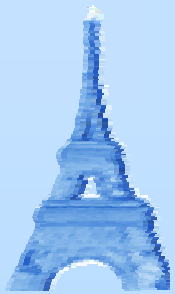
Piotr Kowalski

5 kwietnia 2004



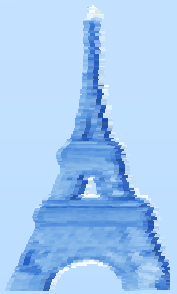
Agenda

- Geneza języka
- Metodologia Eiffel'a
- Podstawowe konstrukcje
- Obiekty i klasy
- Dziedziczenie
- Eiffel w praktyce



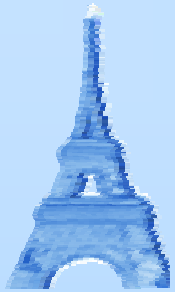
Okoliczności powstania języka

- Opracowany przez firmę Interactive Software Engineering (dziś **Eiffel Software**) w 1985 roku na własne potrzeby
- Pierwsza publiczna prezentacja środowiska **Eiffel 1** miała miejsce w październiku 1986 na konferencji OOPSLA
- Aktualnie rozwijane przez Eiffel Software środowisko programistyczne to **Eiffel Studio**
- Związek założeń projektowych języka z konstrukcją francuskiego inżyniera



Przenośność i dostępność Eiffel'a

- Eiffel Software udostępnia swoje środowisko w wersjach na wiele różnych platform, m.in. Windows (95, 98, ME, NT, 2000, XP), Linux, Unix (Solaris, SunOS, HP 9000, IBM AIX, Unixware, Silicon Graphics, Data General, Fujitsu, DEC OSF/1 etc.), VMS (Alpha, Vax)
- Kompilator Eiffel'a wykonuje tzw. **finalizację**, czyli translację kodu w Eiffel'u do kodu w standardzie ANSI-C (poprzez bajtkod pośredni), dzięki czemu możliwe jest kompilowanie i wykonywanie go na każdej platformie dysponującej kompilatorem C.

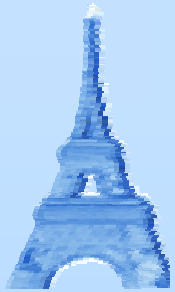


Inne środowiska

- **Visual Eiffel**
  Środowisko dla Windows firmy Object Tools wraz z bogatą biblioteką klas

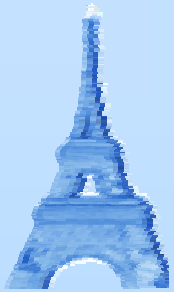
- **SmaRTEiffel**
  Środowisko na licencji GNU

- **Fine (Small Eiffel)**
  Projekt Open Source prowadzony na sourceforge.net



Główne cechy języka

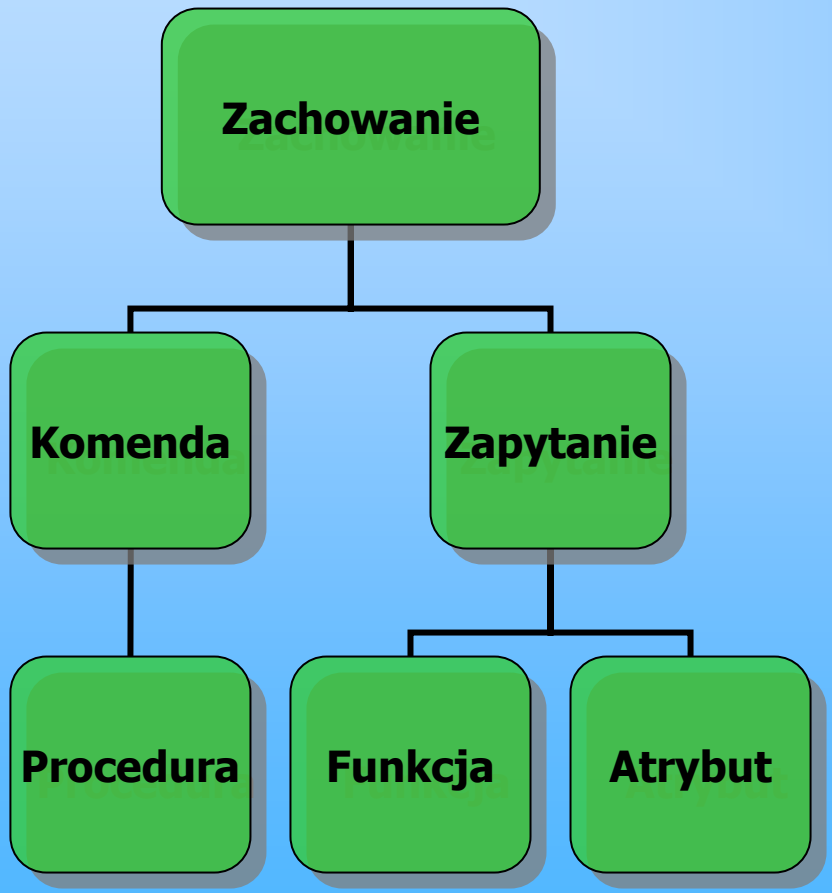
- Podczas projektowania Eiffel'a największy nacisk został położony na ścisły związek języka i metodologii projektowania oprogramowania
- Metodologia Eiffel'a to:
  - zasada client-supplier
  - pełna obiektowość języka
  - integracja kodu i metadanych służących m.in. weryfikacji działania programu (Design by Contract)
  - rozróżnianie funkcji i procedur

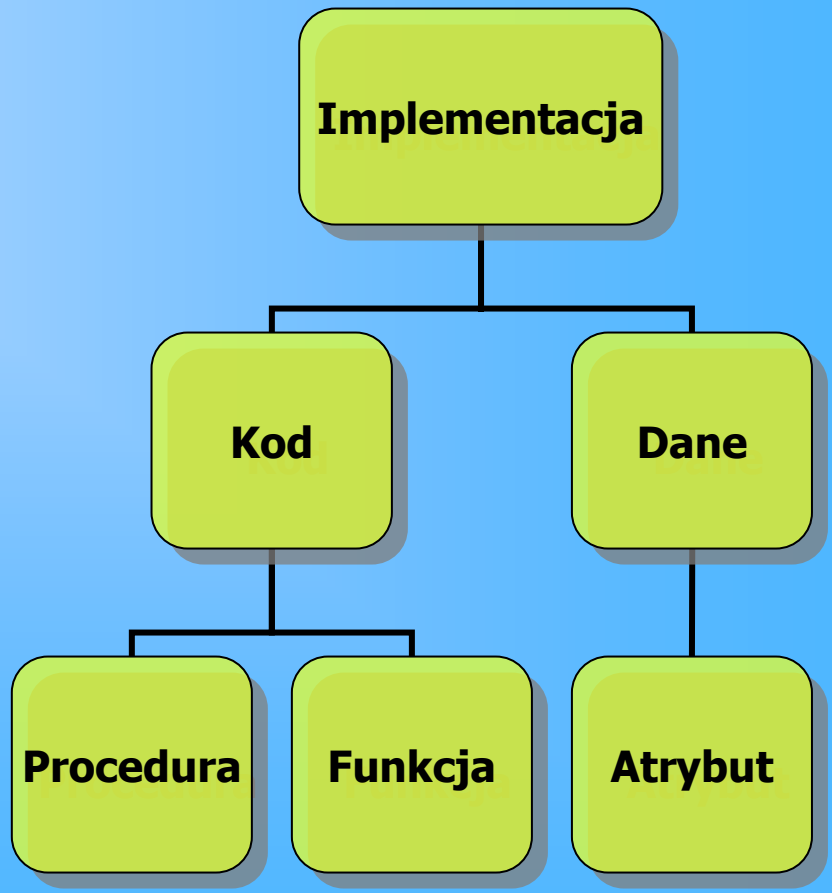


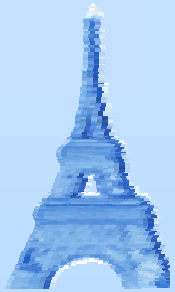
Nazewnictwo

- Programiści Eiffel'a ściśle ustandaryzowali nie tylko nazewnictwo, ale nawet zasady formatowania kodu
- Klasa składa się z **cech** (ang. *features*), czyli
  - **atrybutów** reprezentujących dane
  - **kodu** (ang. *routines*), który może być
    - **funkcją**, jeśli zwraca wartość
    - **procedurą**, jeśli nie zwraca wartości
- **zapytanie** (ang. *query*) to funkcja lub atrybut

Nazewnictwo c.d.

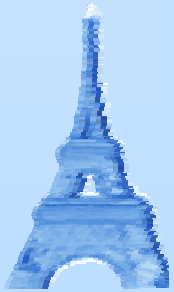






Założenia

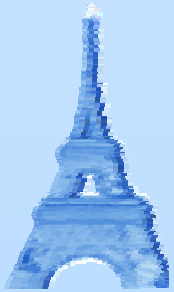
- Kod zawsze jest częścią jakiejś klasy
- Program to zbiór klas
- Uruchomienie programu to wywołanie konstruktora klasy zdefiniowanej jako główna (i tylko tyle!)
- Każda klasa w osobnym pliku tekstowym



Składnia

```
1  class ACCOUNT
2  creation
3    make
4  feature
5    balance:REAL -- The actual account balance.
6
7    make is -- Initialization
8      do
9        io.put_string („Podaj stan początkowy:”)
10       io.read_real
11       balance := io.last_real
12     end -- make
13 end -- class ACCOUNT
```

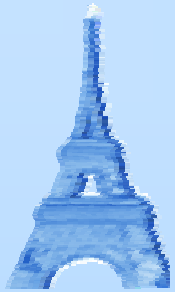
```
1  make is -- nazwa procedury
2    local -- deklaracja zmiennych lokalnych
3      i       : INTEGER; -- inicjalizowane na zero
4      max     : INTEGER;
5      fib_num : INTEGER
6    do
7      io.read_integer;
8      max := io.last_integer;
9
10     from -- pętla until, warunki początkowe (muszą być, choćby puste)
11       io.put_integer (max);
12       i := 1;
13       fib_num := fib (1)
14     until -- warunek końcowy
15       fib_num > max
16     loop -- ciało pętli (może się nie wykonać ani razu)
17       io.put_integer (fib_num);
18       io.put_character (' ');
19       fib_num := fib (i);
20       i := i + 1
21     end; -- loop
22   end -- make
```



Deklaracja zmiennych i typy danych

```
1  -- deklaracja stałej
2      Stala   : INTEGER is 23
3
4  -- wartości wyliczeniowe
5      Kier, Karo, Pik, Trefl: INTEGER is unique
```

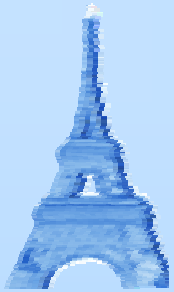
- Typy podstawowe w Eiffel'u to m.in.:
  - BOOLEAN
  - INTEGER
  - REAL
  - DOUBLE
  - CHARACTER



Funkcje

- Metodologia wymaga, aby funkcje **nie modyfikowały żadnych atrybutów** obiektów, aczkolwiek kompilatory mogą dopuszczać taką sytuację
- Deklaracja funkcji niewiele różni się od deklaracji procedury

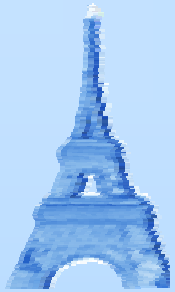
```
1  nazwa (argumenty) : TYP is
2  ...
3    do
4      ...
5      Result := wynik
6    end
```



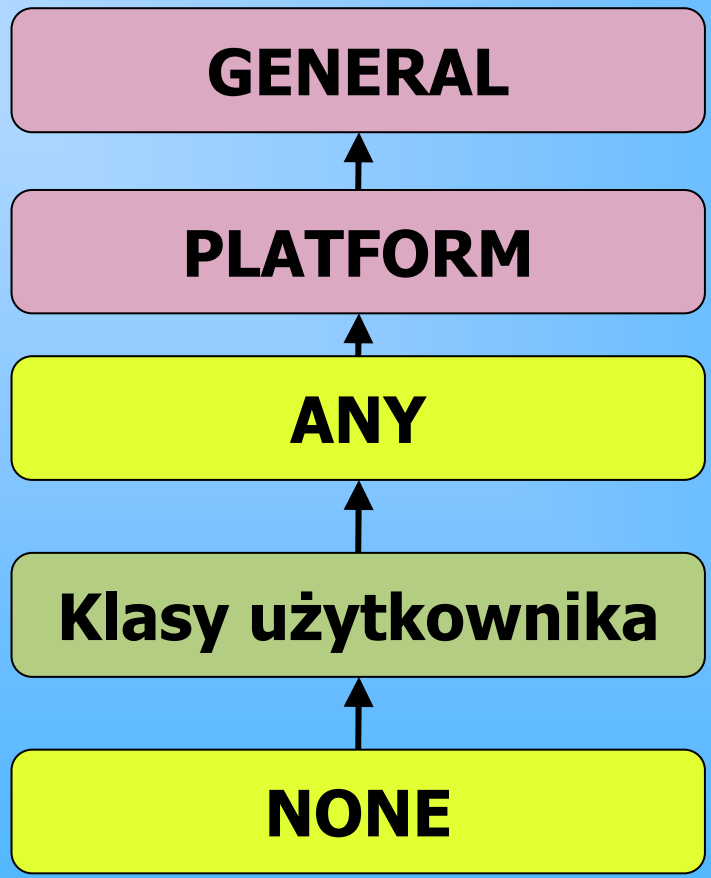
Operatory

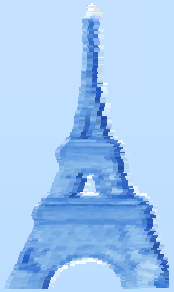
- Oprócz standardowych operatorów można definiować własne, rozpoczynające się od znaków @, #, |, &; operatory użytkownika mają najwyższy priorytet
- Do określania typu wiązania operatora służą słowa kluczowe **prefix**, **infix**, **postfix**

```
   class REAL
   ...
1  infix #percent(percent:REAL):REAL is
2    -- percent of Current
3  do
4    Result := Current * (percent / 100.0)
5  end -- #percent
```



Hierarchia klas



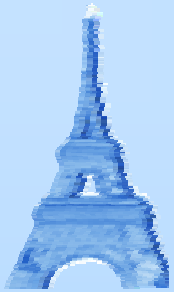


Tworzenie obiektów

- Do utworzenia obiektu służy operator **!!** „bang bang”

```
1  class LINE is
2  feature
3    x, y : POINT
4    make is
5      do
6        !!x.make
7        !!y.make
8      end
```

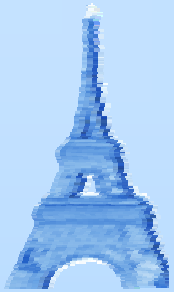
- **x** i **y** są referencjami do obiektu klasy POINT, inicjalizowanymi standardowo na wartość **Void**
- Jeśli klasa tworzonego obiektu posiada choć jeden konstruktor, to musimy podać jego nazwę



Konstruktory

- Listę konstruktorów klasy umieszczamy po słowie kluczowym **creation**
- Eiffel daje możliwość precyzowania różnych konstruktorów dla różnych klas tworzących obiekty; np. POINT może mieć inny konstruktor dla klasy LINE a inny dla EDGE
- Możemy również określić które i tylko które klasy mogą tworzyć obiekty klasy, którą piszemy

```
1  creation {LINE}
2    make_line
3  creation {EDGE}
4    make_edge
```



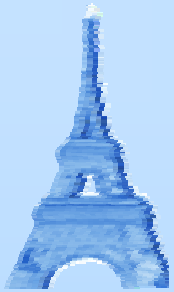
Widoczność

- Notację nawiasów klamrowych możemy również wykorzystać do określenie klas, dla których dana cecha jest widoczna:

```
1 feature {SECRET_AGENT, SECRET_HQ}
2   get_secret_value:BOOLEAN is
3     Result := true
4   end
```

- Przykłady

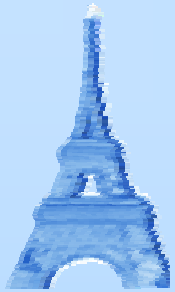
```
5 feature              -- każda klasa ma dostęp
6 feature {ANY}        -- j.w.
7 feature {A, B, C}    -- tylko klasy A, B oraz C mają dostęp
8 feature {}           -- żadna klasa nie ma dostępu
9 feature {NONE}       -- j.w.
```



Równość

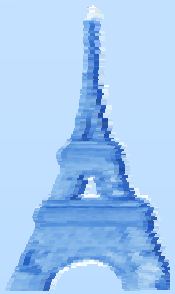
- Eiffel dzieli typy danych na dwa rodzaje:
  - Typy rozwinięte (ang. *expanded*)
  - Referencje
- Operator równości porównuje wartości samych zmiennych, a nie obiektów przez nie wskazywanych
- We wszystkich klasach określona jest operacja **equal**, która porównuje wszystkie pola w obiektach

```
1 make is
2   do
3     !!p1
4     !!p2
5     czy_rowne := equal(p1, p2)
6   end
```



Kopie obiektów

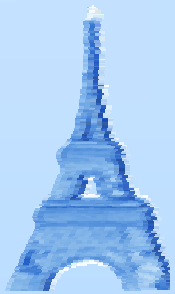
- Eiffel udostępnia dwie operacje służące do kopiowania obiektów:
  - copy
  - clone
- Wykonują one płytką (ang. *shallow*) kopię obiektu, aby utworzyć pełną kopię należy skorzystać z
  - deep_copy
  - deep_clone
- Operacje te zdefiniowane są na każdej klasie
- Mamy również możliwość głębokiego porównywania:
  - deep_equal



Asercje

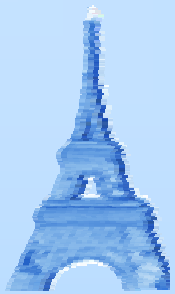
- Eiffel umożliwia precyzowanie warunków, które muszą być spełnione przed i po wykonaniu procedury/funkcji

```
1 pobierz_z_konta (ile : REAL) is
2   require
3     wiecej_niz_zero: ile > 0
4     sa_srodki: ile <= stan_konta
5   do
6     stan_konta := stan_konta - ile
7   ensure
8     zmienilo_sie: stan_konta = old stan_konta - ile
9 end -- pobierz_z_konta
```



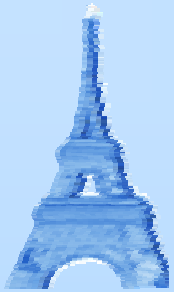
Mechanizm sprawdzania asercji

1. Przypisywane są wartości argumentów.
2. Warunki początkowe są sprawdzane, jeśli nie są spełnione to program zgłasza wyjątek.
3. Tworzone są zmienne lokalne.
4. Wykonywana jest treść procedury (funkcji).
5. Sprawdzane są warunki końcowe, jeśli nie są spełnione to program zgłasza wyjątek.
6. Sterowanie zwracane jest wołającemu.



Asercje c.d.

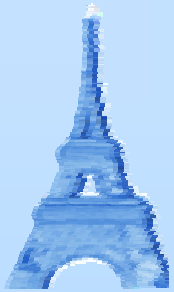
- Mechanizm asercji został stworzony w myśl zasady „jeśli podałeś mi dobre argumenty, to na pewno podam ci dobry wynik”
- Sprawdzanie asercji można ograniczyć lub wyłączyć w pliku **ACE** (*Assembly of Classes in Eiffel*) – odpowiedniku Manifestu w Javie
- Asercje możemy także precyzować dla całych klas, są to **niezmienniki** (ang. *invariants*)



Niezmienniki

- Niezmienniki to asercje, które sprawdzane są dla każdego obiektu danej klasy, przy wchodzeniu i wychodzeniu z każdej cechy z wyjątkiem konstruktora
- Warunki precyzujemy za pomocą słowa kluczowego **invariant**, zgodnie z konwencją na końcu klasy

```
1 class KONTO
2   ...
3   invariant
4     mamy_srodki: stan_konta >= 0.0
5 end -- class KONTO
```

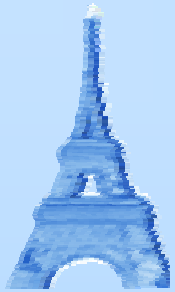


Operator implies

- Oto tabelka wartości operatora **implies**

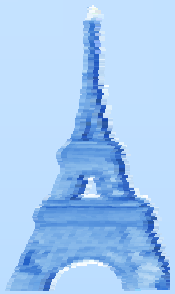
A	B	A=>B
T	T	T
T	F	F
F	T/F	F

```
1   ...
2   invariant
3     not Result implies last_value = 0
4 end
```



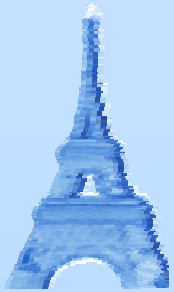
Wyjątki

- Do treści procedury lub funkcji możemy dodać klauzulę **rescue**, która jest wykonywana w przypadku, gdy podczas sprawdzania asercji zgłoszony został wyjątek
- W treści klauzuli rescue możemy używać instrukcji **retry**, powodującej ponowne wykonanie metody
- Zgłaszanie i manipulacja wyjątkami odbywa się za pomocą klas biblioteki Eiffel'a



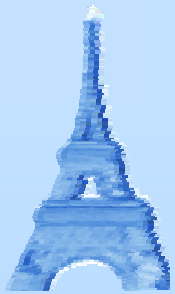
Wyjątki - przykład

```
1   ...
2   zapisz is
3     local
4       already_tried:BOOLEAN -- domyślna inicjalizacja na FALSE
3     do
4       io.put_real (3.14159)
5     rescue
6       if not already_tried then
7         already_tried := true
8         retry
9       end
10    end
```



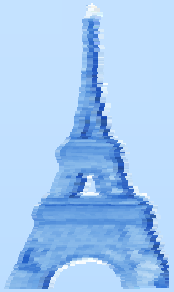
Dziedziczenie

- Eiffel umożliwia **wielodziedziczenie**
- Względem każdej klasy nadrzędnej i jej cechy możemy:
  - dziedziczyć treść cechy bez zmian
  - zmienić nazwę tej cechy w klasie pochodnej
  - zmodyfikować widoczność cechy
  - usunąć cechę z klasy pochodnej
  - przedefiniować treść cechy
  - wybrać cechę jako aktywną



Dziedziczenie - składnia

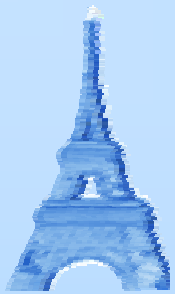
```
1  class PROSTOKĄT
2    inherit FIGURA
3      rename
4        m as n
5      export
6        {TABLICA} rysuj, wypelniaj
7      undefine
8        dlugosc
9      redefine
10       rysuj
11     select
12       make
13   end
14
15   inherit LAMANA
16   ...
```



Dziedziczenie - rename

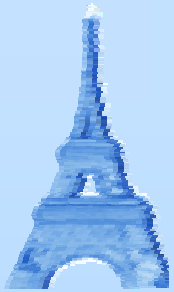
- **rename** pozwala zmienić nazwę cechy w klasie pochodnej, często konstrukcję taką stosuje się dla konstruktorów, np.:

```
1  class PROSTOKAT
2    inherit FIGURA
3      rename
4        make as make_figura
5      redefine
6        make
7    end
8    feature
9      make is
10     do
11       make_figura -- wołamy konstruktor nadrzędny
12       ... -- konstrukcja lokalna
```



Dziedziczenie - export i redefine

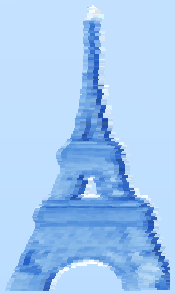
- **export** pozwala zmienić widoczność cech w klasie, uczynić je widocznymi lub zakryć
- **redefine** pozwala zmodyfikować treść cechy w podklasie, możemy zapobiec takiej sytuacji słowem kluczowym **frozen** w deklaracji cechy; do implementacji w nadklasie możemy dostać się za pomocą słowa kluczowego **Precursor**
- wiązanie wywołań procedur i funkcji jest zawsze dynamiczne
- procedury/funkcje abstrakcyjne dekorujemy słowem kluczowym **deferred**



Dziedziczenie - undefine

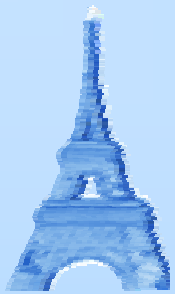
- **undefine** pozwala na usunięcie treści cechy z klasy pochodnej
- dzięki temu mechanizmowi można m.in. zmieniać sygnaturę cechy w podklasie (zachowując zgodność)

```
1  class LICZBA_RZECZYWISTA
2    inherit LICZBA_CALKOWITA
3      undefine
4        modul
5      redefine
6        modul
7    end
8    feature
9      modul(a:REAL):REAL is
10     do
11       ... -- a w CALKOWITEJ mogło być rozmiar(a:INTEGER):INTEGER
```



Dziedziczenie - select

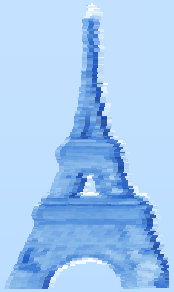
- Słowo kluczowe **select** służy do sprecyzowania, którą cechę uważamy za „aktywną”, jeśli jest więcej cech o tej samej nazwie
- W przypadku wielodziedziczenia może się zdarzyć, że będziemy dziedziczyć więcej niż raz z tej samej klasy
- Jeśli dwie cechy są identyczne co do sygnatury i treści, to kompilator nie zgłasza błędu



Select - przykład

```
 1 class KWADRAT
 2   inherit SYMETRIA
 3     ...
 4     end;
 5
 6   inherit PROSTOKAT
 7     select
 8       rysuj
 9     end
10  ...
```

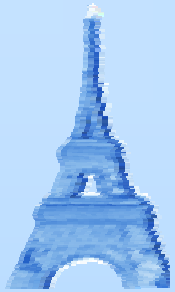
- Jeśli klasa SYMETRIA również udostępnia cechę **rysuj**, to w klasie KWADRAT zawsze wybierana będzie wersja odziedziczona po PROSTOKAT; kompilator kompletnie usuwa treść niewybranej cechy



Wiązanie

- Każda zmienna referencyjna ma dwa typy:
  - **typ statyczny**, który został zadeklarowany
  - **typ dynamiczny**, aktualny typ wskazywanego obiektu (typ statyczny lub jego podklasa)
- Eiffel udostępnia konstrukcję umożliwiającą automatyczne tworzenie obiektu podklasy i zapamiętanie referencji do niego jako nadklasy:

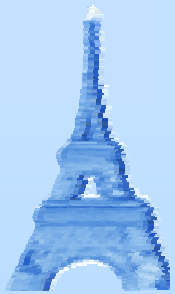
```
1 class MALARZ
2  feature
3    prostokat:PROSTOKAT
4    make is
5     do
6      !KWADRAT!prostokat -- prostokat=(PROSTOKAT)new KWADRAT();
7     end
```



Wiązanie

1. Podczas kompilacji zapamiętywany jest wskaźnik do definicji cechy pobrany z typu obiektu czasu kompilacji.
2. W czasie wykonania, wskaźnik ten uznawany jest za początek przeszukiwania grafu dziedziczenia.
3. Eiffel podąża w głąb ścieżką dziedziczenia tej cechy, aż dojdzie do aktualnej klasy obiektu.

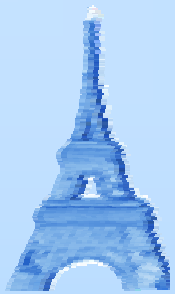
Jeśli cecha została przenazwana za pomocą **rename**, to nie zostanie odnaleziona podczas tego przeszukiwania; kompilator podąża wyłącznie za słówkiem **redefine**.



Bezpieczne przypisanie

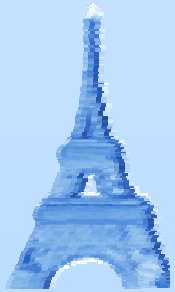
- Eiffel udostępnia instrukcję **a ?= b**, która:
  - Jeśli przypisanie może zostać wykonane przypisuje **b** na **a**
  - Jeśli typy obu stron nie są zgodne, to a otrzymuje wartość **Void** (typu NONE)
- Za pomocą tej instrukcji łatwo możemy sprawdzić, czy obiekt jest jakieś klasy czy nie

```
1   a:KWADRAT;
2   b:PROSTOKAT
3   ...
4     a ?= b
5     if (a = Void) ...
```



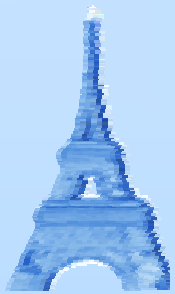
Eiffel w praktyce

- Wysoka wydajność kodu
- Eliminacja częstych błędów programistycznych
- Doskonała integracja z innymi technologiami:
  - C/C++
  - Java
  - COM
  - .NET



Źródła

- www.eiffel.com - witryna Eiffel Software
- www-staff.socs.uts.edu.au/~rist/ - elektroniczna wersja książki „Object-oriented programming in Eiffel”
- www.object-tools.com - strona domowa Visual Eiffel’a
- sourceforge.net/projects/fine - strona projektu Fine na sourceforge’u
- smarteiffel.loria.fr - SmartEiffel (OpenEiffel)



Dziękuję!