Programowanie Obiektowe

C++ podstawy

Dariusz Brzeziński

Politechnika Poznańska, Instytut Informatyki

C++

- zaprojektowany jako rozszerzenie języka C o obiektowe mechanizmy abstrakcji danych
- jest to język pozwalający na programowanie zarówno proceduralne jak i obiektowe
- C++ posiada bardzo rozbudowaną składnię
- posiada rzadkie w innych językach obiektowych: dziedziczenie wielobazowe, unie, bezpośrednie zarządzanie wolną pamięcią, operacje arytmetyczne na wskaźnikach

Strumienie

```
#include "stdafx.h"
#include <iostream>
using namespace std;


int main (void)
{
    char yourname [80];
    cin >> yourname;
    cout << "Hello " << yourname << "!" << endl;
    system("pause");
    return 0;
}
```

Strumienie (iostream):

- wejściowy: cin *(klawiatura) // nie wczytuje spacji!*
- wyjściowy: cout *(ekran)*

Operatory:

>> - wejście

<< - wyjście

Dynamiczny przydział pamięci

- w C++ pojawiają się nowe operatory new i delete
- odpowiedniki z C: malloc, calloc, realloc, free
- dynamicznie przydzielaną pamięć należy zwalniać, gdy wiadomo, że nie będzie już potrzebna
- jeśli dynamiczne przydzielenie pamięci nie powiedzie się rzucany jest wyjątek

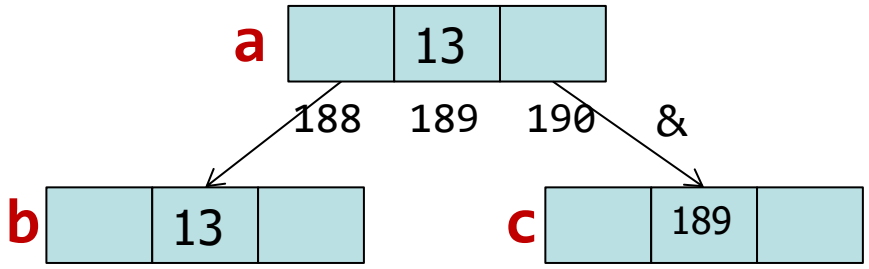
Dynamiczny przydział - przykład

```
#include "stdafx.h"
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "Ile liczb chcesz wprowadzic? : ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == 0)
      cout << "Blad: nie można zaalokowac pamieci!";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Wpisz liczbe: ";
      cin >> p[n];
    }
    cout << "Wprowadziłes nastepujace liczby: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  system("pause");
  return 0;
}
```

Wskaźniki

- Operator referencji &
  - `&zmienna` można czytać jak „adres zmiennej”
  - `a = 13; b = a; c = &a;`



- Operator dereferencji *
  - `*zmienna` można czytać jako „wartość wskazywana przez zmienną/adres”
  - `d = *c; (d == 13)true; //true`

Zmienne wskaźnikowe

Aby rozróżniać na co wskazuje wskaźnik wprowadza się zmienne wskaźnikowe.

```
#include "stdafx.h"
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  cout << "firstvalue is " << firstvalue << endl;
  cout << "secondvalue is " << secondvalue << endl;
  system("pause");
  return 0;
}
```

Operacje na wskaźnikach (ściąga)

Wyrażenie	Można czytać jako
*x	Wartość wskazywana przez x
&x	Adres x
x.y	Pole y obiektu/struktury x
x->y	Pole y obiektu/struktury wskazywanej przez x
(*x).y	Pole y obiektu/struktury wskazywanej przez x
x[0]	Pierwszy obiekt wskazywany przez x
x[1]	Drugi obiekt wskazywany przez x
x[n]	(n+1)-wszy obiekt wskazywany przez x

Klasy

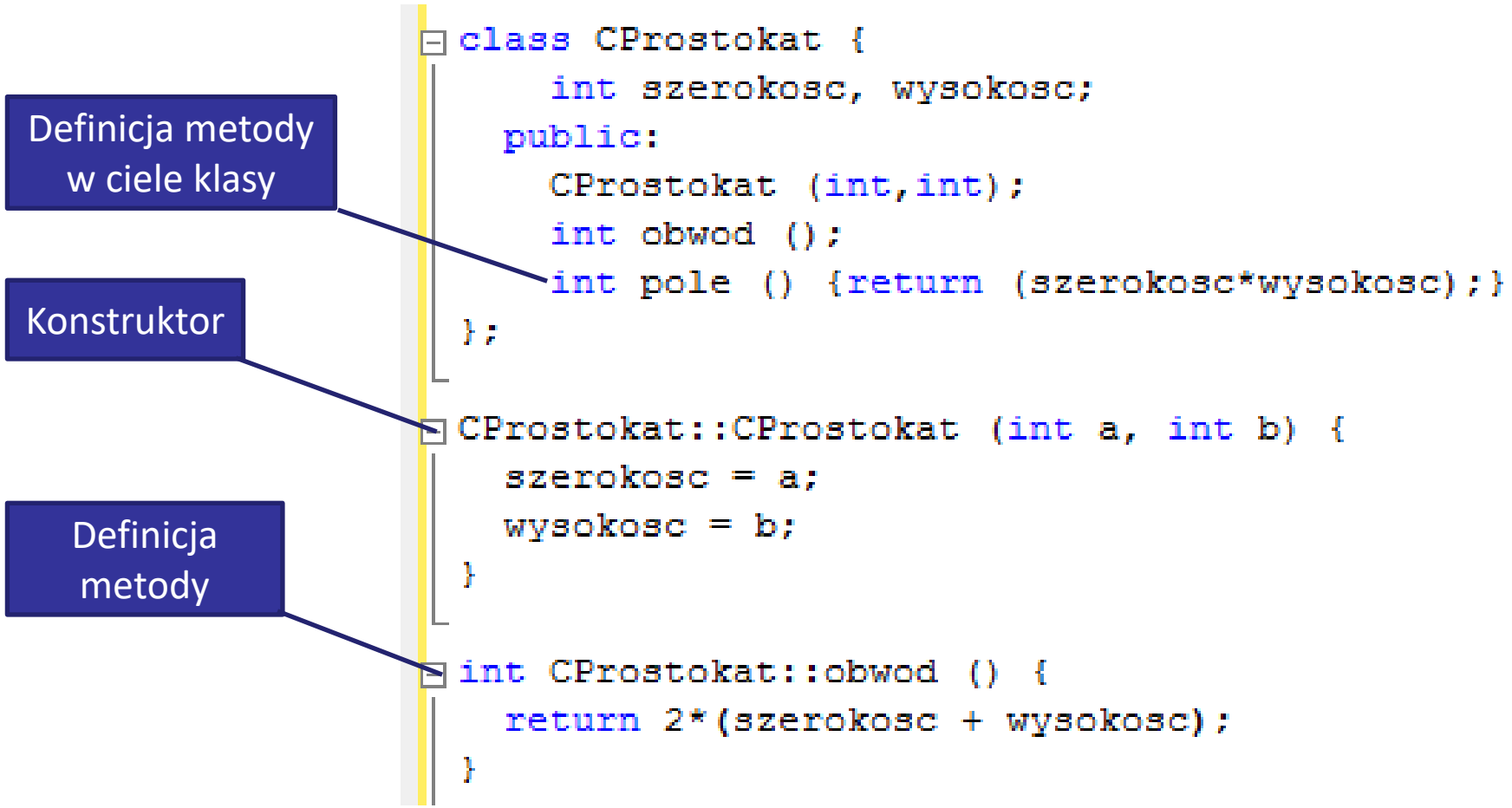
```
class nazwa_klasy {
  modyfikator_dostępu:
    składowa1;
  modyfikator_dostępu:
    składowa2;
  ...
} [nazwy obiektów];
```

- Modyfikatory dostępu:
  - `public` widoczne dla wszystkich
  - `protected` widoczne dla obiektów tej samej klasy i klas dziedziczących
  - `private` widoczne dla obiektów tej samej klasy

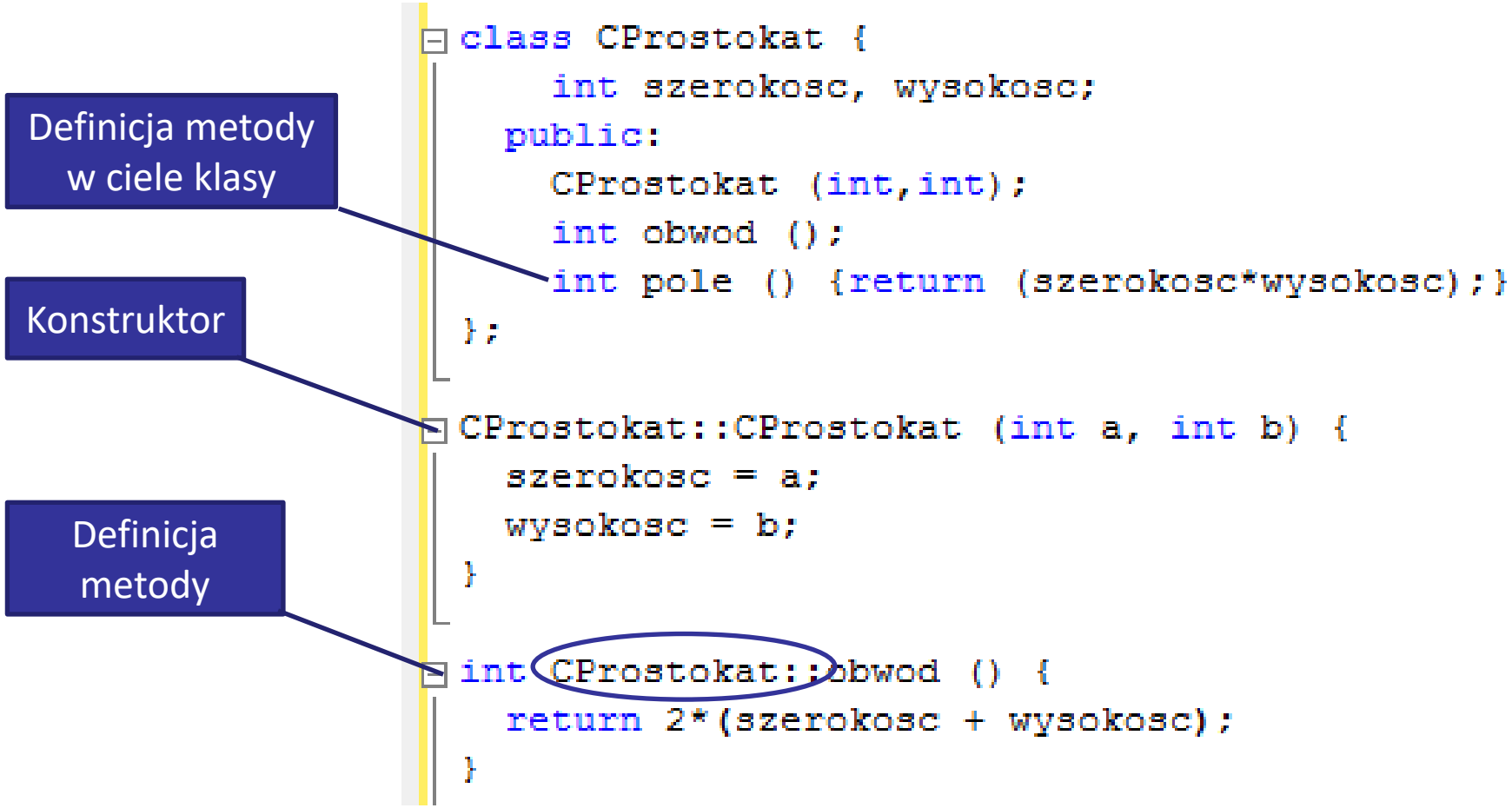
- Klasy można dzielić na deklarację (w pliku *.h) oraz definicję w pliku (*.cpp)

- W C++ struktury i unie też mogą służyć do definiowania klas

Definicje metod i konstruktory



Definicje metod i konstruktory



Przykład

```
class A
{
  public:
    int a;
    void metoda() {}
  protected:
    int b;
  private:
    int c;
};

void main()
{
  A obiekt;
  A *wsk = new A();

    obiekt.a = 1;     // OK
// obiekt.b = 1;     // błąd
// obiekt.c = 1;     // błąd
    obiekt.metoda(); // OK

    wsk->a = 1;     // OK
// wsk->b = 1;     // błąd
// wsk->c = 1;     // błąd
    wsk->metoda(); // OK

  delete wsk;
```

Działanie modyfikatorów dostępu

Wywołania domyślnego konstruktora. Konstruktor domyślny istnieje dopóki programista nie zadeklaruje własnego konstruktora

Destruktory

Desktruktory są miejscem zwalniania dynamicznie przydzielonej pamięci

```
class CProstokat {
      int *szerokosc, *wysokosc;
  public:
    CRectangle (int,int);
    ~CRectangle ();
    int area () {return (*szerokosc * *wysokosc);}
};

CProstokat::CProstokat (int a, int b) {
  szerokosc = new int;
  wysokosc = new int;
  *szerokosc = a;
  *wysokosc = b;
}

CProstokat::~CProstokat () {
  delete szerokosc;
  delete wysokosc;
}
```

Konstruktor kopiujący

```
CProstokat::CProstokat (const CProstokat& rv) {
   szerokosc=rv.szerokosc;   szerokosc=rv.szerokosc;
   }
```

Słowo kluczowe **this**

Słowo kluczowe **this** zwraca wskaźnik na obiekt, której funkcja składowa jest właśnie wykonywana

Składowe statyczne **static**

Składowe statyczne klasy oznacza się słowem kluczowym **static**. W ciele klasy znajduje się jednynie deklaracja zmiennej. Definicja powinna odbyć za ciałem klasy!

Dziedziczenie

- Mechanizm pozwalający na współdzielenie kodu między klasami
- Deklaracja:

  `class` klasa_dziedzicząca **:** `public` klasa_nadrzędna
- Ponownie trzy modyfikatory dostępu przy dziedziczeniu:
  - `public` składowe dziedziczone zgodnie z modyfikatorami w klasie nadrzędnej
  - `protected` składowe public zamieniają się na `protected`
  - `private` wszystkie dziedziczone składowe są `private`

Przykład

Prostokąt i trójkąt dziedziczą pola szerokość i wysokość i mogą korzystać z nich w metodzie pole

Prostokąt i trójkąt odziedziczyły również publiczną metodę ustaw_wartosci

```
class CWielokat {
  protected:
    int szerokosc, wysokosc;
  public:
    void ustaw_wartosci (int a, int b)
      { szerokosc=a; wysokosc=b;}
  };

class CProstokat: public CWielokat {
  public:
    int pole ()
      { return (szerokosc * wysokosc); }
  };

class CTrojkat: public CWielokat {
  public:
    int pole ()
      { return (szerokosc * wysokosc / 2); }
  };

int main () {
  CProstokat rect;
  CTrojkat trgl;
  rect.ustaw_wartosci (4,5);
  trgl.ustaw_wartosci (4,5);
  cout << rect.pole() << endl;
  cout << trgl.pole() << endl;
  return 0;
}
```

Abstrakcja

Prostokąt i trójkąt mogą być traktowane jak wielokąty

Ale metoda pole, choć nazwana tak samo jest charakterystyczna dla każdej klasy

```
int main () {
   CProstokat rect;
   CTrojkat trgl;
   CWielokat * ppoly1 = &rect;
   CWielokat * ppoly2 = &trgl;
   ppoly1->ustaw_wartosci (4,5);
   ppoly2->ustaw_wartosci (4,5);
   cout << rect.pole() << endl;
   cout << trgl.pole() << endl;
   return 0;
}
```

Polimorfizm

```
class CWielokat {
  protected:
    int szerokosc, wysokosc;
  public:
    void ustaw_wartosci (int a, int b)
      { szerokosc=a; wysokosc=b;}
    virtual int pole ()
      { return (0); }
  };

class CProstokat { ... };
class CTrojkat { ... };

int main () {
  CProstokat rect;
  CTrojkat trgl;
  CWielokat * ppoly1 = &rect;
  CWielokat * ppoly2 = &trgl;
  ppoly1->ustaw_wartosci (4,5);
  ppoly2->ustaw_wartosci (4,5);
  cout << ppoly1->pole() << endl;
  cout << ppoly2->pole() << endl;
  return 0;
}
```

Jeśli przedefiniujemy klasę CWielokat, by zawierała *wirtualną* metodę pole()…

… to otrzymujemy polimorfizm, czyli różne zachowanie tej samej funkcji wywołanej dla różnych klas dziedziczących

Klasy abstrakcyjne

- Klasy abstrakcyjne zawierają nieza-implementowane metody wirtualne

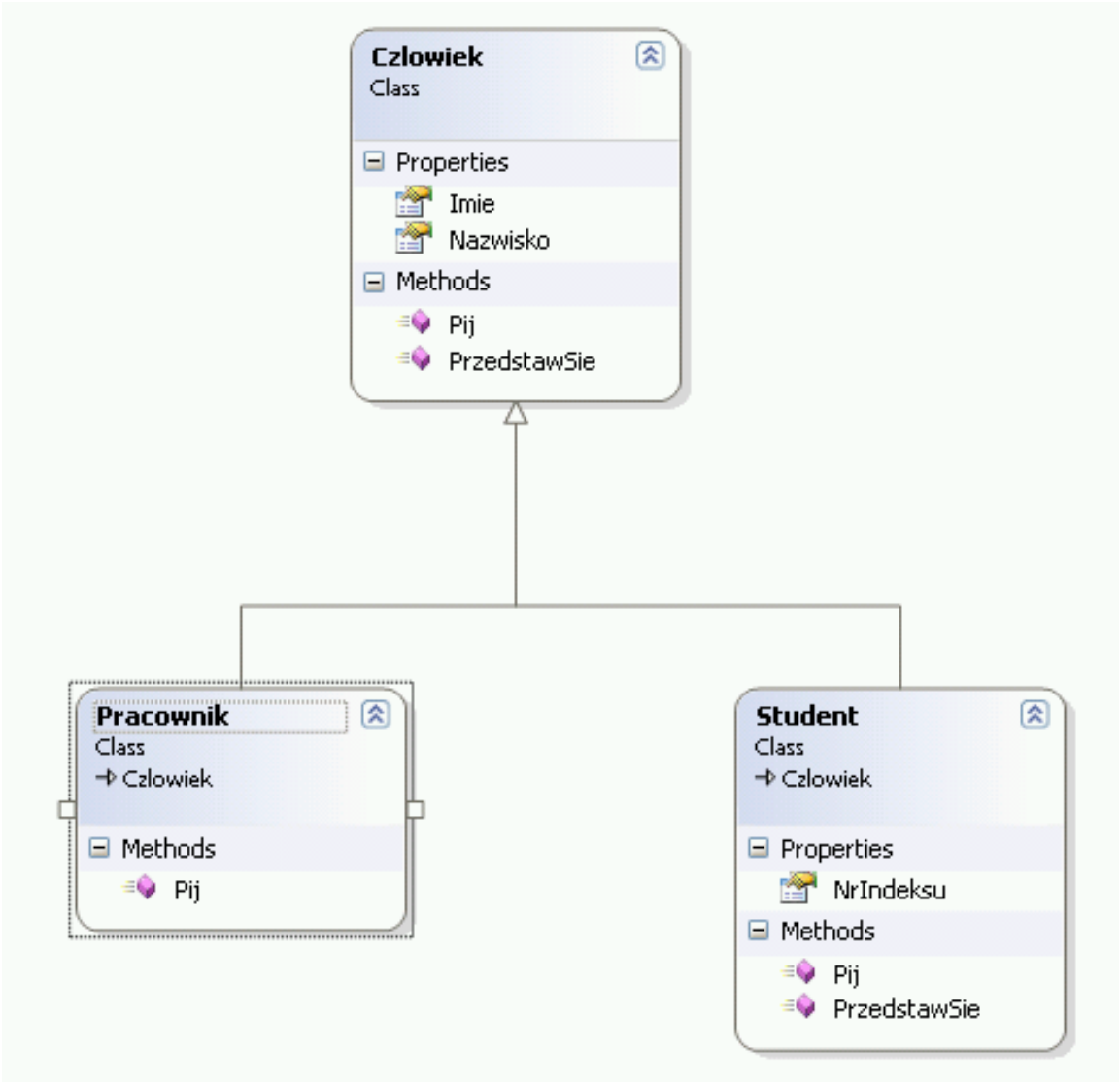
```
class CWielokat {
  protected:
    int szerokosc, wysokosc;
  public:
    void ustaw_wartosci (int a, int b)
      { szerokosc=a; wysokosc=b;}
    virtual int pole ()=0;
  };
```

- Wymusza to implementację tych metod w klasach dziedziczących
- Nie można tworzyć wystąpień klasy abstrakcyjnej!!

Zadanie

- Napisać program, który:
  - z konsoli odczytuję liczbę (długość kolejki przy ladzie)
  - tworzy tablicę (kolejkę przy ladzie) obiektów typu Czlowiek
  - wypełnia kolejkę pracownikami i studentami
  - każe przedstawić się i zamówić/wypić odpowiednią liczbę drinków
  - wynik przedstawienia się i zamówienia wyświetla na konsoli

Zadanie


Czlowiek
Class
Properties
Imie
Nazwisko
Methods
Pij
PrzedstawSie
Pracownik
Class
Czlowiek
Methods
Pij
Student
Class
Czlowiek
Properties
NrIndeksu
Methods
Pij
PrzedstawSie