Języki programowania
JPR222

Plan wykładu JPR222

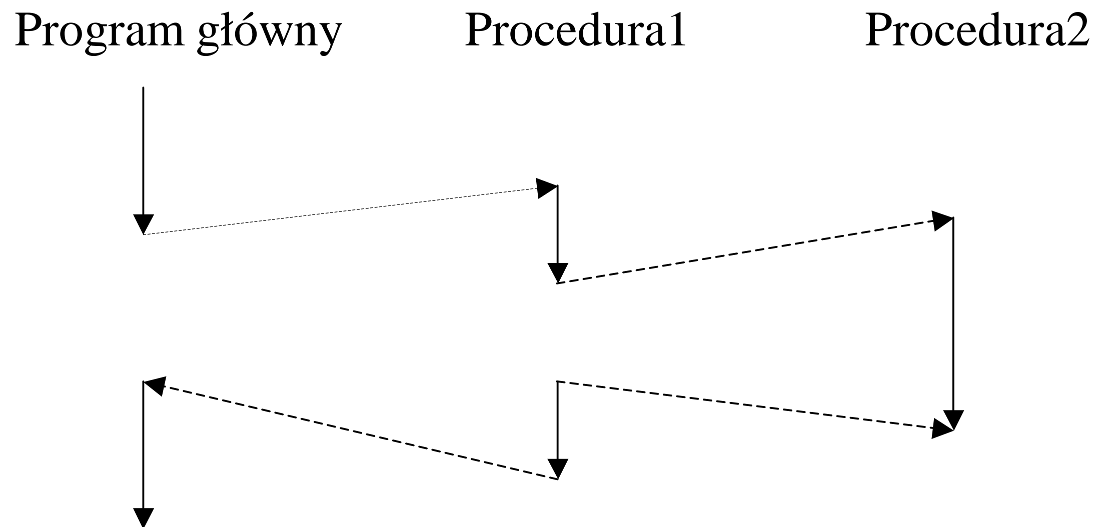
1. Pojęcia OOP (Object Oriented Programming)
2. język C++, OOP w C++, elementy nieobiektowe C++
3. biblioteka standardowa C++ (IO Streams, STL)
4. język Java, OOP w Javie
5. (wybrane?) std pakiety Javy
6. środowisko .NET, język C#
7. język skryptowy/dynamiczny Tcl;
   programowanie komponentowe w Tcl (łączenie C/C++ i Tcl)
8. OOP w Tcl - XOTcl

Techniki programowania czyli sposoby organizacji kodu programu:

Programowanie proceduralne
Programowanie modułowe
Programowanie zorientowane obiektowo (OOP)

każda technika ma pewne niedostatki które usuwa kolejna technika !!!

Programowanie proceduralne:



1. powtarzające się fragmenty kodu umieszcza się w procedurach;
   program główny to ciąg wywołań procedur
2. procedury komunikują się z programem głównym przy pomocy parametrów (in/out !)
   i zwracanych wartości; także przez zmienne globalne!
3. jak się realizuje procedury (z pkt widzenia assemblera):
   stos, parametry i zm lokalne oraz adr powrotny w ramce stosu

Prog. proceduralne; przykład: „lista elementów”; język C

```
#include <stdio.h>

struct Element { int elem; Element* nast; };
Element *wskNaPoczatek= 0;
Element *wskNaKoniec= 0;
Element *wskNaElemBiezacy= 0;

void lista_dolacz(int elem) { /* implementacja */}
bool lista_jestNastepny() { }
int lista_czytajElement() { }
void lista_przewin() { } // czytanie elementów od początku
void lista_czysc() { }

int main(int argc, char* argv[]) {
  lista_dolacz(123); lista_dolacz(321);
  while( lista_jestNastepny() ) {
    printf("%d, ", lista_czytajElement());
  }
  return 0;
}
```

Programowanie modułowe:

1. powiązane procedury i zmienne umieszczamy w *modułach*
2. moduł może zawierać *publiczne* i *prywatne* zmienne i procedury
   prywatne są ukryte przed innymi modułami
   to jest tzw *ukrywanie implementacji*
   inżynieria oprogramowania NALEGA
   aby ukrywać implementację !!!
3. ten sam moduł może być włączony do wielu programów
   forma *powtórnego użycia kodu !!!*
4. w języku C/C++ moduły to osobne pliki źródłowe, które mogą być oddzielnie kompilowane (tego się wymaga od modułów!)
5. w języku C/C++ zmienne globalne (zewnętrzne) oraz funkcje zdefiniowane ze słowem **static** sa prywatne tj NIE są dostępne dla innych modułów wchodzących w skład programu

Programowanie modułowe: „lista elementów”; język C

```
// plik nagłówkowy lista.h; to jest interfejs modułu „lista”
void lista_dolacz(int elem); // prototypy funkcji
bool lista_jestNastepny();
int lista_czytajElement();
void lista_przewin();
void lista_czysc();
```

```
// moduł klient
#include <stdio.h>
#include “lista.h”

int main(int argc, char* argv[]) {
  lista_dolacz(123);
  lista_dolacz(321);
  while( lista_jestNastepny() ) {
    printf("%d, ",
      lista_czytajElement()
    );
  }
  return 0;
}
```

```
// moduł „lista” (dostawca usługi)
#include <stdio.h>
#include “lista.h”
struct Element {
  int elem; Element* nast;
};
static Element *wskNaPoczatek= 0;
static Element *wskNaKoniec= 0;
static Element *wskNaElemBiezacy= 0;

// implementacja funkcji z lista.h
void lista_dolacz(int elem) {
  /* implementacja */
}
bool lista_jestNastepny() {}
int lista_czytajElement() {}
void lista_przewin() {}
void lista_czysc() {}
```

Programowanie modułowe: „lista elementów”; język C

Wady modułowej wersji „listy”:

a. moduł tworzy tylko jedną listę
b. elementy listy są konkretnego typu
   (lepiej żeby były dowolnego typu)

Z tymi wadami można sobie poradzić na gruncie programowania proceduralnego/modułowego, ale nie są to dobre rozwiązania ...

Poniższy kod usuwa (nieelegancko i niebezpiecznie) wadę „a”.

```
// plik nagłówkowy lista.h
void* lista_utworz(); // zwraca uchwyt listy
           // uchwyt jest typu (void*); dlaczego???
void lista_usun(void* uchwytListy)

void lista_dolacz(void* uchwytListy, int elem);
bool lista_jestNastepny(void* uchwytListy);
int lista_czytajElement(void* uchwytListy);
void lista_przewin(void* uchwytListy);
void lista_czysc(void* uchwytListy);
```

Programowanie modułowe: „lista elementów”; język C

```
// moduł klient
#include <stdio.h>
#include “lista.h”

int main(int argc, char* argv[]) {
  void *h1= lista_utworz();
  void *h2= lista_utworz();
    // co się stanie jeśli
    // zapomnimy utworzyć listę ???
  lista_dolacz(h1, 123);
  lista_dolacz(h1, 321);
  while( lista_jestNastepny(h1) ) {
    printf("%d, ",
      lista_czytajElement(h1)
    );
  }
  lista_usun(h1);
  lista_usun(h2);
  return 0;
}
```

```
// moduł „lista” (dostawca usługi)
#include <stdio.h>
#include “lista.h”
struct Element { /* … */ };
struct Lista {
  Element *wskNaPoczatek;
  Element *wskNaKoniec;
  Element *wskNaElemBiezacy;
};
void* lista_utworz() {
  Lista *h= new Lista;
  h->wskNaPoczatek= 0;
  h->wskNaKoniec= 0;
  h->wskNaElemBiezacy= 0;
  return h;
}
// implementacja funkcji z lista.h
void lista_dolacz(
    void* uchwytListy, int elem) {
  Lista* lista= (Lista*) uchwytListy;
      // co się stanie jeśli przekazano
      // nieprawidłowy uchwyt ???
  // ...
}
```

Idziemy w stronę programowania obiektowego ...

Chcemy aby „lista” zachowywała się jak dana zwykłego typu, np. int

```
int i; i=123; i=i+10; i--;
Lista L1,L2; L1.dolacz(123); L2.dolacz(321);
Lista L3; L3=L1+L2; // !!!
```

Co to jest „typ danej”?

Możliwa odpowiedź 1: „typ” to zbiór możliwych wartości danej

Możliwa odpowiedź 2: „typ” to zbiór operacji (=interfejs) jakie można wykonać na danej.

⇒Typ zdefiniowany poprzez podanie zbioru operacji na danej nazywamy **ADT** (=Abstract Data Type); abstrakcyjny typ danych.

⇒Implementacja ADT (co to jest?) nie jest widoczna dla użytkowników; użytkownik widzi wyłącznie interfejs !!!

⇒OOP jako narzędzie definiowania ADT ???

⇒W językach C++/Java/C# ADT jest nazywany klasą (class)

Terminologia OOP:

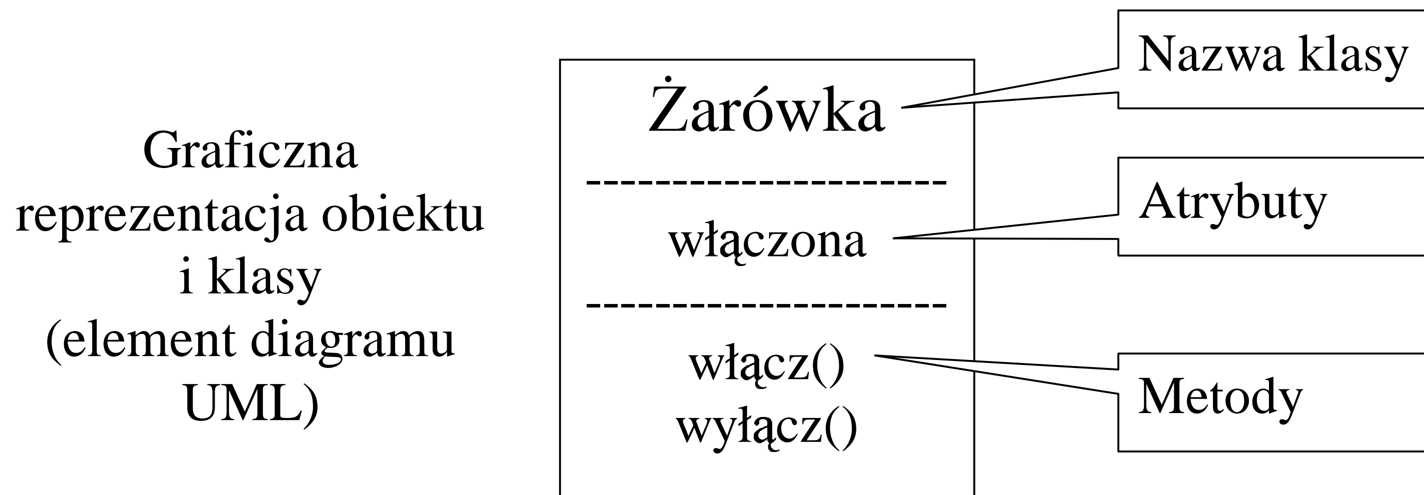
⇒ **Klasa** to ADT; klasa jest typem.

⇒ Zmienne klasy nazywamy **obiektami**; obiekty są też nazywane „egzemplarzami” lub „instancjami” klasy.

⇒ **Metody** to operacje zdefiniowane w klasie; wywołanie metody na rzecz obiektu to „przesłanie komunikatu do obiektu”.

⇒ **Atrybuty** to zmienne zdefiniowane w klasie; zwane także „polami”; przechowują „stan obiektu”.

⇒ „Interfejs obiektu” to publiczne metody klasy obiektu



Terminologia OOP:

⇒ **Hermetyzacja** to ukrycie atrybutów i niepublicznych metod przed użytkownikiem obiektu -innymi słowy jest to ukrywanie implementacji; zwana także enkapsulacją lub kapsułkowaniem; dzięki hermetyzacji użytkownik nie może „uszkodzić" obiektu ... **dlaczego?**

Pozostałe fundamentalne pojęcia to fundamenty OOP:

⇒ **Dziedziczenie** to definiowanie jednej klasy na bazie drugiej

⇒ **Polimorfizm** „cos tak samo wygląda ale inaczej działa"

OOP; jak się definiuje klasę C++ ? (przykład „lista”)

```
class Lista {
public:
  // konstruktor
  Lista() {
    wskNaPoczatek= 0;
    wskNaKoniec= 0;
    wskNaElemBiezacy= 0;
  }
  // destruktor
  ~Lista() { czysc(); }

  // publiczne metody = interfejs obiektu
  void dolacz(int elem);
  bool jestNastepny() { return wskNaElemBiezacy!=0; }
    // metoda zdefiniowana bezpośrednio w klasie
  int czytajElement();
  void przewin();
  void czysc();

private:
  Element *wskNaPoczatek;
  Element *wskNaKoniec;
  Element *wskNaElemBiezacy;
};
```

Konstruktor
wywoływany automatycznie w chwili tworzenia obiektu; inicjalizuje pola obiektu, alokuje dodatkowa pamięć dla obiektu itp

Destruktor
Wywoływany automatycznie w chwili niszczenia ob. Powinien zwalniać dodatkową pamięć obiektu itp.

Metody publiczne
Czyli interfejs obiektu

Ukryte (prywatne) **atrybuty** obiektu.

OOP; jak się definiuje klasę C++?

Brakuje implementacji niektórych metod ...

```
struct Element { int elem; Element* nast; };

void Lista::dolacz(int elem) {
  Element *e= new Element;
  e->elem=elem; e->nast=0;
  if( wskNaKoniec!=0 ) wskNaKoniec->nast=e;
  wskNaKoniec=e;
  if(wskNaPoczatek==0) { wskNaPoczatek=e; wskNaElemBiezacy=e; }
    // do pól obiektu odwołujemy się bez żadnych dodatkowych zabiegów !!!
}
int Lista::czytajElement() {
  int e= wskNaElemBiezacy->elem;
  wskNaElemBiezacy= wskNaElemBiezacy->nast;
  return e;
}
void Lista::czysc() {
  printf("Lista, %p, czysc; na razie nie zaimplementowane\n"
    "powoduje wyciek pamieci !!!!\n", this);
}
// ...
```

OOP; jak się używa klasy C++?

```
/* definicja klasy Lista ma być tutaj */

int main() {
  Lista L1, L2;
    // tworzymy dwa obiekty L1 i L2 klasy Lista

  L1.dolacz(123); L1.dolacz(321);
    // wywołujemy metody na rzecz obiektów
  while( L1.jestNastepny() ) {
    printf("%d, ", L1.czytajElement());
  }
    // ten kod powinien wypisać elementy listy
}
```

pokazać działający przykład ...

Skąd się biorą obiekty i klasy?

⇒ wg OOAD (=Analiza i projektowanie zorientowane obiektowo) obiekty programu powinny być odpowiednikami „konceptów” *dziedziny* w której rozwiązujemy jakiś problem; przez „koncepty” rozumiem pojęcia, rzeczy itp. występujące w dziedzinie.
- o Przewaga OO nad innymi technikami programowania: pozwala posługiwać się pojęciami/konceptami dziedziny!!!

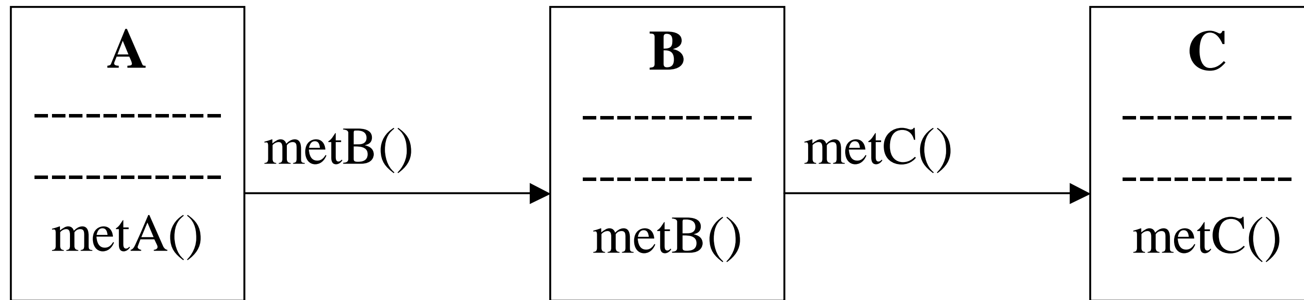
Czym jest program w OOP?

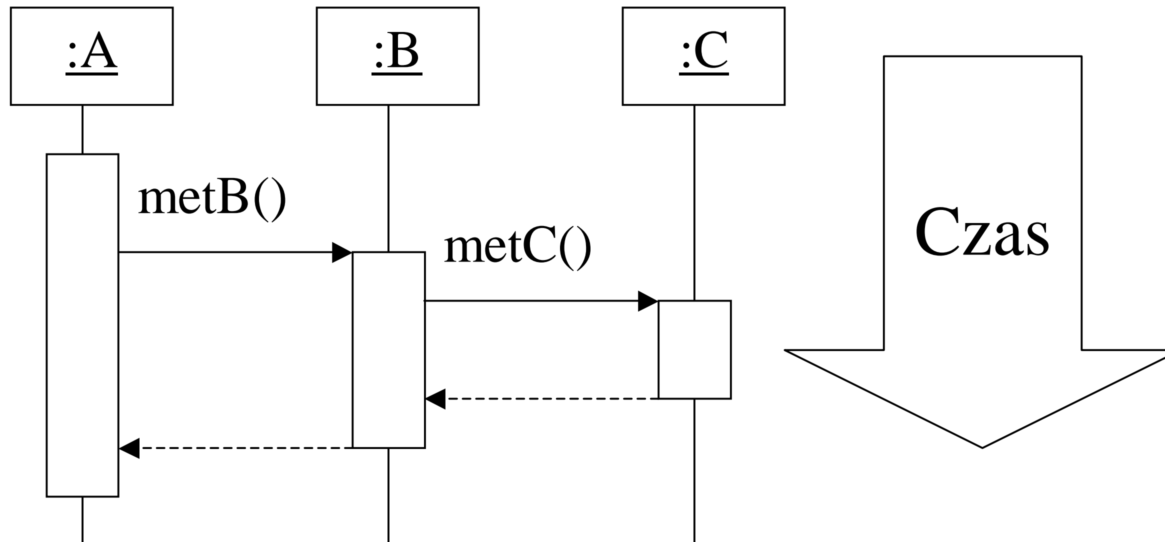
⇒ wg OOP program jest zbiorem obiektów, które przesyłają sobie komunikaty ...

⇒ obiekt może składać się z innych obiektów („niższego rzędu”); jednak patrząc na złożony obiekt widzę prostą rzecz - jego interfejs
- o Dlatego OOP jest sposobem na radzenie sobie ze złożonością oprogramowania!

⇒ *Wnioski*: aby obiekt mógł wysyłać komunikaty do innych obiektów musi posiadać ref/wsk do tych obiektów (jako artybuty); w OOP **nie** używa się zmiennych globalnych!

Diagramy przedstawiające współpracę obiektów:





Klasy C++ - widoczność składowych:

Składowe klasy to:
- zmienne składowe (atrybuty wg terminologii OOP)
- funkcje składowe (metody)

Widoczność składowych:
- **public**
  widoczne dla wszystkich
- **private**
  widoczne tylko dla metod tej samej klasy
- **protected**
  widoczne w klasach dziedziczących z danej klasy

```
class CCC {
  public:
    void setInt(int i) { i=I; }
    int getInt() { return I; }
  private:
    int i;
};
```

Kontrolowany dostęp do atrybutów obiektu!!!

Klasy C++ - czas życia obiektu

```
void fun() {
  CCC c; // obiekt jest tworzony (wywołanie konstruktora)
    // przechowywany na stosie (jak zmienne automatyczne j. C)
  c.setInt(123);
  cout<< c.getInt() <<endl;
    // obiekt jest niszczony (wywołanie destruktora)
      // w rzeczywistości obiekt jest niszczony
      // gdy sterowanie wyjedzie poza {} w którym
      // obiekt zdefiniowano!
      // (jak zmienne automatyczne języka C)
}
int main() {
  fun();
}
```

Klasy C++ - czas życia obiektu; obiekty dynamiczne

```
int main() {
  CCC *pc;
  pc= new CCC(); // dynamicznie tworzymy obiekt na stercie
         // „pc” to zmienna wskazująca na obiekt
  pc->setInt(123);
  cout<< pc->getInt() <<endl;
  delete pc; // obiekt jest niszczony (na żadanie)
         // jest wywoływany destruktor
}
```

Uwaga: jeśli nie zniszczymy obiektu utworzonego przez operator new to nastąpi tzw „wyciek pamięci”

Klasy C++ - konstruktor z parametrami; tablice obiektów

```
class CCC {
  public:
    CCC() { i=0; } // konstruktor bezparametrowy
    CCC(int _i) { i=_i; } // też konstruktor, ale z parmetrem
    void setInt(int _i) { i=_i; }
    int getInt() { return i; }
  private:
    int i;
};
// sposób użycia:
CCC c1, c1a(); CCC c2(123); CCC c3(321), c4(1000);
cout<< c2.getInt() <<endl;
CCC *w1; w1=new CCC(555); cout<< w1->get <<endl; delete w1;

// tablice obiektów, sposób użycia:
CCC tab[10]; // uruchamia się konstruktor bezparametrowy!!!
cout<< tab[5].getInt() <<endl;
CCC *wtab; wtab= new CCC[100]; // też konstruktor bezparametrowy
cout<< wtab[5].getInt() <<endl;
delete [] wtab; // [] są KONIECZNE!!!
  // bez [] usuniemy tylko pierwszy element tablicy!
  // niektóre kompilatory wymagają podania wymiarów tablicy (bc2.0)
```

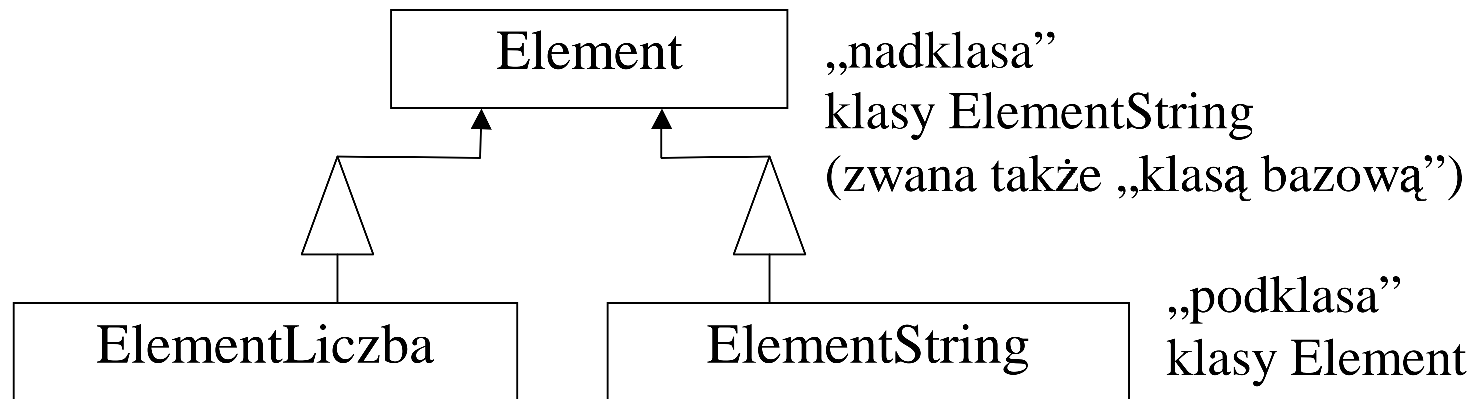
OOP: Dziedziczenie

⇒ Definiujemy klasę na podstawie innej klasy ...

⇒ Taki związek miedzy klasami zwany „generalizacją” lub „gen-spec”

⇒ Terminologia:

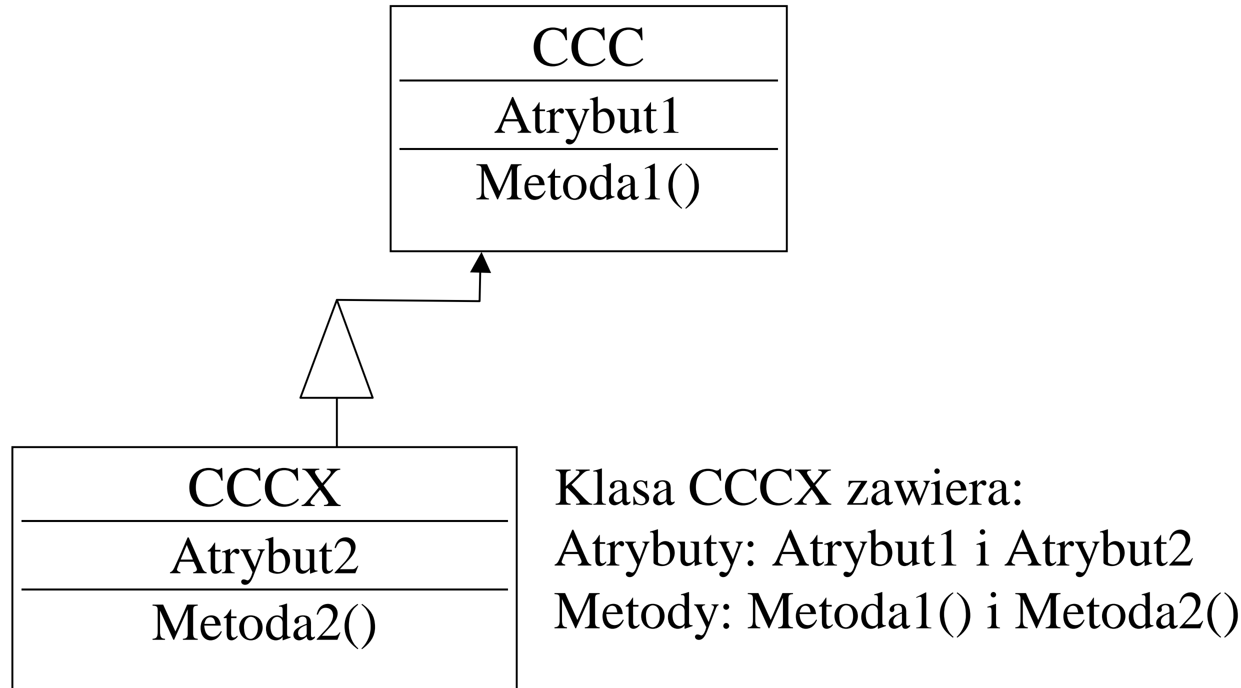
- Klasa z której dziedziczymy to „nadklasa”, klasa dziedzicząca to „podklasa”
- „Dziedzicznie wielokrotne” to dziedziczenie równocześnie z wielu klas; umożliwia je C++; inne nazwy: „dziedziczenie wielobazowe”, „wielorakie”



⇒ Element *W1; ElementLiczba *W2;
W1=W2; // ok. - JEST zgodność w sensie przypisania
// (następuje niejawna, automatyczna konwersja do nadklasy)
W2=W1; // błąd !!!

OOP: Dziedziczenie

⇒ Podczas dziedziczenia możemy podać nowe atrybuty i metody:



⇒ korzyści z dziedziczenia:

- o jeśli dwie klasy mają wspólną część to dajemy temu <u>jawnie</u> wyraz
- o ułatwienie w definiowaniu klas - nie definiujemy klasy „od zera”
- o automatyczna konwersja do nadklasy jest przydatna

OOP: Dziedziczenie w C++

```
class CCC {
public:
  CCC(int i) { pole=i; }
  int pole;
  void metoda();
};
class CCCX: public CCC {
    // klasa CCCX dziedziczy z CCC …
      // przy publicznym dziedziczeniu widoczność składowych się nie zmienia
public:
  CCCX(int i1, int i): CCC(i) { pole1=i1; }
    // pokazujemy jak wywołać konstruktor nadklasy!!!
  int pole1;
  void metoda1();
};

CCCX cx(10,20);
cout<<cx.pole<<” ”<<cx.pole1<<endl; // zwroci 20 10
cx.pole=123; cx.pole1=321; cx.metoda(); cx.metoda1();
```

OOP: Polimorfizm

⇒ Polimorfizm = „coś wygląda tak samo, ale zachowuje się inaczej”

⇒ ... problem „listy elementów”: chcemy aby elementy listy mogły być dowolnego typu

```
//void Lista::dolacz(int elem);
void Lista::dołącz(Element *elem);
Lista L;
L.dolacz(new ElementString(„A ku ku !!!”));
  // następuje niejawna konwesja do (Element*)
Element *e= L.odczytajElement();
  // jak powrócić do ElementString ???
    // jawna konwersja ???
      // ale co jeśli to NIE jest ElementString ???
```

Uwaga: konwersja do nadklasy jest naturalna, natomiast konwersja do podklasy wymaga jawnego rzutowania i JEST NIEBEZPIECZNA !!!

Uwaga 2: C++ posiada narzędzia do w miarę bezpiecznej konwersji do podklasy (są to nowe elementy C++: RTTI, operatory rzutowania jak static_cast<>)

OOP: Polimorfizm

Inne podejście do problemu „listy elementów”:

⇒ Element to ADT !!! - interesują nas wyłącznie *operacje* jakie można wykonywać na elementach listy !!!

⇒ użyjemy „metod polimorficznych” (w języku C++ zwanych „wirtualnymi funkcjami składowymi”) ...

```
class Element {
  public: virtual void pisz() { cout<<"???"<<endl; }
  // ... reszta def klasy
};
class ElementString: public Element {
  public: /*virtual*/ void pisz() { cout<<s<<endl; }
  private: std::string s;
};
L.dolacz(new ElementString(„A ku ku !!!"));
Element *e= L.odczytajElement();
e->pisz();
 // zadziała metoda pisz z klasy ElementString
  // (o ile ob. wskazywany przez we jest klasy ElementString)
```

OOP: Polimorfizm

⇒ Zasada działania metod polimorficznych:

  - jeśli wywołujemy metoda obiektu klasy X poprzez wskazanie na nadklasę to jest uruchamiana metoda klasy X; w przypadku metody niepolimorficznej zostanie uruchomiana metoda nadklasy

⇒ Uwaga 1: „metoda polimorficzna" - to terminologia OOP; „wirtualna funkcja składowa" - to terminologia języka C++

⇒ Uwaga 2: w OOP wszystkie metody są polimorficzne (dlatego się tego pojęcia nie używa!); w C++ niekoniecznie; w Javie przyjęto zasadę OOP (wszystkie metody są polimorficzne)