Technologie informacyjne

Tadeusz Lesiak

Wykład 12

Programowanie obiektowe
na przykładzie języka C++

Rodzaje programowania

1. Liniowe
2. Proceduralne
3. Z ukrywaniem danych
4. obiektowe

Programowanie liniowe

Poszczególne instrukcje kodu są pisane jedna za drugą

Wszystkie zmienne i dane są globalne – dostępne w czasie całego przebiegu programu

„kod a la spaghetti" – liczne instrukcje skoku „GO TO"

Dłuższy kod staje się niezrozumiały (nawet dla autora)

Przykład: **język BASIC**

Programowanie proceduralne

W programie można wyodrębnić części (procedury ,funkcje, subroutines…)

Realizują one pewne określone czynności

W ramach danej procedury można wyodrębnić dodatkowe zmienne LOKALNE – dostępne tylko dla niej

Zmiana „filozofii" programowania: pisanie kodu programu polega na zamianie problemu na serię zadań do wykonania (powtarzających się)

Przykład: język FORTRAN

Wada: ta technika zajmuje się wyłącznie procedurami, nie dbając o to czy poszczególne dane występujące w programie są ze sobą powiązane czy też „porozrzucane" w programie w luźny sposób

Np. dla FORTRANu jedynym sposobem powiązania danych jest umieszczenie ich w tablicy – muszą być one jednak wtedy wszystkie jednego typu

Programowanie z ukrywaniem danych

Wzbogacone w stosunku do strukturalnego o możliwość zapakowania różnych danych w jednym module

Taki moduł stanowi pewną całość np. rekord w PASCAL, struktura w C

Zaleta: dane, które są ze sobą powiązane w życiu, są także logicznie ze sobą związane w programie

Przykłady: PASCAL, C

Programowanie obiektowe

Wzbogacenie w stosunku do programowania z ukrywaniem danych:
zgrupowanym danym dodaje się metody postępowania z nimi (funkcje)

Obiektowe („object oriented", OO) – styl programowania, bazujący na modelowaniu rzeczywistych obiektów otaczającego nas świata.

Modelowanie takie uwzględnia przy tym charakterystyki oraz zachowanie pojedynczych obiektów oraz powiązania między nimi

Następuje przesunięcie akcentów z algorytmów na obiekty i oddziaływania między nimi

Istotą OO jest zaprojektowanie właściwych struktur danych oraz odpowiednich dla tych struktur operacji (funkcji, metod)

Przykład: obiekt = toster;

jego cechy (charakterystyki) → liczba ładowanych jednorazowo grzanek, wymagane napięcie sieci, zużycie prądu, czas opiekania …

jego zachowanie → procedury ładowania grzanek, ich opiekania, wyjmowania gotowych

Wiele z tych cech oraz zachowań jest wspólne dla bardziej ogólnego obiektu np. „urządzenie kuchenne"

Programowanie obiektowe a elektronika

Analogia w ewolucji elektroniki i programowania:

Od lamp i tranzystorów do obwodów scalonych

Nowa filozofia projektowania i budowy złożonych układów elektronicznych z gotowych standardowych kostek (wymagało to zmian sposobu myślenia ale też i nowych metod projektowania i konstrukcji elektroniki

Układy scalone są jednokrotnego użycia

Od kodu liniowego i procedur do obiektów

Nowa filozofia: odtąd systemy, moduły i programy będą konstruowane z gotowych, często wielofunkcyjnych i wysoce efektywnych „kostek programowych"

Kostki programowe (tzw. biblioteki klas) są wielokrotnego użycia – można je składać i łączyć w różne zestawy, tworząc systemy o zadanych własnościach

Programowanie obiektowe a proceduralne

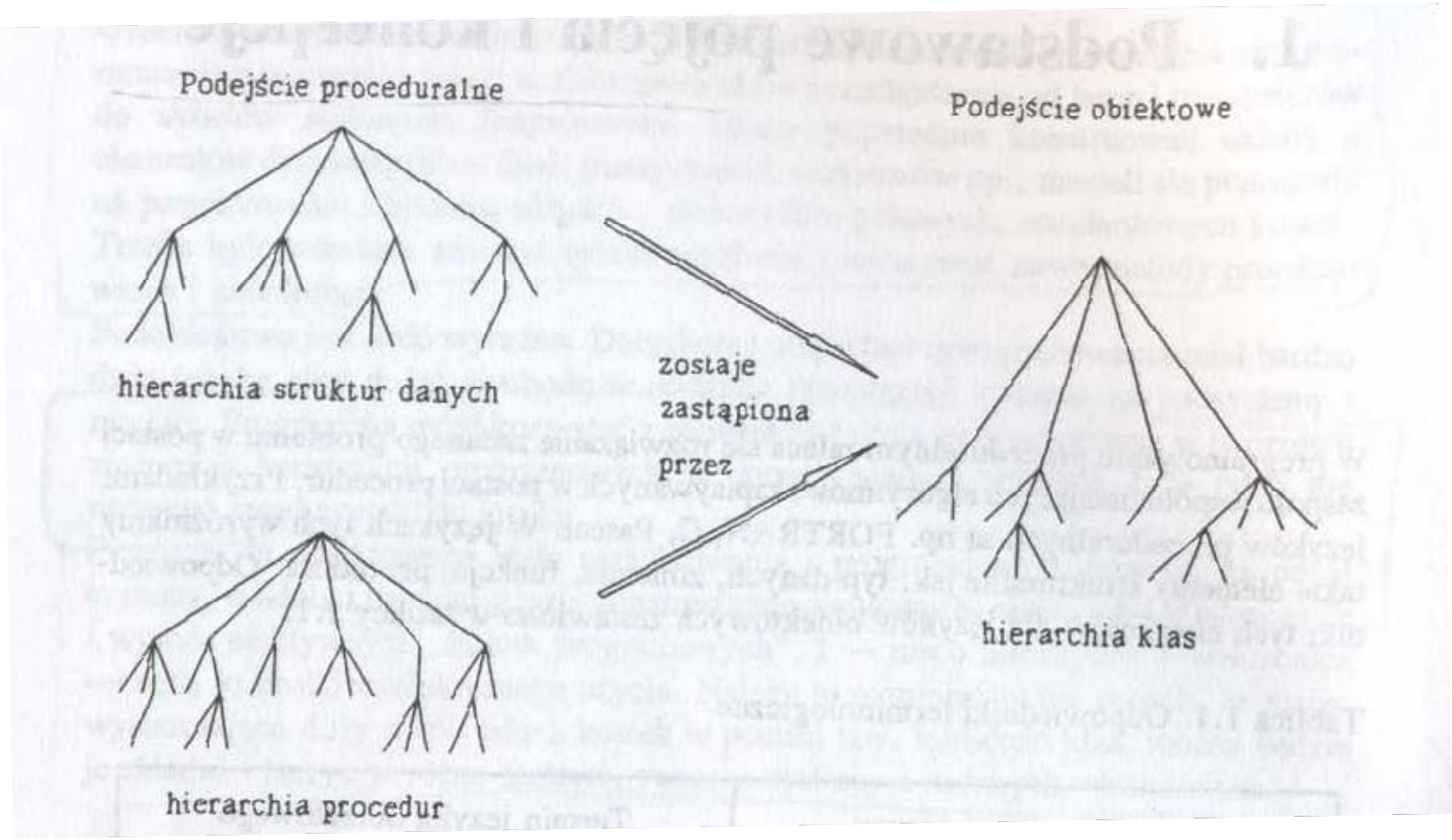
Można napisać kod proceduralny w języku obiektowym, ale nie odwrotnie
(W dodatku, kod proceduralny w języku obiektowym będzie mało czytelny i efektywny)

Odpowiedniki terminologiczne:

Termin programistyczny	Termin języka obiektowego
zmienna	obiekt
wartość zmiennej	stan
typ	klasa
funkcja/procedura	metoda
wywołanie funkcji/procedury	komunikat
hierarchia typów	hierarchia klas

Programowanie obiektowe a proceduralne

Przejście: programowanie proceduralne → programowanie obiektowe



Języki programowania też ewoluują ku obiektowości:

C ➔ C++

PASCAL ➔ OBJECT PASCAL (podstawa środowiska DELPHI)

BASIC ➔ VISUAL BASIC (Microsoft; w pełni obiektowe dopiero .NET)

Obiekty

Obiekt to „cokolwiek"

Wszystko można opisać jako obiekty

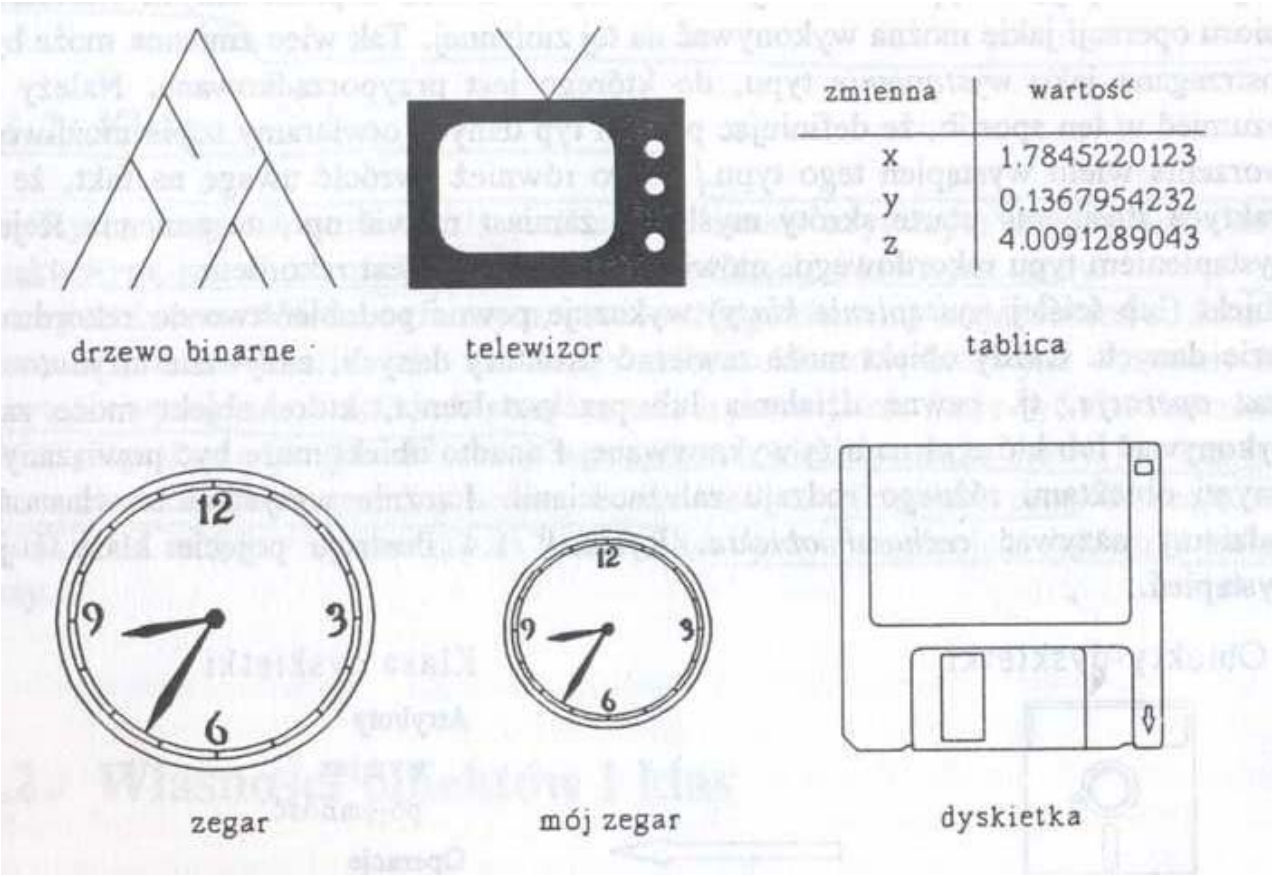
➔ Dowolny program komputerowy (algorytm) można skonstruować z obiektów)

Obiekty

Obiekt w rozumieniu potocznym – wyodrębniony element otaczającego nas świata lub pewne pojęcie abstrakcyjne

Obiektem może być praktycznie wszystko - cokolwiek czemu można przypisać nazwę np. osoba, rzecz, idea

Przykłady obiektów (rzeczywistych i abstrakcyjnych):



Obiekty

Obiekt może oddziaływać na inne obiekty oraz podlegać oddziaływaniu innych obiektów

System = zbiór obiektów podlegających powyższym interakcjom

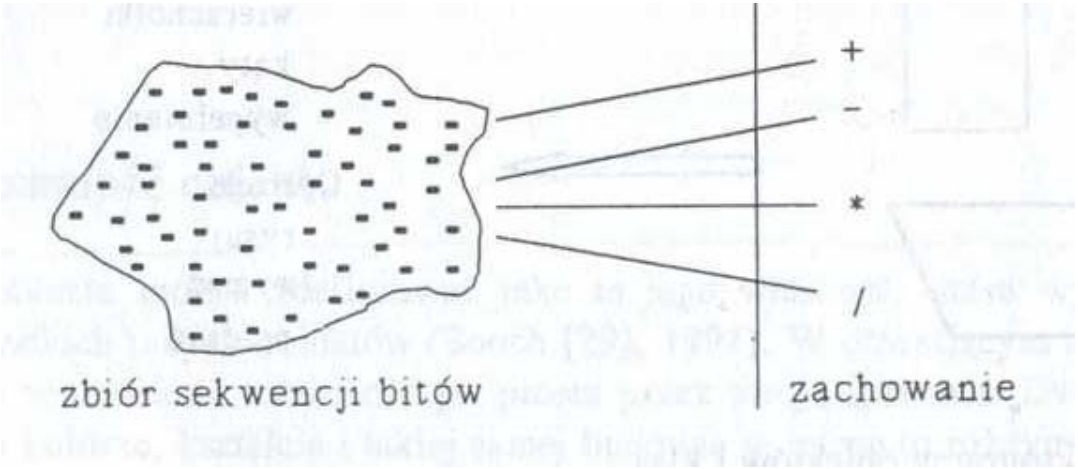
Modelowanie dowolnego systemu na komputerze = zbudowanie i opisanie modelu formalnego tego systemu (w sposób abstrakcyjny)

Obiekt w rozumieniu języka programowania – pewien identyfikowalny fragment oprogramowania - **reprezentacja rzeczywistego pojęcia w programie**

Koncepcja obiektu wiąże się ściśle ze stosowaną w informatyce **abstrakcją danych**
W językach programowania przyjmuje ona postać **abstrakcyjnych typów danych**

Np. **typ Integer**, obejmujący skończony zakres liczb całkowitych

Wystąpienie („instance") typu Integer ma wewnętrzną interpretację w postaci sekwencji bitów



Obiekty

Obiekt składa się z opisujących go danych oraz może on wykonywać ustalone czynności

1. **Obiekty zawierają pola czyli zmienne**. Mają one na celu przechowywanie informacji o obiekcie – jego charakterystyki

2. Obiekty mogą wykonywać pewne działanie tj. uruchamiać zaprogramowane funkcje. Te funkcje nazywamy **metodami** lub **funkcjami składowymi**

Metody czynią obiekt tworem aktywnym – nie jest on jedynie pojemnikiem na dane, lecz może on samodzielnie nimi manipulować

Przykład: obiekt-samochód

Kolor		
Masa	726 kg	
Pozycja	X: 126 m	Y: -60 m
Prędkość	X: 30 m/s	Y: 40 m/s

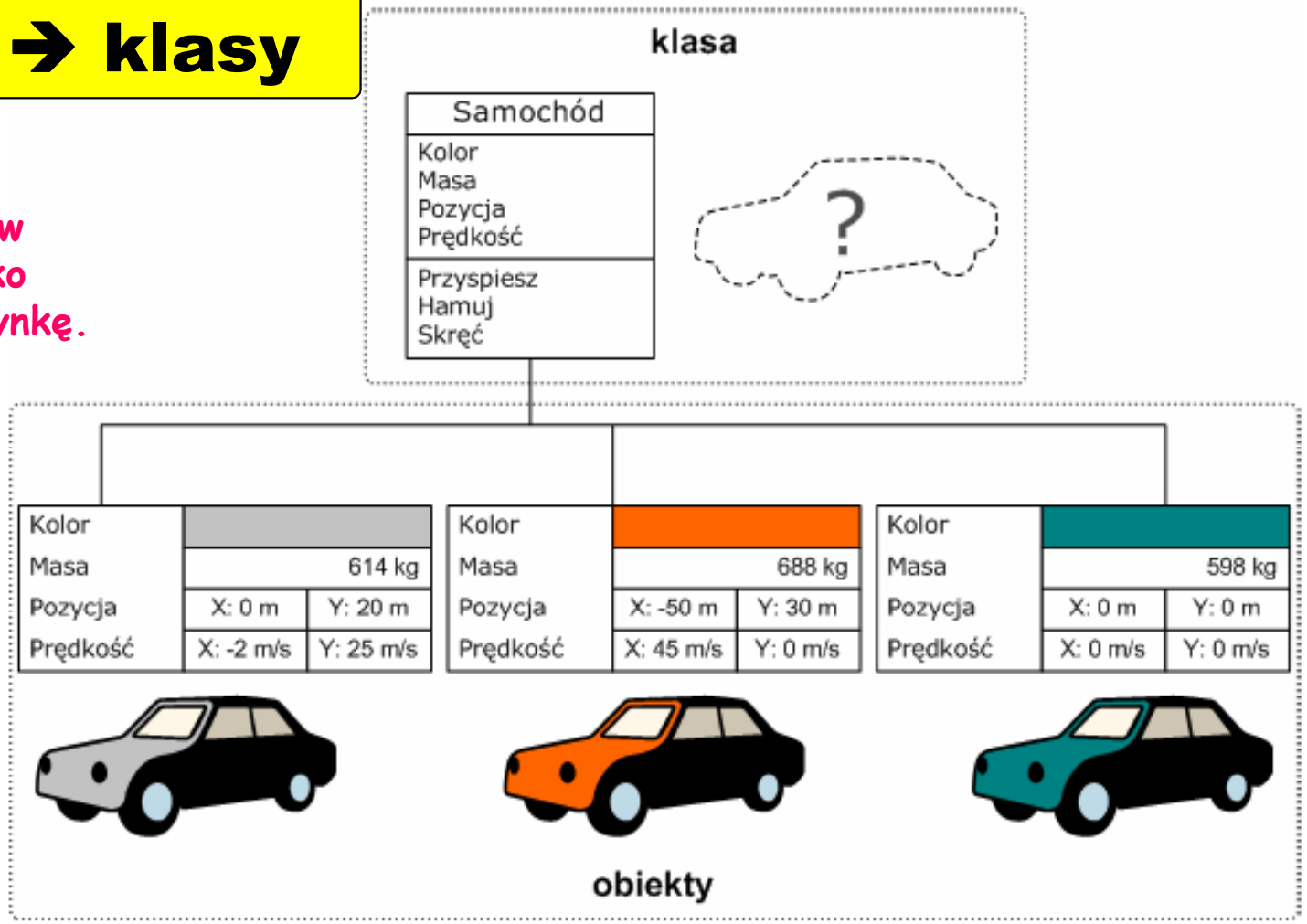
pola

Przyspiesz
Hamuj
Skręć

metody

Obiekty → klasy

Obiekt, jako zestaw pól i metod, rzadko występuje w pojedynkę.



Najczęściej występuje wiele podobnych do siebie obiektów, każdy z właściwymi dla siebie, konkretnymi wartościami pól oraz z tym samym zestawem metod.

Takie obiekty można połączyć w jedną **klasę**

Obiekty ➔ klasy

Każdy obiekt należy do pewnej klasy.

Definicja klasy zawiera pola, z których składa się dany obiekt oraz metody, które są potrzebne do jego opisu

Klasy wprowadzają systematykę (porządek) w świat obiektów

Klasa to coś w rodzaju wzorca (matrycy).

Według niej obiekty „są produkowane", inaczej „tworzone są instancje klasy"

Dla obiektów danej klasy:

Metody są takie same

Wartości poszczególnych pól mogą się różnić i tak się na ogół dzieje

Klasa „car" w c++

```
class CCar
{
    private:
            float m_fMasa;
            COLOR m_Kolor;

            VECTOR2 m_vPozycja;
    public:
            VECTOR2 vPredkosc;

            //--------------------------

            // metody
            void Przyspiesz(float fIle);
            void Hamuj(float fIle);
            void Skrec(float fKat);
};
```

definiowanie klas

nazwa

Deklaracja pól (zmiennych)

Klasa „car" w c++

```
class CCar
{
    private:
            float m_fMasa;
            COLOR m_Kolor;

            VECTOR2 m_vPozycja;
    public:
            VECTOR2 vPredkosc;

            //-------------------------

            // metody
            void Przyspiesz(float fIle);
            void Hamuj(float fIle);
            void Skrec(float fKat);
};
```

Etykieta private

– deklarowane za nią dane i metody są dostępne tylko z wnętrza klasy; Tylko funkcje będące składnikami klasy mogą te prywatne dane odczytywać lub zapisywać
Metody prywatne są dostępne (mogą zostać wywołane) tylko przez inne metody tej klasy

Etykieta public

– deklarowane za nią dane i metody mogą być używane z wnętrza klasy, a także poza jej zakresem

Deklaracja metod klasy

(na ogół podaje się tylko prototypy metod, a sam kod czyli implementacja danej metody, znajduje się gdzie indziej)

Definiowanie klas – implementacja metod

Dotąd podano jedynie definicję klasy (w niej tylko prototypy metod)

Taką definicję wygodnie jest wpisać do **pliku nagłówkowego klasa.h** (np. CCar.h)

Implementacja metod = wpisanie kodu metod do programu

Dla modułu, w którym będzie pisany kod trzeba udostępnić definicję klasy, poprzez komendę:

```
#include "klasa.h"
```

Przy wpisywaniu kodu nazwę funkcji należy poprzedzić nazwą klasy:

```
void CCar::Przyspiesz(float fIle)
{
    // tutaj kod metody
}
```

Klasy: tworzenie obiektów

Po zdefiniowaniu i zaimplementowaniu klasy można już tworzyć jej obiekty.

Można to robić na kilka sposobów.

Najprostszy z nich jest taki sam jak deklarowanie struktury np. w C:

```
CCar Samochod;
```

Ta linia stanowi deklarację nowej zmiennej Samochód typu CCar oraz stworzenie obiektu należącego do tej klasy

Po stworzeniu obiektu można

operować na wartościach jego pól

oraz

wywoływać przynależne do klasy obiektu metody

Operator wywołania = (.) - kropka

```
// przypisanie wartości polu
Samochod.vPredkosc.x = 100.0;
Samochod.vPredkosc.y = 50.0;

// wywołanie metody obiektu
Samochod.Przyspiesz (10.0);
```

Dalsze przykłady klas (C++)

```
class PaczkaZapalek {
public:
   int ile_zapalek;        //Własciwość - ilość zapałek w pudełku
   void Dodaj(int ile);    //Metoda dodaje zapałki do pudełka
   void Wyjmij(int ile);   //Metoda wyjmuje zapałki z pudełka
};
```

```
class TPunkt
{
   private:
    int x;
    int y;
   public:
    int getx(){return x;}
    int gety(){return x;}
    void putx(int xx){x=xx;}
    void puty(int yy){y=yy;}
};
```

```
#include <iostream>
using namespace std ;

class samochod
{
private:
 int ilosc_srub ;
public:
 double pojemnosc ;
 long ilosc_siedzen ;
} ;

int main()
{
 samochod nowy ; // tworzymy konkretny obiekt

 // teraz będziemy się odnosić do składników

 nowy.ilosc_srub = 5 ; // BŁĄD. Składnik ilość_srub jest przecież
                       // prywatny. Nie możemy się do niego odnieść
                       // bo znajdujemy się poza zakresem klasy.
 nowy.pojemnosc = 1.6 ; // OK. Wszystko w porządku. Składnik ten
                        // jest publiczny.
 nowy.ilosc_siedzen = 5 ; // OK. Wytłumaczenie jak wyżej.

 cout << „Ilosc siedzen wynosi : „ << nowy.ilosc_siedzen << endl ;

 system(„pause") ;
 return 0 ;
}
```

Klasy: konstruktory

Konstruktor – specyficzna metoda klasy, wywoływana zawsze podczas tworzenia należącego doń obiektu

Typowe zadania konstruktora: inicjalizacja - nadanie wartości początkowej pól, przydzielenie pamięci wykorzystywanej przez obiekt, uzyskanie jakichś kluczowych danych z zewnątrz

Deklaracja konstruktora: jego nazwa odpowiada nazwie zawierającej go klasy, w najbardziej podstawowej wersji nie zwraca on żadnej wartości

Konstruktor – zawsze publiczny

```
class CFoo
{
   private:
         // jakieś przykładowe pole...
         float m_fPewnePole;
   public:
         // no i przyszła pora na konstruktora ;-)
         CFoo()       { m_fPewnePole = 0.0; }
```

Klasa może posiadać wiele konstruktorów.

Działają one jak funkcje przeciążane.

```
class CSomeObject
{
   private:
         // jakiś rodzaj współrzędnych
         float m_fX, m_fY;
   public:
         // konstruktory
         CSomeObject()                          { m_fX = m_fY = 0.0;     }
         CSomeObject(float fX, float fY)        { m_fX = fX; m_fY = fY; }
};
```

Oznacza to, że decyzja, który z nich faktycznie zostanie wywołany ego nazwa odpowiada nazwie zawierającej go klasy, zależy od instrukcji tworzącej obiekt

Klasy: destruktory

Destruktor – specjalna metoda klasy, przywoływana podczas niszczenia obiektu odpowiadającej mu klasy – ma on „posprzątać po obiekcie

Deklaracja destruktora: jego nazwa, **poprzedzona tyldą (~)**, odpowiada nazwie zawierającej go klasy,

Nie ma on parametrów wejściowych ani też żadnych nie zwraca

```
class CBar
{
    public:
          // konstruktor i destruktor
          CBar()      { /* czynności startowe */ }  // konstruktor
          ~CBar()     { /* czynności kończące */ }  // destruktor
};
```

Destruktor – zawsze metoda publiczna

Problem (windows): znak tyldy trzeba wciskać dwukrotnie, a następnie usuwać jeden nadmiarowy znak

Przykład: klasa lampa

```
class CLamp
{
   private:
         COLOR m_Kolor;          // kolor lampy
         bool m_bWlaczona;       // czy lampa świeci się?
   public:
         // konstruktory
         CLamp()                 { m_Kolor = COLOR_WHITE; }
         CLamp(COLOR Kolor)      { m_Kolor = Kolor; }

         //-------------------------------------------------

         // metody
         void Wlacz()            { m_bWlaczona = true; }
         void Wylacz()           { m_bWlaczona = false; }

         //-------------------------------------------------

         // metody dostępowe do pól
         COLOR Kolor() const     { return m_Kolor; }
         bool Wlaczona() const   { return m_bWlaczona; }
};
```

Tworzenie dwóch obiektów tej klasy

```
CLamp Lampa1(COLOR_RED), Lampa2(COLOR_GREEN);
```

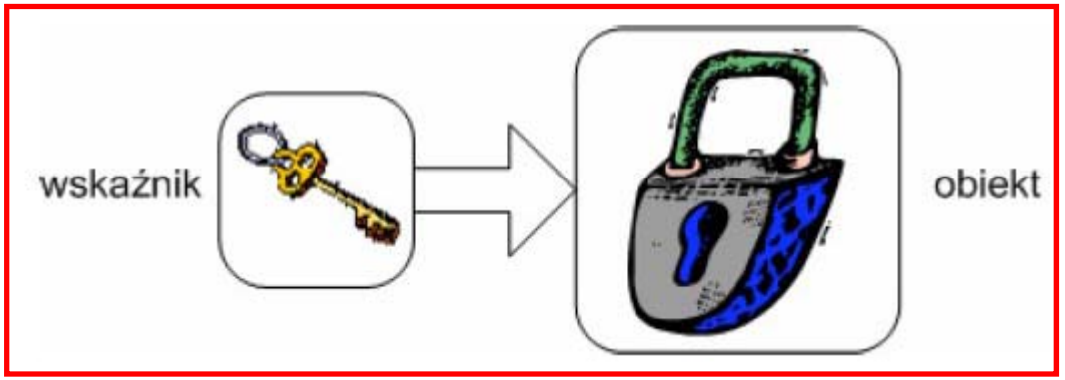
Jeden obiekt można przypisać drugiemu:

```
Lampa1 = Lampa2;
```

Obiekty zachowują się podobnie jak zmienne: czerwona lampa, pierwotnie zawarta w zmiennej Lampa1 zostaje zniszczona; w jej miejsce pojawi się kopia zawartości zmiennej Lampa2

Wskaźniki

Wskaźnik na obiekt (do obiektu) – swego rodzaju klucz do obiektu



„Dziwna" deklaracja obiektu np.

```
CFoo* pFoo;
```

Znak gwiazdki (*) przy nazwie klasy sprawia, że pFoo nie jest zmienną obiektową, lecz wskaźnikiem na obiekt klasy CFoo

pFoo nie jest obiektem, lecz „jedynie" odwołaniem do obiektu

pFoo – nie przechowuje żadnych danych należących do obiektu, lecz przygotowuje dla niego miejsce w programie

Operator new:

Tak „przygotowany" obiekt tworzy się następującą komendą:

```
pFoo = new CFoo;
```

Tworzony obiekt jest umieszczany w dowolnym miejscu pamięci (w nie w którejś ze zmiennych i na pewno nie w pFoo). Za pośrednictwem wskaźnika istnieje jednak pełna swoboda dostępu do stworzonego obiektu – odwoływanie się do jego pól oraz metod ➔

Wskaźniki

Potęga wskaźników staje się wyraźna przy manipulowaniu kilkoma obiektami
np. wskaźnikowa deklaracja obiektu Lampa1:

```
CLamp* pLampa1 = new CLamp;
```

(powołanie do życia obiektu umieszczonego „gdzieś w pamięci"
ze wskaźnikiem pLampa1 jako odwołaniem do niego)

Wprowadzamy drugi wskaźnik i przypisujemy do niego pierwszy:

```
CLamp* pLampa2 = pLampa1;
```

Nie oznacza to, że powstaje para identycznych obiektów.

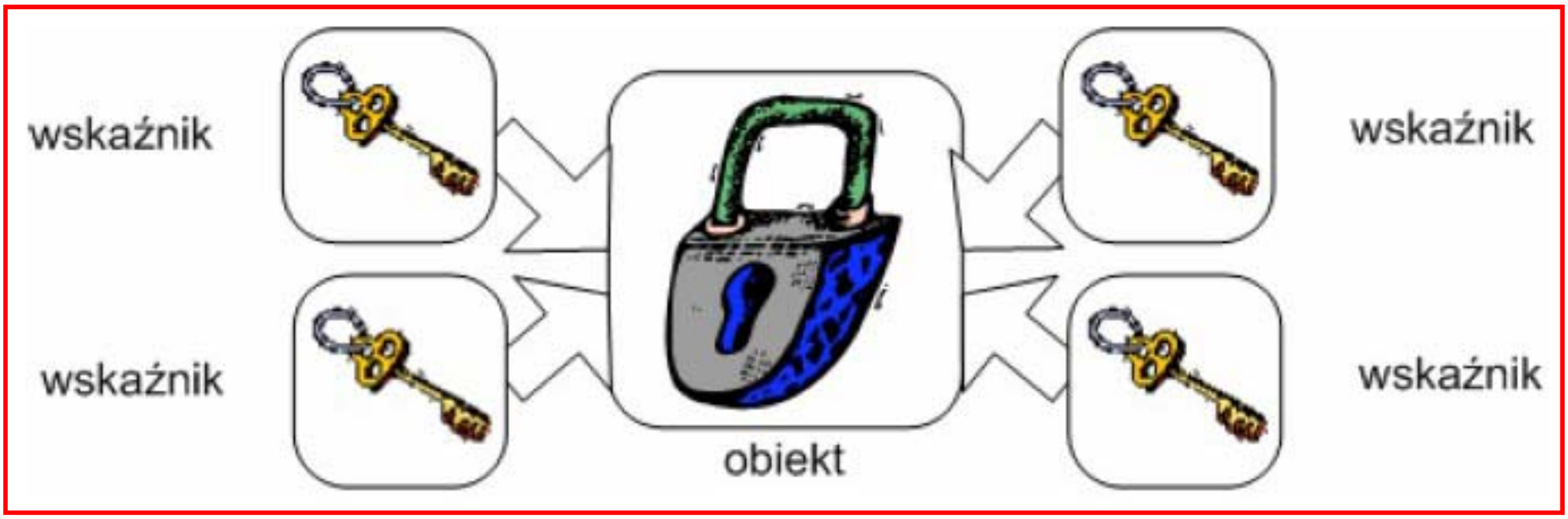
Obiekt Lampa1 nadal egzystuje w pojedynczej postaci

Użycie zarówno wskaźnika Lampa1 jak i Lampa2 jest równoważne z uzyskaniem dostępu **do jednego i tego samego obiektu**

Wskaźniki

➔Istnieją teraz dwa takie same wskaźniki;

➔ Oba pokazują jednak na ten sam obiekt



Zmienne obiektowe - każda z nich reprezentuje i przechowuje jeden obiekt; instrukcje przypisania powodują wykonanie kopii owych obiektów

Wskaźniki – istnieje wiele dróg dostępu („adresów") tego samego obiektu.

Dostęp do obiektów za pośrednictwem wskaźników

Dostęp za pomocą operatora wyłuskania – strzałka (->)

Przykład dostępu do obiektu Lampa1 za pomocą jednego z dwóch jego wskaźników, oraz wykonanie na nim metody Wlacz

```
pLampa1->Wlacz();
```

Sprawdzenie:

metoda

```
pLampa2->Wlaczona();
```

zwraca wartość true

Operator kropki (.) pozwala na uzyskanie **dostępu do składników obiektu zawartego w zmiennej obiektowej**

Operator wyłuskania – strzałka (->) pozwala na uzyskanie **dostępu do wskaźnika na obiekt**

Niszczenie obiektów

Niszczenie obiektu – operator delete wywołujący odpowiedni destruktor oraz **zwalniający zajmowaną przez obiekt pamięć operacyjną**

```
delete pFoo;
```

pFoo – musi być wskaźnikiem **do istniejącego obiektu**

Po zniszczeniu obiektu, wskaźnik nadal wskazuje na miejsce w pamięci, w którym przedtem egzystował obiekt

Po zniszczeniu obiektu wszelkie próby odwołania się do tego obszaru skończą się błędem naruszenia zasad dostępu (*access violation*) oraz awaryjnym zakończeniem działania programu

Klasy - podsumowanie

Klasa to właściwie nowy typ danych (zdefiniowany przez użytkownika), rozszerzający jakościowo zbiór typów dostępnych w programowaniu proceduralnym (real, integer, char…)

Klasa określa typ obiektu (listę pól, metody-funkcje etc.)
Klasa zawiera zarówno obiekty sensu stricte ale też i funkcje na nich operujące

Obiekty są zorganizowane w klasy. →
Klasa jest zbiorem obiektów, które współdzielą te same cechy

Obiekty są zazwyczaj tworzone na podstawie klas:

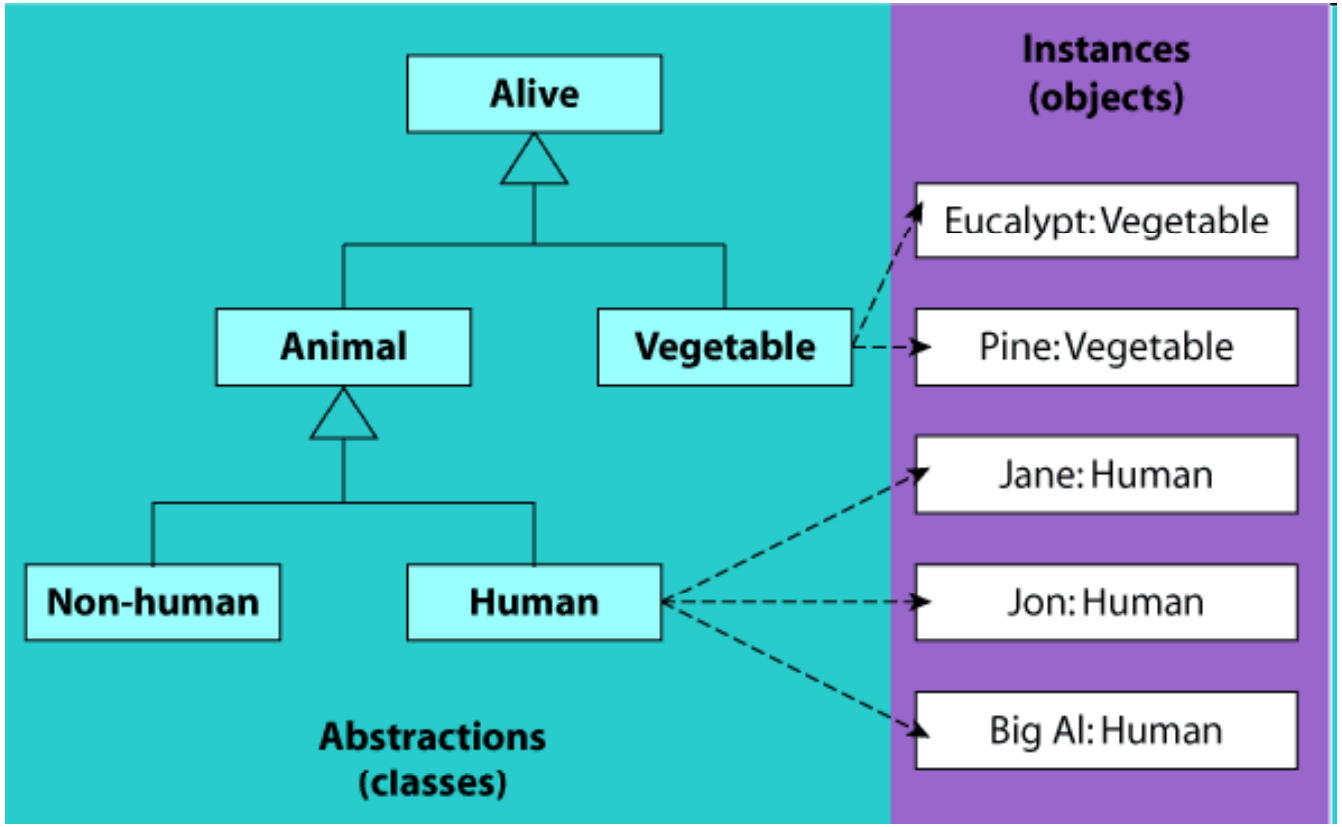
Enkapsulacja (*encapsulation*): ukrywanie pewnych danych składowych lub metod obiektów danej klasy tak, aby one same (a tym samym i możliwości ich modyfikacji) dostępne tylko metodom wewnętrznym danej klasy lub funkcjom z nią zaprzyjaźnionym.

Klasy

Klasę można określić jako abstrakcję zbioru obiektów, która stanowi wzorzec (szablon) dla tworzenia wystąpień obiektów

Dzięki temu wszystkie utworzone obiekty danej klasy mają te same cechy oraz operacje jakie można na nich wykonać

Wśród klas zaczyna rysować się pewna hierarchia ➔



Generalizacja i dziedziczenie

Generalizacja (*generalization*)– zależność między **klasą wyjściową** a **podklasą(-ami)** tj między daną klasą i jedną lub wieloma jej wyspecjalizowanymi lub udoskonalonymi wersjami

Klasa wyjściowa, inaczej superklasa, klasa bazowa (*superclass, base class*)

Atrybuty i operacje wspólne dla grupy podklas są włączane do superklasy, stając się tym samym własnościami każdej z podklas.

Mówimy, że każda z podklas **dziedziczy** *(inheritance)* własności swojej superklasy

Przykład:
Superklasa: zatrudniony
Podklasy: listonosz, rakarz, policjant, mafioso

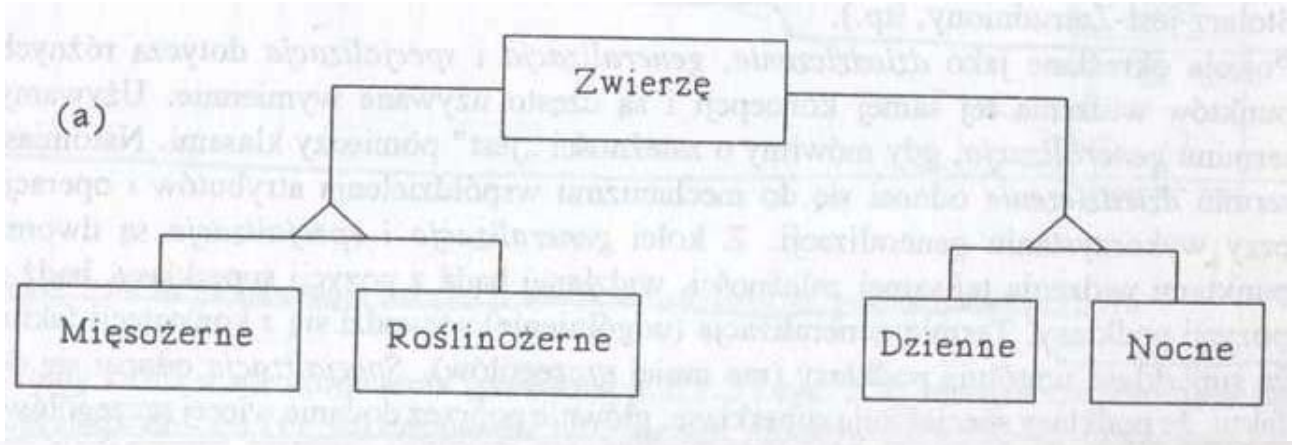
Generalizacja mówi o istnieniu relacji typu JEST między klasami np. policjant jest zatrudniony

Dziedziczenie odnosi się do mechanizmu współdzielenia atrybutów i operacji przy wykorzystaniu generalizacji

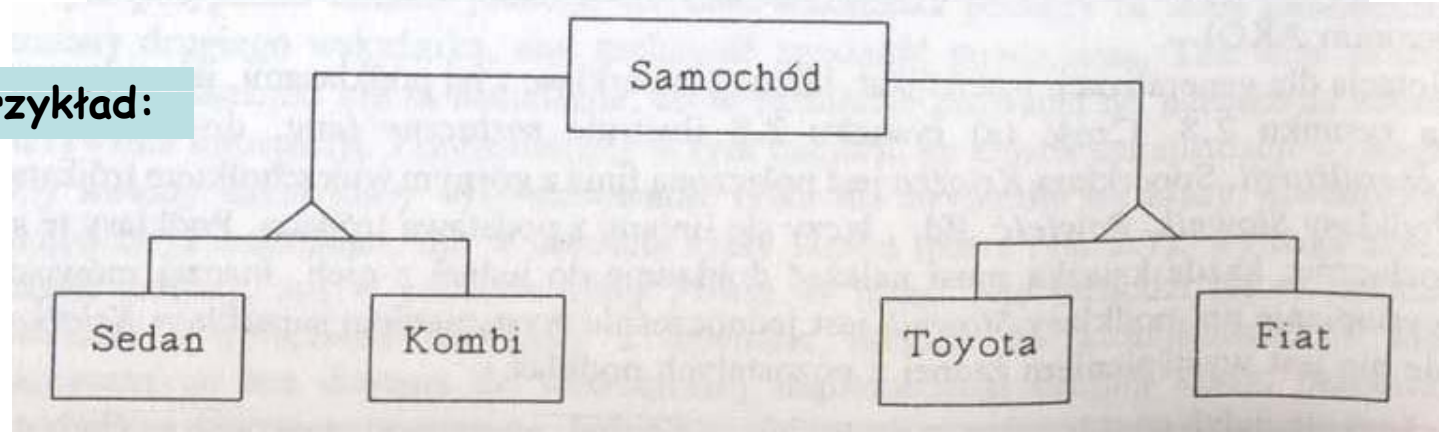
Oba pojęcia to dwa punkty widzenia tej samej zależności albo w pozycji klasy wyjściowej (generalizacja) albo podklasa (dziedziczenie)

Generalizacja

Dla klas zachodzących na siebie (overlapping) istnieją co najmniej dwa różne kryteria klasyfikacji → można dokonać co najmniej dwóch generalizacji klas rozłącznych, każda w oparciu o jedno kryterium:

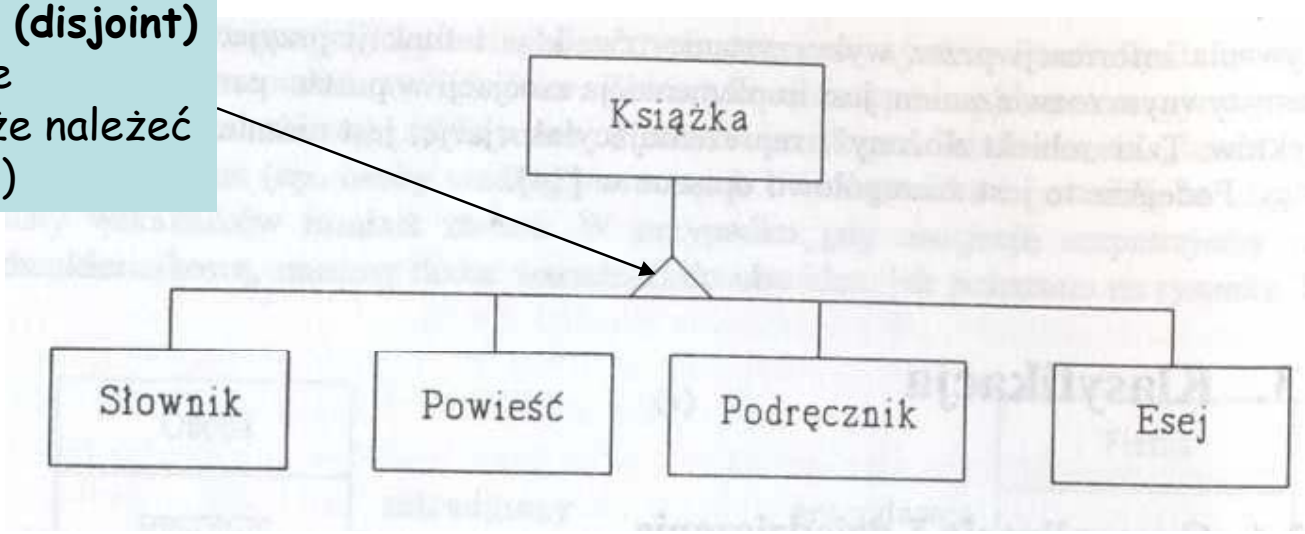


Inny przykład:

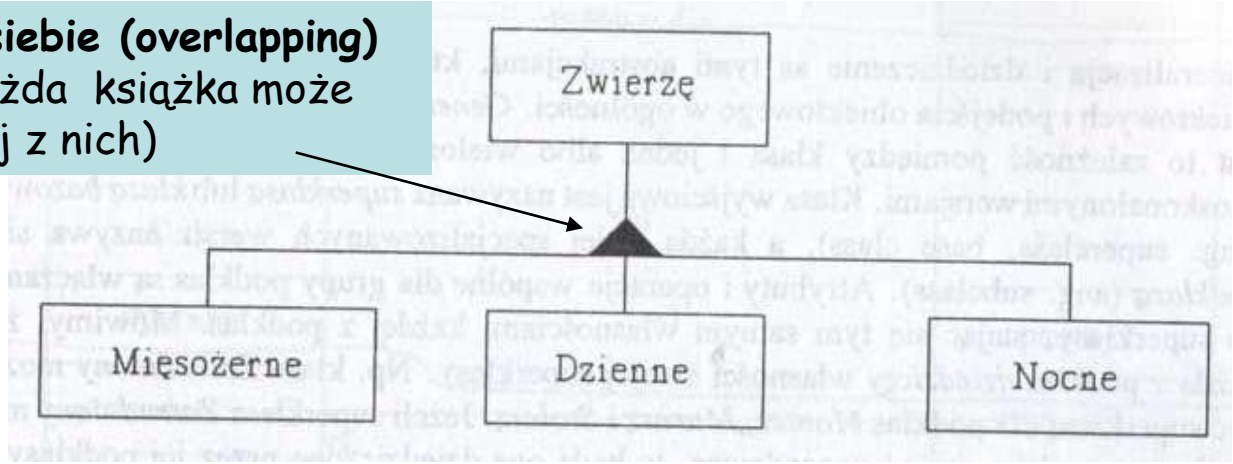


Generalizacja

Dla klas rozłącznych (disjoint)
(podklasy są rozłączne
tj. każda książka może należeć
tylko do jednej z nich)



Dla klas zachodzących na siebie (overlapping)
(podklasy są rozłączne tj. każda książka może
należeć tylko do jednej z nich)



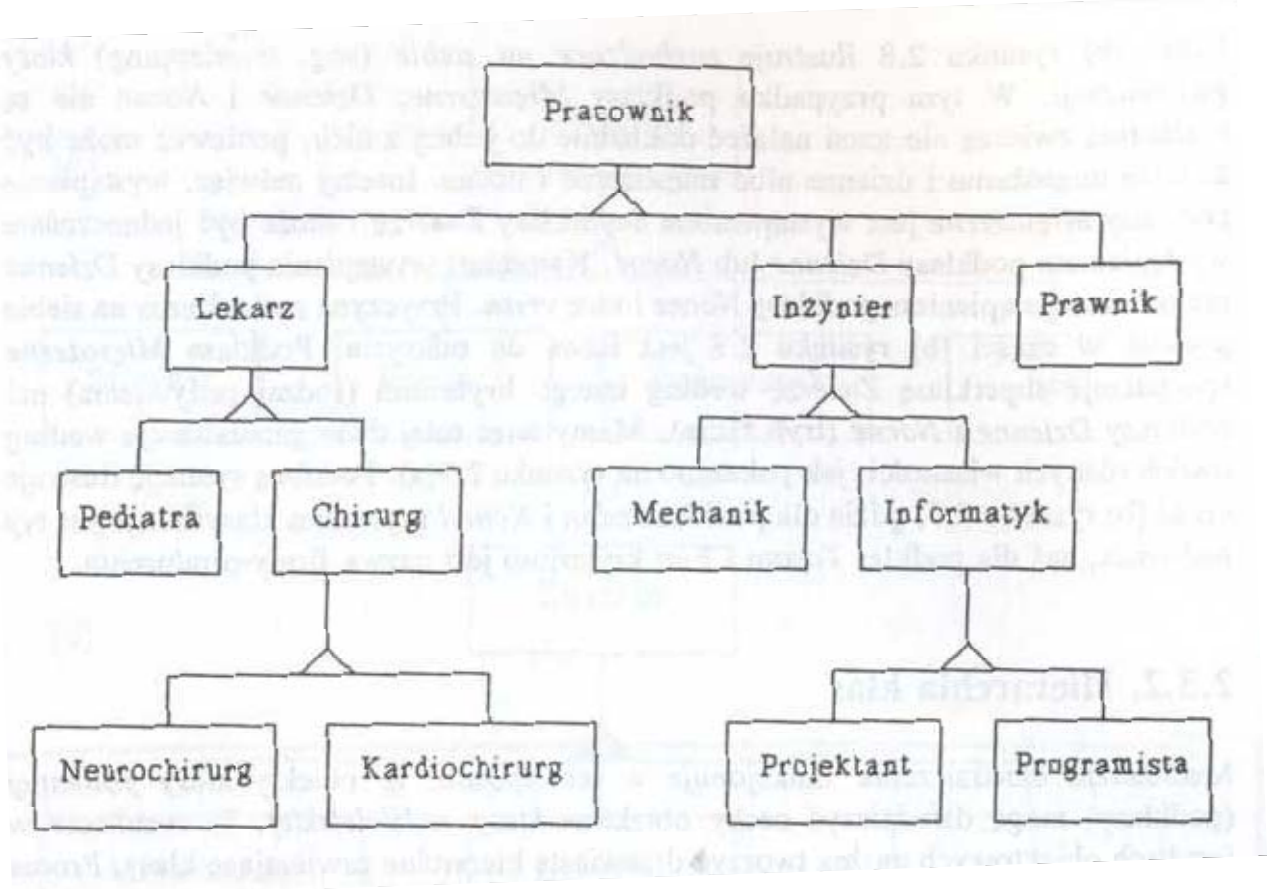
Hierarchia klas

Obiekty klasy potomnej mogą dziedziczyć cechy obiektów klasy rodzicielskiej

Generalizacja i dziedziczenie mogą być przechodnie przez dowolną liczbę poziomów
(jeżeli B jest podklasą A i C jest podklasą B → C jest podklasą A, choć nie bezpośrednią)

Przykład

Czteropoziomowa hierarchia klas

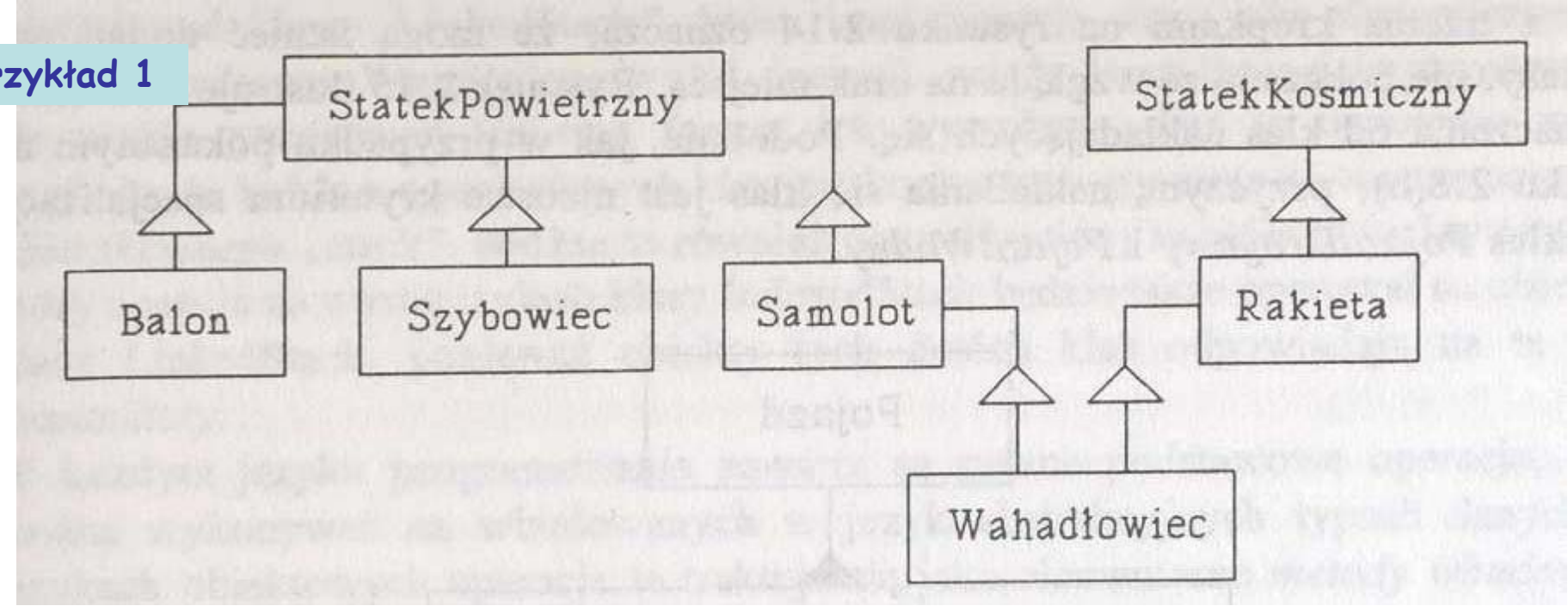


Dziedziczenie mnogie

Dziedziczenie pojedyncze (dotąd rozważane, *single inheritance*) - jedna podklasa dziedziczyła własności jednej, bezpośredniej superklasy

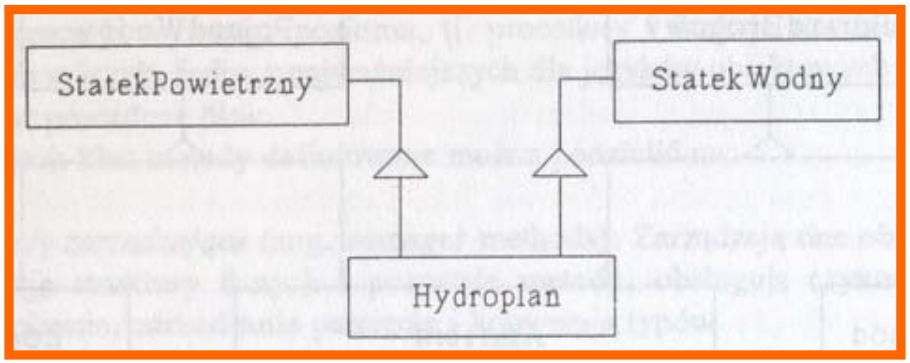
Dziedziczenie mnogie (*multiple inheritance*) – jedna podklasa dziedziczy własności kilku niezależnych klas rodzicielskich

Przykład 1

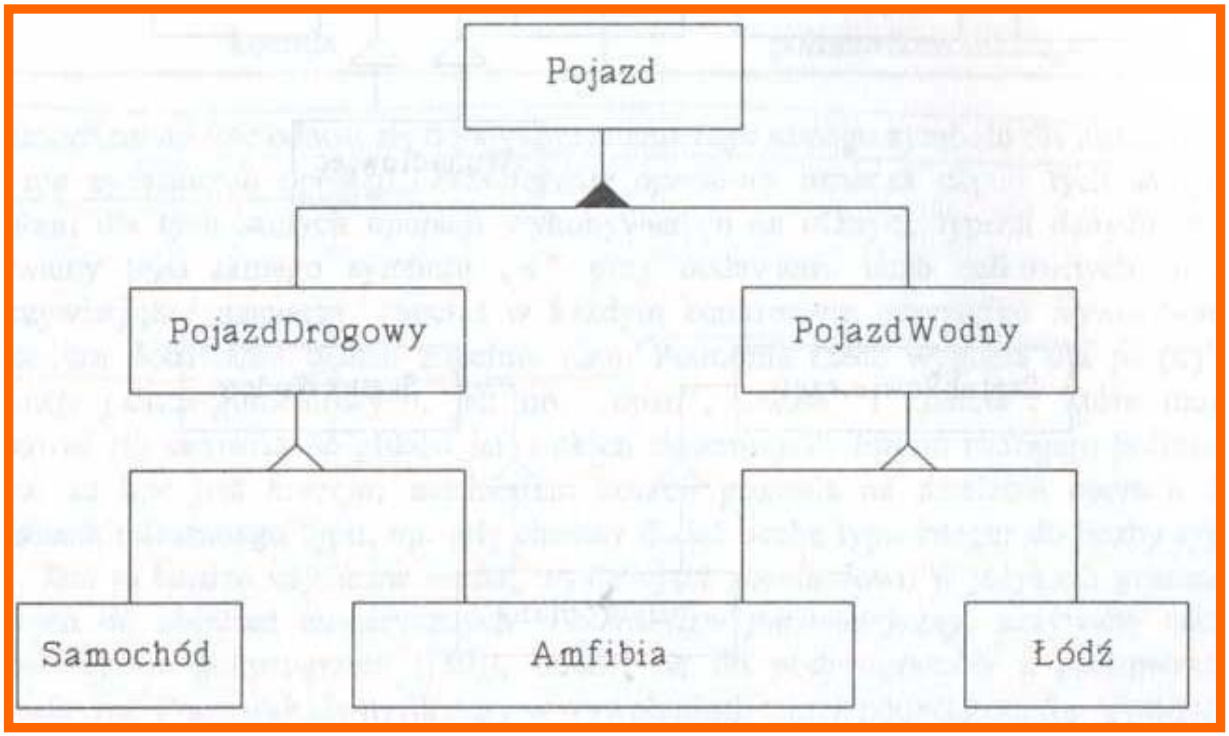


Dziedziczenie mnogie

Przykład 2



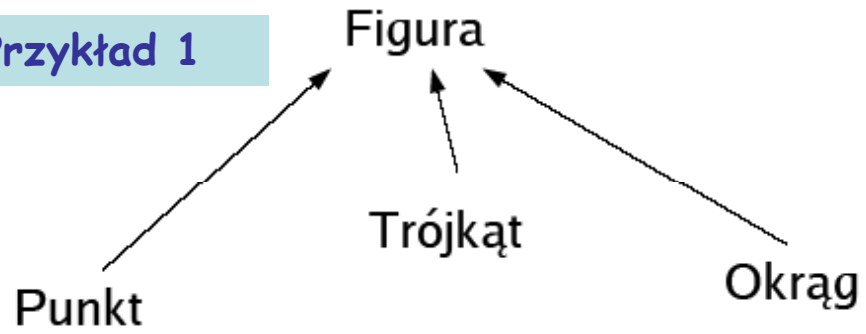
Przykład 3



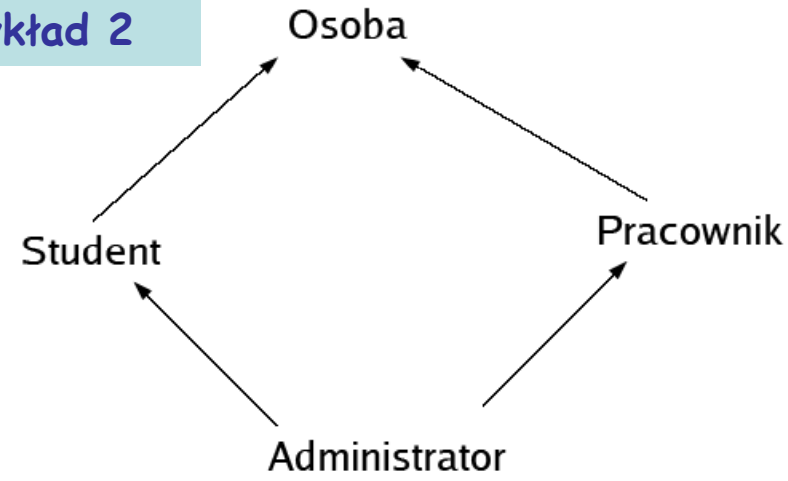
Dziedziczenie (dla programisty)

Utworzenie nowej klasy na podstawie już istniejącej, przez dodanie nowych pól i metod, lub zmianę implementacji już istniejących metod

Przykład 1



Przykład 2



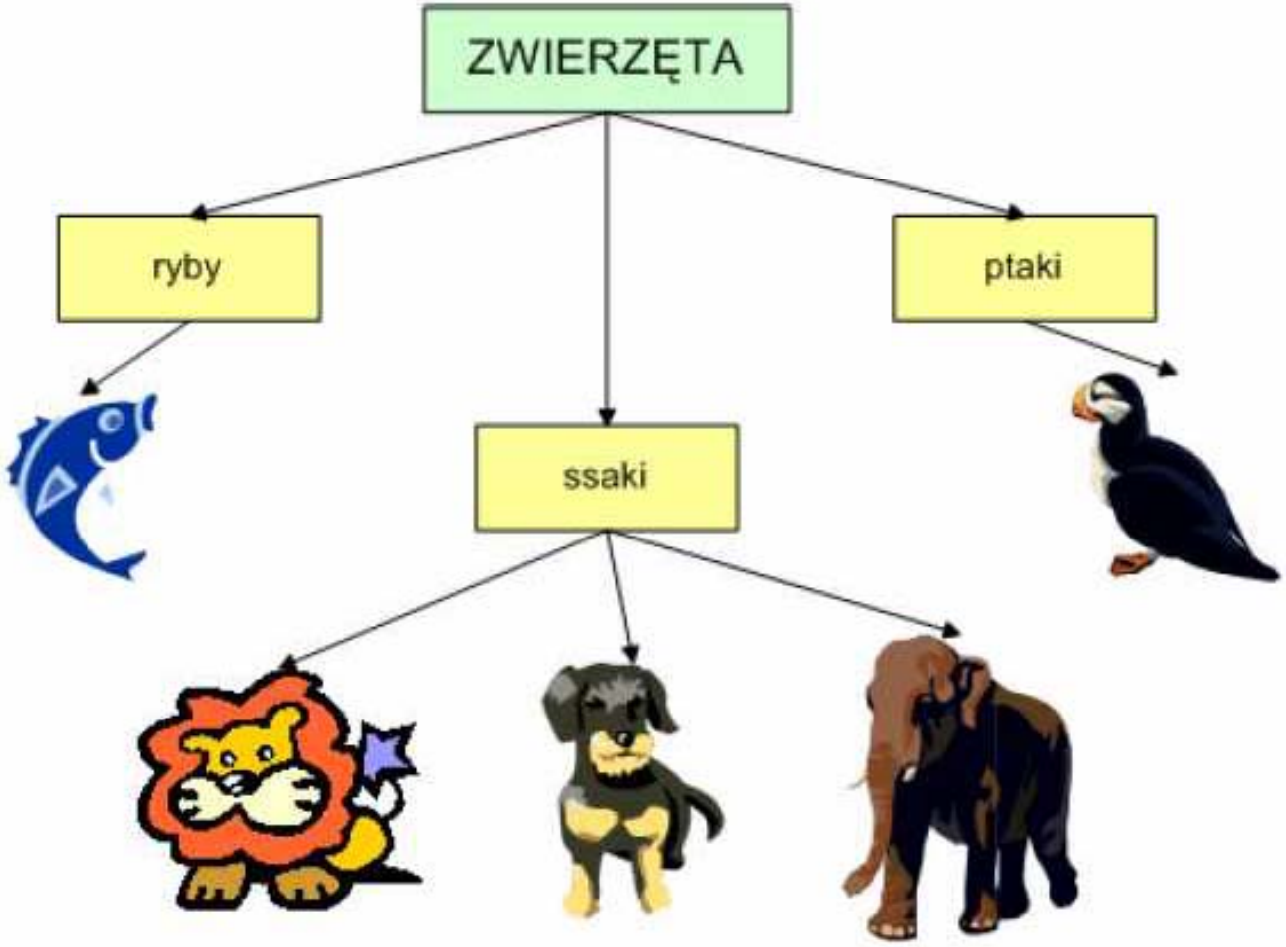

```
class Figura:

  int x;
  int y;

  przesun(int dx, int dy):
    x = x + dx
    y = y + dy
```

Dziedziczenie: podsumowanie

Dziedziczenie (inheritance) – tworzenie nowej klasy na podstawie jednej lub kilku istniejących wcześniej klas bazowych

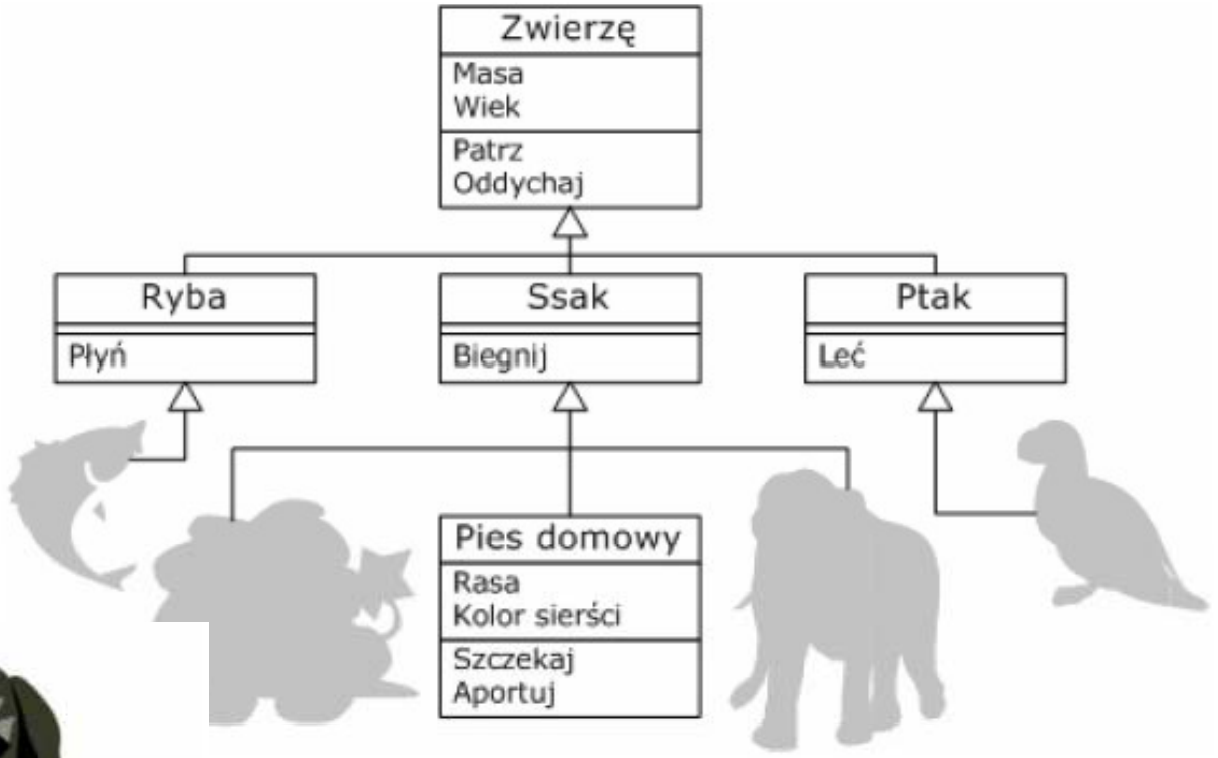


Klasa pochodna – posiada pewne elementy wspólne – **odziedziczone po klasie czy klasach bazowych** – te elementy zostały tam zdefiniowane

Zbiór elementów klasy pochodnej zostaje poszerzony o pola i metody specyficzne dla klas pochodnych

Dziedziczenie: podsumowanie

Pojęcia klasa bazowa i klasa pochodna są względne: dana klasa może pochodzić (dziedziczyć) od innych, a jednocześnie być klasą bazową dla kolejnych klas



Klasa pochodna jest na ogół bogatsza od bazowej

Dziedziczenie: implementacja w C++

W klasie bazowej pojawia się specyfikator **protected** (chronione)

➔ **Działa jak private (chroni te pola i metody przed dostępem spoza klasy), ale są one dziedziczone i występują w klasach pochodnych. Tym samym te pola i metody są dostępne w klasach pochodnych**

```
class nazwa_klasy
{
    [private:]
        [deklaracje_prywatne]
    [protected:]
        [deklaracje_chronione]
    [public:]
        [deklaracje_publiczne]
};
```

Składnia klasy pochodnej:

```
class nazwa_klasy [: [specyfikatory] [nazwa_klasy_bazowej] [, ...]]
{
    deklaracje_składowych
};
```

Dziedziczenie: implementacja w C++

Przykład:

```
class CRectangle
{
    private:
            // wymiary prostokąta
            float m_fSzerokosc, m_fWysokosc;
    protected:
            // pozycja na ekranie
            float m_fX, m_fY;
    public:
            // konstruktor
            CRectangle()        { m_fX = m_fY = 0.0;
                                  m_fSzerokosc = m_fWysokosc = 10.0; }

            //-------------------------------------------------------

            // metody
            float Pole() const    { return m_fSzerokosc * m_fWysokosc; }
            float Obwod() const   { return 2 * (m_fSzerokosc+m_fWysokosc); }
};
```

Dziedziczenie: implementacja w C++

Przykład:

```
class CSquare : public CRectangle      // dziedziczenie z CRectangle
{
   private:
         // zamiast szerokości i wysokości mamy tylko długość boku
         float m_fDlugoscBoku;

   // pola m_fX i m_fY są dziedziczone z klasy bazowej, więc nie ma
   // potrzeby ich powtórnego deklarowania

   public:
         // konstruktor
         CSquare { m_fDlugoscBoku = 10.0; }

         //-------------------------------------------------------

         // nowe metody
         float Pole() const  { return m_fDlugoscBoku * m_fDlugoscBoku; }
         float Obwod() const { return 4 * m_fDlugoscBoku; }
};
```

Hierarchia klas

Każda podklasa nie tylko dziedziczy wszystkie własności swoich superklas, lecz także dodaje specyficzne atrybuty i operacje, rozszerzając w ten sposób swoją bezpośrednią superklasę

Dowolna operacja dopuszczalna na klasie rodzicielskiej może być zastosowana na klasie potomnej

Dodatkowo w klasie potomnej można redefiniować dziedziczone metody, pozostając przy tej samej odziedziczonej nazwie

Wystąpienie podklasy jest jednocześnie wystąpieniem wszystkich jej klas rodzicielskich

Analogia z biologiczną taksonomią gatunków (biolog klasyfikuje istniejące gatunki, informatyk często projektuje nowe klasy wraz z ich hierarchią)

Metody wirtualne

Metoda wirtualna jest przygotowana na zastąpienie siebie przez nową wersję, zdefiniowaną w klasie pochodnej

Aby daną funkcję zadeklarować jako wirtualną, należy poprzedzić jej prototyp słowem kluczowym virtual

Przykład: metoda oddychaj z klasy zwierze

Każde zwierzę musi oddychać. Techniczna realizacja tej czynności jest jednak inna dla każdego rzędu, gatunku etc.

Teoretycznie klasa zwierzę mogłaby być całkowicie „nieświadoma" tego, że jedna z jej metod będzie definiowana w inny sposób w klasie pochodnej.

Wygodnie jest jednak przewidzieć zawczasu taką możliwość poprzez określenie odpowiedniej metody w klasie bazowej jako virtual

Metody wirtualne

Przykład: metoda oddychaj z klasy zwierze

```
#include <iostream>

class CAnimal
{
   // (pomijamy pozostałe składowe klasy)

   public:
          virtual void Oddychaj()
                       { std::cout << "Oddycham..." << std::endl; }
};
```

```
class CFish : public CAnimal
{
   public:
          void Oddychaj()   // redefinicja metody wirtualnej
                 { std::cout << "Oddycham skrzelami..." << std::endl; }
          void Plyn();
};

class CMammal : public CAnimal
{
   public:
          void Oddychaj()   // jak wyżej
                 { std::cout << "Oddycham płucami..." << std::endl; }
          void Biegnij();
};

class CBird : public CAnimal
{
   public:
          void Oddychaj()   // i znowu jak wyżej :)
                 { std::cout << "Oddycham płucami..." << std::endl; }
          void Lec();
};
```

Stosowanie wirtualnych destruktorów w klasie bazowej jest zalecane, wręcz konieczne

Metody czysto wirtualne

Metody czysto wirtualne (*pure virtual*)

- skrajna postać metod wirtualnych

- nie posiadają one żadnej implementacji i są przeznaczone wyłącznie do przedefiniowania ich w klasach pochodnych

```
class CAnimal
{
    // (definicja klasy jest skromna z przyczyn oszczędnościowych :))

    public:
          virtual void Oddychaj() = 0;
};
```

Klasa abstrakcyjna – zawiera przynajmniej jedną czysto wirtualną metodę i z jej powodu nie jest przeznaczona do tworzenia z niej obiektów a jedynie do wyprowadzania z niej klas pochodnych

Polimorfizm (wielopostaciowość)

Polimorfizm w programowaniu obiektowym polega na możliwości przypisania obiektom różnych form działania na etapie działania programu (a nie jego kompilacji)

➔metoda (funkcja) przyjmuje różne formy działania w zależności od tego jaki typ obiektu staje się jej argumentem

➔Przykład: przeciążanie operatora – wykorzystywanie tego samego symbolu dla tych samych operacji wykonywanych na różnych typach danych

Np. ten sam symbol „+" może być używany przy dodawaniu liczb całkowitych, liczb rzeczywistych i macierzy, chociaż w każdym tym konkretnym przypadku wywoływana procedura dodawania będzie zupełnie inna

Polimorfizm (wielopostaciowość)

Polimorfizm – wykorzystanie tego samego kodu do operowania na obiektach przynależnych różnym klasom, dziedziczącym od siebie

W C++ polimorfizm nie dotyczy każdej klasy, a jedynie określonych typów polimorficznych

Typ polimorficzny – klasa zawierająca przynajmniej jedną metodę wirtualną

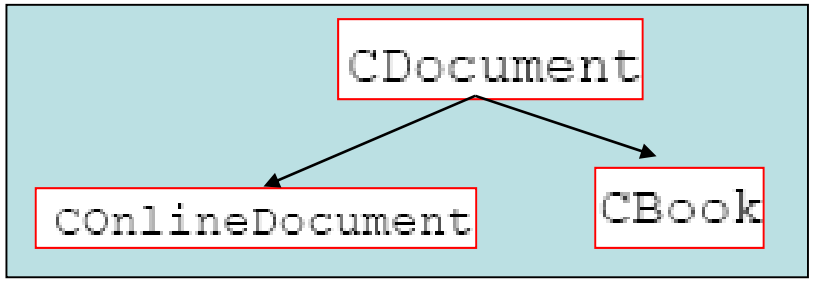
Polimorfizm daje znaczne uproszczenie większości algorytmów:
- takich w których dużą rolę odgrywa zarządzanie wieloma różnymi obiektami
- gdy te same czynności są wykonywane dla wielu obiektów różnych rodzajów

Prosty przykład polimorfizmu:

Wskaźnik na obiekt klasy bazowej może wskazywać także na obiekt którejkolwiek z jego klas pochodnych

Polimorfizm

Klasa bazowa ogólnego dokumentu:




```
#include <string>
#include <ctime>

// klasa dowolnego dokumentu
class CDocument
{
   protected:
         // podstawowe dane dokumentu
         std::string m_strAutor;       // autor dokumentu
         std::string m_strTytul;       // tytuł dokumentu
         tm          m_Data;           // data stworzenia
   public:
         // konstruktory
         CDocument()
               { m_strAutor = m_strTytul = "???";
                 time_t Czas = time(NULL); m_Data = *localtime(&Czas); }
         CDocument(std::string strTytul)
               { CDocument(); m_strTytul = strTytul; }
         CDocument(std::string strAutor, std::string strTytul)
               { CDocument();
                 m_strAutor = strAutor;
                 m_strTytul = strTytul; }

          //-------------------------------------------------------------

          // metody dostępowe do pól
          std::string Autor() const   { return m_strAutor; }
          std::string Tytul() const   { return m_strTytul; }
          tm          Data()  const   { return m_Data;     }
 };

 //-----------------------------------------------------------------------
```

Polimorfizm

pierwsza klasa pochodna: dokument internetowy

```
// dokument internetowy
class COnlineDocument : public CDocument
{
   protected:
         std::string m_strURL;   // adres internetowy dokumentu
   public:
         // konstruktory
         COnlineDocument(std::string strAutor, std::string strTytul)
               { m_strAutor = strAutor; m_strTytul = strTytul; }
         COnlineDocument  (std::string strAutor,
                           std::string strTytul,
                           std::string strURL)
               { m_strAutor = strAutor;
                 m_strTytul = strTytul;
                 m_strURL   = strURL;    }

         //-------------------------------------------------------

         // metody dostępowe do pól
         std::string URL()   const   { return m_strURL; }
};
```

Polimorfizm

druga klasa pochodna: książka

```
// książka
class CBook : public CDocument
{
   protected:
          std::string m_strISBN;  // numer ISBN książki
   public:
          // konstruktory
          CBook(std::string strAutor, std::string strTytul)
                { m_strAutor = strAutor; m_strTytul = strTytul; }
          CBook (std::string strAutor,
                 std::string strTytul,
                 std::string strISBN)
                { m_strAutor = strAutor;
                  m_strTytul = strTytul;
                  m_strISBN  = strISBN; }

          //-------------------------------------------------------

          // metody dostępowe do pól
          std::string ISBN()  const   { return m_strISBN; }
};
```

Polimorfizm

Funkcja wyświetlająca informacje o podanym dokumencie

```
#include <iostream>

void PokazDaneDokumentu(CDocument* pDokument)
{
    // wyświetlenie autora
    std::cout << "AUTOR: ";
    std::cout << pDokument->Autor() << std::endl;

    // pokazanie tytułu dokumentu
    // (sekwencja \" wstawia cudzysłów do napisu)
    std::cout << "TYTUL: ";
    std::cout << "\"" << pDokument->Tytul() << "\"" << std::endl;

    // data utworzenia dokumentu
    // (pDokument->Data() zwraca strukturę typu tm, do której pól
    //  można dostać się tak samo, jak do wszystkich innych - za
    //  pomocą operatora wyłuskania . (kropki))
    std::cout << "DATA : ";
    std::cout << pDokument->Data().tm_mday << "."
              << (pDokument->Data().tm_mon + 1) << "."
              << (pDokument->Data().tm_year + 1900) << std::endl;
}
```

Funkcja ta pobiera tylko jeden parametr – wskaźnik na obiekt typu CDocument.
Wskaźnik ten może jednak pokazywać na dowolny obiekt klasy pochodnej,
a kod będzie za każdym razem działać prawidłowo

Ten przykład to dopiero przygrywka do właściwego polimorfizmu.

Polimorfizm

Wprowadźmy teraz funkcję PokazDaneDokumentu do klasy bazowej
w postaci metody wirtualnej

```
// *** documents.h ***

class CDocument
{
   // (większość składowych wycięto z powodu zbyt dużej objętości)

   public:
         virtual void PokazDane();
};
```

Nieistotna dla nas reszta implementacji – plik documents.cpp

Polimorfizm

```
// *** main.cpp ***

#include <iostream>
#include <conio.h>
#include "documents.h"

void main()
{
    // wskaźnik na obiekty dokumentów
    CDocument* pDokument;

    // pierwszy dokument - internetowy
    std::cout << std::endl << "--- 1. pozycja ---" << std::endl;
    pDokument = new COnlineDocument("Regedit",
                                    "Cyfrowe przetwarzanie tekstu",

                                    "http://programex.risp.pl/?"
                                    "strona=cyfrowe_przetwarzanie_tekstu"
                                    );
    pDokument->PokazDane();
    delete pDokument;

    // drugi dokument - książka
    std::cout << std::endl << "--- 2. pozycja ---" << std::endl;
    pDokument = new CBook("Sam Williams",
                          "W obronie wolnosci",
                          "83-7361-247-5");
    pDokument->PokazDane();
    delete pDokument;

    getch();
}
```

Polimorfizm

output

```
   POLIMORFIZM
------------------

--- 1. pozycja ---
AUTOR: Regedit
TYTUL: "Cyfrowe przetwarzanie tekstu"
DATA : 6.3.2004
URL  : http://programex.risp.pl/?strona=cyfrowe_przetwarzanie_tekstu

--- 2. pozycja ---
AUTOR: Sam Williams
TYTUL: "W obronie wolnosci"
DATA : 6.3.2004
ISBN : 83-7361-247-5
```

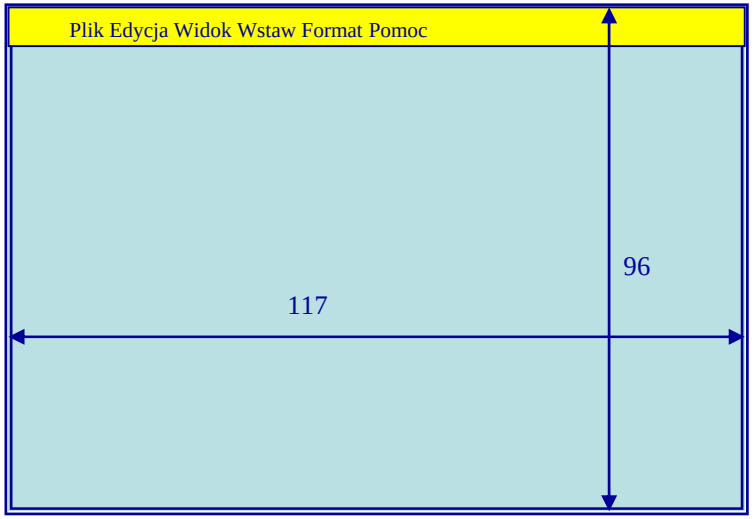
Wywołanie pozornie tej samej metody

```
pDokument->PokazDane();
```

powoduje wyświetlenie informacji o różnych obiektach, należących do różnych klas pochodnych

Polimorfizm zapewnia, że za każdym razem jest wywołana odpowiednia wersja metody PokazDane() – właściwa dla kolejnych obiektów, na które wskazuje pDokument

Programowanie obiektowe: zastosowanie do grafiki




```
class Okno
{
   private int x, y, h, w;

   public Okno(int szer, int wys)
   {
      x = y = 0;
      h = wys; w = szer;
   }
   public void Otworz() { ... }
   public void Zamknij() { ... }
   public void Przesun(int x, int y)
   { ... }
}
```

Praca domowa

Podaj własny przykład hierarchicznej struktury klas (dziedziczenie), projektując dla niej zmienne i najważniejsze metody np.

Klasa bazowa: pojazd
Zmienne: masa, położenie, prędkość
Metody: przyspiesz, skręć

Klasa pochodna: quad
Zmienne: ryk silnika (dB), ile promili alkoholu w pasażerach
Metody: staranuj świstaka, rozjedź ropuchę…