Instytut Informatyki UMCS
Zakład Technologii Informatycznych

Wykład 3

Dziedziczenie

dr Marcin Denkowski

Lublin, 2019

AGENDA

1. Relacje między klasami
2. Klasy pochodne
3. Funkcje wirtualne, polimorfizm
4. „Wirtualny konstruktor”

ENKAPSULACJA (HERMETYZACJA)

Definicja klasy T

```
class T
{
private:
    int m_var;              // skladowa prywatna
    void m_update();        // metoda prywatna

public:
    T();                    // konstruktor domyslny
    T(const T&);            // konstruktor kopiujacy
    ~T();                   // destruktor

    void setVar(int);       // setter
    int var();              // getter

    void method1();
    void method2();
};

int main(){

    T object;
    T* ptr_obj = new T;
}
```

Hermetyzacja
pola i metody prywatne

Interfejs publiczny
(pola i metody publiczne)

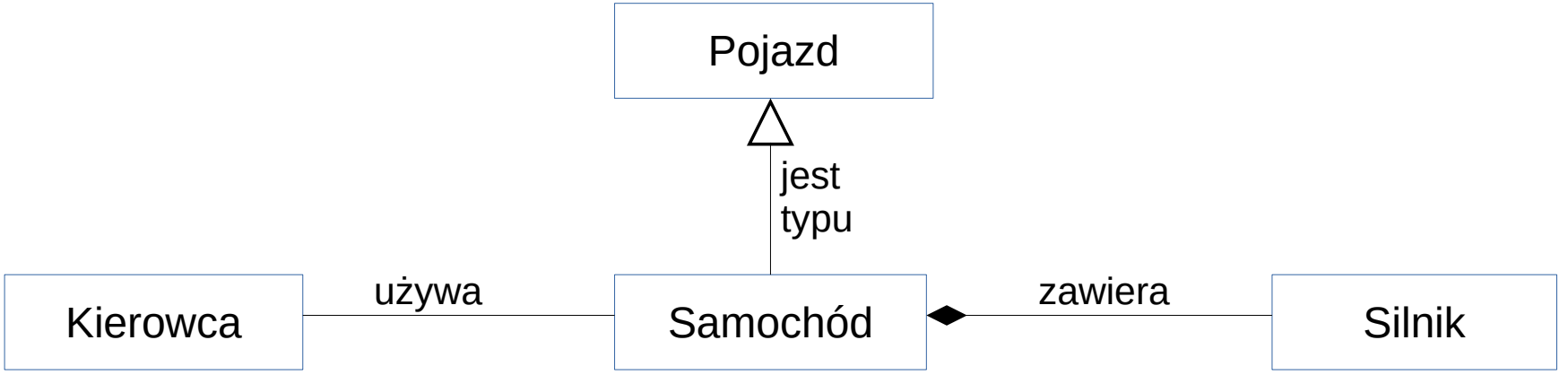
akcesory

pozostałe metody publiczne

Instancje klasy T
- obiekt automatyczny
- obiekt dynamiczny

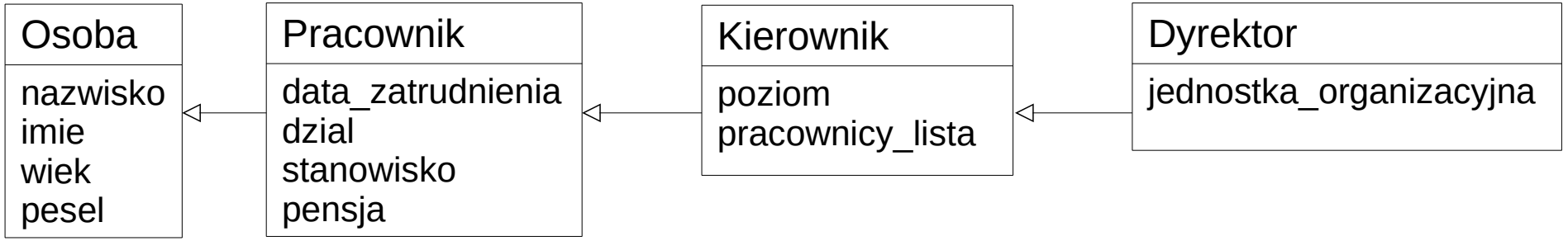
AGENDA

1. Klasy modelują rzeczywistość (programistyczną)
2. Istnieje kilka relacji między obiektami klas
   - klasa używa innej klasy
   - klasa zawiera inną klasę
   - klasa jest rodzajem innej klasy



DZIEDZICZENIE (GENERALIZACJA)

- *Class Inheritance (Generalization, Deriviation)*
- Reprezentuje relację *jest-czymś*
- Mechanizm dodawania udogodnień do istniejącej klasy, bez ponownego programowania
- Wspólny interfejs dla kilku różnych klas



DZIEDZICZNIE KLAS

- Klasa bazowa (*base class*)

```
class Base {
    int a,b,c;
public:
    Base();
    int get_a();
};
```

prywatne składowe, odziedziczone ale niedostępne bezpośrednio

- Klasa pochodna (*derived class*) – zawiera (prawie) wszystkie składowe klasy bazowej

```
class Derived : public Base {
    int d;
public:
    Derived();
    void set_d(int);
};
```

DZIEDZICZNIE KLAS - RZUTOWANIE

```
void f(Base b, Derived d)
{
    Base *pb    = &d;    // OK – rzutowanie w gore

    Derived *pd = &b;    // blad – rzutowanie w dol - niedozwolone
    pd->d = 2;           // blad

    pd = (Derived*)(pb);     // jawne rzutowanie w dol, OK ale…
    pd->d = 2;               // ponieważ pd nie musi wskazywac na klase
                             // pochodna

}
```

- Wskaźnik na klasa podstawowa może być użyty do wskazywania na obiekty klas pochodnych

KOLEJNOŚĆ KONSTRUKCJI KLAS

- Obiekt każdej klasy jest konstruowany w następującej kolejności:
  1) składowe klasy bazowej
  2) konstruktor klasy bazowej
  3) składowe klasy
  4) konstruktor klasy

- Destrukcja obiektu klasy następuje w kolejności:
  1) destruktor klasy
  2) składowe klasy
  3) destruktor klasy bazowej
  4) składowe klasy bazowej

- Klasy składowe i podstawowe konstruuje się w kolejności deklaracji w klasie, a niszczy w kolejności odwrotnej

KONSTRUKTORY W DZIEDZICZENIU

- Aby utworzyć obiekt klasy pochodnej
  – klasa bazowa musi posiadać konstruktor domyślny
  – klasa pochodna wywołuje go jawnie na liście inicjalizacyjnej

```
class Base {
    int a,b,c;
public:
    Base(int _a) { a = _a;}
    int get_a() { return a; }
};

class Derived : public Base {
    int d
public:
    Derived(int _a) : Base(_a) {}
};
```

SKŁADOWE CHRONIONE

- Specyfikator dostępu **`protected`**

```
class Base {
    int a,b;
protected:
    int c;
public:
    ...
};

class Derived : public Base {
    int d;
public:
    Derived();
    void set_c(int v)   { c = v; }
    void set_a(int v)   { a = v; } // nieprawidlowe
};
```

- Do składowych chronionych (**`protected`**) mają dostęp wszystkie metody tej klasy oraz klasy z niej dziedziczące

DZIEDZICZENIE Z DOSTĘPEM NIE-PUBLICZNYM

- Specyfikator dostępu **`protected`**

```
class Base {
    int a,b;
protected:
    int c;
public:
    int d;
};
```

```
class Derived : Base {}; // private
class Derived : private   Base {};
class Derived : protected Base {};
class Derived : public    Base {};
```

- Dziedziczenie z określonym dostępem pociąga za sobą odpowiednie ograniczenie widoczności składowych klasy bazowej

CO NIE PODLEGA DZIEDZICZENIU

- Dziedziczeniu nie podlegają:
  - konstruktory
  - destruktor
  - operator przypisania

METODY W DZIEDZICZENIU

```
class Base {
public:
    void print_class() {cout << "print base"; }
};

class Derived : public Base {
public:
    void print_class() {            //metoda nadpisuje bazowa
        cout << "print derived";
        Base::print_clas();         //jawne wywolanie metody bazowej
    }
};

…
{
    Base b;
    Derived d;
    b.print_class();   // "print base"
    d.print_class();   // "print derived" "print base"
}
```

FUNKCJE WIRTUALNE

- Korzystanie ze wskaźników nie rozwiązuje problemu wyboru wywoływanej metody

```
class Base {
public:
    void print_class() {cout << "print base"; }
};

class Derived : public Base {
public:
    void print_class() { cout << "print derived"; }
};


…
{
    Base* b = new Base;
    Base* d = new Derived;
    b->print_class();   // "print base"
    d->print_class();   // "print base"
}
```

FUNKCJE WIRTUALNE

- Zadeklarowanie metody jako wirtualnej umożliwia „późny” wybór odpowiedniej metody

```
class Base {
public:
    virtual void print_class() {cout << "print base"; }
};

class Derived : public Base {
public:
    void print_class() { cout << "print derived"; }
};


…
{
    Base* b = new Base;
    Base* d = new Derived;
    b->print_class();   // "print base" – metoda klasy bazowej
    d->print_class();   // "print derived" – metoda klasy pochodnej
}
```

POLIMORFIZM

- Polimorfizm – późne wiązanie metody wybieranej zależnie od obiektu, dla którego jest wywoływana
- Możliwe tylko w przypadku wskazywania obiektów wskaźnikiem lub referencją
- Polimorfizm nie zadziała jeżeli do metody odnosimy się poprzez nazwę zmiennej
- Zadeklarowanie metody jako wirtualnej powoduje, że funkcja ta jest wirtualna w klasie bazowej i wszystkich klasach pochodnych

POLIMORFIZM

- Dlaczego nie deklarować wszystkich metod jako wirtualne?
  – wydajność
  – model pojęciowy
- Wirtualne powinny być tylko te funkcje, które mogą wymagać powtórnej definicji w klasach pochodnych
- Każda funkcja wirtualna „tworzy” w klasie dodatkową ukrytą składową: tablicę funkcji wirtualnych **`vtab`** (*virtual function table*)
- Tablice **`vtab`** przechowują adresy funkcji wirtualnych zadeklarowanych dla obiektów tej klasy

WIRTUALNY DESTRUKTOR

- Pomimo że destruktor nie jest dziedziczony, wirtualny destruktor zapewnia polimorficzne jego wywołania podczas destrukcji obiektu

```
class Base {
public:
    virtual ~Base() {}
};

class Derived : public Base {
public:
    ~Derived() {}
};


…
{
    Base* b = new Derived;
    delete b;
}
```

”WIRTUALNY” KONSTRUKTOR

- Nie istnieje coś takiego jak wirtualny konstruktor, ale…
- Można go symulować (wzorzec fabryka)

```
class Base {
public:
    enum Type{BaseClass, DerivedClass1, DerivedClass2} type;
    static Base* New(Type id)
};
class Derived1 : public Base;
class Derived2 : public Base;

Base* New(Type id){
    switch(id) {
        case DerivedClass1:   return new Derived1;
        case DerivedClass2:   return new Derived1;
        default:              return new BaseClass;
    }
}
…

Base* b = Base::New(Base::DerivedClass2);
```

PRZECIĄŻANIE A NADPISANIE METODY

- Przeciążenie (*overloading*) – wprowadza nową metodę o identycznej nazwie ale innej sygnaturze – nie podlega polimorfizmowi
- Nadpisanie (*overriding*) – przesłonięcie metody o identycznej sygnaturze – podlega polimorfizmowi

```
class Base {
public:
    virtual void fun(int);
};

class Derived1 : public Base{
public:
    void fun(int) override;
    void fun(float);
}
```

- Modyfikator **`override`** zapewnia, że jest to metoda nadpisana