

# Podstawy programowania 2

dr inż. Piotr Szwed  
Katedra Informatyki Stosowanej  
C2, pok. 403

e-mail: [pszwed@agh.edu.pl](mailto:pszwed@agh.edu.pl)

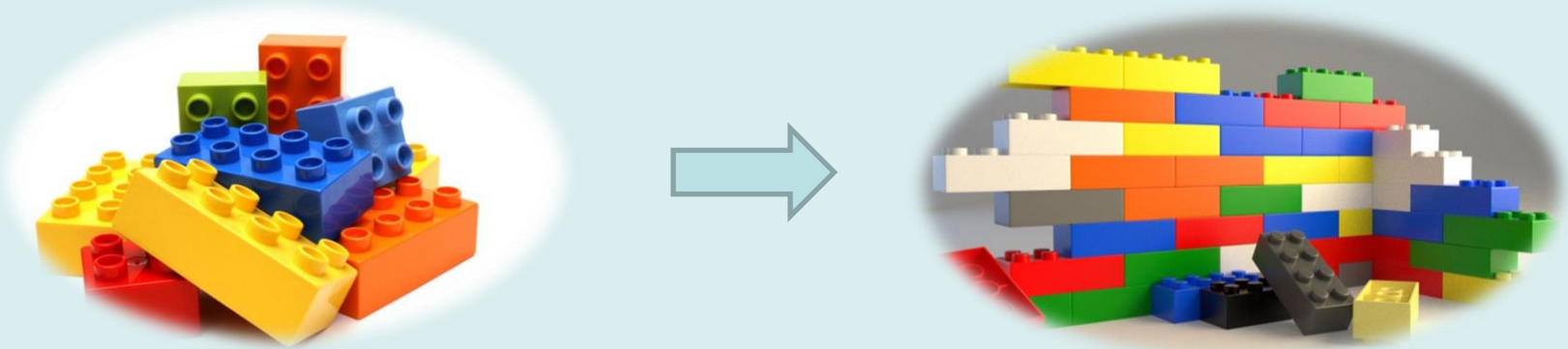
<http://home.agh.edu.pl/~pszwed/>

Aktualizacja: 17.03.2021

# **3. Kompozycja i dziedziczenie**

# Kompozycja i dziedziczenie

- Techniki programowania obiektowego zyskały bardzo dużą popularność ze względu na możliwość łatwego wykorzystania istniejącego kodu (ang. *reuse*).
- Biblioteki obiektowe definiują zazwyczaj zbiór użytecznych komponentów, które następnie bez żadnych zmian oryginalnego kodu można włączać do aplikacji.



# Kompozycja i dziedziczenie

Wykorzystanie istniejących komponentów może być realizowane dwiema drogami:

- przez **kompozycję** (agregację) czyli użycie istniejącego obiektu jako podobiektu (atrybutu) nowej klasy;
- przez **dziedziczenie**, czyli stworzenie nowej klasy obiektów rozszerzających funkcjonalność istniejącej klasy.

# Kompozycja - komponent

```
class Temperature{
    double value;
public:
    Temperature(double v = 0):value(v){}
    double getKelvin(){return value;}
    double getCelsius(){return value-272.15;}
    double getFahrenheit(){return value * 9 / 5 - 459.67;}
    void setKelvin(double v){value = v;}
    void setCelsius(double v){value = v + 272.15;}
    void setFahrenheit(double v){value=(v+459.67)*5/9;}
};
```

- Przechowuje wartość temperatury w stopniach Kelvina
- Potrafi dokonać konwersji do stopni Celsjusza i Fahrenheita

# Kompozycja

- Tworzymy klasę Weather (pogoda)
- Ma przechowywać:
  - Stan nieba (słońce, chmury, deszcz)
  - Temperaturę
  - Wilgotność
  - Prędkość wiatru

```
class Weather{
public:
    enum sky_state{sun,partly_cloudy,cloudy,rain,snow};

    enum sky_state sky;
    Temperature temp;
    double humidity;
    double windSpeed;
    //...
```

# Kompozycja

- Dodajemy konstruktor
- Opcjonalnie: funkcje dostępu (settery i gettery)

```
//...
    Weather(double t, double h, double w, sky_state s):
        temp(t),humidity(h),windSpeed(w),sky(s){
    }
    void setKelvin();
    void setCelsius(double v);
    void setFahrenheit(double v);
    double getKelvin();
    double getCelsius();
    double getFahrenheit();
};
```

# Kompozycja - delegacja

- Weather **deleguje** wykonanie metod do obiektu temp

```
double Weather::getKelvin(){
    return temp.getKelvin();
}

double Weather::getCelsius(){
    return temp.getCelsius();
}

double Weather::getFahrenheit(){
    return temp.getFahrenheit();
}
```

- Napisz settery...
- Jak skłonić kompilator aby utworzył funkcje inline

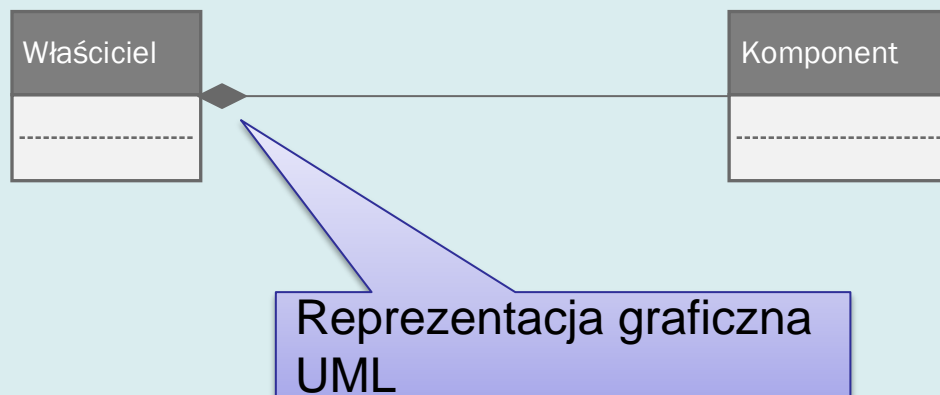


# Kompozycja – analiza obiektowa

Aby podczas analizy sprawdzić, czy pomiędzy klasami zachodzi agregacja lub kompozycja stosujemy frazy języka naturalnego:

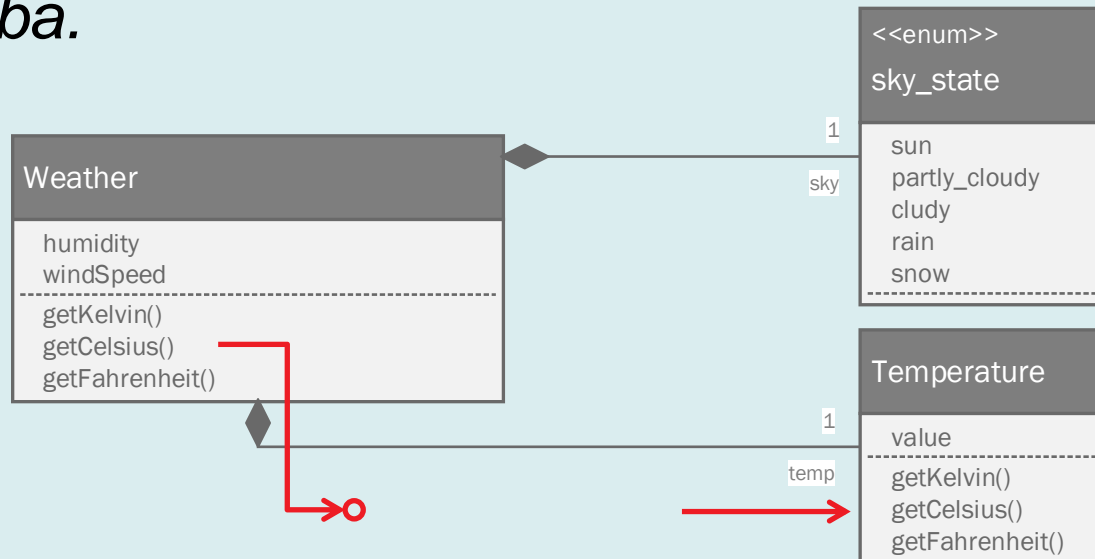
- ma, zawiera, obejmuje (agregacja)
- składa się (kompozycja)

Zazwyczaj przy usuwaniu obiektu nadrzędnego usuwane są jego komponenty (kiedy zburzymy budynek, znikną mieszkania).



# Kompozycja – analiza obiektowa

- Inne przykłady:
  - Powiat składa się z gmin
  - Mieszkanie składa się z pomieszczeń
  - Budynek zawiera mieszkania
  - *Pogoda (dane opisujące pogodę) składają się z temperatury, wilgotności, prędkości wiatru i stanu nieba.*



# Dziedziczenie

- Dziedziczenie umożliwia na stworzenie nowej definicji klasy (tzw. *klasy potomnej*) wykorzystującej istniejącą klasę (*klasę bazową*).
- Interfejs klasy bazowej jest w pełni zachowany. Obiekt klasy potomnej może przyjmować te same komunikaty – a więc należy również do klasy bazowej.
- Klasa potomna może dodawać nowe elementy do interfejsu, a także w odmienny sposób reagować na komunikaty zdefiniowane w interfejsie klasy bazowej.

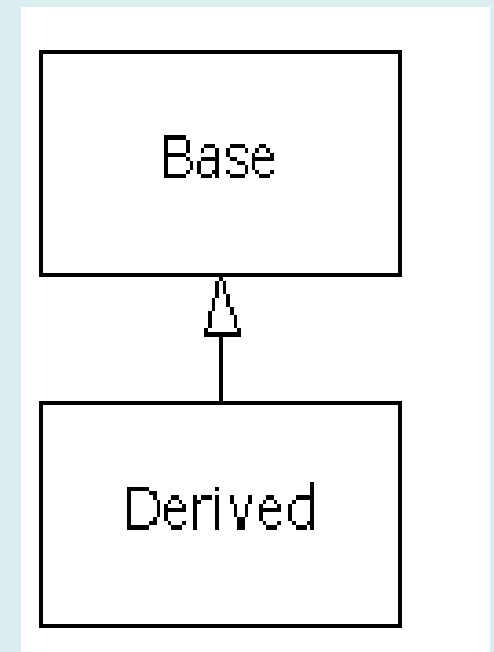
# Dziedziczenie

Często klasa bazowa nazywana jest *generalizacją* natomiast klasa potomna *specjalizacją*.

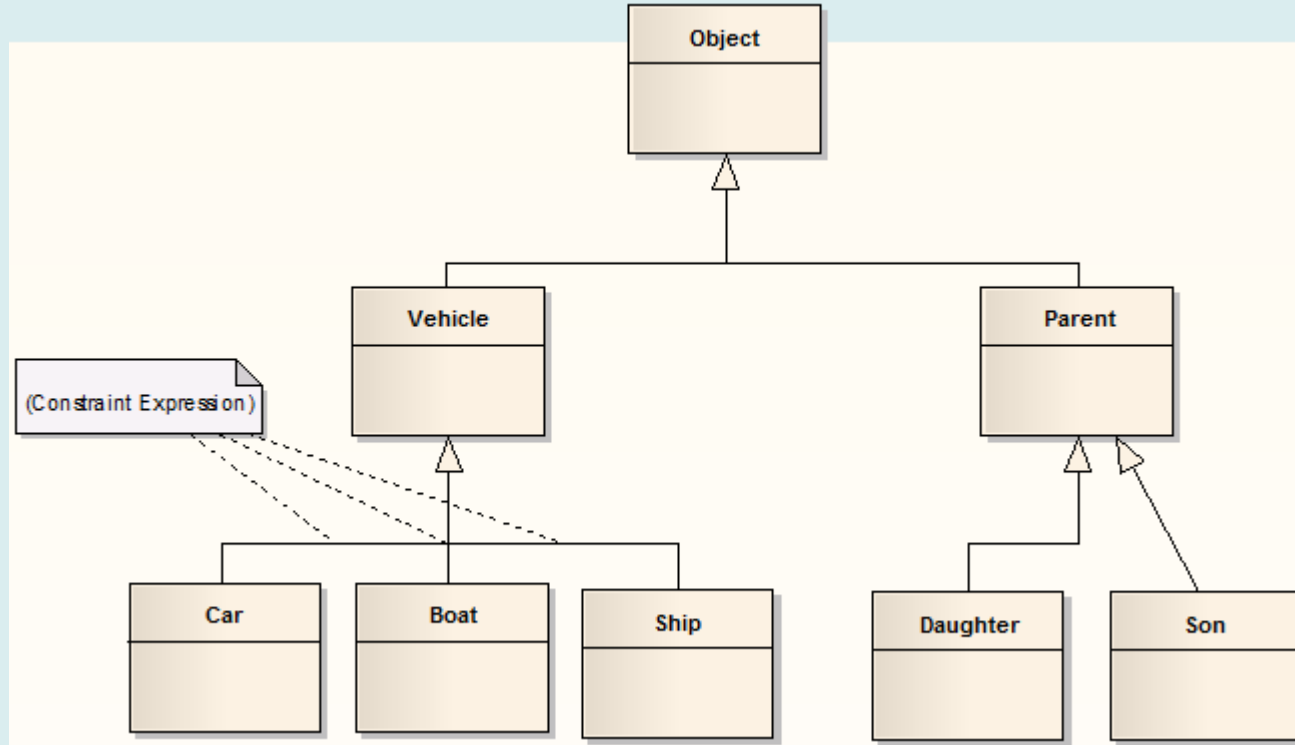
Decydując się na ustalenie, czy pomiędzy obiektami należy wprowadzić relację dziedziczenia posługujemy się frazą „*jest, jest rodzajem*”

- B jest (jest rodzajem) A
- Okrąg jest Kształtem
- Trójkąt jest Kształtem
- SamochódOsobowy jest Pojazdem

Reprezentacja graficzna



# Dziedziczenie



- Klasy połączone relacją dziedziczenia mogą tworzyć rozbudowane hierarchie
- W wielu językach mają one wspólny korzeń – klasę Object, stąd mówimy o hierarchiach obiektowych

# Dziedziczenie

Klasa MeltingTemperature (temperatura topnienia) rozszerza funkcjonalność temperatury – dodaje nazwę substancji.

```
class MeltingTemperature :public Temperature{
    string name;
public:
    MeltingTemperature(const char*n,double v):
        Temperature(v),name(n){
    }
    const char*getName(){return name.c_str();}
    // temperatura?
    bool setCelsius(double v){
        if(v<-272.15 || v>=2000) return false;
        setCelsius(v);
        return true;
    }
};
```

# Dziedziczenie – przeddefiniowanie metod

```
MeltingTemperature water("water", 272.15);  
cout<<water.getCelsius()<<endl;  
water.setCelsius(-500);  
cout<<water.getCelsius()<<endl;  
water.Temperature::setCelsius(-500);  
cout<<water.getCelsius()<<endl;
```

W obu przykładach  
wypisze

0  
0  
-500

```
MeltingTemperature water("water", 272.15),  
cout<<water.getCelsius()<<endl;
```

```
MeltingTemperature&r1 = water;  
r1.setCelsius(-500);  
cout<<r1.getCelsius()<<endl;
```

```
Temperature&r2 = water;  
r2.setCelsius(-500);  
cout<<r2.getCelsius()<<endl;
```

Wywołanie  
setCelsius w  
klasie bazowej

# Dziedziczenie - konstruktor

Klasa bazowa

Atrybut

```
class MeltingTemperature
:public Temperature{
    string name;
public:
    MeltingTemperature(const char*n,double v):
        Temperature(v),name(n){
    }
    //...
};
```

Argument konstruktora klasy bazowej Temperature

Argument konstruktora atrybutu name



# Składnia dziedziczenia

```
class derived : base-list {...}
```

```
base-list :
```

```
    base-specifier
```

```
    base-list , base-specifier
```

```
base-specifier :
```

```
    [virtual] [access-specifier] complete-class-name
```

```
access-specifier :
```

```
    private
```

```
    protected
```

```
    public
```

## Przykłady

```
class X:public A {...} // jednobazowe
```

```
class Y:public A, public B {...} // wielobazowe
```

```
class Z: A, public B, protected C, virtual D  
{...}
```

# Specyfikacja dostępu

Słowa kluczowe `private`, `protected`, `public` definiują prawa dostępu w klasie pochodnej do elementów klasy bazowej. Brak słowa kluczowego (w przypadku deklaracji klasy) jest równoważny zastosowaniu dostępu `private`.

Dostęp w klasie bazowej	Sposób dziedziczenia	Dostęp w klasie pochodnej
<code>public</code> <code>protected</code> <code>private</code>	<code>public</code>	<code>public</code> <code>protected</code> Brak dostępu
<code>public</code> <code>protected</code> <code>private</code>	<code>protected</code>	<code>protected</code> <code>protected</code> Brak dostępu
<code>public</code> <code>protected</code> <code>private</code>	<code>private</code>	<code>private</code> <code>private</code> Brak dostępu

# Przykład

Po zmianie definicji klasy `MeltingTemperature` na

```
class MeltingTemperature : private Temperature  
{...}
```

za pośrednictwem klasy `MeltingTemperature` nie mamy dostępu z zewnątrz do żadnego komponentu (pola, metody, definicji) klasy `Temperature`.

<pre>Temperature t(0); t.setCelsius(0);</pre>	Poprawne
<pre>MeltingTemperature mt; mt.Temperature::setCelsius(0);</pre>	<code>Temperature::setCelsius</code> niedostępne jako metoda obiektu klasy <code>MeltingTemperature</code>
<pre>Temperature&amp;rt = mt;</pre>	Konwersja typu jest potencjalnie możliwa, ale zabroniona przez tryb dziedziczenia

# Przykład

```
class Light{
protected:
    double voltage;
public:
    Light(){voltage=0;}
    virtual void on(){voltage = 230;}
    virtual void off(){voltage=0;}
};
```

Klasa `Light` definiuje chroniony atrybut `voltage` i dwie publiczne metody `on()` oraz `off()`.

```

class DimmableLight:public Light{
    double level;
    void updateVoltage(){
        voltage=level*230;
        std::cout<<voltage<<std::endl;
    }
public:
    DimmableLight(){
        level=0;
        updateVoltage();}
    void dim(){
        if(level>=0.1)level -= 0.1;
        updateVoltage();}
    void brighten(){
        if(level<=0.9)level += 0.1;
        updateVoltage();}
    void on(){
        level=1.0;
        updateVoltage();}
    void off(){
        level=0;
        updateVoltage();}
};

```

- Klasa DimmableLight dziedziczy komponenty klasy Light.
- Dodano w niej prywatny atrybut level i prywatną metodę updateVoltage();
- Interfejs został rozszerzony o dwie dodatkowe metody dim() i brighten().
- Dodatkowo, zdefiniowano metody on() i off()

# Wywołanie

```
int main(){
    Light simpleLight;
    simpleLight.on();
    simpleLight.off();

    DimmableLight advancedLight;
    advancedLight.on();
    advancedLight.dim();
    advancedLight.dim();
    advancedLight.off();
}
```

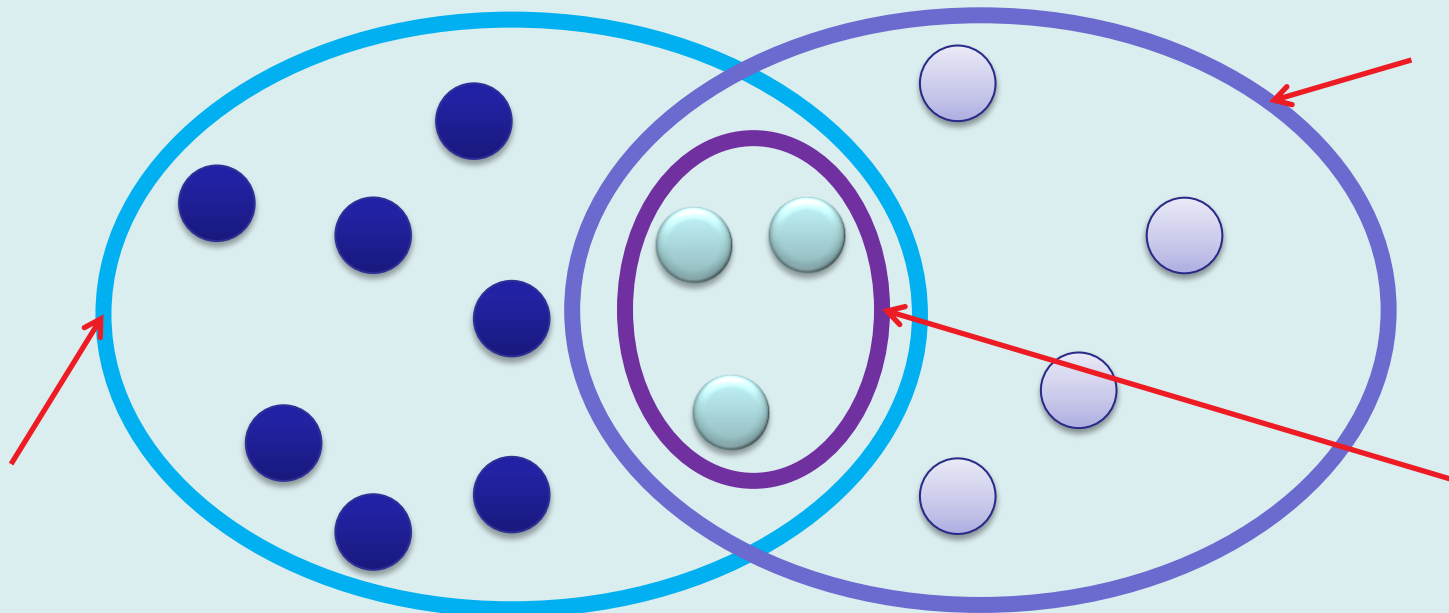
Wynik:

```
0
230
207
184
0
```

# **Dziedziczenie wielobazowe**

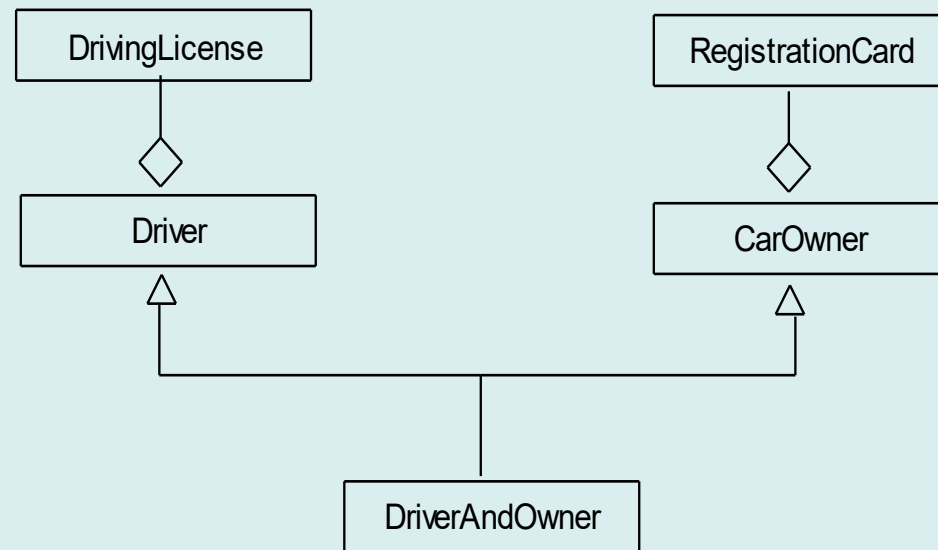
# Dziedziczenie wielobazowe (wielokrotne)

- Hierarchie obiektów są bardzo często rzutem opisu rzeczywistej dziedziny na konstrukcje języka programowania.
- Niejednokrotnie analizując dziedzinę, możemy podać obiekty, które należą do różnych klas, a równocześnie żadna z klas nie może zostać uznana za generalizację lub specjalizacją drugiej.





# Dziedziczenie wielobazowe - przykład



- Charakterystyczną cechą klasy `Driver` jest posiadanie prawa jazdy (`DrivingLicense`). Charakterystyczną cechą posiadacza pojazdu `CarOwner` jest posiadania dowodu rejestracyjnego `RegistrationCard`.
- Bardzo często występują obiekty, które należą do obu klas równocześnie, a więc jak `DriverAndOwner` powinny dziedziczyć po dwóch klasach bazowych.

# Dziedziczenie wielobazowe

- Dziedziczenie wielobazowe jest konstrukcją pozwalającą na stworzenie klasy potomnej dziedziczącej atrybuty i metody kilku klas bazowych.
- W definicji klasy potomnej podaje się na liście klas bazowych kilka klas – potencjalnie stosując różne prawa dostępu.
- W szczególnym przypadku nazwy atrybutów i metod klas bazowych mogą się pokrywać. Wówczas zazwyczaj można uzyskać dostęp do konkretnych składowych podając nazwę klasy bazowej:

```
object_name.Base1::attribute  
object_name.Base2::method(...)
```

# Przykład

```
class A1
{
    int i;
public:
    A1(int _i):i(_i){}
    void f(){
        printf("A1[i=%d] ",i);
    }
};

class A2
{
public:
    A2(){}
    void f(){
        printf("A2 ");
    }
};
```

```
class B: public A1, public A2 {
public:
    B():A2(),A1(7){}

    void f(){
        printf("B[ ");
        A1::f();
        A2::f();
        printf("]");
    }
};

int main(){
    B b;
    b.f();
    b.A1::f();
    b.A2::f();
}
```

# Przykład

```
B():A2(),A1(7){}
```

- Konstruktor klasy potomnej. Na liście inicjalizacyjnej przekazywane są parametry do konstruktorów klas A1 i A2. Wywołanie A2() można pominąć.
- Kolejność wołania konstruktorów jest zgodna z kolejnością deklaracji, a nie kolejnością na liście inicjalizacyjnej.
- Metoda B::f() przeddefiniowuje metody A1::f() oraz A2::f(). Są one jednak dalej dostępne przez podanie nazwy poprzedzonej operatorem zasięgu.

```
int main(){  
    B b;  
    b.f();  
    b.A1::f();  
    b.A2::f();  
}
```

# Dziedziczenie wielokrotne klasy bazowej

W przypadku rozbudowanych hierarchii klas, pewna klasa może zostać odziedziczona wielokrotnie.

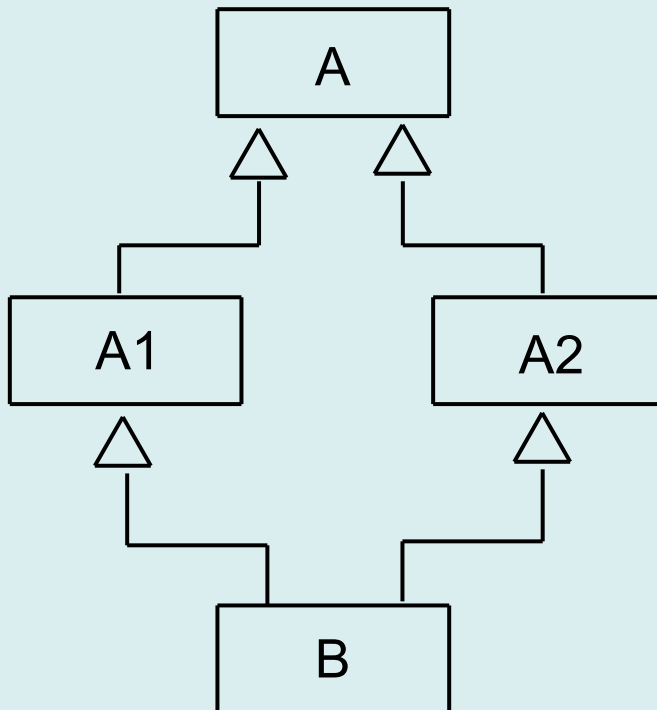
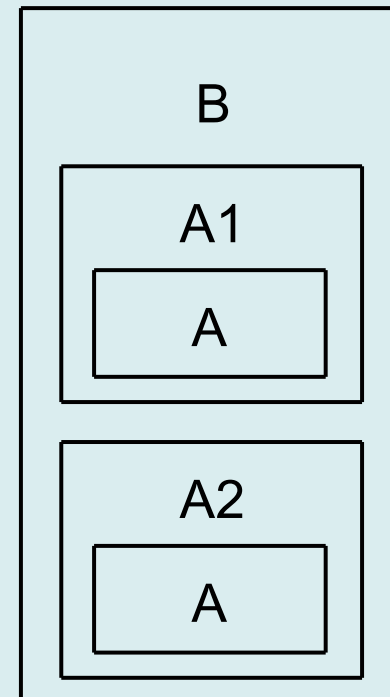


Diagram klas UML



A1 zawiera podobiekt klasy bazowej A,  
A2 zawiera **inny** podobiekt klasy A.  
Klasa B zawiera podobiekt A1 i A2. 29

# Przykład

```
class A {  
    int k;  
public:  
    A(int _k):k(_k){}  
    void f(){  
        printf("A[k=%d]", k);  
    }  
};
```

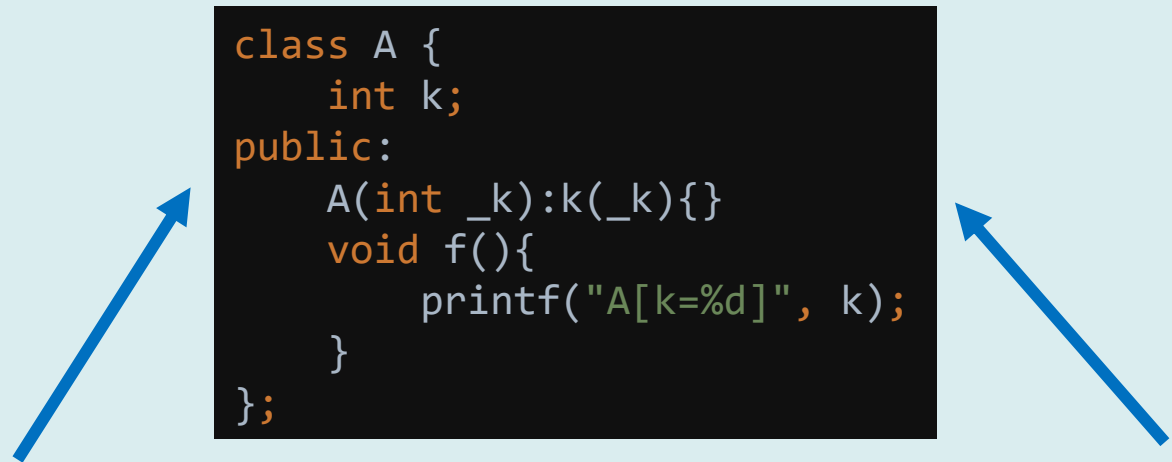
```
class A1 : public A {  
    int i;  
public:  
    A1(int _i):A(1),i(_i){}  
    void f(){  
        printf("A1[i=%d] ",i);  
    }  
};
```

```
class A2 : public A  
{  
public:  
    A2():A(2){}  
    void f(){printf("A2 ");}  
};
```

```
class B: public A1, public A2  
{  
public:  
    B():A2(),A1(7){}  
};
```

# Przykład

```
class A {
    int k;
public:
    A(int _k):k(_k){}
    void f(){
        printf("A[k=%d]", k);
    }
};
```



```
class A1 : public A {
    int i;
public:
    A1(int _i):A(1),i(_i){}
    void f(){
        printf("A1[i=%d] ",i);
    }
};
```

```
class A2 : public A
{
public:
    A2():A(2){}
    void f(){printf("A2 ");}
};
```

Podobiekty A w A1 oraz A w A2 różnią się stanem:

- W pierwszym przypadku  $k==1$
- W drugim  $k==2$

# Przykład

```
class A {
    int k;
public:
    A(int _k):k(_k){}
    void f(){
        printf("A[k=%d]", k);
    }
};
```

```
class A1 : public A {
    int i;
public:
    A1(int _i):A(1),i(_i){}
    void f(){
        printf("A1[i=%d] ",i);
    }
};
```

Możemy wywołać funkcje f()  
podobiektów A1 oraz A2

```
class A2 : public A
{
public:
    A2():A(2){}
    void f(){printf("A2 ");}
};
```

```
int main(){
    B b;
    b.A1::f();
    b.A2::f();
};
```

A1[i=7] A2



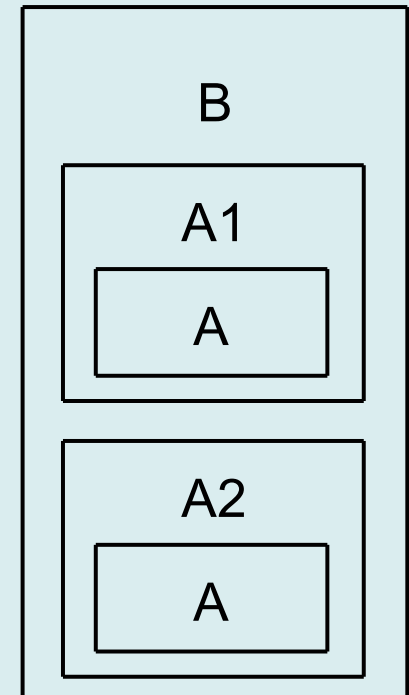
# Przykład

```
int main(){  
    B b;  
    b.A::f()  
};
```



error: 'A' is an ambiguous base of 'B'  
b.A::f()  
 ^

- Kod tej postaci jest błędny ze względu na niejednoznaczność (ang. *ambiguity*).
- Co miałyby wypisać funkcja `f()`
  - `A[k=1]` ?
  - `A[k=2]` ?
- W praktyce nie można bezpośrednio uzyskać dostępu do żadnej instancji klasy `A` osadzonej w `B`.



# Przykład

- Jednym z możliwych rozwiązań jest wyłuskanie podobiektu poprzez rzutowanie w górę (*upcasting*):

```
A2&ra = b;
```

```
ra.A::f();
```

- Podobnie, można posłużyć się wskaźnikami:

```
((A*)(A2*)&b) ->f()
```

# Dziedziczenie wirtualne

Definiując klasy A1 i A2 jako dziedziczące wirtualnie po klasie A można zapewnić użycie wspólnego obiektu klasy A w klasach dziedziczących równocześnie po A1 i A2.

```
class A {
    int k;
public:
    A(int _k):k(_k){}
    void f(){
        printf("A[k=%d]", k);
    }
};

class A1 : virtual public A
{
    int i;
public:
    A1(int _i):A(1),i(_i){}
    void f(){
        printf("A1[i=%d] ",i);
    }
};
```

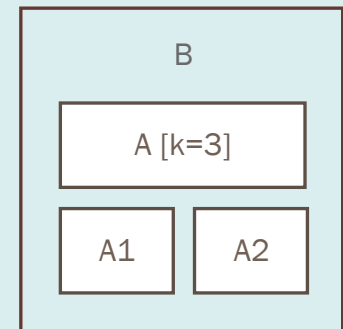
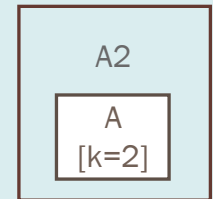
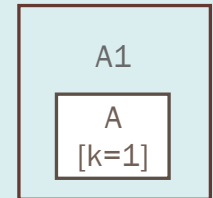
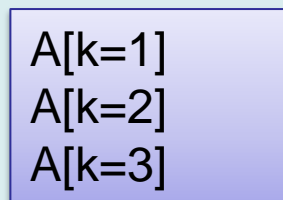
```
class A2 : virtual public A
{
public:
    A2():A(2){}
    void f(){printf("A2 ");}
};

class B:
    public A1,
    public A2
{
public:
    B():A2(),A1(7),A(3){}
};
```

# Dziedziczenie wirtualne

- Samodzielny obiekt klasy A1 będzie zawierał podobiekt A ze stanem k=1
- Samodzielny obiekt klasy A2 będzie zawierał podobiekt A ze stanem k=2
- Obiekt klasy B będzie zawierał trzy podobiekty: A (ze stanem k=3), A1 oraz A2. Konstruktor klasy B musi jawnie zainicjować składowy obiekt klasy A. Inicjacje za pośrednictwem konstruktorów A1 oraz A2 są ignorowane.

```
int main(){  
    A1 a1(23);  
    a1.A::f();  
    A2 a2;  
    a2.A::f();  
    B b;  
    b.A::f();  
};
```

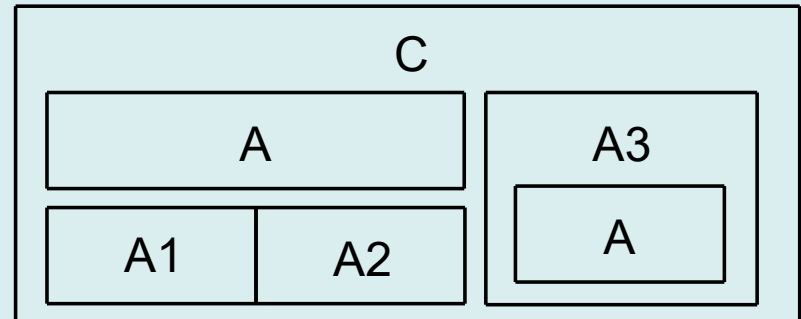


# Ograniczenia dziedziczenia wirtualnego

- Dziedziczenie wirtualne jest związane wyłącznie ze ścieżką dziedziczenia, a nie z klasą.
- Dana klasa może zostać równocześnie odziedziczona wirtualnie i niewirtualnie, stąd aby wykorzystać zalety dziedziczenia wirtualnego, należy je konsekwentnie stosować na wszystkich ścieżkach.

```
class A3 : public A
{
public:
    A3():A(4){}
};

class C : public A1, public A2,
         public A3
{
public:
    C():A(5),A1(7),A2(),A3(){}
};
```



# Ograniczenia dziedziczenia wirtualnego

- W praktyce, dziedziczenie wirtualne stosuje się rzadko.
- Z reguły biblioteki budują hierarchie klas dziedziczących wyłącznie publicznie (niewirtualnie).
- Aby usunąć problemy konfliktu i niejednoznaczności nazw, należałoby zmodyfikować tryb dziedziczenia po klasach znajdujących się u podstawy hierarchii, a więc zmodyfikować kod bibliotek (czego zawsze unika się.)