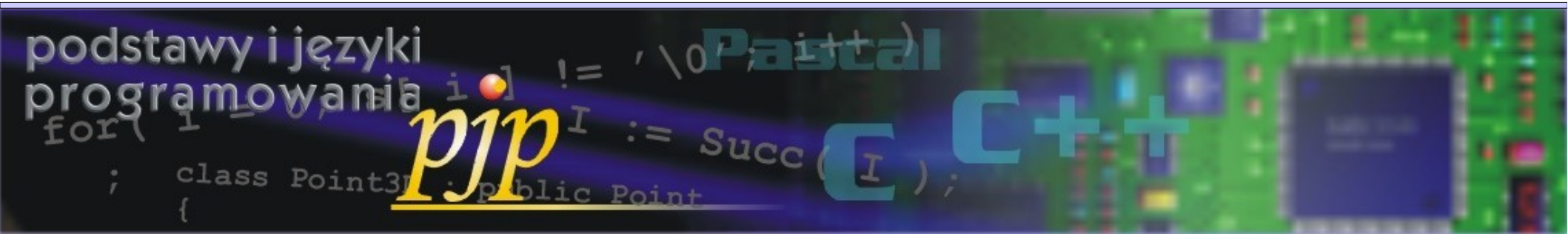
Projektowanie obiektowe

Roman Simiński
roman.siminski@us.edu.pl
www.siminskionline.pl

Polimorfizm



Koncepcja polimorfizmu

Słowo *polimorfizm* pochodzi od dwóch greckich słów:

- *poly* czyli *wiele*,
- *morphos* czyli *postać*.

Polimorfizm oznacza zatem *wielopostaciowość*.

Polimorfizm w *programowaniu* bywa bardzo różnie definiowany — statyczny, dynamiczny, metod, typów, itp., itd..

Od lat *koncepcja polimorfizmu* w programowaniu jest przyczyną gorących sporów na forach internetowych poświęconych programowaniu.

To *jak zdefiniować* polimorfizm są nieco mniej istotne od tego, *po co* jest polimorfizm.

Po co jest polimorfizm w programowaniu?

Polimorfizm pozwala na osiągnięcie pozornie sprzecznych celów:

- **Abstrakcji** – programowanie na na wysokim poziomie ogólności, ukierunkowane na *identyfikowanie głównych obiektów* w systemie i ich *najważniejszych zachowań* oraz na definiowanie *głównych zależności* pomiędzy *obiektami* oraz ogólnych *scenariuszy* ich *zachowań*.
- **Rozszerzalności** – programowanie pozwalające na definiowanie *szczególnych zachowań* obiektów, realizowanych w ramach *ogólnych scenariuszy*.
- **Antycypacji** – programowanie pozwalające na przewidywanie oraz odwzorowanie w kodzie przyszłych, choć jeszcze nie do końca znanych *zmian* i *modyfikacji* wymagań dla systemu.

Dzięki polimorfizmowi, istniejący kod, również skompilowany, może wykonywać nieznane uprzednio akcje, realizowane zwykle przez specyficzne obiekty, utworzone dla konkretnie rozważanego problemu

Po co jest polimorfizm w programowaniu?

- Polimorfizm pozwala na rozszerzanie systemu bez konieczności modyfikacji jego podstawowej struktury i zasady działania.
- Osiąga się to poprzez dołączenia do systemu nowych, specyficznych obiektów, które w łatwy sposób są włączane do *istniejącej struktury* i łatwo wpisują się w *przyjętą zasadę działania* systemu.
- Dzięki polimorfizmowi istniejący kod może zacząć działać inaczej niż wcześniej, co można osiągnąć bez zmiany tego kodu.

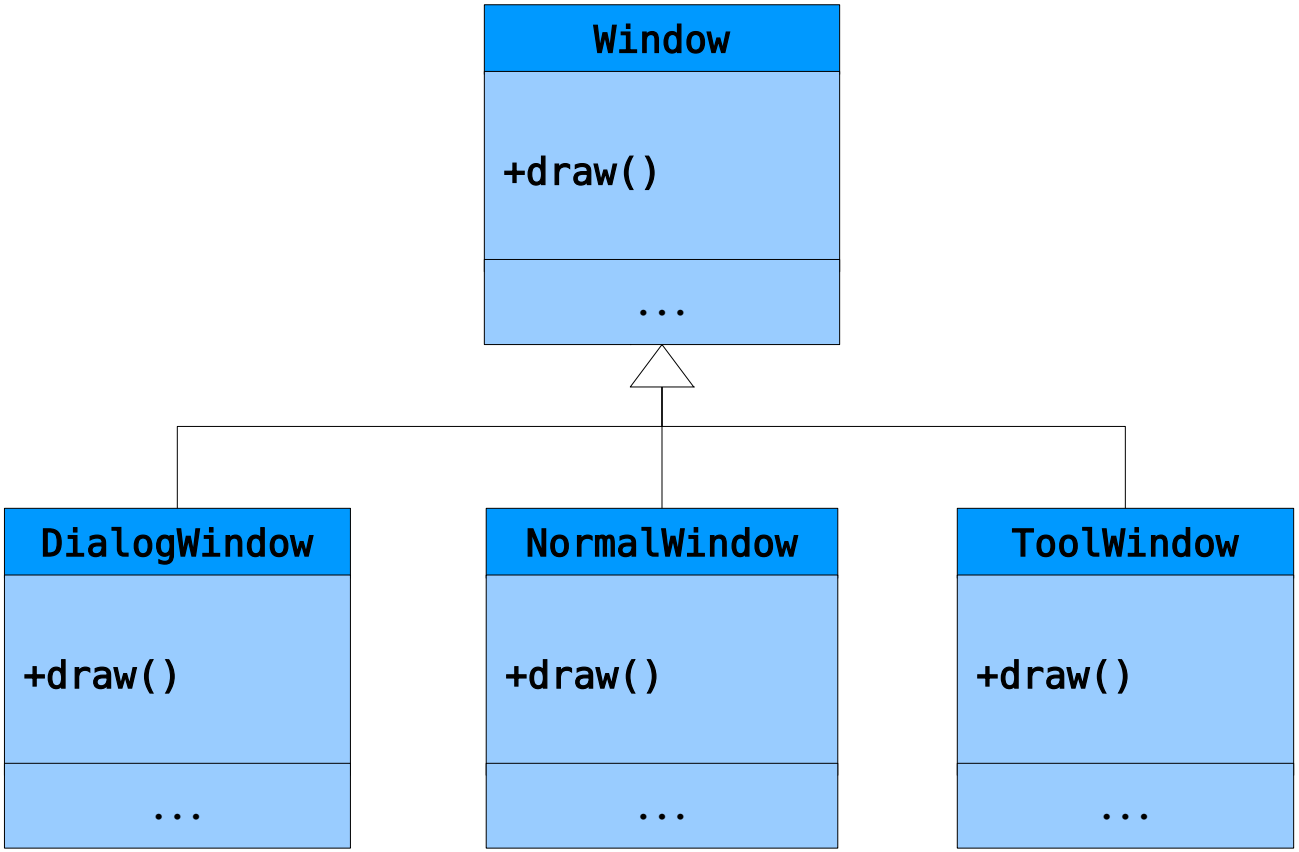
Koncepcja polimorfizmu – jak ją rozumieć tu i teraz

Dla naszych potrzeba założymy, że *polimorfizm* w *programowaniu obiektowym* oznacza możliwość występowania obiektu w *różnych jego wersjach*, które mogą *różnie* reagować na *ten sam* komunikat.

Polimorfizm to zatem *zdolność obiektów do różnych zachowań*, w zależności od bieżącego kontekstu wykonania programu.

Skąd się biorą obiekty polimorficzne?

Załóżmy, że budujemy obiektową bibliotekę GUI, zawierającą klasy opisujące różne typy okien. Klasy powstają w wyniku dziedziczenia, na bazie klasy *Window*:



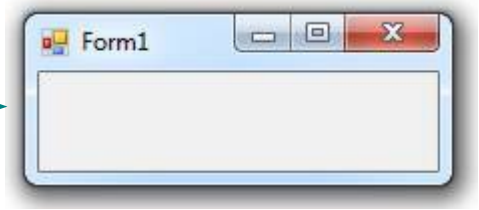
Każda klasa posiada własną implementację funkcji *draw*, rysującej okno w specyficzny dla danej klasy sposób.

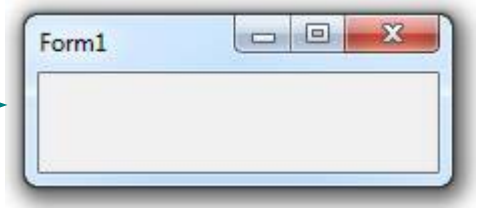
Skąd się biorą obiekty polimorficzne?

Załóżmy, że mamy jakiś C-podobny, obiektowy język programowania, definiujemy różne klasy opisujące okna systemowe, wywodzące się ze wspólnej klasy bazowej:

```
class Window
{
  void draw() { … }
}

class DialogWindow inherited from Window
{
  void draw() { … }
}

class NormalWindow inherited from Window
{
  void draw() { … }
}

class ToolWindow inherited from Window
{
  void draw() { … }
}
```

Fraza **inherited from** oznacza dziedziczenie





Skąd się biorą obiekty polimorficzne?

Załóżmy, że można utworzyć obiekty zdefiniowanych klas:

```
Window          w;
Dialogwindow    d;
NormalWindow    f;
ToolWindow      t
```

Załóżmy, że można utworzyć referencję do obiektów każdej z klas, w tym referencję do obiektów klasy *Window*:

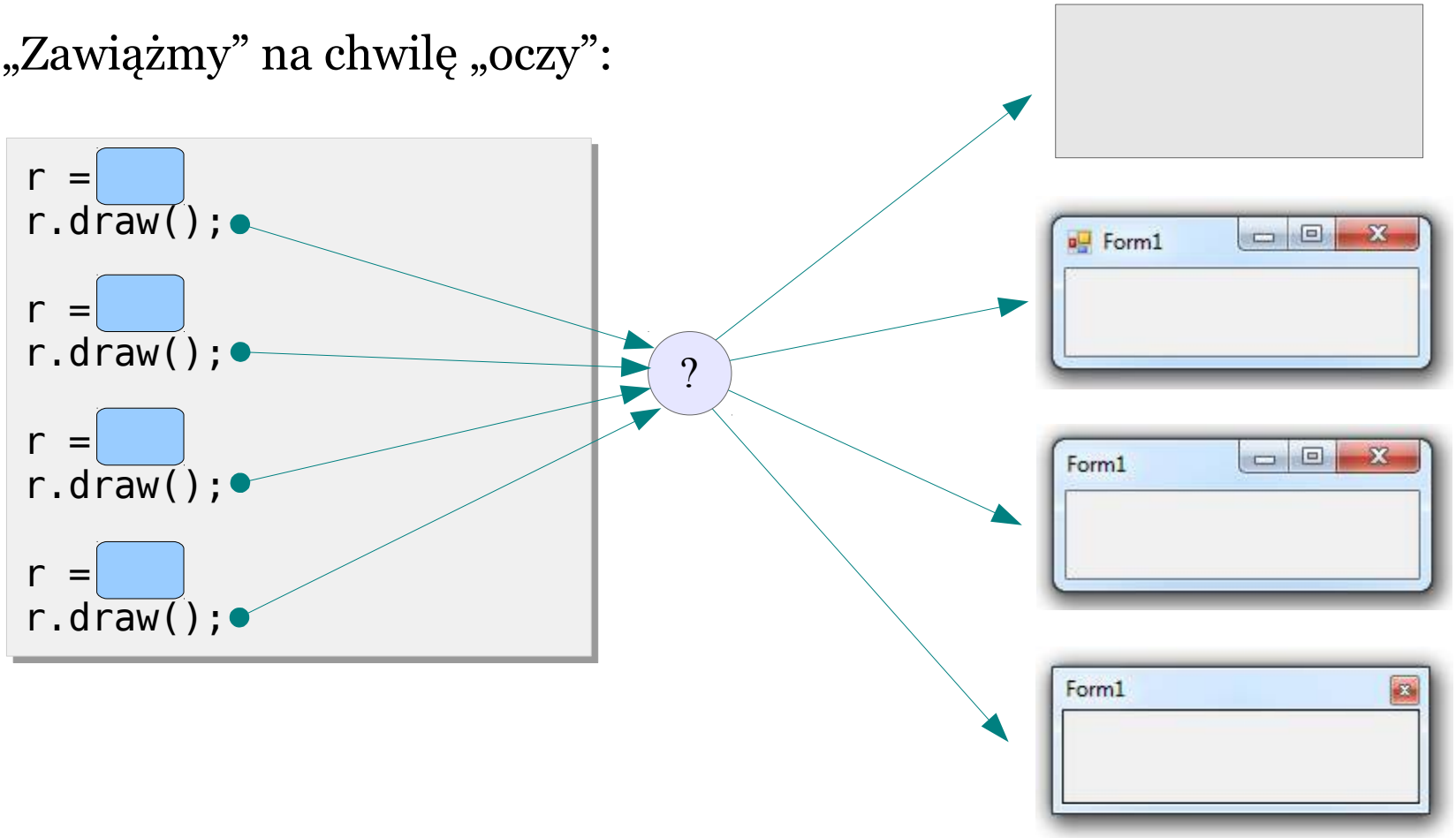
```
Window refereed by r;
```

Zgodnie z semantyką dziedziczenia, zachowuje ono relację *is_a*, zatem wolno dokonać następujących przypisań i wywołań:

```
r = w;
r.draw();
r = d;
r.draw();
r = f;
r.draw();
r = t
r.draw();
```

Skąd się biorą obiekty polimorficzne?

„Zawiążmy” na chwilę „oczy”:



W której postaci występuje *r*?

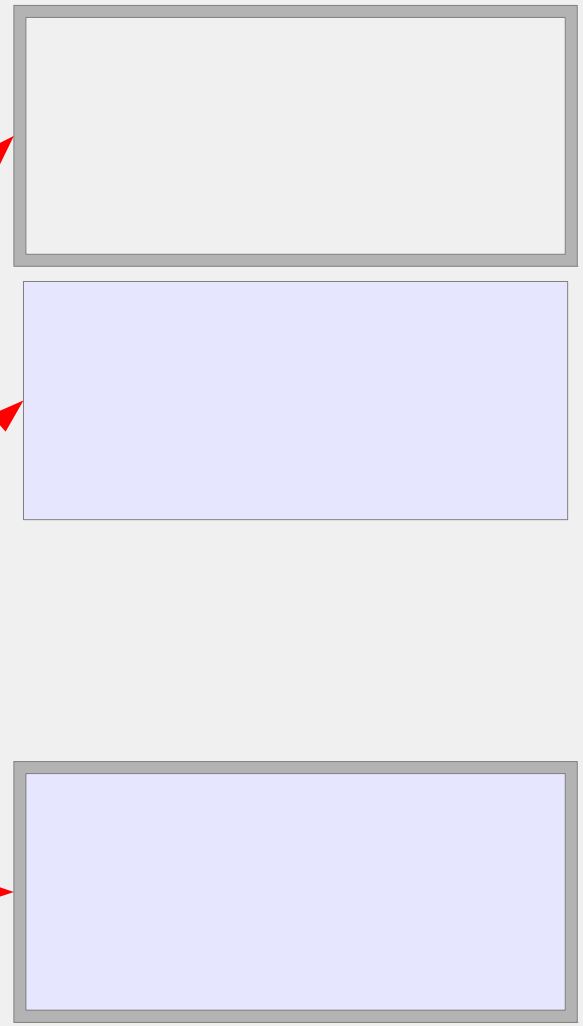
Która wersja metody **draw** zostanie wywołana i dlaczego?

Wczesne i późne wiązanie
Funkcje wirtualne
C++

Przykład, klasa reprezentująca okno — Window

- Załóżmy, że w pewnej bibliotece obsługi GUI zdefiniowano klasę **Window**:

```
class Window
{
  public:
    Window()
    {
    }

    void drawFrame()
    {
      // Tu rysowanie obramowania okna
    }

    void drawInterior()
    {
      // Tu rysowanie wnętrza okna
    }

    void draw()
    {
        drawFrame();
        drawInterior();
    }
};
```



Przykład, klasa reprezentująca okno — Window, cd. ...

- Wyświetlanie okna realizowane może być w następujący sposób:

```
class Window
{
 . . .
};

int main()
{
  Window w;

  w.draw();

  return EXIT_SUCCESS;
}
```

- Załóżmy, że mechanizm obsługi okien klasy **Window** nam odpowiada, nie odpowiada nam jednak wygląd okna – zarówno *wnętrza* jak i *obramowania*, chcemy czegoś takiego:

Przykład, klasa reprezentująca okno — Window, cd. ...

- Aby nie zaciemniać kodu, załóżmy że odpowiednie funkcje składowe wyświetlają komunikaty, pozwalające na stwierdzenie, że dana funkcja zadziałała.

```
class Window
{
  public:
    Window()
    {
    }

    void drawFrame()
    {
      cout << "Window's drawFrame()" << endl;
    }
    void drawInterior()
    {
      cout << "Window's drawInterior()" << endl;
    }

    void draw()
    {
        drawFrame();
        drawInterior();
    }
};
```

Przykład, klasa reprezentująca okno — Window, cd. ...

- Działanie programu wykorzystującego klasę **Window**.

```
class Window
{
    . . .
    void draw()
    {
      drawFrame();
      drawInterior();
    }
};

int main()
{
  Window w;

  w.draw();

  return EXIT_SUCCESS;
}
```

```
Window's drawFrame()
Window's drawInterior()
```

Przykład, klasa Window nie do końca nam odpowiada

- Chcemy innego wypełnienia i innego obramowania.
- Wiemy, że istnieje *dziedziczenie*, pozwalające na *przejęcie cech innych klas*.
- Dziedziczenie pozwala również na zmianę *zachowania obiektu* klasy pochodnej.

```
class MyWindow : public Window
{
  public:
    MyWindow() : Window()
    {
    }

    void drawFrame()
    {
      cout << "MyWindow's drawFrame()" << endl;
    }

    void drawInterior()
    {
      cout << "MyWindow's drawInterior()" << endl;
    }
};
```

Tworzymy nową klasę MyWindow

Redefinicja funkcji składowych

Oczekujemy, że obiekt klasy **MyWindow** będzie inaczej reagował na komunikaty **drawFrame** oraz **drawInterior**

Przykład, klasa MyWindow — testujemy przedefiniowane funkcje

```
class MyWindow : public Window
{
  public:
    MyWindow() : Window()
    {
    }
    void drawFrame()
    {
      cout << "MyWindow's drawFrame()" << endl;
    }
    void drawInterior()
    {
      cout << "MyWindow's drawInterior()" << endl;
    }
};

int main()
{
  MyWindow w;

  w.drawFrame();
  w.drawInterior();

  return EXIT_SUCCESS;
}
```

```
MyWindow's drawFrame()
MyWindow's drawInterior()
```

Przykład, klasa MyWindow — testujemy przedefiniowane funkcje

```
class MyWindow : public Window
{
  public:
    MyWindow() : Window()
    {
    }
    void drawFrame()
    {
      cout <<
    }
    void draw
    {
      cout <<
    }
};

int main()
{
  MyWindow w;

  w.drawFrame();
  w.drawInterior();

  return EXIT_SUCCESS;
}
```

Wszystko OK, ale przecież rysowanie okna miała realizować po prostu funkcja **draw()**, jej działanie nam odpowiada i jej nie przedefiniowaliśmy.

Spróbujmy zatem napisać to wywołując **draw()**...

```
MyWindow's drawFrame()
MyWindow's drawInterior()
```

Przykład, klasa MyWindow — testujemy przedefiniowane funkcje

```
class MyWindow : public Window
{
  public:
    MyWindow() : Window()
    {
    }
    void drawFrame()
    {
      cout << "MyWindow's drawFrame()" << endl;
    }
    void drawInterior()
    {
      cout << "MyWindow's drawInterior()" << endl;
    }
};

int main()
{
  MyWindow w;

  w.draw();

  return EXIT_SUCCESS;
}
```

Nie działa..., uruchomiły się funkcje **drawFrame** i **drawInterior** klasy **Window**, dlaczego?

```
Window's drawFrame()
Window's drawInterior()
```

Klasyczna kompilacja wywołania funkcji

Kompilacja

```
class Window
{ . . .
    void drawFrame()
    {
      cout << "Window's drawFrame()" << endl;
    }
    void drawInterior()
    {
      cout << "Window's drawInterior()" << endl;
    }
    void draw()
    {
      drawFrame();
      drawInterior();
    }
};

class MyWindow : public Window
{ . . .
    void drawFrame()
    {
      cout << "MyWindow's drawFrame()" << endl;
    }
    void drawInterior()
    {
      cout << "MyWindow's drawInterior()" << endl;
    }
};
```

- Kompilując funkcję **draw**() klasy **Window**, kompilator jeszcze *nic nie wie* o dziedziczeniu i generuje kod maszynowy, wywołujący funkcje **drawFrame**() i **drawInterior**() klasy **Window**.
- Klasa **MyWindow** dziedziczy tak skompilowany kod funkcji **draw**(), który wywołuje ***zawsze*** funkcje **drawFrame**() i **drawInterior**() klasy **Window**.

Klasyczna kompilacja wywołania funkcji

Kompilacja

```
class Window
{ . . .
    void drawFrame()
    {
      cout << "Window's drawFrame()" << endl;
```

Kompilując funkcję *draw*() klasy *Window*, kompilator jeszcze *nic nie wie* o dzie-

Chciejstwo:

Czy istnieje możliwość takiego skompilowania funkcji **draw**() klasy **Window**, aby wywoływała ona *odpowiednie* funkcje **drawFrame**() i **drawInterior**() w zależności od tego, dla jakiego obiektu została wywołana?

Chcemy zatem *wariantowości*:

- gdy funkcja **draw***() wywoływana jest dla obiektu klasy* **Window**, niech kompilator wywołuje funkcje **drawFrame**() i **drawInterior**() kl*asy* **Window***;*
- gdy **funkcja draw***() wywoływana jest dla obiektu klasy* **MyWindow**, niech kompilator wywołuje funkcje **drawFrame**() i **drawInterior**() kl*asy* **MyWindow**.

```
    }
};
```

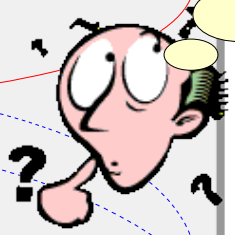
Window.

Klasyczna kompilacja wywołania funkcji

Kompilacja

```
class Window
{ . . .
    void drawFrame()
    {
      cout << "Window's drawFrame()" << endl;
    }
    void drawInterior()
    {
      cout << "Window's drawInterior()" << endl;
    }
    void draw()
    {
      drawFrame();
      drawInterior();
    }
};

class MyWindow : public Window
{ . . .
    void drawFrame()
    {
      cout << "MyWindow's drawFrame()" << endl;
    }
    void drawInterior()
    {
      cout << "MyWindow's drawInterior()" << endl;
    }
};
```

Skąd kompilator ma wiedzieć, że *czasem* chcemy wywołać funkcje klasy *Window* a *czasem MyWindow*?



- W przypadku klasycznego wywołania kompilator *musi* podjąć *jednoznaczną* decyzję, którą funkcję wywołać.
- Ta decyzja zapisywane jest w wynikowym kodzie maszynowym.

Rodzaje wiązania: *wywołanie ↔ konkretna funkcja*

Przypisanie konkretnej *funkcji* do *wywołania* nazywa się *wiązaniem* (ang. *binding*).

We współczesnych językach programowania występują dwa rodzaje wiązania:

- ***Wczesne wiązanie*** (ang. *early binding*) polega na przypisaniu każdemu *wywołaniu funkcji* jednoznacznie określonej, konkretnej funkcji już na etapie *kompilacji programu*. Inna nazwa — *wiązanie statyczne* (ang. *static binding*).
- ***Późne wiązanie*** (ang. *late binding*) polega na przypisaniu każdemu *wywołaniu funkcji* jednoznacznie określonej, konkretnej funkcji dopiero na etapie *wykonania programu*, w zależności od typu obiektu, którego wywołanie dotyczy. Inna nazwa — *dynamiczne wiązanie* (ang. *dynamic binding*).

Wczesne wiązanie jest charakterystyczne dla *języków kompilowanych*, realizacja późnego wiązania wymaga od kompilatora stosowania złożonych mechanizmów.

Późne wiązanie jest charakterystyczne dla *języków interpretowanych*, realizacja późnego wiązania jest zwykle dla interpretera łatwa.

Wiązanie statyczne i dynamiczne

- Wczesne wiązanie jest od lat stosowne we większości języków kompilowanych, występuje np. w językach *Pascal* czy *C*, w których późne wiązanie „oficjalnie” nie występuje.
- We większości nowoczesnych, obiektowych języków programowania, występuje możliwość wyboru sposobu wiązania – realizowane jest to poprzez różne słowa kluczowe (*virtual*, *override*, *final*, *overridable*, *overrides*).
- Domyślny sposób wiązania zależy od języka, np. *C++* i *C#* – domyślnie wiązanie wczesne a *Java, Ruby, Python* – domyślnie wiązanie późne.

Uwaga – na temat późnego wiązania pojawia się wiele *niespójnych informacji*.

Może ono być realizowane na *wiele sposobów*, w zależności od *języka programowania*, *trybu translacji* i *wsparcia* ze strony biblioteki „czasu wykonania”.

Niezależnie od przyjętej koncepcji realizacji, późne wiązanie jest zwykle *wolniejsze od wczesnego* – konieczne jest bowiem ustalenie w czasie działania programu, która wersja funkcji ma być wywołana.

Późne wiązania na przykładzie funkcji drawFrame

Kompilacja

```
class Window
{ . . .
    void drawFrame()
    {
      cout << "Window's drawFrame()" << endl;
    }

    void draw()
    {
      drawFrame();
      drawInterior();
    }

};

class MyWindow : public Window
{ . . .
    void drawFrame()
    {
      cout << "MyWindow's drawFrame()" << endl;
    }
};
```

```
Window w;
. . .
w.draw();
```

```
MyWindow w;
. . .
w.draw();
```

Jeżeli funkcja **drawFrame**() będzie wywoływana na zasadzie późnego wiązania to: dla obiektów klasy **Window** wywołana zostanie wersja **drawFrame**() klasy **Window** a dla obiektów klasy **MyWindow** wywołana zostanie wersja **drawFrame**() klasy **MyWindow**.

W jaki sposób uzyskać dynamiczne wiązanie w C++?

```
class Window
{
  public:
    Window()
    {
    }

    virtual void drawFrame()
    {
      cout << "Window's drawFrame()" << endl;
    }

    virtual void drawInterior()
    {
      cout << "Window's drawInterior()" << endl;
    }

    void draw()
    {
      drawFrame();
      drawInterior();
    }
};
```

To są *funkcje wirtualne*, kompilator C++ wygeneruje kod w taki sposób, aby konkretna wersja wywoływanej funkcji została ustalona w trakcie wykonania programu.

- Dynamiczne wiązanie w języku C++ uzyskuje poprzez wykorzystanie *funkcji wirtualnych*.
- Słowo kluczowe *virtual* w deklaracji funkcji oznacza, że *każde wywołanie* tej funkcji ma odbywać się na zasadzie *późnego wiązania*.
- Słowo *virtual* występuje w deklaracji (prototypie) funkcji.

W jaki sposób uzyskać dynamiczne wiązanie w C++?

```
class Window
{ . . .
    virtual void drawFrame()
    {
      cout << "Window's drawFrame()" << endl;
    }

    virtual void drawInterior()
    {
      cout << "Window's drawInterior()" << endl;
    }
  . . .
};

class MyWindow : public Window
{ . . .
    void drawFrame()
    {
      cout << "MyWindow's drawFrame()" << endl;
    }

    void drawInterior()
    {
      cout << "MyWindow's drawInterior()" << endl;
    }
};
```

Od tego momentu funkcje *drawFrame*() i *drawInterior*() są funkcjami wirtualnymi, podlegającymi późnemu wiązaniu.

Redefinicja *funkcji wirtualnej* w dowolnej klasie pochodnej jest *również funkcją wirtualną*, niezależnie czy słowo *virtual* wystąpi, czy nie.

Dzięki późnemu wiązaniu funkcja draw będzie działać tak jak chcemy

```
class Window
{ . . .
    virtual void drawFrame()
    {
      cout << "Window's drawFrame()" << endl;
    }
    virtual void drawInterior()
    {
      cout << "Window's drawInterior()" << endl;
    }

    void draw()
    {
      drawFrame();
      drawInterior();
    }
};

class MyWindow : public Window
{ . . .
    void drawFrame()
    {
      cout << "MyWindow's drawFrame()" << endl;
    }
    void drawInterior()
    {
      cout << "MyWindow's drawInterior()" << endl;
    }
};
```

```
Window's drawFrame()
Window's drawInterior()
```

```
Window w;
. . .
w.draw();
```

```
MyWindow w;
. . .
w.draw();
```

```
MyWindow's drawFrame()
MyWindow's drawInterior()
```

Funkcja przedefiniowana często wykorzystuje wersję oryginalną

- Często zdarza się, że funkcja wirtualna zdefiniowana w klasie pochodnej wywołuje wersję zdefiniowaną w klasie bazowej.
- Dzieje się tak wtedy, gdy działanie wersji bazowej nam odpowiada i chcemy w klasie pochodnej uzyskać dodatkowe działanie.

```
class MyWindow : public Window
{
  . . .

    void drawFrame()
    {
      Window::drawFrame();

      cout << "MyWindow's drawFrame()" << endl;
    }
  . . .
};
```

To działania nam odpowiada, ale nie jest wystarczające

Dodatkowe działanie w funkcji przedefiniowanej

Problem terminologiczny — overloading vs overriding

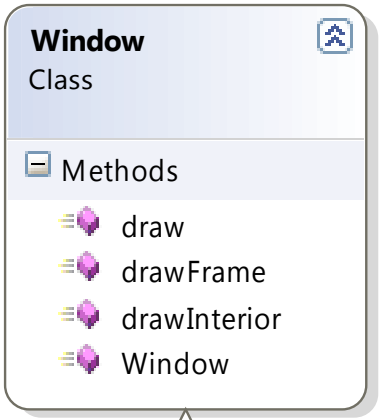
- **Przeciążenie** (ang. *overloading*) metody polegać będzie na zdefiniowaniu *nowej wersji* metody, o takiej *samej nazwie*, lecz *różniącej się parametrami* – ich liczbą i typami.

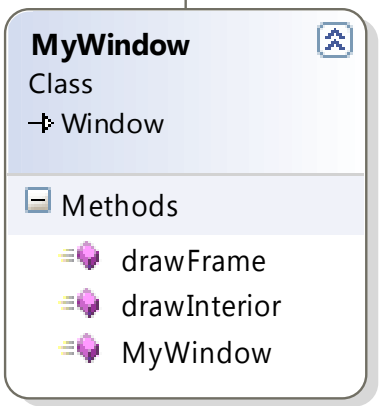
- **Redefinicja metody wirtualnej** (ang. *overriding*) polega na ponownym zdefiniowaniu metody wirtualnej w klasie pochodnej, co skutkuje „nadpisaniem" tej metody – aktywowana jest metoda „nadpisująca" wszędzie tam, gdzie odwołujemy się do obiektu klasy pochodnej, niezależnie od tego, czy odwołanie realizowane odniesieniem zgodnego typu, czy też typu bazowego.

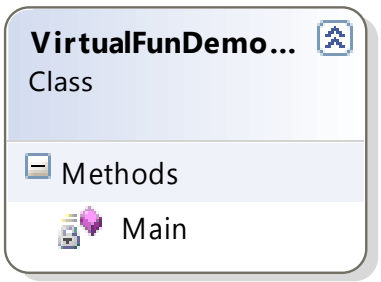
- **Redefinicja metody** (ang. *redefinition*) polega na ponownym zdefiniowaniu metody niewirtualnej w klasie pochodnej, co skutkuje *przesłonięciem* tej metody w klasie pochodnej – aktywowana jest metoda *przesłaniająca* wszędzie tam, gdzie odwołujemy się do obiektu klasy pochodnej zgodnie z jego właściwym typem. W innym przypadku uaktywniana jest wersja *przesłonięta*.

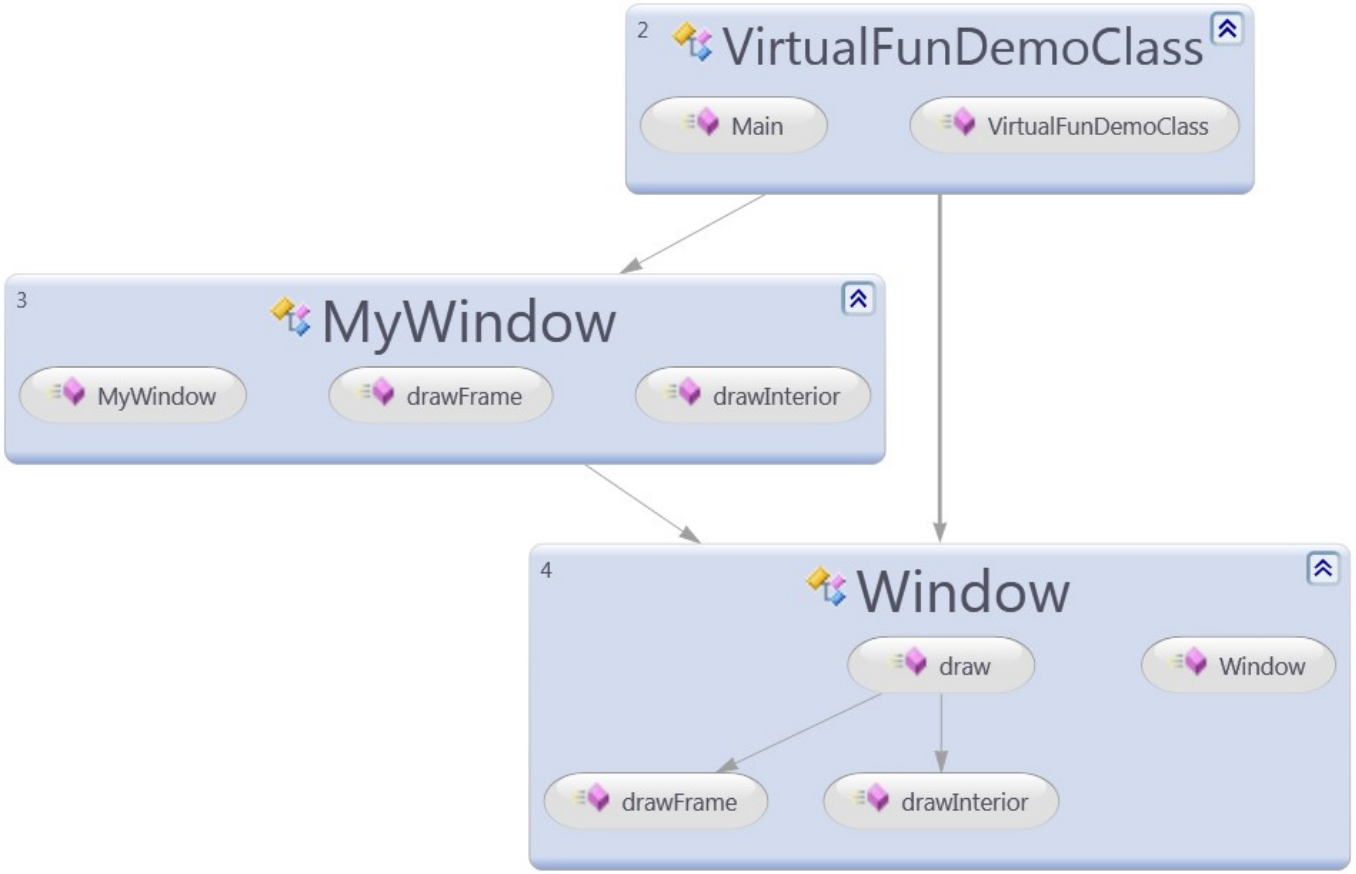
Wczesne i późne wiązanie
Funkcje wirtualne
Java

Przykład Java, hierarchia klas i powiązania między nimi

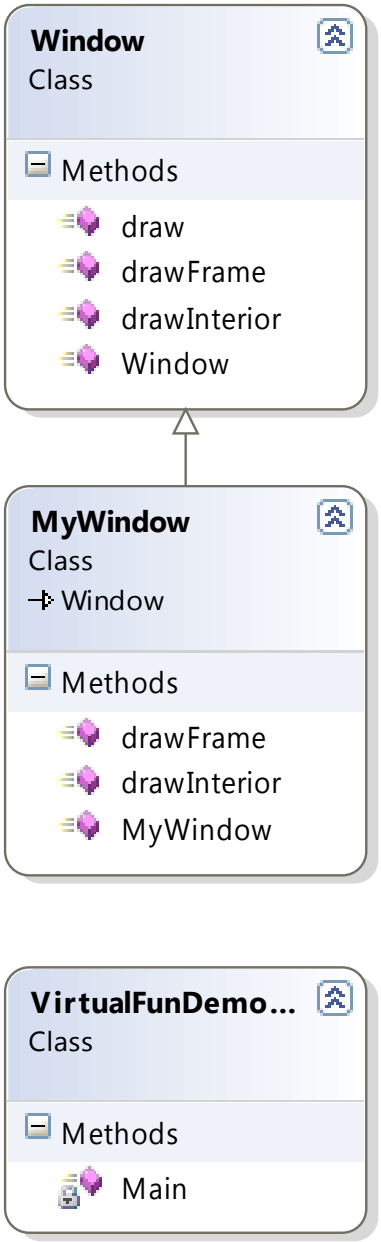








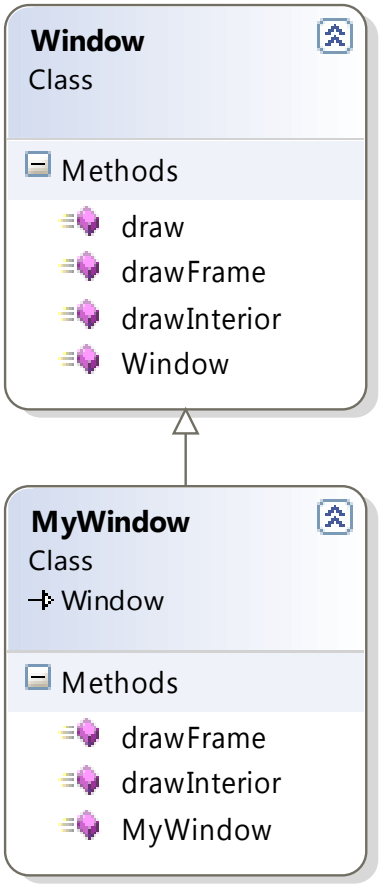
Przykład, hierarchia klas Java — klasa Window



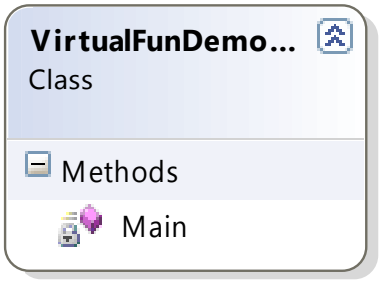

```
class Window
{
    public Window()
    {
    }

    public void drawFrame()
    {
        System.out.println( "Window's frame" );
    }

    public void drawInterior()
    {
        System.out.println( "Window's interior" );
    }

    public void draw()
    {
        drawFrame();
        drawInterior();
    }
}
```

- Późne wiązanie w Javie jest domyślne, nie trzeba użyć słowa kluczowego **virtual**.

Przykład, hierarchia klas Java — klasa MyWindow

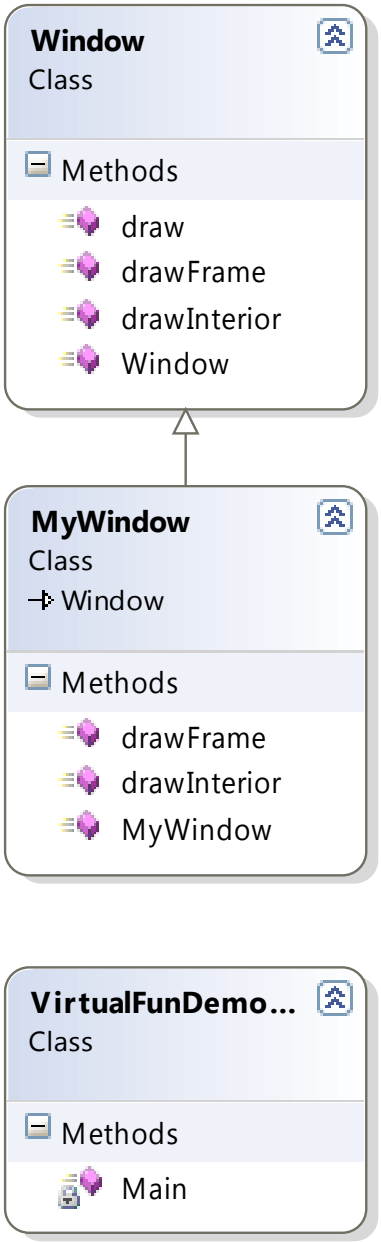



```
class MyWindow extends Window
{
    public MyWindow()
    {
        super();
    }

    public void drawFrame()
    {
        System.out.println( "MyWindow's frame" );
    }

    public void drawInterior()
    {
        System.out.println( "MyWindow's interior" );
    }
}
```



- *Redefinicja metody wirtualnej* w języku Java nie wymaga jawnego użycia żadnego słowa kluczowego.
- Domyślnie każda *nie statyczna* i *nie prywatna* metoda jest wirtualna.

Przykład, hierarchia klas Java — klasa opakowująca funkcję Main




```
public class VirtualFunDemo
{
    public static void main( String[] args )
    {
        Window w = new MyWindow();

        w.draw();
    }
}
```

```
MyWindow's frame
MyWindow's interior
```

- Późne wiązanie *działa* — wywoływane są funkcje właściwe dla utworzonego obiektu **MyWindow** a nie wynikające z typu **Window** zmiennej referencyjnej **w**.
- W języku Java programista domyślnie otrzymuje dynamiczne wiązanie.

Zablokowanie możliwości redefinicji w klasach pochodnych — final

```
class Window
{
    public Window()
    {
    }

    public final void drawFrame()
    {
        System.out.println( "Window's frame" );
    }

    public void drawInterior()
    {
        System.out.prin

    }

    public void draw()
    {
        drawFrame();
        drawInterior();
    }
}
```

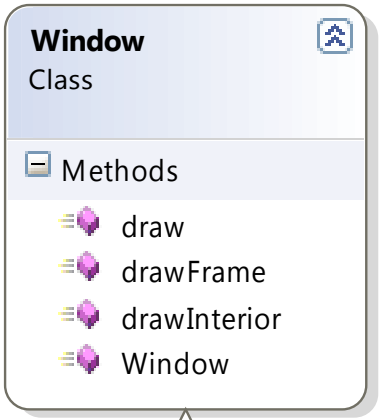
Ta metoda nie może być redefiniowana

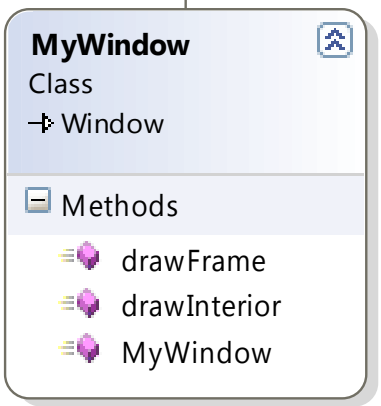
Błędna próba redefinicji

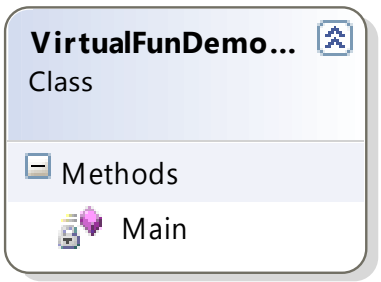
```
class MyWindow extends Window
{
    ...

    public void drawFrame()
    {
        System.out.println( "MyWindow's frame" );
    }

    ...
}
```

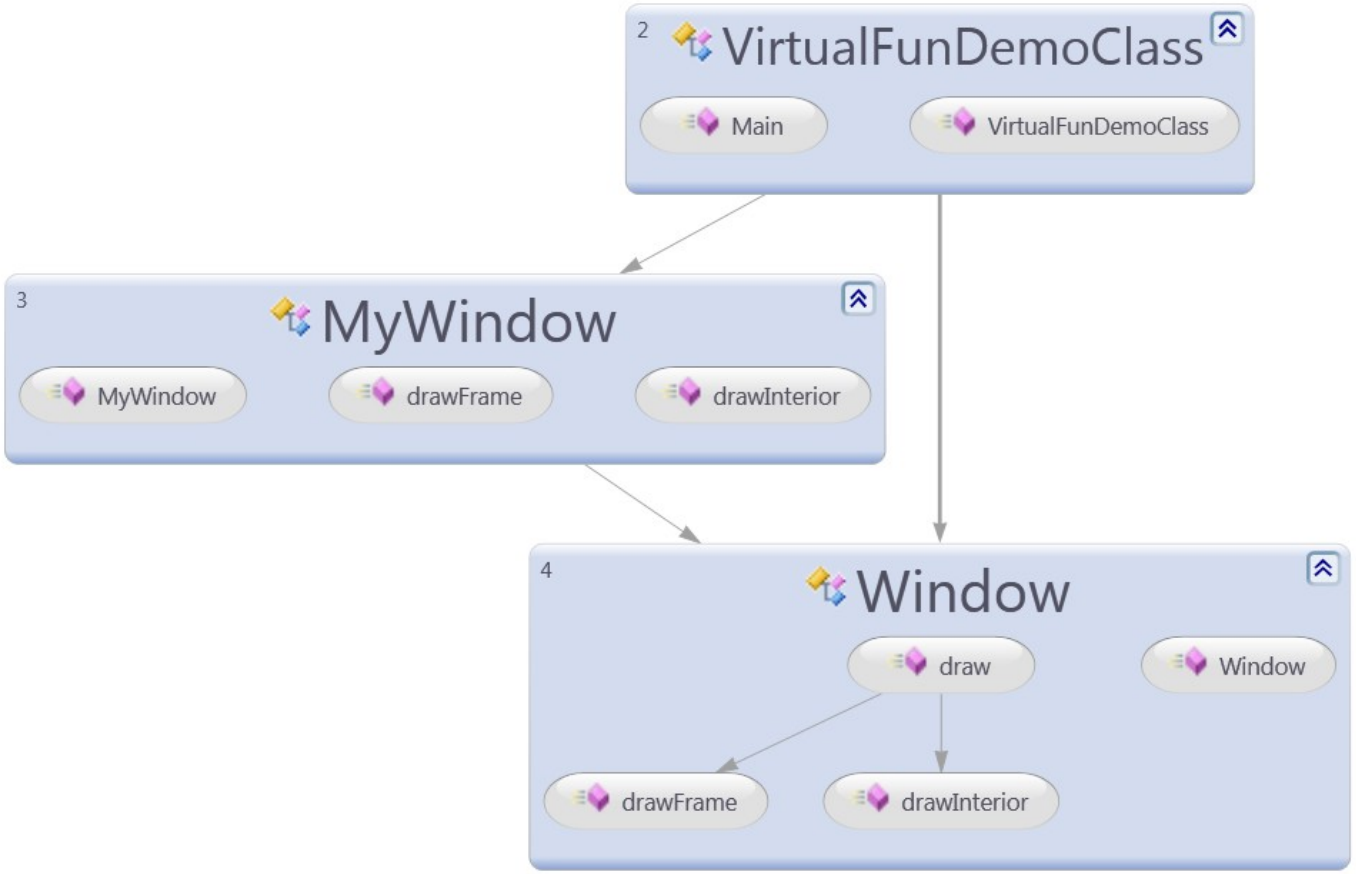
Wczesne i późne wiązanie
Funkcje wirtualne
C#

Przykład C#, hierarchia klas i powiązania między nimi

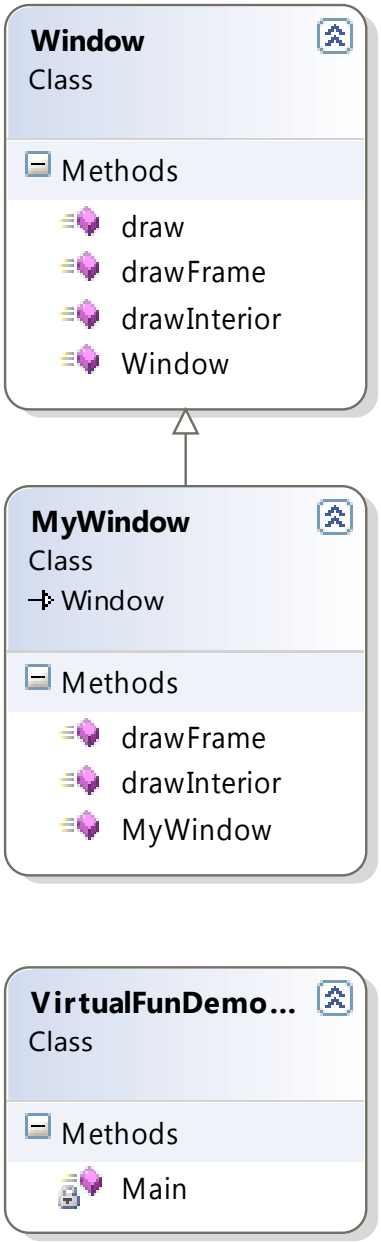








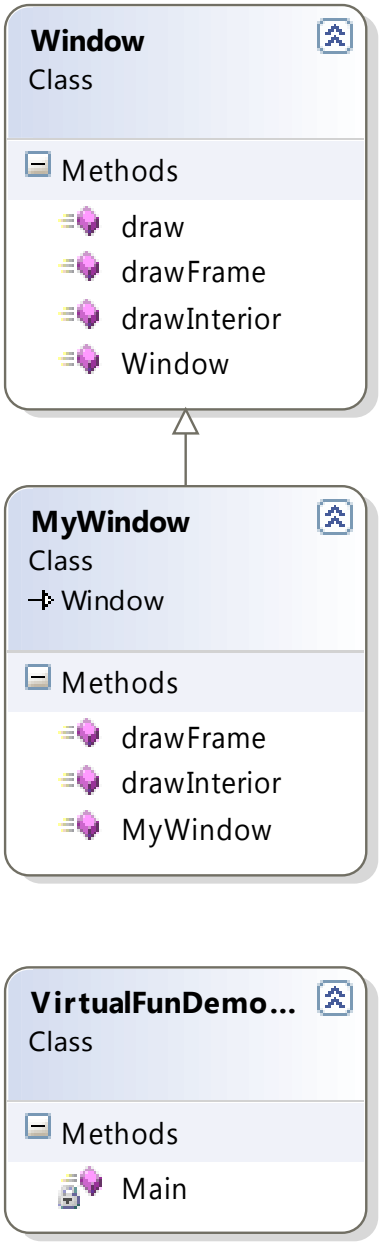
Przykład, hierarchia klas C# — klasa Window




```
class Window
{
    public Window()
    {
    }

    public virtual void drawFrame()
    {
        Console.WriteLine( "Window's drawFrame()" );
    }

    public virtual void drawInterior()
    {
        Console.WriteLine( "Window's drawInterior()" );
    }

    public void draw()
    {
        drawFrame();
        drawInterior();
    }
}
```

- Późne wiązanie w C# nie jest domyślne, należy użyć słowa kluczowego **virtual**.

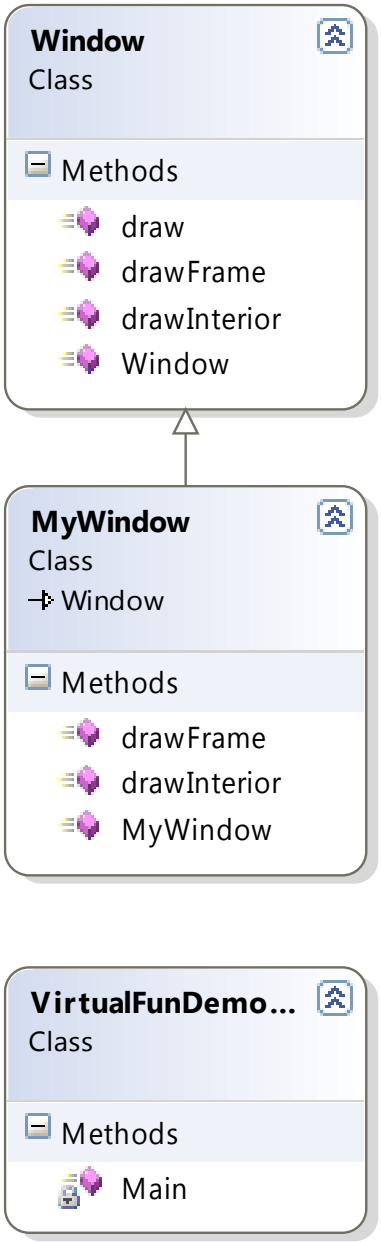
Przykład C#, hierarchia klas — klasa MyWindow




```
class MyWindow : Window
{
    public MyWindow() : base()
    {
    }

    public override void drawFrame()
    {
        Console.WriteLine( "MyWindow's drawFrame()" );
    }

    public override void drawInterior()
    {
        Console.WriteLine( "MyWindow's drawInterior()" );
    }
}
```

- *Redefinicja metody wirtualnej* w języku C# wymaga jawnego użycia słowa kluczowego **override**.

Przykład C#, hierarchia klas — klasa opakowująca funkcję Main




```
class VirtualFunDemoClass
{
    static void Main( string[] args )
    {
        Window w = new MyWindow();

        w.draw();

        Console.ReadKey();
    }
}
```

```
MyWindow's drawFrame()
MyWindow's drawInterior()
```

- Późne wiązanie *działa* – wywoływane są funkcje właściwe dla utworzonego obiektu **MyWindow** a nie wynikające z typu **Window** zmiennej referencyjnej **w**.
- W języku C# programista może *wybrać* rodzaj wiązania.

Odwołanie do przedefiniowanej metody wirtualnej

```
class MyWindow
{
    . . .
    public override void drawFrame()
    {
      base.drawFrame();

      Console.WriteLine( "MyWindow's drawFrame()" );
    }
    . . .
}
```

Odwołanie do przedefiniowanej metody wirtualnej

Przykład C#, gdyby nie było *virtual* i *override*

```
class Window
{
    public Window()
    {
    }

    public void drawFrame()
    {
        Console.WriteLine( "Window's drawFrame()" );
    }

    public void drawInterior()
    {
        Console.WriteLine( "Window's drawInterior()" );
    }

    public void draw()
    {
        drawFrame();
        drawInterior();
    }
}

class MyWindow : Window
{
   public MyWindow() : base()
   {
   }

   public void drawFrame()
   {
      Console.WriteLine( "MyWindow's drawFrame()" );
   }

   public void drawInterior()
   {
      Console.WriteLine( "MyWindow's drawInterior()" );
   }
}
```

```
Window's drawFrame()
Window's drawInterior()
```

- Późne wiązanie *nie działa* – wywoływane są funkcje odpowiednie dla klasy **Window** zmiennej referencyjnej **w**.
- Kompilator języka C# generuje *ostrzeżenie* o redefinicji niewirtualnych składowych, aby tego uniknąć należy użyć słowa kluczowego **new**:

```
class MyWindow : Window
{
   . . .
   new public void drawFrame()
   {
      Console.WriteLine( "MyWindow's drawFrame()" );
   }

   new public void drawInterior()
   {
      Console.WriteLine( "MyWindow's drawInterior()" );
   }
}
```

Metody *virtual*, *override*, *sealed*

```
class Window
{
    public virtual void drawFrame()
    {
      . . .
    }
}

class MyWindow : Window
{
    public override void drawFrame()
    {
      . . .
    }
}

class TerminalWindow : Window
{
    public sealed override void drawFrame()
    {
      . . .
    }
}
```

Słowo kluczowe **virtual** uruchamia *dynamiczne wiązanie* i zezwolenie na potencjalne *redefinicję*

Słowo kluczowe **override** oznacza zamierzone *redefinicję*, które może być kontynuowane w dalszych podklasach

Słowo kluczowe **sealed** oznacza *ostatnią redefinicję*, nikt już tej metody *nie przedefiniuje*

Metody *virtual* a *virtual* i *new*

```
class Window
{
    public virtual void drawFrame()
    {
      . . .
    }
}

class MyWindow : Window
{
    public virtual void drawFrame()
    {
      . . .
    }
}

class OtherWindow : Window
{
    public new void drawFrame()
    {
      . . .
    }
}
```

Słowo kluczowe **virtual** uruchamia *dynamiczne wiązanie* i zezwolenie na potencjalną *redefinicję*

Ostrzeżenie – to nie jest przedefiniowanie metody a raczej ukrycie wersji bazowej w podklasie

Niejednoznaczne

To jest zdefiniowanie nowej wersji metody, ukrywającej wersję wirtualną (statyczne wiązanie)

Metody niewirtualne i *new*

```
class Window
{
    public void drawFrame()
    {
      . . .
    }
}

class MyWindow : Window
{
    public new void drawFrame()
    {
      . . .
    }
}
```

Brak słowa kluczowego **virtual** oznacza *statyczne wiązanie*

Słowo kluczowe **new** oznacza zdefiniowanie nowej wersji metody

Metody abstrakcyjne

- Metody *abstrakcyjne* są z definicji *wirtualne*, jednak *nie mogą* posiadać słowa kluczowego *virtual*.
- Deklaracja metody abstrakcyjnej wprowadza *nową metodę*, jednak bez jej *implementacji*.
- Każda nieabstrakcyjna klasa pochodna *musi* posiadać *implementację* metody.

```
abstract class AbstractWindow
{
    public abstract void drawFrame();
}

class MyWindow : AbstractWindow
{
    public override void drawFrame()
    {
        . . .
    }
}
```

Metoda abstrakcyjna, brak implementacji

Wymagana definicja metody abstrakcyjnej