Opracowanie Obrona - Informatyka przemysłowa

**Autor:** Michał Kapiczyński

# 1. Cechy charakterystyczne i zastosowania programowania obiektowego

**Programowanie obiektowe** - jest jednym z paradygmatów programowania, czyli sposobu patrzenia programisty na przepływ sterowania i wykonywania programu komputerowego. Zgodnie z nim programy definiuje się za pomocą obiektów, komunikujących się między sobą w celu wykonania określonego zadania.Obecnie jest to najpopularniejsza technika programistyczna.

Programowanie obiektowe jest potoczną, ogólnie przyjętą nazwą programowania zorientowanego obiektów.

Istnieje jeszcze:

**Object based design -** programowanie oparte na obiektach - paradygmat, który wykorzystuje jedynie pojęcie obiektu i związaną z tym hermetyzację kodu i ochronę pól,

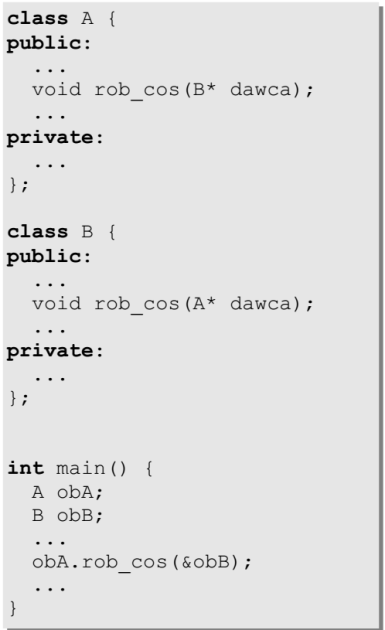
**Object orientated programming -**  programowanie zorientowane obiektowo - tutaj mamy do czynienia z czymś więcej - tworzenie hierarchii klas i obiektów, definicja ich wzajemnych zależności, zmiana zachowania klasy w zależności od typów itd.

**Obiekt -** materialny lub abstrakcyjny byt, który można wyizolować ze środowiska,

**Obiekt (bardziej programistycznie** - struktura danych, która występuje łącznie z operacjami dozwolonymi do wykonania na niej, Obiekt może być złożony tzn. składać się z innych obiektów.

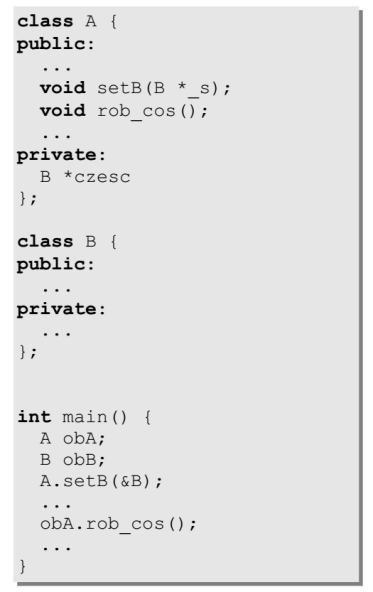
**Obiekt jest charakteryzowany poprzez:**

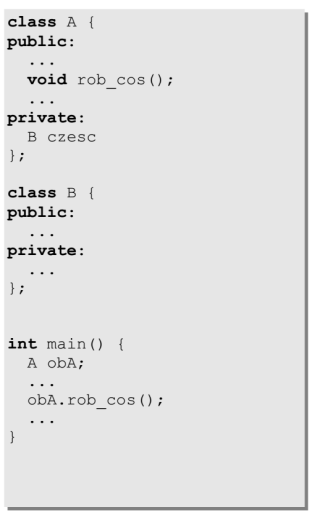
* tożsamość, która odróżnia go od innych obiektów (miejsce zajmowane przez ten obiekt w pamięci komputera),
* stan, który może zmieniać się w czasie (bez zmiany tożsamości obiektu). Stan obiektu w danym momencie jest określany przez aktualne wartości atrybutów i powiązań z innymi obiektami.
* zachowania do niego przypisane, tj. zestaw operacji, które wolno stosować na danym obiekcie,

**Atrybuty obiektów mogą być:**

* proste,
* złożone,
* wskaźnikowe,
* powtarzalne,
* opcjonalne,
* domyślne,
* pochodne (wyliczane),
* wspólne dla zestawu obiektów,
* inne obiekty.

**Relacje między obiektami:**

****

* **Skojarzenie -** obiekt typu A wykorzystuje obiekt typu B i/lub B wykorzystuje A w celu wykonania swoich zadań. A i B są tworzone i pamiętane zupełnie niezależnie.
* **Agregacja** - obiekt typu A zawiera w sobie obiekt typu B, ale oba są tworzone niezależnie od siebie. Obiekt typu A może składać się z wielu obiektów innych typów.
*  **Kompozycja** - obiekt typu A zawiera w sobie obiekt typu B i jest jego panem i władcą tzn. A zarządza czasem życia obiektu typu B - A tworzy i niszczy B. Obiekt B nie może istnieć (nie ma racjonalnego wyjaśnienia takiej możliwości) bez obiektu A.
* **Uogólnienie -** obiekt typu B dziedziczy po obiekcie typu A. A jest klasą nadrzędną, B - podrzędną.

**Przykład obiektu:** Samochód:

Jego atrybutami mogą być:

* pozycja,
* prędkość,
* stan paliwa,
* części składowe np. silnik, koła,

Czynności, które może wykonywać:

* start silnika,
* przemieszczanie się,
* przyspieszanie,
* hamowanie,
* ...

**Klasa**  to szablon obiektu:

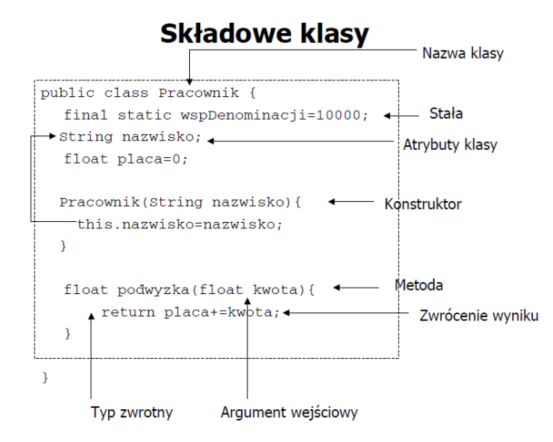
* Jest miejscem przechowywania (definiowania) tych własności grupy obiektów, które są niezmienne (co do definicji, a nie wartości) dla wszystkich członków grupy,
* Ogranicza kontekst, w którym odwołania do obiektu może być użyte w programie,
* Dobrze zbudowana klasa jest starannie wydzieloną abstrakcją pochodzącą ze słownictwa dziedziny danego problemu.
* Obejmuje pewien mały, dobrze określony zbiór zobowiązań, z których jest w stanie się w pełni wywiązać. Zapewnia oddzielenie specyfikacji abstrakcji od jej implementacji. Jest zrozumiała i prosta, a przy tym rozszerzalna i dająca się łatwo dostosować do potrzeb.

Struktura klasy:

* pola - atrybuty (różne wartości dla różnych instancji),
* metody - (takie same dla wszystkich).

Ochrona danych - definiowanie obszaru widoczności zmiennej lub funkcji:

* publiczne (public) - dostępne z dowolnego miejsca w programie,
* prywatne (private) - dostępne jedynie z wnętrza danej klasy,
* chronione (protected) - dostępne z wnętrza danej klasy i klas pochodnych.



**Podstawowe zasady programowania obiektowego:**

* Zasada dekompozycji (metoda walki ze złożonością) **-** rozdzielenie złożonego problemu na podproblemy, które można rozpatrywać i rozwiązywać niezależnie od siebie i od całości,
* Zasada abstrakcji (metoda walki ze złożonością) - budowa abstrakcyjnych struktur i operowanie na nich bez wnikania w ich wewnętrzną strukturę. Eliminacja, ukrycie lub pominięcie mniej istotnych szczegółów rozważanego przedmiotu lub mniej istotnej informacji. Wyodrębnienie cech wspólnych i niezmiennych dla pewnego zbioru bytów i wprowadzenie pojęć lub symboli oznaczających takie cechy,
* Zasada ponownego użycia - wykorzystanie wcześniej wytworzonych schematów, metod, wzorców, bibliotek, komponentów projektu itd.
* Zasada sprzyjania naturalnym ludzkim własnościom (cel nadrzędny obiektowej analizy) - dopasowanie modeli pojęciowych i modeli realizacyjnych systemów do wrodzonych ludzkich własności psychologicznych, instynktów oraz mentalnych mechanizmów percepcji i rozumienia świata.

**Cechy paradygmatu obiektowego:**

* Hermetyzacja

Ukrywanie implementacji, **enkapsulacja**. Zamknięcie pewnego zestawu bytów programistycznych w "kapsułę" o dobrze określonych granicach. Oddzielenie abstrakcyjnej specyfikacji tej kapsuły (obiektu, klasy, moduły etc.) od jej implementacji, ukrycie części informacji zawartej w tej kapsule dla operacji z zewnątrz obiektu. Hermetyzacja jest podstawową techniką abstrakcji tj. ukrycia wszelkich szczegółów danego przedmiotu lub bytu programistycznego, które na danym etapie rozpatrywania (analizy, projektowania, programowania) nie są istotne.

Zapewnia, że obiekt nie może zmieniać stanu wewnętrznego innych obiektów w nieoczekiwany sposób. Każdy typ obiektu prezentuje innym obiektom swój interfejs, które określa dopuszczalne metody współpracy.

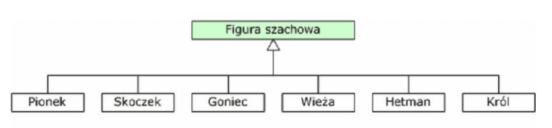
Pewne języki osłabiają to założenie, dopuszczając pewien poziom bezpośredniego (kontrolowanego) dostępu do wnetrza obiektu, ograniczając w ten sposób poziom abstrakcji. Przykładowo w niektórych kompilatorach C++ istnieje możliwość tymczasowego wyłączenia mechanizmu enkapsulacji i otwarcia w ten sposób dostępu do wszystkich pól i metody prywatnych.

* Dziedziczenie

To związek między klasami obiektów określający przekazywanie cech (definicji atrybutów, metod) z nadklasy do jej podklasy. Porządkuje i wspomaga polimorfizm i enkapsulację dzięki umożliwieniu definiowania i tworzenia specjalizowanych obiektów na podstawie bardziej ogólnych. Dla obiektów specjalizowanych nie trzeba redefiniować całej funkcjonalności tylko tę, której nie ma obiekt ogólny.

W typowym przypadku powstają grupy obiektów zwane klasami oraz grupy klas zwane drzewami. Odzwierciedlają one wspolne cechy obiektów.

Istnieje wiele rodzajów dziedziczenia np.: dziedziczenie oparte na klasach, na prototypach, wielokrotne dziedziczenie.



* Abstrakcja

Każdy obiekt w systemie służy jako model abstrakcyjnego "wykonawcy", które może wykonywać pracę, opisywać i zmieniać swój stan oraz komunikować się z innymi obiektami w systemie bez ujawniania, w jaki sposób zaimplementowano dane cechy. Ograniczenie zakresu cech manipulowanych obiektów wyłącznie do cech kluczowych dla algorytmu, a jednocześnie niezależnych od implementacji.

* Polimorfizm

Zdolność obiektów do różnych zachować w zależności od bieżącego kontekstu wykonania programu. Wywołane mogą być różne wersje tej samej funkcji. Pozwala to na rozszerzalność i łatwą modyfikację programu.

**Zastosowania programowania obiektowego:**

* Inżyniera oprogramowania - w języku modelowania UML do analizy oraz modelowania rzeczywistości. Podejście obiektowe ułatwia zrozumienie przez człowieka złożonych bytów informatycznych.
* Języki programowania - na programowaniu obiektowym bazują całe języki programowania takie jak np. język JAVA
* Komunikacja z relacyjnymi bazami danych - ORM (Object Rational Mapping) - sposób odwzorowania obiektów bazodanowych o relacyjnym charakterze poprzez obiektową architekturę systemu informatycznego,
* Biblioteki - programowanie obiektowe ułatwia pisanie reużywalnych bibliotek programistycznych, z których programista ma możliwość korzystania poprzez wystawiony interfejs bez wnikania w szczegóły implementacyjne,
* Komunikacja sieciowa - podejście obiektowe ułatwia komunikację sieciową między programami napisanymi w różnych językach programowania. Przykładem jest ujednolicony format przesyłania danych JSON oparty na podejściu obiektowym.
* Graficzne środowiska tworzenia oprogramowania - RAD (Rapid Application Development) - szybkie tworzenie aplikacji. Jest to ideologia i technologia polegająca na udostępnianiu programiście zestawu gotowych komponentów (obiektów), z których ten ma możliwość tworzenia złożonych programów.
* Współdziałanie systemów heterogenicznych - Przykładem zastosowania jest technologia **CORBA** (Common Object Request Broker Architecture) zapewniająca komunikację między obiektami pracującymi w heterogenicznych (różnorodnych) systemach komputerowych. Obiekty pełniące dowolne funkcje mogą być zaimplementowane w róznych językach programowania, na dowolnej platformie sprzętowej pod kontrolą różnych systemów operacyjnych.

Opis obiektów, a właściwie ich interfejsów znajduję się w specjalnym pliku IDL (Interface Definition Language), które jest kompilowany na kod zajmujący się kontrolą komunikacji w systemie.

Obiekty mają swoje adresy IOR (Interpretable Object Reference) będące kilkusetznakowymi adresami kodującymi wiele informacji m.in. o adresie komputera, programu, nr obiektu, informacje o kolejności zapisu bajów itd.

# Podział programowania obiektowego:

* oparte na klasach - definiowane są klasy, czyli typy zmiennych, a następnie tworzone są obiekty, czyli instancje tych typów
* oparte na prototypach - W tym podejściu nie istnieje pojęcie klasy. Nowe obiekty tworzy się w oparciu o istniejący już obiekt - prototyp, po którym dziedziczone są pola i metody. Dodatkowo można go rozszerzać o nowe. Typ ten spotykany w językach interpretowanych np. JavaScript.

# 2. Mechanizm dziedziczenia, ochrona pól - przykład implementacji

**Dziedziczenie** - to w programowaniu obiektowym mechanizm polegający na współdzieleniu funkcjonalności między klasami. Klasa może dziedziczyć po innej klasie, co oznacza, że oprócz swoich własnych atrybutów oraz zachowań, uzyskuje ("dziedziczy") także te pochodzące z klasy, po której dziedziczy.

Klasa dziedzicząca nazywana jest **klasą pochodną / potomną.**

Klasa, z której następuje dziedziczenie **klasą bazową.**

Klasy pochodne obok swoich własnych pól i metod posiadają również kompletny interfejs klasy bazowej.

W językach programowania z prototypowaniem (np. JavaScript) nie występuje pojęcie klasy, dlatego dziedziczenie zachodzi tam pomiędzy poszczególnymi obiektami.

Każdy obiekt o typie klasy pochodnej jest jednocześnie obiektem o typie klasy bazowej. Innymi słowy w C++ jeżeli mamy dwie klasy Figura i Elipsa, przy czym Elipsa dziedziczy po klasie Figura to wszędzie tam, gdzie spodziewamy się Figury geometrycznej lub wskaźnika do Figury możemy wstawić Elipsę.

W C++ dziedziczenie jest realizowane jako relacja zawierania. Innymi słowy jeśli zadeklarujemy, że klasa B dziedziczy po klasie A, to wówczas wewnętrznie będzie to zaimplementowane w ten sposób, że obiekt klasy B będzie w sobie zawierał obiekt klasy A. Przy czym jest to wewnętrzny mechanizm języka C++.

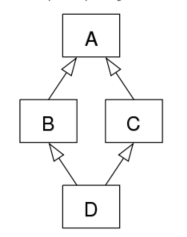
**Kolejność tworzenia obiektów klas bazowych i pochodnych:**

Generalnie podczas alokacji pamięci na obiekt klasy pochodnej najpierw alokowana jest pamięć na obiekt klasy bazowej, a dopiero później na obiekt klasy pochodnej. Dealokacja przebiega odwrotnie.

**Wielodziedziczenie -** nie wspominać, ale dobrze wiedzieć

Mechanizm umożliwiający klasie pochodnej dziedziczenie po więcej niż jednej klasie bazowej. Jest to cecha specyficzna danego języka np. w C++ wielodziedziczenie jest możliwe, a w Javie nie, aczkolwiek z Javą 8 wchodzi możliwość implementowania metod w interfejsach.

Z wielodziedziczeniem związany jest problem diamentu:



Klasa B i C dziedziczą po klasie A, czyli każda z nich ma swoją własną kopię klasy B. Jeśli klasa D dziedziczy zarówno po klasie B jak i C i na obiekcie klasy D spróbujemy wywołać metodę klasy A kompilator nie będzie wiedział, którą kopię metody wywołać (z tę z klasy B, czy z klasy C?).

**Rodzaje dziedziczenia**  - też nie wspominać, ale dobrze wiedzieć

W języku C++ wyróżnia się trzy rodzaje dziedziczenie: prywatne, chronione i publiczne. Jednak dziedziczenie prywatne i chronione są stosowane bardzo rzadko i jest to cecha specyficzna języka C++ (np. w Javie dziedziczenie prywatne i chronione nie istnieje).

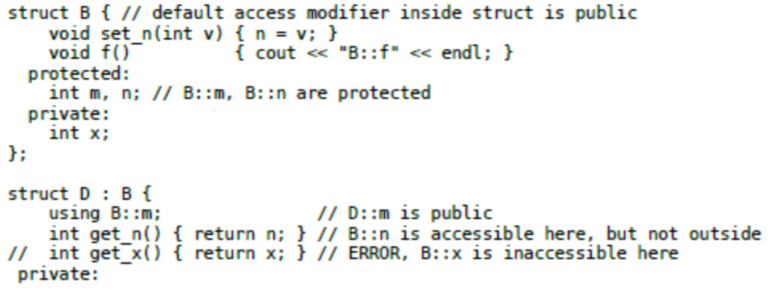
**Ochrona pól:**

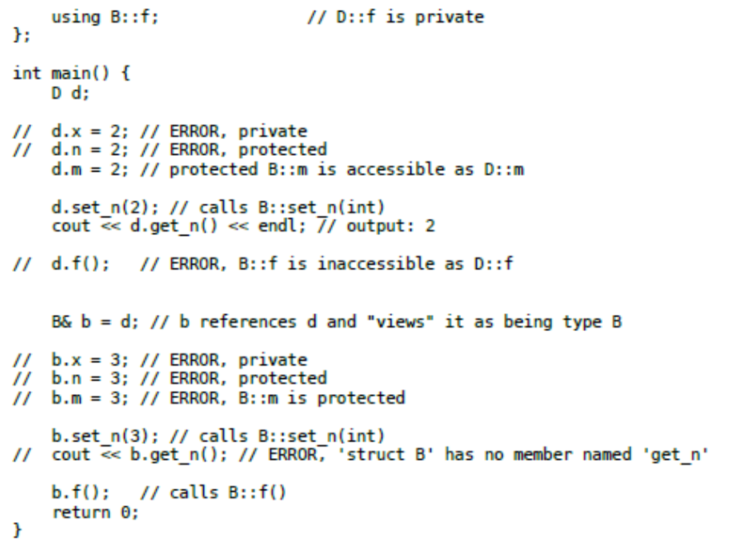
Jest to mechanizm umożliwiający programiście określanie, do których pól i metod będzie można odwołać sie spoza danej klasy. Zazwyczaj jest to realizowane za pomocą tzw. modyfikatorów dostępu:

* private - do pól i metod objętych tym modyfikatorem można odwołać się tylko i wyłącznie z metod tej klasy oraz z funkcji i klas zaprzyjaźnionych z tą klasą,
* protected - podobnie jak private, ale do pól i metod oznaczonych tym modyfikatorem mogą odwoływać się również klasy pochodne,
* public - pola i metody oznaczone tym modyfikatorem są dostępne w całym programie.

W teorii w programowaniu obiektowym w ogóle nie ma mowy o czymś takim jak pola publiczne (publiczne mogą być tylko metody), jednak większość języków (C++, Java) umożliwia programiście deklarację pól publicznych.

Przykład ochrony pól:





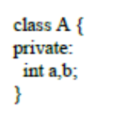
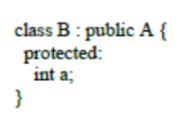
**Dziedziczenie, a zmiany modyfikatorów dostępu.**

Klasa pochodna dziedziczy wszystkie pola i modyfikatory klasy bazowej niezależnie od ich modyfikatorów dostępu, jednak:

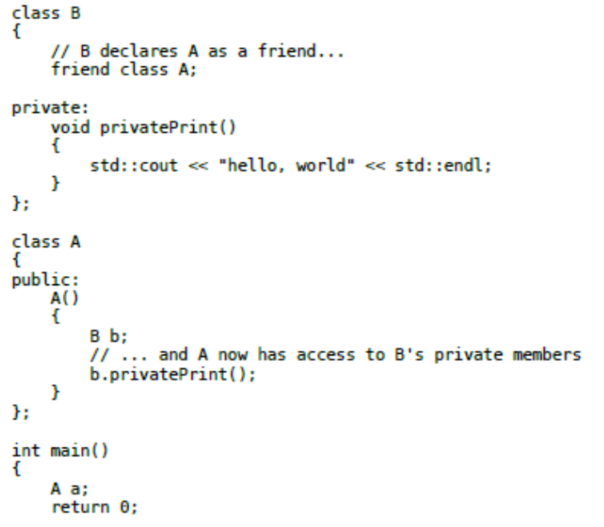
* zmienne i pola zadeklarowane w klasie bazowej jako **publiczne** będą publiczne w klasie pochodnej,
* zmienne i pola zadeklarowane w klasie bazowej jako **chronione**  będą chronione w klasie pochodnej i klasa pochodna będzie miała do nich dostęp,
* zmienne i pola zadeklarowane w klasie bazowej jako **prywatne** zostaną odziedziczone (tak samo jak wszystkie inne), jednak nie będzie do nich bezpośredniego dostępu. Można się do nich dostać tylko przez metody dostępowe odziedziczone z klasy bazowej.

Uwaga Wnuka:

Przyjmijmy, że mamy następujące dwie klasy:

W klasie bazowej i klasie pochodnej mamy zadeklarowaną zmienną **a** o tej samej nazwie, ale o innych modyfikatorach dostępu. Haczyk polega na tym, że ktoś może widząc takie coś pomyśleć, że umieszczona w klasie B deklaracja zmiennej chronionej **a** zmienia modyfikator dostępu zmiennej a odziedziczonej po klasie A. Nic takiego nie ma miejsca.

Dziedziczenie jest realizowane jako zawieranie, w związku z tym w klasie B znajdą się wszystkie zmienne klasy bazowej A wraz z modyfikatorami dostępu zgodnymi z zzasadami dziedziczenia modyfikatorów oraz dodatkowo w klasie B znajdzie się drugie pole o nazwie **a**, ale o modyfikatorze protected. Będziemy więc mieli jedno pole **a** typu int odziedziczone po klasie bazowej, do którego nie będziemy mieli bezpośredniego dostępu oraz drugie pole **a** o modyfikatorze dostępu protected.

**Klasy zaprzyjaźnione** (dodatek dobrze wiedzieć, że coś takiego istnieje)

W tym wypadku obiekt klasy B może wywołac prywatną metodę klasy A.

# 3. Polimorfizm - sposób działania, opisać na przykładzie

**Polimorfizm** jest pojęciem ściśle związanym z programowaniem obiektowym oraz dziedziczeniem. Jest to mechanizm pozwalający na definiowanie jednej metody w wielu postaciach oraz uzależnianie jej działania od typu obiektu, dla którego jest wywoływana.

Polimorfizm możemy określić jako wirtualizację operacji. Jest to możliwość dynamicznego (późnego, realizowanego w fazie wykonania) wiązania nazwy operacji do wielu implementacji (metod) tej operacji w różnych klasach pozostających w relacji dziedziczenia. Wiązaniu towarzyszy mechanizm wyboru konkretnej implementacji. Wybór implementacji zależy od nazwy metody oraz od typu dynamicznego tego obiektu, dla którego została wywołana operacja, a nie od typu zmiennej, wskazującej ten obiekt.

**Co nam daje?**

Dzięki niemu mamy pełną kontrolę nad wykonywanym programem, nie tylko w momencie kompilacji (**wiązanie statyczne**) ale także podczas działania programu (**wiązanie dynamiczne**) – niezależnie od różnych wyborów użytkownika.

Obiekt klasy pochodnej może być wskazywany przez wskaźnik typu klasy bazowej.

**Typem statycznym** obiektu wskazywanego przez wskaźnik jest typ tego wskaźnik, a **typem dynamicznym** obiektu wskazywanego przez wskaźnik jest typ na jaki dany wskaźnik wskazuje.

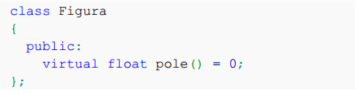
Mechanizm polimorfizmu jest możliwy dzięki metodom wirtualnym.

**Metoda wirtualna** jest to funkcja składowa klasy poprzedzona słowem kluczowym **virtual**, której sposób wywołania zależy od typu dynamicznego wskaźnika, a nie od typu statycznego.

**Metoda wirtualna** to taka metoda, której ciało może zostać przykryte w klasach dziedziczących po klasie bazowej. W ten sposób możemy zdefiniować zupełnie inne zachowania klas dziedziczących po klasie bazowej.

Specyficznym przykładem metody wirtualnej jest metoda czysto wirtualna, która w ogóle nie posiada ciała. Klasa posiadająca taką metodę staje się klasą abstrakcyjną. Oznacza to tyle, iż nie jest możliwe stworzenie obiektu tej klasy. Klasa taka służy jedynie temu, aby zdefiniować pewnego rodzaju interfejs i jest przeznaczona jedynie po to, by po niej dziedziczyć.

W przykładzie poniżej o ile mogą istnieć konkretne figury będące kwadratami, kołami itp. to nie powinien istnieć żaden obiekt klasy Figura ponieważ jest to jedynie abstrakcyjny byt. Natomiast dziedziczenie po tej klasie i jej rozszerzanie powoduje, że stworzymy już konkretną figurę geometryczną. Metodę czysto wirtualną w języku C++ deklaruje się tak:



Taka deklaracja metody wirtualnej uniemożliwia stworzenie jakiegokolwiek obiektu klasy *Figura* oraz zmusza do określenia metody *float pole()* na jednym z poziomów z dziedziczenia. Nie jest możliwe pominięcie takiej implementacji.

Metody wirtualne:

* Definiowane przy wykorzystaniu słowa kluczowego **virtual** (przy implementacji słowa **virtual** nie używamy),
* Przy wywołaniu metody wirtualnej zostanie przeszukana tablica funkcji wirtualny przynależąca do danego obiektu i wyszukana wersja najbliższa w hierarchii dziedziczenia,
* Metody wirtualne w danej hierarchii powinny mieć pełną zgodność. Nie może się różnić liczba parametrów metody. Wartość zwracana może się różnić, ale tylko jeśli w klasie pochodnej metoda zwraca klasę dziedziczącą po typie wartości zwracanej przez metodę klasy bazowej.
* Funkcja zaczyna zaczynać się jak wirtualna w momencie pierwszego pojawienia się słowa **virtual**,
* Zachowanie wirtualne może (ale nie musi) skończyć się po pierwsyzm wsytąpieniu funkcji bez **virtual** w hierarchii dziedziczenia,
* Rodzaj dziedziczenia nie wpływa na zachowanie sie funkcji wirtualnych (zmienia się jedynie ich widoczność),
* Metody statycznie nie mogą być wirtualne,
* W przypadku argumentów domyślnych wykorzystana będzie wartość odpowiadająca wersji funkcji według typu wskaźnika, a nie dynamicznego typu (związane z określaniem wartości domyślnych na etapie kompilacji),
* Konstruktor nie może być wirtualny,
* Destruktor powinien być wirtualny ze względu na niebezpieczeństwo wycieków pamięci jeśli w klasie została zadeklarowana jakakolwiek metoda wirtualna. Deklaracja destruktora powinna znajdować się w klasie bazowej.

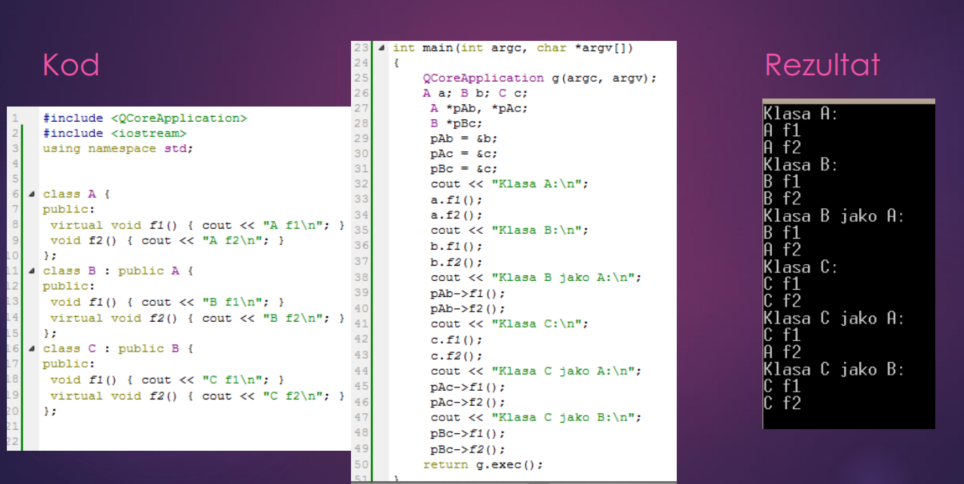
Polimorfizm uzyskujemy dzięki:

* Niejawnemu przekształceniu typów wskaźników do klas pochodnych na wskaźniki do klas typu podstawowego,
* Mechanizmowi funkcji wirtualnych,
* Operacji rzutowania *dynamic\_cast* oraz operatorom *typeid*

**Polimorfizm kosztuje** - Klasy **polimorficzne** zajmują więcej miejsca w pamięci, ponieważ kompilator automatycznie dodaje do nich wskaźnik vptr wskazujący na tablicę vtab. Dla każdej klasy musi istnieć osobny wskaźnik i osobna tablica. Tablica jest generowana automatycznie i zawiera wskaźniki do funkcji, wygenerowane przez kompilator.

W języku JAVA wszystkie metody są wirtualne.

**Przykład polimorfizmu C++**



Klasa B jako A:

* metoda *f1()* zostanie związana dynamicznie z typem na, który wskaźnik pAb będzie wskazywał (typ klasy B).
* metoda *f2()* nie jest zadeklarowana jako wirtualna zatem zostanie związana statycznie z typem wskaźnika w fazie kompilacji - stąd wywołanie metody *f2()* na wskaźniku pAb wywołuje metodę z klasy A

Klasa C jako A taka sama sytuacja.

Klasa C jako B:

* metoda *f1()* była zadeklarowana jako wirtualna w klasie A, zatem zostanie związana dynamiczna i wywołana implementacja z klasy C,
* metoda *f2()* została zadeklarowana jako wirtualna w klasie B, więc również zostanie związana dynamicznie i wywołana implementacja z klasy C

Jawne wywołanie funkcji bazowej jest możliwe poprzez poprzedzenie metody dwukropkiem i nazwą klasy.

# 4. Pojęcie zbioru rozmytego, definicja i interpretacja funkcji przynależności.

**Zbiorem rozmytym A** w pewnej niepustej przestrzeni **X**, co zapisujemy nazywamy zbiór par:

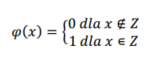
gdzie:

μA(x) jest **funkcją przynależności** zbioru rozmytego A. Funkcja ta każdemu elementowi przyporządkowuje jego stopień przynależności do zbioru rozmytego A, przy czym można wyróżnić 3 przypadki:

* μA(x) = 1 - oznacza to pełną przynależność do zbioru rozmytego A, tzn. ,
* μA(x) = 0 - oznacza to brak przynależności elementu x do zbioru rozmytego A, tzn.
* 0< μA(x) < 1 - oznacza to częściową przynależnośc elementu x do zbioru rozmytego A.

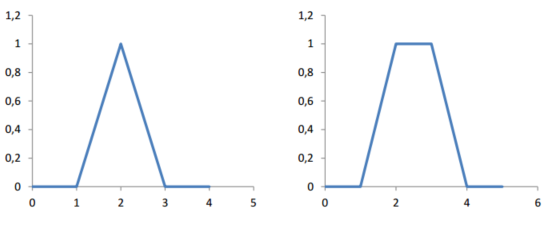
**Funkcja przynależności** jest uogólnieniem funkcji charakterystycznej określonym na zbiorach rozmytych.

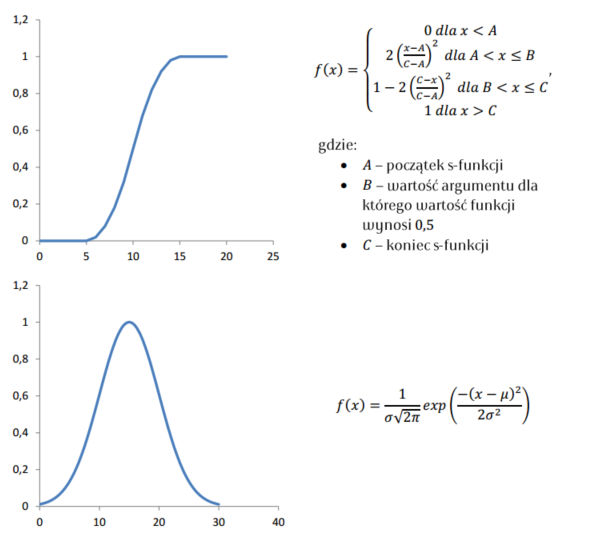
Funkcja charakterystyczna zbioru Z to funkcja φ(x), która elementom przestrzeni X przyporządkowuje wartości 0 i 1:

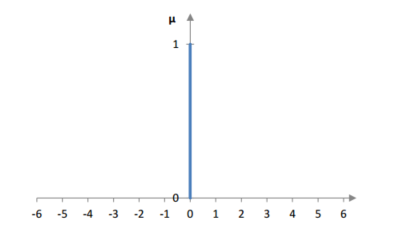


Funkcja charakterystyczna określa, czy dany element x przestrzeni X należy, czy nie należy do zbioru Z, a funkcja przynależności określa dodatkowo stopień przynależności do tego zbioru.

W teorii zbiorów rozmytych istnieją różne funkcje przynależności. Najczęściej stosowane są funkcje trójkątne, trapezowe, singleton, s-funkcje oraz funkcje Gaussa (dwie ostatnie ze względu na to, że są różniczkowalne).







# 5. Elementy składowe wnioskowania rozmytego.