Typy klasowe (klasy)

1. Programowanie obiektowe

Programowanie obiektowe *(ang. object-oriented programming)* to metodologia tworzenia programów komputerowych, która definiuje programy za pomocą „obiektów" – elementów łączących stan (czyli dane) i zachowanie (czyli procedury, tu metody).

Obiektowy program komputerowy wyrażony jest jako zbiór obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań.

W konstrukcji złożonych systemów informatycznych paradygmat obiektowy pozwala tworzyć systemy o lepszej jakości:

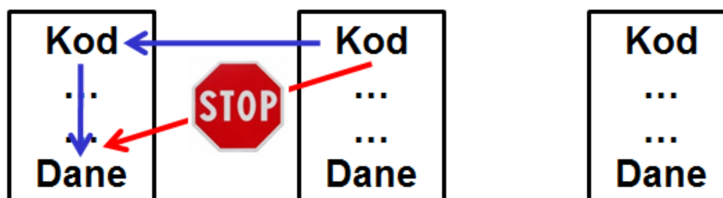
- Łatwość utrzymania i rozumienia kodu
- Odporność architektury na modyfikacje
- Łatwość wielokrotnego użycia
- Lepsze zrozumienie kodu

Zbliżone jest do ludzkiego sposobu postrzegania rzeczywistości stąd łatwiej zrozumieć kod i pomysły innych programistów.

2. Założenia paradygmatu obiektowego:

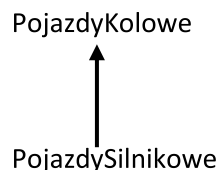
a. **Abstrakcyjne typy danych** – umożliwia tworzenie od podstaw nowych typów. Jest to ograniczenie cech obiektu z otaczającego świata do cech istotnych, kluczowych z punktu widzenia programisty (uproszczenie problemu i zwiększenie jego ogólności).
   i. Specyfikacją i implementacją definiowanego przez programistę abstrakcyjnego typu danych jest klasa.
   ii. Np. traktowanie w danym kontekście instancji klasy Dąb jako instancji klasy Drzewo.

b. **Hermetyzacja (enkapsulacja, kapsułkowanie)** – polega na ukrywaniu szczegółów implementacji przed użytkownikiem obiektu. Zapewnia, że obiekt nie może zmieniać stanu wewnętrznego innych obiektów w nieoczekiwany sposób (poznanie lub zmiana wartości atrybutów obiektu tylko poprzez specjalne metody). Tylko wewnętrzne metody obiektu są uprawnione do zmiany jego stanu. Każdy typ obiektu prezentuje innym obiektom swój „interfejs", który określa dopuszczalne metody współpracy.

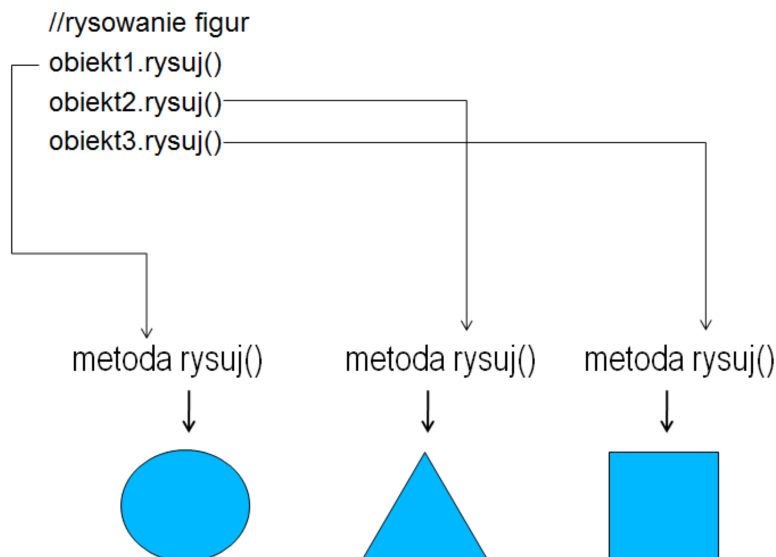


Hermetyczność obiektów: obiekt = kod + dane

c. **Dziedziczenie** – pozwala na tworzenie typów na podstawie już istniejących (bardziej wyspecjalizowanej, rozszerzonej wersji istniejącego typu). Np. klasa Dąb jako specjalizacja klasy Drzewo. Porządkuje i wspomaga polimorfizm i enkapsulację. Dla obiektów specjalizowanych nie trzeba redefiniować całej funkcjonalności, lecz tylko tę, której nie ma obiekt ogólniejszy.
   i. Technika wykorzystania istniejących fragmentów kodu polega na tworzeniu nowych klas na bazie już istniejących.
   ii. Cechy wspólne dla wszystkich podklas definiowane są w nadklasie.
   iii. Podklasa może korzystać z cech nadklasy, nadpisywać jej zachowanie oraz dodawać nowe atrybuty i zachowania



d. **Polimorfizm** – pozwala na dostosowanie działania obiektów do własnych oczekiwań. Programista używający obiektu nie musi wiedzieć czy konkretne zachowanie wykorzystywanego obiektu zostało zrealizowane w danym obiekcie czy też w tym po którym dziedziczy on swoje właściwości. Niektóre języki udostępniają bardziej statyczne (w trakcie kompilacji) rozwiązania polimorfizmu np. szablony i przeciążanie operatorów w C++.
   i. Pozwala w jednolity sposób traktować obiekty klas z hierarchii dziedziczenia przy zachowaniu ich charakterystycznego zachowania.
   ii. Od strony technicznej sprowadza się do tzw. *późnego wiązania* metod przy ich wywołaniu (decyzja z której klasy wywołać metodę podejmowana w trakcie działania programu a nie na etapie kompilacji; kompilator upewnia się, że metoda istnieje, jednak kod jest dopiero ustalany w czasie wykonania).
   iii. W Javie wszystkie metody zachowują się jak metody wirtualne w C++.



3. Definicje

a. **Klasa** – jest zdefiniowanym przez użytkownika typem danych, który ma swój stan (jego reprezentację) oraz pewną liczbę operacji (jego zachowanie). Klasa zawiera pewne wewnętrzne atrybuty (dane, struktury danych) oraz metody (operacje, funkcje) i opisuje zbiór obiektów o takiej samej budowie tj. posiadających takie same cechy i funkcjonalność.

   i. Pola (podobnie jak w rekordzie), własności (łącza do danych, przechowywanych zwykle w polach, przy odczytywaniu i zapisywaniu których można wykonywać określone operacje) i metody (procedury i funkcje) klasy określane są jako **składowe klasy**.

   ii. Np. klasa Kot opisująca cechy oraz zachowanie wspólne dla wszystkich kotów np. kolor sierści i umiejętność miauczenia.

   iii. Dane i metody klasy ukrywa się definiując (deklarując) je zależnie od stopnia ukrycia jako prywatne lub zabezpieczone.

   iv. Definicja pojedynczego typu klasowego ma postać (nawias kwadratowy oznacza, że dany element jest opcjonalny i może być pominięty):

```
type nazwa-typu = class [abstract/sealed] [(przodek)]
                            lista-elementów-klasy
                      end;
```

   1. Przodek – identyfikator innego, wcześniej zdefiniowanego typu klasowego. Jeśli występuje w definicji to oznacza, że mamy do czynienia z dziedziczeniem. Typem klasowym, nadrzędnym w stosunku do wszystkich typów klasowych, jest typ *TObject* predefiniowany w module *System*.
   2. Dyrektywa *sealed* oznacza, że dana klasa nie może mieć potomków.
   3. Dyrektywa *abstract* oznacza klasę abstrakcyjną.
   4. Lista elementów klasy to lista deklaracji pól, własności i metod.

   Np.

```
type moja_klasa = class
                   pole1 : Integer;
                   procedure metoda1(parameter1 : string);
             end;
```

   v. Podstawowe kwalifikatory dostępu do składowych:

   - **public** (dostęp publiczny),
     Metody umieszczone w sekcji `public` są dostępne dla wszystkich innych klas i modułów. W tej sekcji powinny znajdować się konstruktory oraz destruktory, a także metody służące do komunikowania się ze "światem zewnętrznym".

```
type
  TMyClass = class
  public
    procedure Foo; // publiczna metoda
  end;
```

- **published** (takie dane będą dostępne dla inspektora obiektów)
- **protected** (dostęp chroniony w klasie definiowanej i pochodnych), Metoda umieszczona w sekcji protected jest dostępna zarówno dla modułu, w którym znajduje się klasa, jak i dla całej klasy. Jest to jakby drugi poziom ochrony, gdyż metody z sekcji protected są dostępne dla innych klas, które dziedziczą po danej klasie.

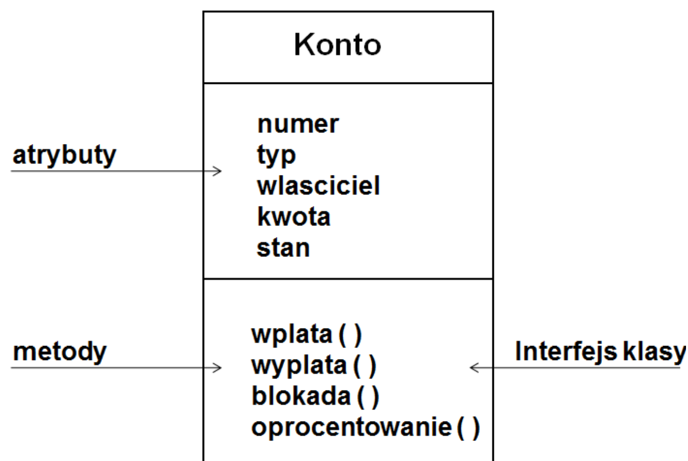
```
type
  TMyClass = class
  protected
    procedure Foo; // chroniona metoda
  end;
```

- **strict protected**
  Metody czy pola umieszczone w sekcji **strict protected** są określone jako ściśle chronione, dostęp do nich z poziomu innej klasy jest niemożliwy, chyba że z poziomu klasy potomnej.
- **private**
  Metody umieszczone w sekcji `private` są określane jako prywatne. Oznacza to, że nie będą dostępne na zewnątrz modułu, w którym znajduje się dana klasa. A zatem po próbie odwołania się do metody umieszczonej w sekcji `private` kompilator zasygnalizuje błąd, że nazwa owej metody nie będzie mogła być przez niego rozpoznana.

```
type
  TMyClass = class
  private
    I : Integer; // prywatne pole
    procedure Foo; // prywatna metoda
  end;
```

- **strict private**
  Metody czy pola umieszczone w sekcji strict private są określone jako ściśle chronione, dostęp do nich z poziomu innej klasy jest niemożliwy.

vi. Składowe klasy dostępne z zewnątrz stanowią interfejs klasy.



b. **Obiekt** – stanowi odwzorowanie wycinka rzeczywistości. Jest elementem klasy, czyli zmienną typu zdefiniowanego przez klasę.
   i. Kiedy program działa obiekt zajmuje pewien obszar w pamięci.
   ii. W Delphi dla każdego obiektu w pamięci rezerwowane jest miejsce na jego pola. Natomiast dla metod danego typu tworzona jest w pamięci tylko jedna kopia i wszystkie obiekty utworzone na podstawie tego typu (klasy) z nich korzystają.
   iii. Zależności między obiektem a klasą są podobne do tych między zmienną i jej typem.
   iv. Po utworzeniu klasy można utworzyć dowolną ilość obiektów tej klasy a następnie operować na nich tak, jakby były rzeczywistymi elementami rozwiązywanego problemu.
   v. Jest instancją (wystąpieniem) klasy np. konkretny kot – Filemon o białym kolorze sierści.
   vi. **Stan obiektu** to zbiór wartości atrybutów obiektu w danej chwili.
   vii. Obiekt może wykonywać na sobie pewne działania (uruchamiać zaprogramowane funkcje czyli metody, funkcje składowe) co czyni obiekt tworem aktywnym – nie tylko pojemnikiem na dane.

c. **Przekazywanie komunikatów** - proces polegający na przekazaniu danych z obiektu do obiektu lub zleceniu wywołania metody na rzecz obiektu.

d. **Metoda** – operacja (funkcja), która może być wykonana na obiekcie. Np. metoda miaucz() klasy Kot może być wywołana na rzecz konkretnego obiektu klasy Kot, np. obiektu Filemon.
   i. Metody wiąże się z klasami głównie po to, żeby nie zaśmiecać kodu źródłowego i samego programu nadmierną ilością funkcji globalnych, które i tak nie zostaną użyte inaczej, niż na rzecz konkretnej klasy.
   ii. Zaletą jest, że metoda wewnętrzna danej klasy ma dostęp do wszystkich składników tej klasy (także prywatnych i chronionych).
   iii. **Konstruktor** - Inicjuje obiekty klasy, konstruuje wartości danego typu.
      1. Może być wywołany przez odwołanie do obiektu lub klasy.

```
type moja_klasa = class
                  pole1 : Integer;
            public
            procedure metoda1(parameter1 : Integer);
            constructor Inicjuj(aut, tyt : string);
      end;
{…}
{deklaracja zmiennej}
var
      moj_obiekt : moja_klasa;
      x : Integer;
begin
{…}
{zainicjowanie obiektu}
moj_obiekt := moja_klasa.Inicjuj(”Autor”, ”Tytuł”);
```

```
{odwołanie do składowych klasy: wywołanie metody - nazwa_obiektu.metoda, odwołanie do pola - nazwa_obiektu.pole }
moj_obiekt.metoda1(x);
{praca z obiektem}
end;
```

4. Przykład:

```
program Klasy;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils;

{definicja klasy o nazwie punkt}
type punkt = class
  x, y : Integer;  {pola}
  public {metody ogólnie dostępne}
  procedure wspolrzedne (nX, nY : Integer);
  function rzedna : Integer;
  function odcieta : Integer;
  constructor Inicjuj(nX, nY : Integer); {konstruktor}
end;

{ustawia nowe współrzędne}
procedure punkt.wspolrzedne (nX, nY : Integer);
begin
  x := nX;
  y := nY;
end;

{zwraca wartość pola x – pole jest prywatne, więc dostęp bezpośredni jest zabroniony}
function punkt.rzedna : Integer;
begin
  rzedna := x;
end;

{zwraca wartość pola y – pole jest prywatne, więc dostęp bezpośredni jest zabroniony}
function punkt.odcieta : Integer;
begin
  odcieta := y;
end;

constructor punkt.Inicjuj(nX, nY : Integer);
begin
  {wywołanie kostruktora typu nadrzędnego (domyślnie jest to typ
  TObject). Taka instrukcja zawsze powinna być pierwszą instrukcją
  każdego definiowanego konstruktora.}
  inherited Create;
```

```
  {przypisanie wartości polom obiektu}
  x := nX;
  y := nY;
end;

var
 x1, y1 : Integer;
 moj_punkt : punkt; {deklaracja zmiennej typu klasowego}
begin
  try
  writeln('Podaj wspolrzedne punktu: ');
  readln(x1, y1);
  moj_punkt := punkt.Inicjuj(x1, y1); {zainicjowanie obiektu}
  writeln('Wartosci poczatkowe: x = ', moj_punkt.rzedna, ', y = ', moj_punkt.odcieta );
  writeln('Podaj wspolrzedne punktu: ');
  readln(x1, y1);
  moj_punkt.wspolrzedne(x1, y1);
  writeln('Wartosci po zmianie: x = ', moj_punkt.rzedna, ', y = ', moj_punkt.odcieta );
  readln;
    { TODO -oUser -cConsole Main : Insert code here }
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end;
end.
```