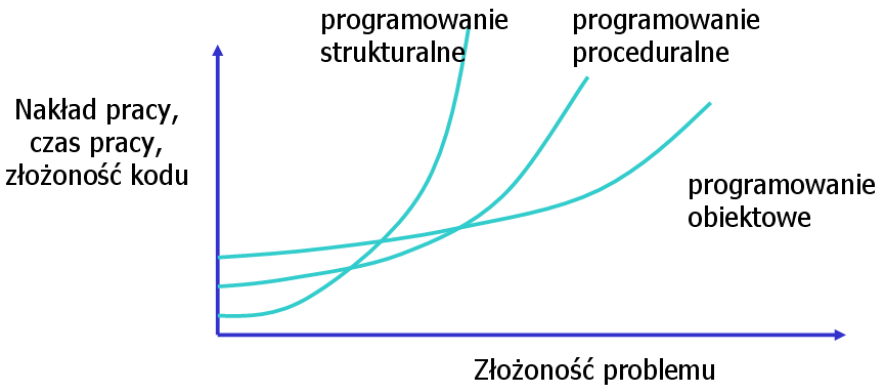
Metodyki programowania

Istnieją trzy główne metodyki programowania:

- Programowanie strukturalne
- Programowanie proceduralne
- Programowanie obiektowe

Zastosowanie "wyższych" metodyk programowania wiąże się zawsze z dodatkowym nakładem pracy, jednak przynosi znaczne korzyści – przy bardziej złożonych programach ten nakład pracy bardzo szybko się zwraca



DRY (ang. don't repeat yourself)

- Dodatkowa korzyść z programowania proceduralnego oraz obiektowego to możliwość wielokrotnego użycia kodu.
- Postulat DRY zaleca unikanie powtórzeń kodu (np. przez Ctrl-C, Ctrl-V), do czego właśnie służą wyższe metodyki: fragment kodu można "zamknąć" w postaci oddzielnej funkcji lub klasy i odwołać się do niego, zamiast go kopiować.
- Co ważne, DRY zmniejsza też liczbę błędów (zatem redukuje czas potrzebny na ich poprawianie) – jeżeli funkcja zawiera błędy, to trzeba je poprawić tylko w jednym miejscu, zaś raz opracowana i pozbawiona błędów funkcja już zawsze będzie działać poprawnie
- DRY ułatwia również utrzymanie kodu – jeżeli jakiś aspekt programu trzeba ulepszyć (np. zastosować szybszy algorytm), to wystarczy to zrobić w jednym miejscu
- Skrajne podejście DRY widać np. w języku Java, gdzie zaleca się stosowanie zasady jeden plik = jedna klasa (publiczna) – w ten sposób każda klasa może łatwo być użyta wielokrotnie
- Postulat DRY dotyczy też innych aspektów programowania, np. użycia narzędzi programistycznych, kompilacji warunkowej, a nawet definiowania stałych

Geneza i historia programowania obiektowego

- Simula, 1967 r., Ole-Johan Dahl i Kristen Nygaard – rozwinięcie języka ALGOL, klasy i obiekty (jako instancje klas); Dahl i Nygaard pracowali nad komputerowymi modelami statków – duża liczba typów statków i ich parametrów (różnych dla różnych typów) oraz relacji pomiędzy nimi była przyczyną problemów. Wpadli na pomysł, aby dane statków każdego typu zgrupować w jeden programowy byt – obiekt
- Smalltalk, 1971 r., Xerox – wiele nowatorskich rozwiązań, m.in. dziedziczenie, maszyna wirtualna, programy GUI (okienkowe), mysz

- C++, 1979 r., Bjarne Stroustrup – obiektowe rozwinięcie C (1973 r., Bell Labs), wspiera paradygmaty proceduralny i obiektowy, umożliwia też wklejanie kodu języka Asembler – ceniony za wydajność
- Java, 1991, Sun Microsystems – koncepcja Smalltalk, składnia C++ oraz kilka innowacji – np. interfejsy;
- C#, 1998 r., Microsoft – nazwa sugeruje związek z C++, koncepcja bardziej zbliżona do Java (maszyna wirtualna, język pośredni IL, zarządzanie pamięcią GC), wiele cennych rozszerzeń, np. delegacje

Pojęcie i definicja klasy

Jeżeli przyjąć, że program komputerowy służy do rozwiązania jakiegoś problemu, to klasa jest modelem elementu jego dziedziny.

- Definicja klasy obejmuje **stan** obiektu (wartości jego pól) oraz **zachowanie** (działanie), przez definicje metod i właściwości
- Klasa jest typem dla obiektu, a obiekt danej klasy to jej instancja

Definicja klasy

```
[ <[> Atrybuty<]> ]
[ <Modyfikatory> ] class <Nazwa> [ : <Klasa-bazowa> ]
{
  // elementy składowe klasy
}
```

- Atrybuty, np. [Serializable] – metadane, dołączane przez kompilator do kodu klasy; wg MS stanowią odpowiednik słów kluczowych (jak np. modyfikatory dostępu private/public), ale dowolnie definiowanych; Mogą być wykorzystane na różne sposoby – np. mogą stanowić wytyczne dla kompilatora JIT, mogą dodawać pewne funkcjonalności
- Modyfikatory – dostępu (np. private/public), związane z dziedziczeniem (abstract, sealed) oraz modyfikator static
- Klasa bazowa – klasa, po której aktualnie definiowana klasa dziedziczy wszystkie elementy składowe; Domyślnie klasa object (System.Object)

Przykład

```
[Serializable]
public class FirstClass
{
  private Double value;

  public FirstClass(Double value)
  {
    this.value = value;
  }

  public Double Value
  {
    get;
    set;
  }
}
```

Modyfikatory dostępu

- **internal** (domyślnie!) – klasa jest dostępna tylko w obrębie tego samego pakietu (assembly = plik .dll lub .exe)
- **public** – klasa jest dostępna z dowolnego zestawu .NET
- protected – tylko klasy zagnieżdżone wewnątrz innej klasy – klasa jest dostępna tylko dla klasy zawierającej i jej klas potomnych
- private – tylko klasy zagnieżdżone wewnątrz innej klasy – klasa jest dostępna tylko dla klasy zawierającej

Pozostałe modyfikatory

- **static** – klasa zawiera wyłącznie metody statyczne, nie można utworzyć obiektu takiej klasy
- abstract – klasa zawierająca metody abstrakcyjne; nie można utworzyć obiektu takiej klasy, ale można zdefiniować jej klasy potomne
- sealed – nie może być przedmiotem dziedziczenia

Nazwa - zasady nieformalne:

- Klasy, pola, metody, stałe – konwencja Pacal,
  nazwy klas i pól – rzeczowniki, metod – czasowniki:
- Interfejsy – Pascal poprzedzone literą "I"
- Nazwy powinny być znaczące:
  Kod powinien być czytelny, kiedy do niego zajrzeć po kilku miesiącach albo dla innego programisty (tzw. samodokumentujący się)

Klasa bazowa

Klasa bazowa to klasa, po których definiowana klasa dziedziczy elementy składowe, i/lub interfejsy, które implementuje

- Klasa, która dziedziczy po innej klasie to klasa potomna, natomiast klasa, po której dziedziczy klasa potomna to klasa bazowa
- Klasa może dziedziczyć po jednej klasie bazowej oraz implementować dowolnie wiele interfejsów (klasa bazowa, o ile występuje, musi być wymieniona przez interfejsami)
- Domyślnie klasa dziedziczy po klasie System.Object

- Dziedziczenie klas i interfejsów ma inny sens:
  – w przypadku klas jest to dosłowne dziedziczenie, klasa potomna otrzymuje wszystkie elementy kładowe klasy bazowej – można z nich korzystać tak samo, jakby były umieszczone w klasie potomnej
  – w przypadku interfejsów dziedziczenie oznacza raczej zobowiązanie do zaimplementowania wszystkich metod interfejsu

Elementy składowe klasy

- Stała – niezmienna wartość (niemodyfikowalna zmienna);
  Należy do klasy, nie do obiektów, podobnie jak metody statyczne
- **Pole** – zmienna należąca do obiektu, część stanu obiektu;
  Zwykle pola są prywatne (dostępne wyłącznie dla metod obiektu)
- **Właściwość** – sposób udostępnienia stanu obiektu (prywatnego pola) w kontrolowany sposób
- **Metoda** – funkcja związana z klasą, określająca działania, jakie może wykonać obiekt klasy (nie dotyczy metod statycznych)

- **Konstruktor** – metoda wykonywana w trakcie tworzenia obiektu
- Destruktor – metoda wykonywana w trakcie "niszczenia" obiektu
- Operator – definicja działania operatora (np. +, *, &&, …)
- Zdarzenie – mechanizm informowania innych obiektów np. o zmianie stanu; w praktyce zmienna typu delegacji
- Typ – zagnieżdżona definicja typu – klasy, interfejsu, delegacji, …

Elementy składowe klasy - modyfikatory dostępu

- internal – element jest dostępny tylko w obrębie tego samego pakietu (assembly = plik .dll lub .exe)
- **public** – element jest dostępny z dowolnego zestawu .NET
- protected – element jest dostępny tylko dla klasy zawierającej i jej klas potomnych
- **private** – (domyślnie) element jest dostępny tylko dla klasy zawierającej

Pola

Pola są zmiennymi należącym do obiektów lub klasy i reprezentują ich wewnętrzny stan; Z zasady powinny być prywatne, natomiast do udostępniania stanu obiektu powinny służyć właściwości

Dodatkowe modyfikatory dla pól

- static – pole statyczne, należy do klasy a nie obiektu
- readonly – pole tylko do odczytu (wartość możne nadać konstruktor)
- const – stała – wartość typu prostego, nadawana w czasie kompilacji i później nie może być zmieniona; należy do klasy (≈ static readonly)

```
class Sample
{
  public const Double PI = 3.1415;
  public static readonly DateTime Start = DateTime.Now;
  private Int32 length;
  // …
}
```

Właściwości

- Właściwości służą do kontrolowanego dostępu do stanu obiektu – umożliwiają odczyt i zmianę stanu (tj. wartości prywatnych pól)
- Właściwość składa się z dwóch metod: accesor (odczyt pola) oraz mutator (zmiana wartości pola); w językach C++/Java służą do tego oddzielne metody (w slangu Java nazywane getter i setter), natomiast w C# połączone w jedną konstrukcję składniową
- Właściwość składa się z sekcji get i/lub set;
  Obie sekcje są w pewnym sensie metodami – można dla nich używać modyfikatorów takich jak dla metod (np. virtual, override)

- Używanie właściwości nie oznacza spowolnienia programu – kompilator JIT może zastępować wywołania prostych właściwości kodem sekcji get lub set

Definicja właściwości:

```
<modyfikator> <typ> Nazwa
{
  [ <modyfikator> ] get
  {
    // dowolny kod
    return <wartość-właściwości>
  }
  [ <modyfikator> ] set
  {
    // dowolny kod
    // wartość przypisywana właściwości
    // jest dostępna jako value
  }
}
```

Przykład

```
private Double promień;

public Double Promień
{
  get
  {
    return promień;
  }
  set
  {
    if (value<0)
      throw new ArgumentException();
    promień = value;
}

public Double Powierzchnia
{
  get
  {
    return Math.PI * promień * promień;
  }
}
```

Metody

- Odpowiadają za wykonywanie działań, które definiują zachowanie się obiektów (można powiedzieć że definiują polecenia, które można wydawać obiektom)
- Mogą być przeciążone
  Przeciążenie metody to definiowanie wielu metod o tej samej nazwie, ale różniących się liczbą i/lub typami argumentów (do przeciążenia nie wystarczy inna nazwa argumentu lub inny typ rezultatu)
- Odpowiadają za polimorfizm (modyfikatory virtual i override)

Metody – dodatkowe modyfikatory

- **static** – metody statyczne są częścią klasy, nie obiektu
- abstract – metoda abstrakcyjna może pojawić się tylko w klasie abstrakcyjnej; zawiera samą metryczkę (bez ciała metody)

- **virtual** – metoda może być nadpisana w klasie potomnej
- **override** – metoda klasy potomnej, nadpisująca metodę odziedziczoną

- new – metoda zastępująca metodę nie-wirtualną w klasie potomnej
- sealed – metoda finalna, nie może być nadpisana
- extern – metoda implementowana zewnętrznie ("pobierana" z DLL dyrektywą DLLImport)

Definicja metody:

```
<modyfikator> <typ-reultatu> Nazwa (<lista-parametrów>)
{
  // ciało metody – dowolny kod;
  // jeżeli typ rezultatu jest inny, niż void,
  // to powinien zawierać instrukcję zwracania
  // rezultatu:
  return <wyrażenie>;
}
```

Przykład

```
private Double radius;

public void SetRadius (Double radius)
{
  this.radius = radius;
}

public Double GetArea ()
{
  return Math.Pi * radius * radius;
}
```

Metody maja nieograniczony dostęp do wszystkich pól, właściwości i metod klasy oraz jej klas bazowych (oprócz private klasy bazowej); Dostęp do własnych elementów jest BEZ operatora dostępu ".", ale można go użyć ze słowem kluczowym **this** (= bieżący obiekt), np. kiedy występuje konflikt nazw pomiędzy polem a argumentem

Polimorfizm

Obiekt klasy potomnej można zawsze przypisać do zmiennej typu klasy bazowej – np. w C# wszystkie klasy dziedziczą po klasie obiect:

```
StreamWriter sw = new StreamWriter(path);
Object o = sw;
```

Zmienna klasy bazowej udostępnia tylko te elementy obiektu, które zostały zdefiniowane w klasie bazowej

Polimorfizm jest implementowany przez metody wirtualne – modyfikator **virtual** w klasie bazowej i **override** w klasach potomnych. W sytuacji jak wyżej, wywołanie metody polimorficznej spowoduje wykonanie metody zdefiniowanej w klasie potomnej, a nie bazowej;

W C# wszystkie klasy mają metodę ToString() – wirtualną, zdefiniowaną w klasie System.Obiect; Jeżeli ta metoda zostanie nadpisana w klasie potomnej, to nawet po przypisaniu referencji obiektu do zmiennej typu object, zostanie wykonana metoda ToString klasy potomnej

```
DateTime dt = DateTime.Now;
obiect o = dt;

String s = o.ToSring();   // = DateTime.ToString()
```

Warto zauważyć, że np. dzięki temu Console.WriteLine akceptuje jako argumenty obiekty dowolnej klasy, również nieznanej w momencie tworzenia .NET – używa ich metody ToString, a dzięki polimorfizmowi wykonywana jest metoda ToString klas potomnych

Inicjacja obiektów i konstruktory

- Pola niezainicjowane otrzymują wartość domyślną – zależnie od typu jest to 0, 0.0, false lub wskazanie puste (null)
- Pola można inicjować razem z ich deklaracją (dla stałych jest to obowiązkowe) albo w konstruktorach

- Konstruktor jest specjalną metodą, używaną wyłącznie do tworzenia obiektów – nie można go wywołać jawnie, natomiast jest wykowywany zawsze podczas tworzenia obiektu operatorem new
- Kompilator C# wyposaża klasę w domyślny konstruktor bezargumentowy, ale tylko kiedy klasa nie ma innych konstruktorów
- Konstruktor może być przeciążony
- Konstruktory nie są dziedziczone;
  konstruktor klasy potomnej może "wywołać" konstruktor klasy bazowej

Przykład

```
class Sample
{
  private Int32 n1 = 13;
  private Int32 n2;       // tj. n2 = 0

  public Sample (Int32 n2)
  {
    this.n2 = n2;
  }
}

Sample s1 = new Sample (13); // s1.n2 = 13
Sample s2 = new Sample();          // błąd!
Sample s3 = new Sample(7, 77);     // błąd!
```

Destruktory

- Destruktor jest specjalną metodą, wywoływaną niejawnie przez GC w momencie "niszczenia" obiektu
- Klasy korzystające wyłącznie z zasobów .NET (zarządzanych) nie wymagają destruktora – GC wykonuje zwalnia zasoby wykorzystywane przez obiekt, zwłaszcza pamięć
- Klasy korzystające z zasobów spoza .NET (niezarządzanych), powinny mieć destruktor, którego zadaniem jest zwolnienie zasobów
- Destruktor musi być bezargumentowy i nie może być przeciążony

```
class Sample
{
  // …
  public ~Sample()
  {
    // zwolnienie zasobów niezarządzanych
  }
}
```

Struktury (struct)

- Służą do definiowania typów wartościowych (klasy – typy referencyjne)
- Mają składnie podobną do klas, z kilkoma różnicami:
  – Domyślny modyfikator elementów: public
  – Nie można jawnie definiować konstruktora bezargumentowego, konstruktor taki jest zawsze dostępny
  – Nie może być dziedziczona, ale może implementować interfejs
- Struktury należy stosować zamiast klas, kiedy w sposób naturalny reprezentują pojedynczą wartość – np. data/czas, temperatura itp.
  Struktury – ze względu na wydajność – powinny zajmować niewiele pamięci (MS: do 16 bajtów), ponieważ są tworzone na stosie, a nie na stercie jak obiekty klas

Zadania

Proszę napisać program, który…

1. Tworzy dwie instancje struktury `Temperature`, wyświetla i modyfikuje wartość publicznego pola `Celsius` tej struktury oraz przypisuje jedną strukturę do drugiej;

   Struktura `Temperature` powinna zawierać wyłącznie jedno publiczne pole `Celsius`

2. Wykonuje zadania z punktu 1, ale po zmianie struktury na klasę
   Klasa `Temperature` powinna zawierać wyłącznie jedno publiczne pole `Celsius`

3. Tworzy dwa obiekty klasy `Temperature`, wyświetla i modyfikuje wartość publicznej właściwości `Celsius` obiektów oraz kopiuje wartość właściwości między obiektami

   Klasa `Temperature` powinna zawierać prywatne pole `celsius` oraz powiązaną z nim publiczną właściwość `Celsius`, poprzez którą można odczytać lub zmienić wartość pola

4. Wykonuje zadania z punktu 3; dodatkowo należy użyć pracy krokowej i sprawdzić kiedy jest wykonywana sekcja `set` a kiedy sekcja `get` właściwości

5. Wykonuje interaktywnie polecenia modyfikujące i odczytujące właściwość `Celsius` obiektu klasy `Temperature`; Należy uwzględnić polecenia „get” – wyświetlenie wartości temperatury oraz „set” – wprowadzenie z klawiatury nowej wartości temperatury;

   Klasa `Temperature` powinna zawierać prywatne pole `celsius` oraz powiązaną z nim publiczną właściwość `Celsius`, poprzez którą można odczytać lub zmienić wartość pola. W przypadku próby przypisania nieprawidłowej wartości temperatury (poniżej -273 C) sekcja `set` powinna rzucić wyjątek, np. `ArgumentOutOfRangeException`, który powinien być obsłużony w głównej pętli programu

6. Wykonuje zadania z punktu 3, ale dodatkowo obsługuje polecenia „getf” i „setf”, które z obiektu tej samej klasy odczytują i zapisują wartość temperatury w skali Fahrenheita;

   Klasa `Temperature` powinna zawierać uzupełniona o właściwość `Fahrenheit`, ale ponieważ wewnętrzny stan obiektu (pole `celsius`) jest w skali Celsjusza, to właściwość `Fahrenheit` powinna przeliczać temperaturę z/na skalę Celsjusza

7. Wykonuje zadania z punktu 6, ale dodatkowo definiuje konstruktor dla klasy `Temperature`

8. Wykonuje zadania z punktu 7, ale nadpisuje metodę `ToString()` klasy `Temperature`, odziedziczoną z klasy `System.Object`;

   Metoda `ToString()` powinna zwracać wartość typu `String`, reprezentującą wewnętrzny stan obiektu (np. `"22,0 °C"`), natomiast wydruk powinien następić w głównej pętli programu; Metoda `ToString()` nie może sama drukować czegokolwiek,

9. Tworzy listę obiektów klasy `Temperature`, do której użytkownik może dodawać nowe obiekty poleceniem „add” oraz wyświetlać wszystkie zapisane na liście obiekty poleceniem „view”;

   Do utworzenia listy należy użyć klasy uniwersalnej `List<>`, zaś do dodawania do listy obiektów metody `Add()` tej klasy, np. w taki sposób:

   ```
   List<Temperature> LogMeteo = new List<Temperature>();
   LogMeteo.Add(t);
   ```

   Użyć pracy krokowej do zlokalizowania błędu w programie, poprawić błąd