Programowanie funkcyjne

Materiały z wykładu i laboratorium

Łukasz Bartczuk

6. Programowanie generyczne

Język C# jest językiem o ścisłym typowaniu (podobnie jak F#). Oznacza to, że typy są przypisywane do każdej wartości, wyrażenia czy symbolu już na etapie kompilacji. Często jednak zdarza się, że pewne fragmenty kodu są jednak identyczne z dokładnością do typów danych, na których działają. Przyjrzyjmy się wręcz akademickiemu przykładowi. Chcemy napisać funkcję, która będzie zamieniała wartości dwóch zmiennych. Dzięki możliwości przeciążania funkcji możemy to zrealizować następująco:

```
static void Zamiana(ref int x, ref int y)
{
  int tmp = x;
  x = y;
  y = tmp;
}

static void Zamiana(ref double x, ref double y)
{
  double tmp = x;
  x = y;
  y = tmp;
}

static void Zamiana(ref char x, ref char y)
{
  char tmp = x;
  x = y;
  y = tmp;
}
```

Przykładowe zastosowanie powyższych funkcji wygląda następująco:

```
static void Main(string[] args)
{
  int x = 10, y = 100;
  Console.WriteLine($"x={x}, y={y}");
  Zamiana(ref x, ref y);
  Console.WriteLine($"x={x}, y={y}");
}
```

Zwróćmy uwagę, że powyższe funkcje różnią się tylko typem danych, na którym działają. Podobna sytuacja może mieć miejsce w przypadku definiowania typów danych. Załóżmy, że chcemy zdefiniować klasę, która może przechować parę wartości:

```
class ParaInt {
  public int Pierwszy {get;}
  public int Drugi {get;}

  public ParaInt(int pierwszy, int drugi) {
    Pierwszy = pierwszy;
    Drugi = drugi;
  }
}

class ParaFloat {
  public float Pierwszy {get;}
  public float Drugi {get;}

  public ParaFloat(float pierwszy, float drugi) {
    Pierwszy = pierwszy;
    Drugi = drugi;
  }
}
```

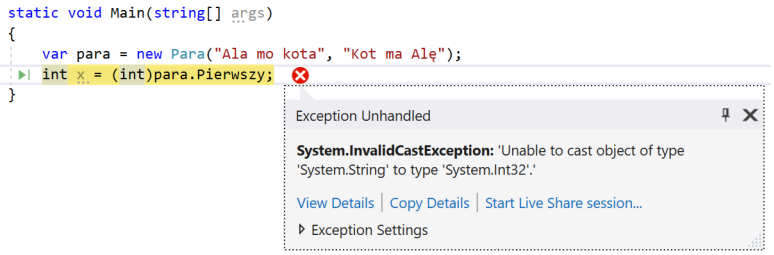
Klasy te również wyglądają bardzo podobnie. Różnią się tylko typami danych i nazwami klas (C# nie pozwala na istnienie w jednej przestrzeni nazw dwóch klas o takiej samej nazwie).

W obu powyższych przykładach nadmiarowość wygenerowanego kodu, aż razi w oczy. Ponadto, jeżeli będziemy potrzebowali podobnej metody, czy klasy działającej na innym, niż do tej pory utworzono, typie danych, musimy ten kod ponownie powielić.

W drugim przypadku, można pomyśleć o ewentualnym obejściu problemu poprzez zastosowanie bazowej dla wszystkich typów .NET klasy object:

```
class Para {
  public object Pierwszy {get;}
  public object Drugi {get;}

  public Para(object pierwszy, object drugi) {
    Pierwszy = pierwszy;
    Drugi = drugi;
  }
}
```



W tym przypadku jednak, może się okazać, że lekarstwo jest gorsze niż choroba. Po pierwsze tworząc taką instancję tej klasy w celu przechowywania danych typów wartościowych (takich jak int, float itd) najpierw C# musi dokonać operacji boxing-u, czyli zamiany danej z typu wartościowego na typ referencyjny. Przy odczycie tej wartości musimy wykonać operację odwrotną tzn. unboxig (zamienić typ referencyjny na wartościowy). Zajmuje to oczywiście czas. Po drugie tracimy możliwość kontroli typów podczas tworzenia aplikacji:

```
var para = new Para("Ala mo kota", "Kot ma Ale");
int x = (int)para.Pierwszy;
```

Oczywiście powyższy kod, jest całkowicie błędny ze względu na typy danych, jakie w nim występują i próba jego wykonania zakończy się otrzymaniem brzydkiego wyjątku:

Straciliśmy więc wszystkie zalety wynikające z faktu, że C# jest językiem o ścisłej kontroli typów.

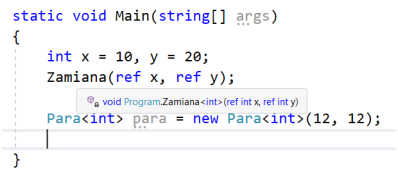
6.1 Programowanie generyczne (uogólnione)

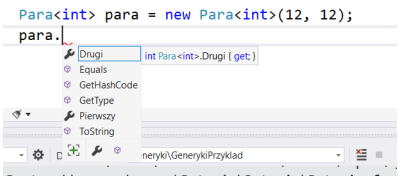
Jeżeli przeanalizujemy powyższe przykłady, dojdziemy do wniosku, że zarówno przedstawiona funkcja, jak i klasa będzie funkcjonować zawsze w ten sam sposób, bez względu na wykorzystane typy danych. Wszystkie nasze problemy zniknęłyby, gdybyśmy mogli te fragmenty kodu napisać w sposób ogólny, nie wymagający podania konkretnych typów danych w momencie tworzenia funkcji lub klasy, ale dopiero podczas ich wykorzystania, podając odpowiednie typy jako parametry. Tym właśnie zajmuje się **programowanie uogólnione**. W różnych językach programowania idea ta określana jest różnymi terminami. W .NET, ale nie tylko, mówimy o generykach i programowaniu generycznym. W językach funkcyjnych (poza F#) określane jest to mianem parametrycznego polimorfizmu, a w C++ i D mówimy o szablonach.

Chcąc powyższe funkcje i klasy przedstawić w formie ogólnej (częściej nazywanej generyczną) zapisalibyśmy je w następujący sposób.

W przypadku funkcji Zamiana:

```
static void Zamiana<T>(ref T x, ref T y)
{
  T tmp = x;
  x = y;
  y = tmp;
```






```
}
```

W przypadku klasy Para:

```
class Para<T>
{
  public T Pierwszy { get; }
  public T Drugi { get; }

  public Para(T pierwszy, T drugi)
  {
    Pierwszy = pierwszy;
    Drugi = drugi;
  }
}
```

Zarówno dla funkcji, jak i klasy w powyższym kodzie pojawił się nowy element. Po nazwach tych elementów (Zamiana i Para) zapisałem w nawiasach ostrych dodatkową zmienną w tym przypadku określaną mianem **zmiennej typu**. Mówi ona, że moja funkcja lub klasa będzie korzystać z pewnego typu, który na tym etapie nie jest jeszcze znany (może nawet nie istnieć), ale będzie podana w momencie wywołania tej metody lub utworzenia instancji klasy:

```
static void Main(string[] args)
{
  int x = 10, y = 20;
  Zamiana<int>(ref x, ref y);

  Para<int> para = new Para<int>(12, 12);
}
```

Zwróćmy uwagę, że podczas zastosowania metody generycznej, oraz utworzenia instancji generycznej klasy w miejsce zmiennej typu, podałem konkretny typ, który w tym przypadku jest mi potrzebny. Określamy go jako **argument typu**.

W przypadku metod C# potrafi najczęściej sam określić dla jakiego typu wygenerować metodę. Pozwala nam to wywołać metodę generyczną dokładnie tak samo jak zwykłą:

```
int x = 10, y = 20;
Zamiana(ref x, ref y);
```

Na poniższych zrzutach ekranu możemy zobaczyć, jak C# widzi wywołanie metody Zamiana oraz właściwości klasy Para<>:

W obu przypadkach podmienił on zdefiniowaną przez nas zmienną typu na podany argument. Pozwala mu to traktować te metody i klasy w taki sposób jakby były utworzone przez nas ręcznie dla konkretnego typu danych.

W razie potrzeby możemy zdefiniować więcej niż jedną zmienną typu. Przykładowo, jeżeli będziemy chcieli zdefiniować naszą klasę Para, tak aby mogła przechowywać wartości różnych typów, możemy to zrobić następująco:

```
class Para<TPierwszy, TDrugi>
{
  public TPierwszy Pierwszy { get; }
  public TDrugi Drugi { get; }

  public Para(TPierwszy pierwszy, TDrugi drugi)
  {
    Pierwszy = pierwszy;
    Drugi = drugi;
  }
}
```

6.2 Ograniczenia typu

Tworząc kod generyczny musimy pamiętać, że musi się on poprawnie kompilować dla dowolnego typu jaki możemy podstawić w miejsce zmiennej typu. Co to oznacza? Przeanalizujmy poniższy kod:

```
class Para<T>
{
  public T Pierwszy { get; }
  public TDrugi { get; }

  public Para(T pierwszy, T drugi)
  {
    Pierwszy = pierwszy;
    Drugi = drugi;
  }

  public T Max() {
    if (Pierwszy < Drugi)
      return Drugi;
    return Pierwszy;
  }
}
```

Niestety kod ten się nie skompiluje, ponieważ nie dla wszystkich typów danych zdefiniowany jest operator <. W tym przypadku otrzymamy następujący błąd:

Poprawa tego błędu wymaga zastosowania innej metody porównania obiektów. W C# możemy to zrobić również za pomocą metody CompareTo interfejsu generycznego IComparable<T>.

```
class Para<T>
{
    public T Pierwszy { get; }
    public T Drugi { get; }

    public Para(T pierwszy, T drugi)
    {
        Pierwszy = pierwszy;
        Drugi = drugi;
    }

    public T Max()
    {
        return Pierwszy < Drugi ? Drugi : Pierwszy;
    }
}
```

T Para<T>.Pierwszy { get; }

CS0019: Operator '<' cannot be applied to operands of type 'T' and 'T'

Załóżmy, że stworzyliśmy klasę Prostokąt:

```
class Prostokat : IComparable<Prosotkat>{
  public int Dlugosc {get;}
  public int Szerokosc {get;}
  public int Pole {get;}

  public Prostokat(int dlugosc, int szerokosc) {
    Dlugosc = dlugosc;
    Szerokosc = szerokosc;
    Pole = dlugosc*szerokosc;
  }

  public CompareTo(Prostokat inny) {
    if (this.Pole < inny.Pole)
      return -1;
    else if (this.Pole > inny.Pole)
      return 1;
    else
      return 0;
  }
}
```

Metoda CompareTo interfejsu IComparable<T> powinna zwracać:

1. wartość mniejszą od zera jeżeli obiekt, na rzecz którego jest wywoływana będzie przez obiektem "inny"w kolejności sortowania (będzie "mniejszy"niż inny)
2. wartość większą od zera jeżeli obiekt, na rzecz którego jest wywoływana będzie za obiektem "inny"w kolejności sortowania (będzie "większy"niż inny)
3. wartość 0 jeżeli obiekty będą takie same (przynajmniej w sensie tej metody)

Ale jak wymusić konieczność implementacji tego interfejsu? Odpowiedzią są tzw. ograniczenia, które możemy nakładać na zmienne typu.

```
class Para<T> where T:IComparable<T>
{
  public T Pierwszy { get; }
  public TDrugi { get; }

  public Para(T pierwszy, T drugi)
```

```
    {
      Pierwszy = pierwszy;
      Drugi = drugi;
    }

    public T Max() {
      if(Pierwszy.CompareTo(Drugi) < 0)
        return Drugi;
      return Pierwszy;
    }
  }
```

Ograniczenia nakładane na zmienne typu wymieniamy po słowie kluczowym where, rozdzielając kolejne ograniczenia tego samego typu po przecinku. Tymi ograniczeniami mogą być:

1. Ograniczenia dziedziczenia - określają po jakim typie powinien dziedziczyć typ, który może być podstawiony w miejsce zmiennej typu lub jakie interfejsy implementować. Przykład tego rodzaju ograniczeń mieliśmy podany powyżej.
2. Ograniczenie konstruktora - określa, czy wymagamy aby typ, który może być podstawiony w miejsce zmiennej typu miał konstruktor domyślny.

   ```
   class Para<T> where T:IComparable<T>,new()
   {...}
   ```

   Uwaga! Ograniczenie konstruktora musi być podawane jako ostatnie na liście ograniczeń
3. Ograniczenia typu - określa, czy typ, który może być podstawiony w miejsce zmiennej typu reprezentował dane typu wartościowego (struct), czy referencyjnego (class). Uwaga! Ograniczenie typu musi być podane jako pierwsze na liście ograniczeń. Ponadto ograniczenie struct nie może być łączone z ograniczeniem new(). Nie jest to konieczne, ponieważ typy wartościowe zawsze mają dostępny konstruktor domyślny.

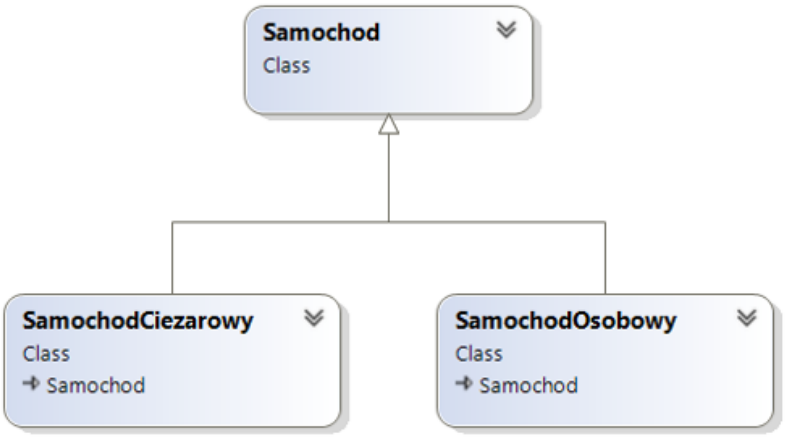
W przypadku, gdy dany kod generyczny ma więcej niż jedną zmienną typu ograniczenia nadajemy osobno na każdą z nich:

```
class Para<TPierwszy, TDrugi>
  where TPierwszy:ICloneable
  where TDrugi:struct
{
  public TPierwszy Pierwszy { get; }
  public TDrugi Drugi { get; }

  public Para(TPierwszy pierwszy, TDrugi drugi)
  {
    Pierwszy = pierwszy;
    Drugi = drugi;
  }
}
```

6.3 Kowariancja, Kontrawariancja i Inwariancja

Załóżmy, że chcemy stworzyć aplikację modelującą jakiś drobny świat fragment świata rzeczywistego: produkcję i serwis samochodów. Zaimplementujemy w niej następującą hierarchię klas:



W kodzie można to przedstawić w następujący sposób:

```
class Samochod
{
}

class SamochodOsobowy : Samochod
{
}

class SamochodCiezarowy : Samochod
{
}
```

Oczywiście w świecie rzeczywistym samochody są produkowane przez różne fabryki. Ponieważ z perspektywy klienta, fabryka jest zakładem, od którego wymagamy wykonania konkretnej usługi (w tym przypadku produkcji auta), to zachowanie to możemy zamodelować za pomoca następującego interfejsu generycznego (w końcu w przyszłości, może będziemy chcieli stworzyć fabryki produkujące inne towary):

```
interface IFabryka<T>
{
  T Stworz();
}
```

R Oczywiście zmienną typu w interfejsie można stosować w dowolnym, ale dozwolonym miejscu np:

```
interfejs IInterfejs<T> {
  T Metoda1();
  void Metoda2(T parametr);
```

```
        T Metoda2(T parametr);
      }
```

Wykorzystując ten interfejs możemy teraz stworzyć trzy fabryki, które będą produkowały nasze samochody:

```
class FabrykaSamochodowOsobowych : IFabryka<SamochodOsobowy>
{
  public SamochodOsobowy Stworz() {
    throw new NotImplementedException();
  }
}

class FabrykaSamochodowCiezarowych : IFabryka<SamochodCiezarowy>
{
  public SamochodCiezarowy Stworz() {
    throw new NotImplementedException();
  }
}

class FabrykaSamochodow : IFabryka<Samochod>
{
  public Samochod Stworz() {
    throw new NotImplementedException();
  }
}
```

Oczywiście, każda z tych klas implementuje interfejs IFabryka dla konkretnego rodzaju samochodów, które produkuje. Przyjąłem założenie, że FabrykaSamochodow może produkować zarówno samochody osobowe, jak i samochody ciężarowe. Wykorzystajmy teraz powyższy kod do stworzenia wirtualnego świata, w którym będziemy potrzebowali samochodu. Taki świat w najprostszy sposób możemy zaimplementować jako klasę. Zawierającą jedno pole prywatne Samochod oraz publiczną metodę PotrzebnySamochod, do której przekażemy fabrykę, która ten samochód ma wyprodukować:

```
class SwiatWirutalny
{
  private Samochod samochod;

  public void PotrzebnySamochod(IFabryka<Samochod> fabryka)
  {
    samochod = fabryka.Stworz();
  }
}
```

Jeżeli przyjrzymy się dokładnie metodzie PotrzebnySamochod dojdziemy do wniosku, że powinna ona poprawnie działać z każdą fabryką, bez względu na to jaki rodzaj samochodu będzie produkować. Jest to oczywiście związane z tym, że pole samochod ma przypisany typ bazowy, więc możemy do niego przypisać zarówno obiekty

tego typu, jak również obiekty typów pochodnych od niego (w końcu zarówno samochód ciężarowy, jak i osobowy jest szczególnym rodzajem samochodu). Idąc dalej tym tokiem rozumowania możemy dojść do wniosku, że implementacja interfejsów IFabryka<SamochodOsobowy> i IFabryka<SamochodCiezarowy> jest szczególnym rodzajem implementacji interfejsu IFabryka<Samochod>. Wobec tego do metody PotrzebnySamochod powinno dać się przekazać instancję każdej z utworzonych fabryk:

```
class Program
{
  static void Main(string[] args)
  {
    var sw = new SwiatWirutalny();

    var fso = new FabrykaSamochodowOsobowych();
    var fsc = new FabrykaSamochodowCiezarowych();
    var fs  = new FabrykaSamochodow();

    sw.PotrzebnySamochod(fso);
    sw.PotrzebnySamochod(fsc);
    sw.PotrzebnySamochod(fs);
  }
}
```

Wprowadzając powyższy kod zauważymy jednak, że tylko ostatnie wywołanie metody PotrzebnySamochod jest poprawne, a pozostałe będą generowały błąd:

```
class Program
{
    static void Main(string[] args)
    {
        SwiatWirutalny sw = new SwiatWirutalny();

        IFabryka<Samochod> fso = new FabrykaSamochodowOsobowych();
        IFabryka<Samochod> fsc = new FabrykaSamochodowCiezarowych();
        IFabryka<Samochod> fs = new FabrykaSamochodow();

        sw.PotrzebnySamochod(fso);
        sw.PotrzebnySamochod(fsc);
        sw.PotrzebnySamochod(fs);

    }
}
```

Dlaczego tak się dzieje? Ponieważ zmienna typu T interfejsu IFabryka<T> jest inwariantna, co oznacza, że do metody PotrzebnySamochod można przekazać tylko obiekty klas, które implementują dokładnie ten interfejs, który jest określony w deklaracji tej metody.

Powyższe błędy można naprawić poprzez zamianę zmiennej typu T interfejsu fabryka w kowariantny:

```
interface IFabryka<out T>
{
  T Stworz();
}
```

Kowariancja ogranicza nam możliwość wykorzystania zmiennej typu do wartości zwracanej z metody lub właściwości (dlatego przed zmienną typu T pojawiło się słowo

kluczowe out). Ale dzięki temu do metody PotrzebnySamochod możemy przekazać instancję klasy, która implementuje określony w jej deklaracji interfejs, lub implementuje ten interfejs dla którego zdefiniowany typ jest typem bazowym.

R **Dlaczego inwariancja jest w tym przypadku problematyczna?**
Wyobraźmy sobie sytuację, w której fabryka może rozpatrywać reklamacje stworzonych pojazdów. Możemy to zamodelować w naszym interfejsie w następujący sposób:

```
interface IFabryka<T>
{
  T Stworz();
  bool RozpatrzReklamacje(T pojazd);
}
```

Oczywiście klasy implementujące ten interfejs muszą posiadać odpowiednią metodę. Dlatego konieczne jest poprawienie naszych fabryk:

```
class FabrykaSamochodowOsobowych
  : IFabryka<SamochodOsobowy>
{
  public SamochodOsobowy Stworz()
    { throw new NotImplementedException(); }

  public bool RozpatrzReklamacje(
    SamochodOsobowy pojazd)
    { throw new NotImplementedException(); }
}

class FabrykaSamochodowCiezarowych
  : IFabryka<SamochodCiezarowy>
{
  public SamochodCiezarowy Stworz()
    { throw new NotImplementedException(); }

  public bool RozpatrzReklamacje(
    SamochodCiezarowy pojazd)
  { throw new NotImplementedException(); }
}

class FabrykaSamochodow : IFabryka<Samochod>
{
  public Samochod Stworz()
    { throw new NotImplementedException(); }

  public bool RozpatrzReklamacje(Samochod pojazd)
  { throw new NotImplementedException(); }
}
```

Jeżeli dalej zmienimy metodę PotrzebnySamochod, aby wykorzystywała nowe możliwości fabryk:

```
class SwiatWirutalny
```

```
{
  private Samochod samochod;

  public void PotrzebnySamochod(
      IFabryka<Samochod> fabryka)
  {
    samochod = fabryka.Stworz();
    if(samochod.PrzetestujSamochod() == false)
      fabryka.RozpatrzReklamacje(samochod);
  }

  private bool PrzetestujSamochod()
      { throw new NotImplementedException(); }
}
```

Czyli w tym przypadku, prosimy fabrykę o stworzenie samochodu, następnie go testujemy i jeżeli test nie przejdzie pomyślnie, to wysyłamy samochód do fabryki z prośbą o rozpatrzenie reklamacji. Przyjrzyjmy się ponownie metodzie PotrzebnySamochod. Parametr tej metody ma typ IFabryka<Samochod>. Wobec tego metoda RozpatrzReklamacje oczekuje instancji klasy Samochod. Wszystko jest w porządku. Przeanalizujmy jednak co by się stało, gdybyśmy przekazali instancję klasy implementującej interfejs IFabryka<SamochodOsobowy>. Metoda RozpatrzReklamacje oczekuje teraz, że otrzyma instancję klasy SamochodOsobowy, ale pole samochod klasy SwiatWirtualny jest ogólnego typu Samochod i kompilator nie ma żadnej gwarancji, że zawsze będzie zawierało wartość typu SamochodOsobowy. A ponieważ do parametru klasy pochodnej nie można podstawić obiektu klasy bazowej więc kod ten nie może się skompilować.

Możemy teraz w naszym świecie wirtualnym otrzymać samochód dowolnego typu. Niestety świat rzeczywisty nie jest idealny i zdarzają się w nim wypadki. Dlatego potrzebujemy serwisu. Możemy go przedstawić w formie interfejsu

```
interface ISerwis<T>
{
  void Napraw(T obiekt);
}
```

który następnie zaimplementujemy w następujący sposób:

```
class SerwisSamochodowOsobowych : ISerwis<SamochodOsobowy>
{
  public void Napraw(SamochodOsobowy obiekt)
  {
    throw new NotImplementedException();
  }
}

class SerwisSamochodowCiezarowych : ISerwis<SamochodCiezarowy>
{
  public void Napraw(SamochodCiezarowy obiekt)
  {
    throw new NotImplementedException();
```

```
    }
  }

  class SerwisSamochodow : ISerwis<Samochod>
  {
    public void Napraw(Samochod obiekt)
    {
      throw new NotImplementedException();
    }
  }
```

Mamy teraz do dyspozycji trzy serwisy. Pierwszy dla samochodów osobowych, drugi dla ciężarowych, a trzeci ogólny (właściciel obiecuje, że naprawi każdy pojazd).

Wysłanie samochodu do serwisu, możemy zaimplementować za pomocą wzorca Wizytator:

```
  class SamochodOsobowy : Samochod
  {
    public void Napraw(ISerwis<SamochodOsobowy> serwis)
    {
      serwis.Napraw(this);
    }
  }

  class SamochodCiezarowy : Samochod
  {
    public void Napraw(ISerwis<SamochodCiezarowy> serwis)
    {
      serwis.Napraw(this);
    }
  }
```

Możemy teraz sprawdzić, czy nasz kod będzie działał poprawnie dla samochodu osobowego. Intuicyjnie poniższy kod powinien działać zarówno dla serwisu samochodów osobowych oraz dla serwisu ogólnego. Natomiast próbując wysłać samochód do serwisu samochodów ciężarowych powinniśmy otrzymać błąd.

```
  class Program
  {
    static void Main(string[] args)
    {
      SamochodOsobowy so = new SamochodOsobowy();

      so.Napraw(new SerwisSamochodowOsobowych());
      so.Napraw(new SerwisSamochodowCiezarowych());
      so.Napraw(new SerwisSamochodow());
    }
  }
```

Niestety C# pozwoli nam przekazać do metody Napraw tylko instancję klasy `SerwisSamochodowOsobowych`. Wyjaśnienie tego faktu, jest identyczne jak w przypadku

interfejsu IFabryka<T>. Interfejs ISerwis<T> dla zmiennej typu T jest inwariantny. Aby aplikacja ta działała poprawnie należy go zmienić w kontrawariantny:

```
interface ISerwis<in T>
{
  void Napraw(T obiekt);
}
```

Musimy pamiętać, że w interfejsie takim zmienną typu T można wykorzystać tylko jako typ parametru metody lub we właściwości tylko do zapisu, co z resztą sugeruje słowo kluczowe in przed nazwą tej zmiennej typu.

6.4 Przydatne typy i metody

Załóżmy, że chcemy zmierzyć czas trwania pewnej operacji w C# . Możemy do tego wykorzystać klasę StopWatch znajdującą się w przestrzeni nazw System.Diagnostics. Robimy to w następujący sposób:

```
using System;
using System.Numerics;
using System.Diagnostics;

namespace Przyklad
{
  class Program
  {
    static BigInteger Silnia(BigInteger n)
    {
      BigInteger silnia = 1;
      for(BigInteger i =1 ; i<n;i++)
      {
        silnia *= i;
      }
      return silnia;
    }

    static void Main(string[] args)
    {
      Stopwatch stoper = new System.Diagnostics.Stopwatch();
      stoper.Start();
      var wynik = Silnia(20000);
      stoper.Stop();
      Console.WriteLine(stoper.ElapsedMilliseconds);
    }
  }
}
```

W powyższej aplikacji stworzyłem metodę statyczną Silnia, a następnie w metodzie Main wykorzystałem klasę StopWatch do zmierzenia czasu trwania tej metody w milisekundach. Jest to najprostszy, ale niestety nie najlepszy sposób na mierzenie czasu. Jest to związane ze sposobem działania tej klasy. Zapisuje ona w pamięci czas, w którym

wywołana została metoda Start oraz metoda Stop. Następnie oblicza różnice pomiędzy nimi. Niestety nie określa to dokładnego czasu wykonania naszej metody, ponieważ klasa StopWatch nie uwzględnia przerw w wykonywaniu naszego kodu.

6.5 Zadania

Zadanie 6.1 Zaimplementuj klasę Para w wersji generycznej i zwykłej. Następnie utwórz instancję obu klas i w pętli wykonaj 100000000 razy dodaj do siebie składowe pary (wynik przypisując do jakieś zmiennej). Zmierz czas wykonania tej pętli dla obu klas. Która wersja jest szybsza?

Zadanie 6.2 Stwórz klasę generyczną, która będzie przechowywała pojedynczą wartość. Typ tej wartości powinien być ograniczony do typu referencyjnego, posiadającego konstruktor domyślny. Inne typy nie są dopuszczalne. Następnie zaimplementuj w tej klasie konstruktor, który pozwoli ustalenie w tej wartości, właściwość która umożliwi jej odczytanie oraz metodę NowaWartosc, która będzie zwracała nową, domyślna instancję tej wartości np:

```
class Osoba
{
   public string Imie {get; set;}
   public string Nazwisko {get;set;}
}
var klasa = new Klasa<Osoba>(
  new Osoba() {Imie = "Ala", Nazwisko="Kot"}
));
Osoba osoba = Klasa.Wartosc;
Osoba nowaOsoba = Klasa.NowaWartosc();
```

Zadanie 6.3 Załóżmy, że chcielibyśmy teraz rozszerzyć poprzedni przykład tak, aby klasa generyczna mogła zwrócić dokładną kopię przechowywanej wartości. Jakich zmian należy dokonać i gdzie?

Zadanie 6.4 Napisz klasę Osoba, która będzie implementowała następujące właściwości: imię, nazwisko, wiek. Następnie zaimplementuj w niej interfejs IComparable, który pozwoli sortować instancje tej klasy wg. wieku w kolejności rosnącej.

Zadanie 6.5 Stwórz generyczną strukturę danych reprezentującą uporządkowane drzewo binarne. Przetestuj działanie tej struktury na liczbach całkowitych oraz na instancjach klasy Osoba.

Zadanie 6.6 Stwórz kolekcję, która będzie pozwalała na przechowywanie obiektów różnych typów pod określoną nazwą (powinno to być coś w rodzaju słownika). Kolekcja ta powinna posiadać metodę dodaj pozwalającą dodać nową pozycję oraz generyczną metodę Pobierz, która pozwoli na zwrócenie wartości konkretnego typu np:

```
var kolekcja  = new Kolekcja();
kolekcja.Dodaj("pierwsza", 12);
kolekcja.Dodaj("druga", 12.3);
kolekcja.Dodaj("trzecia", "Ala");
```

```
kolekcja.Pobierz<int>("pierwsza");
```

Zadanie 6.7 Przyjmijmy, że w aplikacji dana jest następująca hierarchia klas:

```
class Osoba
{
}

class Student : Osoba
{
}

class Pracownik : Osoba
{
}
```

Zdefiniuj odpowiednie typy, tak aby poniższy kod poprawnie się kompilował:

```
Pobieranie<Osoba> magazyn1A = new MagazynDanychoStudentach();
Pobieranie<Osoba> magazyn2A = new MagazynDanychOPracownikach();
Pobieranie<Osoba> magazyn3A = new MagazynDanychoOsobach();

Przechowywanie<Student> magazyn1B = new MagazynDanychoOsobach();
Przechowywanie<Student> magazyn2B = new MagazynDanychoStudentach();
```