Programowanie funkcyjne

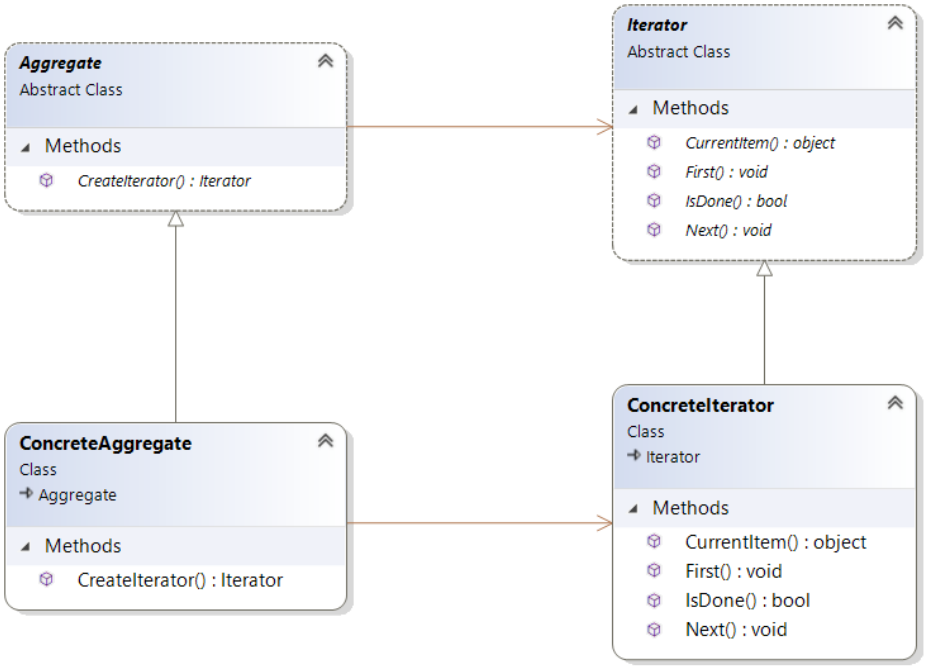
Materiały z wykładu i laboratorium

Łukasz Bartczuk

9. Podstawy programowania funkcyjnego w C# cz.3

9.1 Iteratory

Wzorzec iterator służy do sekwencyjnego poruszania się po dowolnej kolekcji obiektów. Dzięki niemu możemy ukryć wewnętrzną strukturę kolekcji oraz ujednolicić sposób odczytywania kolejnych jej elementów za wspólnym interfejsem. W książce ”Bandy czworga” jest on przedstawiony za pomocą następujących interfejsów i klas:



Abstrakcyjna klasa (lub interfejs) Iterator definiują konieczne metody do poruszania się po i dostępu do elementów danej kolekcji. Klasa ConcreteIterator implementuje ten interfejs dla konkretnej kolekcji. Aggeragate definiuje metodę tworzącą i zwracającą iterator. Interfejs ten powinien być implementowany przez kolekcję, aby ułatwić dostęp do przeznaczonego dla niej iteratora.

W przypadku platformy .NET wzorzec ten jest zdefiniowany za pomocą interfejsów IEnumerable i IEnumerator. Odpowiadają wspomnianym wcześniej klasom abstrakcyjnym Aggregate oraz Iterator. Zaimplementowane są zarówno w wersji generycznej, jak i niegenerycznej:

```
public interface IEnumerator
{
  object? Current { get; }
  bool MoveNext();
  void Reset();
}

public interface IEnumerator<out T> : IEnumerator, IDisposable
{
  T Current { get; }
}

public interface IEnumerable
{
  IEnumerator GetEnumerator();
}

public interface IEnumerable<out T> : IEnumerable
{
  IEnumerator<T> GetEnumerator();
}
```

Zdefiniowane w nich właściwości i metody odpowiadają poszczególnym metodom zdefiniowanym przez ”Bandę czworga”, przy czym metoda Reset odpowiada metodzie First, a metoda MoveNext łączy w sobie działanie metod Next i IsDone (przechodzi do następnego elementu kolekcji i zwraca wartość logiczną określającą, czy istnieje następny element w kolekcji czy nie). Powyższe interfejsy są implementowane przez wszystkie kolekcje w .NET, co umożliwia jednolity, sekwencyjny dostęp do wszystkich elementów bez względu na rodzaj kolekcji. Przykładowo tworząc listę:

```
var lista = new List<int> {1,2,3,4,5}
```

możemy po niej iterować:

```
var iterator = lista.GetEnumerator();
while(iterator.MoveNext())
{
  var el = iterator.Current;
  Console.WriteLine(el);
}
```

Gdybyśmy teraz zmienili tę listę na tablicę:

```
var lista = new [] { 1, 2, 3, 4, 5 };
```

sposób iterowania po niej się nie zmieni i będzie dokładnie taki sam jak dla listy.

Bez względu jednak na rozdaj kolekcji, wykorzystując iterator zawsze musimy najpierw wywołać metodę GetEnumerator w celu pobrania iteratora, a następnie zapisać odpowiednią pętlę. Dlatego też twórcy C# chcąc nam ułatwić życie, stworzyli specjalny rodzaj pętli do obsługi iteratorów:

```
foreach(var el in lista)
  Console.WriteLine(el);
```

Jej sposób działania jest dokładnie taki sam, jak na przedstawionym wcześniej fragmencie wykorzystującym pętlę while.

Gdybyśmy chcieli zaimplementować własną kolekcję, powinniśmy ją wyposażyć w implementację iterfejsu IEnumerable<T> oraz stworzyć odpowiedni iterator. Przykładowo dla listy łączonej zadefiniowanej w znany już wam sposób:

```
class Lista<T> : IEnumerable<T>
{
  public T Wartosc { get; }
  public Lista<T> Nastepny { get; }

  public Lista(T wartosc, Lista<T> nastepny)
  {
    Wartosc = wartosc;
    Nastepny = nastepny;
  }

  public IEnumerator<T> GetEnumerator()
    => new ListaIterator<T>(this);

  IEnumerator IEnumerable.GetEnumerator()
    => GetEnumerator();
}
```

definicja iteratora może wyglądać w następujący sposób:

```
class Iterator<T> : IEnumerator<T>
{
  public T Current
  {
    get
    {
      if (wezel == null)
        return default;
      return wezel.Wartosc;
    }
  }

  object IEnumerator.Current => Current;
```

```
    public Iterator(Lista<T> poczatek)
    {
      wezel = null;
      this.poczatek = poczatek;
    }

    public bool MoveNext()
    {
      wezel = wezel == null ? poczatek : wezel.Nastepny;
      return wezel != null;
    }

    public void Reset()
    {
      wezel = null;
    }

    public void Dispose()
    {
    }

    private Lista<T> wezel;
    private Lista<T> poczatek;
  }
```

Konstruktor tej klasy jako swój argument przyjmuje węzeł listy, od którego chcemy rozpocząć iterację po elementach. Właściwości Current zwracają wartość przechowywaną w obecnie wybranym elemencie listy. Jeżeli aktualny wezeł wskazuje na null, to właściwość ta zwraca wartość domyślną dla danego typu. Metoda MoveNext przesuwa iterator do przodu. Jeżeli aktualnie pokazuje on na null to jest ustawiany na element zawarty w polu poczatek, w przeciwnym przypadku na następny element listy. Metoda ta zwraca wartość true, jeżeli po przesunięciu iteratora pokazuje on na jakiś element lub false, gdy wskazuje na null (co oznacza, że doszliśmy do ostatniego elementu listy lub lista była pusta). Metoda Reset ustawia pole wezel na null, co sprowadza listę do stanu początkowego. W tym przypadku metoda Dispose nie pełni żadnej roli, ale ponieważ jest wymagana przez interfejs IEnumerable<T> musimy zapisać jej definicję.

Wykorzystanie tak zdefiniowanego iteratora pokazuje poniższy przykład:

```
  var lista = Wezel(1, Wezel(2, Wezel(3, Wezel(4, null))));

  foreach (var el in lista)
    Console.WriteLine(el);
```

Zwróćmy uwagę, że sposób jego wykorzystania jest dokładnie taki sam, jak w przypadku kolekcji wbudowanych.

9.2 Generatory

W poprzednim punkcie zdefiniowaliśmy osobne klasy, które implementowały interfejsy IEnumerable<> i IEnumerator<>. Nic nie stoi na przeszkodzie, aby oba te interfejsy były implementowane przez jedną klasę. Klasy takie, w niektórych językach programowania, określa się mianem **generatorów**. Są one bardzo przydatne do tworzenia tzw. **kolekcji leniwych**, w których elementy nie są tworzone od razu, ale dopiero w momencie, gdy są potrzebne. Przykładowo jako generator, moglibyśmy zapisać algorytm Fibonnaciego:

```
class Fibonacci :
  IEnumerator<Tuple<int, BigInteger>>,
  IEnumerable<Tuple<int, BigInteger>>
{
  private BigInteger f_1 = 0;
  private BigInteger v = 0;
  private int element = -1;

  public Tuple<int, BigInteger> Current
     => Tuple.Create(element,v);

  object IEnumerator.Current
     => Tuple.Create(element, v);

  public bool MoveNext()
  {
    element++;
    switch (element)
    {
      case 0: v = 0; break;
      case 1: v = 1; f_1 = 0; break;
      default:
        var tmp = f_1;
        f_1 = v;
        v = f_1 + tmp;
      break;
    }
    return true;
  }

  public void Reset()
  {
    f_1 = 0;
    element = -1;
  }

  public void Dispose() { }

  public IEnumerator<Tuple<int, BigInteger>> GetEnumerator()
    => this;

  IEnumerator IEnumerable.GetEnumerator()
```

```
        => this;
    }
```

Klasa ta umożliwia wygenerowanie dowolnej liczby elementów ciągu Fibonnaciego rozpoczynając zawsze od pierwszego elementu tego ciągu. Każdy wygenerowany obiekt, to para, która jako pierwszy składnik indeks elementu w ciągu, a jako drugi sam element. Cała magia tej klasy ukryta jest w metodzie MoveNext, która jest odpowiedzialna za wyznaczenie wartości następnego elementu ciągu. Myślę, że bez problemu zrozumiecie działanie tej metody. Uwagę zwróćmy tylko na wartość zwracaną z funkcji. Ustawiłem ją na stałe na true, ponieważ ciąg ten nigdy się nie kończy (zawsze da się wyznaczyć następną wartość ciągu, o ile oczywiście pamięć pozwoli). Można powiedzieć, że zastosowanie w tym przypadku generatora pozwala na utworzenie leniwej kolekcji o nieskończonej liczbie elementów.

Poniższy kod pokazuje sposób wykorzystania tej klasy:

```
var fib = new Fibonacci();
foreach(var f in fib)
{
  if(f.Item1 < 10)
    Console.WriteLine($"{f.Item1} {f.Item2}");
  else
    break;
}
```

Pobiera on 10 pierwszych elementów ciągu, przy czym nie są one generowane od razy wszystkie, tylko każdy z nich w kolejności dopiero, gdy jest potrzebny.

9.3 Funkcje (metody) iteratorowe

Jak wcześniej wspomniałem, większość magii klasy Fibonacci zawarta jest w metodzie MoveNext. Jednak oprócz niej trzeba było zaimplementować cały zestaw innych metod i właściwości, które służą do obsługi iteratora lub kolekcji policzalnych. Części z nich w tym przypadku nawet nie implementowaliśmy, a część, takich jak metody GetEnumerator, będą takie same w każdym generatorze. Czy wobec tego nie można generatora zapisać prościej i zrzucić część pracy na kompilator? Okazuje się, że można. Trzeba w tym celu wykorzystać funkcje iteratorowe. Funkcje takie muszą spełniać jednak kilka ograniczeń:

1. muszą zwracać wartość jednego z typów: IEnumerable, IEnumerable<T>, IEnumerator lub IEnumerator<T>, gdzie T jest oczywiście typem elementu w kolekcji po której chcemy iterować
2. nie mogą przyjmować parametrów oznaczonych jako in, ref ani out.
3. do zwrócenia wartości stosujemy instrukcje yield return po którym musi pojawić się wyrażenie tworzące wartość typu T, a do przerwania funkcji yield break. Można oczywiście korzystać ze słowa kluczowego return, ale w takim przypadku musimy pamiętać, aby wyrażenie, które po nim zapiszemy zwracało wartość zgodną z interfejsem IEnumerable<T>.

Najprostszym przykładem takiej funkcji jest:

```
public static IEnumerable<int> FunkcjaIteratorowa()
{
  yield return 1;
  yield return 2;
  yield return 3;
}
```

W tym przypadku funkcja ta kolejno, przy każdym wywołaniu, wygeneruje kolejną wartość 1,2 lub 3. Stosujemy ją podobnie jak inne iteratory:

```
foreach(var i in FunkcjaIteratorowa())
  Console.WriteLine(i)
```

Jeżeli w Visual Studio ustawimy punkt przerwania (ang. *breakpoint*) na instrukcji foreach i wykonamy wykonywania krok po kroku, będziemy mieli wrażenie, że funkcja ta jest wywoływana 4 razy, przy czym każde kolejne jej wywołanie nie spowoduje jej wykonania od początku, tylko od miejsca w którym zakończyło się poprzednie wywołanie.

Na podstawie kodu metody FunkcjaIteratorowa, kompilator generuje zagnieżdżoną klasę, której kod można podejrzeć np. za pomocą oprogramowania dotPeek. Wygenerowany kod przedstawia się następująco:

```
[CompilerGenerated]
private sealed class <FunkcjaIteratorowa>d__0 : IEnumerable<int>, IEnumerable,
    IEnumerator<int>, IEnumerator, IDisposable
{
  private int <>1__state;
  private int <>2__current;
  private int <>l__initialThreadId;

  [DebuggerHidden]
  public <FunkcjaIteratorowa>d__0(int _param1)
  {
    base..ctor();
    this.<>1__state = _param1;
    this.<>l__initialThreadId =
      Environment.CurrentManagedThreadId;
  }

  [DebuggerHidden]
  void IDisposable.Dispose()
  {
  }

  bool IEnumerator.MoveNext()
  {
    switch (this.<>1__state)
    {
      case 0:
```

```
          this.<>1__state = -1;
          this.<>2__current = 1;
          this.<>1__state = 1;
          return true;
        case 1:
          this.<>1__state = -1;
          this.<>2__current = 2;
          this.<>1__state = 2;
          return true;
        case 2:
          this.<>1__state = -1;
          this.<>2__current = 3;
          this.<>1__state = 3;
          return true;
        case 3:
          this.<>1__state = -1;
          return false;
        default:
          return false;
      }
    }

    int IEnumerator<int>.Current
    {
      [DebuggerHidden] get
      {
        return this.<>2__current;
      }
    }

    [DebuggerHidden]
    void IEnumerator.Reset()
    {
      throw new NotSupportedException();
    }

    object IEnumerator.Current
    {
      [DebuggerHidden] get
      {
        return (object) this.<>2__current;
      }
    }

    [DebuggerHidden]
    IEnumerator<int> IEnumerable<int>.GetEnumerator()
    {
      Program.<FunkcjaIteratorowa>d__0 funkcjaIteratorowaD0;
      if (this.<>1__state == -2 && this.<>l__initialThreadId ==
         Environment.CurrentManagedThreadId)
      {
```

```
        this.<>1__state = 0;
        funkcjaIteratorowaD0 = this;
      }
      else
        funkcjaIteratorowaD0
          = new Program.<FunkcjaIteratorowa>d__0(0);
      return (IEnumerator<int>) funkcjaIteratorowaD0;
    }

    [DebuggerHidden]
    IEnumerator IEnumerable.GetEnumerator()
    {
      return (IEnumerator) this.System.Collections.Generic.IEnumerable<System.Int32>.
          GetEnumerator();
    }
  }
```

Jak widzimy, kompilator wykonał całkiem dużo pracy i stworzył kod podobny do tego jaki wygenerowalibyśmy ręcznie. Oczywiście zmianie musi również podlegać sama metoda FunkcjaIteratorowa, która po kompilacji przyjmuje postać:

```
  [IteratorStateMachine(typeof (Program.<FunkcjaIteratorowa>d__0))]
  private static IEnumerable<int> FunkcjaIteratorowa()
  {
    return (IEnumerable<int>) new Program.<FunkcjaIteratorowa>d__0(-2);
  }
```

Jak widzimy metoda ta tworzy i zwraca obiekt naszego generatora.

Funkcję iteratorową możemy również wykorzystać do stworzenia iteratora poruszającego się po kolekcji. Poniższy kod klasy Lista zawiera dwie dodatkowe metody PobierzWszystkie oraz PobierzTylko2, które tworzą odpowiednie iteratory. Pierwszy, który przechodzi po wszystkich elementach kolekcji oraz drugi, który zwróci co najwyżej dwa elementy i przerwie dalszą iterację (pokazując przy okazji zastosowanie instrukcji yield break).

```
  class Lista<T> : IEnumerable<T> {
    public T Wartosc { get; }
    public Lista<T> Nastepny { get; }

    public Lista(T wartosc, Lista<T> nastepny) {
      Wartosc = wartosc;
      Nastepny = nastepny;
    }

    public IEnumerable<T> PobierzWszystkie() {
      var obecny = this;
      while (obecny != null) {
        yield return obecny.Wartosc;
        obecny = obecny.Nastepny;
      }
    }
```

```
    public IEnumerable<T> PobierzTylko2()
    {
      var obecny = this;
      int i = 0;
      while (obecny != null)
      {
        if (i == 2)
          yield break;
        yield return obecny.Wartosc;
        obecny = obecny.Nastepny;
        i++;
      }
    }
  }
```

Oczywiście, powyższa klasa się nam nie skompiluje ponieważ klasa ta nie implementuje metod wymaganych przez interfejs IEnumerable<T>. Aby to naprawić należy zmienić deklarację metody PobierzWszystkie na: `public IEnumerator<T> GetEnumerable()` oraz dopisać niegeneryczną metodę GetEnumerable().

Na koniec tego punktu zobaczmy jeszcze jak wyglądać będzie funkcja iteratorowa stworzona do generowanie kolejnych elementów ciągu Fibonacciego:

```
  public static IEnumerable<Tuple<int,BigInteger>> Fib() {
    BigInteger f_1 = 0;
    BigInteger v = 0;
    int element = 0;

    do {
      switch(element)
      {
        case 0: v = 0; break;
        case 1: v = 1; f_1 = 0; break;
        default:
            BigInteger tmp = f_1;
          f_1 = v;
          v = tmp + f_1;
        break;
      }
      yield return Tuple.Create(element++, v);
    }
    while (true);
  }
```

Analizę kodu tej funkcji pozostawiam jednak do samodzielnego rozpatrzenia.

9.4 Metody rozszerzające

Dodania nowej funkcjonalności do danej klasy możemy wykonać na kilka sposobów:

1. dodanie nowej metody do istniejącej klasy, lub

2. utworzenia klasy pochodnej, która będzie zawierała nową funkcjonalność

Co jednak, gdy żadna z tych metod nie będzie możliwa co wykorzystania. Przykładowo możemy chcieć stworzyć nową funkcjonalność dla klasy String. W tym przypadku, żadna z powyższych metod się nie sprawdzi ponieważ nie będziemy rekompilować platformy .NET, a klasa string jest klasą zapieczętowaną i nie możemy po niej dziedziczyć. W takim przypadku pozostanie nam tylko stworzyć klasę, która będzie zawierała statyczną metodę realizującą wymaganą funkcjonalność. Przykładem wręcz akademickim może być wybranie co drugiego znaku z danego łańcucha. Możemy to zrealizować w następujący sposób:

```
class MyString {

  public static string CoDrugi(string text)
  {
    var sb = new StringBuilder();
    for(int i=0; i< text.Length; i++)
    {
      if(i%2 == 0) sb.Append(text[i]);
    }
    return sb.ToString();
  }

}
```

Pomysł generalnie prosty, ale niestety wykorzystanie takiej metody nie należy do najwygodniejszych:

```
string text = "Ala ma kota";
MyString.CoDrugi(text);
```

Po pierwsze musimy podawać nazwę klasy, z której umieściliśmy daną metodę. A po drugie, ponieważ jest to metoda statyczna wywołujemy ją inaczej niż w przypadku zwykłych metod instancji. Z pierwszym problemem moglibyśmy sobie łatwo poradzić tworząc, klasę MyString jako statyczną, co pozwoli nam na zastosowanie instrukcji using static, jednak drugi będzie wymagał trochę innego podejścia.

Jego rozwiązaniem są tzw. metody rozszerzające, które pozwalają dodać nową funkcjonalność do dowolnego typu .NET, nawet takiego, który zwykle metod nie implementuje (np. interfejsy). Metody takie muszą spełniać kilka warunków:

1. muszą być umieszczone w statycznej klasie
2. muszą być statyczne (co wymusza poprzedni warunek)
3. przed pierwszym parametrem tej metody musi pojawić się słowo kluczowe this. To ono wskazuje, możliwości którego typu chcemy rozszerzyć. Nie łączy się ono ze słowami kluczowymi in, ref oraz out.

Chcąc zmienić statyczną metodę CoDrugi w metodę rozszerzającą powinniśmy wygenerować następujący kod:

```
static class MyString {
```

```
    public static string CoDrugi(this string text)
    {
      var sb = new StringBuilder();
      for(int i=0; i< text.Length; i++)
      {
        if(i%2 == 0) sb.Append(text[i]);
      }
      return sb.ToString();
    }

  }
```

Działania te powodują, że możemy teraz wywołać tę metodę jak zwykłą metodę statyczną (jak pokazano powyżej) lub jak metodę instancji:

```
string text = "Ala ma kota";
text.CoDrugi();
```

Zwróćmy uwagę na dwa aspekty powyższego fragmentu. Po pierwsze wywołując metodę rozszerzającą nie musimy podawać nazwy klasy (dzięki czemu może ona być dowolna), a po drugie wywołując metodę rozszerzającą nie podajemy już jej pierwszego argumentu, ponieważ ten występuje przed kropką.

Jak wspomniałem wcześniej możemy rozszerzyć dowolny typ danych np. interfejs. Przykładowo, jeżeli będziemy chcieli stworzyć metodę, która wybierze co drugi element z kolekcji policzalnej możemy to zrobić w następujący sposób:

```
static class Rozszerzenia
{
  public static IEnumerable<T> CoDrugi<T>(
    this IEnumerable<T> kolekcja)
  {
    var listaWynikowa = new List<T>();
    int i = 0;
    foreach (var e in kolekcja)
      if (i % 2 == 0)
      {
        listaWynikowa.Add(e);
        i++;
      }
    return listaWynikowa;
  }
}
```

lub nawet prościej stosując kolekcję leniwą:

```
static class Rozszerzenia
{
  public static IEnumerable<T> CoDrugi<T>(
    this IEnumerable<T> kolekcja)
  {
    int i = 0;
    foreach (var e in kolekcja)
```

```
        if (i % 2 == 0)
        {
          yield return e;
          i++;
        }
    }
  }
```

Dzięki temu, że rozszerzyłem interfejs IEnumerable<T>, automatycznie udostępniłem tę metodę we wszystkich kolekcjach, które ten interfejs implementują.

9.5 Zadania do samodzielnego wykonania

Zadanie 9.1 Napisz generator, który będzie pozwalał wczytywać liczby z klawiatury i zwracał je w formie leniwej kolekcji nieskończonej. Następnie skorzystaj z tego generatora do wyświetlania tych wartości w konsoli w następujący sposób:

```
static void Main(string[] args)
{
  foreach (var x in PobierzLiczby())
    Console.WriteLine(x);
}
```

Zadanie 9.2 Napisz funkcję, która umożliwi wyświetlanie tylko wartości parzystych, a nieparzyste będzie pomijała. Wywołanie tych funkcji powinno wyglądać następująco:

```
static void Main(string[] args)
{
  foreach (var x in TylkoParzyste(PobierzLiczby()))
    Console.WriteLine(x);
}
```

Zadanie 9.3 Napisz funkcję, która umożliwi wyświetlanie tylko wartości, które spełniają określony warunek określony odpowiednim predykatem. Powinna to być funkcja wyższych rzędów.

Zadanie 9.4 Napisz powyższą funkcję jako generyczną, tak aby mogła współpracować z policzalnymi kolekcjami dowolnego typu

Zadanie 9.5 Napisz funkcję, która umożliwi podniesienie każdej wczytanej wartości do kwadratu. Wywołanie tej funkcji powinno wyglądać następująco:

```
static void Main(string[] args)
{
  foreach (var x in DoKwadratu(PobierzLiczby()))
    Console.WriteLine(x);
}
```

Zadanie 9.6 Napisz funkcję, która możliwi dowolną modyfikacje każdej wczytanej wartości. Powinna to być funkcja wyższych rzędów.

Zadanie 9.7 Napisz powyższą funkcję jako generyczną, tak aby mogła współpracować z policzalnymi kolekcjami dowolnego typu

Zadanie 9.8 Napisz funkcję, która pozwoli wyświetlać te wartości w konsoli. Zastosowanie tej funkcji powinno wyglądać w następujący sposób:

```
static void Main(string[] args)
{
  Wyswietl(DoKwadratu(PobierzLiczby()));
}
```

Zadanie 9.9 Napisz funkcję, która pozwoli podać użytkownikowi tylko n wartości z klawiatury, a następnie przerwie wykonywanie programu. Jej wywołanie powinno wyglądać następująco:

```
static void Main(string[] args)
{
  Wyswietl(TylkoN(DoKwadratu(PobierzLiczby()), 5));
}
```

Co możesz zaobserwować?

Część dla chętnych!! Jak należy zmienić program, aby uniknąć tego efektu.

Zadanie 9.10 Napisz funkcję, umożliwi przerwanie wczytywania wartości z klawiatury, w momencie gdy użytkownik poda wartość ujemną. Wywołanie tej funkcji powinno być następujące:

```
static void Main(string[] args)
{
  Wyswietl(StopJesliUjemna(PobierzLiczby()));
}
```

Zadanie 9.11 Napisz funkcję, który obliczy ile takich nieujemnych wartości wprowadził użytkownik. Jej wywołanie powinno być następujące:

```
static void Main(string[] args)
{
  Console.WriteLine(LiczbaLiczb(StopJesliUjemna(PobierzLiczby())));
}
```

Co się stanie jeżeli z powyższego wywołania usuniemy funkcję StopJesliUjemna?

Zadanie 9.12 Napisz funkcję, który obliczy sumę nieujemnych wartości wprowadzonych przez użytkownika. Jej wywołanie powinno być następujące:

```
static void Main(string[] args)
{
  Console.WriteLine(Sumuj(StopJesliUjemna(PobierzLiczby())));
}
```

Zadanie 9.13 Napisz funkcję, która będzie zapamiętywała wartości wprowadzone przez użytkownika. Ich wyświetlenie powinno nastąpić jeżeli użytkownik wprowadzi wartość ujemną. Jej zastosowanie to:

```
 static void Main(string[] args)
{
  Wyswietl(Buforuj(StopJesliUjemna(PobierzLiczby())));
}
```

Rysunek 9.1: Przykładowe wykonanie aplikacji z zadania 4.13

Przykładowe wykonanie tego programu przedstawione jest na rys. 9.1.

Zadanie 9.14 Napisz funkcję, która będzie zapamiętywała wartości wprowadzone przez użytkownika. Następnie wyświetl je w odwrotnej kolejności.

Zadanie 9.15 W instrukcji pokazałem w jaki sposób stworzyć iterator przechodzący po liście w przód. Rozszerz jego możliwości tak, aby możliwe było również cofnięcie się na wcześniej już odwiedzony element.

Zadanie 9.16 Na wcześniejszych zajęciach miałeś stworzyć program implementujący generyczne drzewo uporządkowane. Napisz iteratory, które umożliwią przejście po wszystkich elementach tego drzewa w kolejnościach preorder, inorder oraz postorder. Informacje na ich temat możesz znaleźć np. na stronie: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/.

Zadanie 9.17 Napisz funkcje z zadań 4.1 - 4.17 jako metody rozszerzające. Zastanów się jaki typ należy rozszerzyć, aby możliwe było zapisanie kodu np. z zadania 4.10 w następujący sposób:

```
static void Main(string[] args)
{
  PobierzLiczby().StopJesliUjemna().Wyswietl();
}
```

Zadanie 9.18 Napisz program, który będzie pozwalał wydawać polecenia do obsługi listy. Przykładowe polecenia jakie powinny być obsługiwane przez aplikację to: dodaj, usun, pokaz, koniec. Komenda ”dodaj” ma jeden parametr i jest to wartość np. numeryczna, którą należy dodać na listę; ”usun” - usuwa ostatnią dodaną wartość, ”pokaz” - pokazuje wszystkie elementy listy; ”koniec” - kończy program. Funkcja main tej aplikacji powinna wyglądać następująco:

```
static void Main(string[] args)
{
  WczytajKomende().Wykonaj();
}
```

Przykładowa sesja pracy z aplikacją pokazana jest poniżej:

Rysunek 9.2: Przykładowe wykonanie aplikacji z zadania 4.18

W aplikacji należy wykorzystać wzorzec komenda oraz iteratory.

Zadanie 9.19 Rozszerz tą aplikację, aby możliwe było zdefiniowanie kalkulatora bazującego na stosie. Zaimplementuj operacje dodawania, odejmowania, mnożenia, dzielenia oraz funkcje sin, cos, sqrt, ln.