



OBLICZENIA LENIWE

Obliczenia zachłanne (eager computation)

Wyrażenie jest obliczane, gdy tylko jest przypisane do zmiennej

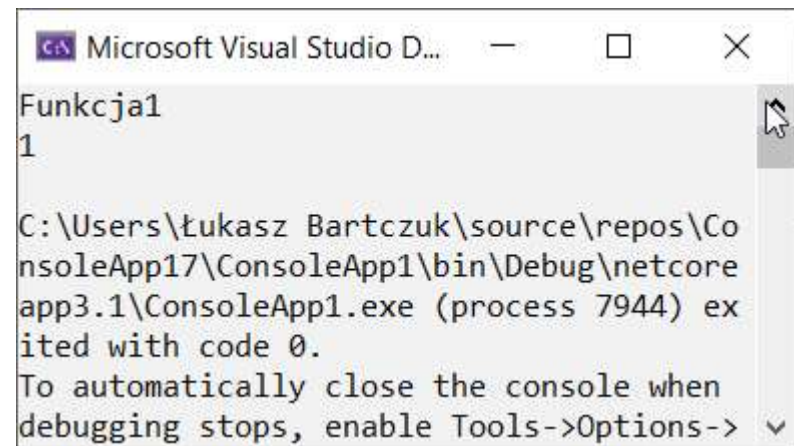
Wartości argumentów są obliczane przed wywołaniem funkcji

Prosty przykład

```
static int Funkcja1() {  
    Console.WriteLine(nameof(Funkcja1));  
    return 1;  
}
```

```
static int Funkcja2() {  
    Console.WriteLine(nameof(Funkcja2));  
    return 2;  
}
```

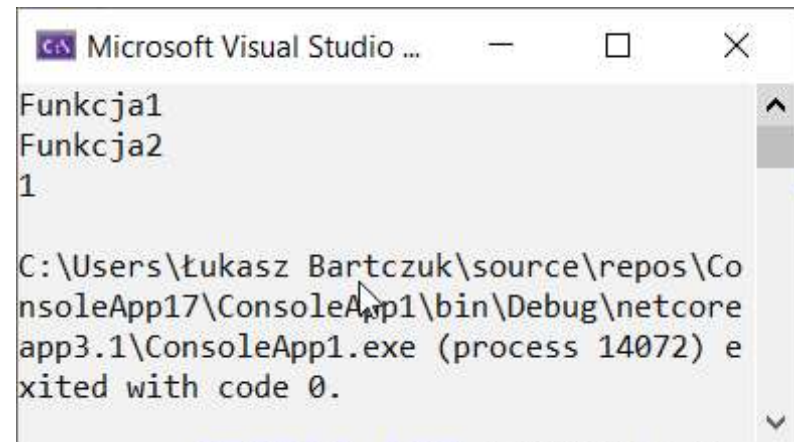
```
static void Main(string[] args) {  
    var wynik = true ? Funkcja1() : Funkcja2();  
    Console.WriteLine(wynik);  
}
```



Prosty przykład

```
static int Funkcja3(  
    bool warunek,  
    int wartosc1,  
    int wartosc2) {  
    return warunek ? wartosc1 : wartosc2;  
}
```

```
static void Main(string[] args) {  
    var wynik = Funkcja3(true, Funkcja1(), Funkcja2());  
    Console.WriteLine(wynik);  
}
```



Zachłanne i leniwe obliczenia

```
static int Funkcja1() { var x=0; return 1/x };
```

```
static int Funkcja2() => 2;
```

```
static int Funkcja3(  
    bool warunek,  
    int wartosc1,  
    int wartosc2) {  
    return warunek ? wartosc1 : wartosc2;  
}
```

```
static void Main(string[] args) {  
    var wynik = Funkcja3(true, Funkcja1(), Funkcja2());  
    Console.WriteLine(wynik);  
}
```



Zachłanne i leniwe obliczenia

```
static int Funkcja1() { var x=0; return 1/x };
```

```
static int Funkcja2() => 2;
```

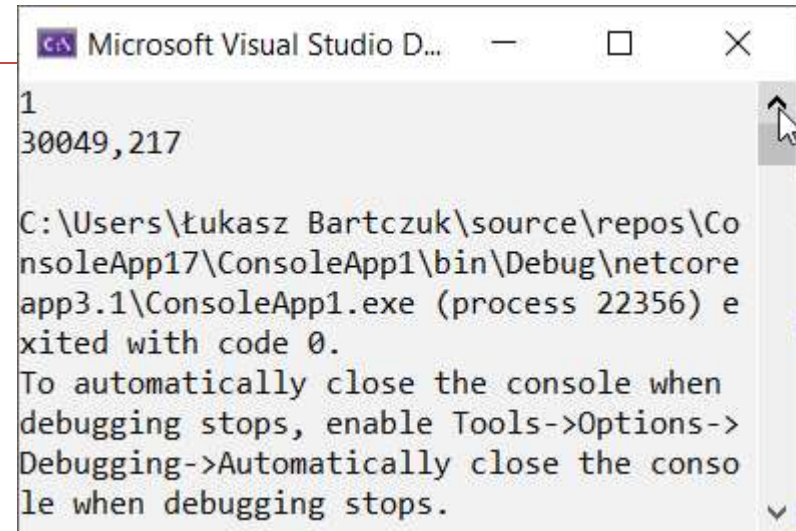
```
static int Funkcja3(  
    bool warunek,  
    int wartosc1,  
    int wartosc2) {  
    return warunek ? wartosc1 : wartosc2;  
}
```

```
static void Main(string[] args) {  
    var wynik = Funkcja3(false, Funkcja1(), Funkcja2());  
    Console.WriteLine(wynik);  
}
```



Zachłanne i leniwe obliczenia

```
static int Funkcja1() {  
    Thread.Sleep(20000); return 1;  
}  
  
static int Funkcja2() {  
    Thread.Sleep(10000); return 2;  
}  
  
static void Przyklad() {  
    var wynik = Funkcja3(false, Funkcja1(), Funkcja2());  
    Console.WriteLine(wynik);  
}  
  
static void Main(string[] args) {  
    Console.WriteLine(ZmierzCzas(Przyklad));  
}
```



The screenshot shows a Visual Studio console window titled "Microsoft Visual Studio D...". The output text is as follows:

```
1  
30049,217  
  
C:\Users\Łukasz Bartczuk\source\repos\ConsoleApp17\ConsoleApp1\bin\Debug\netcoreapp3.1\ConsoleApp1.exe (process 22356) exited with code 0.  
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
```

Obliczenia leniwe (lazy computation)

Obliczenia leniwe wykonywane są tylko wtedy, gdy są konieczne

Wartości argumentów **nie** są obliczane przed wywołaniem funkcji, tylko w trakcie jej działania



Leniwe obliczenia

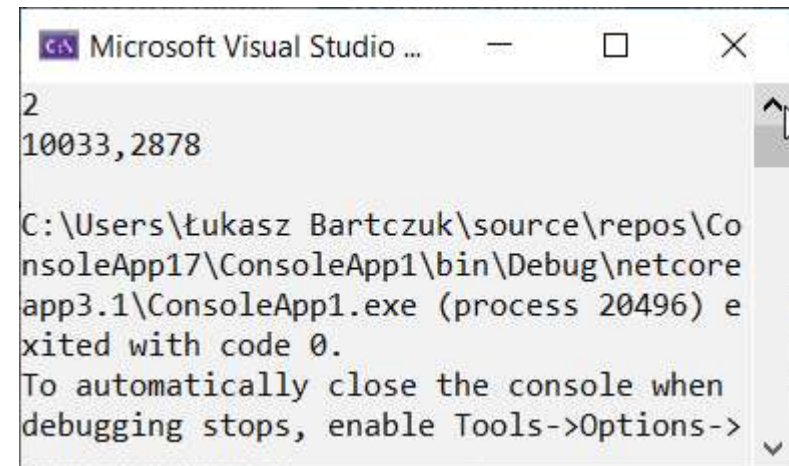
Implementacja za pomocą Func<>

```
static int Funkcja4(  
    bool warunek,  
    Func<int> wartosc1,  
    Func<int> wartosc2) {  
    return warunek ? wartosc1() : wartosc2();  
}
```

Obliczenia leniwe

```
static void Przyklad() {  
    var wynik = Funkcja4(false,  
                          Funkcja1,  
                          Funkcja2);  
    Console.WriteLine(wynik);  
}
```

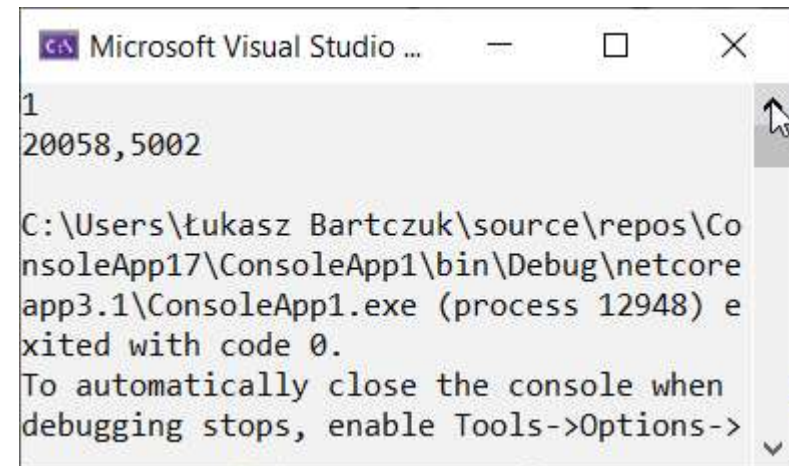
```
static void Main(string[] args) {  
    Console.WriteLine(ZmierzCzas(Przyklad));  
}
```



Obliczenia leniwe

```
static void Przyklad() {  
    var wynik = Funkcja4(true,  
                          Funkcja1,  
                          Funkcja2);  
    Console.WriteLine(wynik);  
}
```

```
static void Main(string[] args) {  
    Console.WriteLine(ZmierzCzas(Przyklad));  
}
```

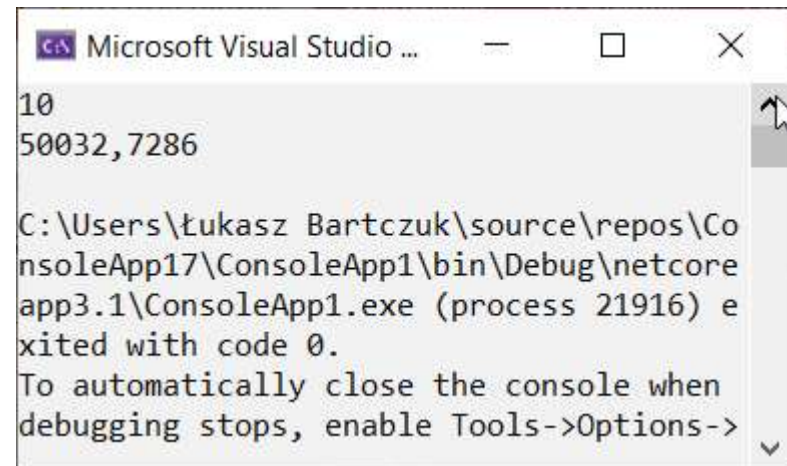


Obliczenia leniwe

```
static int Funkcja5(int liczbaPowtorzen, Func<int> wartosc) {  
  
    int suma = 0;  
    for(int i=0; i<liczbaPowtorzen; i++) {  
        suma += wartosc();  
    }  
  
    return suma;  
  
}
```

Obliczenia leniwe

```
static void Przyklad() {  
    var wynik = Funkcja5(5, Funkcja2);  
    Console.WriteLine(wynik);  
}  
  
static void Main(string[] args) {  
    Console.WriteLine(ZmierzCzas(Przyklad));  
}
```

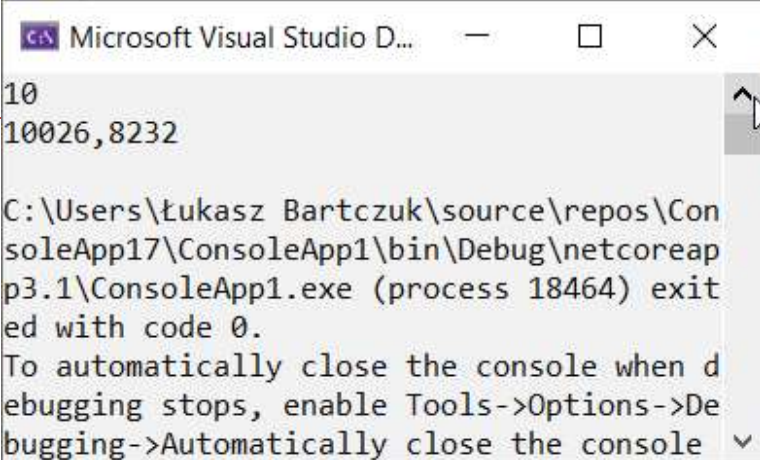


Obliczenia leniwe

```
static int Funkcja6(int liczbaPowtorzen, Lazy<int> wartosc) {  
  
    int suma = 0;  
    for(int i=0; i<liczbaPowtorzen; i++) {  
        suma += wartosc.value;  
    }  
  
    return suma;  
  
}
```

Obliczenia leniwe

```
static void Przyklad() {  
    var wynik =  
        Funkcja6(5, new Lazy<int>(Funkcja2));  
    Console.WriteLine(wynik);  
}  
  
static void Main(string[] args) {  
    Console.WriteLine(ZmierzCzas(Przyklad));  
}
```

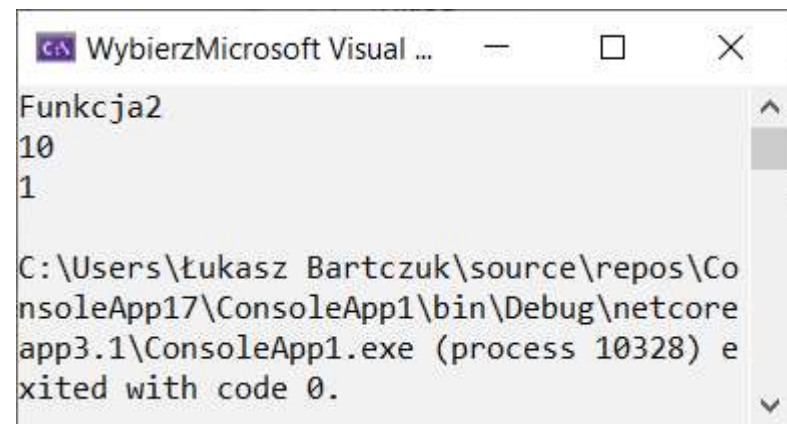


The screenshot shows the 'Output' window of Microsoft Visual Studio. It displays the output of a program execution. The first line is '10', which is the result of the lazy calculation. The second line is '10026,8232', which is the execution time. Below these, there is a message indicating that the application has exited with code 0. At the bottom, there is a tip about automatically closing the console when debugging stops.

```
10  
10026,8232  
  
C:\Users\Łukasz Bartczuk\source\repos\ConsoleApp17\ConsoleApp1\bin\Debug\netcoreapp3.1\ConsoleApp1.exe (process 18464) exited with code 0.  
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console
```

Obliczenia leniwe

```
static int liczbaWywolan;  
  
static int Funkcja2() {  
    Console.WriteLine(nameof(Funkcja2));  
    liczbaWywolan++;  
    return 2;  
}  
  
static void Main(string[] args) {  
    Przyklad();  
    Console.WriteLine(liczbaWywolan);  
}
```



```
WybierzMicrosoft Visual ...  
Funkcja2  
10  
1  
C:\Users\Łukasz Bartczuk\source\repos\ConsoleApp17\ConsoleApp1\bin\Debug\netcoreapp3.1\ConsoleApp1.exe (process 10328) exited with code 0.
```


Zachłanna inicjalizacja obiektów

```
class Autor
{
    public int Id {get;}
    public string Imie {get;}
    public string Nazwisko {get;}
    public Ksiazki Ksiazki{get;}

    public Autor(int id, string imie, string nazwisko, Ksiazki ksiazki) {
        Id = id;
        Imie = imie;
        Nazwisko = nazwisko;
        Ksiazki = ksiazki;
    }
}
```

Zachłanna inicjalizacja obiektów

```
class RepozytoriumKsiazek {  
    public Ksiazki PobierzDlaAutora(int autorId) { return null; }  
}  
  
static void Main(string[] args)  
{  
    var repozytoriumKsiazek = new RepozytoriumKsiazek();  
    var autor = new Autor(1, "Adam", "Mickiewicz",  
        repozytoriumKsiazek.PobierzDlaAutora(1));  
}
```

Leniwa inicjalizacja obiektów

```
class Autor
{
    private Lazy<Ksiazki> ksiazki;

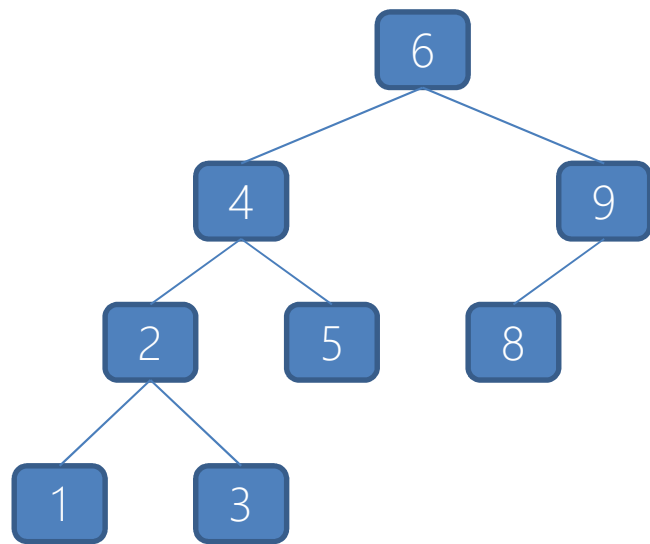
    public int Id {get;}
    public string Imie {get;}
    public string Nazwisko {get;}
    public Ksiazki Ksiazki {get => ksiazki.Value}

    public Autor(int id, string imie, string nazwisko, Lazy<Ksiazki> ksiazki) {
        Id = id;
        Imie = imie;
        Nazwisko = nazwisko;
        this.ksiazki = ksiazki;
    }
}
```

Leniwa inicjalizacja obiektów

```
static void Main(string[] args) {  
  
    var repozytoriumKsiazek = new RepozytoriumKsiazek();  
    var autor = new Autor(1, "Adam", "Mickiewicz",  
        new Lazy<Ksiazki>(()=>repozytoriumKsiazek.PobierzDlaAutora(1)));  
  
}
```

Wzorzec iterator



1,2,3,4,5,6,8,9



1,2,3,4

Wzorzec iterator

Wzorzec iterator zapewnia sekwencyjny dostęp do elementów większego obiektu

- ukrycie wewnętrznej reprezentacji obiektu
- udostępnia jednolity interfejs do poruszania się po zagregowanych strukturach

IEnumerable<T>, IEnumerator<T>

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

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

IEnumerable<T>, IEnumerator<T>

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


Lista

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

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

Lista – iterator (1)

```
class Iterator<T> : IEnumerator<T>
{
    public T Current { ... }

    object IEnumerator.Current => Current;

    public bool MoveNext() { ... }
    public void Reset() {...}

    public void Dispose() { }
}
```

Lista – iterator (2)

```
class Iterator<T> : IEnumerator<T>
{
    private Lista<T> obecny;
    private Lista<T> poczatek;

    public Iterator(Lista<T> lista) {
        obecny = null;
        poczatek = lista;
    }
}
```

Lista – iterator (3)

```
class Iterator<T> : IEnumerator<T> {  
    ...  
    public T Current {  
        get => obecny == null ? default : obecny.Wartosc;  
    }  
  
    object IEnumerator.Current => Current;  
  
    public bool MoveNext() {  
        obecny = obecny == null ? poczatek : obecny.Nastepny;  
        return obecny != null;  
    }  
  
    public void Reset() { obecny = poczatek; }  
  
    public void Dispose() { }  
}
```

Lista

```
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 Iterator<T>(this);

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

Wykorzystanie

```
class Program {  
  
    static void Main(string[] args) {  
        var lista = Lista(1, Lista(2, Lista(3, Lista(4, null))));  
        var wsk = lista.GetEnumerator();  
  
        while (wsk.MoveNext()) {  
            var el = wsk.Current;  
            Console.WriteLine(el);  
        }  
    }  
}
```

Pętla foreach

```
var wsk = lista.GetEnumerator();
```

```
while (wsk.MoveNext()) {  
    var el = wsk.Current;  
    Console.WriteLine(el);  
}
```



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

Leniwe sekwencje

Sekwencje, w których kolejne elementy nie są obliczane natychmiast, ale w momencie gdy są potrzebne.

Generatory

Obiekty, które implementują jednocześnie interfejsy
IEnumerator i IEnumerable

Fibonacci

```
class Fibonacci : IEnumerator<Tuple<int, int>>, IEnumerable<Tuple<int, int>>
{
    private int f_1 = 1;
    private int f_2 = 0;
    private int element = -1;

    public Tuple<int, int> Current => Tuple.Create(f_1, element);

    object IEnumerator.Current => Tuple.Create(f_1, element);

    public bool MoveNext() {
        var tmp = f_1;
        f_1 = f_2;
        f_2 = tmp + f_1;
        element++;
        return true;
    }
}
```

Fibonacci

```
public void Reset() {  
    f_1 = 1;  
    f_2 = 0;  
    element = -1;  
}  
  
public void Dispose() { }  
  
public IEnumerator<Tuple<int, int>> GetEnumerator() => this;  
  
IEnumerator IEnumerable.GetEnumerator() => this;  
}
```

Fibonacci

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

Funkcje iteracyjne

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

Funkcje iteracyjne

```
private sealed class <Example>d__3 :  
    IEnumerable<int>, IEnumerable, IEnumerator<int>,  
    IDisposable, IEnumerator  
{  
    private int <>1__state;  
    private int <>2__current;  
    private int <>1__initialThreadId;  
  
    public <Example>d__3(int _param1) {  
        base..ctor();  
        this.<>1__state = _param1;  
        this.<>1__initialThreadId = Environment.CurrentManagedThreadId;  
    }  
}
```

Funkcje iteracyjne

```
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;
    }
}
```

Funkcje iteracyjne

```
void IDisposable.Dispose() {}
```

```
int IEnumerator<int>.Current { get { return this.<>2__current; } }
```

```
void IEnumerator.Reset() { throw new NotSupportedException(); }
```

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


Funkcje iteracyjne - lista

```
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;  
        }  
    }  
}
```

Funkcje iteracyjne - lista

```
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++;
    }
}
```

Funkcje iteracyjne

```
public static IEnumerable<Tuple<int,int>> Fib() {  
    int f_1 = 0;  
    int f_2 = 1;  
    int element = 0;  
  
    do {  
        int tmp = f_1;  
        f_1 = f_2;  
        f_2 = tmp + f_2;  
        yield return Tuple.Create(tmp, element++);  
    }  
    while (true);  
}
```

METODY ROZSZERZAJĄCE

Problem

Przygotowanie metody, która pozwala z łańcucha znaków wybrać co drugi znak.

Podejście 1 – modyfikacja klasy string


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



Brak możliwości
zmodyfikowania
klasy String.

Podejście 2 – tworzenie klasy pochodnej

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



Klasa string
jest klasą
zapięczętowaną

Podójście 3 – tworzenie nowej klasy statycznej


```
class static 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();  
    }  
}
```

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


Metody rozszerzające

Metody rozszerzające pozwalają na dodanie nowych metod do już istniejących typów danych bez konieczności tworzenia nowego typu lub modyfikowania istniejącego kodu.

Podójście 4 – metody rozszerzające

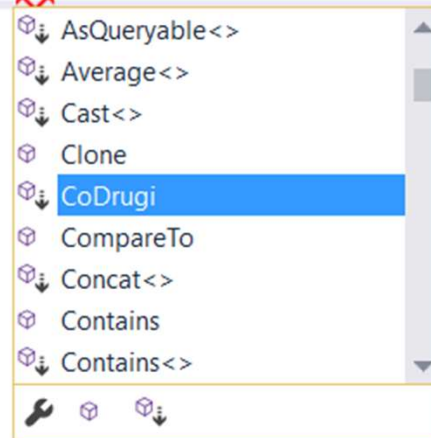


Nazwa klasy jest bez
znaczenia

```
static class Extensions {  
    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();  
    }  
}
```

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

```
public void Metoda()  
{  
    string text = "Ala ma kota";  
    string rezultat = text.  
}
```



(extension) `string string.CoDrugi()`

LINQ

LINQ – Wstęp

LINQ – Language Integrated Query

Biblioteka LINQ pozwala na ujednolicony sposób odpytywania dowolnej sekwencyjnej kolekcji danych.

Składnia LINQ wzorowana jest na języku SQL.

Linq to Objects

LINQ to Objects służy do odpytywania kolekcji obiektów takich jak:

- listy
- tablice
- kolejki
- ...

Zapytania

```
var rezultat =
```

```
(from v in listaLosowa
```

```
where v > 10
```

```
select v * v
```

```
).ToList();
```

Kolekcja źródłowa

Warunek wyboru

Mapowanie

Operacje mapowania elementów

Operacja mapowania listy tworzy nową listę poprzez zastosowanie funkcji f do każdego elementu listy wejściowej

$\text{map}: (A[], A \rightarrow B) \rightarrow B[]$

$\text{map}(\begin{array}{|c|c|c|c|c|c|c|c|c|} \hline a1 & a2 & a3 & a4 & a5 & a6 & a7 & a8 & a9 \\ \hline \end{array}, f)$



$\begin{array}{|c|c|c|c|c|c|c|c|c|} \hline f(a1) & f(a2) & f(a3) & f(a4) & f(a5) & f(a6) & f(a7) & f(a8) & f(a9) \\ \hline \end{array}$

Operacje mapowania elementów

```
public static IEnumerable<TResult> Select<TSource, TResult>  
    (this IEnumerable<TSource> source,  
     Func<TSource, TResult> selector)
```

```
var lista = (from v in Enumerable.Range(0, 100)  
             select rand.Next(0, 99)  
            ).ToList();
```

```
var kwadraty = (from v in lista  
                select v * v  
               ).ToArray();
```

Operacje mapowania elementów

```
List<Punkt> punkty = new List<Punkt> {  
    new Punkt { Kolor = Color.Red, X=1, Y=1 },  
    new Punkt { Kolor = Color.Blue, X=10, Y=10 },  
    new Punkt { Kolor = Color.Green, X=-10, Y=-10 }  
};
```

[🔍] (local variable) List<'a> wybrane

Anonymous Types:

'a is new { int X, int DW }

```
var wybrane = (from p in punkty  
               select new { p.X, DW = p.Y }  
               ).ToList();
```

Odroczone wykonanie

```
IEnumerable<int> numbers =  
    Enumerable.Range(0, 10)  
        .Select(n => n * n);
```



Nie spowoduje
wykonania zapytania

- ToList
- ToArray
- ToDictionary
- ToLookup

Operacja filtrowania listy

Operacja filtrowania listy tworzy nową listę poprzez wybranie tych elementów listy, które spełniają określony warunek

$\text{filtr}: (A[], A \rightarrow \text{bool}) \rightarrow A[]$

$\text{filtr}(\text{a1} \text{ a2} \text{ a3} \text{ a4} \text{ a5} \text{ a6} \text{ a7} \text{ a8} \text{ a9}, p)$



$\text{a1} \text{ a3} \text{ a4} \text{ a7} \text{ a8} \text{ a9}$

Operacja filtrowania listy

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate);
```

```
var parzyste = (from v in lista
                 where v % 2 == 0
                 select v
                );
```

Przykładowe dane

```
class Osoba {  
    public int Id { get; init; }  
    public string Imie { get; init; }  
    public string Nazwisko { get; init; }  
}
```

```
class Pojazd {  
    public string NrRej { get; init; }  
    public int IdOsoby { get; init; }  
}
```

Przykładowe dane

```
var osoby = new [] {  
    new Osoba {Id = 1, Imie="Ala", Nazwisko = "Kot" },  
    new Osoba {Id = 2, Imie="Cezary", Nazwisko = "Adamski"},  
    new Osoba {Id = 3, Imie="Franek", Nazwisko="Relke"},  
    new Osoba {Id = 4, Imie="Tomasz", Nazwisko="Nowak"}  
};
```

```
var pojazdy = new [] {  
    new Pojazd {IdOsoby = 1, NrRej="SC12345"},  
    new Pojazd {IdOsoby = 2, NrRej="WE12345"},  
    new Pojazd {IdOsoby = 1, NrRej="KW12345"}  
};
```

Filtrowanie kolekcji

```
var pojazdyZCzwy = pojazdy
    .Where(p => p.NrRej.StartsWith("SC"))
    .ToList();
```

```
var pojazdyZCzwy = (from pojazd in pojazdy
    where pojazd.NrRej.StartsWith("SC")
    select pojazd
).ToList();
```


Operacja grupowania

Operacja grupowania pozwala podzielić kolekcje na podzbiory

```
var grupy = (from v in lista  
             group v by v % 2 == 0  
             );
```



IEnumerable<IGrouping<bool,int>>

→ Key = true, IEnumerable<int>

→ Key = false, IEnumerable<int>

Operacja grupowania

```
var pojazdy0sob = (from pojazd in pojazdy  
                    group pojazd by pojazd.Id0soby  
                    ).ToList();
```

```
var pojazdy0sob = pojazdy.GroupBy(g=>g.Id0soby).ToList();
```

```
List<IGrouping<int,Pojazd>>
```

Operacja grupowania

```
var pojazdyOsob =  
    (from pojazd in pojazdy  
     group pojazd by pojazd.IdOsoby into grupa  
     orderby grupa.Key  
     select new {IDOsoby = grupa.Key, Pojazdy = grupa.ToArray()}  
    ).ToList();
```

```
var pojazdyOsob = pojazdy  
    .GroupBy(g=>g.IdOsoby,  
             (idOsoby, pojazdy)=> new { IDOsoby = idOsoby,  
                                         Pojazdy = pojazdy.ToArray()  
                                         }).ToList();
```

Złączenia

```
var osobyIPojazdy =  
    (from osoba in osoby  
     join pojazd in pojazdy  
       on osoba.Id equals pojazd.IdOsoby  
     select new { osoba.Imie, osoba.Nazwisko, pojazd.NrRej }  
    ).ToList();
```

```
var osobyIPojazdy =  
    osoby.Join(pojazdy,  
               o => o.Id,  
               p => p.IdOsoby,  
               (o, p) => new { o.Imie, o.Nazwisko, p.NrRej }  
    ).ToList();
```

Złączenia grupujące

```
var osobyIPojazdy =  
    (from osoba in osoby  
     join pojazd in pojazdy on osoba.Id equals pojazd.IdOsoby  
     into grupy  
     select new { osoba.Imie,  
                   osoba.Nazwisko,  
                   pojazdy = grupy.ToList()  
                 }  
    ).ToList();
```

Złączenia grupujące

```
var osobyIPojazdy =  
    osoby.GroupJoin(pojazdy,  
        o => o.Id,  
        p => p.IdOsoby,  
        (o, ps) =>  
            new { o.Imie,  
                  o.Nazwisko,  
                  Pojazdy = ps.ToList()  
            })  
    ).ToList();
```

Złączenie zewnętrzne

```
var osobyIPojazdy =  
    (from osoba in osoby  
     join pojazd in pojazdy on osoba.Id equals pojazd.IdOsoby  
     into grupy  
     from g in grupy.DefaultIfEmpty()  
     select new { osoba.Imie,  
                  osoba.Nazwisko,  
                  NrRej = g?.NrRej ?? "Brak pojazdu"  
     }).ToList();
```

Zmienne zakresu i let

Zmienne zakresu reprezentują każdy element sekwencji w kolejności

```
var wybrane = (from ksiazka in ksiazki
                let pLitera = ksiazka.Tytul[0] == 'P'
                let lStron = ksiazka.Strony > 250
                where pLitera && lStron
                select ksiazka
                ).ToList();
```


Sortowanie list



```
var listaLosowa  
    = new List<int>();  
...  
listaLosowa.Sort();
```

```
var listaLosowa  
    = new List<int>();  
...  
listaLosowa.OrderBy(v=>v)  
    .ToList();
```

Operacja agregowania elementów (1)

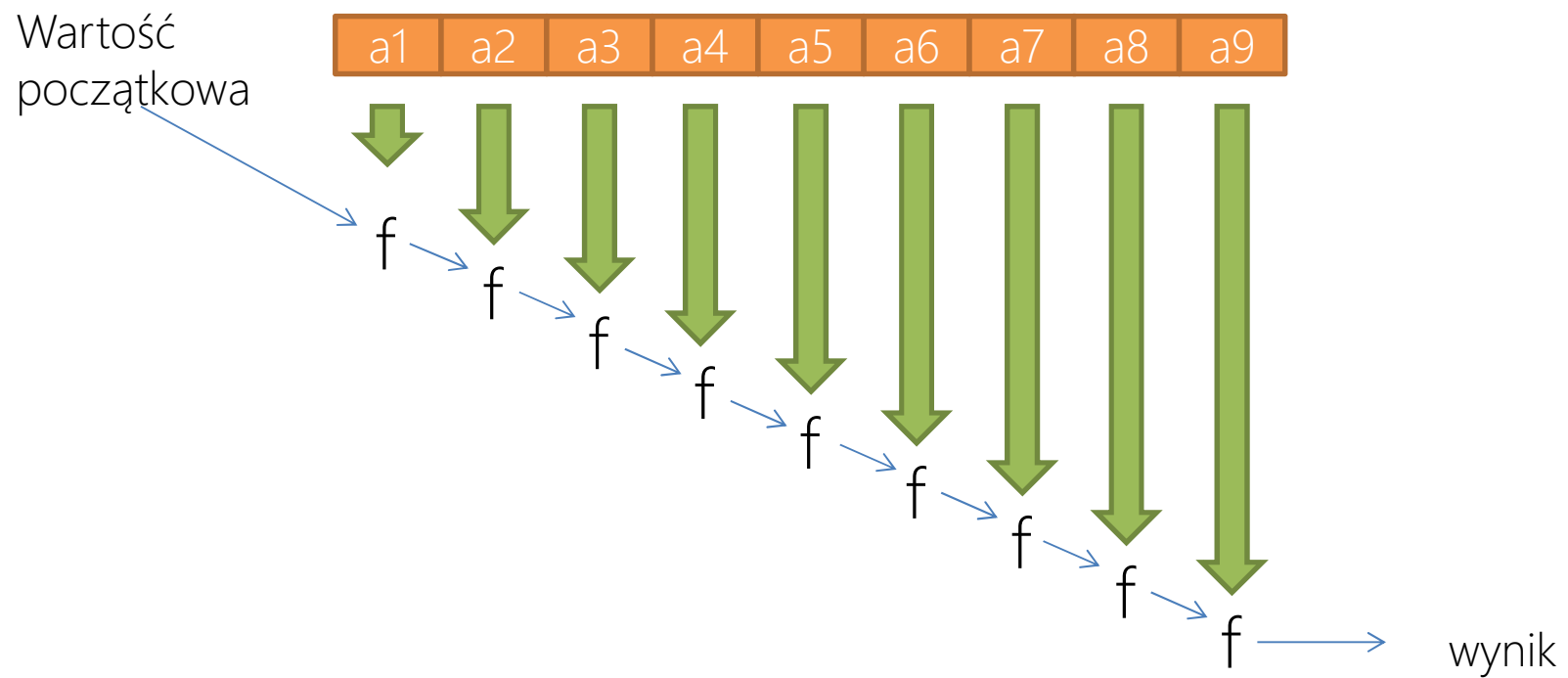
Operacja agregowania tworzy pojedynczą wartość z połączenia elementów listy

$\text{agreguj}(A[], R, (R, A) \rightarrow R) \Rightarrow R$

Wartość początkowa

Funkcja agregująca

Operacja agregowania elementów (2)



Operacja agregowania elementów (3)

```
int sum = Enumerable.Range(0, 10)
    .Aggregate(0, (a, v) => a + v);
```

```
listaLosowa
    .Aggregate(listaLosowa[0],
        (a, v) => a > v ? v : a)
```

Zapytania

```
var rezultat = listaLosowa  
    .Where(v => v > 10)  
    .Select(v => v * v).ToList();
```

```
var rezultat =  
    (from v in listaLosowa  
     where v > 10  
     select v * v  
    ).ToList();
```

SelectMany (1)

Operator SelectMany pozwala spłaszczyć hierarchię sekwencji

```
var przedmioty = new List<string> {  
    "Algebra",  
    "Paradygmaty programowania",  
    "Systemy Operacyjne"  
};  
  
var litery = przedmioty  
    .SelectMany(p => p.ToCharArray())  
    .ToList();
```

SelectMany (2)

```
var litery = przedmioty
    .SelectMany(p =>p.ToCharArray())
    .ToList();
```

```
static IEnumerable<R> SelectMany<A, R>(
    this IEnumerable<A> sequence,
    Func<A, IEnumerable<R>> function)
{
    foreach (A outerItem in sequence)
        foreach (R innerItem in function(outerItem))
            yield return innerItem;
}
```

SelectMany (3)

```
var litery = przedmioty
    .SelectMany(p => p.ToCharArray())
    .ToList();
```

```
var litery = (
    from przedmiot in przedmioty
    from litera in przedmiot.ToCharArray()
    select litera
).ToList();
```


SelectMany (4)

```
var a = new[] { 1, 4, 7 };  
var b = new[] { 2, 5, 8 };  
var wynik = a.SelectMany(i => b, (i, j) => i + j).ToList();
```

```
public static IEnumerable<C> SelectMany<A, B, C>(
    this IEnumerable<A> items,
    Func<A, IEnumerable<B>> function,
    Func<A, B, C> projection)
{
    foreach (A outer in items)
        foreach (B inner in function(outer))
            yield return projection(outer, inner);
}
```

SelectMany (5)

```
var a = new[] { 1, 4, 7 };  
var b = new[] { 2, 5, 8 };  
var wynik = a.SelectMany(i => b, (i, j) => i + j).ToList();
```

```
var wynik = (  
    from v1 in a  
    from v2 in b  
    select v1 + v2  
).ToList();
```

Linq to Objects - operatory

Operator	Grupa	Operator	Grupa
Aggregate	Agregacja	ElementAt	Elementy
All	Liczebność	ElementAtOrDefault	Elementy
Any	Liczebność	Empty	Tworzenie
AsEnumerable	Konwersja	Except	Zbiór
Average	Agregacja	First	Elementy
Cast	Konwersja	FirstOrDefault	Elementy
Concat	Zbiór	GroupBy	Grupowanie
Contains	Liczebność	GroupJoin	Połączenia
Count	Agregacja	Intersect	Zbiór
DefaultIfEmpty	Elementy	Join	Elementy
Distinct	Zbiór	Last	Elementy

Linq to Objects - operator

Operator	Grupa	Operator	Grupa
LastOrDefault	Elementy	SelectMany	Projekcja
LongCount	Agregacja	SequenceEqual	Równość
Max	Agregacja	Single	Elementy
Min	Agregacja	SingleOrDefault	Elementy
OfType	Konwersja	Skip	Podział
OrderBy	Porządkowanie	SkipWhile	Podział
OrderByDescending	Porządkowanie	Sum	Agregacja
Range	Generacja	Take	Podział
Repeat	Generacja	TakeWhile	Podział
Reverse	Porządkowanie	ThenBy	Porządkowanie
Select	Projekcja	ThenByDescending	Porządkowanie

LINQ to Objects - operatory

Operator	Grupa	Operator	Grupa
ToArray	Konwersja	ToLookup	Konwersja
ToDictionary	Konwersja	Union	Zbiór
ToList	Konwersja	Where	Ograniczenia