LINQ i bazy danych

Bartosz Miąskowski

Atrybuty

Atrybuty to pewnego rodzaju „*etykiety z informacją*”, które możesz doczepić do **klas, właściwości czy metod**. Same z siebie **nic nie robią** – ale frameworki i biblioteki **mogą je odczytać** i na ich podstawie podjąć decyzje.

Dzięki temu możesz sterować zachowaniem np. serializacji, walidacji czy mapowania do bazy.

Atrybuty

Co tutaj się dzieje?

- Atrybuty **dodają metadane** do kodu.
- Te metadane mogą być **odczytane w czasie działania programu** (*refleksja*).
- Dzięki nim narzędzia wiedzą, jak traktować Twój kod – *np. jak nazwać pole w JSON*.

Atrybuty to **informacja o kodzie**, a nie logika. Sam atrybut nic nie robi – dopiero framework (*lub biblioteka*) decyduje, jak go wykorzystać.

Jeżeli atrybut przyjmuje jakieś wartości, muszą być to stałe kompilacji!

```
using System.Text.Json.Serialization;

public class User
{
    [JsonPropertyName("full_name")]
    public string Name { get; set; }

    [JsonConverter(typeof(DateOnlyConverter))]
    public DateOnly BirthDate { get; set; }
}
```

```
{
  "full_name": "Adam Kowalski",
  "birthDate": "2002-11-12"
}
```

Jak działają atrybuty?

```
using System;
using System.Reflection;
using System.Text.Json.Serialization;

public class Product
{
    [JsonPropertyName("item_id")]
    public int Id { get; set; }
}

public class Demo
{
    public static void Main()
    {
        var prop = typeof(Product)
            .GetProperty(nameof(Product.Id));

        var attr = prop!
            .GetCustomAttribute<JsonPropertyNameAttribute>();

        Console.WriteLine(attr?.Name); //Output: "item_id"
    }
}
```

Co tutaj się dzieje?

- `GetProperty()` pobiera metadane klasy.
- `GetCustomAttribute()` sprawdza, czy właściwość ma przypięty atrybut.
- Atrybut **jest tylko danymi** – framework musi coś z tym zrobić.

Atrybuty są zapisane w metadanych i dostępne przez refleksję - to właśnie ta mechanika pozwala narzędziom reagować na atrybuty.

Do obsługi atrybutów świetnie spisują się metody rozszerzeń.

Tworzenie własnego atrybutu

Możesz tworzyć **własne atrybuty**, jeśli chcesz oznaczać kod dodatkowymi informacjami – dokładnie tak, jak robi to np. `[JsonPropertyName]`.

Taki atrybut nadal **nie zawiera logiki** — to tylko dane, które ktoś musi później odczytać.

Tworzenie własnego atrybutu

Co warto wiedzieć?

- `AttributeUsage` określa, **gdzie można użyć atrybutu**.
- Atrybut to zwykła klasa dziedzicząca po `System.Attribute`.
- Konstruktor pozwala przekazać dane (*np. opis, nazwę, wersję*).
- Atrybut **nigdy nie wywoła się „*sam*”** – ktoś musi go odczytać.

Własny atrybut to zwykła klasa z danymi – framework lub Twoja aplikacja musi sama zdecydować, co z tymi danymi zrobić.

Tworząc nazwę atrybutu, **zawsze umieszczamy dopisek „*`Attribute`*”**, tak jak na załączonym przykładzie – Documentation**`Attribute`**.

Dopisek „*`Attribute`*” nie będzie widoczny, kiedy użyjemy atrybutu.

```
using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Property)]
public class DocumentationAttribute : Attribute
{
    public string Info { get; }

    public DocumentationAttribute(string info)
    {
        Info = info;
    }
}

[Documentation("Ta klasa reprezentuje użytkownika systemu.")]
public class User
{
    [Documentation("Identyfikator użytkownika.")]
    public int Id { get; set; }
}
```

```
var attrs = typeof(User).GetCustomAttributes<DocumentationAttribute>();
Console.WriteLine(attrs.First().Info);

//Output: "Ta klasa reprezentuje użytkownika systemu."
```

Refleksja

Refleksja to mechanizm, który pozwala aplikacji **zajrzeć do samej siebie** w czasie działania. Dzięki niej możesz odczytać informacje o klasach, właściwościach, metodach, atrybutach – nawet jeśli nie znałeś ich w momencie kompilacji.

To tak, jakby program umiał otworzyć własny kod i powiedzieć: „*zobaczmy, co tu jest*”.

Do czego przydaje się refleksja?

- Automatyczna serializacja (*JSON*, *XML*, *YAML*).
- Entity Framework – mapowanie klas na tabele.
- Dependency Injection.
- Czytanie atrybutów.
- Dynamiczne tworzenie obiektów.

```
var type = typeof(string);

Console.WriteLine(type.Name); //String
Console.WriteLine(type.Namespace); //System
```

Przykład użycia refleksji

Refleksja – operator **`typeof`**

Refleksja zaczyna się od jednego – musisz dostać **typ** obiektu lub klasy. Są trzy najczęstsze sposoby, a każdy przydaje się w trochę innym miejscu.

Słowo kluczowe `typeof` jest operatorem, który pozwala pobrać typ obiektu (`System.Type`), **operując na nazwie typu**, która musi być **znana już w momencie kompilacji** – operator jest rozwiązywany podczas kompilacji.

```
Type t = typeof(int);
Console.WriteLine(t.FullName); //Output: "System.Int32"
```

Refleksja – metoda **`GetType()`**

Metoda **`GetType()`** pozwala pobrać typ obiektu, na podstawie **jego instancji**. Metoda jest obecna w klasie `object`, nie można jej nadpisać.

```
var user = new User();
Type t = user.GetType();

Console.WriteLine(t.Name); //Output: "User"
```

Refleksja – klasa `Assembly`

Klasa `Assembly` dostarcza informacje o całym projekcie. Informacje te, możemy pobrać zarówno **odwołując się do danego typu** (*na jego podstawie pobrać informację o zestawie, do którego należy*). Możemy się również odwołać do **metod statycznych dostępnych w klasie `Assembly`**.

Tego podejścia często używa się przy:

- Automatycznej rejestracji usług.
- Wyszukiwaniu klas.
- Budowaniu generatorów kodu

```
var assembly = typeof(User).Assembly;

foreach (var type in assembly.GetTypes())
{
    Console.WriteLine(type.Name);
}

//Przykładowy output:
//User
//Product
//Order
//DbContext
```

Refleksja – klasa **`Assembly`**

```
var asm = Assembly.GetEntryAssembly();
Console.WriteLine(asm?.FullName);

//Przykładowy output: "MyApp.exe"
```

```
var asm = Assembly.GetExecutingAssembly();
Console.WriteLine(asm.FullName);

//Przykładowy output: "MyLibrary.dll"
```

```
var asm = Assembly.GetCallingAssembly();
Console.WriteLine(asm.FullName);

//Przykładowy output: "MyApp.exe"
```

Wśród metod statycznych w klasie `Assembly` wyróżniamy:

- **`Assembly.GetEntryAssembly()`** – zwraca zestaw, który uruchomił cały program *(czyli główny exe)*.
- **`Assembly.GetExecutingAssembly()`** – zwraca zestaw, w którym aktualnie wykonuje się dany kod, zestaw, w którym jest obecny dany wiersz kodu.
- **`Assembly.GetCallingAssembly()`** – zwraca zestaw, który wywołał daną metodę – *„kto do mnie dzwoni?”*.

Refleksja – pobieranie właściwości klas

Refleksja pozwala Ci zajrzeć do wnętrza klasy i sprawdzić, jakie ma właściwości, ich typy, atrybuty czy wartości.
To fundament działania ORM-ów, serializerów i systemów mapowania.

```
public class User
{
    public int Id { get; set; }
    public string Name { get; set; }
}

var props = typeof(User).GetProperties();

foreach (var p in props)
    Console.WriteLine($"{p.Name} : {p.PropertyType.Name}");

//Output:
//Id : Int32
//Name : String
```

Refleksja – klasa `PropertyInfo`

Dzięki refleksji, możemy również odczytać wartości właściwości obecnych w obiekcie – włącznie z właściwościami prywatnymi *(z słowem kluczowym **`private`**)*, jeżeli użyjemy odpowiednich flag (`BindingFlags.NonPublic`).

```
var user = new User { Id = 10, Name = "Adam" };

var nameProp = typeof(User).GetProperty("Name");
var value = nameProp!.GetValue(user);

Console.WriteLine(value); //Output: "Adam"
```

Tylko właściwości publiczne

```
public class SecretUser
{
    public string PublicName { get; set; } = "Adam";
    private string SecretCode { get; set; } = "XYZ123";
}

var flags = BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic;
var props = typeof(SecretUser).GetProperties(flags);

foreach (var p in props)
    Console.WriteLine($"{p.Name} : {p.PropertyType.Name}");

//Output:
//PublicName : String
//SecretCode : String
```

Włącznie z właściwościami prywatnymi

Refleksja – klasa `PropertyInfo`

Co potrafi klasa `PropertyInfo`?

- **`GetValue(obj)`** – pobiera wartość.
- **`SetValue(obj, value)`** – ustawia wartość.
- **`GetCustomAttributes()`** – pobiera atrybuty.
- **`PropertyType`** – typ właściwości.
- **`Name`** – nazwa właściwości.

Refleksja potrafi odczytywać nazwy, typy i wartości właściwości – to dzięki temu frameworki „*same wiedzą*”, jak traktować Twoje obiekty.

Refleksja – odczytywanie atrybutów

Atrybuty same niczego nie wykonują – trzeba je **odczytać przy pomocy refleksji**, żeby coś z nimi zrobić. Frameworki takie jak np. EF Core czy ASP.NET Core, robią to cały czas.

```
using System.Text.Json.Serialization;

public class Product
{
    [JsonPropertyName("item_name")]
    public string Name { get; set; }
}

var prop = typeof(Product).GetProperty("Name");
var attr = prop!.GetCustomAttribute<JsonPropertyNameAttribute>();

Console.WriteLine(attr!.Name); //Output: "item_name"
```

Odczytywanie atrybutu z właściwości klasy

```
[Obsolete("Ta klasa jest stara")]
public class OldClass { }

var attrs = typeof(OldClass).GetCustomAttributes();

foreach (var a in attrs)
    Console.WriteLine(a.GetType().Name); //Output: "ObsoleteAttribute"
```

Pobieranie wszystkich atrybutów klasy

Refleksja – odczytywanie atrybutów

Najważniejsze metody, w kontekście atrybutów:

- **`GetCustomAttribute<T>()`** – pobiera pierwszy atrybut danego typu.
- **`GetCustomAttributes()`** – pobiera wszystkie atrybuty.
- **`IsDefined(type)`** – sprawdza, czy atrybut istnieje.

Dostępne zarówno na:

- Klasach,
- właściwościach,
- metodach,
- parametrach.

```
[Obsolete("Ta klasa jest stara")]
public class OldClass { }

bool hasAttr = typeof(OldClass)
    .IsDefined(typeof(ObsoleteAttribute), inherit: false);

Console.WriteLine(hasAttr); //Output: "True"
```

Sprawdzanie czy atrybut istnieje

Refleksja – zastosowanie

Entity Framework Core:

- Odnajdywanie klas, reprezentujących encje.
- Odczytanie właściwości klas.
- Wyszukiwanie kluczy, relacji, indexów.
- Budowa modelu bazy danych.

```
[Table("Invitations", Schema = "auth")]
public class UserInvitation
{
  [Key]
  [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
  public Guid Id { get; set; }

  public string Forname { get; set; }

  public string Surname { get; set; }
}
```

ASP.NET Core:

- Odczytuje metadane reprezentujące endpoint.
- Waliduje modele.

```
[HttpGet("{id}")]
[Authorize(Roles = "Administrator, Worker")]
public async Task<IActionResult> Request(string id)
{
    //Ciało metody
}
```

Ograniczenia refleksji

Refleksja jest **potężnym narzędziem**, ale **nie jest darmowa ani idealna**. Używa jej wiele frameworków, ale w aplikacjach krytycznych musisz wiedzieć, gdzie może sprawiać problemy.

- **Refleksja jest wolniejsza niż normalny kod.**
  - ~ 10 – 100 razy wolniejszy dostęp, jeżeli będziemy próbowali odczytać właściwość obiektu przy pomocy refleksji.
  - Narzut obliczeniowy jest akceptowalny głównie w narzędziach.
  - Nie powinna być używana w pętlach, dużych kolekcjach, gorących ścieżkach.
- **Działa poza typami – brak bezpieczeństwa typów.**
  - Ryzyko błędów, w trakcie działania programu.
- **Narusza zasady enkapsulacji / hermetyzacji.**
- **Kod korzystający z refleksji jest trudniejszy do utrzymania i debugowania.**

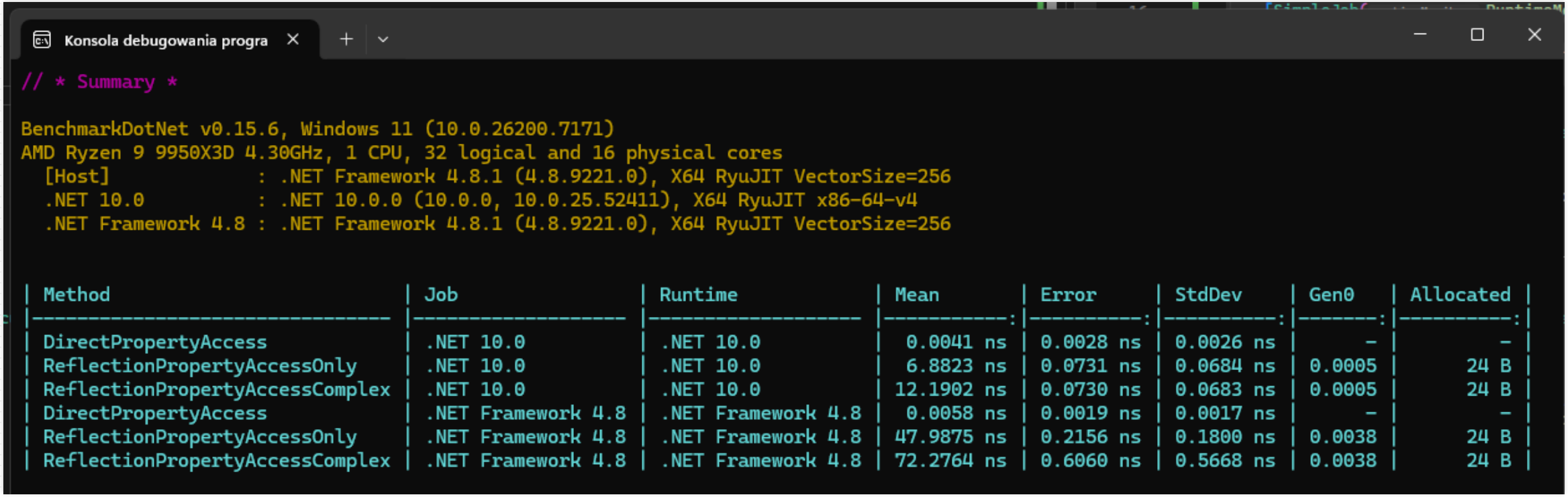
Refleksja – narzut obliczeniowy

```
private readonly User _user = new User() { Id = 42 };
private PropertyInfo _idProperty;

[GlobalSetup]
public void Setup()
{
    _idProperty = typeof(User).GetProperty("Id");
}

[Benchmark]
public int DirectPropertyAccess()
{
    return _user.Id;
}
```

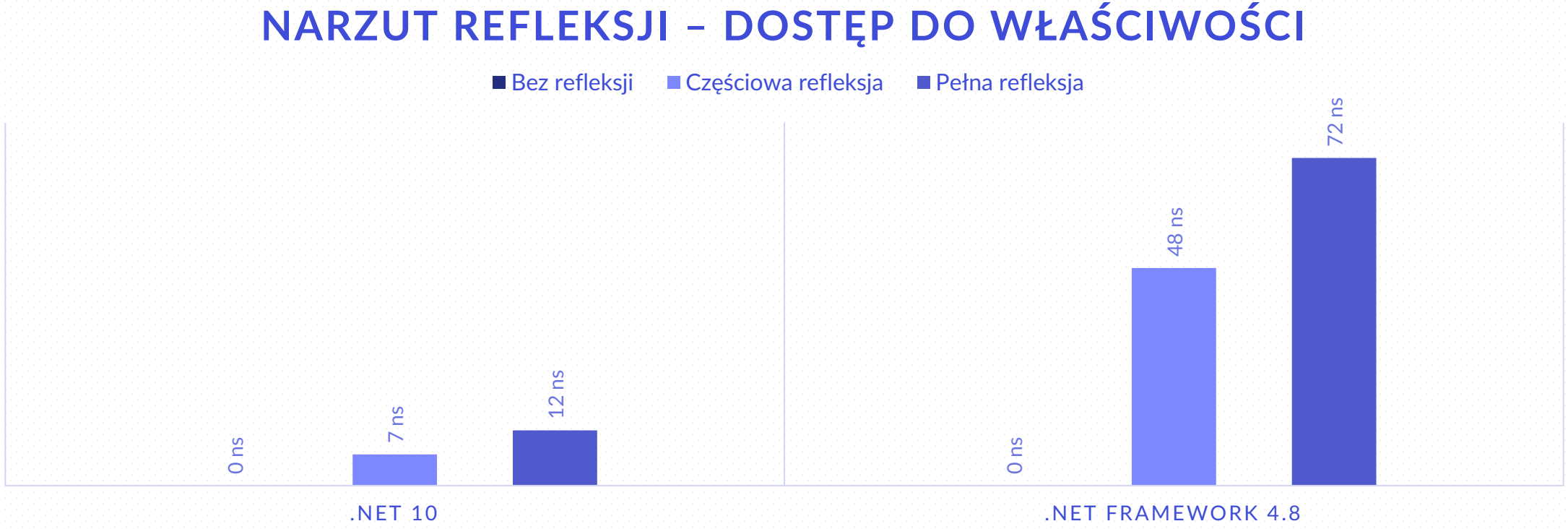
```
[Benchmark]
public int ReflectionPropertyAccessOnly()
{
    return (int)_idProperty.GetValue(_user);
}

[Benchmark]
public int ReflectionPropertyAccessComplex()
{
    var idProperty = typeof(User).GetProperty("Id");
    return (int)idProperty.GetValue(_user);
}

class User
{
    public int Id { get; set; }
}
```

Benchmark – odczytanie właściwości przy użyciu refleksji

Refleksja – narzut obliczeniowy



Refleksja – narzut obliczeniowy



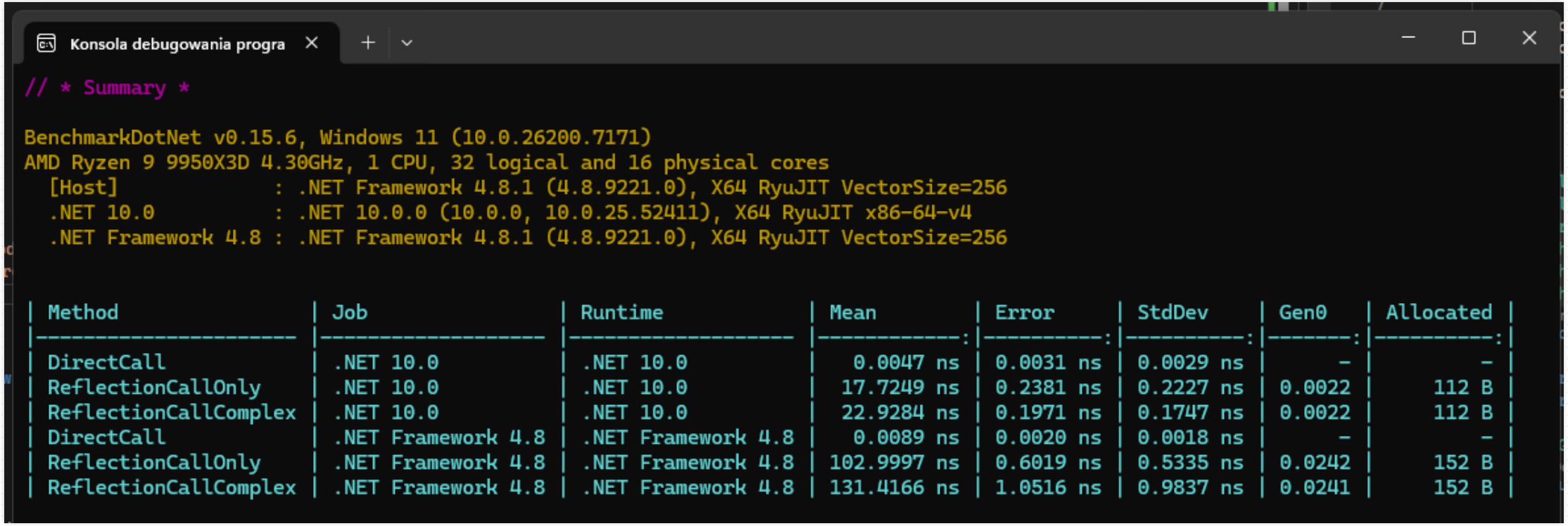
Refleksja – narzut obliczeniowy

```
private readonly Calculator _calculator = new Calculator();
private MethodInfo _addMethod;

[GlobalSetup]
public void Setup()
{
    _addMethod = typeof(Calculator).GetMethod(nameof(Calculator.Add));
}

[Benchmark]
public int DirectCall()
{
    return _calculator.Add(2, 3);
}
```

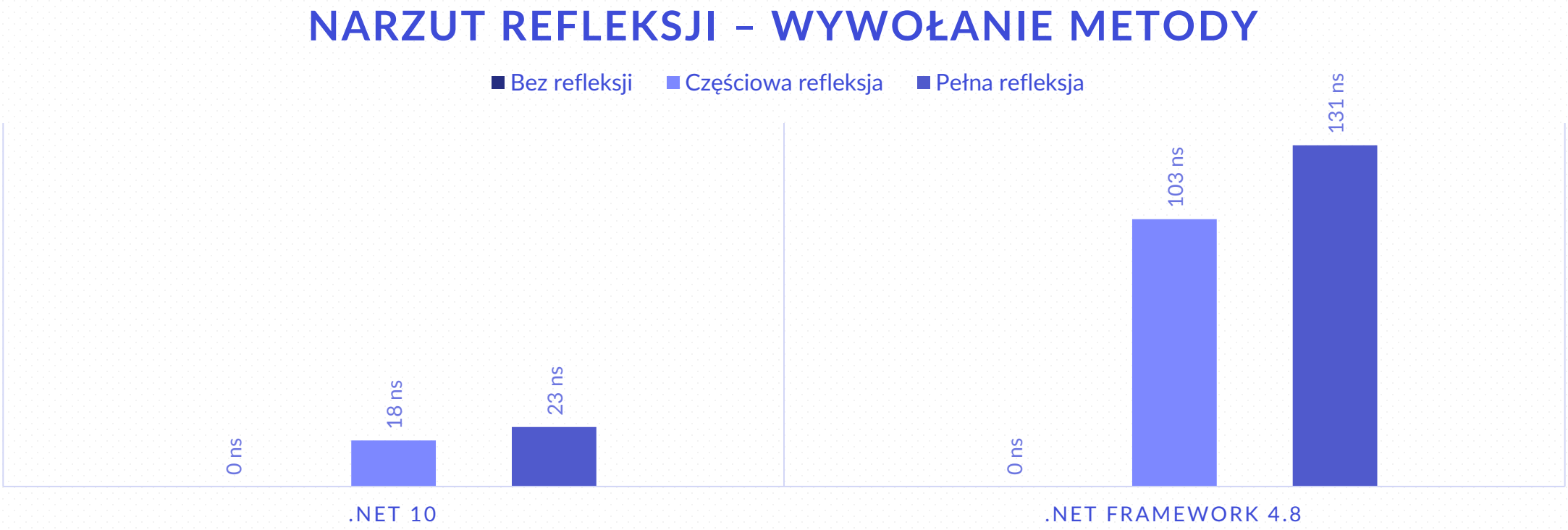
```
[Benchmark]
public int ReflectionCallOnly()
{
    return (int)_addMethod.Invoke(_calculator, new object[] { 2, 3 });
}

[Benchmark]
public int ReflectionCallComplex()
{
    var addMethod = typeof(Calculator).GetMethod(nameof(Calculator.Add));
    return (int)addMethod.Invoke(_calculator, new object[] { 2, 3 });
}

class Calculator
{
    public int Add(int a, int b) => a + b;
}
```

Benchmark – wywołanie metody przy użyciu refleksji

Refleksja – narzut obliczeniowy



Refleksja – narzut obliczeniowy



Czym jest LINQ?

LINQ (*Language Integrated Query*) to zintegrowany język zapytań, wbudowany w C#. Pozwala **filtrować, sortować, grupować i przetwarzać kolekcje** (*oraz **bazy danych**!*) w **prosty, czytelny i spójny** sposób – to tak, jakbyś tworzył mini-zapytania SQL, ale bez wychodzenia z C#.

LINQ działa zarówno na zwykłych kolekcjach implementujących interfejs **`IEnumerable<T>`**, jak i na bazach danych – dzięki **EF Core** – interfejs **`IQueryable<T>`**.

Dlaczego LINQ jest takie rewelacyjne?

- jest **czytelne**,
- jest **bezpieczne typowo**,
- unifikuje pracę z kolekcjami i bazami danych,
- można pisać w dwóch stylach
  - query syntax
  - method syntax

LINQ – style

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

var even = numbers
    .Where(n => n % 2 == 0)
    .Select(n => n * 10);

foreach (var n in even)
    Console.WriteLine(n);

//Output: 20 40
```

LINQ – zapis metodowy (method syntax)

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

var even =
    from n in numbers
    where n % 2 == 0
    select n * 10;

foreach (var n in even)
    Console.WriteLine(n);

//Output: 20 40
```

LINQ – zapis zapytaniowy (query syntax)

Na obu przykładach, kod robi dokładnie to samo

LINQ – query syntax

Query syntax (*choć używany rzadko*), przy niektórych operacjach (*`join`*, *`group by`*) bywa po prostu ładniejszy.

Kompilator zmienia query syntax na method syntax – to są dokładnie te same funkcje, różnią się wyłącznie zapisem (wizualnie).

```
var result = from n in numbers where n > 3 orderby n select n * 10;

//Output: 40 50
```

LINQ – method syntax

Method syntax to **łańcuch wywołań metod rozszerzeń** (*`Where`*, *`Select`*, *`OrderBy`* *itd.*) Jest krótszy, czytelniejszy i daje **dostęp do wszystkich metod** (*wiele metod jest nieobecnych w query syntax*).

```
var result = numbers
    .Where(n => n > 3)
    .Select(n => n * 10)
    .OrderBy(n => n);

//Output: 40 50
```

LINQ – interfejs `IEnumerable<T>`

LINQ nie jest żadnym magicznym tworem, działa dlatego, że **wszystkie metody LINQ są metodami rozszerzeń**, które przyjmują jako pierwszy parametr `IEnumerable<T>`.

Każda kolekcja, która implementuje `IEnumerable<T>`, potrafi:

- Udostępnić sekwencję elementów.
- Zwrócić enumerator – `GetEnumerator()`.
- Pozwolić na iterację – `foreach`.

LINQ po prostu wykorzystuje tę sekwencję, i umożliwia jej filtrowanie, sortowanie i przekształcanie.

Jeżeli kolekcja implementuje `IEnumerable<T>`, to automatycznie „*dostaje*” cały zestaw metod LINQ.

LINQ – interfejs `IEnumerable<T>`

Jak już zostało wspomniane – LINQ **to zestaw metod rozszerzeń**, które operują na `IEnumerable<T>`.

Poniższy przykład pokazuje, jak wygląda metoda `Where()` – *przykład został uproszczony*.

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> source, Func<T, bool> predicate)
{
    foreach (var item in source)
    {
        if (predicate(item))
            yield return item;
    }
}

//Wywołanie
var filtered = numbers.Where(n => n > 5); //Uwaga! filtered to IEnumerable<T>
```

*Zwróć uwagę – **jednym z parametrów jest delegata** `Func<T, bool>`*

LINQ – interfejs `IEnumerable<T>`

LINQ **nie wykonuje zapytania od razu** – do akcji wkracza **odroczone wykonanie**.

Dzięki temu:

- Nie są tworzone nowe listy przy każdym kroku – przy każdej metodzie.
- Elementy nie są przetwarzane od razu.
- Operacje są wykonywane dopiero podczas enumeracji.

```
var query = numbers.Where(n => n > 5); //NIC jeszcze się nie wykonuje

foreach (var n in query) // <-- Wszystko zaczyna się dziać, dopiero tutaj
    Console.WriteLine(n);

//Ewentualnie możemy:
var filtered = query.ToList(); // <-- Wrzucając elementy na listę, wykonujemy zapytanie od razu
```

LINQ – operator `Where()`

Operator (*metoda w zasadzie*) `Where()` przyjmuje jako parametr `Func<T, bool>`, a jego zadaniem jest filtrowanie sekwencji – **wybiera tylko te elementy, które spełniają zadany warunek**.

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

var even = numbers.Where(n => n % 2 == 0);

foreach (var n in even)
    Console.WriteLine(n);

//Output: 2 4
```

LINQ – operator `Select()`

Operator `Select()` jest **odpowiedzialny za projekcję elementów w sekwencji** – zmienia każdy element, w coś innego – mapuje dane. Metoda jako parametr przyjmuje `Func<T, TResult>`.

```
var words = new[] { "Jan", "znowu", "marudzi" };

var lengths = words.Select(w => w.Length);

foreach (var l in lengths)
    Console.WriteLine(l);

//Output: 3 5 7
```

LINQ – operator **`OrderBy()`**

Operator `OrderBy()` **sortuje sekwencję według klucza**. Po użyciu `OrderBy()` sekwencja zmienia się z `IEnumerable<T>` na **`IOrderedEnumerable<T>`**.

Alternatywą do `OrderBy()` jest:

- **`OrderByDescending()`** – sortuje malejąco.

Rozszerzeniami są:

- **`ThenBy()`** – dokładniejsze sortowanie, korzystając z wielu kluczy.
- **`ThenByDescending()`** – malejąco (*wg drugiego klucza*).

```
var numbers = new[] { 5, 2, 9, 1 };

var sorted = numbers.OrderBy(n => n);

foreach (var n in sorted)
    Console.WriteLine(n);

//Output: 1 2 5 9
```

LINQ – operator **ThenBy()**

```
var people = new[]
{
    new { Name = "Adam", Age = 25 },
    new { Name = "Ewa", Age = 22 },
    new { Name = "Adam", Age = 19 },
    new { Name = "Kasia", Age = 30 }
};

var sorted = people
    .OrderBy(p => p.Name)       // 1. po imieniu
    .ThenBy(p => p.Age);        // 2. po wieku

foreach (var p in sorted)
    Console.WriteLine($"{p.Name} - {p.Age}");
```

```
Adam - 19
Adam - 25
Ewa - 22
Kasia - 30
```

LINQ – operator **GroupBy()**

Operator `GroupBy()` pozwala na **dzielenie sekwencji na grupy** – według wskazanego klucza.

```
var people = new[]
{
    new { Name = "Adam", City = "Gdańsk" },
    new { Name = "Ewa", City = "Gdynia" },
    new { Name = "Kasia", City = "Gdańsk" }
};

var grouped = people.GroupBy(p => p.City);

foreach (var group in grouped)
{
    Console.WriteLine($"Miasto: {group.Key}");
    foreach (var person in group)
        Console.WriteLine($"  - {person.Name}");
}
```

```
Miasto: Gdańsk
  - Adam
  - Kasia
Miasto: Gdynia
  - Ewa
```

LINQ – operator `Join()`

Operator `Join()` **paruje elementy dwóch sekwencji** na podstawie **zgodnych kluczy**.

```
var users = new[]
{
    new { Id = 1, Name = "Adam" },
    new { Id = 2, Name = "Ewa" }
};

var orders = new[]
{
    new { UserId = 1, Product = "Laptop" },
    new { UserId = 1, Product = "Myszka" },
    new { UserId = 2, Product = "Monitor" }
};

var result =
    users.Join(
        orders,
        u => u.Id,          // <-- klucz z users
        o => o.UserId,      // <-- klucz z orders
        (u, o) => new { u.Name, o.Product }); // <-- projekcja do typu anonimowego

foreach (var row in result)
    Console.WriteLine($"{row.Name} kupił: {row.Product}");
```

```
Adam kupił: Laptop
Adam kupił: Myszka
Ewa kupiła: Monitor
```

LINQ – operator **`Union()`**

Operator `Union()` **łączy dwie sekwencje**, zwracając **unikalne elementy**.

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

var result = a.Union(b);

foreach (var n in result)
    Console.WriteLine(n);

//Output: 1 2 3 4 5
```

Działa na podstawie równości elementów – `Equals()` i `GetHashCode()`

LINQ – operator `Intersect()`

Operator `Intersect()` zwraca tylko elementy, które **występują w obu sekwencjach**.

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

var result = a.Intersect(b);

foreach (var n in result)
    Console.WriteLine(n);

//Output: 1 2
```

Działa na podstawie równości elementów – `Equals()` i `GetHashCode()`

LINQ – operator **`Except()`**

Operator `Except()` zwraca tylko te elementy, które są dostępne w sekwencji źródłowej (**a**), a nie są obecne w innej sekwencji (**b**).

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

var result = a.Except(b);

foreach (var n in result)
    Console.WriteLine(n);

//Output: 1 2
```

Działa na podstawie równości elementów – `Equals()` i `GetHashCode()`

LINQ – operator **Count()**

Operator `Count()` **zlicza elementy w sekwencji**.

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

//Pobieramy wynik bez odroczonego wykonania
int count = numbers.Count();

Console.WriteLine(count); // Output: 4
```

Zastosowanie tego operatora kończy się wynikiem – brak odroczonego wykonania.

LINQ – operator **Sum()**

Operator `Sum()` oblicza **sumę** wszystkich elementów w kolekcji.

```
var numbers = new[] { 1, 2, 3, 4 };
int sum = numbers.Sum();

//Działa też na obiektach, np:
//orders.Sum(o => o.Stake);

Console.WriteLine(sum); //Output: 10
```

Zastosowanie tego operatora kończy się wynikiem – brak odroczonego wykonania.

LINQ – operator **Average()**

Operator `Average()` oblicza **średnią** z wartości elementów w sekwencji.

```
var numbers = new[] { 1, 2, 3, 4 };
double avg = numbers.Average();

//Podobnie jak Sum() działa też na obiektach.
//Wynikiem Average() jest double.

Console.WriteLine(avg); // Output: 2.5
```

Zastosowanie tego operatora kończy się wynikiem – brak odroczonego wykonania.

LINQ – inne wybrane operatory

Operator	Opis
`Min(), Max()` *(zwraca wynik)*	Zwraca najmniejszą / największą wartość z sekwencji obiektów porównywalnych.
`MinBy(), MaxBy()` *(zwraca wynik)*	Zwraca najmniejszą / największą wartość z sekwencji, obsługuje selektor.
`Any(), Any(predicate)` *(zwraca wynik)*	Sprawdza czy w sekwencji istnieje jakikolwiek element.
`All(predicate)` *(zwraca wynik)*	Sprawdza czy w sekwencji każdy element spełnia zadany warunek.
`Aggregate(func)`	Pozwala utworzyć własną funkcję agregującą.
`Distinct()`	Zwraca unikalne elementy z sekwencji (*bez duplikatów*)
`Take(n)`	Pobiera pierwsze **n** elementów.
`Skip(n)`	Pomija pierwsze **n** elementów.
`TakeWhile(predicate)`	Pobiera elementy dopóki jest spełniony warunek
`SkipWhile(predicate)`	Pomija elementy dopóki jest spełniony warunek

LINQ – operatory kończące

LINQ działa leniwie, czyli nic się nie wykonuje, **dopóki nie poprosisz o wynik**. Zapytanie jest uruchamiane dopiero w momencie enumeracji (*odroczone*) bądź **po użyciu operatora kończącego**.

Operatory kończące przerywają odroczone wykonanie i zmuszają LINQ do zwrócenia wyniku teraz.

Operator	Przykład	Opis
`First()`	`numbers.First()`	Zwraca pierwszy element, gdy brak elementów – rzuca wyjątek.
`FirstOrDefault()`	`numbers.FirstOrDefault()`	Zwraca pierwszy element lub **`default(T)`** gdy sekwencja jest pusta.
`Single()`	`items.Single()`	Oczekuje dokładnie jednego elementu, w przeciwnym razie rzuca wyjątek.
`SingleOrDefault()`	`items.SingleOrDefault()`	Zwraca pojedynczy element lub **`default(T)`** gdy sekwencja ma inną liczbę elementów niż 1.
`Last()`	`numbers.Last()`	Zwraca ostatni element sekwencji, gdy brak elementów – rzuca wyjątek.
`LastOrDefault()`	`numbers.LastOrDefault()`	Zwraca ostatni element sekwencji lub **`default(T)`** gdy sekwencja jest pusta.

LINQ – operatory kończące

Operator	Przykład	Opis
`ToArray()`	`numbers.ToArray()`	Zwraca tablicę z elementami sekwencji.
`ToList()`	`numbers.ToList()`	Zwraca listę z elementami sekwencji.
`ToDictionary()`	`students.ToDictionary(k => k.Id, v => v);`	Zwraca słownik z elementami sekwencji (klucz i wartość)

Dopóki nie użyjesz jednej z metod kończących, **zapytanie nie zostanie przetworzone** – dopiero one kończą odroczone wykonanie, i zmuszają LINQ do przetworzenia kolekcji.

LINQ - pułapki

LINQ z założenia jest leniwe, to znaczy, że dopóki nie użyjesz operatora kończącego, **zapytanie nie jest wykonane**. **Daje to dużą elastyczność**, ale czasami prowadzi do **kosztownych błędów – możesz nieświadomie wykonać to samo zapytanie wiele razy**.

```
var query = numbers.Where(n => n > 5);

foreach (var n in query)
    Console.WriteLine(n);

var list = query.ToList();  // drugi raz wykonuje to samo filtrowanie!
```

❌ Źle

✅ Dobrze

```
var list = numbers.Where(n => n > 5).ToList();

foreach (var n in list)
    Console.WriteLine(n);

// dalsza praca na liście bez ponownego wykonywania LINQ
```

LINQ - pułapki

❌ Źle

```
var query = numbers.Where(n => n > 5);

int c1 = query.Count();  // pełne wykonanie zapytania!
int c2 = query.Count();  // …i jeszcze raz!
```

✅ Dobrze

```
var list = query.ToList();
int c = list.Count;      // szybkie – właściwość listy
```

Uwaga:
LINQ wykonuje zapytanie za każdym razem, gdy pobierasz wynik – aby uniknąć podwójnej pracy, materializuj sekwencję (`ToList()`) wtedy, gdy używasz jej więcej niż raz.

Interfejs `IQueryable<T>`

Interfejs **`IQueryable<T>`** to bliźniaczy interfejs do `IEnumerable<T>`, używany przede wszystkim przez **Entity Framework Core** i inne ORM (*Object-Relational Mapping*). Umożliwia nie tylko przetwarzanie danych, ale przede wszystkim **budowanie drzewa wyrażeń**, które następnie jest **tłumaczone na SQL**.

- **`IEnumerable<T>`** - działa na kolekcjach w pamięci.
- **`IQueryable<T>`** - działa na zdalnych źródłach danych (*bazach danych*)

Interfejs `IQueryable<T>`

IQueryable<T> nie wykonuje operacji od razu, zamiast tego:

1. Zbiera operacje (`where`, `select`, `orderby`).
2. Zapisuje je jako wyrażenie „drzewa” (*expression tree*).
3. Dostawca usług (*provider*) tłumaczy drzewo na SQL (*dostawca czyli sterownik, np. Npgsql, MS-SQL*).
4. Wykonywane jest zapytanie SQL w bazie danych.
5. Wynik zapytania trafia do aplikacji.

```
var adults = _db.Users.Where(u => u.Age > 18);
```

Na tym przykładzie `adults` to `IQueryable<User>`, na tym etapie **nie została jeszcze przeprowadzona żadna operacja na bazie danych**.

Interfejs `IQueryable<T>`

`IQueryable<T>` otrzymujemy przez odwołanie się do **kontekstu bazy danych**, a konkretnie do **jego tabel**. **Każdy `DbSet<T>` w Entity Framework Core implementuje `IQueryable<T>`**.

```
//Kontekst bazy danych
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions options) : base(options)
    {

    }

    //Tabele w bazie danych
    public override DbSet<ApplicationUser> Users { get; set; }
    public DbSet<Break> Breaks { get; set; }
    public DbSet<Invitation> Invitations { get; set; }
    public DbSet<Request> Requests { get; set; }

    //...
}
```

```
DbSet<T> : IQueryable<T>, IEnumerable<T>
```

LINQ do SQL

Przy pracy z bazą danych (**EF Core**) LINQ **nie wykonuje zapytań od razu**. Tak jak w LINQ do kolekcji – operacje typu `where`, `select`, `orderby` tylko **budują plan**. Zapytanie wykona się dopiero, gdy faktycznie poprosisz o dane.

```
//ToList() i ToArray() wykonują całe zapytanie
var users = db.Users.Where(u => u.Age > 18).ToList();
```

```
//First(), FirstOrDefault() = SELECT TOP(1)
var user = db.Users.First(u => u.Id == 5);
```

```
//Zapisujemy konkretną wartość do zmiennej - zapytanie się wykona
var count = db.Users.Where(u => u.IsActive).Count();
```

Każde z powyższych przykładów wykona zapytanie SQL

LINQ do SQL

```
public List<InvitationViewModel> GetAll()
{
    //Odwołujemy się do bazy danych
    List<InvitationViewModel> invitationViewModels = _db.Invitations
        .Select(x => new InvitationViewModel // <-- Projekcja
        {
            Id = x.Id,
            EventName = x.EventName,
            EventDate = x.EventDate
        }).ToList(); // <-- Wykonanie zapytania

    return invitationViewodels;
}
```

Tłumaczenie na SQL

```
-- Zapytanie
SELECT Id, EventName, EventDate FROM Invitations;
```

LINQ do SQL – pętla foreach

Jeżeli nie użyjemy operatora kończącego (*takiego jak `ToList()` czy `ToArray()`*), a zaczniemy przetwarzać wyniki w pętli `foreach`, **zapytanie SQL zostanie wysłane do bazy danych dokładnie w momencie wejścia do tej pętli**.

Dzieje się tak dzięki mechanizmowi **odroczonego wykonania**. Sama **zmienna z zapytaniem LINQ nie przechowuje danych**, lecz "*przepis*" na to, jak je pobrać.

Dopiero **gdy `foreach` zażąda pierwszego elementu**, Entity Framework tłumaczy ten przepis na SQL i wykonuje go na bazie.

```
//query to jest nasz "przepis" (nic nie jest jeszcze wysłane do DB)
var query = db.Users.Where(u => u.Age > 18);

foreach (var user in query) // <-- Dopiero tutaj idzie zapytanie SQL
    Console.WriteLine(user.Name);
    //Przy każdej iteracji foreach, przetwarzamy kolejny rekord z bazy
```

LINQ do SQL – zapytania dynamiczne

Jedną z największych zalet `IQueryable<T>` jest to, że **możesz dokładać warunki do zapytania w biegu** – tak jakbyś składał SQL z klocków. Dopóki nie wykonasz zapytania (*`ToList()`*, *`First()`*, *`foreach`*), możesz je dowolnie modyfikować.

To podstawa zaawansowanych filtrów, wyszukiwarek i API.

```
//1. Tworzymy szkielet zapytania
var query = db.Users.AsQueryable();

//2. W kolejnych krokach dodajemy warunki do zapytania
if (!string.IsNullOrWhiteSpace(name))
{
    query = query.Where(u => u.Name.Contains(name));
}

if (minAge.HasValue)
{
    query = query.Where(u => u.Age >= minAge.Value);
}

if (isActiveOnly)
{
    query = query.Where(u => u.IsActive);
}

//3. Wykonujemy zapytanie
var result = query.ToList();   // SQL wykonuje się dopiero tutaj!
```

LINQ do SQL – operacje nietłumaczalne

Nie wszystko, co zadziała na kolekcjach w pamięci (`IEnumerable`), da się przetłumaczyć na SQL. EF Core musi zamienić Twoje wyrażenie na **konkretne instrukcje SQL**, więc operacje, których SQL nie rozumie, zakończą się błędem – zazwyczaj „*`System.InvalidOperationException`: LINQ expression could not be translated*”

```
var users = db.Users
    .Where(u => DoSomething(u.Name)) // ❌ SQL nie zna tej metody
    .ToList();
```

Nie możemy w zapytaniu użyć metody, która jest zdefiniowana w kodzie.

LINQ do SQL – operacje nietłumaczalne

Jeżeli LINQ nie jest w stanie przetłumaczyć danego wyrażenia do SQL, możemy:

- Wykonać część operacji **po otrzymaniu wyników z bazy** – na `IEnumerable<T>`
- Użyć **wbudowanych funkcji SQL**.

Czy daną funkcję da się przetłumaczyć, **zależy w głównej mierze od sterownika bazy danych do Entity Framework Core** – inne funkcje będzie obsługiwał sterownik dla SQL Server, inne dla PostgreSQL (npgsql)

```
var users = db.Users
    .ToList()                       // wykonaj SQL
    .Where(u => DoSomething(u));  // dokończ w pamięci
```

```
//Korzystamy z wbudowanych funkcji
db.Users.Where(u => EF.Functions.Like(u.Name, "%adam%"));
```

Entity Framework Core

ORM (*Object–Relational Mapper*) to narzędzie, które **pozwala pracować z bazą danych tak jak z obiektami**, zamiast ręcznie pisać SQL. Nie trzeba myśleć w kategoriach tabel, kolumn i joinów – pracujesz na klasach, właściwościach i relacjach obiektów. **Najpopularniejszym ORM-em w .NET jest Entity Framework Core**.

```
//Bez ORM wyglądałoby to tak:
const QUERY = "SELECT * FROM Users WHERE Age > 18;"

var cmd = new SqlCommand(QUERY);
var reader = cmd.ExecuteReader();

//...
```

```
//Dzięki EF Core, wygląda to tak:
var users = db.Users.Where(u => u.Age > 18).ToList();
```

Kontekst bazy danych

Kontekst bazy danych (`DbContext`) to pewnego rodzaju serce EF Core, to właśnie on:

- Zarządza połączeniem z bazą danych.
- Mapuje encje na tabele.
- Śledzi zmiany.
- Wykonuje zapytania SQL.
- Zapisuje dane.

```
//Dodawanie danych do tabeli
_db.Users.Add(user);
_db.SaveChanges(); // <-- Zawsze musimy zapisać zmiany
```

```
using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext // <-- Dziedziczymy po DbContext
{
    //Mapujemy tabele
    public DbSet<User> Users { get; set; }
    public DbSet<Order> Orders { get; set; }

    //Wczytujemy konfigurację
    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options)
    {
    }
}
```

Podejścia do bazy danych w EF Core

W Entity Framework Core istnieją **trzy główne nurty, opisujące podejście do bazy danych** – czyli w jaki sposób chcemy zarządzać zmianami w bazie danych i kodzie.

- **Code first** – najpierw tworzysz klasy, baza danych generuje się na ich podstawie. Aktualizacje schematu tabel odbywają się przez wbudowany do EF Core mechanizmu migracji.
- **Database first** – kod jest tworzony do istniejącej bazy danych.
- **Inżynieria wsteczna (*scaffold*)** – kod jest generowany na podstawie schematu bazy danych.