Wyjątki, kolekcje, typy generyczne, delegaty i zdarzenia

Bartosz Miąskowski

Wyjątki

Wyjątek to obiekt reprezentujący sytuację błędną, która uniemożliwia normalne wykonanie programu.
Powstaje automatycznie lub jest zgłaszany ręcznie (**`throw`**), gdy kod napotyka sytuację, której nie potrafi obsłużyć.

Wyjątek składa się z:

- typu (*np.* *`NullReferenceException`*),
- komunikatu – krótki opis, co się wydarzyło,
- śladu (*stack trace*) – lista wywołanych metod, prowadzących do błędu.

Wyjątki są naturalną częścią programu – pozwalają reagować na błędy, w przewidywalny i kontrolowalny sposób.

Przykładowe wyjątki

```
//Przykład I - dzielenie przez zero
int x = 10;
int y = 0;

Console.WriteLine(x / y); //DivideByZeroException

//Przykład II - odwołanie się do obiektu, który jest null
string text = null;
Console.WriteLine(text.Length); // NullReferenceException

//Przykład III - wyjście poza indeks tablicy
var numbers = new int[] { 1, 2, 3 };
Console.WriteLine(numbers[5]);  // IndexOutOfRangeException
```

Wniosek:
Wyjątek to sygnał, że **program nie może kontynuować pracy** – dzięki nim **wiem dokładnie, co poszło nie tak**.

Obsługa wyjątków – `try` i `catch`

Do obsługi wyjątków **najczęściej** używamy słów kluczowych **`try`** oraz **`catch`**.

- **`try`** – blok kodu, w którym może wystąpić wyjątek.
- **`catch`** – przechwytuje wyjątek i pozwala na niego zareagować, nie dopuszczając do awarii całego programu.

```
try
{
    int x = 10;
    int y = 0;

    Console.WriteLine(x / y);
}

catch
{
    Console.WriteLine("Nie można dzielić przez zero!");
}
```

Uwaga!
Blok `try`, nigdy nie może występować samodzielnie – zawsze musi mu towarzyszyć blok `catch` lub `finally`.

Obsługa wyjątków – `catch (Exception ex)`

Blok **`catch (Exception ex)`** to najbardziej ogólny sposób przechwytywania błędów. Łapie każdy wyjątek, niezależnie od jego typu.

```
try
{
    string text = null;
    Console.WriteLine(text.Length); // NullReferenceException
}

catch (Exception ex)
{
    Console.WriteLine("Wystąpił błąd!");
    Console.WriteLine(ex.GetType().Name);
    Console.WriteLine(ex.Message);
}

//Output:
//Wystąpił błąd!
//NullReferenceException
//Object reference not set to an instance of an object.
```

Obsługa różnych typów wyjątków

W C# można dodać wiele bloków **catch**, aby inaczej reagować na różne rodzaje błędów. Najbardziej szczegółowe wyjątki powinny znajdować się wyżej, a ogólny **catch (Exception)** – na końcu

```
//Jeżeli podasz tekst
Zły format liczby!
```

```
//Jeżeli wpiszesz 0
Nie można dzielić przez zero!
```

```
//Kiedy coś innego pójdzie nie tak...
Nieznany błąd: ...
```


```
try
{
    //FormatException
    Console.Write("Podaj liczbę: ");
    int number = int.Parse(Console.ReadLine()!);

    //DivideByZeroException
    int result = 10 / number;
    Console.WriteLine(result);
}

catch (FormatException)
{
    Console.WriteLine("Zły format liczby!");
}

catch (DivideByZeroException)
{
    Console.WriteLine("Nie można dzielić przez zero!");
}

catch (Exception ex)
{
    Console.WriteLine("Nieznany błąd: " + ex.Message);
}
```

Przechwytywanie obiektu wyjątku

W bloku `catch` możesz:

- **pobrać wyjątek do zmiennej** – `catch (FormatException ex)`
- **pominąć zmienną** – `catch (FormatException)`

Obie formy przechwytują *ten sam typ błędu* — różnica polega tylko na tym, czy potrzebujesz dostępu do szczegółów wyjątku.

```
try
{
    int number = int.Parse("abc");
}

catch (FormatException)
{
    Console.WriteLine("Zły format liczby!");
}

//Output:
//Zły format liczby!
```

```
try
{
    int number = int.Parse("abc");
}

catch (FormatException ex)
{
    Console.WriteLine("Zły format liczby!");
    Console.WriteLine("Szczegóły: " + ex.Message);
}

//Output:
//Zły format liczby!
//Szczegóły: Input string was not in a correct format.
```

Blok `finally`

Blok **`finally`** uruchamia się **zawsze**, niezależnie od tego, czy w **`try`** wystąpił wyjątek, czy nie. Służy głównie do **czyszczenia zasobów**: zamykania plików, połączeń, strumieni.

```
try
{
    Console.WriteLine("Otwieram połączenie...");
    int x = 10 / 0; // Błąd
}

catch (DivideByZeroException)
{
    Console.WriteLine("Błąd: dzielenie przez zero!");
}

finally
{
    Console.WriteLine("Zamykam połączenie (zawsze).");
}
```

Wniosek:
Blok **`finally`** to gwarancja, że **posprzątamy** – nawet jeśli coś pójdzie nie tak.

Blok `finally`

Nawet jeśli w **`try`** *lub* w **`catch`** wystąpi **`return`**, blok **`finally` i tak zostanie wykonany** – **`finally`** ma pierwszeństwo przed zakończeniem metody.

```
int Test()
{
    try
    {
        Console.WriteLine("TRY");
        return 1;
    }

    catch
    {
        return 2;
    }

    finally
    {
        Console.WriteLine("FINALLY");
    }
}

Console.WriteLine("Zwrócono: " + Test());

//Output:
//TRY
//FINALLY
//Zwrócono: 1
```

```
int Test()
{
    try
    {
        throw new Exception();
    }
    catch
    {
        Console.WriteLine("CATCH");
        return 5;
    }
    finally
    {
        Console.WriteLine("FINALLY");
    }
}

//Output:
//CATCH
//FINALLY
//5
```

Wniosek:
Blok **`finally`** wykonuje się zawsze, nawet gdy metoda próbuje wrócić (**`return`**). Jedyne, co go zatrzyma, to **`Environment.Exit()`** albo awaria procesu.

Wyjątki a wpływ na wydajność

```
[Benchmark]
public int ParseInteger_NoTryCatch()
{
    const string INPUT = "123";
    return Int32.Parse(INPUT);
}

[Benchmark]
public int ParseInteger_NoException()
{
    const string INPUT = "123";
    try
    {
        return Int32.Parse(INPUT);
    }

    catch
    {
        return 0;
    }
}
```

Czy wyjątki wpływają na wydajność?

```
[Benchmark]
public int ParseInteger_WithExceptionObject()
{
    const string INPUT = "abcd";
    try
    {
        return Int32.Parse(INPUT);
    }

    catch (FormatException ex)
    {
        return ex.Message.Length;
    }
}
```

```
[Benchmark]
public int ParseInteger_WithNoTypeException()
{
    const string INPUT = "abcd";
    try
    {
        return Int32.Parse(INPUT);
    }

    catch
    {
        return 0;
    }
}

[Benchmark]
public int ParseInteger_WithTypeException()
{
    const string INPUT = "abcd";
    try
    {
        return Int32.Parse(INPUT);
    }

    catch (FormatException)
    {
        return 0;
    }
}
```

Wyjątki a wpływ na wydajność

Konsola debugowania progra

```
// * Summary *

BenchmarkDotNet v0.15.6, Windows 11 (10.0.26200.7171)
AMD Ryzen 9 9950X3D 4.30GHz, 1 CPU, 32 logical and 16 physical cores
  [Host]             : .NET Framework 4.8.1 (4.8.9221.0), X64 RyuJIT VectorSize=256
  .NET 10.0          : .NET 10.0.0 (10.0.0, 10.0.25.52411), X64 RyuJIT x86-64-v4
  .NET Framework 4.8 : .NET Framework 4.8.1 (4.8.9221.0), X64 RyuJIT VectorSize=256
```

Wynik na granicy błędu pomiarowego

Method	Job	Runtime	Mean	Error	StdDev	Gen0	Allocated
ParseInteger_NoTryCatch	.NET 10.0	.NET 10.0	3.681 ns	0.0155 ns	0.0145 ns	-	-
ParseInteger_NoException	.NET 10.0	.NET 10.0	4.017 ns	0.0139 ns	0.0123 ns	-	-
ParseInteger_WithNoTypeException	.NET 10.0	.NET 10.0	1,124.220 ns	2.5754 ns	2.0107 ns	0.0095	480 B
ParseInteger_WithTypeException	.NET 10.0	.NET 10.0	1,116.638 ns	4.4237 ns	4.1379 ns	0.0095	480 B
ParseInteger_WithExceptionObject	.NET 10.0	.NET 10.0	1,113.447 ns	2.0313 ns	1.6963 ns	0.0095	480 B
ParseInteger_NoTryCatch	.NET Framework 4.8	.NET Framework 4.8	31.584 ns	0.3306 ns	0.3092 ns	-	-
ParseInteger_NoException	.NET Framework 4.8	.NET Framework 4.8	31.649 ns	0.2942 ns	0.2752 ns	-	-
ParseInteger_WithNoTypeException	.NET Framework 4.8	.NET Framework 4.8	7,902.219 ns	75.9117 ns	71.0079 ns	0.1221	834 B
ParseInteger_WithTypeException	.NET Framework 4.8	.NET Framework 4.8	7,824.721 ns	44.0855 ns	39.0807 ns	0.1221	834 B
ParseInteger_WithExceptionObject	.NET Framework 4.8	.NET Framework 4.8	7,859.730 ns	62.2341 ns	58.2138 ns	0.1221	834 B

Obiekt wyjątku

Obiekt wyjątku (**`Exception ex`**) zawiera **szczegółowe informacje o błędzie**, które można wykorzystać do logowania lub diagnostyki.

Najważniejsze właściwości:

- **`Message`** – krótki opis błędu.
- **`StackTrace`** – ścieżka wywołań prowadząca do błędu.
- **`InnerException`** – wyjątek źródłowy (*jeśli jeden błąd wynika z innego*).
- **`Source`** – nazwa assembly, w której powstał błąd.
- **`TargetSite`** – metoda, w której rzucono wyjątek.
- **`HResult`** – kod błędu systemowego (*rzadziej wykorzystywany*).

Obiekt wyjątku

```
public class Program
{
    public static void Main()
    {
        try
        {
            //FormatException
            int x = Int32.Parse("abc");
        }

        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
            Console.WriteLine(ex.StackTrace);
            Console.WriteLine(ex.Source);
            Console.WriteLine(ex.TargetSite);
            Console.WriteLine(ex.HResult);
        }
    }
}
```

```
The input string 'abc' was not in a correct format.
   at System.Number.ThrowFormatException[TChar](ReadOnlySpan`1 value)
   at System.Int32.Parse(String s)
   at Program.Main()
System.Private.CoreLib
Void ThrowFormatException[TChar](System.ReadOnlySpan`1[TChar])
-2146233033
```

Wniosek:
Obiekt wyjątku to **pełny raport diagnostyczny** - od komunikatu, przez metodę źródłową, aż po całą ścieżkę wywołań i pierwotną przyczynę (**`InnerException`**).

Owijanie wyjątków - `InnerException`

Właściwość **`InnerException`** to wyjątek **źródłowy** – pierwotna przyczyna błędu. Używa się go, gdy chcemy *owinąć* jeden wyjątek w inny i dodać własny kontekst (np. warstwę biznesową, nazwę operacji, opis).

```
try
{
    SaveUser("abc"); // <-- Metoda rzuci FormatException
}

catch (Exception ex)
{
    throw new Exception("Błąd podczas zapisywania użytkownika", ex);
}

void SaveUser(string ageText)
{
    int age = Int32.Parse(ageText); // <-- FormatException
}

//Output:
//Exception: Błąd podczas zapisywania użytkownika
//InnerException: FormatException
//Message: Input string was not in a correct format.
```

Wniosek:
Właściwość **`InnerException`** pozwala śledzić *całą historię błędu* — od miejsca, gdzie naprawdę powstał, do miejsca, gdzie został przechwycony i opakowany.

Rzucanie wyjątku - **`throw`**

Słowo kluczowe **`throw`** służy do **sygnalizowania błędu**, gdy metoda nie może wykonać swojego zadania. Rzucamy wyjątek, gdy **dane wejściowe są nieprawidłowe**, **operacja jest niemożliwa**, albo **naruszone zostają zasady logiki biznesowej**.

Proste rzucanie wyjątku:

```
throw new Exception("Coś poszło nie tak");
```

Rzucanie wyjątku o konkretnym typie:

```
throw new ArgumentNullException(nameof(name));
throw new InvalidOperationException("Nie można wykonać operacji w tym stanie.");
throw new ArgumentOutOfRangeException(nameof(age));
```

Rzucanie wyjątku - **`throw`**

Rzucanie wyjątku (bardziej kompleksowy przykład):

```
void SetAge(int age)
{
    if (age < 0)
        throw new ArgumentOutOfRangeException(nameof(age), "Wiek nie może być ujemny.");

    Console.WriteLine("Zapisano wiek: " + age);
}
```

Uwaga!
Na powyższym przykładzie **`nameof()`** służy do pobrania nazwy zmiennej (**age**) – bywa to bardzo pomocne w przypadku refaktoryzacji.

Rzucanie wyjątku – `throw` vs `throw ex`

Oba polecenia *ponownie wyrzucają* wyjątek, ale robią to **inaczej**:

- **`throw ex`** – tworzy *nowy* wyjątek i **gubi oryginalny StackTrace**.
- **`throw`** – wyrzuca *ten sam* wyjątek ponownie, **zachowując pełną historię błędu**.

```
try
{
    int x = Int32.Parse("abc");
}

catch (Exception ex)
{
    throw ex; //❌ gubi oryginalny StackTrace
}

//StackTrace:
//at Program.Main() in Program.cs:line 12
```

```
try
{
    int x = int.Parse("abc");
}

catch (Exception ex)
{
    throw; //✅ zachowuje oryginalny StackTrace
}

//StackTrace:
//at System.Int32.Parse(...)
//at Program.Main() in Program.cs:line 5
```

Kiedy powinniśmy rzucać wyjątek?

Wyjątki powinniśmy wyrzucać:

- gdy **metoda nie może wykonać swojego zadania**,
- gdy dane są **niepoprawne lub niespójne**,
- gdy chcemy **jasno przerwać wykonywanie**, zamiast zwracać błędne wartości,
- gdy w logice domenowej coś jest **niemożliwe** (np. płacenie ujemnej kwoty),
- gdy **błąd ma znaczenie w innym miejscu** i ktoś musi go obsłużyć.

Wniosek:
Słowo kluczowe **`throw`** to sposób na powiedzenie: *„coś jest naprawdę nie tak – nie mogę kontynuować”*. Lepsze jedno dobrze rzucone **`throw`**, niż tysiąc cichych błędów – nawet jeżeli okupione wydajnością.

Połykane wyjątki – `catch { }`

```
try
{
    int x = Int32.Parse("abc"); //FormatException
}

catch
{
    //❌ Błąd znika – nie wiemy, co się stało
}
```

„***Połykany wyjątek***” to sytuacja, w której przechwytujesz błąd… **ale nic z nim nie robisz.** Brak logowania, brak reakcji, brak informacji – błąd ginie bez śladu.

Co tutaj się dzieje?

- program działa **jakby nic się nie stało**,
- debugowanie staje się **koszmarem**,
- w większej aplikacji może to prowadzić do **cichych, trudnych do wykrycia błędów**,
- logika programu może wykonać się w stanie **niepoprawnym**.

Wniosek:

- Blok **`catch { }`** to ukrywanie problemu. Ukryty błąd jest **gorszy niż błąd**, bo nie wiadomo, że w ogóle istnieje. Warto go zalogować, lub obsłużyć.
- Samo **`catch { }`** przydaje się głównie przy tworzeniu prostych testów i bardzo prostych aplikacji – np. przypominających skrypty.

Tworzenie własnych wyjątków

Własna klasa wyjątku pozwala nadać błędom **konkretny, domenowy sens**. Używamy ich, gdy standardowe wyjątki (*obecne w .NET*) nie opisują dobrze problemu.

```
public class InvalidAgeException : Exception
{
    public InvalidAgeException(string message) : base(message)
    {

    }
}
```

```
void SetAge(int age)
{
    if (age < 0)
        throw new InvalidAgeException("Wiek nie może być ujemny.");

    Console.WriteLine("Zapisano wiek: " + age);
}
```

```
//Output:
InvalidAgeException: Wiek nie może być ujemny.
```

Tworzenie własnych wyjątków

Możesz rozszerzyć własny wyjątek o **dodatkowe pola i właściwości**, które opisują błąd dokładniej niż zwykłe Message. To świetne rozwiązanie przy błędach domenowych.

```
public class InvalidOrderAmountException : Exception
{
    public decimal Amount { get; }
    public decimal Minimum { get; }

    public InvalidOrderAmountException(decimal amount, decimal minimum)
        : base($"Kwota {amount} jest za niska. Minimalna wartość to {minimum}.")
    {
        Amount = amount;
        Minimum = minimum;
    }
}
```

```
void PlaceOrder(decimal amount)
{
    if (amount < 100)
        throw new InvalidOrderAmountException(amount, 100);

    Console.WriteLine("Zamówienie przyjęte!");
}

//Output:
//InvalidOrderAmountException:
//Kwota 50 jest za niska. Minimalna wartość to 100.
```

```
try
{
  PlaceOrder(99);
}

catch (InvalidOrderAmountException ex)
{
  Console.WriteLine($"Twój amount: {ex.Amount}");
  Console.WriteLine($"Twój minimum: {ex.Minimum}");
}

//Output:
//Twój amount: 99
//Twój minimum: 100
```

Wniosek:
Dodatkowe właściwości sprawiają, że wyjątek staje się bogatszy w kontekst, a obsługa błędu – precyzyjna i czytelna (np. logowanie, komunikaty, decyzje biznesowe).

Tworzenie własnych wyjątków

Kiedy warto tworzyć własne wyjątki?

- gdy logika biznesowa **wymaga konkretnego typu błędu**,
- gdy chcesz, by inny programista mógł **łatwo rozpoznać przyczynę** (np. `InvalidTokenException`),
- gdy operacja jest poprawna, ale **domena** mówi, że nie wolno jej wykonać (np. "*saldo nie może być ujemne*").

Własne wyjątki nadają błędom *język domeny*, dzięki czemu kod staje się klarowny i łatwiejszy w utrzymaniu.

Kolekcje – wprowadzenie

Dlaczego tablice (*array*) już nie wystarczają?

1. **Tablice mają stały rozmiar** – raz ustalony rozmiar tablicy, pozostaje na zawsze – nie da się jej zwiększyć ani zmniejszyć.
2. **Brak podstawowych metod** – tablice nie mają takich metod jak `.Add()`, `.Remove()`, `.Contains()`, `.Sort()`.
3. **Słabo się skalują** – przy większych projektach, tablice szybko stają się niewygodne. Powodują konieczność ręcznego zarządzania ich rozmiarem – utrudnia to pisanie i utrzymanie aplikacji.

Tablice są **szybkie**, ale zbyt **prymitywne** do większości realnych zadań – brakuje im elastyczności i wygody.

Kolekcje – wprowadzenie

Kolekcje to elastyczne „*pojemniki*” na dane, które automatycznie rosną i dają gotowe metody do pracy z elementami.

```
int[] numbers = new int[3];      //sztywny rozmiar

List<int> list = new List<int>(); //rośnie dynamicznie
list.Add(10);
list.Add(20);
```

Kolekcje dawniej – `System.Collections`

Krótki rys historyczny:

Dawne kolekcje działały na typie **`object`**, więc wszystko trzeba było rzutować ręcznie — to powodowało błędy i spadek wydajności. Brak kontroli typów to tylko problemy w czasie działania programu.

```
ArrayList list = new ArrayList();
list.Add(10);
list.Add("tekst"); //działa, ale jest problem…

int x = (int)list[1]; //❌ InvalidCastException

//Output:
//Unhandled exception: InvalidCastException
```

Kolekcje generyczne – obecnie

Nowoczesne kolekcje działają na **konkretnych typach**, więc kompilator pilnuje błędów już na starcie. Są szybsze, bezpieczniejsze i eliminują rzutowania – „*typy wprost, zero zgadywania*”.

```
List<int> numbers = new List<int>();
numbers.Add(10);

//numbers.Add("abc"); <-- ❌ błąd kompilacji – i o to chodzi!
```

Typy generyczne

Typ generyczny to mechanizm pozwalający na tworzenie klas, metod, interfejsów i struktur, w których konkretny typ danych jest podawany jako **parametr** w momencie ich użycia. Pozwala to na stworzenie jednego, uniwersalnego szablonu kodu.

```
public class Box<T>
{
    public T Value { get; set; }
}

var intBox = new Box<int> { Value = 5 };
var textBox = new Box<string> { Value = "Hello" };

//intBox.Value = 5
//textBox.Value = "Hello"
```

Typy generyczne – konwencje

W przypadku typów generycznych stosuje się **krótkie i czytelne nazwy parametrów typu, zaczynające się od litery ‚T'**, aby od razu było jasne, *co oznacza typ*. Dzięki temu kod jest przewidywalny i zgodny ze standardami .NET.

```
//Najczęstsze konwencje:
T             //„typ ogólny”
TItem         //element kolekcji
TElement      //element w strukturach (np. PriorityQueue)
TValue        //wartość w słownikach
TKey          //klucz w słownikach
TResult       //wynik działania

//Przykłady:
Dictionary<TKey, TValue> dict = new();
PriorityQueue<TElement, TPriority> queue = new();
```

Typy generyczne – ograniczenia (*constraints*)

Ograniczenia pozwalają powiedzieć kompilatorowi, **jakiego typu** oczekuje generyk – dzięki temu kod jest bezpieczniejszy i może korzystać z cech tego typu – typy, które nie spełnią ograniczeń, nie będę obsługiwane.

```
//T musi być klasą i mieć konstruktor bezparametrowy
public class Factory<T> where T : class, new()
{
    public T Create() => new T();
}

//typ musi dziedziczyć po BaseClass
public class Storage<T> where T : BaseClass
{

}

//typ musi dać się porównać (implementować interfejs IComparable)
public class SortedBox<T> where T : IComparable<T>
{

}
```

```
Factory<string> f = new Factory<string>(); //✅ string ma new()
Factory<int> f2 = new Factory<int>();      //❌ int nie spełnia warunku class + new()
```

Podstawowe kolekcje generyczne

Kolekcje generyczne to gotowe pudełka na dane, każde do trochę innych zadań. Ważne jest, żeby wiedzieć ***kiedy które pudełko jest najlepsze***, i najlepiej spełni powierzone mu zadanie.

```
//Zwykła lista (zmieniająca rozmiar)
var list = new List<int> { 1, 2, 3 }; //[1, 2, 3]

//Słownik - klucz i wartość
var dict = new Dictionary<string, int>()
dict["HP"] = 120; //[HP] = 120

//Tylko elementy, które się nie powtarzają
var hset = new HashSet<int> { 1, 2, 2, 3 } //[1, 2, 3]

//Kolejka FIFO
var queue = new Queue<string>();
queue.Enqueue("A");
queue.Enqueue("B"); // --> "A" --> "B"

//Kolejka LIFO
var stack = new Stack<string>();
stack.Push("Top");
stack.Push("Bottom"); // --> "Bottom" --> "Top"

//Kolejka priorytetowa
var pq = new PriorityQueue<string, int>();
pq.Enqueue("Low", 10);
pq.Enqueue("High", 1); // --> "High" --> "Low"
```

Listy – `List<T>`

Listy to w pewnym sensie „*tablice na sterydach*” – **rosną w miarę potrzeb** oraz **posiadają masę przydatnych metod**. **W 90% przypadków** kiedy będzie nam potrzebna kolekcja, **skorzystamy właśnie z listy**.

Kiedy korzystać z list?

- Gdy **nie znasz z góry ilości** elementów (*np. lista uczniów, produkty w koszyku*).
- Gdy chcesz **dodawać, usuwać, sortować, wyszukiwać** elementy.
- Gdy potrzebujesz **dostępu po indeksie** (*lista[0], lista[1] itd.*).
- Gdy **dane są uporządkowane** (kolejność ma znaczenie).

Listy – `List<T>`

Jak działa lista?

- `List<T>` trzyma dane **w zwykłej tablicy** w środku.
- **Ma pojemność** (*capacity*) – gdy zabraknie miejsca, tworzy większą tablicę i kopiuje dane.
- Dostęp po indeksie jest **bardzo szybki O(1)**.
- Dodawanie na końcu jest zazwyczaj szybkie, **ale czasem drogie**, gdy musi powiększyć tablicę.
- Wstawianie/usuwanie ze środka listy wymaga **przesuwania elementów** (*wolniejsze*).

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // Tworzymy listę ocen (int)
        List<int> grades = new List<int>();

        // Dodawanie elementów
        grades.Add(5);
        grades.Add(4);
        grades.Add(3);

        // Wstawianie w środek
        grades.Insert(1, 6); // [5, 6, 4, 3]

        // Usuwanie
        grades.Remove(4);    // usuwa pierwszą 4 -> [5, 6, 3]

        // Dostęp po indeksie
        Console.WriteLine("Pierwsza ocena: " + grades[0]);

        // Iteracja
        Console.WriteLine("Wszystkie oceny:");
        foreach (var g in grades)
            Console.WriteLine(g);

        // Metody pomocnicze
        Console.WriteLine("Liczba ocen: " + grades.Count);
        Console.WriteLine("Czy jest 1? " + grades.Contains(1));
    }
}
```

Przykład korzystania z listy.

```
// Adds the given object to the end of this list. The size of the list is
// increased by one. If required, the capacity of the list is doubled
// before adding the new element.
//
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void Add(T item)
{
    _version++;
    T[] array = _items;
    int size = _size;
    if ((uint)size < (uint)array.Length)
    {
        _size = size + 1;
        array[size] = item;
    }
    else
    {
        AddWithResize(item);
    }
}

// Non-inline from List.Add to improve its code quality as uncommon path
[MethodImpl(MethodImplOptions.NoInlining)]
private void AddWithResize(T item)
{
    Debug.Assert(_size == _items.Length);
    int size = _size;
    Grow(size + 1);
    _size = size + 1;
    _items[size] = item;
}
```

Fragment kodu źródłowego `List<T>` z `System.Collections.Generic`

Pojemność listy

Lista rośnie dynamicznie, ale nie „*po jednym miejscu*”. Ma w środku ukrytą tablicę z pewną pojemnością. Gdy się ona skończy, lista **musi stworzyć większą tablicę i przekopiować całą zawartość**. To działa automatycznie, ale czasem może spowolnić działanie programu.

Jak to dokładnie działa?

- **`Count`** – ile faktycznie elementów jest na liście.
- **`Capacity`** – ile elementów może pomieścić zanim lista powiększy tablicę.
- Gdy **`Capacity`** się kończy, lista zwykle podwaja pojemność.

```
List<int> numbers = new List<int>();

for (int i = 0; i < 20; i++)
{
    numbers.Add(i);
    Console.WriteLine($"Dodano: {i}, Count={numbers.Count}, Capacity={numbers.Capacity}");
}

//Output:
//Dodano: 0, Count=1, Capacity=4
//Dodano: 1, Count=2, Capacity=4
//Dodano: 2, Count=3, Capacity=4
//Dodano: 3, Count=4, Capacity=4
//Dodano: 4, Count=5, Capacity=8   <-- powiększenie
//Dodano: 5, Count=6, Capacity=8
//Dodano: 6, Count=7, Capacity=8
//Dodano: 7, Count=8, Capacity=8
//Dodano: 8, Count=9, Capacity=16  <-- powiększenie
```

Czy pojemność listy ma znaczenie?

Konsola debugowania progra

```
// * Summary *

BenchmarkDotNet v0.15.6, Windows 11 (10.0.26200.7171)
AMD Ryzen 9 9950X3D 4.30GHz, 1 CPU, 32 logical and 16 physical cores
  [Host]             : .NET Framework 4.8.1 (4.8.9221.0), X64 RyuJIT VectorSize=256
  .NET 10.0          : .NET 10.0.0 (10.0.0, 10.0.25.52411), X64 RyuJIT x86-64-v4
  .NET Framework 4.8 : .NET Framework 4.8.1 (4.8.9221.0), X64 RyuJIT VectorSize=256


| Method                | Job                | Runtime            | Mean     | Error     | StdDev    | Gen0      | Gen1      | Gen2      | Allocated |
|---------------------- |------------------- |------------------- |---------:|----------:|----------:|----------:|----------:|----------:|----------:|
| WithCapacityChange    | .NET 10.0          | .NET 10.0          | 1.753 ms | 0.0321 ms | 0.0285 ms | 1980.4688 | 1980.4688 | 1980.4688 |      8 MB |
| WithoutCapacityChange | .NET 10.0          | .NET 10.0          | 1.097 ms | 0.0145 ms | 0.0136 ms |  998.0469 |  998.0469 |  998.0469 |   3.82 MB |
| WithCapacityChange    | .NET Framework 4.8 | .NET Framework 4.8 | 2.292 ms | 0.0451 ms | 0.0501 ms | 1996.0938 | 1996.0938 | 1996.0938 |   8.01 MB |
| WithoutCapacityChange | .NET Framework 4.8 | .NET Framework 4.8 | 1.715 ms | 0.0340 ms | 0.0318 ms |  998.0469 |  998.0469 |  998.0469 |   3.82 MB |

// * Hints *
Outliers
  ListCapacity.WithCapacityChange: .NET 10.0            -> 1 outlier  was  removed (1.88 ms)
  ListCapacity.WithCapacityChange: .NET Framework 4.8   -> 1 outlier  was  removed (2.80 ms)
  ListCapacity.WithoutCapacityChange: .NET Framework 4.8 -> 1 outlier  was  removed (2.06 ms)

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Gen0      : GC Generation 0 collects per 1000 operations
  Gen1      : GC Generation 1 collects per 1000 operations
  Gen2      : GC Generation 2 collects per 1000 operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
```

Tak, ma znaczenie… i to bardzo duże.

Pojemność listy

Wnioski .NET 10:

- **Wersja z ustaloną pojemnością jest szybsza o ok. 37%** - program nie musi tracić czasu na wielokrotne powiększanie listy i kopiowanie wszystkich jej elementów.
- Wersja z ustaloną pojemnością wykorzystała **ponad dwa razy mniej pamięci**.
- W wersji z ustaloną pojemnością GC interweniował rzadziej.

Jeśli lista nie wie, ile miejsca potrzebuje, to co chwilę musi wynajmować większe mieszkanie i przenosić cały dobytek.

Jeśli od razu mówisz jej: *„będziesz mieć milion elementów”*, od razu bierze odpowiednio duże mieszkanie i nie musi się przeprowadzać.

```
[Benchmark]
public void WithCapacityChange()
{
    var list = new List<int>();
    for (int i = 0; i < 1_000_000; i++)
    {
        list.Add(i);
    }
}

[Benchmark]
public void WithoutCapacityChange()
{
    var list = new List<int>(1_000_000);
    for (int i = 0; i < 1_000_000; i++)
    {
        list.Add(i);
    }
}
```

Słownik – `Dictionary<TKey, TValue>`

Czasem numer elementu (*indeks*) nas w ogóle nie obchodzi – chcemy **po prostu coś szybko znaleźć po kluczu**: po PESELu, loginie, numerze ucznia, numerze rejestracyjnym auta. Do tego służy Dictionary – **szukaj po kluczu, nie po indeksie**.

Kiedy warto używać słownika?

- Gdy chcesz **błyskawicznie znaleźć** wartość po kluczu (*np. ID -> uczeń*).
- Gdy klucze muszą być **unikalne** (*nie może być dwóch tych samych ID*).
- Gdy kolejność elementów nie jest najważniejsza, **ważne jest wyszukiwanie**.
- Pod spodem Dictionary to tablica z haszowaniem – dzięki temu typowe wyszukiwanie po kluczu jest bardzo szybkie.

```csharp
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // Klucz: numer dziennika, wartość: imię ucznia
        var students = new Dictionary<int, string>();

        students.Add(1, "Kuba");
        students.Add(7, "Ania");
        students[13] = "Ola";   // alternatywny zapis dodania / podmiany

        // Odczyt po kluczu
        Console.WriteLine(students[7]); // Ania

        // Bezpieczne pobieranie wartości
        if (students.TryGetValue(10, out var name))
        {
            Console.WriteLine($"Uczeń nr 10: {name}");
        }
        else
        {
            Console.WriteLine("Uczeń nr 10 nie istnieje.");
        }

        // Iteracja po parach klucz–wartość
        foreach (var pair in students)
        {
            Console.WriteLine($"Nr: {pair.Key}, Imię: {pair.Value}");
        }
    }
}
```

```
Ania
Uczeń nr 10 nie istnieje.
Nr: 1, Imię: Kuba
Nr: 7, Imię: Ania
Nr: 13, Imię: Ola
```

Przykład użycia **`Dictionary<TKey, Tvalue>`**

Słownik – wymagania co do klucza **`TKey`**

Aby klasa mogła być *kluczem* w słowniku, musi **mieć poprawnie zaimplementowane:**

- **`IEquatable.Equals<TKey>(TKey key)`** – porównywanie dwóch obiektów.
- **`GetHashCode()`** – obliczanie skrótu (*hasha*).

`Dictionary` **najpierw porównuje hash klucza**, a dopiero potem (*jeśli trzeba*) używa **`Equals()`**. Jeśli implementacja jest zła, słownik nie znajdzie elementu, który istnieje, duplikaty będą traktowane jako inne klucze, albo elementy trafią do złych kubełków.

```
public class Student : IEquatable<Student>
{
    public int Id { get; set; }
    public string Name { get; set; }

    public bool Equals(Student other)
        => other != null && Id == other.Id;

    public override bool Equals(object obj)
        => Equals(obj as Student);

    public override int GetHashCode()
        => Id.GetHashCode();
}
```

Lista bez duplikatów – `HashSet<T>`

`HashSet<T>` to kolekcja, która działa jak **zbiór matematyczny** – **może zawierać tylko unikalne wartości**. Jeśli spróbujesz dodać drugi raz to samo – po prostu to pominie. Pod spodem, podobnie jak Dictionary, używa **hashowania**, dzięki czemu **operacje są bardzo szybkie**.

Kiedy warto używać `HashSet<T>`?

- Gdy **nie chcesz duplikatów** (*np. unikalne loginy, tagi, ID*).
- Gdy najważniejsze jest **sprawdzanie *„czy element istnieje?”***.
- Gdy chcesz mieć **najszybszą możliwą** operację dodawania i sprawdzania – **O(1)**.
- Zwykle wtedy, gdy `Contains()` na liście byłoby zbyt wolne.

```csharp
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        var visitedRooms = new HashSet<string>();

        visitedRooms.Add("101");
        visitedRooms.Add("204");
        visitedRooms.Add("101"); // zignorowane – już istnieje

        Console.WriteLine("Lista odwiedzonych sal:");

        foreach (var room in visitedRooms)
        {
            Console.WriteLine(room);
        }

        Console.WriteLine("Czy byłeś w sali 204? " + visitedRooms.Contains("204"));
        Console.WriteLine("Czy byłeś w sali 999? " + visitedRooms.Contains("999"));
    }
}
```

```
Lista odwiedzonych sal:
101
204
Czy byłeś w sali 204? True
Czy byłeś w sali 999? False
```

Przykład użycia **HashSet<T>**

`HashSet<T>` - wymagania co do `T`

`HashSet<T>` działa tak samo jak `Dictionary`, ale bez wartości. Aby obiekt został poprawnie obsłużony przez `HashSet` potrzebuje poprawnie zaimplementowanego:

- **`Equals()`** – najlepiej przez `IEquatable<T>`
- **`GetHashCode()`**

Elementy które są umieszczane w `HashSet` muszą posiadać:

- Logiczne porównanie.
- Mieć stabilny hash (*nie zmieniający się w trakcie życia obiektu*)

Możesz wrzucać do `HashSet` studentów, ale **nie wolno zmieniać ich właściwości, od których zależy hash!**

```
var s = new Student { Id = 10 };
set.Add(s);

s.Id = 99;  //hash się zmienia - HashSet "gubi" obiekt
```

❌ ***Zły przykład – tak robić niewolno***

Kolejka FIFO – `Queue<T>`

FIFO – First in, First out

`Queue<T>` działa dokładnie tak jak kolejka do kasy w sklepie – „***kto pierwszy przyszedł, ten pierwszy zostaje obsłużony***”. Używamy jej wtedy, gdy ważna jest kolejność napływających elementów.

Kiedy warto używać `Queue<T>`?

- Gdy przetwarzasz dane **w tej samej kolejności**, w jakiej przyszły.
- Kolejki zadań, komunikatów, zapytań do serwera.
- Gdy chcesz prostego „*ruchu jednokierunkowego*” – dodajesz z jednej strony, odbierasz z drugiej.

Najważniejsze operacje:

- **`Enqueue(item)`** – dodaje na koniec kolejki.
- **`Dequeue()`** – pobiera i usuwa pierwszy element.
- **`Peek()`** – podgląda pierwszy element, ale go nie usuwa.

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        Queue<string> queue = new Queue<string>();

        queue.Enqueue("Kuba");
        queue.Enqueue("Ania");
        queue.Enqueue("Ola");

        Console.WriteLine("Pierwsza osoba w kolejce: " + queue.Peek());

        Console.WriteLine("Obsługa klientów:");
        while (queue.Count > 0)
        {
            string student = queue.Dequeue();
            Console.WriteLine("Obsłużono: " + student);
        }
    }
}
```

```
Pierwsza osoba w kolejce: Kuba
Obsługa klientów:
Obsłużono: Kuba
Obsłużono: Ania
Obsłużono: Ola
```

Przykład użycia **Queue<T>**

Stos – `Stack<T>`

LIFO – Last in, First out (*stos jest kolejką*)

`Stack<T>` działa jak stos talerzy w kuchni – **odkładasz na górę, a zdejmujesz też z góry**. Ostatni element, który dodasz, będzie pierwszy do pobrania.

Kiedy warto używać `Stack<T>`?

- Gdy potrzebujesz **odwrócić kolejność** (*np. odwracanie tekstu*).
- Gdy logika wymaga „zapamiętaj, a potem wróć” (*np. cofanie operacji, historia stron*).
- Gdy przetwarzasz zadania w stylu – **ostatnie przyszło, ale zrób jako pierwsze**.

Najważniejsze operacje:

- **`Push(item)`** – odkłada element na górę stosu.
- **`Pop()`** – pobiera i usuwa element z góry.
- **`Peek()`** – podgląda, co jest na górze.

```cpp
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        Stack<string> history = new Stack<string>();

        history.Push("google.com");
        history.Push("facebook.com");
        history.Push("youtube.com");

        Console.WriteLine("Obecna strona: " + history.Peek());

        Console.WriteLine("\nCofanie stron:");
        while (history.Count > 0)
        {
            string page = history.Pop();
            Console.WriteLine("Cofnięto do: " + page);
        }
    }
}
```

```
Obecna strona: youtube.com

Cofanie stron:
Cofnięto do: youtube.com
Cofnięto do: facebook.com
Cofnięto do: google.com
```

Przykład użycia **Stack<T>**

Kolejka priorytetowa – `PriorityQueue<TElement, TPriority>`

`PriorityQueue` to taka wersja kolejki, gdzie **kolejność nie zależy od czasu przyjścia**, ale od **priorytetu**. Element o mniejszym priorytecie (*np. 1*) zostanie obsłużony szybciej niż element z priorytetem np. 50.

Kiedy warto używać `PriorityQueue<Telement, TPriority>`?

- Gdy zadania mają różną wagę / priorytet.
- Gdy chcesz, żeby system sam decydował, które zadanie „*leci pierwsze*”.
- Algorytmy grafowe, np. Dijkstra.
- Systemy operacyjne, kolejki zadań.

Najważniejsze metody:

- **`Enqueue(element, priority)`** – dodawanie z priorytetem.
- **`Dequeue()`** – pobiera element o najwyższym priorytecie.
- **`Peek()`** – podgląda kolejny element.

```csharp
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        var queue = new PriorityQueue<string, int>();

        // element, priorytet (im mniejszy, tym ważniejszy)
        queue.Enqueue("Złamana noga", 1);     // priorytet wysoki
        queue.Enqueue("Ból głowy", 5);
        queue.Enqueue("Zadrapanie", 10);
        queue.Enqueue("Atak serca", 0);       // priorytet ekstremalny

        Console.WriteLine("Obsługa pacjentów:");
        while (queue.Count > 0)
        {
            Console.WriteLine(queue.Dequeue());
        }
    }
}
```

```
Obsługa pacjentów:
Atak serca
Złamana noga
Ból głowy
Zadrapanie
```

Przykład użycia **PriorityQueue<TElement, TPriority>**

Kolejka priorytetowa – wymagania `TPriority`

Aby dany obiekt mógł pełnić rolę priorytetu w kolejce priorytetowej **musi bezwzględnie implementować interfejs `IComparable<TPriority>`**.

Dzięki temu interfejsowi kolejka wie:

- Co jest „*mniejszym*” priorytetem.
- Co jest „*większym*” priorytetem.

Czyli może ustawiać poszczególne priorytety w odpowiedniej kolejności względem siebie.

```
public class TaskPriority : IComparable<TaskPriority>
{
    public int Value { get; set; } //im mniej, tym ważniejsze

    public int CompareTo(TaskPriority other)
        => Value.CompareTo(other.Value);
}
```

Kolekcje, a obsługa **`foreach`** i LINQ

Wszystkie z wymienionych kolekcji **obsługują `foreach` oraz LINQ** – implementują interfejs **`IEnumerable<T>`**.

W przypadku kolejek, **iteracja nie ściąga elementów z kolekcji** – pozostaną one tam nadal.

Kolekcja	`foreach`	LINQ	Uwagi
`List<T>`	✓	✓	pełna iteracja
`Dictionary<TKey, TValue>`	✓	✓	iteruje po parach klucz–wartość
`HashSet<T>`	✓	✓	kolejność przypadkowa
`Queue<T>`	✓	✓	FIFO
`Stack<T>`	✓	✓	LIFO
`PriorityQueue<TElement, TPriority>`	⚠	⚠	iteracja możliwa, ale nie po priorytetach

Interfejs `IEnumerable<T>`

Każda kolekcja, która chce współpracować z foreach, musi implementować interfejs **`IEnumerable<T>`**, który dostarcza metodę **`IEnumerator<T> GetEnumerator()`**. Metoda ta zapewnia obiekt, który potrafi przechodzić po elementach danej kolekcji.

`IEnumerator<T>`:

- Trzyma indeks / stan.
- Posiada metodę `MoveNext()` – idź do następnego elementu.
- Posiada właściwość `Current` – aktualny element.
- Posiada metodę `Reset()` – powrót do początku.
- Implementuje interfejs `IDisposable`.

Co tak naprawdę robi foreach?

Poniższy przykład pokazuje:

- Dlaczego foreach działa z wieloma kolekcjami.
- Dlaczego możesz pisać własne kolekcje, kompatybilne z foreach i LINQ.
- Jak działa wykonanie odroczone w LINQ.

```
foreach (var x in kolekcja)
{
    Console.WriteLine(x);
}
```

Kompilator zmienia foreach
mniej więcej na to

```
var enumerator = kolekcja.GetEnumerator();

while (enumerator.MoveNext())
{
    var x = enumerator.Current;
    Console.WriteLine(x);
}
```

Wykonanie odroczone – `yield return`

`yield return` to słowo kluczowe, dzięki któremu zwykła metoda może zachowywać się jak kolekcja. Nie musi tworzyć listy, tablicy ani niczego przechowywać – po prostu produkuje kolejne wartości na bieżąco (*jest to fundament wykonania odroczonego*).

Bez słowa kluczowego `yield`, gdy chcesz zwrócić wiele elementów musiałbyś:

1. Utworzyć nową listę.
2. Dodać do niej elementy.
3. Zwrócić listę.

Dzięki **`yield return`** możesz to osiągnąć bez żadnej listy – metoda **będzie zwracała elementy jeden po drugim**, **dokładnie wtedy, kiedy będą one potrzebne**.

Wykonanie odroczone – `yield return`

1. Tworzymy metodę, która zwraca elementy od 1 do 3

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

2. Do obsługi tej metody używamy pętli `foreach`

```
foreach (var n in CountToThree())
{
    Console.WriteLine(n);
}

//Output:
//1
//2
//3
```

W tym przykładzie nie ma żadnej listy.
Kompilator w tle utworzył całą klasę enumeratora.

Wykonanie odroczone – `yield return`

Wykonanie odroczone polega na tym, że **dana operacja jest wykonywana dopiero kiedy poprosimy o kolejny element**. Na podstawie poniższego przykładu – **dopiero po wyświetleniu n-tej liczby parzystej**, przechodzimy do poszukiwania kolejnej.

```
public static IEnumerable<int> EvenNumbers(int max)
{
    for (int i = 0; i <= max; i++)
    {
        Console.WriteLine($"Wartość i: {i}");
        if (i % 2 == 0)
        {
            yield return i;
        }
    }
}

foreach (var n in EvenNumbers(10))
{
    Console.WriteLine(n);
}
```

```
//Output:
Wartość i: 0 // <-- zmetody EvenNumbers()
0 // <-- z foreach
Wartość i: 1 // <-- zmetody EvenNumbers()
Wartość i: 2 // <-- zmetody EvenNumbers()
2 // <-- z foreach
Wartość i: 3 // <-- zmetody EvenNumbers()
Wartość i: 4 // <-- zmetody EvenNumbers()
4 // <-- z foreach
Wartość i: 5 // <-- zmetody EvenNumbers()
Wartość i: 6 // <-- zmetody EvenNumbers()
6 // <-- z foreach
Wartość i: 7 // <-- zmetody EvenNumbers()
Wartość i: 8 // <-- zmetody EvenNumbers()
8 // <-- z foreach
Wartość i: 9 // <-- zmetody EvenNumbers()
Wartość i: 10 // <-- zmetody EvenNumbers()
10 // <-- z foreach
```

Przerwanie wykonania – `yield break`

Czasami chcemy **ręcznie zakończyć enumerację w danym miejscu** – wtedy z pomocą przychodzi **`yield break`**.

```
public static IEnumerable<int> UntilFive()
{
    for (int i = 0; i < 10; i++)
    {
        if (i == 5)
            yield break; // przerwij sekwencję

        yield return i;
    }
}
```

*W tym przypadku bez `yield break`, metoda **zwracałaby kolejne wartości aż do przepełnienia**, które zakończyłoby się wyjątkiem.*

Własna kolekcja z `IEnumerable<T>`

Skoro **`foreach`** działa na kolekcjach, które **implementują `IEnumerable<T>`**, nic nie stoi na przeszkodzie, żeby **napisać swoją własną kolekcję, po której też można iterować**.

Najprościej stworzyć taką kolekcję korzystając z dobrodziejstw słowa kluczowego `yield`.

```
//Definiujemy własną kolekcję - kolekcja leniwa
public class NumberRange : IEnumerable<int>
{
    private readonly int _start;
    private readonly int _end;

    public NumberRange(int start, int end)
    {
        _start = start;
        _end = end;
    }

    public IEnumerator<int> GetEnumerator()
    {
        for (int i = _start; i <= _end; i++)
        {
            yield return i;
        }
    }

    //To jest pieśn przeszłości... ale musi być.
    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

Kod „legacy” – niegeneryczny

```
static void Main()
{
    var range = new NumberRange(3, 7);
    foreach (var number in range)
    {
        Console.WriteLine(number);
    }
}

//Output:
//3
//4
//5
//6
//7
```

Własna kolekcja z `IEnumerable<T>`

Jeżeli mamy taką ochotę *(bądź potrzebę)*, **możemy stworzyć pełną klasę enumeratora**, która będzie odpowiedzialna za iterację elementów w pętli `foreach`.

```
//Klasa enumeratora
public class SimpleEnumerator : IEnumerator<int>
{
    private readonly int[] _data;
    private int _index = -1;

    public SimpleEnumerator(int[] data)
    {
        _data = data;
    }

    public bool MoveNext()
        => return ++_index < _data.Length;

    public int Current => _data[_index];

    object IEnumerator.Current => Current;

    public void Reset() => _index = -1;

    public void Dispose() { /*...*/ }
}
```

```
//Nasza kolekcja
public class SimpleCollection : IEnumerable<int>
{
    private int[] _data = { 10, 20, 30, 40 };

    public IEnumerator<int> GetEnumerator()
    {
        //Zwraca obiekt enumeratora
        return new SimpleEnumerator(_data);
    }

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

```
class Program
{
    static void Main()
    {
        var coll = new SimpleCollection();
        foreach (var value in coll)
        {
            Console.WriteLine(value);
        }
    }
}

//Output:
//10
//20
//30
//40
```

Własna kolekcja z indeksatorem

Kiedy tworzymy własną kolekcję, dobrze byłoby aby w miarę możliwości umożliwiała ona dostęp do elementów wewnątrz, **przy pomocy indeksu** – tak, jak jest to w przypadku tablicy.

Możemy uzyskać taki efekt poprzez **implementację obsługi indeksatora** – jest to specjalna właściwość **`this[…]`**.

```
var item = myCollection[0];
myCollection[1] = 123;
```

Własna kolekcja z indeksatorem

Implementacja indeksatora jest bardzo prosta, wystarczy wewnątrz klasy utworzyć nową właściwość **bez nazwy**, przy pomocy słowa kluczowego `this` oraz zdefiniować **jakiego typu kolekcja będzie obsługiwała indeksy** (*wewnątrz nawiasów kwadratowych*).

Na co zwrócić uwagę?

- **`public int this[int index]`** – indeksator (*specjalna właściwość z **`this`** zamiast nazwy*).
- **W środku walidujemy index i rzucamy `ArgumentOutOfRangeException`**, jak w prawdziwych kolekcjach .NET.
- **Możemy mieć tylko `get`, albo `get + set`** (*jak w **`List<T>`***).

```
public class NumberList //Do obsługi indeksatora nie jest potrzebne IEnumerable
{
    private readonly int[] _data;

    public NumberList(params int[] items)
    {
        _data = items;
    }

    //Indeksator
    public int this[int index]
    {
        get
        {
            if (index < 0 || index >= _data.Length)
                throw new ArgumentOutOfRangeException(nameof(index));

            return _data[index];
        }
        set
        {
            if (index < 0 || index >= _data.Length)
                throw new ArgumentOutOfRangeException(nameof(index));

            _data[index] = value;
        }
    }

    public int Count => _data.Length;
}
```

Enumeracja - podsumowanie

Należy zapamiętać, że:

- `yield return` **nie wykonuje całej metody od razu**.
- Za każdym `MoveNext()` w `foreach`, **wykonywana jest kolejna część metody**.
- Gdy metoda dojdzie do `yield return`, **zwraca wartość i zatrzymuje się**.
- Przy następnym `MoveNext()` **wraca dokładnie w to miejsce, gdzie skończyła**.

Delegaty

Delegata to typ, który reprezentuje odwołanie do metody (*lub metod*) z określoną listą parametrów i typem zwracanym. Można ją traktować jako bezpieczny typowo wskaźnik do funkcji.

Delegata **nie wykonuje pracy** (*metody*), ale wie, **kto ma ją wykonać** (*która metoda*).

„Delegata” (mianownik) – forma żeńska, dopuszczalna jest też forma męska – „delegat”.

Delegaty

```
class Program
{
    public delegate int MyMath(int x, int y);

    static int Add(int a, int b) => a + b;
    static int Multiply(int a, int b) => a * b;

    static void Main()
    {
        MyMath op; //Deklaracja delegaty
        op = Add; //Przypisanie metody
        Console.WriteLine(op(3, 4)); // <-- 7

        op = Multiply;      //Zmiana metody
        Console.WriteLine(op(3, 4));//<-- 12
    }
}
```

Co tutaj się dzieje?

- `MyMath` to **typ delegaty**, mówiący *„metody muszą brać 2 inty i zwracać int”*.
- `op = Add` – delegata wskazuje na metodę `Add()`.
- `op(3,4)` – wywołanie metody przez delegatę (*tak jakbyś napisał `Add(3,4)`*).
- Możesz łatwo podmieniać metody delegaty w trakcie działania programu.

Po co istnieją delegaty?

- Dzięki delegatom możesz przekazywać funkcję jako parametr,
- możesz reagować na zdarzenia (`event`),
- możesz tworzyć pluginy i callbacki,
- i… możesz używać **funkcji lambda** i **LINQ**.

Delegaty – callback

Najważniejsze zastosowanie delegat to callbacki – przekazywanie metody jako argumentu do innej metody.

Co tu się dzieje?

1. `ProcessData` **nie wie**, jak dokładnie ma poinformować o wyniku.
2. Zamiast tego **przyjmuje delegatę** – czyli metodę do wywołania.
3. `ShowMessage` jest przekazana jako parametr.
4. `ProcessData` wykonuje swoje zadanie i **uruchamia callback** `notify("tekst")`.

Dlaczego to jest tak użyteczne?

- Możesz podmienić zachowanie (*metodę*) w zależności od potrzeb.
- Możesz pisać bardziej ogólne funkcje.

```
public delegate void Notifier(string message);

static void ProcessData(int number, Notifier notify)
{
    int result = number * 2;
    notify($"Wynik przetwarzania: {result}");
}

static void ShowMessage(string text)
{
    Console.WriteLine("[LOG] " + text);
}

static void Main()
{
    ProcessData(10, ShowMessage);

    //Output:
    //[LOG] Wynik przetwarzania: 20
}
```

Delegaty predefiniowane – **`Action`**

W C# bardzo rzadko piszemy własne delegaty (*`public delegate`*). Dlaczego? Ponieważ .NET daje nam **trzy gotowe delegaty**, które pokrywają praktycznie wszystkie zastosowania. To właśnie na nich opiera się LINQ, eventy, lambdy i nowoczesny styl programowania.

Delegaty `Action`:

- Służą do obsługi **metod typu `void`**.
- Mogą nie przyjmować żadnych parametrów – `Action`.
- Mogą przyjmować jeden parametr – `Action<T1>`.
- Mogą przyjmować dwa parametry – `Action<T1, T2>`.
- Maksymalnie mogą przyjąć 16 parametrów – `Action<T1, …, T16>`.

```
Action<string> info = msg =>
{
    Console.WriteLine("INFO: " + msg);
};

info("Start programu");

//Output:
//INFO: Start programu
```

Delegaty predefiniowane – **Func**

Func to delegata, która **zwraca wartość** – zwracana wartość, jest zawsze **ostatnim parametrem**:

- **`Func<TResult>`** - nic nie przyjmuje, zwraca `TResult`.
- **`Func<T, TResult>`** - przyjmuje jeden parametr `T`, zwraca `TResult`.
- **`Func<T1, T2, TResult>`** - przyjmuje dwa parametry (`T1` i `T2`), zwraca `TResult`.
- **`Func`** przyjmuje maksymalnie 16 parametrów – od `T1` do `T16`.

```
Func<int, int, int> add = (a, b) => a + b;

Console.WriteLine(add(3, 4));

//Output:
//7
```

Delegaty predefiniowane – **`Predicate<T>`**

Delegata **`Predicate<T>`** to specjalna wersja delegaty **`Func<T, bool>`**, służąca do **testowania warunku**.

- **`Predicate<T>` zawsze przyjmuje tylko jeden parametr** – T.
- **`Predicate<T>` zawsze zwraca tylko wartość logiczną.**

```
Predicate<int> isEven = n => n % 2 == 0;

Console.WriteLine(isEven(10)); // True
Console.WriteLine(isEven(11)); // False
```

Delegaty predefiniowane

Delegaty predefiniowane, **mogą również wskazywać na istniejącą metodę**, jednakże metoda ta musi:

- Posiadać **ten sam typ zwracany** co delegata.
- Przyjmować **dokładnie takie same parametry** jak delegata.

Jeżeli wszystko się zgadza, kompilator pozwoli nam przypisać daną metodę do delegaty.

```
void Show(string msg)
{
    Console.WriteLine("MSG: " + msg);
}

Action<string> printer = Show;

printer("Hello!");
```

```
int Multiply(int a, int b)
{
    return a * b;
}

Func<int, int, int> mul = Multiply;

Console.WriteLine(mul(3, 4));
```

```
bool IsAdult(int age)
{
    return age >= 18;
}

Predicate<int> check = IsAdult;

Console.WriteLine(check(20)); // True
Console.WriteLine(check(15)); // False
```

```
public class BettingPrice
{
    public double Price { get; set; }

    public string Source { get; set; }
}

public List<BettingPrice> FilterPrices(List<BettingPrice> prices, Predicate<BettingPrice> filter)
{
    var filtered = new List<Price>(prices.Count);
    foreach (var p in prices)
    {
        if (filter(p))
        {
            filtered.Add(p);
        }
    }

    return filtered;
}

private static bool IsPinnacle(BettingPrice p)
{
    return p.Source == "Pinnacle";
}

//Mamy listę jakiś priców...
var prices = new List<BettingPrice>()
{
  new() { Price = 1.97, Source = "Pinnacle" },
  new() { Price = 2.01, Source = "Bet365" },
  new() { Price = 1.98, Source = "William Hill" }
};

//Chcemy pricy które są większe od 2:
var greaterThan2 = FilterPrices(prices, x => x.Price > 2); // <-- Wyrażenie lambda

//Chcemy pricy tylko od Pinnacle
var onlyPinny = FilterPrices(prices, IsPinnacle); // <-- Przekazujemy metodę (bez nawiasów)
```

Bardziej przyziemny przykład

Funkcje anonimowe

Funkcja anonimowa to metoda bez nazwy, którą można przekazać tam, gdzie oczekiwana jest delegata.

Po co istnieją funkcje anonimowe?

- **Nie musimy tworzyć metod pomocniczych**, aby przekazać je jako delegatę.
- **Nie musimy nadawać jej nazwy**.

```
bool IsEven(int x) { return x % 2 == 0; }
Predicate<int> p = IsEven;
```

Jako delegatę wysyłamy istniejącą metodę `IsEven()`

```
Predicate<int> p = delegate(int x) { return x % 2 == 0; };
```

Nie potrzebujemy istniejącej metody, wystarczy nam funkcja anonimowa

Funkcje anonimowe

Składnia funkcji anonimowych jest bardzo prosta – wystarczy **słowo kluczowe delegate oraz przyjmowane parametry**. **Typ zwracany jest wnioskowany** przez kompilator, na podstawie ciała metody.

Funkcja anonimowa zawsze jest zastępowalna wyrażeniem lambda!

```
delegate (parametry)
{
    // ciało funkcji
}
```

Wyrażenia lambda

Wyrażenia lambda pozwalają nam definiować atomowe funkcje (*często zapisywane w jednej linii*) bez nazwy. Zapis jest znacznie krótszy i przyjemniejszy niż w przypadku funkcji anonimowych.

Jak wygląda wyrażenie lambda?

- x – parametr (*typ wnioskowany*)
- => – operator wyrażeń lambda
- x * 2 – ciało funkcji

To jest to samo

```
x => x * 2 // <-- Pełne wyrażenie lambda
```

=

```
//Odpowiednik w postaci funkcji anonimowej
delegate(int x) { return x * 2; }
```

Wyrażenia labda

Wyrażenia lambda bardzo elegancko **działają z delegatami predefiniowanymi**.

```
Action<string> hello = name => Console.WriteLine("Hi, " + name);
hello("Kuba");
//Output: Hi, Kuba

Func<int, int, int> multiply = (a, b) => a * b;
Console.WriteLine(multiply(3, 4));
//Output: 12

Predicate<int> isAdult = age => age >= 18;
Console.WriteLine(isAdult(20));
//Output: True
```

Wyrażenia lambda

Wyrażenie lambda może być rozpięte na kilka wierszy – wtedy, w takim wyrażeniu obecny jest **blok { }**.

```
x =>
{
    Console.WriteLine("Przetwarzanie...");
    return x * x;
}
```

Wyrażenie lambda nie zawsze musi przyjmować parametry:

```
() => Console.WriteLine("Hello!")
```

Zdarzenia

Zdarzenia to mechanizm, który pozwala klasie **powiadamiać** inne elementy programu o tym, że coś się stało. To fundament UI, gier, komunikacji między obiektami i reaktywnego kodu.

Czym jest zdarzenie (event)?

- Zdarzenie to sygnał „*coś się wydarzyło*”, który inne części programu mogą **zasubskrybować**, a później odebrać.
- Jedna klasa wysyła eventy, a inne klasy (*subskrybenci*) reagują.

Zdarzenia

Zdarzenia są oparte na delegatach – zwykle używa się delegat predefiniowanych. Na poniższym przykładzie zdarzenie to delegata `Action<string>`. Aby wywołać zdarzenie, **należy użyć metody `Invoke()`**.

Zdarzenie można wywołać tylko z wnętrza klasy, w której zostało zdefiniowane!

```
public class Chat
{
    //Zdarzenie
    public event Action<string> MessageSent;

    public void Send(string msg)
    {
        Console.WriteLine("Wysyłanie: " + msg);
        MessageSent?.Invoke(msg); // <-- Wywoływanie zdarzenia
    }
}
```

Jeżeli nikt nie zasubskrybował danego zdarzenia, będzie ono miało wartość null. Dlatego przed wywołaniem zdarzenia, bardzo często dodaje się operator „?”.

Wywoływanie zdarzenia

```
static void Main()
{
    Chat chat = new Chat();

    chat.MessageSent += msg =>
        Console.WriteLine("Odebrano wiadomość: " + msg);

    chat.Send("Hello!");
}

//Output:
//Wysyłanie: Hello!
//Odebrano wiadomość: Hello!
```

Subskrybowanie zdarzenia

Zdarzenia

Podsumowanie:

- Zdarzenie to specjalna delegata, która **może być wywołana wyłącznie z wnętrza danej klasy**.
- **Z zewnątrz można jedynie zasubskrybować** zdarzenie (**+=**) bądź **anulować subskrypcję** (**-=**).
- Każde zdarzenie może mieć **wielu subskrybentów**.
- Każde zdarzenie **może nie mieć w ogóle subskrybentów** – pamiętaj o operatorze (**?**).
- Zdarzenie wywołuje **wszystkie podpięte pod siebie metody**.

W przypadku rozbudowanych aplikacji, zawsze należy pamiętać o usuwaniu subskrypcji do zdarzeń, kiedy nie jest ona dłużej potrzebna. Brak kontroli nad subskrypcjami często prowadzi do poważnych wycieków pamięci.

Zdarzenia – `EventHandler`

`EventHandler` to **standardowy delegat do obsługi zdarzeń**, używany w większości frameworków .NET (*WinForms*, *WPF*, *MAUI*, *kontrolek*, *bibliotek*, *itd.*). To najlepszy i najbardziej „oficjalny” sposób definiowania eventów.

Dotychczas mieliśmy eventy oparte na `Action<string>` albo własnych delegatach. Ale w dużych projektach chcemy **jednolitą strukturę** zdarzenia:

- informacje o **nadawcy**,
- dodatkowe **dane o zdarzeniu**.

Dlatego istnieje gotowy delegat.

```
public delegate void EventHandler(object? sender, EventArgs e);
```

Zdarzenia – `EventHandler`

Podstawowy `EventHandler`, który posiada **referencję do obiektu, z którego wysyłany jest dany event** oraz nie przyjmuje żadnych dodatkowych danych:

- **`this`** – obiekt wysyłający event (*sender*),
- **`EventArgs.Empty`** – brak dodatkowych danych.

```
//Zdarzenie wewnątrz jakieś klasy
public event EventHandler SomethingHappened;
```

```
//Wywołanie eventu (z wnętrza danej klasy)
SomethingHappened?.Invoke(this, EventArgs.Empty);
```

Zdarzenia – `EventHandler`

Jeżeli zdarzenie ma przekazywać jakieś informacje, **musimy stworzyć własną klasę dziedziczącą po `EventArgs`.**

1. Tworzymy klasę danych zdarzenia

```
//Tworzymy własne zdarzenie
public class MessageEventArgs : EventArgs
{
    public string Message { get; }

    public MessageEventArgs(string msg)
    {
        Message = msg;
    }
}
```

2. Definiujemy zdarzenie w klasie

```
public event EventHandler<MessageEventArgs> MessageSent;
```

3. Wywołujemy zdarzenie (z wnętrza danej klasy) z parametrami

```
MessageSent?.Invoke(this, new MessageEventArgs("Hello world!"));
```

4. Subskrypcja zdarzenia (w innej klasie)

```
chat.MessageSent += (sender, e) =>
{
    Console.WriteLine("Wiadomość: " + e.Message);
};

//sender <-- obiekt który wywołał event
//e.Message <-- dane zdarzenia
```

Zdarzenia – `EventHandler`

Dlaczego warto używać `EventHandler`ów?

- **Jest to standard w .NET** – Windows Forms, Blazor, MAUI.
- **Spójność** – `sender` + `EventArgs`.
- **Kod łatwiejszy do utrzymania w przypadku dużych projektów**.

Zdarzenia a wycieki pamięci

Zdarzenia choć są bardzo użyteczne, charakteryzują się jedną (*dość subtelną*) pułapką – **obiekt wysyłający zdarzenie trzyma referencję do subskrybenta**. Oznacza to, że jeśli nie usuniesz subskrypcji danego zdarzenia, to obiekt, który powinien zostać usunięty z pamięci… **zostaje już na zawsze przy życiu**.

Dzieje się tak pomimo obecności GC w .NET.

Zdarzenia a wycieki pamięci

Wyobraźmy sobie dwa obiekty:

- **`Publisher`** (*nadawca*)
- **`Subscriber`** (*odbiorca*)

```
publisher.SomeEvent += subscriber.OnEvent;
```

Kiedy zasubskrybujesz zdarzenie, **`publisher` będzie trzymał referencję do `subscriber`**, a dopóki ktoś (*lub coś*) trzyma referencję do obiektu, **Garbage Collector nie może go usunąć z pamięci**.

```
//1. Mamy listę obiektów
var items = new List<Item>() { ... };

//2. Każdy obiekt subskrybuje zdarzenie
foreach (var item in items)
{
    publisher.UpdateEvent += item.HandleUpdate;
}

//3. Usuwamy pierwszy element z listy
items.RemoveAt(0);

//❌ I mamy wyciek pamięci... 😳😱
```

Zdarzenia a wycieki pamięci

Jak można zapobiec wyciekom pamięci?

- **Zawsze usuwaj subskrypcje eventów** – (`-=`).
- **Stosuj `IDisposable` tam, gdzie są subskrypcje**.
- **Uważaj na zdarzenia globalne i statyczne** – tam często tworzą się wycieki.

```
pub.Updated -= sub.OnUpdate;
```

```
public void Dispose()
{
    publisher.Updated -= OnUpdate;
}
```