Typy danych i podstawowe instrukcje C#

Bartosz Miąskowski

Typ danych

Definicja:

Typ danych to **opis zbioru dozwolonych wartości** (np. liczby całkowite, teksty, wartości logiczne) **oraz operacji**, które można na nich wykonać (np. dodawanie, konkatenacja, porównania), **wraz z zasadami przechowywania i konwersji**.

Innymi słowy: **typ odpowiada na pytania „jakie wartości?”, „co mogę z nimi robić?” i „jak są reprezentowane i sprawdzane?”**.

Rola typów danych

- **Bezpieczeństwo i błędy na etapie kompilacji** – kompilator wykrywa niedozwolone operacje (np. string + 1 bez konwersji).
- **Wydajność i pamięć** – typ determinuje rozmiar, sposób kopiowania i przechowywania.
- **Czytelność i narzędzia** – lepsze podpowiedzi, przeciążenia metod, dopasowanie operatorów.
- **Konwersje** – określa, kiedy rzutowanie jest dozwolone (jawne vs. niejawne).

Jak to wygląda w C#

- C# jest **statycznie i silnie typowany**: każdy obiekt ma znany w czasie kompilacji typ.
- Każdy typ w .NET jest obiektem (np. int to System.Int32), więc nawet liczba ma metody:

```
int n = 42;
string s = n.ToString(); // działa, bo int = System.Int32 z metodami
```

Rodzaje typów danych

Dwie fundamentalne kategorie (*omówimy szerzej później*):

- **Typy wartościowe „*Value types*”** (np. int, bool, struct) – przechowują wartość bezpośrednio.
- **Typy referencyjne „*Reference types*”** (np. string, klasy, tablice) – zmienna przechowuje **referencję** do obiektu.

Automatyczne wnioskowanie typu

Zamiast jawnego określania typu, można zastosować słowo kluczowe `var`. Jest to typ wnioskowany automatycznie podczas kompilacji – nie można go później zmienić.

```
var x = 5;   // x : int
var t = "hi"; // t : string
```

Uwaga!
Słowo kluczowe `var` działa tylko dla zmiennych lokalnych z inicjalizatorem, gdy kompilator potrafi jednoznacznie wywnioskować typ.

Typy wartościowe

Typy całkowite

Typ	Liczba bitów	Wartość minimalna	Wartość maksymalna
sbyte	8	−128	127
byte	8	0	255
short	16	−32 768	32 767
ushort	16	0	65 535
int	32	−2 147 483 648	2 147 483 647
uint	32	0	4 294 967 295
long	64	−9 223 372 036 854 775 808	9 223 372 036 854 775 807
ulong	64	0	18 446 744 073 709 551 615
char	16	0 (‘\u0000’)	65 535 (‘\uFFFF’) — Unicode znak

Typy zmiennoprzecinkowe

Typ	Liczba bitów	Zakres przybliżony (wartości niezerowe)	Dokładność (cyfry znaczące)	Przykład literału	Opis
float	32	$\pm 1{,}5 \times 10^{-45} \rightarrow \pm 3{,}4 \times 10^{38}$	ok. 7 cyfr	`3.14f`	Liczby rzeczywiste pojedynczej precyzji (ang. *single precision*).
double	64	$\pm 5{,}0 \times 10^{-324} \rightarrow \pm 1{,}7 \times 10^{308}$	ok. 15–16 cyfr	`3.14` lub `3.14d`	Liczby rzeczywiste podwójnej precyzji (ang. *double precision*).

Zagrożenia i pułapki przy typach zmiennoprzecinkowych

Zaokrąglenia i utrata precyzji:
Komputer nie potrafi dokładnie zapisać większości liczb dziesiętnych — przechowuje je w postaci **binarnej aproksymacji**.

```
double a = 0.1;
double b = 0.2;
Console.WriteLine(a + b == 0.3); //False 😳
Console.WriteLine(a + b); //0.30000000000000004
```

Dlatego liczb **zmiennoprzecinkowych** nie używa się do pieniędzy, ani operacji wymagających **ścisłej dokładności**.

Zagrożenia i pułapki przy typach zmiennoprzecinkowych

Nigdy nie rzucają wyjątków

Operacje na `float` i `double` **nigdy nie rzucają wyjątków** — nawet w sytuacjach „*matematycznie błędnych*”.

Sytuacja	Wynik	Przykład
Zbyt duża wartość	`Infinity`	`double x = 1e308 * 10; // ∞`
Zbyt mała wartość (blisko zera)	`0.0`	`double y = 1e-324 / 10; // 0.0`
Działanie nieokreślone	`NaN` („Not a Number”)	`0.0 / 0.0`, `Math.Sqrt(-1)`
Działanie na `NaN`	`NaN`	`NaN + 5` → `NaN`

To oznacza, że **nie zobaczysz wyjątku**, ale program może zwracać bezsensowne wartości, jeśli tego nie sprawdzisz.

Zagrożenia i pułapki przy typach zmiennoprzecinkowych

Istnieją dwa zera

- W większości przypadków działają tak samo, ale pewne operacje (np. 1.0 / +0 → ∞, 1.0 / -0 → -∞) odróżniają je.
- To ma znaczenie np. w analizie numerycznej lub przy implementacji algorytmów naukowych.
- Porównując zero dodatnie i zero ujemne otrzymamy `true`.

https://www.pikademia.pl/zapis-liczb-zmiennoprzecinkowych-w-systemie-binarnym/

Zarówno `double` jak i `float` korzystają z standardu IEEE 754

Zagrożenia i pułapki przy typach zmiennoprzecinkowych

Niejednoznaczność obliczeń na różnych CPU

Niektóre procesory (np. x87 – bardzo stary standard) wykonują obliczenia **z większą precyzją niż wymaga typ**, np. używając 80-bitowego rejestru zamiast 64-bitowego.

C# **pozwala na to**, żeby nie obniżać wydajności, ale skutkiem ubocznym może być:

- delikatna różnica wyników między komputerami,
- sytuacja, gdzie wyrażenie `x * y / z` da **skończony wynik zamiast Infinity**, bo procesor użył chwilowo wyższej precyzji.

To rzadko problem w codziennym kodzie, ale warto wiedzieć, że **„float ≠ matematyczna liczba rzeczywista”**.

Przypadek Mario

https://www.youtube.com/watch?v=TiUwJSOCbYE

Przypadek Mario

W tym przypadku Mario pierwotnie wydane na N64, było emulowane na Wii.

Konsola	CPU	Architektura	FPU	Tryb zaokrąglania	Skutek w Mario
N64	NEC VR4300 (MIPS R4300i)	64-bit RISC	MIPS FPU	Round to Nearest (IEEE)	Ruchy platform stabilne
Wii	IBM Broadway (PowerPC 750CL)	32-bit PowerPC G3	PPC FPU	*Round Toward Zero* (błąd emulatora)	Platforma powoli „płynie”

Typ dziesiętny

- `decimal` to **128-bitowy typ liczbowy**, który przechowuje liczby **dziesiętne (base 10)** z **bardzo dużą dokładnością – aż do 28 miejsc po przecinku**.
- Zakres wartości: od około **–7,9 × 10^{-28}** do **7,9 × 10^{28}**

Jak działa `decimal`?

- W przeciwieństwie do `float` i `double`, które liczą w systemie binarnym (base 2), `decimal` liczy w systemie dziesiętnym (base 10) – takim jak kalkulator.
- Dzięki temu liczby takie jak 0.1, 0.01, 123.45 są dokładne (a nie przybliżone).
- W praktyce każda liczba to:

$$(-1)^{znak} \times c \times 10^{-e}$$

*gdzie **c** to liczba całkowita, a **e** to skala (ile miejsc po przecinku).*

Typ dziesiętny

Cechy decimal:

- Idealny do obliczeń księgowych i finansowych.
- Nie traci dokładności przy prostych dziesiętnych (*np. 0.1 + 0.2 = 0.3 dokładnie).*
- Wolniejszy niż double (ponieważ dokładniejszy).
- Ma mniejszy zakres – szybciej można przekroczyć limit liczb.

Porównanie float, double i decimal

Cecha	float / double	decimal
System liczbowy	binarny (2)	dziesiętny (10)
Dokładność	7 / 15 cyfr	28–29 cyfr
Zakres	bardzo szeroki	węższy
Dokładność 0.1 + 0.2	≈ 0.30000000000000004	dokładnie 0.3
Przeznaczenie	nauka, grafika, fizyka	finanse, waluty

Typ wartości logicznej

```
bool isAdult = age >= 18;
bool isEmpty = text.Length == 0;

if (isAdult)
    Console.WriteLine("Dorosły");
else
    Console.WriteLine("Niepełnoletni");
```

Typ logiczny (bool) służy do reprezentowania warunków (odpowiedzi typu *tak/nie)*, który może przyjąć dwie wartości: **true** oraz **false**.

Typ wartości logicznej

W języku **C#** bool jest **oddzielny od typów liczbowych**.

```
int a = 1;
if (a) { } // ❌ błąd kompilacji
```

```
if (a != 0) { } // ✅ poprawnie
```

W C i C++ można pisać if (a) – w C# **nie.**

W C# nie można również **rzutować** bool oraz typów liczbowych.

Typ wyliczeniowy

Typ wyliczeniowy (enum) pozwala zdefiniować **własny typ danych** z **nazwanymi stałymi wartościami**. Zamiast używać liczb – używamy **czytelnych nazw**.

```
enum DayOfWeek
{
    Monday, //Wartość: 0
    Tuesday, //Wartość: 1
    Wednesday, //Wartość: 2
    Thursday, //Wartość: 3
    Friday, //Wartość: 4
    Saturday, //Wartość: 5
    Sunday //Wartość: 6
}
```

Typ wyliczeniowy

- Każda wartość w enum ma **liczbę całkowitą** w tle (domyślnie int).
- Jeśli nie podamy wartości — zaczyna od **0** i rośnie o 1:
  - Monday = 0, Tuesday = 1, itd.

```
enum AccessLevel
{
    Guest = 1,
    User = 2,
    Admin = 10
}
```

Można jawnie ustawić własne wartości

Typ wyliczeniowy

```
enum Status : byte
{
    Ok = 0,
    Warning = 1,
    Error = 2
}
```

Enumy domyślnie przyjmują wartości **int**, jednakże można ustawić inny **typ całkowity**.

Typ wyliczeniowy

Przykładowe użycie:

```
enum TrafficLight
{
    Red,
    Yellow,
    Green
}

TrafficLight light = TrafficLight.Red;

if (light == TrafficLight.Red)
    Console.WriteLine("Stop!");

else if (light == TrafficLight.Green)
    Console.WriteLine("Go!");
```

Typ wyliczeniowy

Można konwertować między enum a typem bazowym.

```
int value = (int)TrafficLight.Yellow;  // 1
TrafficLight tl = (TrafficLight)2;     // Green
```

Typ wyliczeniowy

```
enum Level { Low = 1, Medium = 2, High = 3 }

Level lvl = (Level)5;   // ! działa, ale...
Console.WriteLine(lvl); // Wynik: "5"
```

W C# **możesz przekonwertować dowolną liczbę całkowitą na enum** – nawet jeśli nie jest ona zdefiniowana jako żadna z nazwanych wartości.

Co się wtedy dzieje?

- Wartość 5 **nie ma nazwy** w enum, ale jest **dopuszczalna** – bo enum to po prostu int z etykietami.
- C# nie sprawdza, czy liczba jest w zdefiniowanym zestawie.
- ToString() zwróci:
  - nazwę (np. "Medium") jeśli wartość jest zdefiniowana,
  - liczbę ("5"), jeśli nie jest.

Typ wyliczeniowy

Czasem chcemy, żeby enum mógł reprezentować **wiele stanów naraz**. Przykładowo, plik może mieć uprawnienia do **odczytu i zapisu jednocześnie**. Wtedy używamy atrybutu **[Flags]**.

```
[Flags]
enum FileAccess
{
    None    = 0,    // 0000
    Read    = 1,    // 0001
    Write   = 2,    // 0010
    Execute = 4,    // 0100
    All     = Read | Write | Execute  // 0111
}
```

Każda wartość to **bit** – potęga dwójki!

Jak to działa w praktyce?

```
FileAccess perms = FileAccess.Read | FileAccess.Write;

Console.WriteLine(perms); // Read, Write

//Wartość binarna: 0001 | 0010 = 0011
```

Czyli jeden enum zawiera dwa stany naraz.

Dzięki [Flags], ToString() wypisze **nazwy połączone przecinkiem**.

Typ wyliczeniowy

& (*AND*) sprawdza, czy dana flaga jest ustawiona.

| (*OR*) dodaje nową flagę.

^ (*XOR*) przełącza flagę.

~ (*NOT*) neguje wszystkie bity.

```
if (perms.HasFlag(FileAccess.Read))
    Console.WriteLine("Odczyt dozwolony.");

if ((perms & FileAccess.Write) != 0)
    Console.WriteLine("Zapis dozwolony.");
```

Typ krotki

Czym jest krotka?

Krotka to **mały „pakiet" kilku wartości** (różnych typów), które są przechowywane razem.
Każdy element ma swoją **kolejność** (i opcjonalnie nazwę).

Można o niej myśleć jak o **tymczasowej mini-strukturze**.

```
(int, string) person = (1, "Ala");
Console.WriteLine(person.Item1); // 1
Console.WriteLine(person.Item2); // Ala
```

Cechy krotek

- Krotka ma **stałą liczbę elementów** → np. 2, 3, 4… *(nazywamy to **arnością** – arity)*.
- Każdy element ma **swój typ**.
- Można nadać elementom **nazwy**:

```
(int id, string name) student = (1, "Jan");
Console.WriteLine(student.id);   // 1
Console.WriteLine(student.name); // Jan
```

- Jeśli nie podasz nazw, C# tworzy domyślne Item1, Item2, Item3 itd.

Typ krotki

Krotki są idealne, gdy funkcja ma **zwrócić kilka rzeczy naraz**:

```
(int min, int max) FindRange(int[] numbers)
{
    return (numbers.Min(), numbers.Max());
}

var result = FindRange(new[] { 3, 5, 9 });
Console.WriteLine(result.min); // 3
Console.WriteLine(result.max); // 9
```

Typ krotki

- Nie trzeba używać **new**:

```
var t = (10, "ten"); // zamiast new ValueTuple<int, string>(10, "ten"
```

- Krotki są typami **wartości** (*struct*).
- Możne je **rozpakowywać** (*deconstruction*):

```
var (id, name) = (5, "Ola");
```

- Nazwy elementów **nie są przechowywane w runtime** – więc np. po odbiorze krotki z innej biblioteki, może ich nie być.

Typ krotki

Ograniczenia:

- Maksymalnie 7 elementów bezpośrednio (potem ValueTuple się zagnieżdża).
- Nie można nadać nazw zaczynających się od Item niezgodnych z pozycją.
- Krotka **nie zastępuje klasy czy rekordu**, gdy dane mają mieć logikę lub trwałość.

Podsumowując:

Krotka to szybki sposób na zwrócenie lub przekazanie kilku wartości naraz. Nie zastępuje ona jednak poważnych typów – klasy, rekordy czy nawet struktury.

Typ nullable

Czym jest typ nullable?

Normalnie typy wartościowe (np. int, double, bool) nie mogą być null. Czasami jednak chcemy, żeby mogły oznaczać brak wartości – do tego służą właśnie typy nullable.

```
int? age = null;   // OK
int normalAge = null; // ❌ błąd kompilacji
```

Jak działają nullable?

Składnia	Oznacza
`int?`	`System.Nullable<int>`
`bool?`	`System.Nullable<bool>`
`DateTime?`	`System.Nullable<DateTime>`

T? *to skrót od* ***System.Nullable<T>***

Każdy taki obiekt ma **dwie właściwości**:

```
int? x = 10;
Console.WriteLine(x.HasValue); // True
Console.WriteLine(x.Value);    // 10
```

```
int? y = null;
Console.WriteLine(y.HasValue); // False
Console.WriteLine(y.Value);    // ❌ InvalidOperationException
```

Właściwości i operatory nullable

Właściwość / Operator	Opis	Przykład
`.HasValue`	`true`, jeśli wartość nie jest `null`	`if (age.HasValue)`
`.Value`	zwraca rzeczywistą wartość	`int a = age.Value;`
`==`, `!=`	działają normalnie	`if (age == null)`
`??` (operator null-coalescing)	wartość domyślna, jeśli `null`	`int realAge = age ?? 0;`
`?.` (null-conditional)	bezpieczne wywołanie	`age?.ToString()`

*Nauczenie się podstaw pracy z typami nullable, jest **bardzo ważne**.*

Przykład użycia nullable

```
int? score = null;

if (score == null)
    Console.WriteLine("Brak wyniku");

else
    Console.WriteLine($"Wynik: {score.Value}");

// Operator ?? – domyślna wartość:
int final = score ?? 0;
Console.WriteLine(final); // 0
```

Typy referencyjne

String (ciąg znaków)

string to **ciąg znaków Unicode**, czyli tekst: słowa, zdania, imiona, komunikaty itp.
Pod spodem to **klasa (System.String)**, dziedzicząca z object.

```
string name = "Ala ma kota";
Console.WriteLine(name);
```

String (ciąg znaków)

```
string s = "Ala";
s += " ma kota";  // tworzony jest nowy string w pamięci
```

- Reprezentuje **ciąg znaków Unicode** (UTF-16)
  - każdy znak zajmuje **2 bajty** (16 bitów).
- **Jest niezmienny** (*immutable*) – nie można go modyfikować po utworzeniu.

String (ciąg znaków)

Właściwość	Wartość / opis
Długość (`Length`)	liczba znaków (nie bajtów)
Rozmiar jednego znaku	2 bajty (UTF-16)
Pamięć potrzebna na tekst	`Length × 2 bajty` + kilka bajtów narzutu obiektowego
Maksymalna długość	ok. **2 147 483 647 znaków** (czyli ok. **2 GB** danych) — teoretyczny limit klasy `System.String` w .NET
W praktyce	ograniczona pamięcią RAM i limitem obiektu .NET (~2 GB)

String (ciąg znaków)

Przykłady literałów:

```
string normal = "To jest tekst";
string multiline = @"C:\Folder\plik.txt"; // @"..." – dosłowny ciąg (bez escape'ów)
string interpolated = $"Witaj, {name}!"; // interpolacja
string raw = """Tekst "surowy" w wielu liniach""" ; // C# 11+
```

String (ciąg znaków)

Najważniejsze właściwości i metody:

```
string text = "Hello World";

Console.WriteLine(text.Length);       // 11
Console.WriteLine(text.ToUpper());    // HELLO WORLD
Console.WriteLine(text.Contains("or")); // True
Console.WriteLine(text[0]);           // 'H'
Console.WriteLine(text.Substring(6)); // "World"
```

String (ciąg znaków)

Każdy string w .NET jest **internowany** – identyczne literały w kodzie wskazują na **ten sam obiekt w pamięci**, by oszczędzać RAM.

Jeśli potrzebujesz **zmiennego tekstu**, użyj StringBuilder:

```
var sb = new StringBuilder();
sb.Append("Ala");
sb.Append(" ma kota");
string s = sb.ToString();
```

String a StringBuilder

StringBuilder pozwala **doklejać tekst bez tworzenia nowych obiektów**. Idealny, gdy często coś dopisujesz, np. w pętli.

```
var sb = new StringBuilder();
sb.Append("Ala");
sb.Append(" ma kota");
sb.Append(" i psa");

string result = sb.ToString();
```

StringBuilder działa w jednej strukturze pamięci, **bez kopiowania** oraz jest **szybki przy wielu modyfikacjach**.

String a StringBuilder

Cecha	`string`	`StringBuilder`
Typ	Klasa (`System.String`)	Klasa (`System.Text.StringBuilder`)
Zmienność	❌ niezmienny	✅ zmienny
Wydajność przy + w pętli	❌ słaba	✅ bardzo dobra
Zużycie pamięci	rośnie z każdą modyfikacją	stałe (bufor w pamięci)
Konwersja na string	—	`.ToString()`
Zastosowanie	krótkie teksty, napisy stałe	duże teksty, raporty, logi

StringBuilder jest używany raczej kiedy przetwarzamy naprawdę duże teksty, często wystarczy alternatywa w postaci listy ze stringami lub charami.

Typ tablicy

Tablica to **kolekcja elementów tego samego typu**, zapisywana w pamięci **jeden po drugim** i dostępna **po indeksie (od 0)**.

```
int[] numbers = { 1, 2, 3, 4, 5 };
Console.WriteLine(numbers[0]); // 1
Console.WriteLine(numbers[4]); // 5
```

Tworzenie tablic

```
int[] a = new int[3];                       // [0, 0, 0]
string[] names = new string[2];             // [null, null]
double[] values = { 1.2, 3.4, 5.6};         // inicjalizacja z wartościami
string[] cars = ["vw", "fiat", "renault"];  // inicjalizacja z wartościami
var motorcycles = new[] { "Yamaha", "Kawasaki", "Honda" }; // inicjalizacja z wartościami
```

Uwaga!

- po utworzeniu tablica ma **stałą długość** (*nie można jej rozszerzyć*),
- indeksy zaczynają się od **0**, kończą na **Length - 1**.

Właściwości i metody tablic

Właściwość / metoda	Opis	Przykład
`Length`	liczba elementów	`arr.Length`
`arr[i]`	dostęp do elementu	`arr[2] = 99;`
`foreach`	iteracja po elementach	`foreach (var n in arr)` `Console.WriteLine(n);`
`Array.Sort(arr)`	sortowanie	`Array.Sort(numbers);`
`Array.Reverse(arr)`	odwrócenie	`Array.Reverse(numbers);`

*Jest jeszcze metoda **Array.Resize()**, ale raczej się z niej nie korzysta – nie zmienia ona faktycznego rozmiaru tablicy, a tworzy nową o określonym rozmiarze i kopiuje do niej dane z poprzedniej… Jeżeli nie znamy liczby elementów, lepiej użyć jednej z dostępnych kolekcji – jak lista.*

Typy tablic

Tablica **jednowymiarowa** (*najbardziej powszechna*)

```
int[] a = { 1, 2, 3 };
```

Tablica **wielowymiarowa** (*maksymalnie 32 wymiary*)

```
int[,] matrix = new int[2, 3]
{
    {1, 2, 3},
    {4, 5, 6}
};

Console.WriteLine(matrix[1, 2]); // 6
```

Tablica **tablic** (*jagged*)

```
int[][] jagged = new int[2][];
jagged[0] = new int[] { 1, 2 };
jagged[1] = new int[] { 3, 4, 5 };

Console.WriteLine(jagged[1][2]); // 5
```

Przekroczenie indeksu tablicy

```
int[] arr = new int[3];
arr[3] = 10; // ❌ IndexOutOfRangeException – ostatni indeks to 2
```

Zawsze sprawdzaj długość tablicy – `arr.Length`

Typy wartościowe a typy referencyjne

W C# wszystkie typy dzielą się na dwie wielkie grupy:

- **Typy wartościowe** (*value types*) – zmienna przechowuje dane bezpośrednio (np. liczby, bool, struct)
- **Typy referencyjne** (*reference types*) – zmienna przechowuje adres (*referencję*) do danych przechowywanych w pamięci (*na stercie*).

Typy wartościowe a typy referencyjne

Typy wartościowe:

- Przechowywane na **stosie** (*stack*).
- **Kopiowane przez wartość**.
- Każda zmienna ma swoją **kopię danych**.
- **Nie mogą być null** (*chyba że są nullable*).

Przykłady: int, double, bool, char, decimal, struct, enum.

```
int a = 5;
int b = a;   // kopia
b = 10;

Console.WriteLine(a); // 5
Console.WriteLine(b); // 10
```

***a** nie zmieniło się — każda zmienna ma swoje dane.*

Typy wartościowe a typy referencyjne

Typy referencyjne:

- Przechowywane na stercie (*heap*).
- Kopiowane przez referencję.
- Dwie zmienne mogą wskazywać na ten sam obiekt.
- Mogą być null.

Przykłady: string, class, object, array, delegate, class.

```
class Person { public string Name; }

Person p1 = new Person() { Name = "Ala" };
Person p2 = p1;  // kopiujemy referencję
p2.Name = "Ola";

Console.WriteLine(p1.Name); // Ola
```

*Obie zmienne wskazują na **ten sam obiekt**.*

Typy wartościowe a referencyjne

Typy wartościowe

```
void AddOne(int x)
{
    x = x + 1;
    Console.WriteLine($"Wewnątrz metody: {x}"); //Wynik: 6
}

int a = 5;
AddOne(a);
Console.WriteLine($"Po wywołaniu: {a}"); //Wynik: 5
```

Jeśli metoda przyjmuje **typ wartościowy** (np. int, bool, struct), otrzymuje **kopię wartości**, a nie oryginał.

Zmiany wewnątrz metody **nie wpływają na zmienną wywołującą**.

Typy referencyjne

```
class Person { public string Name; }

void ChangeName(Person p)
{
    p.Name = "Ola";  //Zmiana pola w tym samym obiekcie
}

Person p1 = new Person() { Name = "Ala" };
ChangeName(p1);
Console.WriteLine(p1.Name); //Wynik: Ola
```

Jeśli metoda przyjmuje **typ referencyjny** (np. **class, string[]**), otrzymuje **kopię referencji** – czyli **adres obiektu w pamięci**.

Obie zmienne wskazują na **ten sam obiekt**, więc modyfikacja jego zawartości **wpływa na oryginał**.

Typy wartościowe a typy referencyjne

Cecha	Typ wartościowy	Typ referencyjny
Przechowywanie	Stos (stack)	Sterta (heap)
Zawartość zmiennej	Wartość	Adres (referencja)
Kopiowanie	Kopiuje dane	Kopiuje adres
`null`	Nie (chyba że `?`)	Tak
Domyślna wartość	0 / false / '\0'	`null`
Przykłady	`int`, `bool`, `struct`, `enum`	`string`, `class`, `array`, `record`

Stos i sterta

Kiedy program działa, dane są przechowywane w **pamięci operacyjnej (RAM)** w dwóch głównych obszarach:

Obszar	Nazwa	Do czego służy
📚 **Stos (stack)**	szybki magazyn dla **danych tymczasowych**	np. zmienne lokalne, parametry metod
📦 **Sterta (heap)**	magazyn dla **obiektów dynamicznych**	np. obiekty klas, tablice, stringi

Stos

```
void Test()
{
    int x = 5;       // na stosie
    double y = 2.5; // na stosie
}
```

Po zakończeniu Test() cały stos funkcji zostaje „sprzątnięty” — błyskawicznie.

- Działa jak **stos talerzy** – ostatni włożony, pierwszy zdjęty (**LIFO**).
- Przechowuje **wartości lokalne** i **adresy powrotu z metod**.
- Wszystko znika automatycznie po wyjściu z metody.

Sterta

- Przechowuje obiekty **utworzone przez *new***, np. klasy, tablice, stringi.
- Dane są przechowywane, dopóki coś do nich **trzyma referencję**.
- Sprzątane automatycznie przez **Garbage Collector (GC)**.

```
class Person { public string Name; }

void Test()
{
    Person p = new Person(); //p na stosie, obiekt na stercie
    p.Name = "Ala";
}
```

*Po wyjściu zmetody **p** (referencja) znika ze stosu,a obiekt **Person** zostaje na stercie – dopóki nie zostanie usunięty przez GC.*

Garbage Collector

- **Garbage Collector** to automatyczny mechanizm w .NET, który **usuwa z pamięci obiekty, których program już nie używa.**
- Nie musisz ręcznie zwalniać pamięci (delete, free, itp.) — GC robi to za Ciebie.

Cecha	Opis
♻️ **Automatyczny**	Nie musisz zwalniać pamięci ręcznie
🚀 **Działa w tle**	Uruchamia się, gdy pamięć jest potrzebna
🧮 **Zlicza referencje**	Usuwa obiekty, do których nikt już nie sięga
🪣 **Działa generacyjnie**	Dzieli obiekty wg „wieku” (młode, starsze, długożyjące)
🔒 **Bezpieczny**	Nie usunie obiektu, jeśli nadal jest używany

Warto pamiętać:

- GC nie usuwa natychmiast po p = null — robi to, gdy uzna za potrzebne.
- Można go ręcznie wymusić, ale tego się nie robi w normalnych programach.
- Jeśli obiekt trzyma inne obiekty, to one też zostaną usunięte razem z nim.

Typ object

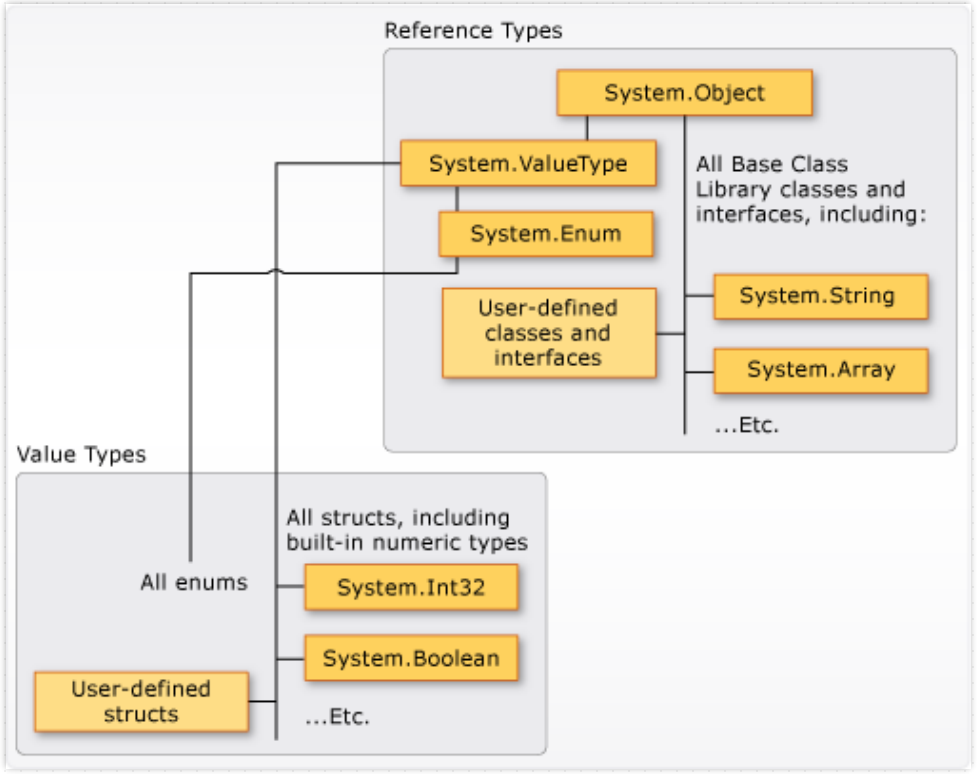
```
object x = 5;
object y = "Ala ma kota";
object z = new Person();
```

*To znaczy, że **każdy typ można traktować jak object**.*

`object` to **najbardziej podstawowy typ w C#**. Każdy inny typ — **wartościowy i referencyjny — dziedziczy z `object`**.

Typ object

Wszystko w C# dziedziczy po object
(System.Object)



https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/

Co object zapewnia wszystkim typom?

Każdy typ (*bo każdy dziedziczy po object*) automatycznie posiada następujące metody:

```
ToString()        // zwraca tekstową reprezentację obiektu
Equals(object)    // porównuje dwa obiekty
GetHashCode()     // zwraca unikalny kod liczbowy (np. do słowników)
GetType()         // zwraca typ obiektu w runtime
```

Na powyższym przykładzie znajdują się 4 podstawowe metody, które można wykonać na instancji każdego obiektu w C#.

Przykład:

```
int a = 5;
Console.WriteLine(a.ToString());   // "5"
Console.WriteLine(a.GetType());    // System.Int32
```

Boxing i unboxing

Ponieważ **object** może przechowywać wszystko, typy wartościowe muszą być **„opakowane” (boxed)**, gdy przypisujemy je do object, a potem **„odpakowane” (unboxed)** przy ponownym użyciu.

```
int a = 42;
object o = a;        // 🧱 boxing – kopia wartości trafia na stertę
int b = (int)o;      // 🔓 unboxing – pobranie wartości z powrotem
```

Boxing – tworzy nowy obiekt na stercie, więc jest wolniejszy.
Unboxing – rzutuje typ (*cast*) z powrotem do oryginału.

*Zbyt częsty boxing i unboxing **obniża wydajność** (szczególnie w pętlach).*

Rzutowanie

object nie zna typu, dopóki nie wykonasz rzutowania:

```
object obj = "Hello";
string s = (string)obj; // rzutowanie wymagane
```

Rzutowanie jest niebezpieczne, może skończyć się błędem:

```
object n = 123;
string s = (string)n; // ❌ InvalidCastException
```

Typ dynamiczny

`dynamic` to specjalny typ w C#, który – podobnie jak object – może przechowywać dowolną wartość. Różnica polega na tym, że **sprawdzanie typów odbywa się dopiero w czasie uruchomienia (runtime)** ,a nie podczas kompilacji.

Typ dynamiczny

Cecha	`object`	`dynamic`
Sprawdzanie typów	W czasie kompilacji	W czasie działania
Rzutowanie wymagane?	✅ Tak	❌ Nie
Bezpieczny typowo?	✅ Tak	⚠️ Nie (może rzucać wyjątki)
Typ w runtime	Zawsze znany (`GetType()`)	Określany dynamicznie

Porównanie `object` oraz `dynamic`

Typ dynamiczny

Jak to działa?

`dynamic` to tak naprawdę mechanizm oparty na `System.Dynamic` i DLS (**Dynamic Language Runtime**),czyli warstwie umożliwiającej **dynamiczne wiązanie (late binding)** – podobnie jak w Pythonie czy JavaScript.

```
object a = "Hello";
Console.WriteLine(a.Length); // ❌ błąd kompilacji
Console.WriteLine(((string)a).Length); // ✅ działa

dynamic b = "Hello";
Console.WriteLine(b.Length); // ✅ działa
Console.WriteLine(b.Nope()); // 💥 błąd dopiero w runtime!
```

*Kompilator **ufa Ci na słowo** – jeśli metoda nie istnieje, dowie się o tym dopiero w trakcie działania.*

Typ dynamiczny

Kiedy korzysta się z `dynamic`?

- **Współpraca z interfejsami COM** (*Component Object Model*) – wiele metod COM zwracało kiedyś `System.Object`, co wymagało ręcznego rzutowania i zgadywania typu. `dynamic` pozwala na traktowanie tych obiektów w bardziej bezpośredni, dynamiczny sposób, często eliminując konieczność rzutowania.
- **Interoperacyjność z językami dynamicznymi** – ułatwia współpracę z bibliotekami i obiektami z języków dynamicznych, takich jak IronPython czy IronRuby.
- **Przetwarzanie danych o nieznanej strukturze** – jest przydatny przy pracy z danymi, których struktura jest dynamiczna i nie jest reprezentowana przez dedykowane klasy, np. przy parsowaniu JSON lub XML (np. używając bibliotek takich jak Json.NET, które mogą deserializować do dynamic).
- **Uproszczenie refleksji** – w niektórych przypadkach dynamic może zastąpić skomplikowany kod używający refleksji do wywoływania metod lub dostępu do właściwości.
- **Tworzenie własnych obiektów dynamicznych** – używając klas z przestrzeni nazw `System.Dynamic` (np. `ExpandoObject` lub dziedzicząc po `DynamicObject`), można tworzyć obiekty, których składniki (właściwości, metody) można określać w czasie wykonania.

Wady korzystania z `dynamic`

- Brak podpowiedzi IntelliSense
- Błędy wychodzą dopiero w trakcie działania programu, kompilator ich nie wykryje.
- Wolniejszy – wszystko jest wiązane dynamicznie.
- Trudniejszy w utrzymaniu – słabsza analiza kodu.

```
dynamic person = new System.Dynamic.ExpandoObject();
person.Name = "Ala";
person.Age = 25;

Console.WriteLine($"{person.Name}, {person.Age}");
```

Przykład z `ExpandoObject`.

Słowa kluczowe

Czym są słowa kluczowe?

Definicja:

Słowa kluczowe (*keywords*) to **zarezerwowane wyrazy języka C#**, które mają **specjalne znaczenie** i **nie mogą być używane jako nazwy zmiennych, metod czy klas**.

Słowa kluczowe są bardzo ważne, **każde z nich posiada określoną funkcję**, przykładowo:

- `if` – sprawdza warunek.
- `class` – definiuje klasę.
- `return` – kończy działanie metody i zwraca wartość.

Słowa kluczowe umożliwiają komunikację z kompilatorem, nadają programowi strukturę i logikę oraz zapewniają czytelność i spójność kodu.

Najważniejsze słowa kluczowe w C#

Typy danych

```
bool, byte, sbyte, short, ushort, int, uint, long, ulong,
float, double, decimal, char, string, object
```

Najważniejsze słowa kluczowe w C#

Modyfikatory dostępu i klasy

```
public, private, protected, internal, class, struct, interface,
record, enum
```

Najważniejsze słowa kluczowe w C#

Zmienne i stałe

```
var, const, readonly, static, new
```

Najważniejsze słowa kluczowe w C#

Sterowanie przepływem

```
if, else, switch, case, default, for, foreach, while, do, break,
continue, return, goto
```

Najważniejsze słowa kluczowe w C#

Metody i parametry

```
void, ref, out, in, params, async, await, yield
```

Najważniejsze słowa kluczowe w C#

Dziedziczenie i obiekty

```
base, this, abstract, virtual, override, sealed
```

Najważniejsze słowa kluczowe w C#

Operatory logiczne i warunkowe

```
true, false, null
```

Najważniejsze słowa kluczowe w C#

Inne istotne słowa kluczowe

```
using, namespace, try, catch, finally, throw, checked, unchecked,
lock, is, as, or, and, from, when, where
```

Słowo kluczowe jako identyfikator

W języku C# można użyć **słowa kluczowego** jako **identyfikatora** (*np. nazwa zmiennej, metody, klasy*), jeżeli poprzedzimy je znakiem @.

Przydaje się to, kiedy dobra praktyka wymusza na nas nazwanie czegoś zarezerwowanym słowem – w kontekście dobrej czytelności kodu.

```
int @class = 10;
string @namespace = "MojeDane";
Console.WriteLine(@class);
```

Używaj @ **tylko gdy naprawdę musisz**, konieczność jej użycia to rzadkość – nadmierne wykorzystywanie tej notacji tylko pogorszy czytelność kodu.

Instrukcje sterujące

Instrukcje sterujące

Definicja:

Instrukcje sterujące (*control statements*) pozwalają **decydować o przebiegu działania programu** — czyli *kiedy*, *jak długo i ile razy* wykonuje się określony fragment kodu.

Instrukcje warunkowe

Instrukcje `if` i `else` pozwalają **wykonać określony fragment kodu tylko wtedy, gdy spełniony jest warunek logiczny**.

```
if (warunek)
{
    // Kod, gdy warunek jest prawdziwy (true)
}

else
{
    // Kod, gdy warunek jest fałszywy (false)
}
```

Składnia

```
int age = 18;

if (age >= 18)
    Console.WriteLine("Pełnoletni!");

else
    Console.WriteLine("Niepełnoletni!");
```

Przykład

Instrukcje warunkowe

Dobre praktyki

- Używaj **czytelnych warunków** (np. `if (isLoggedIn)` zamiast `if (x == true)).`
- Dodawaj nawiasy `{}` nawet przy jednej linii – zwiększa bezpieczeństwo kodu.

Typowe błędy

- Brak nawiasów klamrowych `{}` przy więcej niż jednej instrukcji.
- Użycie = zamiast == w warunku.
- Zbyt zagnieżdżone `if` – utrata czytelności.

Instrukcje warunkowe

Zbyt zagnieżdżony if – zła czytelność kodu.

```
if (loggedIn)
{
    if (isAdmin)
    {
        if (hasPermission)
        {
            Console.WriteLine("Dostęp przyznany!");
        }

        else
        {
            Console.WriteLine("Brak uprawnień.");
        }
    }

    else
    {
        Console.WriteLine("Nie jesteś administratorem.");
    }
}

else
{
    Console.WriteLine("Najpierw się zaloguj.");
}
```

Tutaj już trochę lepiej to wygląda.

```
if (!loggedIn)
{
    Console.WriteLine("Najpierw się zaloguj.");
}

else if (!isAdmin)
{
    Console.WriteLine("Nie jesteś administratorem.");
}

else if (!hasPermission)
{
    Console.WriteLine("Brak uprawnień.");
}

else
{
    Console.WriteLine("Dostęp przyznany!");
}
```

Po co używamy `else if`?
Kiedy chcemy sprawdzić **więcej niż jeden warunek**, możemy połączyć kilka instrukcji `if` za pomocą `else if`. Ostatni `else` wykona się tylko wtedy, gdy **żaden wcześniejszy warunek nie był prawdziwy**.

Wyrażenia warunkowe

Wyrażenie warunkowe (*conditional expression*) to **krótsza forma instrukcji `if...else`**, która **zwraca wartość** w zależności od wyniku warunku.

Alternatywne nazwy to **operator trójargumentowy** (*ternary operator*) lub **operator warunkowy**.

```
warunek ? wartość_gdy_true : wartość_gdy_false;
```

Składnia

```
int age = 20;
string result = age >= 18 ? "Pełnoletni" : "Niepełnoletni";

Console.WriteLine(result);
//Wynik: "Pełnoletni"
```

Przykład

Uwaga!
Wyrażenie warunkowe **nie jest równoważną alternatywą** dla `if else` – nie pozwala ono na wykonywanie wielu instrukcji, jego zadaniem jest **tylko zwrócenie danej wartości**. Przy zagnieżdżaniu wyrażeń warunkowych, **kod robi się bardzo nieczytelny**.

Instrukcja wyboru / wielokrotnego wyboru

Instrukcja switch

Instrukcja `switch` pozwala **zastąpić wiele instrukcji `if-else if`**, gdy sprawdzamy **wartość jednej zmiennej**. Dzięki temu kod jest **krótszy i bardziej czytelny**.

```
switch (wyrażenie)
{
    case wartość1:
        // kod
        break;

    case wartość2:
        // kod
        break;

    default:
        // kod, gdy żadna wartość nie pasuje
        break;
}
```

Składnia

Uwaga!
Każdy `case` (włącznie z `default`) **musi być zakończony** słowem kluczowym `break`. W przeciwnym razie **kod się nie skompiluje**.

```
int day = 3;

switch (day)
{
    case 1:
        Console.WriteLine("Poniedziałek");
        break;

    case 2:
        Console.WriteLine("Wtorek");
        break;

    case 3:
        Console.WriteLine("Środa");
        break;

    case 6:
    case 7:
        Console.WriteLine("Weekend");
        break;

    default:
        Console.WriteLine("Nieznany dzień");
        break;
}
```

Przykład

Switch i pattern matching

```
DisplayMeasurements(3, 4);  // Output: First measurement is 3, second measurement is 4.
DisplayMeasurements(5, 5);  // Output: Both measurements are valid and equal to 5.

void DisplayMeasurements(int a, int b)
{
    switch ((a, b))
    {
        case (> 0, > 0) when a == b:
            Console.WriteLine($"Both measurements are valid and equal to {a}.");
            break;

        case (> 0, > 0):
            Console.WriteLine($"First measurement is {a}, second measurement is {b}.");
            break;

        default:
            Console.WriteLine("One or both measurements are not valid.");
            break;
    }
}
```

Switch expression

`switch expression` (*wyrażenie*) to **nowsza, bardziej zwięzła wersja** instrukcji `switch`. Zamiast wielu `case` i `break`, zwraca **wartość**, którą możemy przypisać do zmiennej.

Znak podłogi _ to „*wildcard*” – w tym przypadku, jest to odpowiednik `default`.

```
var wynik = wyrażenie switch
{
    wartość1 => wynik1,
    wartość2 => wynik2,
    _        => wynikDomyślny //W przypadku braku, poleci exception.
};
```

Składnia

```
int day = 3;

string dayName = day switch
{
    1 => "Poniedziałek",
    2 => "Wtorek",
    3 => "Środa",
    4 => "Czwartek",
    5 => "Piątek",
    6 or 7 => "Weekend"
};

//Brak wildacrd spowoduje wyrzucenie wyjątku, jeżeli wartość nie znajduje się w przedziale.

Console.WriteLine(dayName);
```

Przykład

*Tutaj również działa **pattern matching** oraz słowo kluczowe **when**.*

Pętla for

Pętla for służy do **wielokrotnego wykonywania tego samego fragmentu kodu** — zazwyczaj wtedy, gdy z góry wiemy, **ile razy** ma się powtórzyć.

```
for (inicjalizacja; warunek; zmiana)
{
    // kod wykonywany w każdej iteracji
}
```

Składnia

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine($"Iteracja {i}");
}

//Wynik:
//Iteracja 0
//Iteracja 1
//Iteracja 2
//Iteracja 3
//Iteracja 4
```

Przykład

Uwaga!
Unikaj **magicznych liczb** – zamiast `i < 5` użyj np. `i < numbers.Length`

Pętla while

Pętla `while` wykonuje kod **tak długo, jak warunek jest prawdziwy (`true`)**. Nie wiemy z góry, ile razy się powtórzy – wszystko zależy od **warunku logicznego**

```
while (warunek)
{
    // kod wykonywany dopóki warunek == true
}
```

Składnia

```
int i = 0;

while (i < 3)
{
    Console.WriteLine($"Iteracja {i}");
    i++;
}

//Wynik:
//Iteracja 0
//Iteracja 1
//Iteracja 2
```

Przykład

Uwaga!
Upewnij się, że **warunek kiedyś nie będzie spełniony** (*aby wyjść z pętli*). Stosuj `while`, gdy **nie znasz z góry liczby powtórzeń**.

Pętla do...while

Pętla `do...while` **zawsze wykona się przynajmniej raz**, ponieważ **warunek sprawdzany jest dopiero po wykonaniu bloku kodu**.

```
do
{
    // kod do wykonania
}

while (warunek);
```

Składnia

```
int number;

do
{
    Console.Write("Podaj liczbę większą od 0: ");
    number = Int32.Parse(Console.ReadLine());
}

while (number <= 0);

Console.WriteLine($"Podałeś {number}!");
```

Przykład

Uwaga!
Stosuj do...`while`, gdy **blok musi się wykonać co najmniej raz.**

Pętla foreach

Pętla `foreach` służy do **przechodzenia po wszystkich elementach kolekcji** (*np. tablicy, liście, słowniku*).
Nie potrzebuje licznika — automatycznie pobiera każdy element po kolei.

```
foreach (var element in kolekcja)
{
    // kod wykonywany dla każdego elementu
}
```

Składnia

```
string[] fruits = { "jabłko", "banan", "gruszka" };

foreach (string fruit in fruits)
{
    Console.WriteLine($"Owoc: {fruit}");
}

//Wynik:
//Owoc: jabłko
//Owoc: banan
//Owoc: gruszka
```

Przykład

Uwaga!
W przypadku korzystania z `foreach`, **nie możemy modyfikować kolekcji** po której iterujemy!

Pętla foreach

Pętla foreach **nie pozwala modyfikować kolekcji**, po której aktualnie iteruje. Oznacza to, że nie można usuwać ani dodawać elementów w trakcie jej działania.

Dlaczego tak się dzieje?

Pętla `foreach` korzysta z **enumeratora** (*obiektu, który śledzi pozycję w kolekcji*). Gdy kolekcja się zmienia, enumerator **traci spójność** i nie wie, jak kontynuować iterację.

```
List<string> fruits = new() { "jabłko", "banan", "gruszka" };

foreach (var fruit in fruits)
{
    if (fruit == "banan")
    {
        fruits.Remove(fruit); //❌ InvalidOperationException
    }
}
```

Pętle nieskończone

```while
while (true)
{
    Console.WriteLine("Działa w nieskończoność...");
}
```

```for
for (;;)
{
    Console.WriteLine("Działa w nieskończoność...");
}
```

Nieskończona pętla to taka, która **nigdy nie przestaje się wykonywać**, ponieważ jej **warunek jest zawsze prawdziwy** (`true`) lub **nie istnieje**.

Kiedy korzysta się z nieskończonych pętli?

- Aplikacje serwerowe oraz działające w tle.
- Gry, programy monitorujące, pętle główne w mikrokontrolerach.
- Oczekiwanie na zdarzenie (*np. komunikat od użytkownika lub sygnał sieciowy*).

Uwaga!
W pętlach nieskończonych **zawsze powinno być jakieś opóźnienie** (np. `Thread.Sleep()` lub `Task.Delay()`), w innym przypadku **wykorzystanie CPU może wzrosnąć** do 100%. Również powinieneś zawsze **uwzględnić warunek wyjścia**.

Pętle – podsumowanie.

Rodzaj pętli	Kiedy używać	Cechy charakterystyczne	Przykład
`for`	Gdy **wiemy, ile razy** chcemy powtórzyć kod	Zawiera licznik (`i`), warunek i inkrementację w jednym miejscu	`for (int i = 0; i < 10; i++)`
`while`	Gdy **nie znamy z góry liczby powtórzeń**, ale znamy warunek	Sprawdza warunek **przed** każdą iteracją	`while (x > 0)`
`do...while`	Gdy kod ma się wykonać **co najmniej raz**, niezależnie od warunku	Warunek sprawdzany **po** wykonaniu kodu	`do { ... } while (x > 0);`
`foreach`	Gdy chcemy **przejść przez wszystkie elementy kolekcji**	Bezpieczna dla odczytu, nie pozwala modyfikować kolekcji	`foreach (var item in list)`
Nieskończona (`while(true)` / `for(;;)`)	Gdy program ma działać **bez końca**, np. serwer, gra, pętla główna	Musi mieć sposób zakończenia (`break`, `return`) lub opóźnienie	`while (true) { ... }`