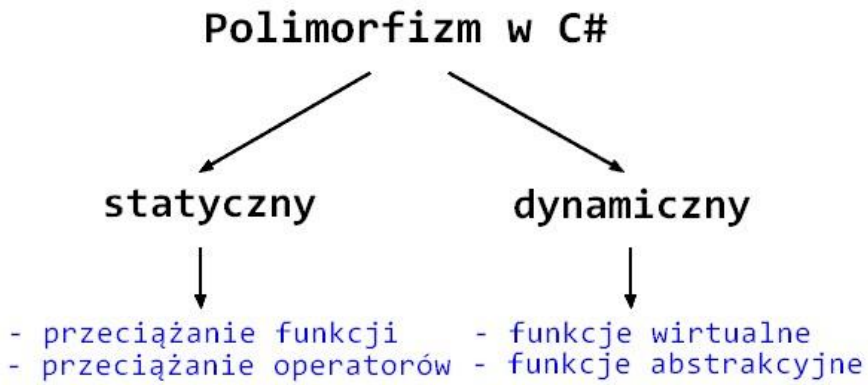
**POLIMORFIZM**

**Polimorfizm** w programowaniu to taka funkcja lub właściwość kodu, która pozwala na korzystanie z jednego "narzędzia" (np. metody, funkcji) do różnych zadań. Polimorfizm ułatwia rozbudowę i utrzymanie kodu, pozwalając na elastyczne rozszerzanie funkcjonalności bez zmiany istniejących klas. Dzięki temu kod jest bardziej czytelny i łatwiejszy w modyfikacji.




**Polimorfizm dynamiczny w C# to przesłanianie funkcji.**

**1) Metoda wirtualna (funkcja wirtualna)** to metoda, która może zostać nadpisana w klasach dziedziczących. Metody wirtualne umożliwiają polimorfizm. Metody wirtualnej nie musimy przesłaniać, ale jeżeli chcemy to musimy to zrobić za pomocą słowa kluczowego override.

**2) Metoda abstrakcyjna** zachowuje się dokładnie tak samo jak metoda wirtualna, jedyna różnica leży w braku definicji ciała funkcji. Tworząc metodę abstrakcyjną deklaruje się funkcję (deklaracja to typ zwracany, nazwa i argumenty) ale nie definiuje się ciała funkcji.

**Metodę abstrakcyjną można zadeklarować tylko w klasie abstrakcyjnej** (także poprzedzonej słowem abstract). Klasa abstrakcyjna jest klasą, której instancji nie da się stworzyć. Można po niej tylko dziedziczyć, rozszerzając ją o inne klasy. Klasy dziedziczące muszą implementować metody abstrakcyjne klasy abstrakcyjnej.

**METODY FORMATOWANIA ŁAŃCUCHÓW ZNAKÓW**

1. **Konkatenacja (+)** `Console.WriteLine (a + "world”);`
2. **String.Format** `("{0} ma {1} lat", "Jan", 30);` Pozwala na użycie miejsc zastepczych
3. **Interpolacja ciągów** `Console.WriteLine($”Mam na imie {imie} i mam {wiek} lat”);`
4. **String.Concat i String.Join**
   - Concat łączy elementy bez seperatora `string.Concat("A”, "B”, "C”); => ABC`
   - Join łączy z seperatorem np. `string.Join(",", "A”, "B”, "C”); => A,B,C`
5. **String.Builder** – wydajny przy częstych modyfikacjach łańcucha

```
var sb = new StringBuilder();
sb.Append("Hello”);
sb.Append(" ");
sb.Append("World");
string result = sb.ToString();
```

6. **Metody formatowania liczb i dat**
   Można je stosować w String.Format i interpolacji:

```
double liczba = 12345.6789;
Console.WriteLine($"{liczba:F2}"); // 12345,68 (2 miejsca po przecinku)
Console.WriteLine($"{liczba:N}"); // 12 345,68 (z separatorem tysięcy)

DateTime dt = DateTime.Now;
Console.WriteLine($"{dt:dddd, dd MMMM yyyy}"); // np. "niedziela, 14 września 2025"
```

**DZIEDZICZENIE**

**Dziedziczenie** to kluczowy mechanizm programowania obiektowego (OOP), pozwalający tworzyć nowe klasy na podstawie istniejących, odziedziczając ich właściwości i metody. Dzięki niemu kod jest bardziej zwięzły i mniej powtarzalny.

**Podstawy dziedziczenia:**

- **Klasa bazowa** (base class) przekazuje cechy klasie pochodnej (derived class).
- **Klasa pochodna** dziedziczy właściwości i metody klasy bazowej.
- W C# używa się operatora **:** do łączenia klas.

**Ważne zasady:**

1. **Dziedziczenie jednokrotne** — klasa może dziedziczyć tylko po jednej klasie.
2. **Modyfikator protected** pozwala na dostęp do członków tylko w klasie bazowej i pochodnych.
3. **Słowo kluczowe base** służy do wywoływania konstruktorów i metod klasy bazowej.
4. **Metody oznaczone virtual** można nadpisać w klasie pochodnej słowem override.

```
class Zwierze {
  public string Nazwa { get; set; }
  public void Oddychaj() { Console.WriteLine($"{Nazwa} oddycha."); }
}
class Pies : Zwierze {
  public void Szczekaj() { Console.WriteLine($"{Nazwa} szczeka."); }
}
```

**HERMETYZACJA**

**Hermetyzacja (enkapsulacja)** to ukrywanie w obiekcie elementów, do których użytkownik nie powinien mieć dostępu. Dzięki temu ograniczamy dostęp do zmiennych i metod wewnątrz klasy, chroniąc je przed nieautoryzowanym użyciem.

**Modyfikatory dostępu w C#:**

- **public** – dostęp dla wszystkich klas i metod,
- **private** – dostęp tylko w obrębie danej klasy,
- **protected** – dostęp w klasie i klasach dziedziczących.

**Właściwości (gettery i settery)**

**Gettery i settery** to specjalne metody pozwalające kontrolować dostęp do prywatnych pól klasy.

**Przykład 1:**

```
public class Person
{
    private string name;   // prywatne pole

    public string Name     // właściwość
    {
        get { return name; }
        set { name = value; }
    }
}

class Program
{
    static void Main()
    {
        Person person = new Person();
        person.Name = "Jan";   // przypisanie przez właściwość
        Console.WriteLine(person.Name);   // odczyt przez właściwość
    }
}
```

**Można też użyć automatycznych właściwości:**

```
public class Person
{
    public string Name { get; set; }
}

class Program
{
    static void Main()
    {
        Person person = new Person();
        person.Name = "Jan";
        Console.WriteLine(person.Name);
    }
}
```

**Przykład 2 – z walidacją wartości:**

```
public class Person
{
    private string name;

    public string Name
    {
        get { return name; }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                name = value;
            }
            else
            {
                throw new ArgumentException("Imię nie może być puste");
            }
```

```
        }
    }
}
```

*// użycie:*

```
Person person = new Person();
person.Name = "Jan";
Console.WriteLine(person.Name);
```

**W tym przykładzie:**

- **name** jest prywatne i niedostępne z zewnątrz,
- **właściwość Name** kontroluje dostęp i umożliwia dodanie logiki (np. sprawdzanie pustej wartości).

**OBSŁUGA WYJĄTKÓW**

**Wyjątki w C#**

- **Wyjątki** to błędy występujące podczas wykonania programu, np. błędny indeks tablicy lub nieprawidłowe dane.
- Standardowo program przerywa działanie i wyświetla komunikat o błędzie (wyjątek jest "rzucany" - throw).

**Obsługa wyjątków**

- Blok **`try`** zawiera kod, który może spowodować wyjątek.
- Blok **`catch`** przechwytuje wyjątek i pozwala na odpowiednią reakcję, np. wyświetlenie komunikatu.
- Blok **`finally`** jest opcjonalny i wykonuje się zawsze po `try` lub `catch`, np. do zwalniania zasobów.

**Przykład obsługi wyjątków:**

```
try
{
    int[] numbers = {1, 2, 3};
    Console.WriteLine(numbers[10]); //Wyrzuca IndexOutOfRangeException
}
catch (Exception e)
{
    Console.WriteLine("Wystąpił błąd: " + e.Message);
}
finally
{
    Console.WriteLine("Blok try-catch zakończony.");
}
```

**Tworzenie własnych wyjątków i throw**

- **`throw`** służy do ręcznego zgłaszania wyjątków.
- Można używać wbudowanych klas wyjątków (np. ArithmeticException, IndexOutOfRangeException), lub tworzyć własne.

**Przykład throw:**

```
void CheckAge(int age)
{
```

```
    if (age < 18)
        throw new ArithmeticException("Dostęp tylko dla osób
pełnoletnich.");
    else
        Console.WriteLine("Dostęp przyznany.");
}
CheckAge(15);  // Wyrzuci wyjątek
```

**OBSŁUGA PLIKÓW**

**Praca z plikami w C#**

- Do obsługi plików służy klasa File z przestrzeni nazw System.IO.
- Klasa File udostępnia metody do tworzenia, usuwania, kopiowania i czytania plików.

**Przydatne metody klasy `File`**

- **`AppendText()` — dopisuje tekst na końcu istniejącego pliku**
- **`Copy()` — kopiuje plik**
- **`Create()` — tworzy nowy plik lub nadpisuje istniejący**
- **`Delete()` — usuwa plik**
- **`Exists()` — sprawdza, czy plik istnieje**
- **`ReadAllText()` — odczytuje całą zawartość pliku jako tekst**
- **`Replace()` — zastępuje zawartość pliku inną zawartością**
- **`WriteAllText()` — tworzy plik i zapisuje podany tekst; jeśli plik istnieje, nadpisuje go**

**Przykład zapisu i odczytu pliku:**

```
using System.IO;
string writeText = "Hello World!";
File.WriteAllText("filename.txt", writeText);   // Tworzy plik i zapisuje tekst
string readText = File.ReadAllText("filename.txt");   // Odczytuje zawartość pliku
Console.WriteLine(readText);   // Wyświetla: Hello World!
```

**ZŁOŻONE TYPY DANYCH**

**Klasa List<T> w C#**

- **`List<T>`** to generyczna kolekcja dynamiczna z przestrzeni nazw `System.Collections.Generic`
- Pozwala na przechowywanie elementów o określonym typie **z dostępem przez indeks.**
- Listy mogą rosnąć i kurczyć się dynamicznie.
- **Pozwala na duplikaty i wartości null** (dla typów referencyjnych).
- **Nie jest** domyślnie **posortowana**.
- Automatycznie zwiększa pojemność (capacity) w razie potrzeby.

**Deklaracja i podstawowe operacje**

```
List<int> numbers = new List<int>();
numbers.Add(10);
numbers.Add(20);
numbers.Add(30);
```

**Ważne właściwości**

- **`Count`** – liczba elementów w liście,
- **`Capacity`** – pojemność wewnętrznej struktury danych,
- **`Item[int index]`** – dostęp do elementu przez indeks.

**Popularne metody**

- **`Add(T element)`** – dodaje element na koniec,
- **`AddRange(IEnumerable<T>)`** – dodaje kolekcję elementów,
- **`Remove(T element)`** – usuwa pierwsze wystąpienie elementu,
- **`RemoveAt(int index)`** – usuwa element pod danym indeksem,
- **`Clear()`** – usuwa wszystkie elementy,
- **`Contains(T element`**) – sprawdza, czy element jest na liście,
- **`Sort()`** – sortuje elementy,
- **`Find(Predicate<T>)`** – znajduje pierwszy element spełniający warunek,
- **`ForEach(Action<T>)`** – wykonuje akcję na każdym elemencie.

**Przykład wyświetlania i usuwania elementów**

```
List<int> list = new List<int>() { 17, 19, 21, 9, 75, 19, 73 };
```

```
Console.WriteLine("Przed usunięciem: " + string.Join(", ", list));
list.RemoveAt(3);   // usuwa element o indeksie 3
list.Remove(19);      // usuwa pierwsze wystąpienie 19
Console.WriteLine("Po usunięciu: " + string.Join(", ", list));
```

**HashSet<T> w C#**

- **`HashSet`** to kolekcja unikalnych elementów, bez duplikatów i bez określonego porządku.
- Wykorzystuje wewnętrznie tablicę mieszającą (hash table), dzięki czemu dodawanie, usuwanie i wyszukiwanie elementów jest szybkie (średnio O(1)).
- Implementuje interfejs ISet<T>, pozwalający na operacje na zbiorach jak suma, część wspólna, różnica.

**Tworzenie i dodawanie elementów**

```
HashSet<int> hs = new HashSet<int>();
hs.Add(10);
hs.Add(20);
hs.Add(30);
hs.Add(10); // drugi 10 nie zostanie dodany
```

**Dostęp do elementów**

- Elementy HashSet przeglądamy pętlą **`foreach`**, brak indeksowania.

**Usuwanie elementów**

- `Remove(T)` – usuwa konkretny element,
- `RemoveWhere(Predicate)` – usuwa elementy spełniające warunek,
- `Clear()` – usuwa wszystkie elementy.

**Operacje na zbiorach**

- `UnionWith(IEnumerable)` – suma zbiorów,
- `IntersectWith(IEnumerable)` – część wspólna,
- `ExceptWith(IEnumerable)` – różnica zbiorów.

**Przykład operacji:**

```
HashSet<int> set1 = new HashSet<int> {1, 2, 3, 5};
HashSet<int> set2 = new HashSet<int> {3, 4, 5};

set1.UnionWith(set2);       // {1, 2, 3, 4, 5}
```

```
set1.IntersectWith(new HashSet<int> {3, 5});  //{3, 5}
set1.ExceptWith(new HashSet<int> {5});        //{3}
```

**Dictionary w C#**

- **`Dictionary`** to generyczna kolekcja przechowująca pary klucz-wartość.
- Należy do przestrzeni nazw `System.Collections.Generic`.
- **Klucze muszą być unikalne** – próba dodania duplikatu powoduje wyjątek.
- Zapewnia szybkie wyszukiwanie, dodawanie i usuwanie elementów (średnio O(1)).

**Tworzenie i dodawanie elementów**

```
Dictionary<int, string> dict = new Dictionary<int, string>();
dict.Add(1, "C#");
dict.Add(2, "Java");
```

Można też użyć inicjalizatora:

```
var dict = new Dictionary<int, string> { {1, "C#"}, {2, "Java"} };
```

**Dostęp do elementów**

- Przez pętlę **`foreach`**:

```
foreach (var kvp in dict)
{
    Console.WriteLine($"Klucz: {kvp.Key}, Wartość: {kvp.Value}");
}
```

- Po kluczu: **`dict[1]`** zwraca wartość.
- Uwaga: odwołanie do nieistniejącego klucza rzuca `KeyNotFoundException`.

**Usuwanie elementów**

- **`Remove(klucz)`** – usuwa parę o danym kluczu,
- **`Clear()`** – usuwa wszystkie elementy.

**Sprawdzanie obecności**

- **`ContainsKey(klucz)`** – sprawdza, czy klucz istnieje,
- **`ContainsValue(wartość)`** – sprawdza, czy wartość istnieje.