

## Interfejsy w C#

**UWAGA:** W C# 8 w istotny sposób zmodyfikowano założenia dotyczące interfejsów (ich syntaksy, semantyki i pragmatyki). Podane poniżej stwierdzenia dotyczą natywnej koncepcji interfejsu - obowiązującej do C# 7 włącznie.

### Interfejs - co to jest

#### Interfejs

w programowaniu obiektowym jest definicją typu (abstrakcyjnego), posiadającego jedynie metody, a nie dane.

Zatem **nie można** utworzyć instancji interfejsu, można jednak deklarować zmienne typu interfejsu.

O *klasie konkretniej*, która implementuje wszystkie metody interfejsu mówimy, że **implementuje dany interfejs**.

Klasa może dziedziczyć tylko z jednej klasy, ale implementować wiele interfejsów. Koncepcję interfejsu jako wydzielonej jednostki programistycznej wprowadzono już w Java 1.0 (1996 r.) w celu wyeliminowania/ograniczenia problemów związanych z wielodziedziczeniem (występujących w C++).

#### Założenia dotyczące interfejsów:

- Interfejs określa odpowiedzialność klasy lub struktury (co ma robić), ale nie określa w jaki sposób ta odpowiedzialność ma zostać zrealizowana (jak ma to robić).
  - | Realizowane jest to poprzez sygnatury metod
- W interfejsie nie deklarujemy składników prywatnych.
  - | Użycie składników prywatnych jest wnikiem w szczególne implementacyjne
- Domyslnie wszystkie składniki interfejsu są publiczne i abstrakcyjne.
  - | Interfejs opisuje budowę klasy widzianej z zewnątrz i nie dostarcza żadnej implementacji
- Interfejs jest typem języka C#. Definiowany jest słowem kluczowym `interface`.
- Interfejs nie może zawierać pól, ale może zawierać deklaracje *properties*.
  - | Deklaracja pól jest wnikiem w szczególne implementacyjne.  
*Property* (`get`, `set`) jest - z formalnego punktu widzenia - metodą, widzaną tak, jak pole. Definiuje działanie operatora przypisania w określonym kontekście.
- Interfejs nie może zawierać deklaracji konstruktora.
  - | Byłyby to szczegół implementacyjny
- Interfejs może być traktowany jako kontrakt (pierwowzór klasy, na bazie której później tworzone będą obiekty). W interfejsie zatem nie można deklarować składników statycznych.
  - | Mówimy, że interfejs "pracuje na rzecz instancji".
- Dzięki interfejsom w C# można zrealizować koncepcję wielodziedziczenia.
  - | Klasa/struktura mogą implementować wiele interfejsów. Klasa może dziedziczyć wyłącznie z jednej klasy bazowej.

#### Składnia deklaracji interfejsu:

```
// deklaracja
interface IPrzyklad
{
    // deklaracja `event`-ów
    // deklaracja indekserów
    // deklaracja metod
    // deklaracja properties
}

// użycie - implementacja interfejsu
class MojPrzyklad : IPrzyklad
{
    // ... implementacje
}
```

#### Przykład konkretny:

Deklaracja interfejsu `IStos<T>` (jako typ abstrakcyjny), w formie kontraktu do zrealizowania :

```
public interface IStos<T>
{
    void Push(T value);      //wstawia element t typu T na stos
    T Pop();                //zdejmuje szczytowy element ze stosu
    T Peek { get; }          //zwraca szczytowy element stosu
    int Count { get; }       //zwraca liczbę elementów na stosie
    bool IsEmpty { get; }    //zwraca true, jeśli stos jest pusty, a false w przeciwnym przypadku
    void Clear();            //opróżnia stos
    T this[int index] { get; } //indekser, read-only
}
```

Jako typ abstrakcyjny, podany stos spełnia poniższe założenia (aksjomaty), które formalnie definiują jego działanie:

1. Bezpośrednio po utworzeniu, stos jest pusty. Po opróżnieniu (`clear()`), stos jest pusty.
2. Po wykonaniu `Push(x)` stos nie jest pusty (zawiera co najmniej jeden element).
3. Sekwencja metod `Push(x)` a następnie `Pop()` nie zmienia zawartości stosu.
4. Po wykonaniu `Push(x)`, metoda/property `Peek` zwróci wstawiony element `x`.
5. Wykonanie metod `Pop()` lub `Peek` na stosie pustym zgłasza wyjątek `StackEmptyException`.

Przykładowa implementacja stosu (niekompletna), wykorzystującego tablicę jako nośnik danych:

```
public class StosWTablicy<T> : IStos<T>
{
    private T[] tab;
    private int szczyt = -1;

    public StosWTablicy(int size = 10)
    {
        tab = new T[size];
        szczyt = -1;
    }

    public T Peek => throw new NotImplementedException();
    public T Pop() => throw new NotImplementedException();
    public void Push(T value) => throw new NotImplementedException();
    public int Count => szczyt + 1;
    public bool IsEmpty => szczyt == -1;
    public void Clear() => szczyt = -1;
    public T this[int index] => throw new NotImplementedException();
    public int Capacity => tab.Length;
    public void TrimExcess() => throw new NotImplementedException();
}
```

Przykładowa implementacja stosu (niekompletna), wykorzystującego struktury wiązane jako nośnik danych:

```
public class StosLinkedNodes<T> : IStos<T>
{
    private class Node
    {
        T data;
        Node next;
    }
    private Node head;
    private Node top;
    private int counter;

    public StosLinkedNodes() { head = top = null; counter = 0; }
    public int Count => counter;
    public void Clear() { top = head = null; counter = 0; }
    public bool IsEmpty => counter == 0;
    public T Peek => throw new NotImplementedException();
    public T Pop() => throw new NotImplementedException();
    public void Push(T value) => throw new NotImplementedException();
    public T this[int index] => throw new NotImplementedException();
}
```

Obie implementacje realizują *kontrakt* zapisany w interfejsie, aczkolwiek robią to zupełnie inaczej - inne są podstawy zapamiętywania elementów stosu. Ponadto `StosWTablicy` dostarcza dwie dodatkowe metody bezpośrednio związane z reprezentacją wewnętrzną (property typu `get Capacity` zwracające aktualną pojemność stosu (nie liczbę zapamiętywanych elementów) oraz metodę `TrimExcess()` optymalizującą rozmiar tablicy przechowującej elementy stosu). Metody te nie miały sensu w przypadku realizacji wiązanej.

## Klasy abstrakcyjne versus interfejsy

Interfejs może być postrzegany jako forma kontraktu dla implementatorów. Ale klasa abstrakcyjna również może tę rolę spełniać. Zobaczmy to na przykładzie stosu:

```
public abstract class AbstractStos<T>
{
    private int counter;

    public AbstractStos()
    {
        counter = 0;
    }

    public abstract void Push(T value); //wstawia element t typu T na stos
    public abstract T Pop(); //zdejmuję szczytowy element ze stosu
    public abstract T Peek { get; } //zwraca szczytowy element stosu
    public int Count => counter; //zwraca liczbę elementów na stosie
    public bool IsEmpty => Count == 0; //zwraca true, jeśli stos jest pusty, a false w przeciwnym przypadku
    public abstract void Clear(); //opróżnia stos
    public abstract T this[int index] { get; } //indeks, read-only
}
```

Klasa abstrakcyjna może zawierać częściową implementację. W takim razie być może konieczne są jakieś pola i składowe prywatne. Również należy przewidzieć konstruktor, choć nie może on być bezpośrednio wywołany operatorem `new`. Część metod opatrzona jest słowem `abstract` - one **muszą** być zaimplementowane w klasie konkretnej, dziedziczącej z klasy abstrakcyjnej. Inne metody, dla których w tym momencie żądaną implementację można będzie

uznaję z klasą abstrakcyjną. Te metody - dla których w tym momencie znany jest sposób realizacji - mogą być w klasie abstrakcyjnej zaimplementowane (np. `IsEmpty` na bazie `Count`).

Widzimy zatem podstawowe różnice między interfejsem a klasą abstrakcyjną:

| klasa abstrakcyjna  | interfejs  |
|---|--|
| Składa się z deklaracji metod (abstrakcyjnych) i/lub metod zaimplementowanych   | Składa się wyłącznie z deklaracji metod (właściwości, indekserów, event-ów)  |
| Może dziedziczyć tylko z jednej klasy (bazowej)   | Zapewnia realizację wielodziedziczenia   |
| Może zawierać konstruktor (jeśli nie zawiera, to tworzony jest automatycznie domyślny)  | Nie może zawierać konstruktorów  |
| Może zawierać statyczne składniki   | Nie może zawierać statycznych składników   |
| Składniki mogą być publiczne, prywatne, <code>protected</code> , ...  | Może zawierać wyłącznie składniki publiczne, ponieważ - z założenia - interfejs jest publiczny. Przy deklaracji składników nie używamy słowa <code>public</code>                   |
| Wydajność klas abstrakcyjnych jest większa niż interfejsów  | Wydajność interfejsów jest mniejsza niż klas abstrakcyjnych, ponieważ czasochłonne jest wyszukiwanie metod implementujących metody interfejsu w klasach konkretnych                |
| Jest wykorzystywana w charakterze klasy węzłowej w hierarchii klas, definiującej podstawowe funkcjonalności potomków  | Jest wykorzystywany do definiowania dodatkowej funkcjonalności klas implementujących   |
| Klasa konkretna może użyć tylko jednej klasy abstrakcyjnej (dziedziczenie)  | Klasa konkretna może użyć wielu interfejsów (implementacja)  |
| Jeśli wiele klas implementuje <b>w ten sam sposób te same</b> funkcjonalności, to celownym jest umieścić je w klasie bazowej, która stanie się węzłem w hierarchii klas i z której pozostałe <i>podobne</i> klasy będą otrzymać funkcjonalności dziedzicząc. Klasa ta będzie najprawdopodobniej klasą <i>abstrakcyjną</i> | Jeśli wiele implementacji jedynie współdzieli metody o tej samej sygnaturze, wtedy wskazane jest zapisanie tej sygnatury w interfejsie, zaś klasy implementować będą ten interfejs |
| Klasy abstrakcyjne zawierają mogą stałe, pola, metody, ...  | Interfejsy zawierają mogą jedynie metody (a zatem również <i>properties</i> , indeksery, eventy - które formalnie są metodami <i>pod przykryciem</i> )                             |
| Klasy abstrakcyjne mogą być całkowicie, częściowo lub wcale nie implementowane (wtedy klasą dziedziczącą jest klasa abstrakcyjna)   | Interfejsy muszą być w całości implementowane (lub częściowo przez klasę abstrakcyjną)   |

## Interfejs jest typem (referencyjnym)

Interfejsy definiują typy (referencyjne). Można zatem deklarować zmienne (typu "interfejsowego").

```
IStos<int> s1;
```

Ale nie można utworzyć obiektu typu interfejsowego (przecież interfejs opisuje totalną abstrakcję i nie może definiować chociażby konstruktora).

```
s1 = new IStos<int>(); //compilation error
```

```
C:\Przedmioty\CSHarp-2020-github\Types-Interfaces\o4uiwo917_code_chunk(14,6): error CS0144: Nie można utworzyć wystąpienia klasy lub interfejsu
```

Można za to przypisać zmiennej typu interfejsowego obiekt każdego typu implementującego ten interfejs.

```
IStos<int> s = new StosWTablicy<int>();
Console.WriteLine( s is IStos<int> );
Console.WriteLine( s is StosWTablicy<int> );
```

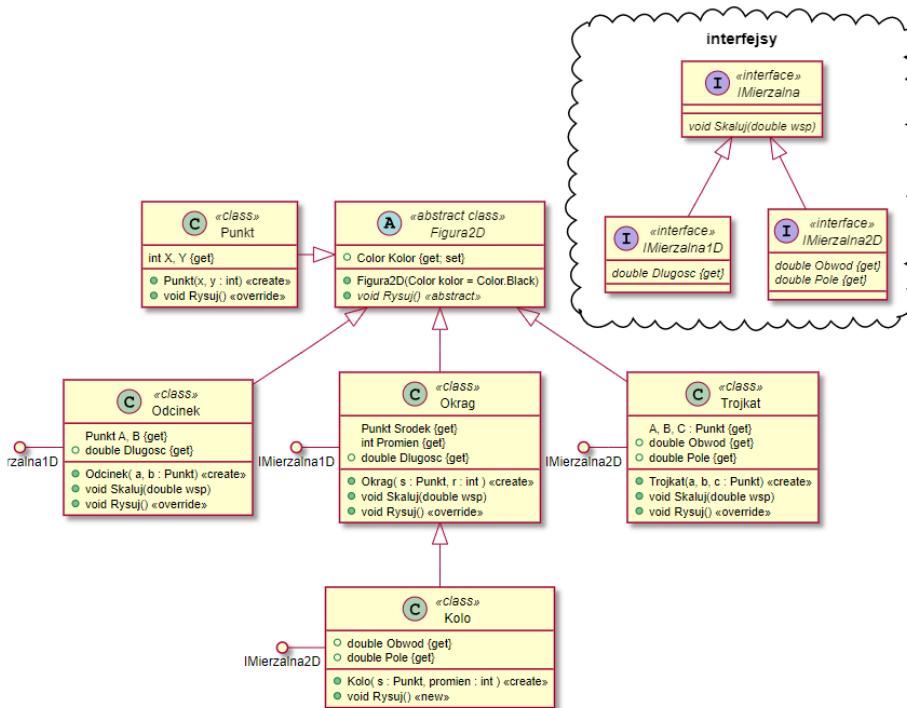
W powyższym przykładzie widać, że zmienna `s` jest równocześnie dwóch typów: interfejsowego `IStos<int>` oraz konkretnego `StosWTablicy<int>` (formalnie jest jeszcze typu `object`).

Możemy zatem bezpośrednio skorzystać z każdej funkcjonalności zdefiniowanej w interfejsie (na rzecz zmiennej `s`) lub pośrednio, wiedząc, że `s` jest konkretnego typu, przez rzutowanie:

```
IStos<int> s = new StosWTablicy<int>(10);
Console.WriteLine( s.IsEmpty );
//Console.WriteLine( s.Capacity ); // compilation error - w interfejsie nie ma takiej metody
Console.WriteLine( ((StosWTablicy<int>)s).Capacity );
```

**Przykład - Figury:**

Rozważmy inny przykład - hierarchię figur geometrycznych (np. w aspekcie tworzonej aplikacji graficznej) rysowanej na siatce z całkowitymi współrzędnymi (np. pikseli).



Zauważamy kilka ważnych właściwości:

- Interfejsy mogą budować własną hierarchię dziedziczenia. Opisują właściwości, które spełniać będą obiekty. W tym przypadku nadzcznąą właściwością jest *mierzalność*, a każda figura mierzalna jest skalowalna (wg swoich zasad - inaczej odcinek, inaczej okrąg).
  - Obiekty mierzalne w jednym wymiarze mają długość (np. odcinek, okrąg).
  - Obiekty mierzalne w 2 wymiarach mają obwód i pole powierzchni.
- Kluczową klasą jest `Figura2D` - klasa abstrakcyjna, definiująca podstawową funkcjonalność każdej figury (tu: fakt przypisania koloru oraz możliwość narysowania). Oczywiście obiektu typu `Figura2D` nie można narysować, dopóki nie będziemy wiedzieli, jakiego jest szczegółowego typu (czy jest odcinkiem, czy trójkątem, ...). Klasa abstrakcyjna jest węzłową dla całej hierarchii.
- Typ `Punkt` jest klasą konkretną, dziedziczy z klasy `Figura2D` funkcjonalności (można go narysować w określonym kolorze). Nie będzie on miał podtypów (od struktury nie można dziedziczyć). Nie ma on właściwości mierzalności.
- Typ `Odcinek` realizowany jest jako klasa konkretna, ma właściwość mierzalności w 1 wymiarze.
- Typ `Kolo` realizowany jest jako klasa konkretna, ma właściwość mierzalności w 2 wymiarach (obwód i pole). Ponieważ dziedziczy z klasy `Okrag` (koło jest okręgiem wraz z wypełnieniem), można mówić również o *długości koła*.
- Każde koło jest jednocześnie okręgiem (wynika z dziedziczenia). Poprawnym będzie zatem zapis:

```
Okrag o = new Kolo( new Punkt(0,0), 1 );
```

a następnie:

```
Console.WriteLine( o.Dlugosc ); // poprawnie
Console.WriteLine( o.Pole ); // compilation error
Console.WriteLine( (Kolo)o.Pole ); // poprawnie
```

- W klasie `Kolo` metoda `Rysuj()` nie przesłania (override) tej z klasy `Okrag`, ale definiuje nową (`new`).

Przeanalizujmy skutki. Gdyby implementacja wyglądała tak, jak na poniższym listingu:

```
public class Okrag : Figura2D, IMierzalna1D
{
    ...
    public override string ToString() => $"({Srodek.X},{Srodek.Y}), {Promien})";
    public override void Rysuj() => WriteLine($"Okrag({ToString()}), {this.Kolor}, dlugosc = {Dlugosc}");
}

public class Kolo : Okrag, IMierzalna2D
{
    ...
    public override void Rysuj() => WriteLine($"Kolo({ToString()}), {this.Kolor}, obwod = {Obwod}, pole = {Pole}");
}
```

czyli `Rysuj()` z klasy `Kolo` przesłaniałoby `Rysuj()` z klasy `Okrag`, to poniższy kod, dzięki polimorfizmowi, wywołowałby metodę `Rysuj()` zdefiniowaną w `Kolo`:

```
Okrag o = new Kolo( new Punkt(0,0), 1 );
o.Rysuj();
// output: Kolo(0,0, 1) Red obwod = 6.283185307179586 pole = 3.141592653589793
```

Jeśli w klasie `Kolo` i metodzie `Rysuj()` słowo kluczowe `override` zastąpimy słowem `new`, działanie będzie odmienne - wykona się metoda `Rysuj()` z klasy `Okrag`:

```
public class Okrag : Figura2D, IMierzalnaID
{
    // ...
    public override string ToString() => $"S{{Srodek.X},{Srodek.Y}}, {Promien})";
    public override void Rysuj() => WriteLine($"Okrag({ToString()}), {this.Kolor}, dlugosc = {Dlugosc})"
}

public class Kolo : Okrag, IMierzalna2D
{
    public new void Rysuj() => WriteLine($"Kolo({ToString()}, {this.Kolor}, obwod = {Obwod}, pole = {Pole})");
}

Okrag o = new Kolo( new Punkt(0,0), 1 );
o.Rysuj();
// output: Okrag((0,0), 1, Red, dlugosc = 6,283185307179586)
```

Pełna implementacja hierarchii klas i interfejsów zaprezentowana jest na poniższym listingu:

```
using System;
using Color = System.ConsoleColor;
using static System.Math;
using static System.Console;

namespace src_ex_figury
{
    #region hierarchia interfejsów
    public interface IMierzalna { void Skaluj(double wsp); }
    public interface IMierzalna1D { double Dlugosc { get; } }
    public interface IMierzalna2D
    {
        double Obwod { get; }
        double Pole { get; }
    }
    #endregion hierarchia interfejsów

    #region hierarchia klas
    public abstract class Figura2D
    {
        public Color Kolor { get; set; }
        public Figura2D(Color kolor = Color.Black) { Kolor = kolor; }
        public abstract void Rysuj();
    }

    public class Punkt : Figura2D
    {
        public int X { get; private set; }
        public int Y { get; private set; }
        public Punkt(int x, int y) { X = x; Y = y; }
        public override string ToString() => $"({X},{Y})";
        public override void Rysuj() => WriteLine($"Punkt({this}, {Kolor})");
    }

    public class Odcinek : Figura2D, IMierzalna1D
    {
        public Punkt A { get; private set; }
        public Punkt B { get; private set; }
        public Odcinek(Punkt a, Punkt b) : base(Color.Blue)
        {
            A = a; B = b;
            A.Kolor = B.Kolor = Color.Blue;
        }

        public void Skaluj(double wsp) // rozciąganie, A niezmienny, przesuwamy B
        {
            if (wsp < 0) throw new ArgumentOutOfRangeException("wsp. skalowania nie może być ujemny");
            B = new Punkt((int)(A.X + (B.X - A.X) * wsp), (int)(A.Y + (B.Y - A.Y) * wsp));
        }
        public double Dlugosc => Sqrt(Pow(A.X - B.X, 2) + Pow(A.Y - B.Y, 2));

        public override string ToString() => $"(A({A.X},{A.Y}), B({B.X},{B.Y}))";
        public override void Rysuj() => WriteLine($"Odcinek({ToString()}), {this.Kolor}, dlugosc = {Dlugosc})");
    }

    public class Okrag : Figura2D, IMierzalna1D
    {
        public Punkt Srodek { get; private set; }
        public int Promien { get; private set; }
        public Okrag(Punkt s, int r) : base(Color.Cyan) { Srodek = s; Promien = r; }
        public void Skaluj(double wsp)
        {
            if (wsp < 0) throw new ArgumentOutOfRangeException("wsp. skalowania nie może być ujemny");
            Promien = (int)(Promien * wsp);
        }
        public double Dlugosc => 2 * PI * Promien;

        public override string ToString() => $"S{{Srodek.X},{Srodek.Y}}, {Promien})";
        public override void Rysuj() => WriteLine($"Okrag({ToString()}), {this.Kolor}, dlugosc = {Dlugosc})");
    }

    public class Kolo : Okrag, IMierzalna2D
```

```

    {
        public Kolo(Punkt s, int r) : base(s, r) { Kolor = Color.Red; }

        public double Obwod => Dlugosc;
        public double Pole => PI * Promien * Promien;
        public new void Rysuj() => WriteLine($"Kolo{ToString()}, {this.Kolor}, obwod = {Obwod}, pole = {Pole}")
    }

    #endregion hierarchia klas
}

```

Przykładowe użycie klas i interfejsów tam zdefiniowanych przedstawia poniższy listing:

```

using System;
using Color = System.ConsoleColor;
using static System.Console;
using System.Collections.Generic;

namespace src_ex_figury
{
    public class Program
    {
        static void Main(string[] args)
        {
            Punkt p = new Punkt(1, 2);
            p.Rysuj();

            Odcinek odc = new Odcinek(p, new Punkt(2, 3));
            odc.Rysuj();
            odc.Skaluj(2);
            odc.Rysuj();

            Okrag okr = new Okrag(new Punkt(0, 0), 1);
            okr.Rysuj();

            Kolo k = new Kolo(new Punkt(0, 0), 1);
            k.Rysuj();

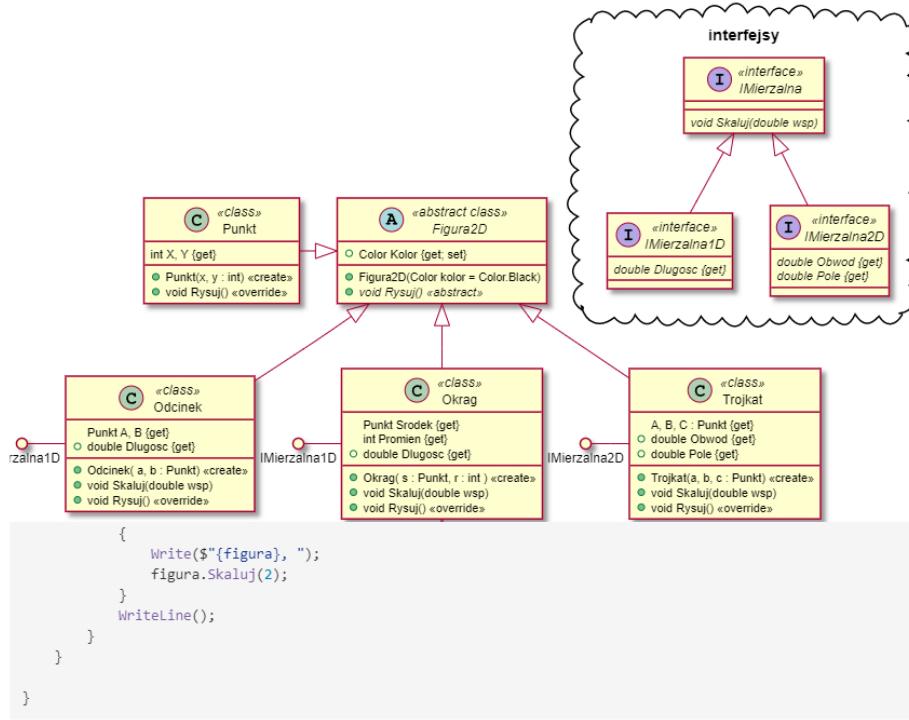
            Okrag okr2 = k;
            WriteLine(((Kolo)okr2).Pole);
            okr2.Rysuj();

            // == lista figur ==
            Stos<int> s = new Stos<int>();
            foreach (var figura in list<Figura2D> f)
            {
                s.Push(f.Pole);
            }
            Console.WriteLine("s.Count: " + s.Count);
            Console.WriteLine("s.IsEmpty");
            //Console.WriteLine("s.Capacity"); // compilation error - w interfejsie nie ma takiej metody
            Console.WriteLine(((Stos<int>)s).Capacity);
        }
    }

```

### Przykład - Figury:

Rozważmy inny przykład - hierarchię figur geometrycznych (np. w aspekcie tworzonej aplikacji graficznej) rysowanej na siatce z całkowitymi współrzędnymi (np. pixeli).



**Operatory is oraz as dla interfejsów**

W poprzednim temacie widzieliśmy, że można użyć operatora rzutowania, aby otrzymać referencję do interfejsu obiektu. Trochę lepszym rozwiązaniem niż rzutowanie jest użycie dedykowanego operatora `as`. Dlaczego? Jeśli spróbujesz wykonać rzutowanie referencji obiektu na referencję interfejsu którego klasa nie implementuje, otrzymasz w efekcie zgłoszenie wyjątku o nieprawidłowym rzutowaniu. Ten wyjątek powinieneś przechwycić i obsłużyć.

Więcej na temat: [Type-testing and cast operators \(C# reference\)](#)

Używając operatora `as` wykonujemy konwersję, jeśli jest ona możliwa. W przeciwnym przypadku operator zwraca `null`.

Operator `is` testuje, czy zmienna jest określonego typu, zaś operator `as` wykonuje rzutowanie na wskazany typ - nie zwracając wyjątku jeśli się nie powiedzie.

```
List<Figura2D> figury = new List<Figura2D>();
// dodano do listy kilka różnego typu figur
figury.Add( new Punkt(0,0) );
figury.Add( new Odcinek( new Punkt(1,1), new Punkt(2,2) ) );
figury.Add( new Okrag( new Punkt(0,0), 1 ) );
figury.Add( new Kolo( new Punkt(1,1), 2 ) );

double sumarycznePole = 0;
foreach( var figura in figury )
{
    11. IMierzalna2D fig = figura as IMierzalna2D;
    12. if( fig != null ) // w C#7 równoważne: !(fig is null)
        sumarycznePole += fig.Pole;
}

double sumarycznaDlugosc = 0;
foreach( var figura in figury )
{
    19. if( figura is IMierzalna1D )
        sumarycznaDlugosc += ((IMierzalna1D)figura).Dlugosc;
}
```

UWAGA: W C# 7 operator `is` został rozszerzony o możliwość dopasowania do wzorca. Zatem - zamiast operatora `as` - można tylko wykorzystać operator `is` w tym rozszerzonym wariantie (`zmienna is Typ wzorzec`) realizując bezpieczne rzutowanie (safe casting):

```
double sumarycznyObwod = 0;
foreach( var figura in figury )
{
    4. if(figura is IMierzalna2D f)
        sumarycznyObwod += f.Obwod;
}
```

## Główne zastosowanie interfejsów

Interfejsy opisują pewne oczekiwane funkcjonalności, jakie klasy lub struktury je implementujące powinny spełniać.

W standardowych bibliotekach C# jest zadeklarowanych wiele takich interfejsów, o kluczowym praktycznym znaczeniu:

- **`IEquatable` lub jego generyczna wersja `IEquatable<T>`.**  
Implementacja tego interfejsu oznacza, że w zbiorze instancji projektowanego typu zdefiniowano pojęcie "taki sam" - czyli własny (a nie domyślny) sposób stwierdzania, czy dwie instancje są sobie równe.
- **`IComparable` lub jego generyczna wersja `IComparable<T>`.**  
Implementacja tego interfejsu oznacza wdrożenie w projektowanym typie "porządku wewnętrznego".  
Każde dwie instancje typu są porównywalne (można stwierdzić, która jest *większa* a która *mniejsza*) - wiedza ta jest potrzebna chociażby do sortowania czy szybkiego wyszukiwania (*binary search*).
- **`IComparer` lub jego generyczna wersja `IComparer<T>`.**  
Klasa implementująca ten interfejs dostarcza metodę do porównywania dwóch obiektów.  
Wykorzystywane w celu dostarczenia "zewnętrznego" sposobu porównywania elementów dla algorytmów sortowania i szybkiego wyszukiwania.
- **`IDisposable`**  
Klasy implementujące ten interfejs zapewniają własne mechanizmy zwalniania zasobów zewnętrznych (np. dostęp do plików i strumieni danych czy bazy danych) i "współpracują" z instrukcją `using` (nie deklaracją `using`).
- **`ICloneable`**  
Klasy implementujące ten interfejs dostarczają własną, niestandardową metodę tworzenia kopii elementu (zamiast predefiniowanej `object.MemberwiseClone`).
- **`IEnumerable` lub jego generyczna wersja `IEnumerable<T>`.**  
Klasy (kolekcje) implementujące ten interfejs definiują *iterator* i między innymi pozwalają na przeglądanie swojej zawartości za pomocą pętli `foreach`. Typ `IEnumerable` wykorzystywany jest w technologii LINQ.
- **`ICollection`, `IList`, `ISet`, `IDictionary`, ...** - definiują minimalny zestaw funkcjonalności dla odpowiednich kolekcji.
- **`ISerializable`**

- ... i wiele innych.

Zwyczajowo, nazwy interfejsów poprzedzane są literą `I` - dla zwiększenia czytelności (np. `ICloneable`) - ale nie jest to obowiązek. Zwyczajowo, interfejsy dzielimy na te, które:

- definiują pożadaną cechę - i one występują w wariancie **przymiotnikowym** (np. `IComparable` czyli "porównywalny" lub `ICloneable` - klonowalny). Klasa implementująca takie interfejsy opisuje obiekty spełniające tę pożadaną cechę, jako prawdopodobnie jedną z wielu,
- definiują określona funkcjonalność poprzez metodę lub zestaw metod opisując "kontrakt" dla obiektów (np. `IComparator`, `ICollection`, `ISet`, `IList`, ...). Ich nazwy występują w wariancie **rzeczownikowym**,
- dostarczają meta-informacji na temat klasy implementującej.  
Teoretycznie interfejs może być pusty - nie zawierać żadnych metod. Klasa/struktura implementująca taki interfejs są "oznaczone", że należą do pewnej grupy. W teorii programowania obiektowego takie interfejsy nazywane są **marker interface**. Na przykład w Java jest nim `java.io.Serializable`. W C# nie jest zalecane używanie interfejsów pustych ([CA1040: Avoid empty interfaces](#)).

□

Rozważmy następujący kod, w którym zmiennej `kolekcja` typu "interfejsowego" `ICollection<int>` przypisujemy konkretny obiekt, np. typu `HashSet<int>`:

```
ICollection<int> kolekcja = new HashSet<int> {2, 1, 5, 3, 4};
kolekcja.Add(0);
// inne polecenia na kolekcji
Console.WriteLine(string.Join(' ', kolekcja)); // 2 1 5 3 4 0
```

Oczywiście na tak zdefiniowanej kolekcji możemy wykonywać działania zdefiniowane w interfejsie `ICollection` (bez rzutowania): `Add`, `Contains`, `Count`.

Jeśli jednak nagle zmienimy zdanie i oczekiwaliśmy raczej kolekcji, która przechowywałaby elementy w sposób posortowany (zgodnie z porządkiem zdefiniowanym w typie elementów), wystarczy zmiana kodu w jednej linijce:

```
① ICollection<int> kolekcja = new SortedSet<int> {2, 1, 5, 3, 4};
kolekcja.Add(0);
// inne polecenia na kolekcji
Console.WriteLine(string.Join(' ', kolekcja)); // 0 1 2 3 4 5
```

Deklarowanie zmiennej odwołującej się do interfejsu, następnie wykorzystanie tylko tych metod, które zdefiniowane są w interfejsie (bez rzutowania) oraz przypisanie tej zmiennej referencji do konkretnego obiektu pozwala na zwiększenie elastyczności kodu. W dowolnym momencie, przy niewielkim wysiłku, możemy zmienić kod uzyskując np. zwiększenie wydajności poprzez zastosowanie np. bardziej wydajnej struktury danych dla określonego problemu.

Przykład konkretny: W dostarczonym Ci kodzie, do przechowywania elementów wykorzystana jest lista (`List<T>`). Często dodajesz do niej elementy, często wypisujesz ją. Sporadycznie sprawdzasz, czy element należy do listy, elementy nie są usuwane z listy. W Twojej aplikacji zależy Ci jednak na jak najlepszym gospodarowaniu pamięcią (np. tworzysz rozwiązanie dla IoT). Struktura danych `List<T>` implementowana jest w tablicy i ma własność samorozszerzania. Dodanie nowych elementów powoduje automatyczne utworzenie nowej tablicy o większym rozmiarze, a następnie przekopiowanie do niej poprzedniej zawartości. Proces ten jest czasochłonny ( $O(n)$ ) i wymaga dodatkowej pamięci ( $O(n)$ ). Rozsądniejszym jest w tym przypadku zastosowanie `LinkedList<T>` (jeżeli rozmiar jest dopasowany do aktualnej liczby przechowywanych elementów, czas dodania nowego elementu jest stały  $O(1)$ ).

## Domyślna (*implicit*) i jawną (*explicit*) implementacja interfejsów

Domyślnym sposobem implementacji interfejsu jest prześłonięcie metod w nim zadeklarowanych:

```
interface IA
{
    void M();
}

class A : IA
{
    public void M() { Console.WriteLine("implementacja"); }
}

// Main()
A a = new A();
a.M();
```

**⚠ Ponieważ deklaracje metod w interfejsie są publiczne (mimo, iż tego się nie podaje), zatem metoda przesłaniająca musi być jawnie zadeklarowana z modyfikatorem `public`.**

W niektórych sytuacjach możemy/musimy użyć jawniej (*explicit*) implementacji interfejsu.

```
interface IA
{
    void M();
}
```

```

class A : IA
{
    void IA.M() { Console.WriteLine("implementacja"); }

    // Main()
    A a = new A();
    //a.M(); // compilation error
    ((IA)a).M();
}

```

### implementacja

- Implementacja *explicit* polega na wskazaniu pełnej kwalifikowanej nazwy w deklaracji (`void IA.M()`). **Nie wolno** w tym przypadku podawać modyfikatora dostępu.
  - Skoro zaimplementowana jawnie metoda nie ma modyfikatora dostępu, to jest prywatna, zatem nie jest dostępna z poziomu instancji
- ```
a.M(); // compilation error`
```
- Jeśli wywołamy metodę podając pełną kwalifikowaną nazwę (czyli wywołujemy metodę na rzecz interfejsu), będzie ona dostępna (w interfejsie - jej widoczność jest publiczna).
- ```
((IA)a).M();
```

Można rzec, że implementacja *explicit* metody interfejsu jest realizowana "po cichu". Metoda nie jest eksponowana z poziomu instancji. Oczywiście zawsze można ją wywołać wykorzystując referencję do interfejsu.

Jednym z przykładów jest użycie jawnej implementacji w celu ukrycia szczegółów interfejsu, który deweloper klas uważa za prywatny. Chociaż poziom prywatności nie jest tak wysoki, jak w przypadku użycia modyfikatora `private`, może być użyteczny w niektórych okolicznościach.

### Przykład praktyczny (konkretny) - implementacja `IDisposable`

Podstawowym zastosowaniem interfejsu `IDisposable` jest zwolnienie niezarządzanych zasobów. Moduł usuwania elementów bezużytecznych automatycznie zwalnia pamięć przydzieloną do obiektu zarządzanego, gdy ten obiekt nie jest już używany.

Poniższy kod przedstawia ogólny wzorzec zastosowania `IDisposable`. W tym przypadku metoda `Dispose()` jest ukryta przez jawną implementację, ponieważ jest ona tak naprawdę szczegółem implementacji, który nie jest istotny dla użytkowników klasy `MyFile`. Nie jest widoczna z poziomu mechanizmu IntelliSense.

```

//interface IDisposable
//{
//    void Dispose();
//}

class MyFile : IDisposable
{
    void IDisposable.Dispose()
    {
        Close();
    }

    public void Close()
    {
        // Do what's necessary to close the file
        System.GC.SuppressFinalize(this);
    }
}

```

□

Innym znaczącym przykładem wykorzystania implementacji *explicit* interfejsów jest rozwiązywanie konfliktów nazw przy implementacji wielu interfejsów. W poniższym przykładzie metoda `P()` oraz property `P` powodują konflikt nazw przy implementacji.

### Przykład:

```

interface ILeft
{
    int P { get; }
}
interface IRight
{
    int P();
}

class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}

```

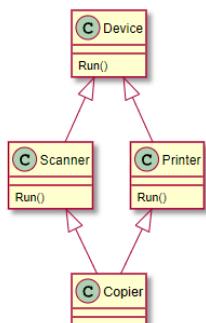
□

## Wielodziedziczenie

Wielokrotnie wcześniej zasygnalizowano, że interfejsy umożliwiają realizację wielodziedziczenia. Zresztą wymyślono je właśnie po to, aby uniknąć niejednoznaczności związanych z wielodziedziczeniem (tzw. *diamond problem*).

**Diamond problem:**

[https://en.wikipedia.org/wiki/Multiple\\_inheritance](https://en.wikipedia.org/wiki/Multiple_inheritance)

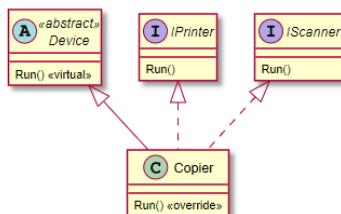


Która z metod `Run()` zostanie odziedziczona w klasie `Copier`?

W tym kontekście pojawia się wiele problemów, np. czy koparka powinna mieć jedną czy dwie kopie `Device` oraz jak rozwiązać niektóre niejednoznaczności odwołań.

W zasadzie problemy wielodziedziczenia można ominąć (najlepiej nie stosować, zastępując wzorcem delegacji) lub rozwiązać za pomocą jawnego określania zakresu, ale wiąże się to z dodatkowymi kosztami utrzymania kodu, jego złożonością i problemami wydajnościowymi (przeszukiwanie hierarchii klas).

Języki C#, Java zapewniają dziedziczenie po jednej klasie bazowej, więc teoretycznie problem nie występuje. Wielodziedziczenie symulowane jest implementacją wielu interfejsów. Interfejsy definiują sygnatury metod, ale nie przewidują konkretnej implementacji. Ta wykonywana jest dopiero na poziomie klasy konkretnej.



```
abstract class Device
{
    // ...
    public abstract void Run();
}

interface IPrinter
{
    void Run();
}

interface IScanner
{
    void Run();
}

class Copier : Device, IScanner, IPrinter
{
    public override void Run()
    {
        //...
    }
}
```

Problem rozwiążemy stosując jawną implementację interfejsu.

```
public abstract class Device
{
    // ...
    public abstract void Run();
}

public interface IPrinter
{
    void Run();
}
```

```

public interface IScanner
{
    void Run();
}

public class Copier : Device, IScanner, IPrinter
{
    void IScanner.Run() => Console.WriteLine("... scanning");
    void IPrinter.Run() => Console.WriteLine("... printing");

    public override void Run()
    {
        ((IScanner)this).Run();
        ((IPrinter)this).Run();
        // ...
    }
}

// Main()
Copier xerox = new Copier();
xerox.Run();

```

Implementacje *explicit* metod interfejsów są osiągalne jedynie poprzez referencje do interfejsów (np. `((IScanner)this).Run()`). Zatem inne metody klasy nie mają do nich bezpośredniego dostępu, jedynie poprzez rzutowanie na typ interfejsu.

Ograniczenie to ma istotny wpływ na dziedziczenie. Skoro inni członkowie klasy nie mają bezpośredniego dostępu do jawnych implementacji elementów interfejsu, to członkowie klas wywodzących się z klasy również nie mogą bezpośrednio uzyskać do nich dostępu. Zawsze należy uzyskać do nich dostęp poprzez odniesienie do interfejsu.

Więcej informacji i przykładów na temat jawnych implementacji interfejsów na stronie [C# Programming guide](#):

- [Explicit Interface Implementation](#),
- [How to: Explicitly Implement Interface Members](#),
- [Explicitly Implement Members of Two Interfaces](#).

## Podsumowanie

### Zalety stosowania interfejsów:

- Interfejsy, w powiązaniu z klasami abstrakcyjnymi, budują fundamenty biblioteki klas.
- Dostarczają abstrakcję projektowanego rozwiązania (*total abstraction*).
- Ich zastosowanie pozwala na zachowanie odpowiedniego poziomu elastyczności kodu (*lose coupling*). Do biblioteki możesz swobodnie dodać kolejne klasy konkretne, spełniające określone w interfejsach założenia.
- Są narzędziem umożliwiającym realizację programowania opartego na komponentach *component-based programming*.
- Umożliwiają bezpieczną i efektywną realizację wielodziedziczenia.
- Interfejsy "dodają mechanizm *plug and play*" do architektury aplikacji.

### Wady:

- Skompilowany kod z interfejsami jest mniej wydajny niż ten, bez ich stosowania.
  - Modyfikacja fundamentów architektury biblioteki klas (zmiana interfejsów czy klas abstrakcyjnych) najczęściej wymaga modyfikacji kodu klas implementujących i ponowną komplikację całości.
- Przykładowo, dopisanie deklaracji metody `int Wymiar {get;}` w interfejsie `IHierzalna` z przykładu opisującego hierarchię figur powoduje konieczność uzupełnienia kodu w praktycznie każdej klasie wchodzącej w skład tej hierarchii.

Problem ten częściowo rozwiązało się w C# 8, wprowadzając istotne rozszerzenia w definiowaniu i użyciu interfejsów.

Referencje:

- [Interfaces \(C# Programming Guide\)](#)
- [Interface \(C# Reference\)](#)
- [Interfaces In: Learning C# 3.0: Master the fundamentals of C# 3.0 by Jesse Liberty, Brian MacDonald, published by O'Reilly Media](#)