

## Interfejsy w C# 8 - nowe rozdanie

**UWAGA:** poszukując informacji na temat domyślnych implementacji metod interfejsów należy sugerować się datą publikacji. Wiele artykułów dostępnych w Internecie traktuje o *proposals*, które zostały zmienione w wersji dystrybucyjnej. Niektóre publikacje mogą wprowadzać w błąd!

(<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-8.0/default-interface-methods>)

## C# 8 - domyślna implementacja składników interfejsu

### Idea

W wersji 8 języka C# (a wcześniej w Java 8) złamano podstawową zasadę interfejsu, że wszystkie metody są abstrakcyjne (nie ma żadnej implementacji). Wprowadzono możliwość **domyślnej implementacji metod**. Zmiany obowiązują od października 2019 r.

Przykład wiodący (klasyczny) - tworzymy aplikację, w której implementujemy mechanizm *logowania* (rejestrowania) informacji o działaniu aplikacji. Chcemy mieć możliwość "przełączania" strumienia komunikatów między np. konsolę, plik, a może i inne, w tej chwili nie znane miejsce).

#### Przykład 1:

Zapisujemy interfejs `ILogger`:

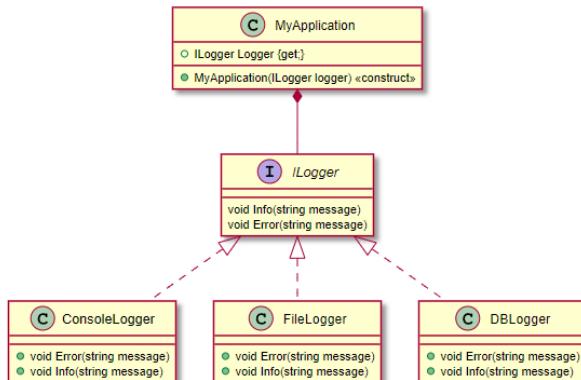
```
public interface ILogger
{
    void Info(string message);
    void Error(string message);
}
```

Implementujemy wariant logowania na konsolę:

```
public class ConsoleLogger : ILogger
{
    public void Error(string message)
    {
        Console.WriteLine("error: " + message);
    }

    public void Info(string message)
    {
        Console.WriteLine("info: " + message);
    }
}
```

i obiekt klasy `ConsoleLogger` wykorzystujemy w aplikacji.



Wyobraźmy sobie teraz, że interfejs `ILogger` z jakichś powodów został zmodyfikowany, poprzez dodanie deklaracji nowej metody:

```
public interface ILogger
{
    void Info(string message);
    void Error(string message);

    // New method
    void Warn(string message);
}
```

Próba komplikacji (C# 7.X i wcześniejsze) spowoduje błąd - brak implementacji metody `Warn` w klasach implementujących interfejs `ILogger`.

W C# 8 (.Net Core 3, .Net Framework 4.8, .Net Standard 2.1) umożliwiono **domyślną implementację** metody interfejsu - czyli zawarcie kodu, który zostanie wykonany w sytuacji nie przesłonięcia metody w klasach implementujących.

### Przykład 2:

```
// C# 8
public interface ILogger
{
    void Info(string message);
    void Error(string message);

    // New method with implementation (C# 8)
    void Warn(string message) => Debug.WriteLine("warn default: " + message);
}

public class OldConsoleLogger : ILogger
{
    public void Error(string message) => Console.WriteLine("error: " + message);
    public void Info(string message) => Console.WriteLine("info: " + message);
}

public class NewConsoleLogger : OldConsoleLogger, ILogger
{
    public void Warn(string message) => Console.WriteLine("warn: " + message);
}

// ===

static void Main()
{
    #if NET48 || NETSTANDARD2_1 || NETCOREAPP3_0
        Console.WriteLine("C# 8");
    #endif
    ILogger logger = new OldConsoleLogger();
    logger.Warn("ostrzeżenie od logger");
    ILogger logger1 = new NewConsoleLogger();
    logger1.Warn("ostrzeżenie od logger1");
    OldConsoleLogger logger2 = new OldConsoleLogger();
    //logger2.Warn("ostrzeżenie od logger2"); //compilation error, klasa ConsoleLogger nie implementuje `Warn`
    ((ILogger)logger2).Warn("ostrzeżenie od logger2"); // konieczne rzutowanie na interfejs
    NewConsoleLogger logger3 = new NewConsoleLogger();
    logger3.Warn("ostrzeżenie od logger3");
}
```

```
C# 8
warn default: ostrzeżenie od logger
warn: ostrzeżenie od logger1
warn default: ostrzeżenie od logger2
warn: ostrzeżenie od logger3
```

Wprowadzenie domyślnej implementacji metod w interfejsie - z technicznego punktu widzenia - rozszerza funkcjonalność tej konstrukcji języka.

Zwróćmy uwagę na kilka szczegółów:

- dodanie nowej metody do interfejsu, wraz z jej domyślną implementacją, nie narusza poprawności istniejącego kodu,
- możemy korzystać z nowej funkcjonalności, o ile zmienna odwołuje się do interfejsu:

```
① ILogger logger = new OldConsoleLogger();
    logger.Warn("ostrzeżenie od logger");
```

- Klasa `OldConsoleLogger` nie implementuje nowej metody `Warn`, ale możemy skorzystać z implementacji domyślnej, • jeśli jednak zmienna odwołuje się do klasy (1), a klasa nie implementuje metody domyślnej, pojawi się błąd kompilacji (2).

```
① OldConsoleLogger logger2 = new OldConsoleLogger();
② logger2.Warn("ostrzeżenie od logger2"); //compilation error, klasa ConsoleLogger nie implementuje `Warn`
③ ((ILogger)logger2).Warn("ostrzeżenie od logger2"); // konieczne rzutowanie na interfejs
```

Aby skorzystać z domyślnej implementacji, **musimy rzutować na interfejs** (3).

- w każdym przypadku, jeśli zaimplementujemy metodę, możemy odwoływać się do niej bezpośrednio (bez rzutowania).

```
ILogger logger1 = new NewConsoleLogger();
② logger1.Warn("ostrzeżenie od logger1");

NewConsoleLogger logger3 = new NewConsoleLogger();
③ logger3.Warn("ostrzeżenie od logger3");
```

- **Zapamiętaj:** klasa implementująca interfejs **nie dziedziczy** metody z domyślną implementacją, ale może z niej skorzystać - odwołując się do tej metody poprzez referencje do interfejsu

Przeciwnicy tego rozwiązania twierdzą jednak, że zakłóca ono planowaną architekturę i funkcjonalność hierarchii klas (już po zaprojektowaniu hierarchii interfejsów i klas można zmienić/uzupełnić założenia - bez negatywnych implikacji). Jest - swego rodzaju - wstrzykiwaniem nowej funkcjonalności do istniejącego kodu. Zwolennicy są zaś zachwyzeni.

Default implementations may actually turn out to be very useful. Suppose we have public library used by other developers all around the world. We want to release new version where we have improved some interfaces and – of course – we want to avoid breaking changes. As we have been on interfaces with our library for God knows how long then we cannot just jump between interfaces and abstract classes like we wish. If we add new members to our interfaces then we can define implementations so programs using previous versions of our libraries can use also new versions without changes to their codebase. I guess for many developers it makes life way easier.

W tym momencie interfejsy "zbliżają się" funkcjonalnością do klas abstrakcyjnych (również poprzez modyfikacje składowych). Jednocześnie C# 8 zmierza w kierunku języków funkcyjnych.

Rozbudujmy (w duchu języka C# 8) poprzedni przykład.

### Przykład 3:

```
// C# 8
public interface ILogger
{
    void Write(string message); // obowiązek implementacji
    void Info(string message) => Write("info: " + message);
    void Error(string message) => Write("error: " + message);
    void Warn(string message) => Write("warning: " + message);
}

public class ConsoleLogger : ILogger
{
    public void Write(string message) => Console.WriteLine(message);
}

public class ConsoleLogger1 : ILogger
{
    public void Write(string message) => Console.WriteLine(message);
    public void Warn(string message) => Console.WriteLine("debug warning: " + message);
}

// ===

//static void Main()
{
    ILogger logger = new ConsoleLogger();
    logger.Warn("ostrzeżenie");
    ILogger logger1 = new ConsoleLogger1();
    logger1.Warn("ostrzeżenie");
}
```

```
warning: ostrzeżenie
debug warning: ostrzeżenie
```

W powyższym przykładzie, odpowiednio projektując interfejs, unikamy powtarzania kodu (w klasach implementujących wymagana jest jedynie implementacja metody `Write()`, nikt nie zabrania ale i nie nakazuje implementowania pozostałych metod).

Ten sam efekt osiągneliśmy za pomocą klasy abstrakcyjnej (ale pamiętajmy - klasa dziedziczy tylko z jednej klasy, zaś implementować może wiele interfejsów).

## Konsekwencje

<https://www.talkingdotnet.com/default-implementations-in-interfaces-in-c-sharp-8/>

Złamanie fundamentalnej zasady deklarowania i działania interfejsu niesie ze sobą kolejne konsekwencje.

## Składniki prywatne interfejsów

Skoro w interfejsie możemy definiować kod, to - chociażby ze względów praktycznych - przydałoby się umieszczenie kodu prywatnego. **W C# 8 interfejsy mogą mieć składniki prywatne**, do ich własnego użytku:

```
public interface ILogger
{
    void Write(string message);
    void Info(string message) => Write("info: " + clean(message));
    void Error(string message) => Write("error: " + clean(message));
    void Warn(string message) => Write("warning: " + clean(message));

    private string clean(string message)
    {
        message.Trim();
        return DateTime.Now + ":" + message;
    }
    public const string version = "0.1.1";
}
```

UWAGA: w interfejsie **nie można definiować zmiennych instancji**. Można za to definiować stałe.

## Składniki statyczne interfejsów

Interfejsy **mogą definiować składniki statyczne**, mogą to być metody, ale również pola. Mogą być one użyte do parametryzacji domyślnej implementacji.

```
public interface ILogger
{
    void Write(string message);
    void Info(string message) {Write("info: " + clean(message)); count++;}
    void Error(string message) {Write("error: " + clean(message)); count++;}
    void Warn(string message) {Write("warning: " + clean(message)); count++;}

    private string clean(string message)
    {
        message.Trim();
        return DateTime.Now + ":" + message;
    }
    public const string version = "0.1.1";
    private static int count = 0;
    public static int CountMessages => count;
}
```

W powyższej implementacji (naciąganej na siłę) wymuszały implementację lub akceptację metody `CountMessages`, która zlicza liczbę logowanych komunikatów. Składniki statyczne są szczegółem implementacji, ale być może w niektórych sytuacjach są konieczne. Pierwotne założenie, że interfejsy "pracują na rzecz instancji" zostaje złamane.

## Składniki publiczne interfejsów

W C# 8 można dodać specyfikator dostępu `public` w deklaracji składników (w C# 7 kompilator wskazuje błąd). Jego użycie nie ma żadnych konsekwencji.

Można również deklarować składniki `internal` oraz `protected`.

## Hierarchie interfejsów

Interfejsy mogą budować swoje hierarchie dziedziczenia (odizolowane od hierarchii klas). Skoro pojawiają się implementacje, to pojawia się również ich przesłanianie.

We wcześniejszych wersjach C# metody interfejsu były abstrakcyjne (i wirtualne). Również teraz to się nie zmienia (dla metod bez modyfikatora). Możemy jednak dodawać modyfikatory `abstract` oraz `virtual` do sygnatur metod.

Drobne różnice pojawiają się w kontekście hierarchii interfejsów.

Modyfikator `abstract`: nie wnosi żadnej różnicy, metoda wymaga implementacji w klasie implementującej interfejs. Domyślnie, metody interfejsu bez implementacji domyślnej są `abstract`.

Modyfikator `virtual`: domyślnie metody interfejsu z implementacją są `virtual`, o ile nie zostały one oznaczone jako `private` lub `sealed`. Można ten fakt zaznaczyć modyfikatorem (ale nie jest to wymagane).

```
public interface ILogger2
{
    abstract void Write(string message); // abstract nie wymagane
    virtual void Info(string message) {Write("info: " + clean(message)); count++;} // virtual nie wymagane
    virtual void Error(string message) {Write("error: " + clean(message)); count++;}
    virtual void Warn(string message) {Write("warning: " + clean(message)); count++;}

    private string clean(string message)
    {
        message.Trim();
        return DateTime.Now + ":" + message;
    }
    public const string version = "0.1.1";
    private static int count = 0;
    public static int CountMessages => count;
}
```

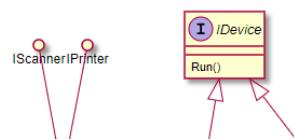
Oczywiście użycie `virtual` czy `abstract` w odniesieniu do metody prywatnej nie ma sensu!

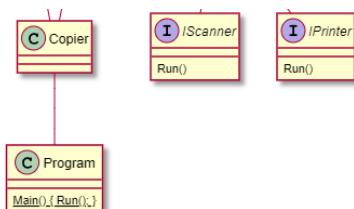
Zasady budowania hierarchii interfejsów i dziedziczenia domyślnych implementacji metod są takie same, jak dla hierarchii klas. Może z wyjątkiem słowa kluczowego `override`, którego użycie jest zabronione (metody przesłaniane bez modyfikatora `override` lub ukrywane z modyfikatorem `new` - pod groźbą ostrzeżeń).

Modyfikator `protected` ogranicza widoczność metod do odpowiedniej gałęzi w hierarchii interfejsów.

## Diamond problem:

Wprowadzenie możliwości implementacji metod w interfejsach wraca nas do problemu komplikacji z wielodziedziczeniem.





Która z metod `Run()` zostanie wywołana?

```

public static class Program
{
    public static void Main()
    {
        Copier xerox = new Copier();
        xerox.Run(); //compilation error
    }
}

interface IDevice
{
    void Run() { Console.WriteLine("Device Run");}
}

interface IScanner : IDevice
{
    void IDevice.Run() { Console.WriteLine("Scanner Run");}
}

interface IPrinter : IDevice
{
    void IDevice.Run() { Console.WriteLine("Printer Run");}
}

class Copier :
    IScanner, //compilation error
    IPrinter
{
}

```

error CS1061: Element "Copier" nie zawiera definicji "Run" i nie odnaleziono dostępnej metody rozszerzenia "Run", która przyjmuje pi  
error CS8705: Składowa interfejsu "IDevice.Run()" nie ma najbardziej specyficznej implementacji. Ani implementacja "IScanner.IDevice

W powyższym przykładzie kompilator wskazuje na 2 błędy:

Pierwszy błąd dotyczy ogólnej zasady - w klasie `Copier` nie przeprowadzono implementacji `Run()`. Zatem dostęp do domyślnych implementacji możliwy jest za pośrednictwem referencji do odpowiedniego interfejsu. Gdyby w metodzie `Main()` wykonać rzutowanie:

```
((IScanner)xerox).Run();
((IPrinter)xerox).Run();
```

błędu komplikacji by nie było.

Ale pozostały drugi błąd (związany z problemem wielodziedziczenia).

W dokumentacji Microsoft czytamy:

A class implementation of an interface member should always win over a default implementation in an interface, even if it is inherited from a base class. Default implementations are always a fall-back only for when the class does not have any implementation of the member at all. [C# Language Design Notes for Apr 19, 2017](#).

W momencie uruchomienia kodu środowisko .Net **musi** jednoznacznie wiedzieć, którą metodę ma wywołać.

Zatem - w klasie `Copier` należy zaimplementować metodę `Run()` przesłaniającą domyślne implementacje w interfejsach:

```

class Copier :
    IScanner,
    IPrinter
{
    public void Run() { Console.WriteLine("Copier run"); }

    // Main()
    Copier xerox = new Copier();
    xerox.Run(); // output: Copier run

```

Można również przesłonić domyślną implementację tą z interfejsu `IDevice`:

```

class Copier :
    IScanner,
    IPrinter
{

```

```
        void IDevice.Run() => Console.WriteLine("Copier run");
    }
```

Wtedy, bez względu na typ interfejsu, na który będziemy rzutować, uruchomi się właśnie ta nowa implementacja:

```
Copier xerox = new Copier();
((IScanner)xerox).Run(); // output: Copier run
((IPrinter)xerox).Run(); // output: Copier run
```

Ostrzeżenie: Poniższa implementacja jest błędna:

```
class Copier :
    IScanner,
    IPrinter
{
    public void Run()
    {
        ((IScanner)this).Run();
        ((IPrinter)this).Run();
    }

    // Main()
    Copier xerox = new Copier();
    xerox.Run(); // stack overflow
```

Kompilacja przechodzi. **Uuuups.** Po uruchomieniu otrzymujemy przepełnienie stosu - czyli rekurencja bez zakończenia.

Przyczyna:

1. wywołanie `xerox.Run()` prowadzi do metody `Run()` w klasie `Copier`.
2. wywołanie `((IScanner)this).Run()` - mimo, iż rzutujemy na interfejs, typem (dynamicznym) `this` jest `Copier`.
3. zatem ponownie wywołujemy metodę `Run()` z klasy `Copier`.
4. ... i mamy rekurencję.

Jeśli jednak do klasy `Copier` dopiszemy jawną implementację `Run` przesłaniającą tę z `IDevice`, błędu już nie będzie:

```
class Copier :
    IScanner,
    IPrinter
{
    void IDevice.Run() => Console.WriteLine("Copier run");
    public void Run()
    {
        ((IScanner)this).Run();
        ((IPrinter)this).Run();
    }

    // Main()
    Copier xerox = new Copier();
    xerox.Run();
    // output:
    // Copier run
    // Copier run
```

Jednak najlepszym rozwiązaniem jest nie używanie nazwy `Run()` w klasie `Copier` (Można zastąpić ją np. `CopierRun()`) oraz nie tyle przesłanianie *explicit* domyślnych implementacji metod, ile definiowanie nowych (`new`).

```
public static class Program
{
    public static void Main()
    {
        Copier xerox = new Copier();
        xerox.CopierRun();
    }
}

interface IDevice
{
    void Run() { Console.WriteLine("Device Run"); }
}

interface IScanner : IDevice
{
    new void Run() { Console.WriteLine("Scanner Run"); }
}

interface IPrinter : IDevice
{
    new void Run() { Console.WriteLine("Printer Run"); }
}

class Copier : IScanner, IPrinter
{
    public void CopierRun()
    {
        ((IScanner)this).Run();
        ((IPrinter)this).Run();
```

```
}
```

```
Scanner Run  
Printer Run
```

## Podsumowanie

The C# syntax for an interface in .NET compiler is extended to accept the new keywords in the interfaces which are listed below. For example, you can write a private method in the interface and the code will still compile and work.

- A body for a method or indexer, property, or event accessor
- Private, protected, internal, public, virtual, abstract, new, sealed, static, extern
- Static fields
- Static methods, properties, indexers, and events.
- Explicit access modifiers with default access is public
- Override modifiers

Not allowed:

- Instance state, instance fields, instance auto-properties

---

This feature also enables C# to interoperate with APIs that target Android or Swift, which support similar features. Default interface methods also enable scenarios similar to a "traits" language feature.

---

Referencje:

- [What's new in C# 8.0](#)
- [Tutorial: Update interfaces with default interface methods in C# 8.0](#)
- [Evolution of Interfaces in History of Java](#)