**[KURS C#] Gwarantowany zestaw złączy**



Rozmawialiśmy już o **dziedziczeniu klas**. Ma ono kilka zastosowań, m.in. pozwala przenieść wspólne fragmenty do klasy nadrzędnej jednocześnie zachowując możliwość **indywidualnych zachowań**.

Jednak cokolwiek sobie pomyślałeś poznając mechanizm dziedziczenia to tak naprawdę nie jest on aż tak potrzebny jak Ci się wydaje. To czego najczęściej potrzebujemy to pewnego **„zestawu złączy”** jakie powinna zawierać klasa. Dzięki temu będziemy mogli korzystać w taki sam sposób z obiektów wielu klas, które teoretycznie nie są ze sobą w ogóle związane, nie musząc wiedzieć jakie konkretnie klasy się tam mogą znaleźć.

Można to porównać do złącza **USB w komputerze** – oczekujemy, że urządzenia będą miały takie samo złącze USB, które podepniemy w ten sam sposób, i które zapewni takie same możliwości. Mimo, że tymi urządzeniami może być jednocześnie myszka, klawiatura czy smartfon. Dzięki temu, że mamy standard USB to z tego samego portu może korzystać bardzo wiele kompletnie nie związanych ze sobą urządzeń. I podobnego zachowania oczekujemy w naszych programach.

W języku **C#** takie możliwości dają **interfejsy**.

**Zestaw złączy**

Być może słowo **„interfejs”** od razu skojarzyło Ci się z jakimś panelem z przyciskami, albo właśnie jakimś typem wejść/wyjść jakiegoś urządzenia. Ewentualnie z interfejsem użytkownika w systemie. I są to skojarzenia jak najbardziej słuszne. Po prostu oczekujemy po interfejsie tego, że niezależnie co siedzi pod spodem to pewne **zestawy dostępnych opcji** są takie same. Dokładnie tak ma się sprawa z interfejsami w języku **C#**.

Innym porównaniem, które w pewnym sensie będzie oddawać ducha interfejsów w języku **C#** będą **kategorie**, **gatunki**. Przykładowo jeżeli mówimy, że książka jest z gatunku *fantasy* to znaczy, że posiada pewne cechy wspólne ze wszystkimi innymi książkami *fantasy*. Mimo, że różni je autor, bohaterowie, fabuła itd. Podobnie mamy tutaj.

Jednak nic nie działa tak dobrze jak przykład dlatego od razu przejdźmy do przykładu. A ponieważ jeszcze lepiej od przykładu działa przykład związany z tym co robimy to od razu zastosujmy interfejs w naszej **aplikacji bankowej**.

**Drukujemy**

Na dosyć wczesnym etapie powstawania aplikacji bankowej dla naszego kolegi Jacka dodaliśmy w niej coś takiego jak **drukarka**. Aktualnie potrzebowaliśmy jedynie drukować dane na ekranie w jeden konkretny sposób. Ale co jeżeli teraz w tych wszystkich miejscach chcielibyśmy drukować za pomocą faktycznej drukarki? Albo przy starcie systemu wybierać jaki styl drukowania albo jakiego urządzenia do tego przeznaczonego chcemy użyć?

W tym momencie każda taka zmiana albo dawanie wyboru powodowałyby, że trzeba wszędzie zmieniać wystąpienie naszej klasy na inne. A jeżeli dalibyśmy wybór to w ogóle mielibyśmy tyle różnych typów zmiennych ile drukarek.

Ktoś może powiedzieć, że przecież można zrobić **dziedziczenie** i wszystko będzie dziedziczyło z jednej klasy **Printer**. Ale czy ma to sens? Czy drukarka atramentowa i drukarka PDF mają część wspólną dlatego, że mogą robić to samo – drukować tekst? Tyle, że jedno z nich robi to na kartce, a drugie zapisuje do pliku? Niekoniecznie.

To co je łączy to jedna rzecz – metoda **„Drukuj”**. I dokładnie do tego możemy wykorzystać **interfejsy**! Tworzymy sobie pewien kontrakt, który określa, że wszystkie klasy zgodne z tym interfejsem muszą mieć konkretny zestaw metod.

Ale czy jedna będzie w tej metodzie robiła rzecz X, a druga Y to już nas w pewnym sensie nie obchodzi. Konkretnie to nie obchodzi to innych klas i metod, które z tych interfejsów korzystają. Bo jednak dobrze żebyśmy sami wiedzieli co nasz program wyczynia.

**Gatunek? Drukarka**

Wróćmy więc do naszej drukarki. Jej kod wygląda w ten sposób:

using System;

namespace Bank

{

class Printer

{

public void Print(Account account)

{

Console.WriteLine("Dane konta: {0}", account.AccountNumber);

Console.WriteLine("Typ: {0}", account.TypeName());

Console.WriteLine("Saldo: {0}", account.GetBalance());

Console.WriteLine("Imię i nazwisko właściciela: {0}", account.GetFullName());

Console.WriteLine("PESEL właściciela: {0}", account.Pesel);

Console.WriteLine();

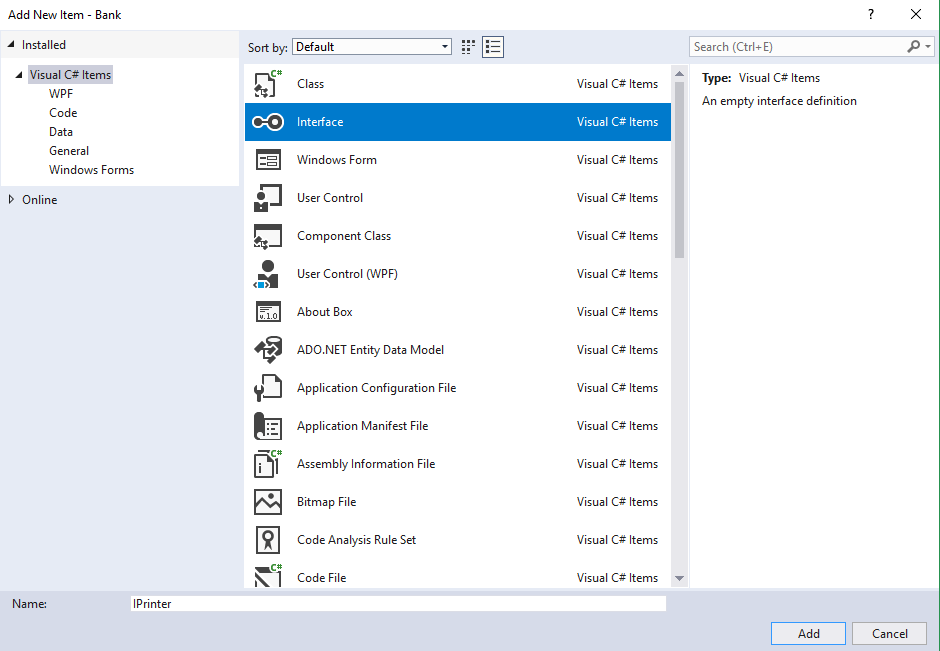
}

}

}

Jedna metoda, która odpowiada za drukowanie. Kiedy chcemy coś wydrukować na ekranie to po prostu tworzymy obiekt tej klasy i wykonujemy funkcję Print().

Czynnością, którą tutaj chcemy wykonać jest drukowanie. Dlaczego by więc nie przygotować tylko kontraktu mówiącego, że chcę mieć dostępną metodę Print(), a to co się będzie pod nią kryło wybiorę później? W ten sposób możemy dodać w naszym projekcie pierwszy interfejs! Zróbmy to klikając prawym przyciskiem myszy na nazwę projektu w **Solution Explorerze**, potem **New->New Item…** i z okna, które się pojawi wybieramy **Interface**. Jako nazwę podając **IPrinter**:



Po kliknięciu **„Add”** powinniśmy dostać plik z taką oto zawartością:

namespace Bank

{

interface IPrinter

{

}

}

Jest to właśnie nasz nowy **interfejs** dla drukarek. Na razie pusty. W tym momencie jedyna różnica względem klasy to zmiana słowa kluczowego z class na interface.

Drugą sprawą jest dodanie litery **I** (duże i) na początku nazwy interfejsu. Nie jest to konieczne, i bez tego jak najbardziej wszystko będzie działać. Jednak jest to przyjęta konwencja żeby łatwo było odróżnić **interfejs** od jego **implementacji**.

A jeżeli już przy **implementacji** jesteśmy, to tak jak jedna klasa dziedziczy po drugiej tak klasa interfejs implementuje. Robi się to w ten sam sposób, tzn. po **nazwie klasy** dodając **dwukropek** i **nazwę interfejsu**. Przejdźmy więc do klasy **Printer** i dodajmy w niej implementację naszego interfejsu:

using System;

namespace Bank

{

class Printer : IPrinter

{

public void Print(Account account)

{

Console.WriteLine("Dane konta: {0}", account.AccountNumber);

Console.WriteLine("Typ: {0}", account.TypeName());

Console.WriteLine("Saldo: {0}", account.GetBalance());

Console.WriteLine("Imię i nazwisko właściciela: {0}", account.GetFullName());

Console.WriteLine("PESEL właściciela: {0}", account.Pesel);

Console.WriteLine();

}

}

}

Jak widzisz zrobiliśmy to dodając IPrinter po dwukropku za nazwą klasy. Jednak najważniejsza różnica w porównaniu do dziedziczenia jest tutaj taka, że klasa może implementować **dowolnie dużo interfejsów**.

Tak samo jak książka czy film może być jednocześnie thrillerem, komedią i historią opartą na faktach. Tak samo klasa może jednocześnie należeć do kilku różnych „gatunków” tzn. implementować wiele różnych interfejsów.

**Po prostu drukuj**

Dobra, ale dopóki **interfejs** jest pusty to ta implementacja go niewiele nam dała. Dodajmy więc w nim **metodę**, której potrzebujemy:

namespace Bank

{

interface IPrinter

{

void Print(Account account);

}

}

Wygląda to podobnie jak przy **metodzie abstrakcyjnej**, którą pokazywałem przy okazji [omawiania dziedziczenia](https://zajacmarek.com/kurs-c-genetyka-klas/). Jedyna różnica jest taka, że po pierwsze nie ma tutaj słowa abstract, a po drugie nie ma **modyfikatora dostępu**.

Jest tak dlatego, że w interfejsie wszystkie metody są **domyślnie publiczne**. W końcu jaki sens by miało określanie jakie metody musi mieć klasa wewnątrz siebie, jeżeli nie mielibyśmy z nich żadnego pożytku bo byłby niedostępne na zewnątrz?

Od teraz każda klasa, która zaimplementuje taki interfejs będzie musiała posiadać podaną powyżej metodę. Dzięki temu wszędzie gdzie potrzebujemy klasy, która ją posiada możemy jako typ zmiennej podać właśnie ten **interfejs**.

Także od teraz nic nie stoi na przeszkodzie, żeby nasz system w wielu miejscach wymagał jedynie tego, żeby mieć dostęp do funkcji drukowania. Nie przejmując się, która klasa tak naprawdę tym drukowaniem się zajmie. Możemy więc jako typ zmiennej podawać od teraz IPrinter:

IPrinter printer = new Printer();

**Ćwiczenie 1**

Zamień w kodzie typ zmiennej przechowującej obiekt naszej drukarki z Printer na IPrinter. Uruchom program i zobacz czy wszystko działa tak jak poprzednio.

**Ćwiczenie 2**

Dodaj drugą klasę dla drukarki. Nazwij ją **SmallerPrinter** i zaimplementuj w niej interfejs **IPrinter**. Niech ta klasa nie wyświetla wszystkich informacji o koncie, a jedynie **numer konta** i **imię** i **nazwisko właściciela**.

Następnie spróbuj utworzyć obiekt tej klasy i przypisz go do wcześniej utworzonej zmiennej printer. Uruchom program i sprawdź czy nowa drukarka została poprawnie „podłączona”.

**Duża dowolność**

**Interfejsy** to naprawdę przydatny mechanizm w języku **C#**. To co tutaj zobaczyliśmy jest jedynie wstępem do tego na co pozwalają w dużych programach.

Ich siła staje się widoczna kiedy mówimy o wzajemnym wykorzystaniu klas. Jeżeli jedna klasa będzie potrzebowała wykorzystać jaką metodę innej klasy to zamiast oczekiwać obiektu tej konkretnie klasy możemy oczekiwać interfejsu. Dzięki temu oczekujemy jedynie dostępu do metody, nie łącząc dwóch klas na stałe.

To tak jak zastosowanie złącza USB w komputerze zamiast łączenia na stałe przewodu myszki z płytą główną. W pierwszym przypadku możemy w razie potrzeby po prostu odłączyć starą myszkę i podłączyć nową. W drugim przypadku operacja będzie zdecydowanie bardziej czasochłonna i błędogenna. Dokładnie tak samo jest w programie – im silniejsze powiązanie dwóch konkretnych klas tym podmiana jednej z nich będzie kosztowniejsza czasowo. Na szczęście mamy interfejsy.

# Kurs C# cz. 8 – Interfejsy

16 października 2016 o 11:00 Autor: [Marek Zając](https://zajacmarek.com/author/marekzajac/)

Ta część nie będzie może za długa, ale poruszę w niej konstrukcję języka, która w dobrych programach jest używana bardzo często. Pomówimy dzisiaj o interfejsach.

Przypomnij sobie tekst gdzie omawiałem budowę klas. Teraz usuń z klas wszystkie zmienne, prywatne metody i ciała metod publicznych – otrzymałeś interfejs ;)

A tak bardziej serio to interfejsy faktycznie mogą delikatnie przypominać klasy, ale na pewno mocno z klasami się łączą. Najłatwiej interfejs w C# będzie chyba wytłumaczyć porównując go do portów w komputerze. Wyobraź sobie, że interfejs jest np. portem USB. Sam port USB określa w jaki sposób są przesyłane dane, jak wygląda gniazdo itd. Ale to jakie konkretnie urządzenie podłączymy już portu USB nie obchodzi, tak samo nie obchodzi go co się stanie dalej z przesłanymi danymi, on tylko mówi jak mają być one przesyłane między komputerem i urządzeniem. Ważne jest tylko, żeby wtyczka pasowała. I właśnie podobnie działają interfejsy – określają sposób komunikowania się między klasami, ale jakie to są konkretnie klasy samego interfejsu już nie obchodzi. Dodatkowo tak samo jak jakieś urządzenie może obsługiwać kilka różnych portów tak samo klasa może implementować kilka interfejsów.

Budowa interfejsu w języku C# jest stosunkowo prosta:

interface IUsb

{

void Send(string data);

string Receive();

}

Jak widać słowo **class** zastąpione zostało słowem ***interface***. Wewnątrz interfejsu mamy deklaracje metod. Same deklaracje bez implementacji ponieważ interfejs służy tylko do zakomunikowania jakiej metody możemy użyć, nie określa jednak co dalej ona będzie robić. Dodatkowo nie użyłem tutaj żadnego słowa kluczowego określającego widoczność dalej metody. Jest tak ponieważ z założenia wszystkie metody w interfejsie będą zaimplementowane jako publiczne, w końcu jaki sens miałaby taka metoda służąca do komunikacji ze światem, która byłaby prywatna i niedostępna poza konkretną klasą?

W nazwie interfejsu przed słowem **Usb**dodałem literę **I**. Zrobiłem to nieprzypadkowo. Otóż konwencja nazewnicza języka C# sugeruje żeby nazwy interfejsów właśnie od tej litery się zaczynały i dobrą praktyką jest trzymanie się tego.

Ok, to teraz jak używać tych interfejsów? Jak utworzyć z ich pomocą jakieś obiekty? Interfejsu używamy na dwa sposoby, które są ze sobą ściśle związane.

Po pierwsze interfejs może być implementowany przez klasę. Korzystając z pokazanego przed chwilą interfejsu wygląda to w ten sposób:

class Usb : IUsb

{

public void Send(string data)

{

//zrób coś z danymi

}

public string Receive()

{

return "Dane";

}

}

Jak widać aby klasa implementowała dany interfejs podajemy jego nazwę za nazwą klasy, po dwukropku. Następnie już wewnątrz samej klasy musimy zdefiniować wszystkie metody, które dostarcza interfejs. Dodatkowo muszą one być publiczne.

Jeśli chcielibyśmy aby klasa implementowała kilka interfejsów podajemy kolejne nazwy po przecinku. Tutaj przykład klasy implementującej 3 interfejsy:

class Komputer: IUsb, IHdmi, IDvd

{

//tutaj implementacja

}

Oczywiście wewnątrz klasy powinny się znaleźć wszystkie metody dostarczane przez te interfejsy.

**Teraz pora na drugi sposób użycia interfejsu – użycie go jako typu zmiennej.**  
W tym momencie cały sens użycia interfejsów stanie się bardziej przejrzysty. Tak jak w poprzednich częściach kursu mogłeś utworzyć zmienne, których typem była konkretna klasa, którą utworzyłeś tak samo możesz utworzyć zmienne, których typem będzie interfejs. Pozostając przy pokazanych wcześniej przykładach:

IUsb usb;

Tutaj magii nie ma. Dodaliśmy zmienną typu ***IUsb***. Ale co teraz? Nie możemy utworzyć obiektu **IUsb** poprzez new IUsb();  ponieważ nie da się utworzyć obiektu interfejsu. I teraz właśnie zaczyna się magia związana z interfejsami.

**Do tak przygotowanej zmiennej możemy przypisać obiekt klasy implementującej dany interfejs:**

IUsb usb = new Usb();

Korzystając teraz z tej zmiennej będziemy mieli dostęp tylko do metod, które w danym interfejsie zadeklarowaliśmy nawet jeśli ta konkretna klasa posiadała by ich więcej.

Gdyby w naszym programie było kilka klas, które implementują ten interfejs możliwe by było przypisanie obiektów z nich utworzonych do podanej powyżej zmiennej i używanie implementacji metod z każdej z nich.

Dobrym przykładem tutaj są kolekcje czy tablice w języku C#. Większość z nich implementuje interfejs IEnumerable, który określa m.in. że dostępne są metody pozwalające przechodzić po kolejnych elementach kolekcji jednak w żaden sposób nie narzuca tego jak dana klasa będzie to obsługiwała pod spodem. Dzięki temu niezależnie od tego której kolekcji użyjemy możemy mieć pewność, że każda z nich będzie zawierała pewien zestaw metod, które możemy wykorzystać. Z tego faktu korzystają chociażby biblioteki dostarczone wraz z frameworkiem .NET, które pozwalają np. sortować kolekcje. Nie obchodzi ich jaka kolekcja jest użyta i w jaki sposób przechowuje dane. Im wystarczy, że mogą np. wywołać metodę Next() itp.

Tak jak możemy tworzyć zmienne, których typem jest interfejs tak samo możemy dodawać metody, które przyjmują bądź zwracają interfejs i jest to najczęściej używana cecha tych konstrukcji. Przykładowa metoda może wyglądać w ten sposób (zakładamy, że IUsb i ICom to interfejsy dodane w naszym programie):

public ICom Convert(IUsb usb)

{

// coś się dzieje

}

Powyższa metoda przyjmuje jako parametr obiekt klasy implementującej interfejs IUsb i zwraca obiekt klasy implementującej interfejs ICom.

I to na tyle jeśli chodzi o tą część kursu dotyczącą interfejsów. Przekazałem Wam tutaj większość niezbędnych informacji dotyczących tej konstrukcji językowej. Być może na początku nie za bardzo będziecie czuli sens stosowania interfejsów jednak polecam próbować dodawać je czasami nawet na siłę tam gdzie następuje komunikacja pomiędzy różnymi klasami. Metoda jakiejś klasy przyjmuje w parametrze obiekt innej klasy? Dodaj interfejs z metodami z drugiej klasy, które są używane przez pierwszą, spraw żeby druga klasa go implementowała, a w metodach pierwszej klasy zamień typ parametru z drugiej klasy na interfejs, który ona implementuje.