**[KURS C#] Nie wszystko publiczne**



**UWAGA!**

**Nowa wersja kursu dostępna** jest na moim portalu z kursami. Nadal jest on **BEZPŁATNY**! A jednocześnie posiada więcej lekcji, wygodną nawigację i możliwość zapamiętania, w którym miejscu skończyłeś ostatnio!. Znajdziesz go [**POD TYM LINKIEM**](https://kurs-szarpania.pl/nauka/product/szkola-szarpania-intro)!

**System bankowy** posiada już możliwość zakładania kont. Wiemy jak wykonywać pierwsze operacje za pomocą funkcji i potrafimy przechowywać konta w przeznaczonych na nie pudełkach.Ale bank jest też instytucją, która nie wszystko chce pokazywać **publicznie**. Są rzeczy, które skrywa tylko dla siebie. Podobnie mają klasy. Też nie chcą się dzielić wszystkim co mają, czasami mają coś do ukrycia. I dzisiaj o tym porozmawiamy – jak dać klasom odrobinę **prywatności**.

**Pokażesz? Nie!**

Każdy z nas posiada jakieś rzeczy prywatne, którymi nie dzielimy się ze wszystkimi. Jest też część rzeczy, którymi podzielimy się z rodziną czy przyjaciółmi, ale nie z kimś obcym. Posiadamy różne mechanizmy dające nam możliwość zachowania tej **częściowej prywatności**. Czy są to hasła do komputera czy skrzynki mailowej, czy zamki w drzwiach, albo inne tego typu elementy.

Ale przedmioty codziennego użytku też posiadają swoje tajemnice. **Ukrywają** pewne szczegóły, które są nieistotne dla użytkownika ale niezbędne do działania tego przedmiotu. Ot chociażby zegar. Mechanizm, który porusza wskazówkami jest przed nami zazwyczaj ukryty. Nie mamy potrzeby go oglądać ani tym bardziej bezpośrednio w nim grzebać. Chcemy tylko żeby zegar działał, wskazywał poprawną godzinę i dawał możliwość jej ustawienia. Podobnie sprawa ma się z klasami. One też mogą posiadać takie **ukryte fragmenty**.

Takie ukrywanie fragmentów klasy przed dostępem z zewnątrz przydaje się w przypadku kiedy potrzebujemy dodać jakąś **metodę** albo **zmienną**, której będziemy używali **wewnątrz klasy**, ale absolutnie nie chcemy aby dało się jej użyć z zewnątrz. Np. dlatego, że zmienienie czegoś spoza klasy mogłoby spowodować, że jej działanie stałoby się nieprzewidywalne albo po prostu nie robiła by tego co powinna.

**Ukrywanie szczegółów**

W tym miejscu poznajemy jedynie podstawy **modyfikatorów dostępu** (bo o nich jest właśnie mowa).

Działamy w obrębie jednego projektu i rozpatrujemy najpowszechniejsze przypadki. Dlatego rodzaje dostępu jakie tutaj omawiamy to:

* publiczny
* prywatny
* chroniony

**Dostęp publiczny**

To ten, którego używaliśmy do tej pory. Pamiętasz jak mówiłem żeby do wszystkiego w naszych klasach dodawać słowo public? To właśnie było to. Właśnie dodanie słowa public oznacza, że dana zmienna albo metoda będzie dostępna z zewnątrz.

A co oznacza **dostęp** z zewnątrz? Oznacza, że kiedy utworzymy obiekt takiej klasy to będziemy mogli się dostać spoza tej klasy do wszystkich **elementów publicznych**. Tak jak to robiliśmy do tej pory. Dlatego właśnie oznaczaliśmy wszystko jako publiczne, żeby móc wykonać np. taką operację na obiekcie klasy **Account**:

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();

}

To właśnie **dostęp publiczny** pozwolił nam „dobrać się” do zmiennych i metod klasy **Account** poza nią samą. W tym wypadku umożliwiło nam to wyświetlenie danych konta przez naszą drukarkę.

Oprócz tego oczywiście publiczne elementy możemy używać w klasach, które dziedziczą po klasie, w której taki publiczny element się znajduje.

**Dostęp prywatny**



Elementy oznaczone jako prywatne to są właśnie te **najbardziej ukryte**. Tworzymy je korzystając ze słowa kluczowego private. Mamy do nich dostęp tylko **wewnątrz klasy** w jakiej się znajdują. Są wykorzystywane najczęściej do funkcji, w których chcemy wydzielić jakieś fragmenty kodu, a które mają sens tylko wewnątrz tej jednej klasy.

Przykładowo mamy metodę publiczną, która robi kilka operacji. I teraz uznajemy, że część z tych operacji dobrze by było zamknąć jako jedną osobną funkcję, którą możemy w tym miejscu użyć, albo, którą będziemy mogli wykorzystać też w innej funkcji w tej klasie. Wtedy ten wydzielony fragment kodu zamykamy w funkcji prywatnej, która jest spokojnie używana wewnątrz klasy, np. w publicznych metodach, ale nikt inny, nikt poza tą klasą nie wie o jej istnieniu i nie może jej wykorzystać. Nawet **klasy dziedziczące** nie mają do tego elementu dostępu i nie wiedzą o jego istnieniu.

Warto zapamiętać, że jeżeli przy jakiejś zmiennej albo metodzie w klasie nie podamy żadnego modyfikatora to **domyślnie** ustawiany jest właśnie **private**!

W tym miejscu możemy stworzyć przykład zastosowania funkcji prywatnej i przy okazji usprawnić naszą aplikację ograniczając ilość parametrów konstruktora kont.

**Ale numer**

Jako, że **numer konta** ma określoną strukturę, która bazuje na **ID konta** (numerze identyfikacyjnym, coraz większym dla kolejnych kont) to nic nie stoi na przeszkodzie, a nawet jest wskazane, żeby to konto generowało ten numer dostając jedynie informację o **ID**.

Taka funkcja, która zwróci numer konta jak najbardziej nadaje się aby być funkcją prywatną. Chcemy wydzielić ją jako jeden osobny blok kodu, a jednocześnie jest to coś co dotyczy tylko tej klasy i nie powinno być dostępne z zewnątrz.

Przypomnijmy sobie jak wygląd numer konta w naszej aplikacji:

* Jest to **12 znaków**, z których pierwsze dwa to liczba **94**, a pozostałe **10** to **ID konta** uzupełnione z przodu cyfrą **0**. Przykładowo dla **ID** równego **5** **numer konta** ma postać **94000000000**5.

Funkcja, która takie coś nam zwróci może wyglądać w ten sposób:

string generateAccountNumber(int id)

{

var accountNumber = string.Format("94{0:D10}", id);

return accountNumber;

}

Ta dziwna wartość :D10 w funkcji do formatowania tekstu oznacza *„wstaw tutaj liczbę, która jest argumentem, ale całość uzupełnij zerami tak żeby w sumie było 10 znaków”*. Czyli dokładnie to czego potrzebujemy.

**Nowe pole w klasie**

Jednak skoro potrzebujemy **ID** to skądś je musimy wziąć. Dodajmy więc kolejne pole, tym razem typu int , do klasy **Account**. Będzie ono wypełniane przez **konstruktor**, więc musimy dodać parametr id w konstruktorze **klasy bazowej** i w **klasach dziedziczących**.

namespace Bank

{

abstract class Account

{

public int Id;

public string AccountNumber;

public decimal Balance;

public string FirstName;

public string LastName;

public long Pesel;

public Account(int id, string accountNumber, decimal balance, string firstName, string lastName, long pesel)

{

Id = id;

AccountNumber = accountNumber;

Balance = balance;

FirstName = firstName;

LastName = lastName;

Pesel = pesel;

}

namespace Bank

{

class SavingsAccount : Account

{

public SavingsAccount(int id, string accountNumber, decimal balance, string firstName, string lastName, long pesel)

:base(id, accountNumber, balance, firstName, lastName, pesel)

{

}

namespace Bank

{

class BillingAccount : Account

{

public BillingAccount(int id, string accountNumber, decimal balance, string firstName, string lastName, long pesel)

:base(id, accountNumber, balance, firstName, lastName, pesel)

{

}

**Generujemy numery**

Teraz taką funkcję z powodzeniem można użyć w klasie **Account** i za jej pomocą **generować numer konta** w **konstruktorze**:

namespace Bank

{

abstract class Account

{

public int Id;

public string AccountNumber;

public decimal Balance;

public string FirstName;

public string LastName;

public long Pesel;

public Account(int id, string firstName, string lastName, long pesel)

{

Id = id;

AccountNumber = generateAccountNumber(id);

Balance = 0.0M;

FirstName = firstName;

LastName = lastName;

Pesel = pesel;

}

public abstract string TypeName();

public string GetFullName()

{

string fullName = string.Format("{0} {1}", FirstName, LastName);

return fullName;

}

private string generateAccountNumber(int id)

{

var accountNumber = string.Format("94{0:D10}", id);

return accountNumber;

}

}

}

Usunęliśmy od razu przekazywanie numeru konta w konstruktorze. Skoro ustawiamy go wewnątrz klasy to nie ma potrzeby przyjmowania go z zewnątrz.

Mam nadzieję, że zauważyłeś dodaną na końcu klasy **funkcję**, o której rozmawialiśmy. Jak widać ma ona ustawiony **modyfikator dostępu** private, co oznacza, że np. w próba wykonania czegoś takiego:

Account account = new SavingsAccount(....);

Console.WriteLine(account.GenerateAccountNumber(1));

Spowoduje wyświetlenie się błędu bo do tej metody nie mamy **dostępu poza klasą**.

Za to użyta jest ona **wewnątrz klasy**. Konkretnie w konstruktorze. Zamiast przyjmować **numer konta** jako parametr po prostu generujemy go z użyciem napisanej przed chwila **funkcji** i **ID**, które dostajemy.

Przy okazji usunąłem też z **parametrów konstruktora** wartość odpowiedzialną za **stan konta**. W końcu każde konto, które zakładamy **jest na początku puste**. Dlatego spokojnie możemy tą wartość również ustawiać w konstruktorze. Tym razem po prostu korzystając ze stałej liczby **0.0**.

Oczywiście ograniczenie ilości parametrów w klasie **Account** wiąże się też z modyfikacją klas, które po niej dziedziczą. Jako, że korzystamy tam z **konstruktora bazowego**. Dlatego w klasie **SavingsAccount** konstruktor wygląda teraz w ten sposób:

public SavingsAccount(int id, string firstName, string lastName, long pesel)

: base(id, firstName, lastName, pesel)

{

}

Analogicznie sprawa się ma z konstruktorem klasy **BillingAccount**.

**Dostęp chroniony**

Ostatni typ **modyfikatora dostępu** jaki tutaj omawiamy do **dostęp chroniony** czyli protected.

Jest on ściśle związany z omawianym poprzednio dziedziczeniem. A to dlatego, że działa on prawie tak samo jak **dostęp prywatny** z tą różnicą, że uchyla rąbka tajemnicy **klasom dziedziczącym**. Czyli nadal mamy element, którego nie użyjemy gdzieś na zewnątrz tworząc obiekt klasy. Ale jednocześnie mamy dostęp do tego elementu w klasach dziedziczących, dzięki czemu mogą one w jakiś sposób go wykorzystać.

Przykładowo możemy mieć metodę, która ustawia jakąś wartość i wykorzystujemy ją wewnątrz klasy bazowej chociażby w konstruktorze. Ale oznaczając ją jako **protected** mamy możliwość użycia jej w klasie dziedziczącej, która przy jakiejś operacji będzie potrzebowała w ten sam sposób ustawić tą samą wartość. Dzięki temu jednocześnie unikamy duplikacji kodu pomiędzy klasą bazową i dziedziczącą oraz zachowujemy prywatność pewnych zmiennych i funkcji.

Nie jest on najczęściej wykorzystywanym **modyfikatorem**. Ale mimo wszystko warto go znać i wiedzieć kiedy warto zastosować.

**Hermetyzacja**

**Hermetyzacja** zwana też **enkapsulacją** oznacza udostępnianie na zewnątrz klasy tylko niezbędnego **minimum informacji**. Jest ona możliwa właśnie dzięki poznanym przed chwilą **modyfikatorom dostępu**.

Zawsze projektując jakąś klasę powinniśmy się zastanowić do czego dostęp jest niezbędny z zewnątrz. I tylko tym elementom ustawić modyfikator **public**. Wszystkie pozostałe nich zostaną **prywatne**. Dzięki temu ograniczamy ryzyku zmiany stanu obiektu w sposób jakiego nie przewidzieliśmy.

Przykładowo niech nasz system nadaje kontom z góry narzuconą wartość **ID**. Jest ona generowana na podstawie już istniejących obiektów. I niech teraz większość metod do szukania, sortowania itd. opiera się o tą wartość. Jeżeli udostępnimy metodę do zmiany wartości **ID** konta na zewnątrz, a więc oznaczymy jako public to nic nie stoi na przeszkodzie aby ktoś korzystający z naszej części kodu podmienił ją na inną.

Skutki mogą być takie, że nagle ten obiekt zacznie być postrzegany jako zupełnie inny! A co gorsza może się okazać, że w katalogu kont nagle pojawią się dwa konta z takim samym ID, bo jedno z nich będzie miało to ID w niepoprawny sposób zmienione! Jest duża szansa, że spowoduje to poważne problemy, których w dodatku nie zobaczymy od razu.

Dlatego zasada jest prosta – **domyślnie wszystko ustawiamy na private**. Dopiero kiedy **MUSIMY** z tego skorzystać gdzieś indziej to zastanawiamy się czy trzeba zmienić modyfikator dostępu na **protected** lub **public**.

**Zachowajmy to w tajemnicy**



Jak już wspomniałem na początku tej części bank wiąże się z pewnym ukrywaniem informacji. Nie chcemy żeby sąsiad wiedział ile mamy na koncie. Dlatego kontrolując jakie dane i metody są dostępne dla innych części systemu zapewniamy sobie większą **prywatność** i **bezpieczeństwo**.

Ograniczamy też ryzyko związane z nieprzewidzianymi modyfikacjami informacji. Dlatego nie wszystko w naszych klasach powinno być otwarte dla każdego. Tak samo jak nasze mieszkanie posiada drzwi, które posiadają zamek, który kontrolujemy wybierając kogo chcemy wpuścić.