# [KURS C#] Zrobisz coś dla mnie?



**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)!

Mimo, że bank jeszcze oficjalnie nie wystartował z działalnością nasza aplikacja dla Jacka już posiada pierwsze konta! Dlatego nie zwalniamy tempa i dzisiaj poznamy sposób na sprawienie aby różne elementy naszego projektu mogły wykonywać jakieś czynności – poznamy jak się wprowadza do programu **funkcje**.

Z funkcji już korzystaliśmy w kodzie. Jednak po pierwsze ktoś już je za nas napisał, a po drugie warto by było jednak wiedzieć co my z nimi właściwie robimy i robić możemy.

# Coś trzeba zrobić?

Klasy opisują jak ma być coś zbudowane. Zmienne to przechowują. Ale program jest po to żeby coś robił. I właśnie do tego służą funkcje. **Funkcje** odpowiadają za czynności, za jakieś procesy. Kiedy mówimy, że program coś robi to mamy zazwyczaj na myśli używanie jednej lub większej ilości funkcji.

Kiedy omawialiśmy teoretycznie najważniejsze pojęcia to wspomniałem, że na tym etapie możemy zamiennie używać pojęcia funkcja i metoda, dlatego nie zdziw się jak spotkasz w tekście oba te słowa – będą oznaczały to samo.

Na poziomie na jakim składamy nasze projekty, **metody**, które dodajemy zawsze korzystają z innych metod. Czy to tych, które również my napisaliśmy jak i tych dostarczonych nam poprzez **.NET Framework**. Więc pierwszą informacją na temat funkcji w C# jest to, że możemy używać jednej funkcji wewnątrz drugiej.

## Dane dla funkcji

Jak już wspomniałem w innej części **funkcje w programowaniu** są podobne do funkcji w matematyce. Jednak różnica jest taka, że zarówno dane dla funkcji jak i zwracanie wyniku są opcjonalne. Więc nie ma problemu, żeby nasza funkcja nie dostała żadnych informacji i dopiero wewnątrz siebie jakieś dane stworzyła i coś z nimi zrobiła. Nie jest też niczym dziwnym, że funkcja wzięła jakieś dane, ale na końcu nie zwróciła niczego.

Jednak zarówno w jednym jak i drugim przypadku najczęściej mamy do czynienia z funkcjami, które zmieniają **stan klasy**, w której się znajdują, czyli modyfikują te zmienne, które znajdują się w naszych obiektach. Bo kolejną informacją o **funkcjach w C#** jest to, że zawsze muszą być przez nas pisane w jakiejś klasie. Nie możemy dodać funkcji niezwiązanej z żadną klasą.

# Budujemy drukarkę

Nasz program bankowy się rozrasta. Po ostatnich zmianach mamy już **trzy konta**, których dane w dodatku wyświetlamy na ekranie:

namespace Bank

{

class Program

{

static void Main(string[] args)

{

string name = "Nazwa: Bank";

string author = "Autor: Marek Zajac";

Console.WriteLine(name);

Console.WriteLine(author);

Console.WriteLine();

SavingsAccount savingsAccount = new SavingsAccount();

savingsAccount.AccountNumber = "940000000001";

savingsAccount.Balance = 0.0M;

savingsAccount.FirstName = "Marek";

savingsAccount.LastName = "Zając";

savingsAccount.Pesel = 92010133333;

SavingsAccount secondSavingsAccount = new SavingsAccount();

secondSavingsAccount.AccountNumber = "940000000002";

secondSavingsAccount.Balance = 0.0M;

secondSavingsAccount.FirstName = "Marek";

secondSavingsAccount.LastName = "Zając";

secondSavingsAccount.Pesel = 92010133333;

Console.WriteLine("Dane konta:");

Console.WriteLine("Numer konta: {0}", savingsAccount.AccountNumber);

Console.WriteLine("Saldo: {0}zł", savingsAccount.Balance);

Console.WriteLine("Imię właściciela: {0}", savingsAccount.FirstName);

Console.WriteLine("Nazwisko właściciela: {0}", savingsAccount.LastName);

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

Console.WriteLine();

Console.WriteLine("Numer konta: {0}", secondSavingsAccount.AccountNumber);

Console.WriteLine("Saldo: {0}zł", secondSavingsAccount.Balance);

Console.WriteLine("Imię właściciela: {0}", secondSavingsAccount.FirstName);

Console.WriteLine("Nazwisko właściciela: {0}", secondSavingsAccount.LastName);

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

BillingAccount billingAccount = new BillingAccount();

billingAccount.AccountNumber = "940000000003";

billingAccount.Balance = 0.0M;

billingAccount.FirstName = savingsAccount.FirstName;

billingAccount.LastName = savingsAccount.LastName;

billingAccount.Pesel = savingsAccount.Pesel;

Console.WriteLine();

Console.WriteLine("Numer konta: {0}", billingAccount.AccountNumber);

Console.WriteLine("Saldo: {0}zł", billingAccount.Balance);

Console.WriteLine("Imię właściciela: {0}", billingAccount.FirstName);

Console.WriteLine("Nazwisko właściciela: {0}", billingAccount.LastName);

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

Console.ReadKey();

}

}

}

Powoli zawartość funkcji Main() przestaje się mieścić na ekranie. Jest to dobry moment aby skorzystać z możliwości pisania **własnych funkcji**.

Drugim dobrym momentem jest ten kiedy jakieś **czynności się powtarzają**. Wyobraźmy sobie fabrykę jakichś urządzeń. Produkcja każdej sztuki wygląda tak samo. Bierze się materiały i na podstawie z góry ustalonego planu je obrabia. Nikt tam nie przenosi np. plastiku w wiadrze, nie wrzuca do maszyny, która go podgrzeje, potem ręcznie wybiera i zanosi do maszyny, która coś z niego uformuje itd. Tam tylko daje się plastik czy inny materiał na wejściu. Całość działa automatycznie wg z góry ustalonego sposobu postępowania i na końcu maszyna zwraca gotowy produkt.

Tymi pojedynczymi maszynami są **operacje**, które wykonujemy. A tą całą automatyczną linią produkcyjną są nasze funkcje, w których możemy zamknąć powtarzające się czynności i korzystać z nich jako jednej całości, która dostając coś na początku zwraca nam gotowy produkt.

W powyższym kodzie pierwszą rzeczą, która nadaje się do zamknięcia w jednej funkcji jest **wypisywanie danych konta na ekranie**. Robimy to za każdym razem tak samo, więc bez sensu dodawać każdą linijkę oddzielnie. Zróbmy więc własną drukarkę w postaci funkcji wypisującej dane podanego konta w konsoli systemowej.

Jeżeli miałeś już do czynienia z kursami programowania obiektowego czy to znalezionymi samodzielnie czy np. w postaci ćwiczeń na uczelni to pewnie spotkałeś się z przykładami gdzie drukowanie danych na ekranie było funkcją zawartą w klasie, której dane chcieliśmy wydrukować.

Ile razy zdjęcie, które zrobiłeś potrafiło wydrukować samo siebie? No właśnie. Dlatego tutaj będziemy dodawać funkcję drukującą w osobnej klasie-drukarce.

## Planujemy druk

Na początek ustalmy co nasza drukarka ma robić.

Powinna dać możliwość podania obiektu konta, o którym informacje chcemy wydrukować. No i powinna te informacje wydrukować na ekranie. Zawsze tak samo. Tak więc **danymi wejściowymi** będzie konto. Nie mamy nic do zwrócenia w tym wypadku. Drukarka wszystko co zrobi to wyświetli tekst na ekranie.

Robimy wszystko na porządnie, w końcu nie chcemy koledze oddać czegoś byle jakiego, dlatego nasza drukarka będzie **nowym obiektem**. A jej funkcją będzie **drukowanie**.

## Projektujemy drukarkę



Dodawanie **klas** nie powinno stanowić dla Ciebie problemu. Dodajmy więc klasę ***Printer*** do naszego projektu – to będzie **schemat drukarki**, której potrzebujemy:

namespace Bank

{

class Printer

{

}

}

Ale to już znasz. Pora więc na najważniejsze, czyli na funkcję, która nam wydrukuje dane kont:

namespace Bank

{

class Printer

{

public void Print(SavingsAccount account)

{

}

}

}

Jeszcze nie przenieśliśmy samego drukowania do niej. W tym momencie będzie „wypluwać puste kartki” ;) Na początek omówmy co tutaj się zadziało.

### Jak to jest zrobione?

Jak już wspomniałem, funkcję musieliśmy umieścić w klasie – tak się stało.

Teraz dowiedzmy się z czego tak naprawdę się ona składa. Słowo public działa tak samo jak przy zmiennych w klasie. Omówimy je dokładniej później, teraz po prostu wiedzmy, że dzięki niemu naszej funkcji będziemy mogli używać **też poza drukarką**.

Później mamy słowo void . Jest to **typ danych** jakie funkcja zwraca. Ale co ten typ oznacza? Jakie dane zwracane przez funkcję? Przecież ustaliliśmy, że nie mamy co zwracać! I własnie to oznacza słowo void  – **funkcja nie zwraca niczego**.

Następnie jest **nazwa**, tak samo dowolna jak przy zmiennych. Dobrze, żeby nazwa odzwierciedlała co nasza funkcja robi. Dzięki temu używając jej będziemy od razu wiedzieć czego się spodziewać. **Czasownik** jest tutaj naturalnym wyborem.

Na końcu została ta część w nawiasach okrągłych czyli **parametry funkcji**. To nic innego jak lista zmiennych, które będziemy wypełniać używając funkcji. Parametry są to nasze **dane wejściowe**. Tak jak w matematyce parametrem funkcji f(x) jest x tak w funkcji powyżej takim parametrem jest SavingsAccount account.

Za pomocą parametrów **przesyłamy do funkcji wartości**, które mogą się zmieniać przy każdym użyciu tej funkcji – u nas to będą **różne konta oszczędnościowe**, jakich dane będziemy chcieli wyświetlić. Dzięki temu możemy wykorzystać taką raz napisaną metodę dla wielu wartości.

### Mechanizm drukujący

Zostało nam jeszcze wypełnienie **funkcji** jakąś treścią, a więc operacjami jakie ma **wykonywać**. To będzie to wszystko co znajduje się pomiędzy **nawiasami klamrowymi**. Dodajmy więc tutaj kod podobny do tego, którym do tej pory drukowaliśmy **dane kont oszczędnościowych**:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Bank

{

class Printer

{

public void Print(SavingsAccount account)

{

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

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

Console.WriteLine("Imię właściciela: {0}", account.FirstName);

Console.WriteLine("Nazwisko właściciela: {0}", account.LastName);

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

Console.WriteLine();

}

}

}

Wszystko co znalazł się **pomiędzy nawiasami klamrowymi** od tej pory będzie **stanowić całość** zamkniętą pod nazwą ***„Print”***.

## Drukujemy

Wróćmy więc do miejsca, od którego wszystko się zaczęło – do funkcji Main  w pliku ***Program.cs***. To tutaj potrzebujemy wykorzystać nasz mechanizm drukujący. Usuń więc linijki odpowiedzialne za wyświetlanie danych kont **savingsAccount** i **secondSavingsAccount**. Spokojnie, za moment znów nasze dane będą mogły pojawić się na ekranie.

Najpierw musimy **wyprodukować** jedną sztukę drukarki, którą zaprojektowaliśmy. Dlatego pod kodem wypełniającym **dane kont oszczędnościowych** dopisz tworzenie drukarki:

Printer printer = new Printer();

Drukarka gotowa. Czas ją wykorzystać. W tym celu pod dodaną przed chwilą linijką użyjmy punkcji Print()  tej naszej drukarki

printer.Print(savingsAccount);

printer.Print(secondSavingsAccount);

Jak widać, do funkcji dostępnych w jakiejś klasie dostajemy się tak samo jak do zmiennych – przez **znak kropki**. Tak więc piszemy **nazwę zmiennej**, w której znajduje się obiekt, potem **kropkę** i na końcu **nazwę funkcji** wraz z **nawiasami okrągłymi**.

To co odróżnia funkcję od zmiennej to właśnie te **nawiasy**, których tutaj używamy. Oznaczają one **wywołanie funkcji**, czyli po prostu jej użycie. Dodatkowo jeśli metoda przyjmuje jakieś **argumenty** to w tym miejscu je przekazujemy. W powyższym przypadku przekazujemy jako argument obiekt konta oszczędnościowego, którego dane chcemy wyświetlić za pomocą naszej funkcji.

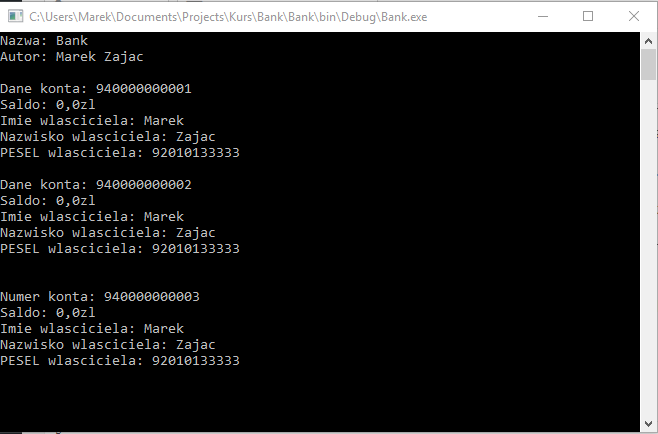
Jak widzisz dzięki **parametrom** możemy używać tej samej metody dla różnych danych.

W tekście pojawiają się dwa słowa – **parametr** i **argument**. Warto więc wyjaśnić czym się różnią.

**Parametrem** nazywamy tą **zmienną**, którą wpisujemy projektując funkcję. To do niej będą wpadały przekazywane wartości i to właśnie z parametrów funkcji korzystamy w kodzie tej funkcji.

**Argumentem** jest zaś przekazywana aktualnie **wartość**, która trafia do naszej funkcji i jest przypisywana do parametru.

Po uruchomieniu programu naszym oczom powinien ukazać się poniższy widok:

[](https://zajacmarek.com/wp-content/uploads/2018/11/functions_1.png)

Zaś ostateczny kod, w którym korzystamy z nowej funkcji wygląda tak:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Bank

{

class Program

{

static void Main(string[] args)

{

string name = "Nazwa: Bank";

string author = "Autor: Marek Zajac";

Console.WriteLine(name);

Console.WriteLine(author);

Console.WriteLine();

SavingsAccount savingsAccount = new SavingsAccount();

savingsAccount.AccountNumber = "940000000001";

savingsAccount.Balance = 0.0M;

savingsAccount.FirstName = "Marek";

savingsAccount.LastName = "Zając";

savingsAccount.Pesel = 92010133333;

SavingsAccount secondSavingsAccount = new SavingsAccount();

secondSavingsAccount.AccountNumber = "940000000002";

secondSavingsAccount.Balance = 0.0M;

secondSavingsAccount.FirstName = "Marek";

secondSavingsAccount.LastName = "Zając";

secondSavingsAccount.Pesel = 92010133333;

Printer printer = new Printer();

printer.Print(savingsAccount);

printer.Print(secondSavingsAccount);

BillingAccount billingAccount = new BillingAccount();

billingAccount.AccountNumber = "940000000003";

billingAccount.Balance = 0.0M;

billingAccount.FirstName = savingsAccount.FirstName;

billingAccount.LastName = savingsAccount.LastName;

billingAccount.Pesel = savingsAccount.Pesel;

Console.WriteLine();

Console.WriteLine("Numer konta: {0}", billingAccount.AccountNumber);

Console.WriteLine("Saldo: {0}zł", billingAccount.Balance);

Console.WriteLine("Imię właściciela: {0}", billingAccount.FirstName);

Console.WriteLine("Nazwisko właściciela: {0}", billingAccount.LastName);

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

Console.ReadKey();

}

}

}

# A co z innym typem konta?

Mam nadzieję, że to pytanie pojawiło się w Twojej głowie zanim jeszcze je tutaj podałem. Faktycznie jest tak, że nie wzięliśmy pod uwagę **konta rozliczeniowego** i nasza funkcja nie jest na nie gotowa. Próba przekazania go do niej i uruchomienia skończy się poniższym **błędem w Visual Studio**:

[](https://zajacmarek.com/wp-content/uploads/2018/11/functions_2.png)

**C#** nie potrafi zamienić jednego **typu konta** na drugie. Skoro funkcja przyjmuje **konta oszczędnościowe** to tylko takie możemy podać.

Ale są na to rozwiązania. Moglibyśmy np. napisać drugą drukarkę, która by obsługiwała inny rodzaj konta. Jednak nikt chyba nie robi drukarki, która potrafi wydrukować litery, ale cyfr to już nie bardzo.

Dlatego innym rozwiązaniem jest zastosowanie **przeciążenia funkcji**.

Brzmi groźnie, jednak jest to nic innego jak napisanie **w jednej klasie funkcji**, która ma taką samą nazwę jak poprzednia ale przyjmuje **innego typu** lub w **innej ilości parametry**. Dzięki przeciążeniu funkcji możliwe będzie posiadanie dwóch funkcji **Print**, które będą obsługiwać różne **typy kont**.

Tak więc jeżeli chcemy mieć funkcję o nazwie **Print**, która obsłuży drukowanie danych konta oszczędnościowego i w tej samej drukarce funkcję, która **również ma nazwę Print** ale obsłuży drukowanie danych konta rozliczeniowego to wystarczy, że takie dwie funkcje napiszemy:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Bank

{

class Printer

{

public void Print(SavingsAccount account)

{

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

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

Console.WriteLine("Imię właściciela: {0}", account.FirstName);

Console.WriteLine("Nazwisko właściciela: {0}", account.LastName);

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

Console.WriteLine();

}

public void Print(BillingAccount account)

{

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

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

Console.WriteLine("Imię właściciela: {0}", account.FirstName);

Console.WriteLine("Nazwisko właściciela: {0}", account.LastName);

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

Console.WriteLine();

}

}

}

I tyle. Załatwione.

Teraz korzystając z naszej drukarki możemy przekazać **oba typy kont** jako argumenty. I oba zostaną poprawnie wydrukowane.

Zróbmy więc to w naszym programie. Teraz klasa Program  wygląda następująco:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Bank

{

class Program

{

static void Main(string[] args)

{

string name = "Nazwa: Bank";

string author = "Autor: Marek Zajac";

Console.WriteLine(name);

Console.WriteLine(author);

Console.WriteLine();

SavingsAccount savingsAccount = new SavingsAccount();

savingsAccount.AccountNumber = "940000000001";

savingsAccount.Balance = 0.0M;

savingsAccount.FirstName = "Marek";

savingsAccount.LastName = "Zając";

savingsAccount.Pesel = 92010133333;

SavingsAccount secondSavingsAccount = new SavingsAccount();

secondSavingsAccount.AccountNumber = "940000000002";

secondSavingsAccount.Balance = 0.0M;

secondSavingsAccount.FirstName = "Marek";

secondSavingsAccount.LastName = "Zając";

secondSavingsAccount.Pesel = 92010133333;

BillingAccount billingAccount = new BillingAccount();

billingAccount.AccountNumber = "940000000003";

billingAccount.Balance = 0.0M;

billingAccount.FirstName = savingsAccount.FirstName;

billingAccount.LastName = savingsAccount.LastName;

billingAccount.Pesel = savingsAccount.Pesel;

Printer printer = new Printer();

printer.Print(savingsAccount);

printer.Print(secondSavingsAccount);

printer.Print(billingAccount);

Console.ReadKey();

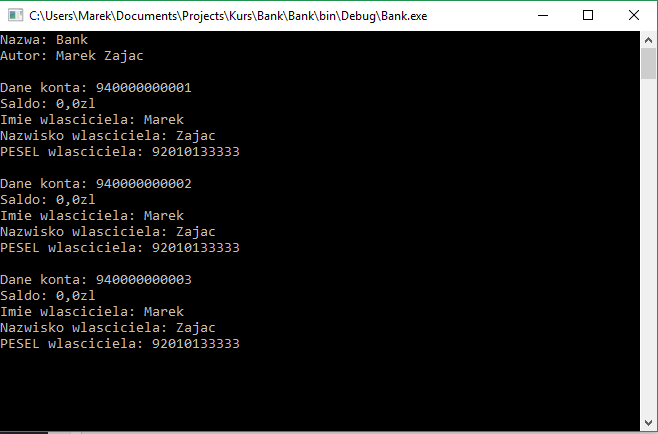
}

}

}

Skoro możemy już drukować dane wszystkich kont to trochę uporządkowałem kod i przeniosłem budowanie drukarki i jej użycie na sam koniec, zaraz przed użyciem funkcji, która oczekuje wciśnięcia klawisza.

Zwróć uwagę, że do funkcji Print()  przekazuję teraz zmienne zarówno z **obiektami kont oszczędnościowych** jak i z **kontem rozliczeniowym**. I po uruchomieniu aplikacji wszystko zadziała prawidłowo:

[](https://zajacmarek.com/wp-content/uploads/2018/11/functions_3.png)

# To może by tak te konta…

… wypełniać danymi poprzez funkcję?

Jak najbardziej jest to możliwe. Jeżeli piszemy **funkcję w klasie**, która posiada jakieś wartości (**zmienne**) to mamy możliwość dobrania się do nich w naszej funkcji. Tak samo jak zegarek posiada funkcję „przesuń wskazówkę” to nie tworzy w tej funkcji nowej wskazówki tylko korzysta z tej, która jest w zegarze.

Do zmiennych w klasie odnosimy się **po samej nazwie**. W końcu nie ma potrzeby mówić o jaki obiekt nam chodzi skoro zarówno funkcja jak i zmienna są w tym samym obiekcie.

Skoro wiemy już, że funkcje mogą mieć dostęp do zmiennych w klasie, wiemy jak używać w funkcji parametrów i jak przekazywać argumenty to czas na napisanie funkcji, która uwolni nas od wypełniania każdego pola klasy z osobna. Zróbmy to dla klasy **SavingsAccount**:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Bank

{

class SavingsAccount

{

public string AccountNumber;

public decimal Balance;

public string FirstName;

public string LastName;

public long Pesel;

public void Init(string accountNumber, decimal balance, string firstName, string lastName, long pesel)

{

AccountNumber = accountNumber;

Balance = balance;

FirstName = firstName;

LastName = lastName;

Pesel = pesel;

}

}

}

Przyjęło się, że nazwy parametrów funkcji rozpoczynamy małą literą. A że **C#** rozróżnia wielkość liter to umożliwia nam to nazwanie tak samo zmiennej w klasie jak i parametru funkcji – będą się one różniły jedynie wielkością pierwszej litery.

Funkcja jest bardzo prosta, a jedyne co robi to przerzuca wartości z jednej zmiennej do drugiej. Takie ładowanie węgla z samochodu do piwnicy.

Użycie takiej metody jest dla Ciebie już proste bo będziemy to robić tak samo jak poprzednio. Tak więc teraz możemy zamienić przypisywanie wartości dla każdej zmiennej z osobna i po prostu użyć **naszej funkcji** dla obiektów klasy **SavingsAccount**:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Bank

{

class Program

{

static void Main(string[] args)

{

string name = "Nazwa: Bank";

string author = "Autor: Marek Zajac";

Console.WriteLine(name);

Console.WriteLine(author);

Console.WriteLine();

SavingsAccount savingsAccount = new SavingsAccount();

savingsAccount.Init("940000000001", 0.0M, "Marek", "Zając", 92010133333);

SavingsAccount secondSavingsAccount = new SavingsAccount();

secondSavingsAccount.Init("940000000002", 0.0M, "Marek", "Zając", 92010133333);

BillingAccount billingAccount = new BillingAccount();

billingAccount.AccountNumber = "940000000003";

billingAccount.Balance = 0.0M;

billingAccount.FirstName = savingsAccount.FirstName;

billingAccount.LastName = savingsAccount.LastName;

billingAccount.Pesel = savingsAccount.Pesel;

Printer printer = new Printer();

printer.Print(savingsAccount);

printer.Print(secondSavingsAccount);

printer.Print(billingAccount);

Console.ReadKey();

}

}

}

Użyliśmy funkcji Init()  dla obu **kont oszczędnościowych**. Kod w metodzie Main()  znowu uległ skróceniu bo kolejny element znalazł się tam gdzie jego miejsce.

# Konstruujemy

Jednak **klasy** w języku **C#** mają pewien specjalny rodzaj metod przygotowany właśnie na tego typu operacje. Chodzi o **konstruktor**.

Pamiętasz jak pierwszy raz tworzyliśmy obiekt z naszej klasy? Powiedziałem wtedy, że te nawiasy, które dajemy po słowie new  i **nazwie klasy** omówimy później. I to jest właśnie ten moment, bo tymi nawiasami było **wywołanie konstruktora**.

**Konstruktor** jest to taki specjalny **rodzaj metody**, przy której nie podajemy zwracanego typu i która ma **taką samą nazwę jak nazwa klasy**. Jest on używany właśnie podczas tworzenia/konstruowania obiektu, co sugeruje już sama nazwa.

Zamieńmy więc metodę Init()  w klasie **SavingsAccount** na **konstruktor**:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Bank

{

class SavingsAccount

{

public string AccountNumber;

public decimal Balance;

public string FirstName;

public string LastName;

public long Pesel;

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

{

AccountNumber = accountNumber;

Balance = balance;

FirstName = firstName;

LastName = lastName;

Pesel = pesel;

}

}

}

Ogólnie wszystko wygląda podobnie. Jedyne różnice to właśnie **brak typu zwracanego**, bo i nic z konstruktora nie możemy zwrócić jak i nazwa, która jest **nazwą klasy**.

Teraz mając konstruktor w klasie **SavingsAccount** użyjmy go w naszym kodzie:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Bank

{

class Program

{

static void Main(string[] args)

{

string name = "Nazwa: Bank";

string author = "Autor: Marek Zajac";

Console.WriteLine(name);

Console.WriteLine(author);

Console.WriteLine();

SavingsAccount savingsAccount = new SavingsAccount("940000000001", 0.0M, "Marek", "Zając", 92010133333);

SavingsAccount secondSavingsAccount = new SavingsAccount("940000000002", 0.0M, "Marek", "Zając", 92010133333);

BillingAccount billingAccount = new BillingAccount();

billingAccount.AccountNumber = "940000000003";

billingAccount.Balance = 0.0M;

billingAccount.FirstName = savingsAccount.FirstName;

billingAccount.LastName = savingsAccount.LastName;

billingAccount.Pesel = savingsAccount.Pesel;

Printer printer = new Printer();

printer.Print(savingsAccount);

printer.Print(secondSavingsAccount);

printer.Print(billingAccount);

Console.ReadKey();

}

}

}

Zrobiliśmy to podczas **tworzenia obiektów** klasy **SavingsAccount**:

SavingsAccount savingsAccount = new SavingsAccount("940000000001", 0.0M, "Marek", "Zając", 92010133333);

Konstruktor, w przeciwieństwie do innych metod można użyć dla obiektu **tylko raz** bo tylko raz obiekt jest tworzony.

Dopóki nie napiszemy własnego konstruktora każda klasa posiada **konstruktor domyślny**. Dzięki temu zawsze możemy utworzyć jej obiekt, nawet jeżeli w klasie nic nie napisaliśmy. Nie podajemy wtedy żadnych argumentów, czyli robimy to tak jak robiliśmy wcześniej. Jednak jeżeli dodajemy pierwszy konstruktor to ten **domyślny przestaje istnieć**. Tak więc w tym momencie musimy przy tworzeniu obiektu podać te wszystkie **wartości**, których nasz konstruktor oczekuje.

Jednak konstruktory, tak jak inne funkcje, **można przeciążać**. Dlatego możemy mieć kilka konstruktorów na różne okazje – zasada jest ta sama, muszą różnić się parametrami. Więc jeżeli potrzebowalibyśmy takiego konstruktora, który nic nie robi, bo np. będziemy chcieli wartości i tak uzupełnić później to możemy dodać drugi, bez parametrów. Wtedy nasza klasa będzie wyglądała w ten sposób:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Bank

{

class SavingsAccount

{

public string AccountNumber;

public decimal Balance;

public string FirstName;

public string LastName;

public long Pesel;

public SavingsAccount()

{

}

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

{

AccountNumber = accountNumber;

Balance = balance;

FirstName = firstName;

LastName = lastName;

Pesel = pesel;

}

}

}

Dopisaliśmy coś w rodzaju konstruktora domyślnego – nie przyjmuje żadnych parametrów i nie wykonuje żadnych operacji. Teraz moglibyśmy znów utworzyć konto bez danych i uzupełniać je później. Jednak taka funkcjonalność jest nam niepotrzebna dlatego usuńmy ten pusty konstruktor.

# Daj mi to

Do tej pory nasze funkcje przyjmowały jakieś **argumenty** i wykonywały jakieś **operacje**. Brakuje jednak tego żeby coś **zwracały**. W końcu już dawno dowiedzieliśmy się, że taką cechę funkcje też posiadają – mogą **zwracać wartości**.

Dodajmy więc możliwość zwrócenia **imienia** i **nazwiska** posiadacza konta jako jednego tekstu.

Zatem w klasie **SavingsAccount** pojawi się nowa funkcja:

public string GetFullName()

{

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

return fullName;

}

Typem, który zwraca jest **string**, a więc jakiś tekst. Nie przyjmuje ona **żadnych parametrów** bo będziemy korzystali z danych znajdujących się już w obiekcie.

Korzystamy tutaj też z funkcji string.Format() , która działa tak samo jak **formatowanie tekstu** z parametrami kiedy wyświetlaliśmy go na ekranie za pomocą Console.WriteLine() . Różnica jest taka, że ta funkcja zamiast tekst wyświetlić zwróci nam go jak nowy **string**. Przy okazji widzisz już jak wygląda używanie funkcji, która **zwraca jakąś wartość** – po prostu robimy to samo co przy dwóch zmiennych, gdzie przypisywaliśmy wartość z jednej do drugiej.

Na końcu metody jest nowe słowo – return  – i za nim nazwa zmiennej z jakiej chcemy zwrócić wartość.

Użycie słowa **return** powoduje zwrócenie **wartości** i **zakończenie funkcji**, dlatego zwykle się je używa na jej końcu, bo wszystko co potem już się nie wykona.

Zamiast zmiennej moglibyśmy z powodzeniem wstawić tam **sam tekst**, który chcielibyśmy zwrócić:

return "My short text";

Albo ominąć tworzenie zmiennej i bezpośrednio zwrócić wartość jaką zwraca nam funkcja string.Format() :

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

Ważne żeby **typ wartości** jaką zwracamy przez **return** był taki sam jak typ, który podaliśmy przy nazwie funkcji.

Cała klasa **SavingsAccount** wygląda teraz w ten sposób:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Bank

{

class SavingsAccount

{

public string AccountNumber;

public decimal Balance;

public string FirstName;

public string LastName;

public long Pesel;

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

{

AccountNumber = accountNumber;

Balance = balance;

FirstName = firstName;

LastName = lastName;

Pesel = pesel;

}

public string GetFullName()

{

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

return fullName;

}

}

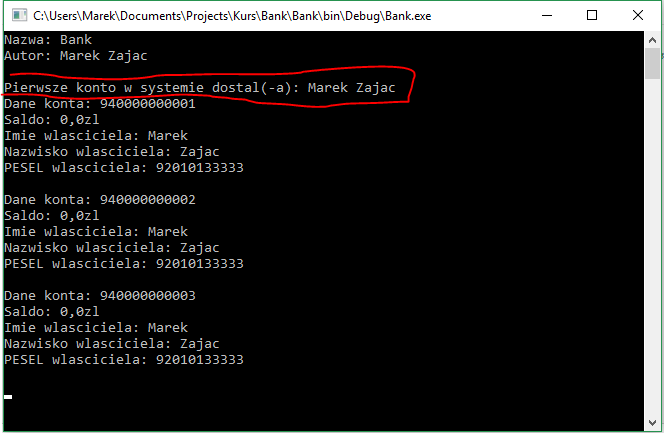
}

Użyjmy więc w funkcji Main() , w klasie **Program**, tej metody, którą właśnie napisaliśmy i wyświetlmy **imię** i **nazwisko** właściciela pierwszego konta:

string fullName = savingsAccount.GetFullName();

Console.WriteLine("Pierwsze konto w systemie dostał(-a): {0}", fullName);

Po uruchomieniu aplikacji powinniśmy na ekranie dostać taki wynik:

[](https://zajacmarek.com/wp-content/uploads/2018/11/functions_4.png)

Na czerwono zaznaczyłem fragment, który przed chwilą dodaliśmy. Jak widzisz wszystko działa i funkcja zwróciła nam poprawne dane.

# Ćwiczenie 1

Dodaj **konstruktor** do klasy **BillingAccount**, który pozwoli uzupełnić w niej **wszystkie dane**. Oraz dodaj taką samą funkcję GetFullName()  jak w klasie **SavingsAccount**.

Następnie zastąp dotychczasowe uzupełnianie danych **konta rozliczeniowego** użyciem dodanego przed chwilą **konstruktora**.

# Ćwiczenie 2

Dodaj w klasach **SavingsAccount** i **BillingAccount** funkcję GetBalance() , która **zwróci tekst** z saldem na koncie z dodaną walutą (złotówki), np. **„0,0zł”**.

# Zrobione

Przebrnęliśmy przez ważną część naszej pracy – poznaliśmy **jak się tworzy i używa funkcji**. Dzięki temu potrafimy dodawać w naszych programach **operacje** i zbiory operacji zamknięte pod wspólną nazwą, które możemy wielokrotnie używać.

Następny krok będzie krótki, ale ważny w kontekście obsługi pieniędzy w banku – nauczymy się **liczyć**!