**[KURS C#] I jeszcze raz!**



**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)!

Poprzednio nauczyliśmy się jak **przechowywać więcej obiektów** w łatwy sposób. Wykorzystaliśmy do tego **listę**.

Jednak wyciąganie każdego elementu za pomocą liczby odpowiadającej jego pozycji nie jest zbyt wygodne. Tym bardziej kiedy chcemy to zrobić np. dla wszystkich obiektów w liście, a nie wiemy ile ich tak naprawdę w niej jest. Dlatego w tym celu nauczymy się wykorzystywać **pętle** czyli mechanizm w języku **C#** pozwalający wykonać pewne operacje **kilka razy** bądź dla **kilku elementów**. Od teraz nie będzie mieć znaczenia jak dużo kont zostało utworzonych w systemie i jak dużo klientów ma bank prowadzony przez Jacka.

**Różne oblicza pętli**

W języku C# możemy wyróżnić **kilka rodzajów pętli**. Ten podział nie jest sztywny i same **pętle** mogą być używane w różnych przypadkach. Jednak każda z nich sprawdza się najlepiej w konkretnych zastosowaniach. Zapoznajmy się więc z nimi i wybierzmy te, które najbardziej nam się przydadzą.

**Od…do**

Pierwszy rodzaj pętli to taka, która zazwyczaj sprawdza się kiedy chcemy w jakiś sposób przejść przez jakiś **zakres liczb**, np. **od 1 do 10** i dla każdej z nich coś wykonać. Tym rodzajem pętli jest pętla for, której budowę przedstawię na przykładzie, który wypisze na ekranie **kolejne liczby od 0 do 20**:

for (int i = 0; i <= 20; i++)

{

Console.WriteLine(i);

}

Wszystko zaczyna się od słowa kluczowego for. Potem w **nawiasie** mamy kolejno **utworzenie zmiennej**, którą użyjemy w **pętli**. Nie musimy tworzyć nowej zmiennej, jednak przypadek pokazany powyżej jest tym najpowszechniejszym. Ustawiamy ją od razu na liczbę, od której będziemy **liczyli**. Następnie **po średniku** mamy **warunek**, który będzie sprawdzamy **po każdym wykonaniu pętli**. Dopóki będzie od zwracał true to **kod w pętli** będzie się wykonywał.

W tym wypadku pętla się będzie wykonywać dopóki wartość przechowywana w zmiennej i będzie **mniejsza lub równa 20**.

Po **kolejnym średniku** mamy zaś kod, który jakoś **zmodyfikuje** naszą zmienną, jest on wykonywany **po uruchomieniu** tego co znajduje się w pętli. W tym wypadku zwiększamy ją za każdym razem **o 1**. Tak więc jeżeli w naszej funkcji Main() w pliku **Program.cs** napiszemy:

namespace Bank

{

class Program

{

static void Main(string[] args)

{

for (int i = 0; i <= 20; i++)

{

Console.WriteLine(i);

}

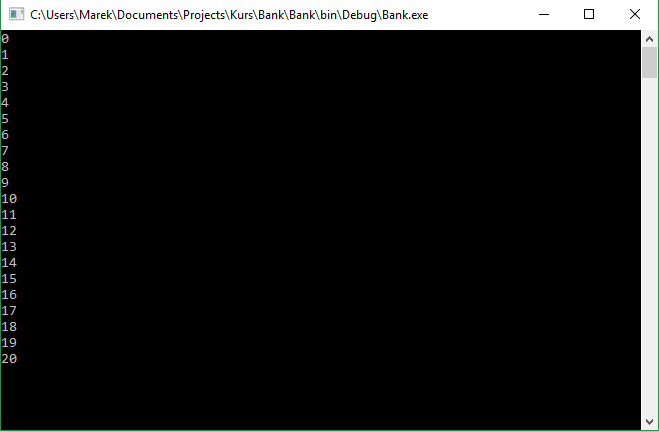
Console.ReadKey();

}

}

}

To na ekranie zobaczymy taki wynik:

[](https://zajacmarek.com/wp-content/uploads/2018/12/loop_1.png)

Czyli dokładnie to czego się spodziewaliśmy. Program wypisuje na ekranie **liczby od 0 do 20**, a więc tak jak powinien.

**Dopóki…**

Kolejny rodzaj pętli to taka, która będzie się wykonywać **dopóki jakiś warunek jest spełniony**. Różni się ona tym, że jej budowa nie sugeruje żadnego konkretnego zakresu w jakim chcemy działać. Po prostu *„dopóki warunek zwraca true to działaj”*. Przedstawicielem tego typu pętli jest pętla while. Jej budowa jest bardzo prosta.

Przedstawię ją na przykładzie podobnym do poprzedniego, a więc wypisującym kolejne liczby. Jednak jak zapewne się domyślisz takie zastosowanie nie jest najczęstszym wykorzystaniem pętli while:

int i = 0;

while (i <= 20)

{

Console.WriteLine(i);

i++;

}

Mamy tutaj słowo kluczowe while  i **warunek**, który będzie sprawdzany **za każdym razem**. Jeżeli warunek od razu nie jest prawdziwy to **pętla while** może się nie wykonać nigdy. Efekt po zmianie poprzedniej pętli na tą powyżej będzie taki sam. Również dostaniemy na ekranie kolejne **liczby od 0 do 20**.

Innym, częstym zastosowaniem tego typu pętli jest np. sytuacja kiedy chcemy wysyłać jakąś wiadomość dopóki ktoś nie odpowie. Wtedy **warunkiem w nawiasie** będzie sprawdzenie czy dostaliśmy odpowiedź, a w pętli zawrzemy samo wysyłanie wiadomości od nas.

**Dla każdego**

Ostatni typ **pętli** jakim się zajmiemy to ten, którego prawdopodobnie będziemy używać bardzo często. Jest to pętla używana do przechodzenia **po elementach wszelkich list**. Tą pętlą jest pętla foreach. Pozwala ona **brać kolejne wartości** wpisane do listy i coś z nimi robić.

Najlepiej będzie przejść od razu do przykładu. Tutaj również pobawmy się liczbami, tym razem dodanymi do listy.

List<int> lista = new List<int>();

lista.Add(2);

lista.Add(45);

lista.Add(11);

foreach (int liczba in lista)

{

Console.WriteLine(liczba);

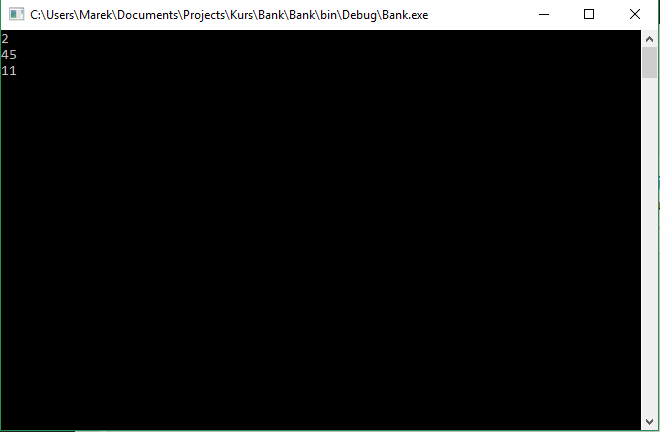
}

Na początku przygotowujemy sobie **listę z jakimiś wartościami**. To już znamy. Potem zaczyna się najważniejsze czyli **pętla foreach**. Znowu wszystko rozpoczyna **słowo kluczowe**.

Potem **w nawiasach** dzieje się to co odróżnia tą pętlę od pozostałych – **pobieranie kolejnych elementów z listy**. Tworzymy tutaj **zmienną pomocniczą**, do której będą wpisywane **kolejne wartości** z listy. I dla każdej z tych wartości możemy **coś zrobić wewnątrz pętli**.

**Pętla foreach** bierze kolejne wartości z listy, **od pierwszego do ostatniego**, i wpisuje je do zmiennej utworzonej w nawiasie za słowem kluczowym foreach. Zmienna ta jest dostępna wewnątrz pętli.

Wynik działania powyższego fragmentu kodu, kiedy użyjemy go tak jak poprzednich prezentuje się następująco:

[](https://zajacmarek.com/wp-content/uploads/2018/12/loop_2.png)

Jak widać program wypisał kolejne wartości znajdujące się na liście, dla której użyliśmy pętli.

**Dla każdego konta**

Zobaczmy jak ten **typ pętli** możemy już teraz wykorzystać w naszym programie. Wróćmy do kodu, który napisaliśmy ostatnio dodają **managera kont**:

namespace Bank

{

class Program

{

static void Main(string[] args)

{

AccountsManager manager = new AccountsManager();

manager.CreateBillingAccount("Marek", "Zajac", 1234567890);

manager.CreateSavingsAccount("Marek", "Zajac", 1234567890);

manager.CreateSavingsAccount("Aaaaa", "Bbbbb", 0987654321);

IList<Account> accounts = (IList<Account>)manager.GetAllAccounts();

Printer printer = new Printer();

printer.Print(accounts[0]);

printer.Print(accounts[2]);

Console.ReadKey();

}

}

}

Mówiłem wtedy, przy okazji tej zamiany **IEnumerable** na **IList**, że już niedługo poznamy inne sposoby dobrania się do elementów listy. I tym innym sposobem jest np. poznana przed momentem **pętla foreach**.

Teraz zamiast wyciągać kolejne konta za pomocą **indeksów** możemy po prostu wszystkie je wyświetlić korzystając ze wspomnianej pętli:

using System;

using System.Collections.Generic;

namespace Bank

{

class Program

{

static void Main(string[] args)

{

AccountsManager manager = new AccountsManager();

manager.CreateBillingAccount("Marek", "Zajac", 1234567890);

manager.CreateSavingsAccount("Marek", "Zajac", 1234567890);

manager.CreateSavingsAccount("Aaaaa", "Bbbbb", 0987654321);

IEnumerable<Account> accounts = manager.GetAllAccounts();

Printer printer = new Printer();

foreach(Account account in accounts)

{

printer.Print(account);

}

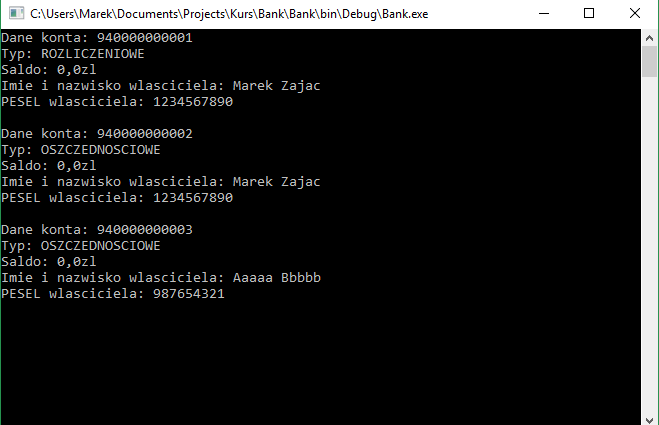
Console.ReadKey();

}

}

}

Wynikiem działania tego programu jest wypisanie danych wszystkich kont na liście:

[](https://zajacmarek.com/wp-content/uploads/2018/12/loop_3.png)

**Nie do końca**

Każda z pętli ma jakiś **zakres w jakim działa**. Nieważne czy jest to zakres liczb czy lista. Ewentualnie **pętla while** może bez problemu działać w nieskończoność. Jednak zdarzają się przypadki kiedy chcemy jednak **przerwać działanie pętli** przed tym jak zrobi to warunek podany dla tej pętli na początku. Przykładowo kiedy szukamy jakiegoś elementu na liście za pomocą **pętli foreach** nie mamy potrzeby szukania dalej po znalezieniu tego czego potrzebujemy. W takim wypadku możemy zastosować **słowo kluczowe** break, które **przerywa działanie najbliższej pętli**.

Poniższy przykład pokazuje jak pętla kończy się po znalezieniu wartości **11** w liście:

using System;

using System.Collections.Generic;

namespace Bank

{

class Program

{

static void Main(string[] args)

{

List<int> lista = new List<int>();

lista.Add(2);

lista.Add(45);

lista.Add(11);

lista.Add(22);

foreach (int liczba in lista)

{

Console.WriteLine(liczba);

if (liczba == 11)

{

break;

}

}

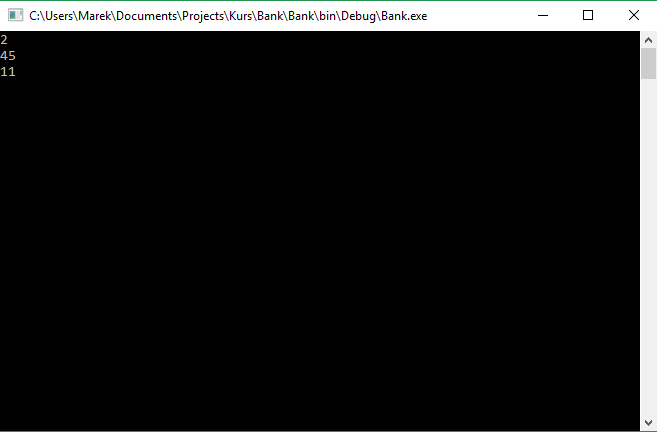
Console.ReadKey();

}

}

}

Jak widzisz jest tutaj dodany **warunek**, który sprawdza czy aktualnie wybrana liczba to **11**. Jeżeli tak jest to **przerywamy wykonywanie pętli**, a więc na ekranie zobaczymy **tylko 3 pierwsze liczby** z listy:

[](https://zajacmarek.com/wp-content/uploads/2018/12/loop_4.png)

**Manager kont na całego**

System bankowy zyskał poprzednio **managera kont**. Jednak nie jest on zbyt rozbudowany. Zawiera jedynie funkcje do **dodawania kont** i **pobierania listy wszystkich**, które są już dodane:

using System.Collections.Generic;

using System.Linq;

namespace Bank

{

class AccountsManager

{

private IList<Account> \_accounts;

public AccountsManager()

{

\_accounts = new List<Account>();

}

public SavingsAccount CreateSavingsAccount(string firstName, string lastName, long pesel)

{

int id = generateId();

SavingsAccount account = new SavingsAccount(id, firstName, lastName, pesel);

\_accounts.Add(account);

return account;

}

public BillingAccount CreateBillingAccount(string firstName, string lastName, long pesel)

{

int id = generateId();

BillingAccount account = new BillingAccount(id, firstName, lastName, pesel);

\_accounts.Add(account);

return account;

}

public IEnumerable<Account> GetAllAccounts()

{

return \_accounts;

}

private int generateId()

{

int id = 1;

if (\_accounts.Any())

{

id = \_accounts.Max(x => x.Id) + 1;

}

return id;

}

}

}

To co chcemy dodać aby uznać, że **manager** robi wszystko co nam potrzebne to kolejno:

* Lista wszystkich kont klienta
* Pobranie jednego konta
* Lista klientów
* Zamknięcie miesiąca
* Wpłata i wypłata pieniędzy z konta

Zróbmy to więc teraz kiedy już poznaliśmy jak poruszać się po liście.

**Wszystkie konta użytkownika**

Częściej niż **pobieranie wszystkich kont** dostępnych w systemie potrzebujemy dostać wszystkie konta, które **należą do jednego klient**a, którego akurat obsługujemy przy okienku.

W takiej sytuacji musimy mieć możliwość wybrania takich kont w systemie. Dzięki temu, że wszystkie rodzaje kont bankowych w naszej aplikacji mają jedną wspólną **klasę bazową**, z której korzystamy w przypadku listy, mamy wszystkie konta w jednym miejscu – na jednej liście. Już to jest ułatwieniem. Jednak trzeba jeszcze **przefiltrować** wszystkie konta i wybrać te, które należą do **klienta z podanymi danymi**. Możemy to zrobić korzystając z **pętli foreach** i dodać je do **pomocniczej listy**, którą zwrócimy:

public IEnumerable<Account> GetAllAccountsFor(string firstName, string lastName, long pesel)

{

List<Account> customerAccounts = new List<Account>();

foreach(Account account in \_accounts)

{

if (account.FirstName == firstName && account.LastName == lastName && account.Pesel == pesel)

{

customerAccounts.Add(account);

}

}

return customerAccounts;

}

**Dla każdego konta** w liście sprawdzamy czy zapisane w nim **dane właściciela** pasują do tych podanych jako **parametry metody**. Jeżeli tak to takie konto dodajemy do listy wybranych kont. Na koniec całość zwracamy.

Jednak jest to dosyć dużo pisania jak na tak prostą czynność, prawda? Na szczęście możemy skorzystać z **Linq**, o którym wspomniałem poprzednio. Mamy w tej bibliotece dostępne również **funkcje filtrujące** dane na liście. Nam przyda się tutaj funkcja Where(), która wybiera te obiekty na liście, dla których funkcja podana jako parametr zwróci **wartość** true:

public IEnumerable<Account> GetAllAccountsFor(string firstName, string lastName, long pesel)

{

return \_accounts.Where(x => x.FirstName == firstName && x.LastName == lastName && x.Pesel == pesel);

}

Powyższy kod ostatecznie daje taki sam rezultat jak poprzednio – dostajemy listę wszystkich kont spełniających warunek, w tym wypadku **należących do podanego klienta**.

A to jeszcze nie koniec możliwości **biblioteki Linq**. Dlatego jest ona często uznawana wręcz za element języka **C#**, bo bez niej operacje na listach były by dużo cięższe i wymagały mnóstwa powtarzającego się kodu.

**Pobieranie jednego konta**

Kolejna **funkcja**, która na pewno przyda się w naszej **aplikacji bankowej** to możliwość wyciągnięcia z listy kont tego jednego, które nas interesuje. Skoro konta mają swój **unikatowy numer** to oczywiście ten fakt wykorzystamy.

Moglibyśmy w tym miejscu korzystać z **pętli foreach** i w momencie kiedy **podany numer** będzie pasował do aktualnie sprawdzanego konta to schowamy sobie to konto do zmiennej, przerwiemy pętlę i zwrócimy **zapamiętane konto**:

public Account GetAccount(string accountNo)

{

Account account;

foreach (Account acc in \_accounts)

{

if (acc.AccountNumber == accountNo)

{

account = acc;

break;

}

}

return account;

}

Jednak po raz kolejny skorzystamy z bardzo pomocnej **biblioteki Linq**. Tym razem możemy wykorzystać funkcję Single(), która zgodnie z nazwą zwróci **dokładnie jeden element** z listy. Więc lepiej niech obsługa banku się nie pomyli bo funkcja będzie bezwzględna i zgłosi błąd kiedy nie znajdzie niczego albo znajdzie więcej kont z podanym numerem.

Użycie jej jest również bardzo proste i sprowadza się do przekazania **delegaty**, która sprawdzi czy **numery konta** się zgadzają:

public Account GetAccount(string accountNo)

{

return \_accounts.Single(x => x.AccountNumber == accountNo);

}

**Lista klientów**

Lista klientów to coś co pozwoli nam sprawdzić ilu klientów ma nasz bank i wyszukać ich danych, w razie jakbyśmy potrzebowali wyciągnąć jakieś konto.

W tym miejscu od razu przedstawię rozwiązanie za pomocą **Linq**. Wykorzystamy tym razem dwie metody: Select() i Distinct().

Pierwsza z nich pozwala wybrać **konkretną wartość** z obiektu, który jest w liście albo np. **utworzyć i zwrócić** inny obiekt na podstawie danych z obiektu z listy. Więc od razu nasuwa się myśl, że dzięki temu będziemy mogli wyciągnąć **imię**, **nazwisko** i **PESEL** właściciela każdego konta i zwrócić je jako **string**, nie zwracając całego konta.

Zadaniem drugiej metody jest **usunięcie wszystkich powtórzeń**. Bo jeżeli klient założył kilka kont to po wykonaniu Select() jego nazwisko pojawi się kilka razy na liście klientów. Dzięki funkcji Distinct() wszystkie te powtórzenia usuniemy i dostaniemy po jednym elemencie dla każdego klienta jaki założył konto u Jacka.

public IEnumerable<string> ListOfCustomers()

{

return \_accounts.Select(a => string.Format("Imię: {0} | Nazwisko: {1} | PESEL: {2}", a.FirstName, a.LastName, a.Pesel)).Distinct();

}

Kod może wyglądać na skomplikowany, ale tak naprawdę wszystko w nim jest proste.

Bierzemy listę kont. Używamy funkcji Select(). Jako jej parametr przekazujemy **funkcję**, która zwróci jakąś wartość na podstawie każdego obiektu z listy kont. W tym przypadku ta funkcja zwraca nowy **string** sformatowany tak żeby zawierał **imię**, **nazwisko** i **PESEL** właściciela oddzielone od siebie znakiem **|**. Jest to możliwe gdyż funkcja przekazana w parametrze dostanie po kolei każdy obiekt jaki mamy w liście.

Ponieważ funkcja Select() zwraca w tym wypadku typ **IEnumerable<string>** to możemy zastosować od razu kolejną metodę, która usunie wszystkie powtórzenia, czyli Distinct(). Ta metoda też zwraca typ **IEnumerable<string>** i taki właśnie typ zwraca cała nasza funkcja do wyciągania listy wszystkich klientów banku.

Przykładowe użycie tej funkcji wygląda tak:

using System;

using System.Collections.Generic;

namespace Bank

{

class Program

{

static void Main(string[] args)

{

AccountsManager manager = new AccountsManager();

manager.CreateBillingAccount("Marek", "Zajac", 1234567890);

manager.CreateSavingsAccount("Marek", "Zajac", 1234567890);

manager.CreateSavingsAccount("Aaaaa", "Bbbbb", 0987654321);

IEnumerable<string> users = manager.ListOfCustomers();

foreach (string user in users)

{

Console.WriteLine(user);

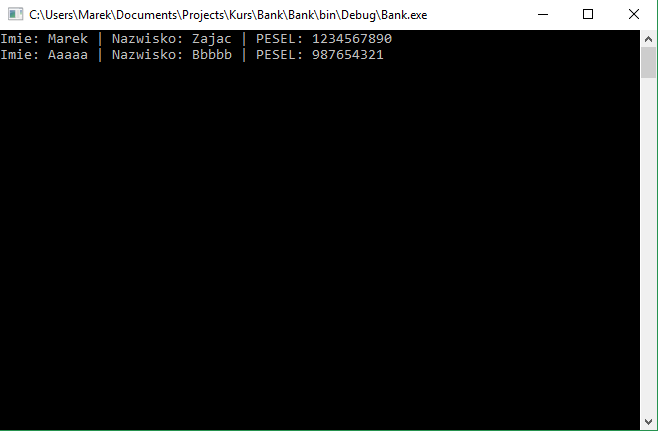
}

Console.ReadKey();

}

}

A jej rezultat prezentuje się następująco:

[](https://zajacmarek.com/wp-content/uploads/2018/12/loop_5.png)

Zwróć uwagę, że na liście mieliśmy **trzy konta**, a funkcja zwróciła **dwa wpisy**. To właśnie **efekt działania** metody Distinct(), która usunęła powtarzający się wpis z klientem pierwszych dwóch kont.

**Ćwiczenie**

Spróbuj **napisać funkcję**, która będzie działała w ten sam sposób jak pokazana powyżej, tzn. zwracała listę klientów. Jednak nie korzystając w tym celu z funkcji z biblioteki **Linq**. Być może zauważysz, że ten sam efekt wymagać będzie dużo większej ilości kodu.

**Zamknięcie miesiąca**

Jeżeli wrócimy pamięcią do **pierwszych etapów pracy** nad projektem to przypomnimy sobie o tym, że Jacek chciałby móc **zamykać miesiąc**. Miałoby to skutkować dodaniem **odsetek** na kontach oszczędnościowych i pobraniem **opłaty** za prowadzenie konta na kontach rozliczeniowych.

W tym celu po pierwsze każdy z typów konta musi zyskać odpowiednią metodę. **Konto rozliczeniowe** dostanie metodę do **pobierania opłaty**, a **konto oszczędnościowe** do **naliczania odsetek**.

Metody te prezentują się następująco i jestem pewien, że są dla Ciebie już oczywiste bo zawierają tylko to o czym się uczyliśmy wiele razy:

namespace Bank

{

class SavingsAccount : Account

{

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

: base(id, firstName, lastName, pesel)

{

}

public void AddInterest(decimal interest)

{

Balance += Balance \* interest;

}

public override string TypeName()

{

return "OSZCZĘDNOŚCIOWE";

}

}

}

namespace Bank

{

class BillingAccount : Account

{

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

: base(id, firstName, lastName, pesel)

{

}

public void TakeCharge(decimal value)

{

Balance -= value;

}

public override string TypeName()

{

return "ROZLICZENIOWE";

}

}

}

Mamy więc dostępne metody AddInterest()  i TakeCharge(), które odpowiednio dodają **odsetki** i pobierają **opłatę**. Teraz chcemy ich użyć **dla wszystkich kont**.

Od razu nasuwa Ci się myśl o użyciu **pętli foreach**. I bardo dobrze, bo ją wykorzystamy.

Jednak jest pewien problem. Metody dodaliśmy osobno dla każdego typu konta, a lista trzyma je jako **obiekty klasy bazowej Account**, która nie posiada ani metody AddInterest(), ani TakeCharge(). Jak sobie z tym poradzić?

Tutaj z pomocą przyjdzie znowu **Linq** oraz dodatkowo **operator** is, który pozwala sprawdzić czy jakiś obiekt **jest podanego przez nas typu**. Dzięki is  możemy sprawdzić czy konto kryjące się pod obiektem klasy bazowej **Account** jest tak naprawdę typu **BillingAccount** czy **SavingsAccount**. Użycie **Linq** i **is** jest bardzo proste, a cała **funkcja zamykania miesiąca** prezentuje się następująco:

public void CloseMonth()

{

foreach(SavingsAccount account in \_accounts.Where(x => x is SavingsAccount))

{

account.AddInterest(0.04M);

}

foreach(BillingAccount account in \_accounts.Where(x => x is BillingAccount))

{

account.TakeCharge(5.0M);

}

}

Mamy tutaj **dwie pętle**. Dla pierwszej wybieramy tylko **konta oszczędnościowe**, dzięki temu możemy na nich użyć metody AddInterest() z podanym w parametrze oprocentowaniem wynoszącym **4%**. W drugiej zaś korzystamy tylko z kont, których rzeczywisty typ to **BillingAccount**. Co pozwala na użycie metody TakeCharge() pobierającej opłatę w wysokości **5z**ł.

**Wpłata i wypłata pieniędzy**

Chcąc jakoś **gromadzić oszczędności** przydałoby się żeby klient mógł jakoś **wpłacać**, a potem **wypłacać** pieniądze z konta. Posłuży nam do tego metoda ChangeBalance() w klasie **Account**:

public void ChangeBalance(decimal value)

{

Balance += value;

}

Jednak sama w sobie jest nudna. Dodaje podaną jako parametr ilość pieniędzy do **salda konta**. My chcemy aby nasz **manager** pozwalał wykonać tą operację podając **numer konta** i **kwotę**. Osobno dla **wpłaty** i **wypłaty** konta.

Przez to, że wcześniej napisaliśmy **funkcję do pobierania konta** po jego **numerze** to teraz takie metody są dla nas czymś trywialnym. Każda z nich to ledwo dwie linijki:

public void AddMoney(string accountNo, decimal value)

{

Account account = GetAccount(accountNo);

account.ChangeBalance(value);

}

public void TakeMoney(string accountNo, decimal value)

{

Account account = GetAccount(accountNo);

account.ChangeBalance(-value);

}

Od teraz możemy dodawać do konta wpłaconą przez klienta gotówkę. Albo umożliwić mu jej wypłatę. Wystarczy, że poda **numer konta** i **kwotę** jaka ma zostać wypłacona bądź wpłacona.

**Zapętliliśmy się**

W końcu nasz **program bankowy** zawiera praktycznie wszystko czego oczekiwał od niego Jacek. Całe zaplecze jest gotowe na przyjmowanie klientów.

Jednak pozostało coś jeszcze – **punkt obsługi tych klientów** czyli jakiś sposób na komunikację z użytkownikiem. Tym jednak zajmiemy się już za moment w następnej części.