**[KURS C#] Lista wartości**



**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)!

Projekt, który rozwijamy zawiera już coraz więcej możliwości. A przede wszystkim pozwala tworzyć **obiekty kont oszczędnościowych** i **rozliczeniowych**. Jednak największy problem jaki istnieje w naszej aplikacji to **ilość kont** jakie obsługujemy.

W tym momencie każde konto wymaga swojej **osobnej zmiennej**, która będzie je przechowywała. Chcąc utworzyć kolejne konto musimy dodać w kodzie nową zmienną. A przecież bank tak nie działa! Jacek na pewno nie chciałby musieć przy każdym nowym koncie zgłaszać się do nas abyśmy dopisali kawałek kodu, który przechowa **kolejne konto**. Przydałby nam się inny sposób na przechowywanie większej ilości kont. **Jakaś lista**…

**Chcemy więcej!**

Coś takiego jak **listy**, czy w szerszym ujęciu **kolekcje** jak najbardziej są dostępne w języku **C#**. Jest to nieodzowny element wszystkich aplikacji, które korzystają z jakichś **danych**. W końcu mało kiedy mamy tylko kilka wartości. Zwykle projekty komercyjne mają do dyspozycji jakieś **zbiory kilku typów danych**, z którymi muszą coś zrobić.

I dokładnie tak jest również w sytuacji naszej aplikacji dla banku kolegi. Główna funkcjonalność tego projektu to operacje na **pojedynczych kontach** i **liście kont**. Dlatego na tym etapie w końcu będziemy mogli dodać możliwość przechowywania **dowolnej ilości kont** bez dodawania osobnej zmiennej dla każdego z nich.

**Ogólnie mówiąc**

Zanim jednak przejdziemy do samych list potrzebne nam będzie jeszcze jedno zagadnienie. Chodzi o **generyczność klas** w języku **C#**. Temat poruszę tutaj tylko na takim poziomie abyśmy mogli rozmawiać o listach, które nas interesują.

Kiedy producent wprowadza do sprzedaży jakieś pudełko to może to może ono być **wyspecjalizowane** pod konkretny typ przedmiotu. Jednak częściej pudełko jest **uniwersalne**. To od nas zależy jaki **typ przedmiotów** będziemy w nim przechowywać. I właśnie z taką uniwersalnością możemy kojarzyć **generyczność**.

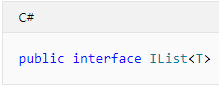
Samo pudełko ma swoje parametry, funkcje takie jak dodatkowe zamknięcie itd. Jednak to my możemy powiedzieć, że jest to pudełko np. na zabawki – podpisując je.

I mimo, że same cechy pudełka się nie zmieniły to od tej pory zawierać będzie jedynie zabawki. W podobny sposób mamy możliwość stworzenia klas, którym obojętne jakim typem będą się w pewnych ich częściach posługiwać. Są to **klasy generyczne**.

Można je poznać po tym, że kiedy się je opisuje to **po nazwie klasy** pojawiają się nawiasy **trójkątne z literą T**:

Klasa<T>

Ta litera **T** to po prostu miejsce, w którym podamy wybrany przez nas **typ**. Tak samo sprawa wygląda z interfejsami. Tzn, możemy mieć **interfejsy generyczne**, w przypadku których również obecne są **nawiasy trójkątne** z literą **T**. Np. w dokumentacji na stronie Microsoftu **interfejs listy** wygląda tak:

[](https://zajacmarek.com/wp-content/uploads/2018/12/list_1.png)

Idealnym przykładem **klas** i **interfejsów generycznych** są właśnie listy. W końcu każda lista sama w sobie wygląda i zachowuje się tak samo – można do niej coś dodać, usunąć itd. Jedyna różnica to **typ jaki będzie przechowywać**. Tak jak wspomniane wcześniej pudełko, któremu dopiero po kupnie **nadajemy przeznaczenie**.

Obiekty takich klas tworzymy jak zwykłe klasy, jednie uzupełniając całość o te dodatkowe nawiasy trójkątne i podając w nich nazwę typu jakiego chcemy użyć w tej klasie. Przykładowo klasa generyczna, której obiekt specjalizujemy w obsłudze typu **int** wyglądać będzie następująco:

Klasa<int> zmienna = new Klasa<int>();

**To samo ale inne**

**Pełny typ** takiego obiektu zawiera również to co znajduje się **w nawiasach**. Dlatego nie możemy wartości przypisanej do zmiennej powyżej przenieść do zmiennej, która co prawda też jest typu **Klasa** jednak posiada **inny typ w nawiasach**. Poniższy kod jest niepoprawny i spowoduje zwrócenie błędu:

Klasa<int> zmienna = new Klasa<int>();

Klasa<double> innaZmienna = zmienna; // TUTAJ POJAWI SIĘ BŁĄD

Tego typu klasy używa się w sytuacji kiedy samo zachowanie klasy nie jest bezpośrednio **związane z konkretnym typem** i może być ona używana tak samo dla jednego jak i drugiego typu.

Powyższy przykład można porównać do utworzenia dwóch klas dla obu tych typów, które różnią się właśnie tylko tym typem zmiennej, pozostała zawartość pozostaje taka sama:

class KlasaInt

{

public int Value {get; set;}

// ...

}

class KlasaDouble

public double Value {get;set;}

// ... taka sama zawartość jak w klasie KlasaInt

**Zbierzmy je razem**

Powyżej został już pokazany **interfejs list** – IList. Sama **klasa listy** to po prostu List. I to z niej będziemy korzystać.

Jednak jest jeszcze jeden interfejs, którym będziemy się często posługiwać, i z którego **interfejs IList dziedziczy**. Mianowicie chodzi o IEnumerable. Zawiera on jedynie metody do pobierania wartości z listy, przechodzenie po kolejnych jej elementach.

Samo **IList** posiada dodatkowo metody do **dodawania**, **usuwania** czy **sortowania** elementów ale w przypadku kiedy po prostu chcemy dostać listę wartości, z której możemy coś odczytać to najlepiej będzie użyć bardziej uniwersalnego interfejsu **IEnumerable** zamiast związanego z konkretnym rodzajem kolekcji interfejsu **IList**.

Lista to jest jeden z rodzajów kolekcji. Jedną z jego cech jest to, że elementy w niej są w jakiś sposób ułożone jeden po drugim. Istnieją też typy kolekcji gdzie po prostu mamy zbiór elementów, ale nie możemy powiedzieć „pierwszy element”, czy „n-ty element”.

Wracając do samego **przechowywania** jakichś wartości w większych grupach to sprawa jest dosyć prosta. Najpierw musimy **utworzyć jakąś listę**. W sensie obiekt tej listy. Tak samo jak ktoś musi wyprodukować pudełko, które potem użyjemy:

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

Od teraz mamy już **pojemniczek** na **liczby** typu **int**.

Możemy do niego dodawać odpowiednie wartości. Bez dużego zaskoczenia, robimy to przez funkcję Add(). Tak więc chcąc dodać kilka wartości do naszej listy zrobimy to w ten sposób:

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

lista.Add(1);

lista.Add(54);

lista.Add(128);

Dobra, mamy już wartości wrzucone do pudełka, znaczy do listy. To jak je wyciągnąć? Na to również jest sposób. Możemy to zrobić przez operator [], podając pomiędzy tymi nawiasami **liczbę porządkową element**u, który chcemy wyciągnąć. Numeracja zaczyna się od **0**, dlatego **pierwszy element** to tak naprawdę **zerowy element**. Tak więc chcąc wyciągnąć kolejne elementy zrobimy to w ten sposób:

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

lista.Add(1);

lista.Add(54);

lista.Add(128);

int pierwszy = lista[0];

int druga = lista[1];

int trzeba = lista[2];

**Więcej niż jedno konto**

Przerzućmy więc ten temat na pole naszych kont. Od teraz dodajmy nasze konta do listy zamiast do pojedynczych zmiennych. Wymieńmy więc zawartość metody Main() w pliku **Program.cs** na poniższe:

using System;

using System.Collections.Generic;

namespace Bank

{

class Program

{

static void Main(string[] args)

{

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

accounts.Add(new SavingsAccount(1, "Marek", "Zajac", 1234567890));

accounts.Add(new SavingsAccount(2, "Marek", "Aaaa", 1234554321));

accounts.Add(new SavingsAccount(3, "Zzzz", "Bbb", 0987654321));

accounts.Add(new BillingAccount(4, "Marek", "Zajac", 1234567890));

accounts.Add(new BillingAccount(5, "Marek", "Aaaa", 1234554321));

Printer printer = new Printer();

printer.Print(accounts[1]);

printer.Print(accounts[4]);

Console.ReadKey();

}

}

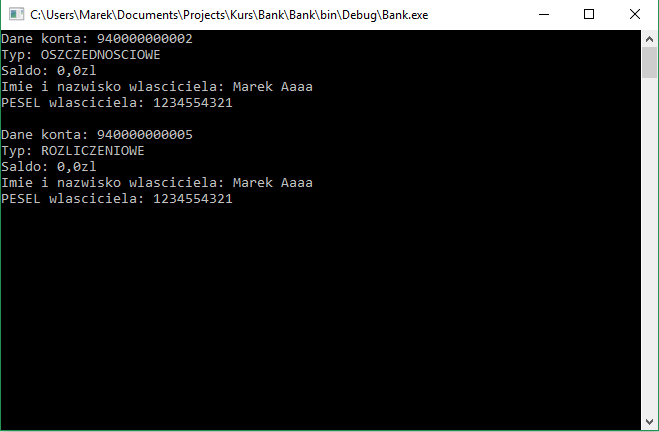
}

Utworzyliśmy tutaj **listę przechowującą konta**. Co więcej, korzystamy tutaj z tego, że oba typy kont **dziedziczą** po klasie **Account**. Dzięki temu możemy mieć jedną listę dla obu typów kont! I korzystamy z tego dodając do listy obiekty zarówno konta oszczędnościowego jak i rozliczeniowego.

Dodajemy je tutaj **tworząc nowe konta** bezpośrednio w metodzie Add(). Moglibyśmy tutaj też wykorzystać zmienne pomocnicze i to je przekazywać do tej metody.

Na koniec tworzymy obiekt drukarki, który wykorzystujemy do wydrukowania danych **konta drugiego** i **piątego**.

Jeżeli uruchomimy taki program to dostaniemy poniższy wynik:

[](https://zajacmarek.com/wp-content/uploads/2018/12/list_2.png)

W ten oto sposób wykorzystaliśmy listę w naszym programie. Dla formalności poniżej również przypisanie **listy** do wspomnianych wcześniej **interfejsów**:

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

IEnumerable<int> lista2 = new List<int>();

**Manager kont**

Skoro nasz bank w końcu dostał łatwy sposób na przechowywanie **dużej ilości kont** to warto zatrudnić **managera**, który będzie tym zarządzał!

Napiszmy więc klasę, która będzie zawierała **listę kont** i pośredniczyła w ich tworzeniu. Dzięki temu będziemy mogli w elegancki sposób dodać np. nadawanie kolejnych ID nowym kontom. W jaki sposób? Przekonajmy się!

Zacznijmy od utworzenia klasy **AccountsManager**. Dodajmy też od razu **prywatną listę kont** w nim:

using System.Collections.Generic;

using System.Linq;

namespace Bank

{

class AccountsManager

{

private IList<Account> \_accounts;

public AccountsManager()

{

\_accounts = new List<Account>();

}

}

}

Sam **obiekt listy** tworzymy w **konstruktorze**. Nowa lista będzie więc produkowana przy tworzeniu **obiektu managera**.

Od razu dodajmy metodę **zwracającą wszystkie dodane konta**. Przyda się nam ona potem:

using System.Collections.Generic;

using System.Linq;

namespace Bank

{

class AccountsManager

{

private IList<Account> \_accounts;

public AccountsManager()

{

\_accounts = new List<Account>();

}

public IEnumerable<Account> GetAllAccounts()

{

return \_accounts;

}

}

}

Jak widać skorzystaliśmy tutaj z interfejsu **IEnumerable**, ponieważ i tak jakiekolwiek operacje na liście powinniśmy od teraz wykonywać poprzez nasz **manager**. Dlatego jedyne do czego potrzebujemy listę kont to wyciągnięcie ich w celu np. wyświetlenia.

**Profesjonalne zakładanie kont**

Przyszła pora na bardziej zaawansowane operacje związane z naszymi kontami – dodawanie ich wraz z **automatycznym nadawaniem ID**.

Na początku zajmijmy się tą drugą częścią czyli **generowaniem ID**. Chociaż samo generowanie to mocne słowo w stosunku do tego co będziemy robić. ID dla nowego konta powinno być po prostu zawsze **o 1 większe** od ostatniego jakie jest na liście. Najpierw pokażę **prywatną metodę**służącą do tego, a potem omówimy temat:

private int generateId()

{

int id = 1;

if (\_accounts.Any())

{

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

}

return id;

}

Metodę tę oczywiście dodajemy w klasie **AccountsManager**. Początek jest prosty – tworzymy **zmienną liczbową**, do której przypisujemy **najniższe możliwe** w naszym systemie **ID** czyli **1**. Więcej się dzieje dalej.

**Wyciągniemy wszystko**

Najpierw to co jest w **warunku** w instrukcji if. Pojawiła się tajemnicza metoda Any(). Należy ona do biblioteki **Linq**, która jest domyślnie dostępna w naszym kodzie, wystarczy, że na początku pliku dodaliśmy using System.Linq;.

**Linq** jest czymś co w połączeniu z listami będziemy używali bardzo często. Jest to pewien sposób na **operowanie na listach**. Jeżeli ktoś zna **SQLa** to może tutaj znaleźć analogię – **Linq** pozwala na składanie **zapytań dla listy**. Sama metoda Any() bez żadnych parametrów zwraca informację czy w liście istnieje jakakolwiek wartość. Inaczej mówiąc w ten sposób możemy sprawdzić czy **lista nie jest pusta**. Jeżeli jest to zgodnie z warunkiem jaki dodaliśmy, następną zwróconą wartością **ID** będzie **1**. Jednak jeżeli w liście już jakieś **konto istnieje**…

… to musimy jakoś **wyciągnąć najwyższe ID** jakie w tej liście mamy dodane. Do tego wykorzystamy kolejną metodę z biblioteki Linq – metodę Max().

Zwraca ona maksymalną wartość liczbową na liście. Jednak tutaj pojawia się parametr, który wygląda tajemniczo. Jest to **wyrażenie lambda** (można o nim poczytać np. [na stronie Microsoftu](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions)). Pozwala ono w prosty sposób przekazać jako parametr **funkcję**. W tym wypadku przekazujemy funkcję, która dla każdego konta na liście zwróci jego ID. Metoda Max() będzie wykonywała tą **przekazaną funkcję** dla każdego elementu na liście i dopiero na zwróconych wartościach będzie operować. Wyrażenie przekazane w parametrze wyglądałoby tak jak poniżej jeżeli byśmy chcieli je zapisać jako **osobną funkcję**:

int Func(Account x)

{

return x.Id;

}

Kiedy dostaniemy wartości ID wszystkich kont to funkcja Max() może na ich podstawie znaleźć **wartość maksymalną**, czyli najwyższe ID jakie dodaliśmy do listy. Po tym wystarczy tylko zwiększyć je o **1** aby dostać kolejną **wolną wartość**. Na koniec zostało zwrócenie **„wygenerowanego” ID**.

**Linq** jest rozbudowaną biblioteką i jedną z ciekawszych rzeczy dostępnych w języku C#. To o czym tutaj mówimy jest jedynie wspomnieniem tematu i podstawami. Krótkie omówienie zamieściłem również jakiś czas temu w [jednym z postów na blogu](https://zajacmarek.com/2014/09/c-tips-linq-kolekcje/).

**Załóż konto**

Przyszła pora na dodanie funkcji pozwalających utworzyć i dodać do listy odpowiednio **konto oszczędnościowe** i **rozliczeniowe**. Dodatkowo zwracajmy te utworzone konta. Bez przedłużania dodajmy w klasie **AccountsManager** dwie funkcje tworzące konta na podstawie podanych danych:

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;

}

„Generujemy” sobie **nowe ID**. Potem tworzymy obiekt **nowego konta** przekazując to **ID** i argumenty funkcji z **imieniem** i **nazwiskiem** właściciela i jego numerem **PESEL**. Potem **dodajemy to konto do listy**. Na sam koniec zwracamy je również, dzięki czemu będziemy mogli np. bardzo łatwo wyświetlić wszystkie dane nowo utworzonego konta.

**Niech pracuje**

Wykorzystajmy więc naszego **managera kont** i użyjmy go w metodzie Main() :

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);

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

Printer printer = new Printer();

printer.Print(accounts[0]);

printer.Print(accounts[2]);

Console.ReadKey();

}

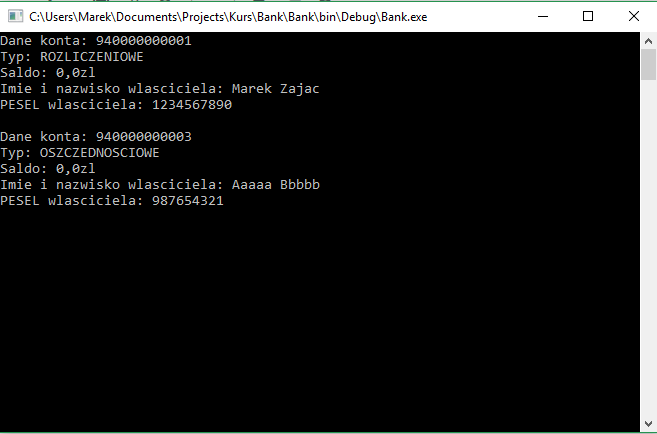
}

}

**Tworzymy obiekt managera**. Dodajemy do niego kilka kont. Następnie bierzemy wszystkie i wypisujemy dane dwóch z nich. Musieliśmy jednak zamienić typ zwracanej listy z **IEnumerable** na **IList** ponieważ **IEnumerable** nie posiada **operatora []**. Jednak mogliśmy to tutaj spokojnie zrobić bo wiemy, że pod spodem tak naprawdę jest **lista**.

Jednak nie jest to zbyt elegancki sposób, ale już w następnej części się go pozbędziemy bo będziemy inaczej korzystać z wartości w liście dlatego sobie na niego pozwoliłem.

W każdym razie uruchamiając teraz program dostaniemy taki wynik:

[](https://zajacmarek.com/wp-content/uploads/2018/12/list_3.png)

Więc jak widać **manager** rozpoczął swoją pracę, a my mamy teraz przyjemny w użyciu sposób na **dodawanie nowych kont** w programie. Warto zwrócić uwagę na **numery kont** – zawierają one ID konta, a więc widać tutaj. że to **ID** nadawane jest poprawnie – wyświetlamy pierwsze i trzecie konto i właśnie takie ID znajdują się w ich numerach.

Cała klasa **AccountsManager** prezentuje się następująco:

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;

}

}

}

**Podsumowując**

W końcu Jacek nie będzie musiał prosić nas o dodanie **nowej zmienne**j w kodzie za każdym razem kiedy ktoś będzie chciał **założyć nowe konto**. Dodatkowo zamknęliśmy tę operację w osobnej klasie przez co wystarczy, że przekażemy dane właściciela do odpowiedniej metody i konto zostanie utworzone, a nawet będzie posiadać poprawne, coraz większe ID.

W następnej części dowiemy się jak możemy się **poruszać po liście**, a więc aplikacja bankowa stanie się jeszcze potężniejsza. Projekt zbliża się powoli do końca.