**[KURS C#] Gotowy system**



**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)!

Po długiej przeprawie przez **podstawy języka C#** doprawione elementami **projektowania struktury kodu** czas na podsumowanie i przejrzenie tego co stworzyliśmy!

**On żyje!**

Na początek prezentacja tego jak **aplikacja** działa u mnie i jak, mam nadzieję, działa i wygląda ona u Ciebie:

Myślę, że jak na **pierwsze zlecenie**, które robiliśmy efekt jest świetny! Z rzeczy, które zrobiliśmy to:

* wyświetlanie menu
* pobieranie informacji od użytkownika
* tworzenie nowych obiektów – konta oszczędnościowego i konta rozliczeniowego
* przechowywanie obiektów na liście
* wyświetlanie danych z obiektów, które dynamicznie dodawaliśmy
* modyfikacja wartości w już utworzonych obiektach
* wyświetlanie wybranych danych z listy – lista klientów

Jest tego całkiem sporo. Mimo, że sam kurs nie był szczególnie długi i na pewno nie sprawił Ci zbyt wielu trudności.

**Czego się nauczyliśmy?**

W trakcie trwania tego kursu udało nam się przygotować działający system dla nowo otwartego banku naszego kolegi Jacka. Ale co ważniejsze nauczyliśmy się **podstaw języka C#**, które są dobrym wstępem do wejścia w świat **komercyjnego programowania**. Zagadnienia jaki poruszyliśmy to:

* tworzenie nowego projektu w Visual Studio
* struktura programu konsolowego
* dodawanie własnych klas, a w konsekwencji modelowanie rzeczy i zachowań wewnątrz naszych programów
* posługiwanie się zmiennymi, które pozwalają przechowywać obiekty, na których pracujemy
* pisanie własnych funkcji w celu zamykania fragmentów kodu jako osobnych bloków, które możemy wykorzystać wielokrotnie
* dziedziczenie klas – klasy w końcu mogą mieć swoich rodziców
* interfejsy będące swego rodzaju kontraktem, który powoduje, że oczekujemy dostępu do konkretnych zachowań, ale nie wymagamy żadnej konkretnej implementacji ani powiązania między klasami
* decydowanie o ścieżkach jakimi podążać będzie wykonanie programu – to wszystko z pomocą instrukcji warunkowych i operatorów porównania
* listy czyli świetny sposób na przechowywanie wielu obiektów tego samego typu
* przechodzenie po wszystkich obiektach w listach z wykorzystaniem pętli – od teraz możemy w prosty sposób wykonać dowolną operację na każdym elemencie jaki mamy

**Kod ostateczny**

Poniżej znajduje się zawartość **wszystkich plików**, które w trakcie kursu powinniśmy mieć napisane. Jeżeli w którymś miejscu nie byłeś pewny czy na pewno poprawnie wykonałeś polecenie albo szukasz błędu w swoim kodzie to tutaj możesz sprawdzić **gotowy kod** jaki napisałem. Każdy z plików opisałem w kilku słowach.

Jeżeli ktoś zna **Gita** i chciałby ściągnąć cały projekt, który powstał to może to zrobić korzystając z [repozytorium na GitHubie](https://github.com/zajacmarekcom/kurs-c-sharp), które zawiera **commity** z każdej lekcji. Dzięki temu można również prześledzić kolejne etapy powstawania kodu.

**Program.cs**

Główny plik i klasa naszej aplikacji. Tutaj wszystko się rozpoczyna. Jednocześnie ograniczyliśmy kod w tym miejscu do minimum, zostawiając jedynie to co pozwala rozpocząć działanie programu.

namespace Bank

{

class Program

{

static void Main(string[] args)

{

BankManager bankManager = new BankManager();

bankManager.Run();

}

}

}

**IPrinter.cs**

**Interfejs** dla klas drukujących w naszym systemie.

namespace Bank

{

interface IPrinter

{

void Print(Account account);

}

}

**Printer.cs**

**Klasa** drukarki, którą wykorzystujemy w aplikacji. **Implementuje** pokazany powyżej interfejs.

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();

}

}

}

**Account.cs**

**Abstrakcyjna klasa bazowa** dla naszych kont. Zbiera wszystkie zmienne i funkcje, które są takie same dla każdego konta.

namespace Bank

{

abstract class Account

{

public int Id { get; }

public string AccountNumber { get; }

public decimal Balance { get; protected set; }

public string FirstName { get; }

public string LastName { get; }

public long Pesel { get; }

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;

}

public string GetBalance()

{

return string.Format("{0}zł", Balance);

}

public void ChangeBalance(decimal value)

{

Balance += value;

}

private string generateAccountNumber(int id)

{

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

return accountNumber;

}

}

}

**SavingsAccount.cs**

Klasa **konta oszczędnościowego**. Dziedziczy po klasie bazowej **Account** i rozszerza jej działanie o funkcję do dodawania odsetek.

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";

}

}

}

**BillingAccount.cs**

Klasa **konta rozliczeniowego**. Dziedziczy po klasie **Account** i rozszerza ją dodając funkcję do pobierania opłaty za prowadzenie konta.

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";

}

}

}

**AccountsManager.cs**

Klasa **managera kont**. Trzyma listę otwartych kont w jednym miejscu i pozwala na tworzenie, modyfikowanie i dodawanie kont w aplikacji.

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;

}

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

{

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

}

public Account GetAccount(string accountNo)

{

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

}

public IEnumerable<string> ListOfCustomers()

{

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

}

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);

}

}

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);

}

private int generateId()

{

var id = 1;

if (\_accounts.Any())

{

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

}

return id;

}

}

}

**BankManager.cs**

Klasa **managera banku**. Ponieważ nasza aplikacja jest aplikacją konsolową to potrzebowaliśmy stworzyć przejrzysty **sposób komunikacji z użytkownikiem**. Ta klasa zawiera całą logikę związaną z wyświetlaniem menu i uruchamianiem funkcji programu.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Bank

{

class BankManager

{

private AccountsManager \_accountsManager;

private IPrinter \_printer;

public BankManager()

{

\_accountsManager = new AccountsManager();

\_printer = new Printer();

}

public void Run()

{

int action;

do

{

PrintMainMenu();

action = SelectedAction();

switch (action)

{

case 1:

ListOfAccounts();

break;

case 2:

AddBillingAccount();

break;

case 3:

Console.Clear();

AddSavingsAccount();

Console.ReadKey();

break;

case 4:

AddMoney();

break;

case 5:

Console.Clear();

TakeMoney();

Console.ReadKey();

break;

case 6:

ListOfCustomers();

break;

case 7:

ListOfAllAccounts();

break;

case 8:

CloseMonth();

break;

default:

Console.Write("Nieznane polecenie");

break;

}

}

while (action != 0);

}

private void PrintMainMenu()

{

Console.Clear();

Console.WriteLine("Wybierz akcję:");

Console.WriteLine("1 - Lista kont klienta");

Console.WriteLine("2 - Dodaj konto rozliczeniowe");

Console.WriteLine("3 - Dodaj konto oszczędnościowe");

Console.WriteLine("4 - Wpłać pieniądze na konto");

Console.WriteLine("5 - Wypłać pieniądze z konta");

Console.WriteLine("6 - Lista klientów");

Console.WriteLine("7 - Wszystkie konta");

Console.WriteLine("8 - Zakończ miesiąc");

Console.WriteLine("0 - Zakończ");

}

private int SelectedAction()

{

Console.Write("Akcja: ");

string action = Console.ReadLine();

if (string.IsNullOrEmpty(action))

{

return -1;

}

return int.Parse(action);

}

private void ListOfAccounts()

{

Console.Clear();

CustomerData data = ReadCustomerData();

Console.WriteLine();

Console.WriteLine("Konta klienta {0} {1} {2}", data.FirstName, data.LastName, data.Pesel);

foreach (Account account in \_accountsManager.GetAllAccountsFor(data.FirstName, data.LastName, data.Pesel))

{

\_printer.Print(account);

}

Console.ReadKey();

}

private CustomerData ReadCustomerData()

{

string firstName;

string lastName;

string pesel;

Console.WriteLine("Podaj dane klienta:");

Console.Write("Imię: ");

firstName = Console.ReadLine();

Console.Write("Nazwisko: ");

lastName = Console.ReadLine();

Console.Write("PESEL: ");

pesel = Console.ReadLine();

return new CustomerData(firstName, lastName, pesel);

}

private void AddBillingAccount()

{

Console.Clear();

CustomerData data = ReadCustomerData();

Account billingAccount = \_accountsManager.CreateBillingAccount(data.FirstName, data.LastName, data.Pesel);

Console.WriteLine("Utworzono konto rozliczeniowe:");

\_printer.Print(billingAccount);

Console.ReadKey();

}

private void AddSavingsAccount()

{

Console.Clear();

CustomerData data = ReadCustomerData();

Account savingsAccount = \_accountsManager.CreateSavingsAccount(data.FirstName, data.LastName, data.Pesel);

Console.WriteLine("Utworzono konto rozliczeniowe:");

\_printer.Print(savingsAccount);

Console.ReadKey();

}

private void AddMoney()

{

string accountNo;

decimal value;

Console.WriteLine("Wpłata pieniędzy");

Console.Write("Numer konta: ");

accountNo = Console.ReadLine();

Console.Write("Kwota: ");

value = decimal.Parse(Console.ReadLine());

\_accountsManager.AddMoney(accountNo, value);

Account account = \_accountsManager.GetAccount(accountNo);

\_printer.Print(account);

Console.ReadKey();

}

private void TakeMoney()

{

string accountNo;

decimal value;

Console.WriteLine("Wypłata pieniędzy");

Console.Write("Numer konta: ");

accountNo = Console.ReadLine();

Console.Write("Kwota: ");

value = decimal.Parse(Console.ReadLine());

\_accountsManager.TakeMoney(accountNo, value);

Account account = \_accountsManager.GetAccount(accountNo);

\_printer.Print(account);

Console.ReadKey();

}

private void ListOfCustomers()

{

Console.Clear();

Console.WriteLine("Lista klientów:");

foreach (string customer in \_accountsManager.ListOfCustomers())

{

Console.WriteLine(customer);

}

Console.ReadKey();

}

private void ListOfAllAccounts()

{

Console.Clear();

Console.WriteLine("Wszystkie konta:");

foreach (Account account in \_accountsManager.GetAllAccounts())

{

\_printer.Print(account);

}

Console.ReadKey();

}

private void CloseMonth()

{

Console.Clear();

\_accountsManager.CloseMonth();

Console.WriteLine("Miesiąc zamknięty");

Console.ReadKey();

}

}

class CustomerData

{

public string FirstName { get; }

public string LastName { get; }

public long Pesel { get; }

public CustomerData(string firstName, string lastName, string pesel)

{

FirstName = firstName;

LastName = lastName;

Pesel = long.Parse(pesel);

}

}

}

**Co dalej?**

Ukończony kurs to dobry początek. Jednak chcąc myśleć o karierze programisty trzeba dokonać kolejnych wyborów i brnąć w kolejne zagadnienia. Przedstawię Ci więc tutaj listę przykładowych decyzji, ścieżek i zagadnień jakie możesz rozważyć w kolejnych etapach nauki.

*Poniżej znajdziesz przykładowe książki, które polecam. Lista będzie się sukcesywnie powiększać. Dodatkowo kupując je przez zamieszczone poniżej linki wspierasz rozwój bloga.*

**Dalsza nauka języka**

To czego się nauczyliśmy w trakcie kursu to oczywiście nie wszystko. Nawet jeżeli mówimy tylko o języku **C# i .NET Frameworku**, nie wkraczając w świat **zewnętrznych bibliotek i frameworków**, to będziemy mieli czym się zajmować i co poznawać.

Nie tylko nie poruszyliśmy wszystkich często używanych zagadnień takich jak **typ wyliczeniowy**. Oprócz tego nie wspomnieliśmy nawet o zagadnieniach związanych z **wielowątkowością**, elementami sieciowymi czy bardziej zaawansowanych zagadnieniach związanych z **dziedziczeniem**, **operatorami** czy **funkcjami**.

Książki:

[„C# 7.1 i .NET Core 2.0 dla programistów aplikacji wieloplatformowych”](https://helion.pl/view/10861k/c-7-1-i-net-core-2-0-dla-programistow-aplikacji-wieloplatformowych-mark-j-price,c71nc2.htm)

**Testy**

Testowanie aplikacji to nie tylko jej uruchamianie i sprawdzanie czy działa poprzez klikanie wszystkiego. Podstawowym narzędziem chroniącym przed większością błędów i dającym informację zwrotną czy nasza zmiana nie zepsuła tego co działało wcześniej są **testy jednostkowe**. Dzięki nim możemy sprawdzić czy małe fragmenty kodu w konkretnej sytuacji, takiej jak konkretny zbiór danych, zachowują się dokładnie tak jak tego oczekiwaliśmy.

Książki:

[„Testy jednostkowe. Świat niezawodnych aplikacji.”](https://helion.pl/view/10861k/testy-jednostkowe-swiat-niezawodnych-aplikacji-wydanie-ii-roy-osherove,tesjed.htm)

**Typ aplikacji**

Dla uproszczenia tłumaczenia zagadnień cały kurs opierał się o **aplikację konsolową**. Jednak tego typu programy, poza studiami i specjalistycznymi narzędziami, są zdecydowaną niszą. Dlatego warto dosyć szybko po poznaniu podstaw języka wybrać w jaki **typ aplikacji** chcemy iść. Zasadniczo w **C#** mamy do wyboru dwie duże ścieżki

* Aplikacje webowe
* Aplikacje desktopowe

Pierwszy ma zdecydowaną przewagę jeśli chodzi o ilość pracy. Większość firm przenosi swoje narzędzia do przeglądarki. **Możesz się nauczyć programowania webowego np. korzystając z mojego kursu ASP.NET Core, które powstaje i jest dostępny w przedsprzedaży** [**O TUTAJ**](https://kurs-szarpania.pl/nauka/szkola-web-szarpania-poczatek/)**.**

**„Chcę pisać gry!”**

A może zacząłeś się uczyć programowania bo skusiła Cię do tego wizja stworzenia swojej gry? Jeżeli tak to również tutaj C# jest dobrym wyborem bo wykorzystać go możemy w silniku Unity i za ich pomocą spełnić swoje marzenie o zostaniu programistą gier.

Książki i linki:

[„Unity i C#. Podstawy programowania gier”](https://helion.pl/view/10861k/unity-i-c-podstawy-programowania-gier-ewa-ross-jacek-ross,unityc.htm)

[„Unity w akcji”](https://helion.pl/view/10861k/unity-w-akcji-joe-hocking,uniwak.htm)

[Blog mojego kolegi Marka poświęcony głównie tworzeniu gier z wykorzystaniem Unity](https://mwin.pl)

**Zagadnienia uniwersalne**

Poza nauką samego języka i związanych z nim dodatkowych bibliotek i technologii warto też poznawać zagadnienia, które nie są bezpośrednio związane z jednym językiem. Powodują one, że nasza praca jest bardziej wydajna, a kod, który tworzymy będzie wysokiej jakości.

W tym miejscu wchodzi też takie zagadnienie jak „jakość kodu. Co to jest, jak i po co o to dbać napisałem w [jednym ze swoich tekstów na blogu](https://zajacmarek.com/2019/01/po-co-dbac-o-jakosc-kodu/).

**Git**

Git jako narzędzie do **wersjonowania kodu** pozwala na większą swobodę w modyfikowaniu kodu i ułatwia pracę w zespole.

Pierwsza zaleta wynika z tego, że mogąc zapamiętać stan kodu w dowolnym momencie i mogąc potem bez problemu do tego momentu wrócić możemy dużo odważniej wprowadzać zmiany. Nieważne jak dużo napsujemy to mamy możliwość powrotu jednym kliknięciem do ostatniej działającej wersji.

Druga zaleta po części wynika z pierwszej, a po części wynika z tego, że **Git** daje możliwość łatwego **śledzenia zmian** wprowadzanych przez innych i łączenia ich z tymi, które my wprowadziliśmy.

Książki:

[„Git. Rozproszony system kontroli wersji”](https://helion.pl/view/10861k/git-rozproszony-system-kontroli-wersji-wlodzimierz-gajda,gitroz.htm)

[„Pro Git” dostępny w wersji online za darmo](https://git-scm.com/book/pl/v2)

**Wzorce projektowe**

Czyli powtarzające się w wielu projektach sposoby rozwiązania wybranych problemów. Lista spisanych wzorców projektowych zawiera schematy, które w sprytny sposób pozwalają podejść do struktury naszych programów i przepływu danych w nich.

Książki:

[„Wzorce projektowe. Rusz głową!”](https://helion.pl/view/10861k/wzorce-projektowe-rusz-glowa-eric-freeman-bert-bates-kathy-sierra-elisabeth,wzorvv.htm)

[„Design Patterns in C#”](https://www.apress.com/gp/book/9781484236390)

**Dobre praktyki**

Magiczne skróty takie jak **SOLID**, **KISS**, **DRY** powinny być Ci dosyć szybko znane. Są to zasady, które kierują Twój kod w stronę wysokiej jakości łatwej w utrzymaniu i modyfikowaniu struktury. Bardziej ogólnym pojęciem, z którym warto się zapoznać jest **czysty kod**.

Książki:

[„Czysty kod. Podręcznik dobrego programisty”](https://helion.pl/view/10861k/czysty-kod-podrecznik-dobrego-programisty-robert-c-martin,czykov.htm) <- pozycja wręcz obowiązkowa

[„Mistrz czystego kodu. Kodeks postępowania profesjonalnych programistów”](https://helion.pl/view/10861k/mistrz-czystego-kodu-kodeks-postepowania-profesjonalnych-programistow-robert-c-martin,mckkod.htm)

**Podsumowując**

Kończąc ten kurs nabyłeś **podstawową wiedzę** na temat języka **C#**. Mam szczerą nadzieję, że będzie to dla Ciebie dobry start do dalszej nauki i podjęcia w przyszłości pracy bądź zajmowania się tym hobbistycznie.

Zachęcam też do pozostania na dłużej i zapoznania się z [resztą tekstów obecnych na blogu](https://zajacmarek.com). Dzielę się tam zarówno poradami technicznymi jak i przemyśleniami powiązanymi z pracą programisty. Udzielam się również na [fanpagu bloga na Facebooku](https://www.facebook.com/zajacmarekblog), do którego śledzenia również zapraszam.

W razie pytań, wątpliwości, uwag bądź znalezienia błędów w kursie możesz pisać do mnie wiadomość prywatną na Facebooku albo wysłać email na [contact@zajacmarek.com](mailto:contact@zajacmarek.com)