**[KURS C#] Genetyka klas**



**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)!

Każdy z nas odziedziczył pewne cechy od rodziców. Nasze rodziny posiadają jakieś cechy wspólne, które widać w zachowaniu czy wyglądzie. A co jeżeli Ci powiem, że **klasy** w języku **C#** też mogą korzystać z podobnego mechanizmu? Dowiedzmy się jak **klasy** mogą po sobie **dziedziczyć**!

Omówimy tutaj jedynie **podstawy dziedziczenia**, nie zagłębiając się zbytnio w temat, bo jest to jedna z tych rzeczy, którą najlepiej poznać w praktyce i poprzez popełnione błędy dowiedzieć się kiedy nie działa ;)

**Jestem Twoim dzieckiem**

Od razu powinniśmy sobie zaznaczyć, że **dziedziczenie** ludzi czy zwierząt różni się od dziedziczenia klas w języku C#. W pierwszym wypadku powstaje kombinacja cech dwójki rodziców i może być ona różna. W drugim przypadku następuje przejęcie pewnych cech po rodzicu klasy i jest ono pewne, stałe i w 100% przewidywalne.

Praktycznie **każda klasa** może dziedziczyć z innej klasy. Tworzą one wtedy hierarchię **rodzica** i **dziecka**. Jednak w przeciwieństwie do ludzi, klasa w C# może mieć **tylko jednego rodzica**. Jest to odgórnie wprowadzone ograniczenie, które chroni nas przed pewnymi głupimi decyzjami (programujący w  C++ pewnie rozumieją). Nieważne. Ważne jest to jak możemy dokonać **dziedziczenia** i co nam ono daje.

**Część wspólna**

Nie ma co przedłużać, przejdźmy od razu do kodu.

Przyjrzyjmy się dwóm klasom reprezentującym nasze **dwa typy kont bankowych**:

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;

}

public string GetBalance()

{

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

}

}

class BillingAccount

{

public string AccountNumber;

public decimal Balance;

public string FirstName;

public string LastName;

public long Pesel;

public BillingAccount(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;

}

public string GetBalance()

{

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

}

}

Być może już dawno zauważyłeś, że posiadają one **takie same zmienne**, które przechowują informacje na ich temat.

Dzieje się tak ponieważ mimo, że są różnymi **typami** to w gruncie rzeczy obie klasy reprezentują jakieś **konto bankowe**. Nieważne czy **oszczędnościowe** czy **rozliczeniowe** – każde z nich musi posiadać pewną wspólną **pulę danych**. Może dałoby się tę część wyciągnąć? W końcu kopiowanie tego samego kodu nie jest nam na rękę, zwłaszcza kiedy trzeba będzie coś zmienić w obu miejscach.

Uspokoję Cię – jest na to sposób. Możemy stworzyć **klasę bazową**.

**Bazując na…**

Klasa bazowa to po prostu inna klasa, z której **dziedziczą** inne. W naszym przypadku oba typy kont bazować będą na klasie, która po prostu reprezentuje **konto**, dowolne. Dodajmy więc do projektu klasę **Account** i wypełnijmy ją w następujący sposób:

namespace Bank

{

class Account

{

public string AccountNumber;

public decimal Balance;

public string FirstName;

public string LastName;

public long Pesel;

}

}

Po prostu wstawiamy do niej te dane, które są **takie same** dla obu typów konta. W naszym przypadku będą to wszystkie dane.

Pora więc wykorzystać fakt, że mamy jakąś bazę. Przejdźmy do klasy **SavingsAccount** i zmodyfikujmy ją w pokazany poniżej sposób:

namespace Bank

{

class SavingsAccount : Account

{

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;

}

public string GetBalance()

{

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

}

}

}

To co zrobiliśmy to po pierwsze usunęliśmy z niej te **pola**, które dodaliśmy do klasy **Account**, a po drugie **zastosowaliśmy dziedziczenie**.

Taki zapis:

class SavingsAccount : Account

Oznacza, że klasa **SavingsAccount** **dziedziczy** z klasy **Account**. Dziedziczyć możemy tylko z jednej klasy naraz, tzn. że nie możemy tutaj dodać po przecinku kolejnej klasy, która również by była klasą bazową dla naszej nowej klasy.

Od teraz mamy dostęp w klasie **SavingsAccount** do wszystkich **zmiennych** oznaczonych jako public w klasie **Account**. O co chodzi z tym public przekonasz się już wkrótce. Ważne, że od tego momentu możesz korzystać z tych zmiennych jakby były one częścią klasy **SavingsAccount**, bo w pewnym sensie takimi się stają.

**Nie tylko zmienne**

W taki sam sposób jak to robimy ze **zmiennymi** możemy też postąpić z **metodami**. Jeżeli przeniesiemy **publiczne funkcje** do **klasy bazowej** to nadal możemy z nich korzystać wykorzystując **klasę dziedziczącą**. Dzięki temu również metody, które są wspólne dla wszystkich typów kont możemy przenieść do klasy **Account**:

namespace Bank

{

class Account

{

public string AccountNumber;

public decimal Balance;

public string FirstName;

public string LastName;

public long Pesel;

public string GetFullName()

{

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

return fullName;

}

public string GetBalance()

{

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

}

}

}

W klasie **SavingsAccount** zostanie więc tylko uzupełnienie wartości zmiennych poprzez **konstruktor**:

namespace Bank

{

class SavingsAccount : Account

{

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

{

AccountNumber = accountNumber;

Balance = balance;

FirstName = firstName;

LastName = lastName;

Pesel = pesel;

}

}

}

Jednak i to możemy zmienić!

**Konstruktor bazowy**

Tak samo jak możemy w **klasach dziedziczących** korzystać ze **zmiennych** i **funkcji** tak samo nic nie stoi na przeszkodzie, żeby skorzystać z jej **konstruktora**. Użycie go nie jest takie trudne, chociaż nieznacznie różni się od użycia zwykłej funkcji.

Najpierw dodajmy w klasie **Account** konstruktor, który uzupełnia przekazane mu **dane**:

namespace Bank

{

class Account

{

public string AccountNumber;

public decimal Balance;

public string FirstName;

public string LastName;

public long Pesel;

public Account( 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;

}

public string GetBalance()

{

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

}

}

}

Skoro klasa **Account** posiada **konstruktor**, który uzupełnia wszystkie zmienne to bez sensu robić to samo w **klasie dziedziczącej**. Skorzystajmy z niego. Nawet musimy z tego skorzystać, ponieważ klasa bazowa straciła właśnie swój **konstruktor domyślny**.

Robimy to w konstruktorze **klasy dziedziczącej**. Nadal musi on przyjmować wszystkie niezbędne wartości jednak zamiast samemu coś z nimi robić przekazuje je do **konstruktora klasy bazowej**.

Taka operacja jest bardzo prosta, wystarczy, że użyjemy słowa kluczowego base() , które reprezentuje **konstruktor klasy bazowej**. Jedyna trudność to miejsce w jakim to robimy, bo nie używamy go wewnątrz konstruktora klasy dziedziczącej, ale zaraz za jego **listą parametrów**:

namespace Bank

{

class SavingsAccount : Account

{

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

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

{

}

}

}

W tym przypadku przeniosłem wywołanie bazowego konstruktora do nowej linii, żeby poprzednia nie była za długa. Jednak jak widzimy jego użycie jest jeszcze **przed nawiasami klamrowymi** oznaczającymi treść konstruktora.

Ogólnie wykorzystanie konstruktora bazowego wygląda w następujący sposób:

NazwaKlasy(parametry) : base(parametry\_konstruktora\_z\_klasy\_bazowej)

{

}

Dzięki temu **część wspólną** dla wszystkich klas dziedziczących możemy całkowicie przenieść do klasy bazowej i tylko z tej implementacji korzystać przy okazji kolejnego dziedziczenia.

Innym, dosyć często stosowanym, zastosowaniem konstruktora klasy bazowej i konstruktorów w klasach dziedziczących jest ustawianie konkretnych wartości. Przykładowo **klasa bazowa**, np. Głośnik, oczekuje w konstruktorze przekazania wartości dotyczącej mocy. Ale już dziedziczący po nim konkretny typ głośnika może mieć konstruktor bez parametrów i moc w klasie bazowej podać na sztywno poprzez użycie **konstruktora bazowego**. Tym samym przykrywając możliwość **parametryzacji klasy**.

**Pod przykrywką**

Fakt, że klasa bazowa może zawierać część wspólną wszystkich klas, które z niej dziedziczą to nie jedyna jej zaleta. Inną jest fakt, że tworząc zmienną, której typem jest właśnie tak klasa możemy do niej przypisać obiekty dowolnej klasy dziedziczącej.

Account account = new SavingsAccount(...);

Dlatego powyższy kod jest całkowicie poprawny.

Jednak w takim przypadku chcąc cokolwiek z tą zmienną zrobić jesteśmy ograniczeni jedynie do elementów, które zawiera klasa **Account**. Więc jeżeli, któraś z klas dziedziczących zawierałaby coś unikatowego dla niej to przez taką zmienną nie dostaniemy się do tego w dotychczas nam znany sposób.

**Ćwiczenie 1**

Zmodyfikuj klasę **BillingAccount** w taki sposób aby również **dziedziczyła** z klasy **Account** tak samo jak klasa **SavingsAccount**. Pamiętaj o wszystkich elementach takich jak użycie **konstruktora bazowego**.

**Co za abstrakcja!**



Dodaliśmy tutaj klasę, na której bazują wszystkie konta. Ma ona swoje **zmienne** i **funkcje**. Posiada też swój **konstruktor**. Oznacza to, że możemy utworzyć w aplikacji obiekt typu **Account**. A przecież to nie ma sensu!

Konto, jeżeli nie jest konkretnego typu, nie stanowi czegoś co powinno mieć **rzeczywistą reprezentację** w postaci gotowego obiektu. Konto jest w tym przypadku czymś **abstrakcyjnym**. Czymś na czym chcemy oprzeć konkretne typy kont, ale samo w sobie nie stanowi czegoś co nadaje się do wyprodukowania, utworzenia.

Kiedy idziesz do banku to nie zakładasz po prostu konta. Zakładasz konto oszczędnościowe, rozliczeniowe, inwestycyjne. Wszystkie one są kontami, ale nie możemy założyć po prostu konta jako takiego.

Na szczęście język **C#** pozwala nam dodawać klasy, które możemy użyć przy dziedziczeniu jednak nie mamy możliwości utworzenia bezpośrednich obiektów tych klas. Takie klasy są **klasami abstrakcyjnymi**. Do ich definiowania służy słowo kluczowe abstract. Wystarczy, że w klasie, przed słowem class dodamy słowo abstract i już z takiej klasy nie utworzymy **żadnego obiektu**. Jedyna możliwość jej użycia to będzie użycie w roli **klasy bazowej** dla innej klasy.

Skoro w naszym przypadku **Account** nie jest czym czego obiekty chcemy móc tworzyć to dobrym rozwiązaniem jest zrobienie z niej **klasy abstrakcyjnej**. Wystarczy, że dodamy na początku, przez słowem **class** i **nazwą klasy**, słowo **abstract**:

namespace Bank

{

abstract class Account

{

// zawartosc klasy

}

}

Pozwoliłem sobie nie wklejać tutaj całej zawartości klasy żebyś lepiej widział gdzie słowo kluczowe abstract  się znalazło.

Jeżeli teraz chcielibyśmy wykonać gdzieś w kodzie instrukcję new Account() to dostaniemy błąd, który będzie oznaczał, że taka operacja dla **klasy abstrakcyjnej** nie jest możliwa.

Jednak możliwe jest utworzenie zmiennej, której typem będzie taka klasa.

**Tylko o tym wspomnę**

Z klasami abstrakcyjnymi wiąże się jeszcze coś takiego jak **metody abstrakcyjne**.

Skoro korzystamy ze **zmiennych**, których typem jest **klasa bazowa** oznaczona jako **abstrakcyjna** to chcemy móc jakoś to wykorzystać nie musząc za każdym razem wyciągać prawdziwego typu obiektu, który tam siedzi. W tym celu możemy w naszej klasie abstrakcyjnej tylko **dać znać**, że taka metoda jest dostępna, a tak naprawdę jej zachowanie wpisać już w każdej z **klas dziedziczących** osobno, tak że każda z nich będzie robiła coś innego.

**Czy to Ty czy to ja?**

W przypadku aplikacji, którą piszemy dobrym przykładem będzie nowa **metoda**, która zwraca tekst z **nazwą typu konta** – **„OSZCZĘDNOŚCIOWE”**, **„ROZLICZENIOWE”**. Sama klasa **Account** nie ma tutaj nic do gadania bo nie jest żadnego typu. Jednak chcemy móc operować na zmiennej **typu Account** nie przejmując się tym co się pod nią kryje:

Account account;

account = new SavingsAccount();

Console.WriteLine(account.TypeName());

account = new BillingAccount();

Console.WriteLine(account.TypeName());

W podanym powyżej przykładzie naszym celem jest to, żeby pierwszy przypadek wypisał na ekranie słowo „OSZCZĘDNOŚCIOWE”, a drugi „ROZLICZENIOWE” mimo, że korzystamy ciągle ze zmiennej **Account** i program nie widzi tego co jest dostępne tylko w klasie **SavingsAccount** albo **BillingAccount**.

**Przedstaw się wszystkim**

Jak w takim razie rozwiązać to zadanie?

Możemy to zrobić w następujący sposób. Dodajmy w klasie **Account** **abstrakcyjną metodę** TypeName() :

namespace Bank

{

abstract class Account

{

public string AccountNumber;

public decimal Balance;

public string FirstName;

public string LastName;

public long Pesel;

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

{

AccountNumber = accountNumber;

Balance = balance;

FirstName = firstName;

LastName = lastName;

Pesel = pesel;

}

// ABSTRAKCYJNA METODA

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);

}

}

}

Jej cechą charakterystyczną jest to, że nie posiada **żadnej implementacji**. W tej klasie jedynie mówimy, że taka **metoda istnieje**. Dzięki temu możliwe jest jej użycie kiedy posługujemy się zmienną typu **Account**.

Drugą cechą jest to, że taką metodę muszą teraz **zaimplementować** wszystkie klasy, które dziedziczą po klasie, w której znajduje się **metoda abstrakcyjna**. Oprócz tego klasa, która posiada metodę abstrakcyjną sama musi być abstrakcyjna, bo inaczej chcąc utworzyć obiekt takiej klasy i próbując korzystać z tej funkcji nie mielibyśmy czego wykonać.

**Implementacja** takiej metody w klasie dziedziczącej jest banalnie prosta. Wystarczy, że wstawimy w niej funkcję o takiej samej nazwie tylko słowo abstract  zamienimy na override, co będzie oznaczało, że świadomie **nadpisujemy** taką funkcję. W przypadku klasy **SavingsAccount** sprawa wygląda następująco:

namespace Bank

{

class SavingsAccount : Account

{

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

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

{

}

public override string TypeName()

{

return "OSZCZĘDNOŚCIOWE";

}

}

}

Teraz jeżeli mamy **zmienną Account**, do której przypiszemy **obiekt klasy SavingsAccount** to korzystając z metody TypeName() skorzystamy właśnie z tej, którą zapisaliśmy powyżej. Tak samo będzie ze wszystkimi innymi klasami, które **dziedziczą** po **Account**.

To o czym tutaj mówię jest częścią mechanizmu, który nazywa się **polimorfizmem**. Jego działanie przedstawia się często na studiach czy w tutorialach na przykładzie klasy Zwierze i dziedziczących klas konkretnych zwierząt, które mają własne implementacje metod typu DajGłos().

Jednak takie przykłady są mocno oderwane od prawdziwych wyzwań jakie polimorfizm rozwiązuje w programowaniu. Lepsze przykłady pokazują polimorfizm w kontekście implementacji różnych brył geometrycznych.

[Przykładowy opis z dyskusji na StackOverflow](https://stackoverflow.com/questions/1031273/what-is-polymorphism-what-is-it-for-and-how-is-it-used/1031385" \l "1031385).

**Ćwiczenie 2**

Zaimplementuj metodę TypeName() również w klasie **BillingAccount**.

**W końcu to wiemy!**

Dzięki temu co dodaliśmy przed chwilą możemy do naszej drukarki danych o koncie dodać wyświetlanie jakiego typu konto to jest!

Dodatkowo skoro przenieśliśmy metody zwracające stan konta wraz z walutą i imię i nazwisko właściciela do klasy bazowej to również je możemy wykorzystać. Także teraz klasa **Printer** wygląda następująco:

using System;

namespace Bank

{

class ConsolePrinter

{

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();

}

}

}

Operując cały czas na parametrze **typu Account** możemy wyświetlić różne wartości w polu **„Typ”** w zależności od tego jakiego konkretnie typu obiekt się tam znajdzie. Niesamowicie pomocna sprawa.

**Ćwiczenie 3**

Zamień typ zmiennej przechowującej konto rozliczeniowe (**BillingAccount**) na **Account** i sprawdź co się stanie. Następnie spróbuj przypisać wartość z tej zmiennej do zmiennej, w której trzymałeś wcześniej **konto oszczędnościowe**. Sprawdź czy program się uruchomi i czy odpowiednia informacja o typie konta zostanie wyświetlona przez drukarkę.

**Rodzice i dzieci**

Wkroczyliśmy właśnie w świat wielopokoleniowych klas. Jest dobrze. Prawdziwą siłę tego mechanizmu zobaczysz już niedługo kiedy będziemy dodawać zarządzanie wszystkimi kontami i interakcję z użytkownikiem. M.in. dziedziczenie mocno pomoże w uniknięciu powtarzania kodu i wykorzystania jednego mechanizmu dla wszystkich typów kont.