EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

Název projektu	Rozvoj vzdělávání na Slezské univerzitě v Opavě
Registrační číslo projektu	CZ.02.2.69/0.0./0.0/16_015/0002400

Objektové programování

Distanční studijní text

Radomír Perzina

Karviná 2021

SLEZSKÁ UNIVERZITA
OBCHODNĚ PODNIKATELSKÁ FAKULTA V KARVINÉ

Obor: Informatika

Klíčová slova: Programování, C#, Microsoft Visual Studio, objekt, třída, konstruktor, instance, zapouzdření, dědičnost, polymorfismus, virtuální metoda, přetěžování, operátor, grafické uživatelské rozhraní, databáze, Java.

Anotace: Tento text je určen studentům bakalářského stupně studia na Slezské univerzitě, Obchodně podnikatelské fakultě v Karviné. Jakožto studijní opora je určen především studentům kombinované formy studia, mohou jej však stejně dobře využít i studenti prezenční formy. Tato opora je zaměřena na základy objektového programování, proto ji nejvíce ocení studenti, kteří již mají s programováním nějaké zkušenosti, ale může být užitečná i pro začátečníky. Jsou zde vysvětleny základní konstrukce objektového programování v jazyce C#. Text je orientován na praktické použití, proto jsou všechny příkazy a konstrukce demonstrovány na ukázkách reálného kódu v Microsoft Visual Studiu. Výukový text je členěn tak, že v první kapitola je věnována historii programování a seznámení s vývojovým prostředím. Následující tři kapitoly se podrobněji věnuje základům objektového programování. V následujících třech kapitolách jsou zmíněny pokročilejší prvky a zejména aplikace objektového programování. Poslední kapitola se potom specifikům programovacího jazyka Java.

Autor: **Ing. Radomír Perzina, Ph.D.**

Obsah

ÚVODEM

Do rukou se Vám dostává výukový text, který se snaží čtenáře seznámit se základy objektového programování v jazyce C# a se základy používání vývojového prostředí Microsoft Visual C#. Tento text je vhodný jako distanční opora ve výuce předmětů na vysokých školách se zaměřením na informatiku. Obsahem výkladu je stručné seznámení s historií programování, výklad základních principů objektového programování, ukázka možností grafického uživatelského rozhraní včetně práce s databázemi a zároveň jsou zde zmíněna specifika programovacího jazyka Java.

Text je strukturován do sedmi kapitol. Každá kapitola začíná stručným seznámením s jejím obsahem v rychlém náhledu kapitoly, dále obsahuje stručné cíle a klíčová slova. V samotném textu se vyskytují distanční prvky, které čtenáře upozorní na důležité části k zapamatování a na texty pro zájemce. Každá kapitola končí kontrolními otázkami, krátkým shrnutím a odpověďmi na otázky.

RYCHLÝ NÁHLED STUDIJNÍ OPORY

V současné době jsou základy programování důležitou součástí každého absolventa vysoké školy nejen se zaměřením na informatiku. Cílem této opory je poskytnout čtenáři základní seznámení s objektovým programování v jazyce C# a Java. Text je členěn do osmi kapitol.

V první kapitole je zmíněna stručná historie programovacích jazyků a jsou zde demonstrovány základy práce s vývojovým prostředím Microsoft Visual Studio. Druhá kapitola je zaměřena na základy objektového programování a možnostmi ochrany konzistence objektů pomocí zapouzdření. Třetí kapitola se věnuje způsobu, jak můžeme znovupoužít a rozšířit kód pomocí dědičnosti. Cílem čtvrté kapitoly je seznámit se s polymorfismem a přetěžováním operátorů. Pátá kapitola se zabývá knihovnami tříd a možnostmi dokumentace projektů. V šesté kapitoly jsou uvedeny základy práce s formulářovými aplikacemi a seznámení s nejčastěji používanými grafickými ovládacími prvky. Sedmá kapitola je věnována úvodu do databázových aplikací. Závěrečná kapitola potom stručně zmiňuje programovací jazyk Java se zaměřením na jeho odlišnosti od jazyka C#.

1 ZÁKLADY PROGRAMOVÁNÍ

RYCHLÝ NÁHLED KAPITOLY

V této kapitole se dozvíte, jaký software budeme v rámci studia programování pomocí této knihy používat, jak jej získat, nainstalovat a jak s ním pracovat. Dozvíte se něco historii programování, co je to program, programovací jazyk a vytvoříte si svůj první program v jazyce C#. Neopomeneme ani komentáře a jakým způsobem můžeme hledat a odstraňovat chyby v programu.

CÍLE KAPITOLY

Po prostudování této kapitoly budete umět:

- Vyjmenovat z historického hlediska nejvýznamnější programovací jazyky.
- Nainstalovat Microsoft Visual Studio.
- Vytvořit jednoduchý program v jazyce C#.
- Okomentovat části kódu.
- Definovat rozdíl mezi syntaktickou a sémantickou chybou.
- Krokovat program.
- Vložit do programu bod přerušení.

KLÍČOVÁ SLOVA KAPITOLY

Program, programování, MS Visual studio, komentář, syntaktická chyba, sémantická chyba, ladění, bod přerušení.

1.1 Historie programovacích jazyků

Historie informatiky nebo, chcete-li, výpočetní techniky začíná už v době starověkých civilizací výpočetními pomůckami (abakus, čínské počítací hůlky) a pokračuje v novověku od dob renesance dokonalejšími mechanickými kalkulátory buď digitálními (konstrukce Shickardova, Pascalova, Leibnizova, Colmarova, Odhnerova), nebo analogovými (logarit-

mické pravítko), aby v 19. století mechanická etapa vývoje vyvrcholila velkolepými projekty jednoúčelových počítačů – diferenčních strojů, jež dokázaly automaticky počítat a tisknout tabulky matematických funkcí, a Hollerithových děrnoštítkových systémů, jež dokázaly zpracovávat rozsáhlé relační databáze.

První počítače měly velmi daleko k dnešním univerzálním nástrojům, provádějícím výpočty, zpracovávajícím informace různých druhů a umožňujícím rychlou komunikaci prostřednictvím světové počítačové sítě internet. První počítače prováděly skutečně „pouze“ výpočty, které předtím ručně realizovaly týmy lidských počtářů. V angličtině se právě v tuto dobu posunul význam slova computer od původního počtář (člověk profesionálně provádějící výpočty) k počítač (stroj, který provádí výpočty).

První generace počítačů se programovala ručním propojováním vodičů (propojováním zdířek spojovacími vodiči na ovládacím panelu, přepínáním přepínačů apod.), k němuž s o něco později přidalo čtení vstupních instrukcí a dat z papírové děrné pásky. Přeprogramování počítače na jinou úlohu byl pracný úkon, který vyžadoval trpělivost a pečlivost a souhru několika členů týmu. Kromě programátorů, kteří vymysleli a zapsali algoritmus, byli potřeba kodéři, kteří přepsali celý program do posloupnosti dvojkových čísel a operátoři, kteří řídili vlastní spuštění a provedení kódu. Ladění programu bylo velmi obtížné a časově náročné, protože pro něj neexistovaly žádné pomůcky.

Další nevýhodou první generace byla neexistence jednotících norem. Co počítač, to originál s jinou sadou příkazů procesoru, s jiným strojovým kódem. Programátor musel myslet nejen na algoritmus, ale i na uložení kódu v operační paměti, stejně jako na alokaci paměti pro všechny proměnné, pole proměnných atd. To vše v absolutní pozici v paměti, tedy konkrétní instrukce programu (početní operace, rozhodovací operace, skok v programu, …) měla v podstatě předem pevně dané místo operační paměti, ve kterém byla uložena.

S rozvojem výpočetní techniky bylo brzy zřejmé, že software (strojový kód) zaostává za možnostmi hardware a že je potřeba vymyslet novou „technologii“ programování, která by využití počítačů nebrzdila. Prvním krokem od strojového kódu k symbolickému jazyku přístupnému člověku byla skupina jazyků symbolických adres (JSA; angl. assembly language), které místo číselných kódů pro jednotlivé instrukce procesoru používaly symbolické zkratky (např. ADD pro součet, JMP pro skok, MOV pro přesun hodnoty z paměti do registru) a současně nepožadovaly absolutní alokaci paměti, místo níž použily symbolické adresy, jejichž konkretizaci pak zařídil překladač typu assembler (tedy „sestavovač“, program, který z programu v JSA sestaví strojový kód). Často se nesprávně název assembler používá i pro JSA, ve skutečnosti je assembler historicky první typem překladače programovacího jazyka, který překlad z JSA do strojového kódu provádí na několik průchodů, při nichž postupně provede potřebnou alokaci paměti pro vlastní program i pro potřebné proměnné.

Assemblery se pro některé speciální úlohy používají i v současnosti. Např. tam, kde je nutný přímý přístup k hardware (ovladače zařízení), vysoký výpočetní výkon, nebo řízení procesů v reálném čase. V JSA se také psaly první operační systémy pro sálové počítače.

Dalším typickým prvkem, který usnadní programování v JSA, je zavedení pseudoinstrukcí a maker, čímž se zjednoduší zápis opakujících se částí kódu. Jde o předstupeň plnohodnotných podprogramů.

JSA jsou tak jako strojový kód vázané na konkrétní typ, resp. konkrétní řadu procesorů. Programy napsané v JSA pro určitý hardware nejsou buď přenositelné vůbec, nebo přenositelné jen s velkými obtížemi na jinou hardwarovou platformu. JSA přinesly programátorům velké zjednodušení práce, byly typickým nástrojem druhé generace počítačů, ale bylo zřejmé, že ani ony nejsou elegantním a jednoduchým řešením zápisu algoritmů. Na cestě od stroje k člověku bylo potřeba učinit další krok.

Třetí generaci softwaru představují obecné problémově orientované programovací jazyky. Často se jim říká také vyšší programovací jazyky a myslí se tím, že se od nízké úrovně strojového kódu či JSA dostali značně vysoko směrem k algoritmickému myšlení člověka plánujícího složitý výpočet.

Podobně označení „problémově orientovaný“ znamená, že programátor nemusí myslet na procesor a jeho instrukční sadu, ale může se plně soustředit na problém, který chce vyřešit. V praxi to značí nezávislost na konkrétním hardware, a tedy snadnou přenositelnost napsaných programů.

Takových programovacích jazyků existovala už na konci 50. let celá řada. Kdybychom se snažili postihnout celou šíři vývoje, hrozilo by, že ztratíme potřebný nadhled. Proto se soustředíme pouze na tři nejvýznamnější představitele vývoje, za něž považuji jazyky FORTRAN, COBOL a ALGOL.

Jazyk FORTRAN (z angl. FORmula TRANslator) vyvíjela ve firmě IBM od roku 1954 skupina, kterou vedl John Backus (1924 – 2007). Důraz při vývoji jazyka a překladače byl kladen na rychlost a efektivnost zejména vědecko-technických výpočtů, přenositelnost nebyl na prvním místě, protože šlo o projekt orientovaný na sálové počítače IBM řady 700. Díky této koncepci se FORTRAN stal jedním z nejrozšířenějších programovacích jazyků své doby. Vývoj jazyka intenzivně pokračoval do roku 1963 (FORTRAN IV), jazyk byl později normalizován (standardy ANSI a ISO), byly vydávány jeho nové verze a své uživatele má ještě dnes (FORTRAN 2008).

John Backus je znám také jako spolutvůrce Backus-Naurovy formy pro popis gramatik formálních jazyků. Používá se jak pro popis gramatik programovacích jazyků, instrukčních sad, komunikačních protokolů a dokonce i částí gramatik skutečných jazyků. Ještě před jazykem FORTRAN vyvinul jazyk Speedcoding, který se však nerozšířil. Později spolupracoval na vývoji jazyka ALGOL a také podnítil zájem o funkcionální programování, o němž budeme mluvit později.

Jazyk COBOL (z angl. COmmon Business Oriented Language) byl oficiálně vydán roku 1959 sdružením CODASYL, založeným z iniciativy Ministerstva obrany USA. Administrativní pokyn vlády USA z roku 1960 pak nařídil vybavit překladačem COBOLu každý

komerčně instalovaný počítač, čímž došlo jeho k masovému rozšíření. Matkou COBOLu bývá nazývána Grace Hopper (1906 – 1992), která stavěla na předchozích zkušenostech z práce matematičky a programátorky ve firmě Eckert-Mauchly Corporation (výrobce počítačů BINAC a UNIVAC), pro níž a pro firmu Remington Rand zpracovala programovací jazyky FLOW-MATIC a MATH-MATIC. Tato dáma je autorkou první příručky počítačové terminologie a matkou myšlenky použití jednotné matematické notace a jednoduchých anglických příkazů k sestavení zápisu programu v podobě srozumitelné člověku i počítači. COBOL umožnil kromě vědecko-technických výpočtů také základní zpracování databází. Po formální stránce nedosáhl přehlednosti FORTRANu či ALGOLu, ale z praktického hlediska šlo o snadno implementovatelný a dobře použitelný programovací jazyk pro běžné rutinní výpočty v komerční sféře.

Oba výše uvedené jazyky vznikly v USA a odtud se šířily do celého světa. Třetí – programovací jazyk ALGOL (z angl. ALGOrithmic Language) vznikl z iniciativy evropského Odborného výboru pro programování (GAMM), která na společném zasedání s americkou ACM (Association for Computing Machinery) v roce 1958 vydala první verzi jazyka – ALOGOL 58. Velmi rychle byla zapracována řada připomínek a vylepšení a vydána verze ALGOL 60. Významný podíl na vývoji konečné verze ALGOL 68 měl svými pracemi, věnovanými operativní sémantice, holandský informatik Adriaan van Wijngaarden (1916 – 1987). Jazyk ALGOL se od konce 60. let dále nevyvíjel, nikdy nedosáhl tak masového rozšíření v praxi jako FORTRAN či COBOL, ale se stal obecně uznávaným standardem pro publikování algoritmů v odborných časopisech a zároveň byl východiskem a inspirací pro řadu dalších programovacích jazyků, např. Pascal, Simula 67 či Ada.

V roce 1961 přinesla titulní strana časopisu „Communications of the ACM“ (Vol. 4, No. 1) seznam 73 různých programovacích jazyků a v průběhu 60. let tento počet dál prudce rostl. Proto se budeme věnovat opět jen několika vybraným jazykům. Hlavním hlediskem pro výběr je rozšířenost jazyků v programátorské praxi.

Velmi populárním jazykem je pro svoji jednoduchost BASIC (Bigginer’s All-purpose Symbolic Instruction Code), který roku 1963 navrhli John Kemeny (1926 – 1992) a Thomas Kurtz (*1928) a který zažil obrovský rozmach v 80. letech jako základní programovací nástroj na osmibitových domácích počítačích a v současnosti zažívá comeback díky firmě Microsoft a jejím produktům Visual Basic, MS Office (makrojazyk VBA) a MS Internet Explorer (skriptovací jazyk VB Script).

Společnou nevýhodou FORTRANu a BASICu je fakt, že nenutí programátory k přehledným, dobře strukturovaným zápisům. Proto se i sám tvůrce programu může s určitým časovým odstupem špatně orientovat ve zdrojovém kódu, který vypracoval. To spolu s otázkou přesného popisu struktury dat, která pak usnadní psaní algoritmu, motivovalo profesora Niklause Wirtha (*1934) z Vysoké školy technické v Curychu k vytvoření programovacího jazyka Pascal. První implementaci Pascalu realizoval Niklaus Wirth v roce 1970.

Popularizaci jazyka prospěla levná a přitom kvalitní implementace Turbo Pascal, díky které se masově rozšířil na osobních počítačích typu IBM PC. Ta vznikla ve firmě Borland

v roce 1983 a hlavní zásluhy na jejím vývoji mají dánský programátor Anders Hejlsberg (*1960) a francouzský informatik Philippe Kahn (*1962). Jazyk Pascal je dodnes součástí vývojového prostředí Delphi. Kromě toho existují open source alternativy Free Pascal a Lazarus. Niklaus Wirth navrhl několik dalších programovacích jazyků (Algol W, Modula, Oberon) a je autorem vynikající, dodnes používané učebnice programování Algorithms + Data Structures = Programs.

Dalším významným jazykem začátku 70. let byl v roce 1972 programovací jazyk C. Jeho autorem je Denis Ritchie (*1941). Referenční příručku „The C Programming Language“ napsal v roce 1978 společně s Brianem Kernighanem (*1941). Popsaná verze se stala de facto standardem K&R jazyka C.

Na přelomu 60. a 70. let se objevují také první objektově orientované jazyky. Nejprve jazyk Simula norských informatiků Ole-Johana Dahla (1931 – 2002) a Kristena Nigaarda (1926 – 2002), který sice nebyl masově rozšířený, ale svým přístupem k řešení problémů položil základy objektově orientovaného programování. Na práce norských průkopníků pak navázal americký počítačový vědec Alan Key (*1940) s jazykem Smalltalk, který svou důslednou objektovostí byl mimo jiné vhodný k programování grafických aplikací. Díky tomu se Allan Key stal také tvůrcem grafického uživatelského rozhraní se systémem překrývajících se oken, jaké známe z moderních operačních systémů pro osobní počítače. Zajímavé je, že Alana Keye kromě prací norských předchůdců ovlivnil také Seymour Papert (autor jazyka LOGO), který ho přivedl ke studiu konstruktivismu.

Významným procedurálním jazykem je z historického hlediska rovněž Forth. Vyvinul jej roku 1970, nejprve pro svoji osobní potřebu Charles Moore (*1938) a použil jej pro ovládání soustavy radioteleskopů v Národní radioastronomické observatoři (NRAO) na hoře Kitt Peak v Arizoně. O jazyk však měli zájem další programátoři a rychle se rozšířil, i když ne tak masově jako výše uvedené jazyky. Roku 1973 založil Charles Moore společnost Forth inc. a začal jazyk nabízet komerčně. Roku 1976 se jazyk stal standardem programování v Mezinárodní astronomické unii. Dodnes je udržován a dále rozvíjen. Forth je typický prací se zásobníky a použitím postfixové notace. Nepotřebuje pro svoji práci operační systém a hodí se pro programování řídících a měřicích systémů, založených na mikroprocesorech s kratší délkou slova (dnes 8, 16 bit), tedy ve vestavěných (angl. embedded) systémech a pro práci v reálném čase.

Již jsme se zmínili, že John Backus (1924 – 2007) podnítil zájem o funkcionální programování. Přestože byl autorem komerčně velmi úspěšného procedurálního jazyka FORTRAN, nepovažoval procedurální přístup za jediný možný. Při převzetí Turingovy ceny v roce 1977 přednesl přednášku s názvem Může být programování osvobozeno od von Neumannova stylu?, v níž představil projekt jazyka FP. Tento projekt byl dokončen a distribuován, na rozdíl od pozdějšího FL (Function Level), který zůstal na úrovni firemního projektu IBM a distribuován nebyl. Podnítil však zájem o směr deklarativního programování, kterému říkáme funkcionální.

Moderním funkcionálním jazykem je Miranda, navržená v roce 1985 britským informatikem Davidem Turnerem (*19..) a její následník Haskell vyvinutý v roce 1990 skupinou informatiků a dále zdokonalovaný v současnosti. Jazyk Haskell je pojmenován po matematikovi, který vybudoval matematické základy funkcionálního programování. Byl to Haskell Brooks Curry (1900 – 1982), zakladatel kombinatorické logiky.

Mnohem starším funkcionálním jazykem je LISP (LISt Processing), který pochází z roku 1960. Jeho autorem je John McCarthy (1927), který je známý také svým výrazným podílem na vymezení pojmu umělá inteligence. V oblasti umělé inteligence je LISP stále využíván. Má jednoduchou strukturu, protože nerozlišuje mezi kódem a daty. Na vše se dívá jako na seznamy. Problematická je nutnost používání velkého množství závorek. Staví na něm některé systémy pro zpracování jazyků, ať přirozených (editor textů Emacs), či formálních (systém počítačové algebry Maxima). Vychází z něj další programovací jazyky (Smalltalk, Scheme) a dokonce i dětský programovací jazyk Logo.

Roku 1972 navrhl francouzský informatik Alain Colmerauer (*1941) deklarativní, neprocedurální jazyk Prolog, založený na predikátové logice prvního řádu. Deklarativní jazyk je přímým opakem procedurálního, to znamená, že popíšeme výchozí situaci (zadání) a cíl výpočtu, ale vůbec neříkáme, jakým postupem by se program měl k výsledku dostat. Počítač vlastně výsledek vyvozuje ze zadání pomocí pravidel formální matematické logiky. Proto mluvíme o logickém programování.

Novějším logickým programovacím jazykem je jazyk Gödel, který roku 1992 vytvořila dvojice informatiků John Lloyd a Patricia Hillová. Byl nazván na počest rakouského logika, matematika a filozofa, brněnského rodáka Kurta Gödla (1906 – 1978).

1.2 Paradigma objektového programování

Strukturované programování nebo také strukturovaný programovací jazyk označuje programovací techniku, kdy se implementovaný algoritmus rozděluje na dílčí úlohy, které se spojují v jeden celek. Objektově orientované programování představuje v dnešní době nejrozšířenější metodu vytváření programů oproti klasické metodě strukturovaného programování. Metody OOP napodobují vzhled a chování objektu z reálného světa s možností velké abstrakce. Základními programovacími prvky v OOP jsou útvary zvané objekty. Tyto útvary v zásadě modelují objekty reálného světa: osoby, předměty, dokumenty, události. Každý objekt je nositelem jistých informací o sobě samém (tzv. stavu) a má schopnost na požádání tento stav měnit. Například objekt faktura z podnikového informačního systému má svoje číslo a platební informace. Svůj stav může změnit, například z neproplacené faktury se může stát faktura proplacená apod.

V objektově orientovaném paradigmatu je program strukturalizován nikoli podle procedur, které se vykonávají, ale podle objektů, které se v systému vyskytují. Objekt v sobě zapouzdruje jak data, tak procedury pracující nad těmito daty. Průběh výpočtu je pak určen

posíláním zpráv mezi jednotlivými objekty. Pomocí zpráv objektům říkáme, jak mají změnit svůj stav. Na zaslanou zprávu oslovený objekt nějak reaguje. Tuto reakci definujeme v kódu, který bývá označován jako metoda. Programátoři proto většinou neříkají, že objekt posílá druhému objektu zprávu, ale že volá jeho metodu. Činnost metody, tj. reakce osloveného objektu na zprávu, záleží na tom, kým a jak byla zpráva odeslána, a na tom, v jakém stavu se objekt v okamžiku obdržení zprávy nacházel.

Víme, že objekty se v okolním světě často opakují. Třídu můžeme chápat jako šablonu, která definuje, jak se budou vytvářet objekty, které označíme jako instance dané třídy. Tato šablona definuje, jaké budou mít její instance atributy a jaké budou mít metody.

Shodnost atributů jednotlivých instancí ale neznamená shodnost jejich hodnot. Můžeme např. definovat třídu Auto, jejíž instance budou mít atribut motor. Každá instance však bude mít svůj vlastní motor a tyto motory se navíc mohou vzájemně lišit výkonem, spotřebou, provedením apod. Třída ale může definovat i vlastní atributy a metody a ty pak její instance sdílejí. Když instance změní hodnotu některého svého atributu, ostatní instance se o tom nedozvědí. Pokud však změní hodnotu atributu třídy, budou to hned všechny instance vědět, protože s ní tento atribut sdílejí.

Při vývoji objektově orientovaných programů se klade velký důraz na to, aby jednotlivé části programu nemohly využívat své „znalosti“ o tom, jak je protější část naprogramována. U všech entit (objekty, třídy, metody atd.) se proto rozlišují dvě charakteristiky:

- Rozhraní, které definuje, co o dané entitě ví okolní program.
- Implementace, která specifikuje, jak je dosažené správné funkčnosti dané entity.

Jedním z nejdůležitějších pravidel správného objektově orientovaného programování je zásada programovat proti rozhraní a ne proti implementaci. Moderní programovací jazyky se snaží neponechávat dodržování této zásady pouze na programátorovi, ale snaží se její dodržení kontrolovat a případně také vynucovat.

Se skrýváním implementace souvisí nejdůležitější rys objektově orientovaného programování, kterým je zapouzdření. Zapouzdřením označujeme dvě věci:

- Umístění dat (atributů) a kódu (metod), který s těmito daty pracuje, pohromadě do definice třídy. Programátor tak má přehled o vlastnostech a stavu zpracovávaných dat a dělá proto mnohem méně chyb.
- Znemožnění ostatním částem programu manipulovat s těmito daty jakýmkoliv jiným způsobem, než prostřednictvím metod vlastníka těchto dat, tj. metod příslušného objektu. Jinými slovy: k datům objektu je možné přistupovat pouze způsobem definovaným v rozhraní daného objektu.

Důsledné zapouzdření všech částí programu dramaticky zvyšuje efektivitu vývoje i spolehlivost výsledného programu. Všechny další rysy jazyka jsou proto často posuzovány podle toho, nakolik podporují nebo naopak narušují optimální zapouzdření.

Jednou z konstrukcí, které nabízí většina objektových programovacích jazyků, je možnost definice dědičnosti. Dědičnost představuje speciální druh skládání, při němž je do objektu vložen jiný objekt, tzv. podobjekt předka, a nový majitel tohoto vloženého objektu přebírá jeho rozhraní a případně k němu přidává své vlastní rysy. Vložený objekt, resp. jeho třída, je pak označen za předka (odtud také název podobjekt předka) a jeho majitel za potomka. Dědičnost přináší elegantní způsob, jak si ušetřit programování. Potomci totiž dědí všechny dostupné metody svých předků, takže jim stačí, když sami definují pouze ty, jejichž definice předka jim nevyhovuje, a ty, které předek vůbec nemá. Dědičnost je však typickou konstrukcí, která narušuje zapouzdření, a to v obou jeho rysech:

- Potomkům bývá často umožněno pracovat primo s daty svého předka, takže přestává platit, že kód je blízko zpracovávaných dat se všemi z toho plynoucími důsledky.
- Předek musí často prozradit potomkovi něco o své implementaci, protože jinak by potomek nebyl schopen definovat své metody bezchybně. To opět zvyšuje pravděpodobnost chyb.

Obecná zásada proto říká, že dědičnost máme použít pouze tehdy, když jakékoliv jiné řešení je výrazně těžkopádnější. Kromě toho platí další zásada: má-li program zůstat stabilní, musí být potomek vždy speciálním případem předka.

Jak jsme již uvedli v souvislosti se zapouzdřením, v objektově orientovaném programování se důsledně odděluje rozhraní a implementace. Programovací jazyk Java dokonce zavedl speciální konstrukci interface, která definuje pouze rozhraní bez jakékoliv implementace. Interface své metody pouze deklaruje, avšak nijak je neimplementuje. To ponechává na třídách, které se přihlásí k jeho implementaci. Takové třídy pak mohou své instance vydávat za instance implementovaného interface. Protože interface nedefinuje žádnou implementaci, nemůže mít ani žádné instance. Požaduje-li některá část programu instancí interface, musí ji zastoupit instance nějaké třídy, která dany interface implementuje.

Mohli bychom říci, že implementovaný interface se chová podobně jako předek tedy že implementující třída také přebírá jeho rozhraní. Protože však instance implementující třídy nepřebírají žádný podobjekt předka, nemůže dojít ani k jednomu z obou výše uvedených problémů, s nimiž se setkáváme u dědičnosti.

1.3 Jazyk C#

Jazyk C# je vysokoúrovňový objektově orientovaný programovací jazyk vyvinutý v roce 2002 firmou Microsoft zároveň s platformou .NET Framework, později schválený standardizačními komisemi ECMA (ECMA-334) a ISO (ISO/IEC 23270). Jazyk C# je založen na jazycích C++ a Java a je tedy nepřímým potomkem jazyka C, ze kterého čerpá syntaxi.

Název jazyka C# (vyslovované anglicky jako C Sharp, /si: ša:p/) je odvozen z hudební notace, kde křížek označuje zvýšení noty o půl tónu a v tomto případě by označoval notu

cis, tedy C zvýšené o půl tónu. Podobně vznikl název jazyka C++ jako zlepšení jazyka C: „++“ totiž v syntaxi jazyka C znamená zvýšení hodnoty proměnné o 1. Křížek na počítačové klávesnici (#) a křížek v hudební nauce (♯) jsou dva odlišné znaky. Pro zápis názvu jazyka C Sharp se nepoužívá znak hudebního křížku z technických důvodů, protože tento se na standardní klávesnici nevyskytuje, ale pro zjednodušení se používá klasický křížek. Toto je zakotveno ve specifikaci jazyka C#, ECMA-334. Toto opatření je spíše praktického rázu, takže v případech jako jsou různé marketingové materiály se stále často používá znak křížku z hudební notace.

Od doby svého vzniku se stal jazyk C# velmi populární a dnes ho můžeme používat za standard pro vývoj různých druhů aplikací, jako např. databázových aplikací, webových aplikací i služeb, formulářových aplikací i aplikací pro mobilní zařízení.

Jazyk C# podobně jako jazyk Java využívá pro svoji činnost takzvaný virtuální stroj. Zdrojový kód v jazyce C# je nejprve přeložen do tzv. mezikódu, kterému říkáme CIL (Common Intermediate Language). Jedná se v podstatě o strojový (binární) kód, který má ale o poznání jednodušší instrukční sadu a přímo podporuje objektové programování. Tento mezikód je potom díky jednoduchosti relativně rychle interpretovatelný tzv. virtuálním strojem (tedy interpretem, v případě .NET je to tzv. CLR - Common Language Runtime). Výsledkem je strojový kód pro náš procesor.

Ve spojitosti s programovacím jazykem C# se často setkáme s pojmem .NET Framework. .NET Framework je softwarová platforma poskytující širokou škálu prostředků pro programy. Její popis by zcela jistě vydal na celou řadu knih a není cílem této publikace ji detailně popisovat. Prostředky z této platformy budeme při programování v jazyce C# používat, např. pro výpis na konzoli.

1.4 Základy práce s Microsoft Visual Studiem

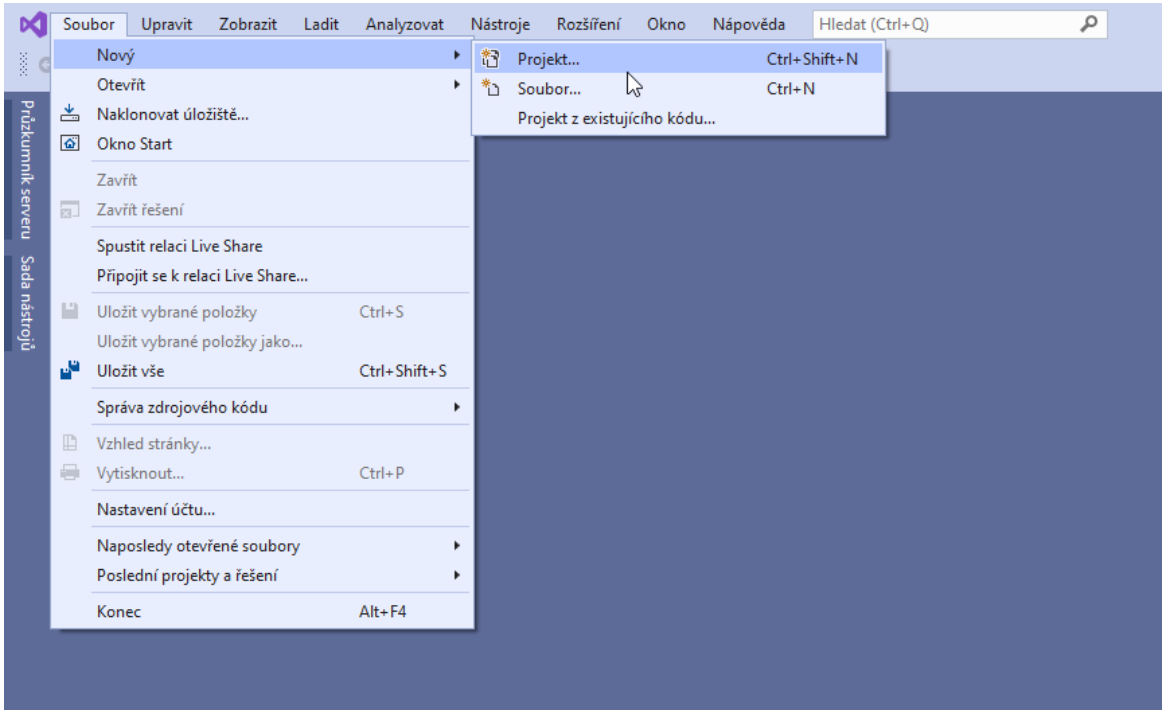
1.4.1 Instalace

Než začneme vytvářet náš první program je nezbytné si nainstalovat vývojové prostředí Microsoft Visual Studio. V tomto textu budeme používat verzi Microsoft Visual Studio 2019 Community Edition, která je k dispozici zdarma ke stažení zde: https://visualstudio.microsoft.com/vs/community/. Vývojové prostředí nainstalujte podle pokynů uvedených na webových stránkách a v instalačním průvodci vývojového prostředí Microsoft Visual Studio.

1.4.2 První program

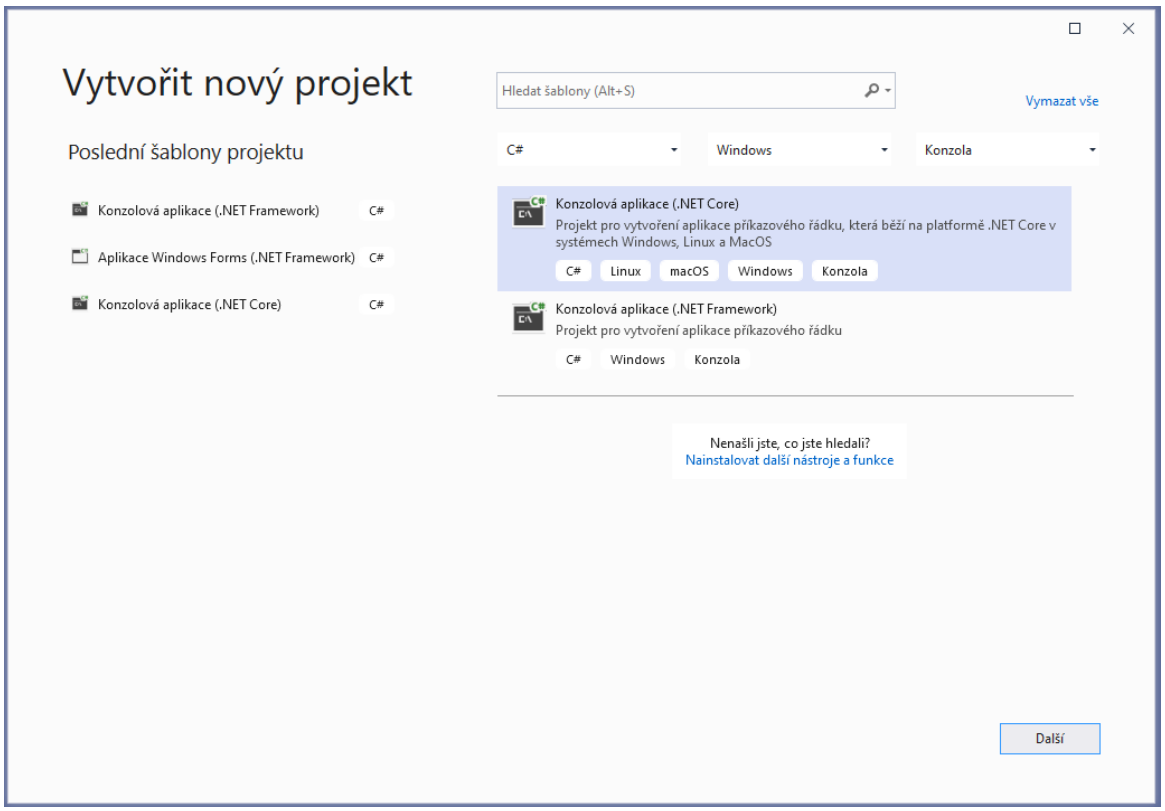
Pokud máte nainstalované vývojové prostředí Microsoft Visual Studio, se můžete pustit do vytvoření svého prvního programu v jazyce C#. Spusťte vývojové prostředí Microsoft

Visual Studio. Zobrazí se vám úvodní obrazovka, na které vyberte z hlavního menu možnost *Soubor → Nový → Projekt*.



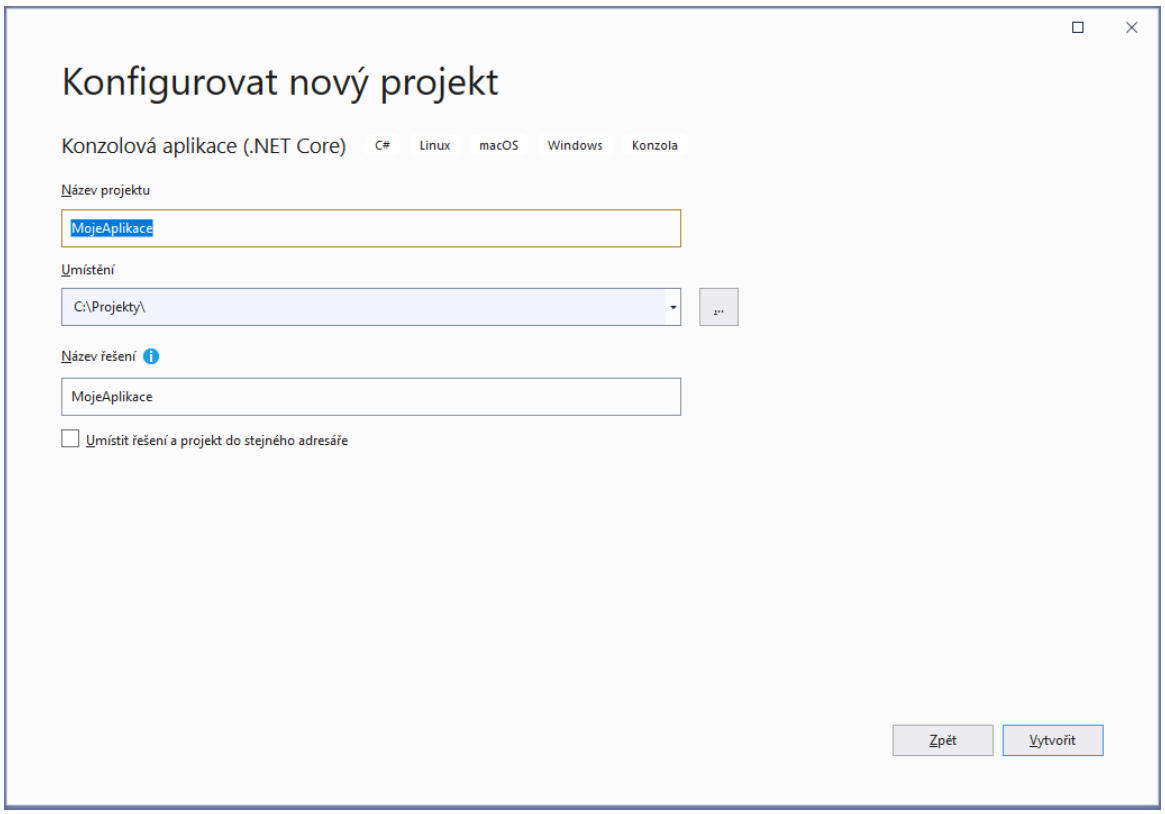
Obrázek 1.1: Nový projekt

Zobrazí se nám poměrně široká nabídka typu projektů, které můžeme filtrovat podle tří kritérií. V prvním kritériu si vybereme programovací jazyk, v němž chceme naši aplikaci vytvořit – zde zvolíme *C#*. Druhé kritérium určuje cílový operační systém – zvolme *Windows* a poslední položka je upřesnění typu aplikace, kde si vybereme položku *Konzola*. Po zvolení těchto kritérií se nám zobrazí dva typy projektů Konzolová aplikace (.NET Core) a Konzolová aplikace (.NET Framework). Pro naše účely si můžeme zvolit kteroukoliv z těchto dvou možností, přičemž .NET Core je novější, proto i my zvolíme tuto možnost.



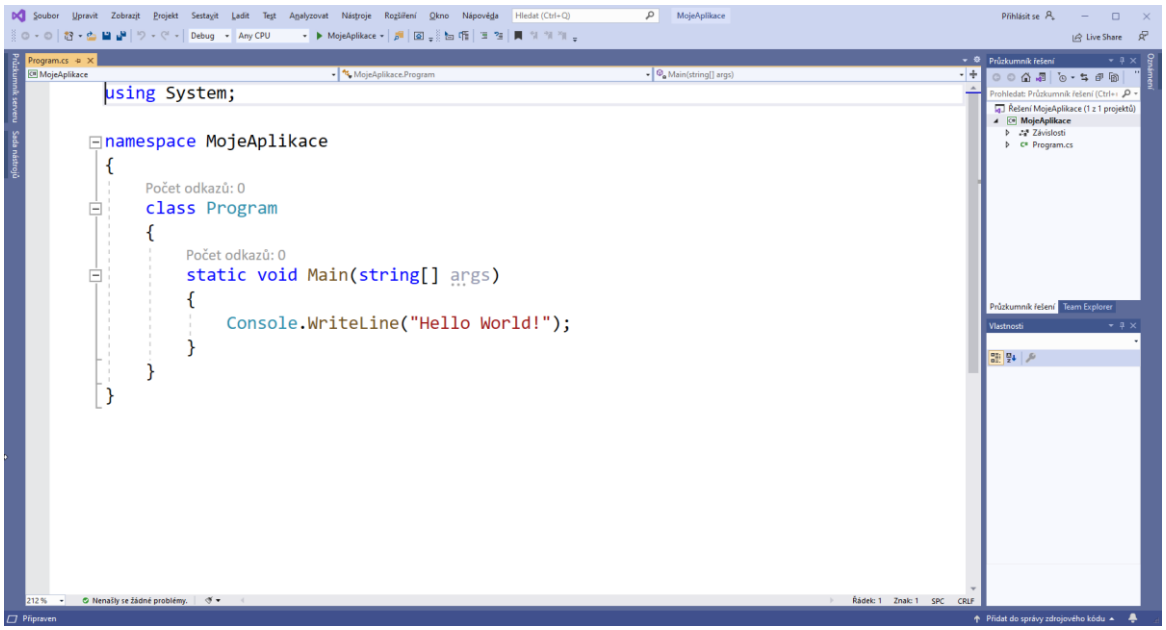
Obrázek 1.2: Typ projektu

V dalším kroku musíme vyplnit název našeho projektu a jeho umístění na disku.



Obrázek 1.3: Název projektu

Po kliknutí na tlačítko *Vytvořit* se nám vytvoří nový projekt. Dříve než se pustíme do vlastního programování řekneme si něco o rozložení oken v tomto prostředí. Vpravo nahoře se nachází okno Průzkumník řešení, které obsahuje seznam souborů, z nichž se projekt skládá. Vpravo dole je potom okno Vlastnosti, které zobrazuje vlastnosti aktuálně vybraného prvku v jiných oknech, např. úplnou cestu k vybranému souboru. Poslední okno, která zabírá největší plochu obrazovky, obsahuje zdrojový kód aktuálně vybraného souboru.

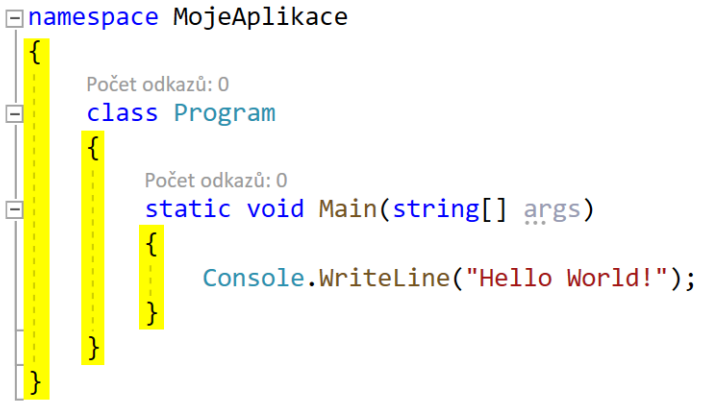


Obrázek 1.4: Prázdný projekt

Můžeme si všimnout, že zdrojový kód není úplně prázdný, ale obsahuje už předpřipravenou šablonu, která obsahuje základní strukturu programu, čímž je nám ušetřena práce psaním našeho programu úplně od píky.

Nyní si projděme základní strukturu programu. První řádek začínající klíčovým slovem *using* importuje jmenné prostory z knihoven, které chceme používat v našem programu. Těchto jmenných prostorů můžeme používat více, v takovém případě každý jmenný prostor zapíšeme na samostatný řádek vždy začínající klíčovým slovem *using*. Jmenný prostor můžeme chápat jako pojmenovanou část projektu.

Další část kódu začíná klíčovým slovem *namespace*, kterým definujeme svůj vlastní jmenný prostor. Ve výchozím stavu je název jmenného prostoru roven názvu projektu, ale můžeme ho přepsáním změnit. Pod tímto řádkem následuje znak levé množinové závorky „{“, která označuje začátek bloku kódu, v tomto případě začátek jmenného prostoru *MojeAplikace*. Každý blok kódu je vždy ukončen znakem pravé množinové závorky „}“, viz poslední znak ve zdrojovém kódu. Pro přehlednost je blok kódu, tj. sobě odpovídající množinové závorky zvýrazněn svislou přerušovanou čarou.



Obrázek 1.5: Blok kódu

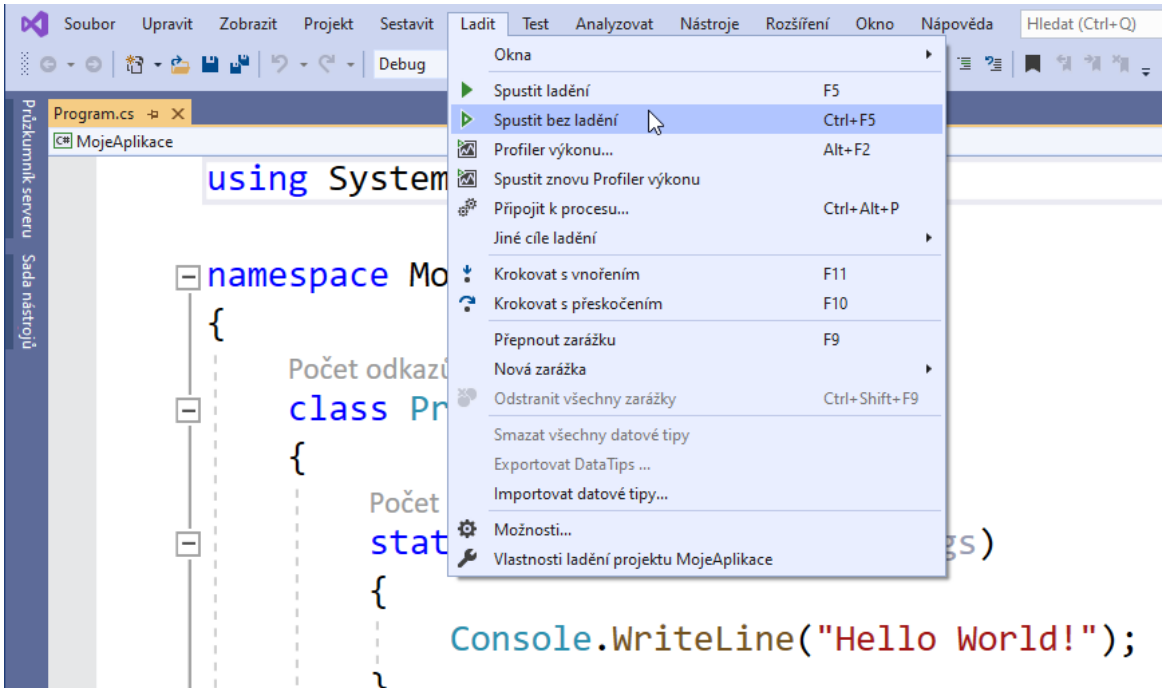
Uvnitř jmenného prostoru je potom definována třída *Program* pomocí klíčového slova *class*. Tento řádek označuje hlavní program našeho projektu. A znovu blok kódu patřící do hlavního programu je uzavřen v množinových závorkách. Poslední část kódu začínající *static void Main* definuje hlavní metodu, která je vstupním bodem programu, tj. při spuštění našeho projektu se začne provádět kód, který je zapsán v metodě *Main*. Jakýkoliv další kód v projektu se může spustit pouze tak, že je zavolán z této metody *Main*.

Všimněme si, že v metodě *Main* je již řádek

Console.WriteLine(“Hello World!”);

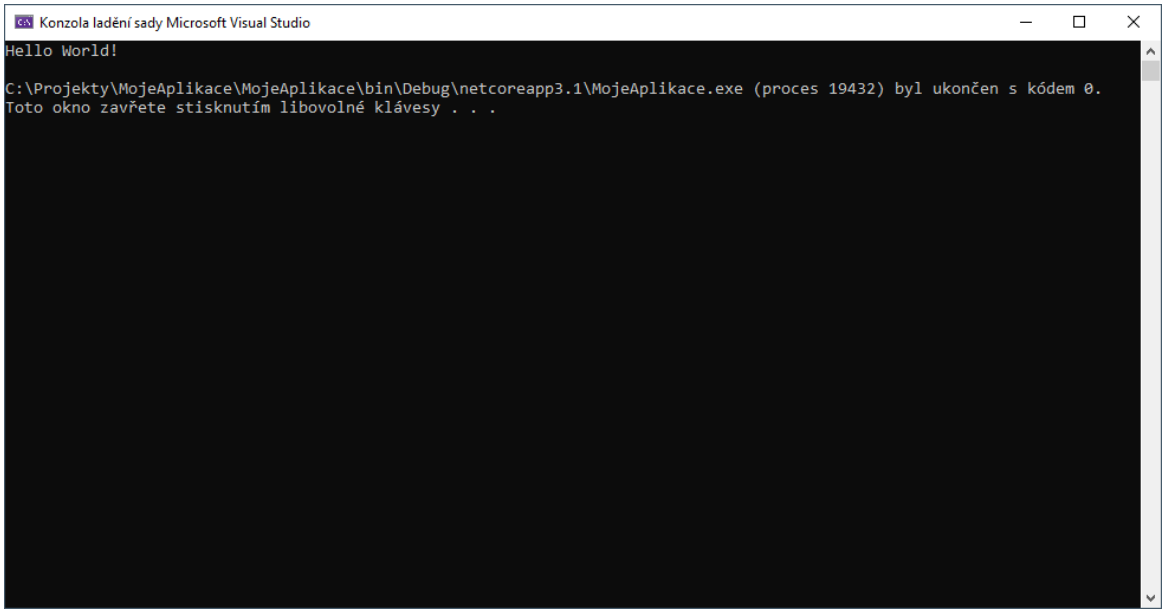
Tento řádek vypíše po spuštění programu na obrazovku text *Hello World!*, kterýžto se vžil pro prezentaci fungování každého programovacího jazyka. Při vytváření vlastní aplikace samozřejmě tento řádek můžeme smazat.

Nyní bychom chtěli náš program spustit. Toto provedeme pomocí menu *Ladit* → *Spustit bez ladění*.



Obrázek 1.6: Spuštění programu

Po potvrzení se nám objeví nové okno s tzv. konzolou, tedy oknem umožňující pouze textový výstup a na něm by měl být zobrazen text *Hello World!*



Obrázek 1.7: Výstup programu

Tímto jsme úspěšně vytvořili a spustili náš první program v jazyce C#. Vzhledem k tomu, že projekty budeme spouštět velmi často je vhodné zapamatovat si klávesovou zkratku *Ctrl + F5*.

1.4.3 KOMENTÁŘE

Pokud vytváříme nějakou jednoduchou aplikaci, je z kódu obvykle patrné, co daná aplikace dělá a jak funguje. Jiná je ovšem situace, pokud je náš program většího rozsahu, zvláště pokud na daném programu pracuje více programátorů. V takovém případě může být orientace v programu náročná, a proto je vhodné používat při psaní kódu komentáře. Komentáře jsou části kódu, které slouží pouze pro naši potřebu a překladačem jsou ignorovány. Existují dva základní druhy komentářů, a to jednořádkové a víceřádkové. Jednořádkové používáme tehdy, pokud je komentář krátký a vleze se nám na jeden řádek. Takový komentář začíná dvěma symboly lomítka //. Vše, co napíšeme za //, je překladačem ignorováno až na konec řádku. Pro přehlednost jsou komentáře v prostředí MS Visual Studia zobrazeny zeleně.

Pokud chceme okomentovat určitou část kódu podrobněji, je výhodné použít komentáře víceřádkové, které začínají dvojicí symbolů lomítka a hvězdičky /* a končí stejnou dvojicí symbolů, ale v opačném pořadí, tj. */. Zde můžeme vidět příklad jednořádkového a víceřádkového komentáře.

```
Počet odkazů: 0
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");  //Výpis textu na obrazovku

    /*
     * Toto
     * je
     * víceřádkový komentář
     */
}
```

Obrázek 1.8: Komentáře

Komentáře však mají i jiné využití než jen popis kódu. Často se využívají i pro dočasné odstranění či zablokování části kódu nebo při úpravách kódu, aby byl vidět původní kód.

1.4.4 LADĚNÍ

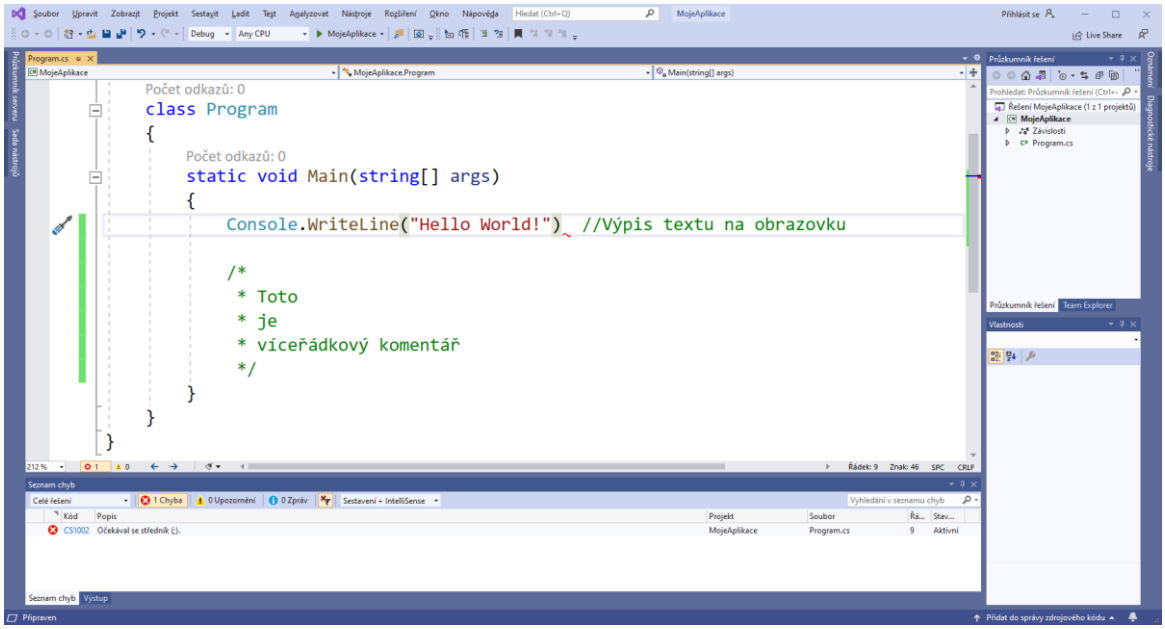
Mohlo by se zdát, že hlavní náplní práce programátora je psaní kódu. Ve skutečnosti tato činnost zabírá pouze menší část času. Mnohem časově náročnějším úkolem je hledání a odstraňování chyb. Rozlišujeme dva základní druhy chyb v programu, a to chyby syntaktické a sémantické.

Začněme chybami syntaktickými, které jsou mnohem jednodušší k nalezení i odstranění, protože MS Visual Studio nám oznámí nejen, že v programu jsou syntaktické chyby, ale dokonce nám oznámí i kde přesně v kódu se nacházejí a často i, jak je máme opravit,

byť tato rada nemusí být vždy správná. Typická syntaktický chyba je např. zapomenutý středník za příkazem. Zkusme si tuto chybu demonstrovat tak, že odstraníme středník za příkazem *Console.WriteLine*. Můžeme si všimnout, že na místě, kde byl středník se nám zobrazí červená vlnovka a při najetí myši na tuto vlnovku se nám zobrazí nápověda s popisem chyby.

Obrázek 1.9: Syntaktická chyba

Pokud bychom si této červené vlnovky nevšimli a pokusili se náš program spustit, zobraz se nám v dolní části obrazovky okno Seznam chyb, kde jsou zobrazeny všechny syntaktické chyby v programu.



Obrázek 1.10: Seznam chyb

Pokud klikneme na popis chyby v seznamu chyb přesune se nám kurzor na místo, kde se daná chyba nachází. Pokud je v programu více chyb, vždy odstraňujeme chyby v pořadí,

v jakém jsou uvedeny v seznamu chyb. Po opravení všech syntaktických chyb můžeme program spustit.

Druhým typem chyb jsou chyby sémantické, které se vyznačují tím, že program jde spustit, ale jeho výstup je jiný, než jsme očekávali. Abychom si mohli demonstrovat tuto chybu, musíme náš úvodní program mírně rozšířit a to tak, že si budeme definovat celočíselnou proměnnou *i* s výchozí hodnotou 5. Poté k této proměnné přičteme číslo 4 a následně tuto proměnnou *i* ještě vynásobíme číslem 3. Nakonec vypíšeme obsah proměnné *i* na obrazovku. Výsledný kód bude vypadat takto.

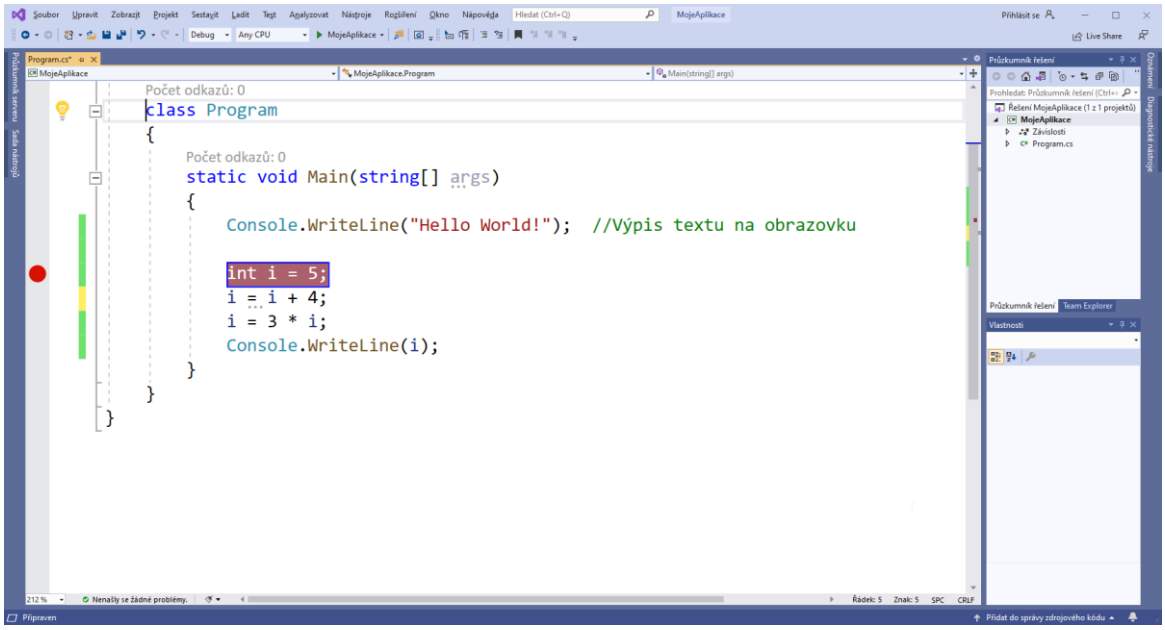
```
Počet odkazů: 0
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");  //Výpis textu na obrazovku

    int i = 5;
    i = i + 4;
    i = 3 * i;
    Console.WriteLine(i);
}
```

Obrázek 1.11: Syntaktická chyba

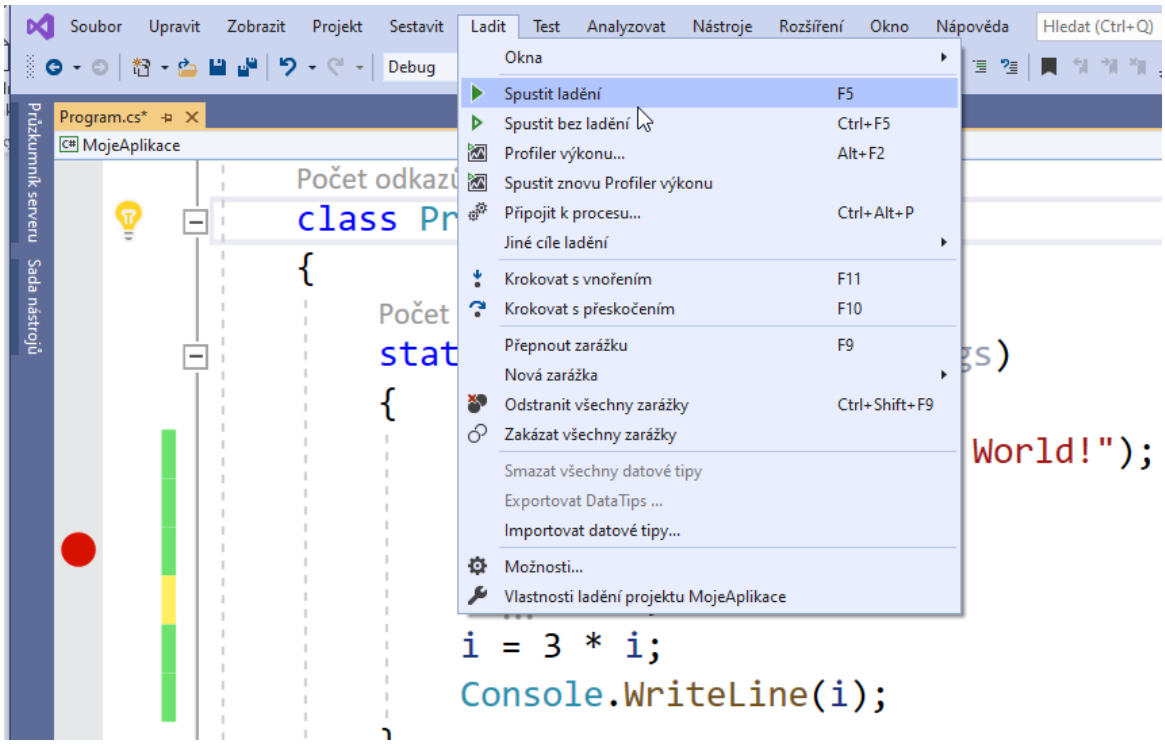
Po spuštění programu se nám na výstupní konzoli zobrazí hodnota 27, ovšem my jsme očekávali, že by výstupem mělo být číslo 18. Z výš uvedeného jednoduchého kódu je zřejmé, jakým způsobem jsme dostali výsledek 27 a jak můžeme kód opravit, aby výsledkem bylo číslo 18. Ovšem v praxi bude kód mnohem delší a komplikovanější, a tudíž na první pohled nebude zřejmě, proč nám program poskytuje jiný výsledek, než jsme očekávali.

Obecný postup pro hledání sémantických chyb se nazývá *ladění* nebo *debugging*. Základem je vložit do programu bod *přerušení* nebo též *breakpoint*. Nejprve ale musíme přibližně odhadnout, kde se může příslušná chyba nacházet. V našem případě to bude někde v místě, kde se pracuje s proměnnou *i*, ale nevíme, kde přesně, takže vložíme bod přerušení na místo prvního výskytu proměnné *i*. Samotný bod přerušení vložíme do kódu kliknutím do levého šedého pruhu u příslušného řádku, čímž se nám příslušný řádek zabarví červeně a v levém šedém pruhu se zobrazí červené kolečko.



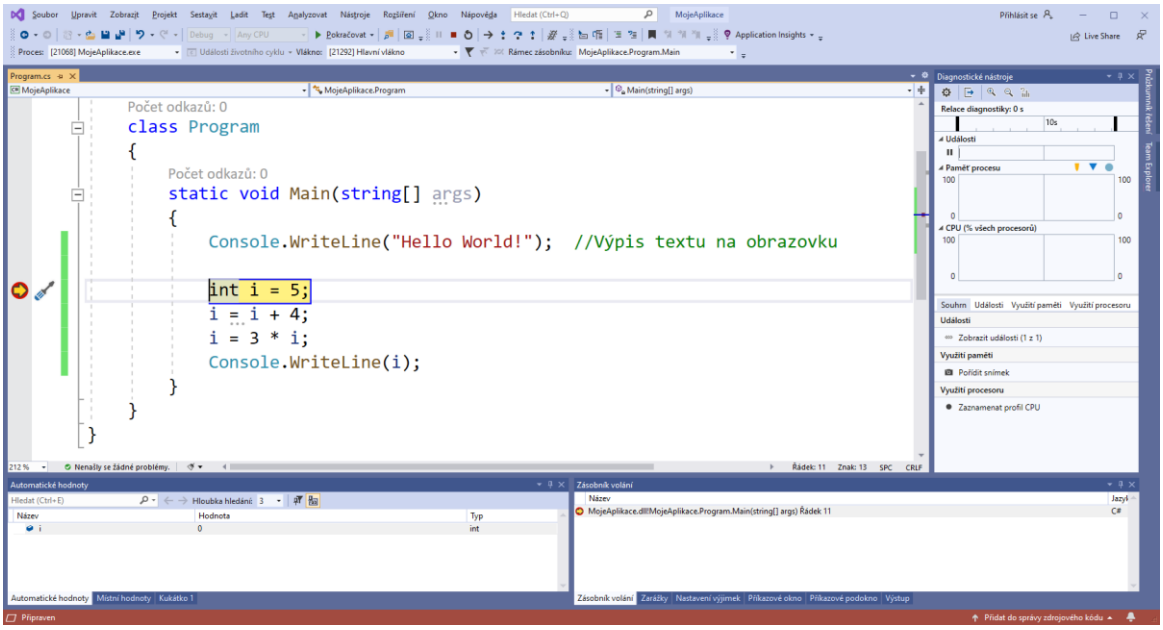
Obrázek 1.12: Bod přerušení

Pokud nyní spustíme program stejným způsobem, jako jsme to dělali doposud, celý program proběhne a ihned skončí. Abychom spustili ladění programu, musíme v menu zvolit *Ladit → Spustit ladění*.



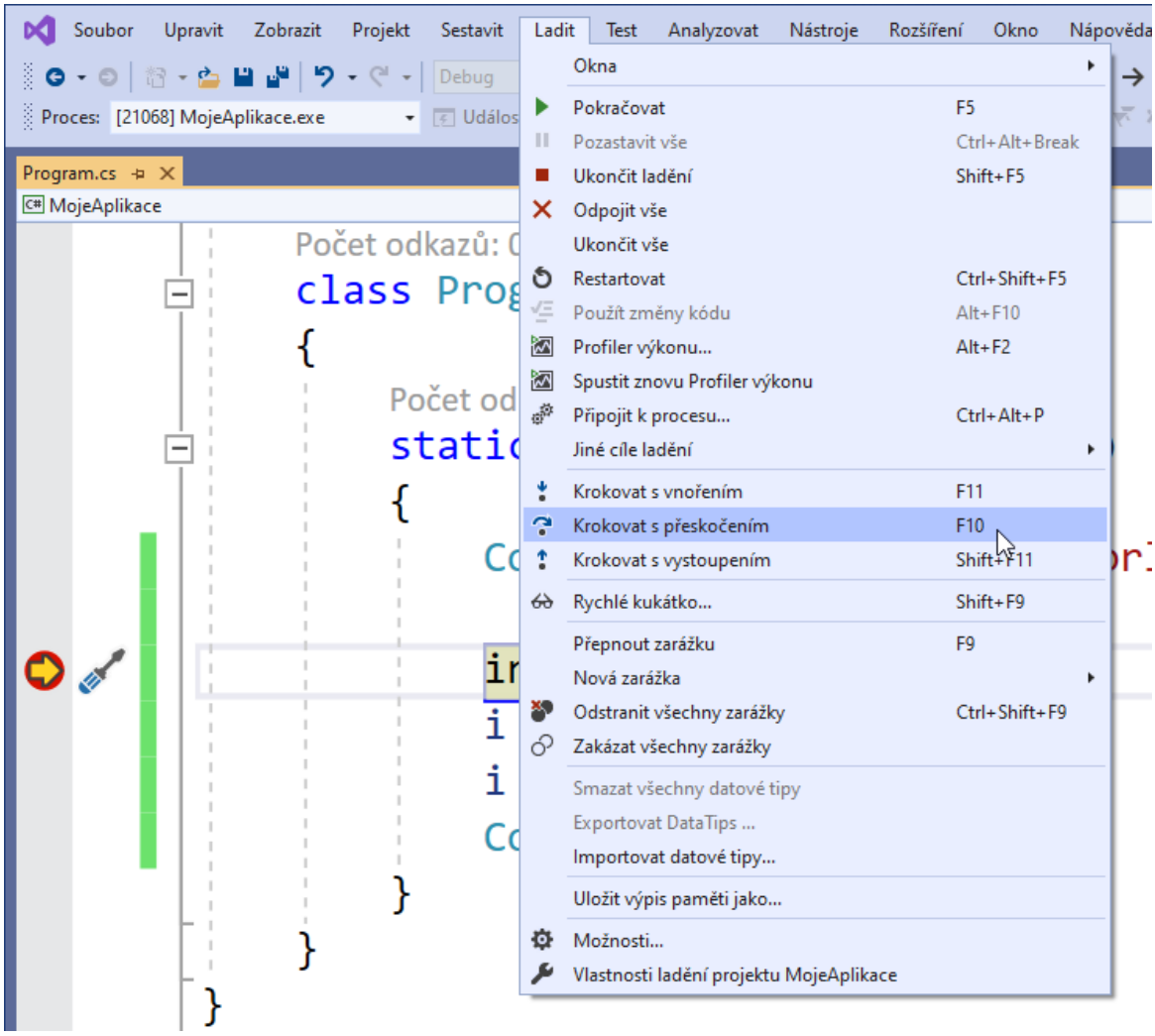
Obrázek 1.13: Ladění

Nyní se nám běh programu přeruší v nastaveném bodě přerušení, což poznáme tak, že se řádek, kde se přerušil běh programu, zvýrazní žlutě.



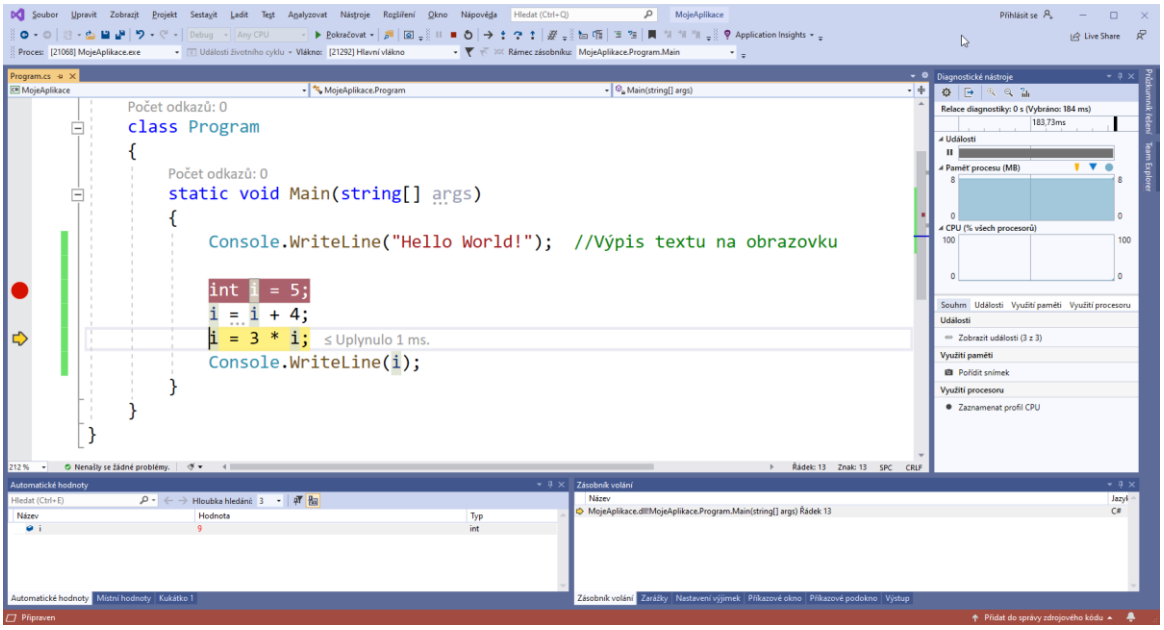
Obrázek 1.14: Ladění

Zároveň si můžeme všimnout, že se nám změnilo rozložení oken tak, aby se zobrazily informace důležité pro ladění. V pravém horním okně *Diagnostické nástroje* můžeme vidět, kolik náš program zabírá paměti, či kolik spotřebovává procent procesoru. V pravém dolním okně *Zásobník volání* můžeme vidět, z jaké metody byl zavolán aktuální příkaz. Nás však bude v tuto chvíli nejvíce zajímat levé dolní okno *Automatické hodnoty*, které nám zobrazuje hodnoty proměnných v aktuálním kontextu. V našem programu je definována pouze jedna proměnná i, která je rovněž zobrazena v tomto okně. Ve sloupci Hodnota můžeme vidět, že její hodnota je 0, je tomu tak proto, že příkaz přiřazení $i = 5$ ještě nebyl vykonán. Pomocí menu *Ladit* → *krokovat s přeskočením* můžeme vykonat jeden příkaz za bodem přerušení.



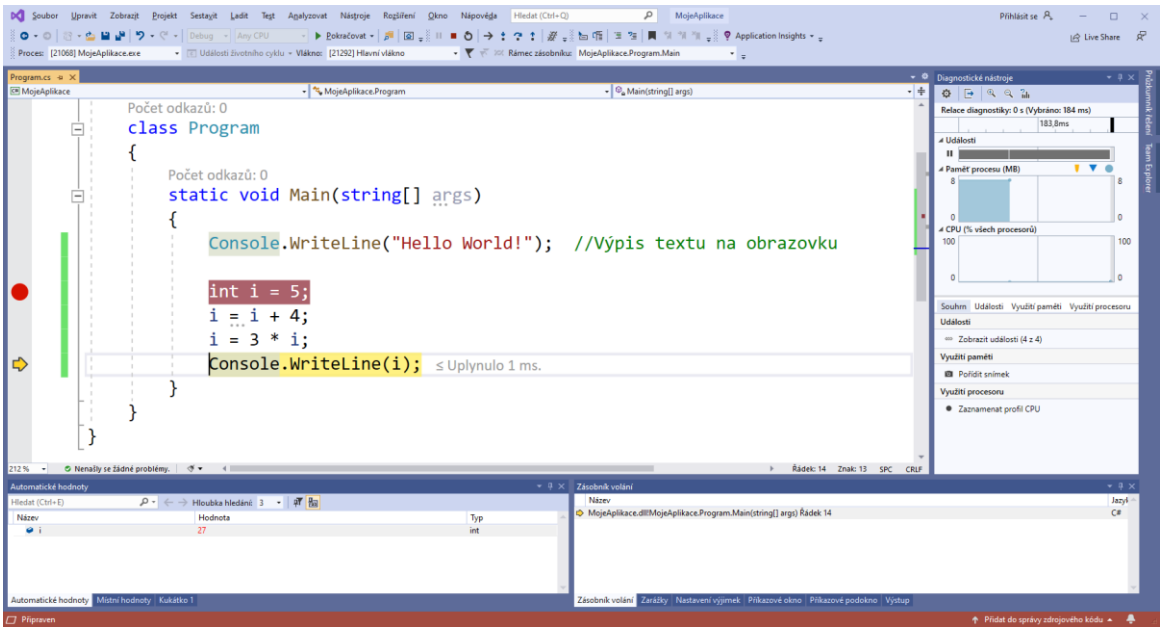
Obrázek 1.15: Krokování 1

Vykonáním příkazu *int i = 5* došlo přiřazení hodnoty 5 do proměnné *i*, což můžeme vidět v okně Automatické hodnoty. Všimněte si rovněž, že hodnota 5 je zobrazena červeně, čímž je zvýrazněno, že v daném kroku došlo ke změně hodnoty této proměnné. Toto zejména oceníme, pokud je v daném kontextu definováno více proměnných. Zároveň si můžeme všimnout, že je nyní zvýrazněn následující řádek, což nám ukazuje, který řádek bude vykonán v následujícím kroku. Nyní provedeme znovu krokování s přeskočením buď pomocí menu nebo klávesou *F10*.



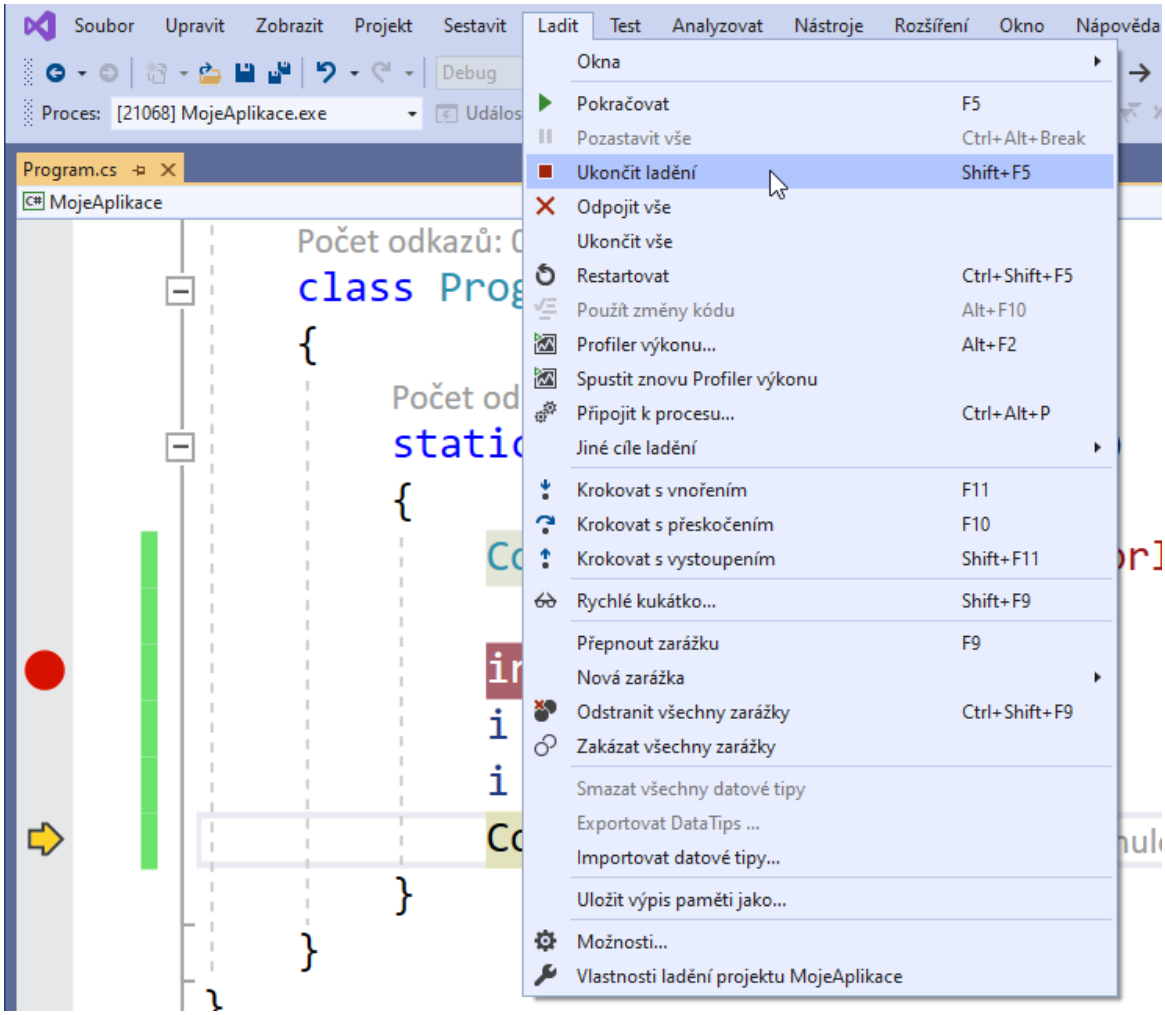
Obrázek 1.16: Krokování 2

Tímto jsme vykonali příkaz přiřazení i = i + 4. V okně Automatické hodnoty vidíme, že hodnota proměnné i je nyní 9. Teď ještě jednou provedeme krokování s přeskočením.



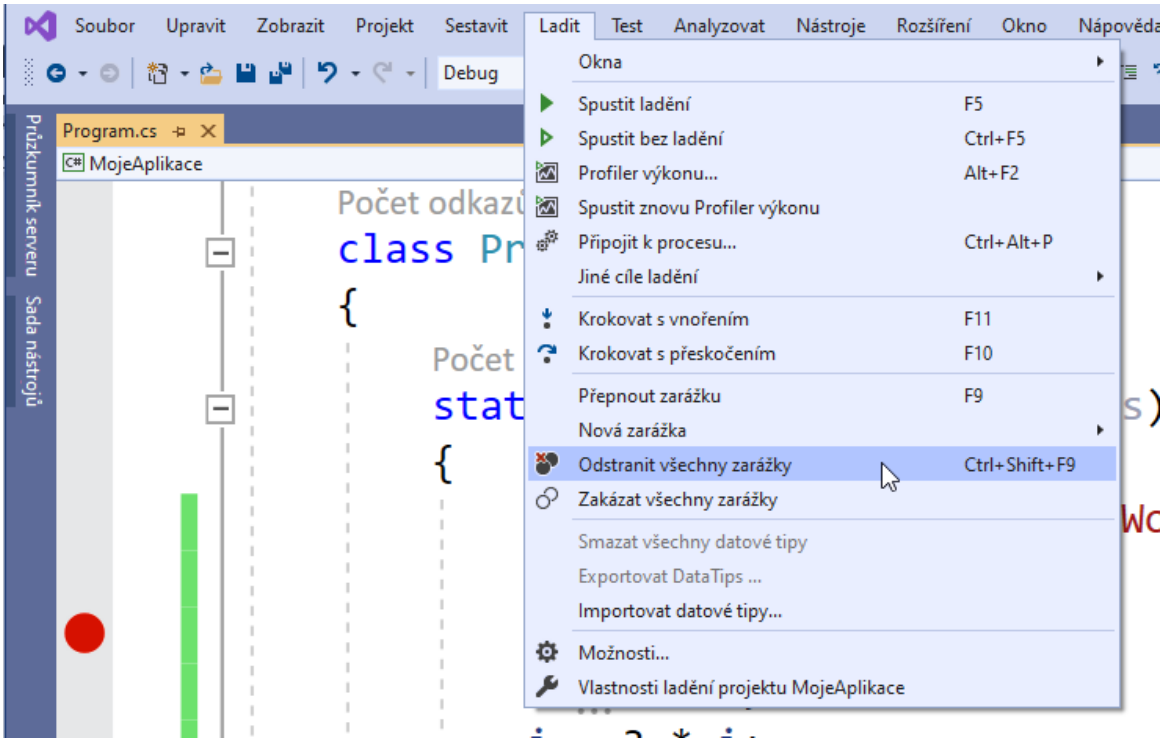
Obrázek 1.17: Krokování 3

Provedením dalšího příkazu vidíme, že nyní je hodnota proměnné *i* 27. Jelikož by správně měla být nyní hodnota proměnné *i* 18, zjistili jsme, že v posledním řádku, který jsme vykonali je chyba a místo násobení číslem 3 by tam mělo být násobení číslem 2. Můžeme tedy ukončit ladění pomocí menu Ladit → Ukončit ladění.



Obrázek 1.18: Ukončení ladění

Po ukončení ladění můžeme program editovat a opravit chybu. Na závěr je ještě vhodné odstranit všechny body přerušení, což provedeme opětovným kliknutím na červené kolečko v levém šedém pruhu nebo v menu *Ladit → Odstranit všechny zarážky*.



Obrázek 1.19: Odstranění bodu přerušení

OTÁZKY

1. Jazyk C# vychází z jazyka
    a. Basic
    b. C
    c. Prolog
2. Jazyk C# je
    a. Kompilovaný jazyk
    b. Interpretovaný jazyk
    c. Jazyk s virtuálním strojem
3. Pro komentáře v jazyce C# používáme symbol
    a. //
    b. \\
    c. –
4. Pokud je v programu syntaktická chyba potom
    a. Program lze spustit, ale výstup programu je jiný než očekávaný.
    b. Při pokusu o spuštění programu se MS Visual se zhroutí.
    c. Program nelze spustit a MS Visual Studio nám zobrazí seznam syntaktických chyb v programu.
5. Pokud je v programu sémantická chyba potom
    a. Program lze spustit, ale výstup programu je jiný než očekávaný.

   b. Při pokusu o spuštění programu se MS Visual se zhroutí.
   c. Program nelze spustit a MS Visual Studio nám zobrazí seznam syntaktických chyb v programu.
6. Sémantické chyby obvykle hledáme pomocí
   a. Křišťálové koule
   b. Ladění
   c. Doplnění středníku
7. Místo v programu, kde se zastaví provádění programu při ladění se nazývá
   a. Podmínka
   b. Operand
   c. Bod přerušení
8. Proces, při kterém postupně vykonáváme jednotlivé příkazy se nazývá
   a. Krokování
   b. Spuštění
   c. Inspekce
9. Který příkaz vypíše na obrazovku text?
   a. Console.Writeline
   b. Console.writeline
   c. Console.WriteLine
10. Jak se v jazyce C# ukončuje příkaz?
   a. Novým řádkem
   b. Středníkem
   c. Tečkou

SHRNUTÍ KAPITOLY

Tato kapitola obsahovala základy programování. Seznámili jsme se se stručnou historií programování, dále jsme si ukázali, jak nainstalovat Microsoft Visual Studio a konečně jsme vytvořili svůj první program v jazyce C#. Pro lepší čitelnost jsme si ukázali, jak náš kód okomentovat. Nakonec jsme se podívali na rozdíl mezi syntaktickými a sémantickými chybami a zejména jsme si ukázali, jak chyby v programu najít a odstranit.

ODPOVĚDI

1. b
2. c
3. a
4. c
5. a

6. b
7. c
8. a
9. c
10. b

2 OBJEKTY A ZAPOUZDŘENÍ

RYCHLÝ NÁHLED KAPITOLY

Tato kapitola je zaměřena na základy objektového programování. Seznámíme se se základními pojmy, jako je třída, objekt, instance či konstruktor. Podrobněji se potom podíváme, jak můžeme ochránit vnitřní stav objektu pomocí zapouzdření. Uvedeme si různé modifikátory přístupu a způsoby nastavování proměnných pomocí metod a vlastností. Na závěr se zmíníme o statických metodách a proměnných.

CÍLE KAPITOLY

Po prostudování této kapitoly budete umět:

- Vytvořit třídu a její instanci.
- Deklarovat proměnné a metody uvnitř třídy.
- Používat konstruktory.
- Říci, kdy dojde k uvolnění objektu z paměti.
- Vyjmenovat základní modifikátory přístupu.
- Používat vlastnosti pro přístup k zapouzdřeným proměnným.
- Co jsou to statické metody a proměnné.

KLÍČOVÁ SLOVA KAPITOLY

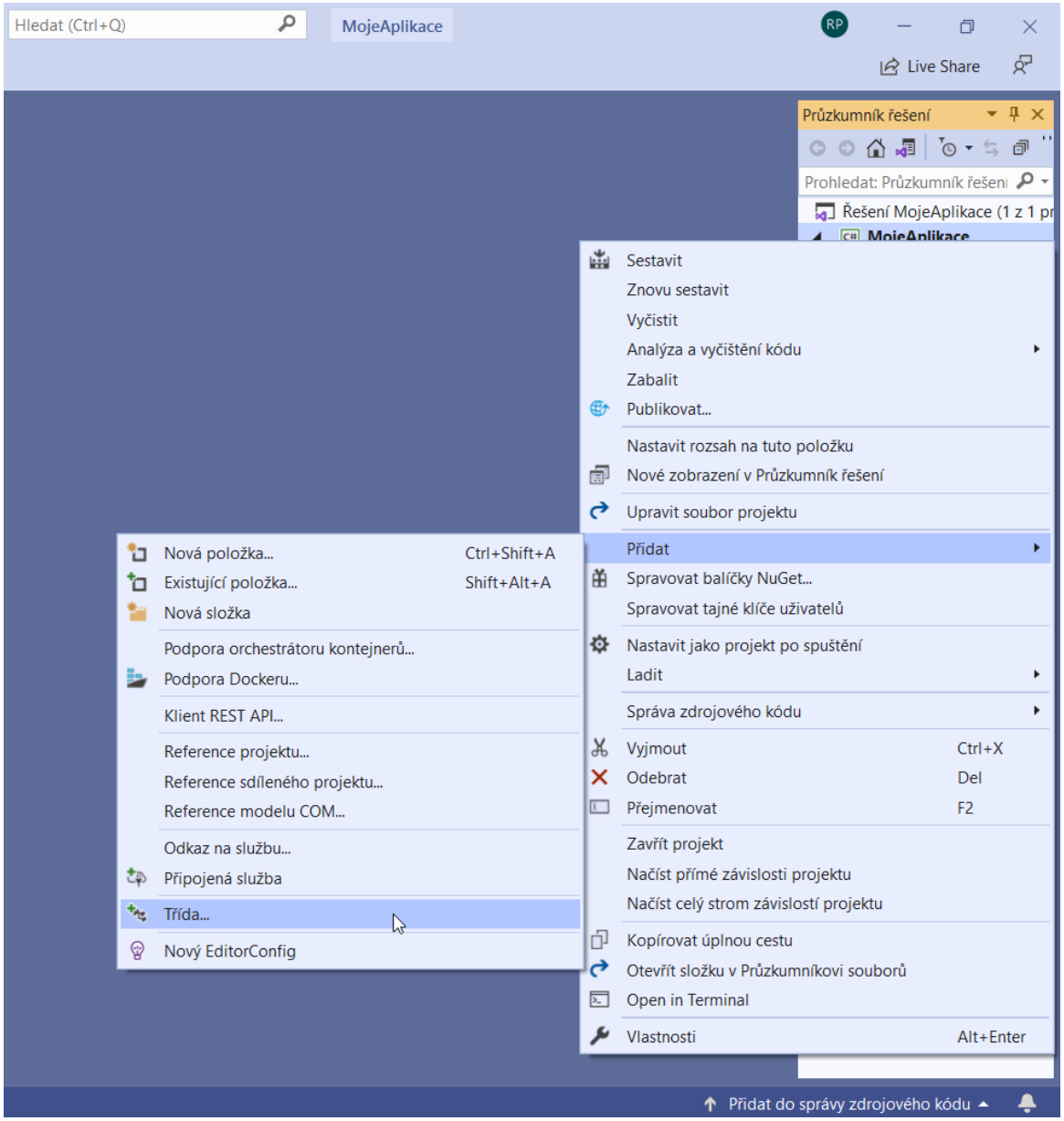
Třída, objekt, instance, metoda, proměnná, konstruktor, destruktor, zapouzdření modifikátor přístupu, vlastnost, statická metoda, statická proměnná.

2.1 Třídy a objekty

Z hlediska objektového programování je pojem třída nadbytečný. Každý objekt reálného světa lze do jisté míry modelovat objektem ve smyslu objektového programování. Představme si ale situaci, že bychom chtěli vytvořit program, který by uchovával jméno. Příjmení a věk několika osob. V případě, že bychom neznali pojem třída, museli bychom pro každou osobu vytvořit zcela nový objekt a definovat v nich všechny nezbytné proměnné a metody. Je zřejmé, že při takovém přístupu by to vedlo k výrazné duplicitě v kódu, což je

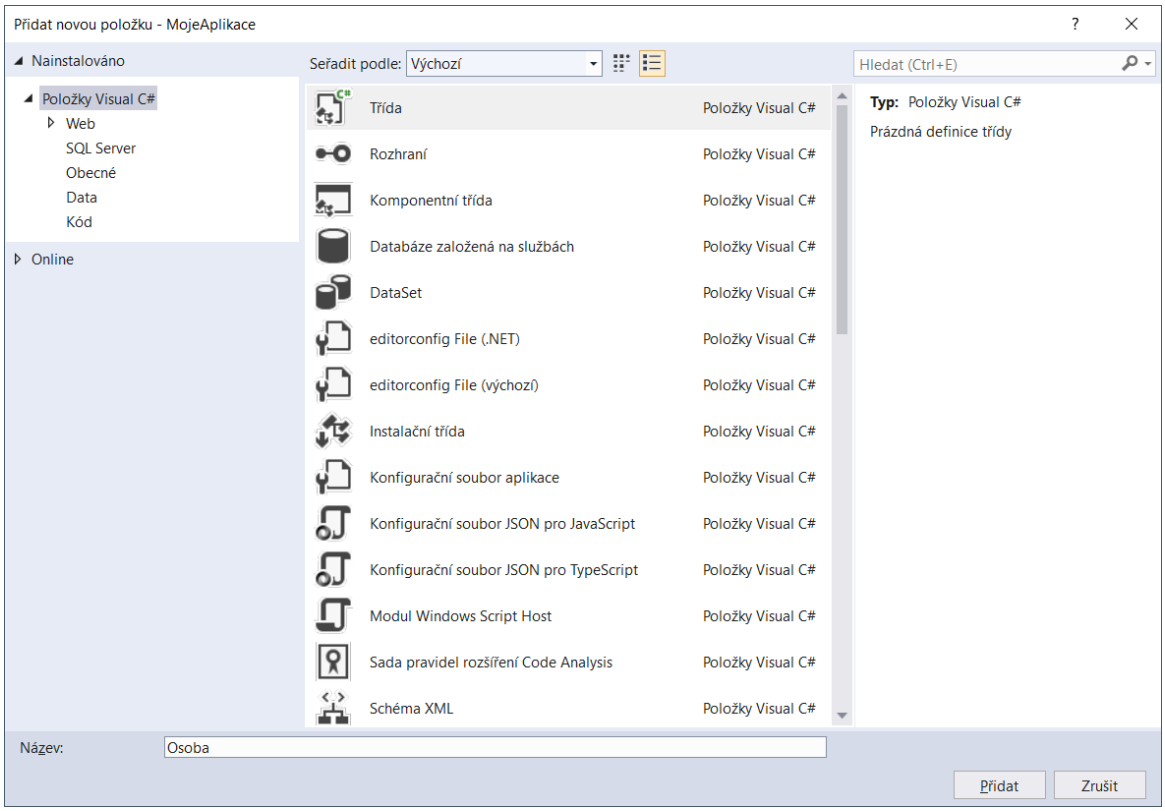
jistě nežádoucí. Z tohoto důvodu se v objektově orientovaných jazycích využívají třídy, které můžeme chápat jako nějaký předpis či šablonu pro vytvoření konkrétního objektu. Každý objekt je vytvořen na základě předpisu nějaké třídy. Zde se velmi často setkáme s pojmem instance. Pokud je nějaký objekt vytvořen na základě předpisu třídy, můžeme říct, že objekt je instancí příslušné třídy.

Zkusme si nyní ukázat praktickou ukázku vytvoření třídy *Osoba*, která bude obsahovat proměnné *Jmeno*, *Prijmeni* a *Vek*, přičemž *Jmeno* a *Prijmeni* budou typy *string* a *Vek* typu *int*. V jazyce C# je zvykem, že každá třída je umístěna v samostatném souboru. Třídu v MS Visual Studiu vytvoříme v *Průzkumníku řešení* kliknutím pravým tlačítkem myši na složku projektu a poté zvolíme *Přidat→Třída*.



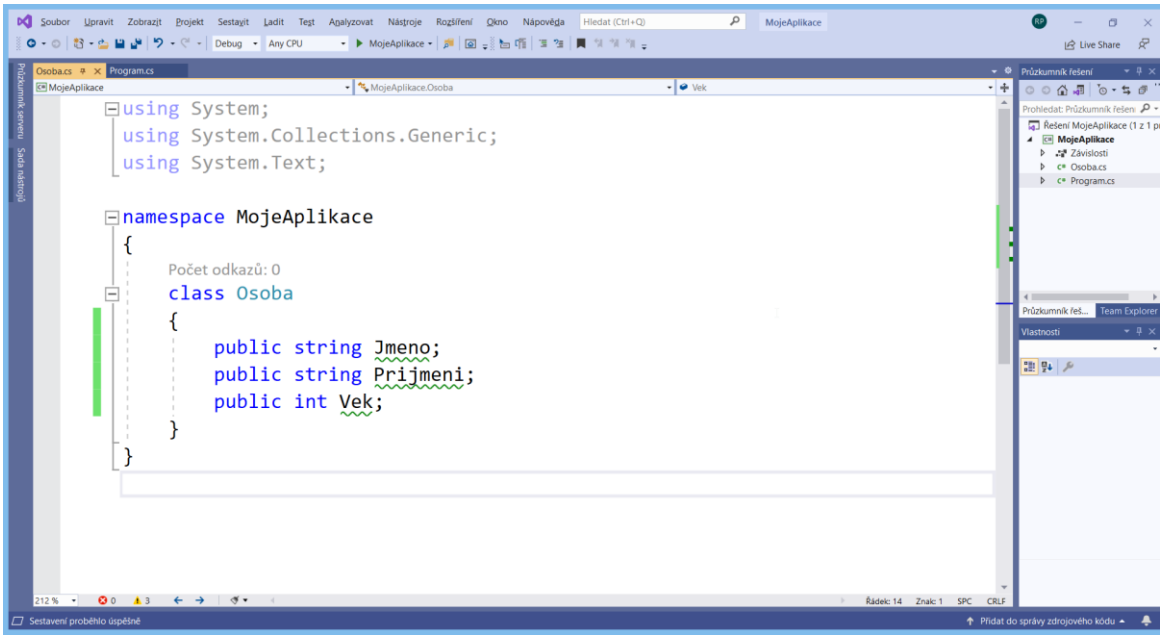
Obrázek 2.1: Přidání nové třídy

Po potvrzení se nám zobrazí formulář, ve kterém zadáme ve spodní části název třídy. V našem případě zadáme název *Osoba*.



Obrázek 2.2: Název nové třídy

Po potvrzení tlačítkem *Přidat* dojde k vytvoření nového souboru s názvem *Osoba.cs*, což si můžeme zkontrolovat v *Průzkumníku řešení*. Zároveň v editoru se nám zobrazí zdrojový kód této třídu, můžeme si všimnout, že jeho struktura je podobná zdrojovému kód třídy *Program*, která se vytváří automaticky při vytvoření nového projektu. Pouze uvnitř třídy již není uvedena metoda *main*. Nyní si ukážeme, jak v naší nové třídě deklarujeme výše uvedené proměnné *Jmeno*, *Prijmeni* a *Vek*.



Obrázek 2.3: Třída Osoba

Jak můžeme vidět deklarace proměnných uvnitř třídy je analogická jako deklarace lokálních proměnných uvnitř metody. Jediný rozdíl je v tom, že před příslušným datovým typem je vždy uvedeno klíčové slovo *public*, což je tzv. modifikátor přístupu, který určuje, že k těmto proměnným bude možné přistupovat odkudkoli. Modifikátorů přístupu existuje více, podrobněji se jimi budeme zabývat později. Těmto proměnným prozatím nemůžeme přiřadit žádné hodnoty. Aby to bylo možné, nejprve musíme vytvořit instanci této třídy. Obecný zápis pro vytvoření instance třídy vypadá takto:

```
Třída název_instance = new Třída ();
```

Můžeme si všimnout, že levá část je analogická s deklarací proměnné, kdy třídu můžeme chápat jako datový typ. Pravá část potom obsahuje klíčové slovo *new*, které vytváří vlastní instanci třídy.

Nyní si vytvoříme dvě instance třídy *Osoba*, u každé instance nastavíme hodnoty proměnných, a nakonec hodnoty těchto proměnných vypíšeme na obrazovku. Vše provedeme v metodě *main* třídy *Program*,

```
class Program
{
    Počet odkazů: 0
    static void Main(string[] args)
    {
        Osoba a = new Osoba();
        a.Jmeno = "Pavel";
        a.Prijmeni = "Novák";
        a.Vek = 25;
        Console.WriteLine(a.Jmeno + " " + a.Prijmeni + " " + a.Vek);

        Osoba b = new Osoba();
        b.Jmeno = "Roman";
        b.Prijmeni = "Pokorný";
        b.Vek = 25;
        Console.WriteLine(b.Jmeno + " " + b.Prijmeni + " " + b.Vek);
    }
}
```

Obrázek 2.4: Vytvoření instance třídy

Ve výše uvedeném příkladu jsou identifikátory *a* a *b* instancemi třídy *Osoba*, přičemž každá z těchto instancí má jiné hodnoty proměnných *Jmeno*, *Prijmeni* a *Vek*. Tyto instance nazýváme objekty. Z jedné třídy lze vytvořit libovolné množství instancí, z nichž každá může mít různé hodnoty svých proměnných. Dále si všimněte způsobu, jakým přistupujeme k proměnným jednotlivých instancí. Vždy napíšeme název instance (objektu), potom tečku a nakonec název proměnné. Dále již s tímto identifikátorem pracujeme jako by to byla obyčejná proměnná, ať už se jedná o přiřazení hodnoty nebo ke čtení hodnoty.

Způsob, jakým jsme doposud používali třídy, by bylo možné zcela nahradit uživatelsky definovaným datovým typem. Třídy však mají tu výhodu, že kromě proměnných můžeme ve třídách definovat i metody, které budou s proměnnými tříd pracovat. Zkusme si nyní vytvořit metodu *Vypis*, která vypíše hodnoty všech proměnných definovaných v třídě *Osoba*.

```
class Osoba
{
    public string Jmeno;
    public string Prijmeni;
    public int Vek;

    Počet odkazů: 0
    public void Vypis()
    {
        Console.WriteLine(Jmeno + " " + Prijmeni + " " + Vek);
    }
}
```

Obrázek 2.5: Deklarace metody

Vidíme, že deklarace metody začíná opět identifikátorem přístupu *public*, které označuje, že metoda bude přístupná odkudkoli. Následuje klíčové slovo *void* určující, že daná metoda nebude vracet žádnou návratovou hodnotu. Potom je uveden název metody spolu s kulatými závorkami, které odlišují deklaraci metody od proměnné. Nakonec je ve složených závorkách uvedeno tělo metody, tedy kód, které se vykoná po zavolání této metody. Samotný výpis na obrazovku je realizován metodou *Console.WriteLine*, všimněte si ale, že u jednotlivých proměnných již neuvádíme název instance třídy, jako jsme to dělali v předchozím příkladu. Je tomu tak proto, že nyní se nacházíme uvnitř třídy a máme přímý přístup k jednotlivým proměnným. Po vytvoření instance třídy se použijí hodnoty proměnných v příslušné instanci. Podívejme se teď, jak bude vypadat kód v metodě *main*, pokud pro výpis na obrazovku využijeme metodu *Vypis*.

```
static void Main(string[] args)
{
    Osoba a = new Osoba();
    a.Jmeno = "Pavel";
    a.Prijmeni = "Novák";
    a.Vek = 25;
    a.Vypis();

    Osoba b = new Osoba();
    b.Jmeno = "Roman";
    b.Prijmeni = "Pokorný";
    b.Vek = 25;
    b.Vypis();
}
```

Obrázek 2.6: Volání metody

Můžeme vidět, že přístup k metodám je podobný jako přístup k proměnným. Začneme vždy názvem instance třídy, potom napíšeme tečku a nakonec názvem metody bezprostředně následovaným kulatými závorkami. Pokud tento zápis srovnáme s předchozím kódem pro výpis na obrazovku, vidíme, že kód je kratší. Významnější výhodou ale je, že nyní je formát výpisu definovaný pouze na jednom místě a pokud bychom se rozhodli tento formát změnit, stačí to provést pouze na jednom místě (v metodě *Vypis*).

2.2 Konstruktor

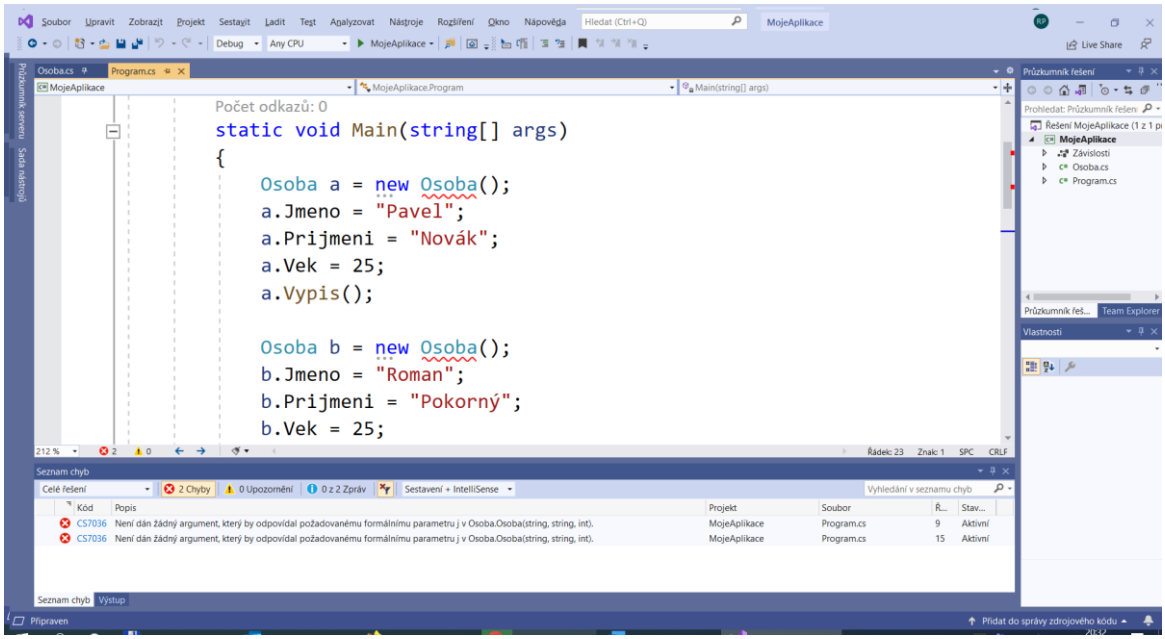
Už jsme si ukázali, jak můžeme ve třídě deklarovat proměnné i metody. Existuje však jedna speciální metoda, která se vyznačuje tím, že se spustí automaticky při vytváření instance třídy. Pro vytvoření konstruktoru se nepoužívá žádné speciální klíčové slovo, ale je definován tím, že název metody konstruktoru je shodný s názvem třídy. Oproti jiným metodám také nemá definovaný žádný návratový typ a dokonce ani klíčové slovo *void*. Typické použití konstruktoru je pro nastavení výchozích hodnot proměnných instance třídy. Zkusme si tedy v naší třídě Osoba deklarovat konstruktor, který umožní nastavit hodnoty proměnných při vytváření instance. Ve třídě jsou deklarovány tři proměnné, proto konstruktor bude mít rovněž tři parametry. Konstruktor tedy deklarujeme následovně.

```
class Osoba
{
    public string Jmeno;
    public string Prijmeni;
    public int Vek;

    Počet odkazů: 2
    public Osoba(string j, string p, int v)
    {
        Jmeno = j;
        Prijmeni = p;
        Vek = v;
    }
```

Obrázek 2.7: Konstruktor s parametry

Vzhledem k tomu, že jde o metodu, která se vykoná vždy jako první, je zvykem tyto metodu deklarovat jako první metodu ihned za deklarací proměnných, ale program bude fungovat, i když konstruktor budeme deklarovat kdekoli jinde ve třídě. Všimněte si, že v konstruktoru máme uvedeny tři parametry *j*, *p* a *v*, pro každou proměnnou třídy jeden parametr. Jejich názvy můžeme definovat libovolně, jen by měly být odlišné od názvu proměnných ve třídě, aby je bylo možné rozlišit. V těle konstruktoru potom přiřadíme každý parametr příslušné proměnné třídy. Pokud se pokusíme po přidání konstruktoru náš program spustit, obdržíme dvě syntaktické chyby.



Obrázek 2.8: Chyba chybějící konstruktor

Samotný popis chyb není na první pohled příliš zřejmý, ale situace je jasnější, pokud se podíváme na místa v kódu, kde se chyby nacházejí. Doposud jsme si nevysvětlili, co znamená při vytváření instance třídy část za klíčovým slovem *new*. Tato část totiž specifikuje, který konstruktor se při vytváření instance zavolá. V našem kódu v metodě *main* máme uvedeno *new Osoba()*, což znamená že se volá konstruktor bez parametrů. Uvedená chyba tedy znamená, že ve třídě Osoba není definován bezparametrický konstruktor, což je pravda, žádný takový konstruktor jsme nedeklarovali. Otázka ovšem je, proč jsme již tuto chybu neobdrželi před tím, než jsme deklarovali konstruktor se třemi parametry. V jazyce C# totiž platí, že každá třída musí mít definovaný alespoň jeden konstruktor a pokud nedeklarujeme vlastní konstruktor překladač automaticky vytvoří bezparametrický konstruktor s prázdným tělem. Pokud však deklarujeme alespoň jeden vlastní konstruktor, tento prázdný bezparametrický konstruktor se již automaticky překladačem nevytváří, proto jsme chybu obdrželi až po deklarování vlastního konstruktoru. Abychom tedy tuto chybu odstranili musím deklarovat ještě jeden bezparametrický konstruktor s prázdným tělem. Upravená třída bude tedy vypadat následovně.

```
class Osoba
{
    public string Jmeno;
    public string Prijmeni;
    public int Vek;

    Počet odkazů: 0
    public Osoba(string j, string p, int v)
    {
        Jmeno = j;
        Prijmeni = p;
        Vek = v;
    }

    Počet odkazů: 2
    public Osoba()
    {
    }
```

Obrázek 2.9: Bezparametrický konstruktor

Nyní již program půjde spustit bez chyb. Zkusme teď upravit náš program v metodě *main*, kdy k nastavení hodnot proměnných využijeme náš nový konstruktor se třemi parametry.

```
static void Main(string[] args)
{
    Osoba a = new Osoba("Pavel", "Novák", 25);
    a.Vypis();

    Osoba b = new Osoba("Roman", "Pokorný", 25);
    b.Vypis();
}
```

Obrázek 2.10: Volání konstruktoru s parametry

Pokud porovnáme předchozí způsob nastavení hodnot proměnných s nastavením pomocí konstruktoru, vidíme, že použití konstruktoru nám zápis kódu výrazně zjednodušujeme. Musíme jen dávat pozor, abychom uváděli hodnoty proměnných ve stejném pořadí jako jsme definovali pořadí parametrů v konstruktoru.

V souvislosti s konstruktorem by někoho mohlo napadnout, zda existuje metoda, která se naopak volá při ukončení instance třídy. Odpověď je kladná a taková metoda se nazývá destruktor. Deklarace destruktoru je podobná deklaraci konstruktoru s tím rozdílem, že destruktor nemá žádný modifikátor přístupu, nemá žádné parametry, před názvem má znak vlnovky ~ a ve třídě může být deklarován nejvýše jednou. Podívejme se na příklad deklarace destruktoru.

```
Počet odkazů: 2
public Osoba(string j, string p, int v)
{
    Jmeno = j;
    Prijmeni = p;
    Vek = v;
}

Počet odkazů: 0
public Osoba()
{
}

Počet odkazů: 0
~Osoba()
{
    Console.WriteLine("Destruktor byl zavolán");
}
```

Obrázek 2.11: Destruktor

Potíž s destruktory v jazyce C# je ale v tom, že nikdy nevíme, kdy a zda vůbec bude destruktor zavolán. V jazyce C# totiž veškeré uvolňování paměti po nepotřebných objektech řeší automaticky tzv. *Garbage Collector*. K uvolnění objektu z paměti dojde obvykle tehdy, pokud se blíží vyčerpání paměti a na daný objekt již neexistuje žádný odkaz, který by mohl k objektu přistupovat.

2.3 Zapouzdření

Doposud jsme všechny proměnné ve třídách deklarovali s modifikátorem přístupu public, tj. že jsou přístupné odkudkoli. Ovšem tento přístup není úplně ideální, protože porušuje jednu z důležitých vlastností objektově orientovaného programování a to zapouzdření, kdy k proměnným nelze přímo přistupovat z vnějšku a přístup k vnitřním proměnným se realizuje pouze pomocí metod. Zkusme si nyní demonstrovat princip zapouzdření na zjednodušené třídě *Osoba*, která bude obsahovat pouze jednu proměnnou *Vek* a nebude mít žádné metody ani konstruktor.

```
class Osoba
{
    private int Vek;

    Počet odkazů: 1
    public void NastavVek(int v)
    {
        Vek = v;
    }

    Počet odkazů: 1
    public int VratVek()
    {
        return Vek;
    }
}
```

Obrázek 2.12: Zapouzdření

Jak vidíme, u proměnné *Vek* jsme nahradili modifikátor přístupu *public* modifikátorem přístupu *private*, což znamená, že k této proměnné lze přistupovat pouze z metod definovaných uvnitř této třídy. Pokud bychom se pokusili k této proměnné přistupovat odjinud, obdrželi bychom syntaktickou chybu. Abychom však mohli nějakým způsobem nastavit hodnotu proměnné Vek nebo přečíst její hodnotu, vytvořili jsme dvě jednoduché metody *NastavVek* a *VratVek*. Metoda *NastavVek* nastaví hodnotu proměnné *Vek* na hodnotu předanou parametrem *v*. Metoda *VratVek* potom vrátí hodnotu proměnné *Vek* jako návratovou hodnotu. Nyní si ukážeme, jak můžeme nastavit a přečíst hodnotu proměnné *Vek* v metodě *Main*.

```
static void Main(string[] args)
{
    Osoba a = new Osoba();
    a.Vek = 5;  //Nelze přímo přistupovat k proměnné -> chyba
    a.NastavVek(5);
    Console.WriteLine(a.VratVek());
}
```

Obrázek 2.13: Přístup k soukromé proměnné

Nabízí se samozřejmě otázka, proč kód takto komplikovat. Hlavní výhoda tohoto přístupu se ukáže, pokud bychom se pokusili nastavit nějakou nepřípustnou hodnotu, např. kdybychom se pokusili do proměnné *Vek* nastavit zápornou hodnotu, čímž by došlo k porušení konzistence objektu. Pokud bychom měli proměnnou *Vek* veřejně přístupnou a k nastavení hodnoty použili prosté přiřazení, nemohli bychom objekt před nepřípustnými hodnotami nijak ochránit. V případě použití metody pro nastavení hodnoty můžeme tuto metodu jednoduše upravit tak, že do ní doplníme podmínku a pokud zadaná hodnota nebude této podmínce vyhovovat, změnu proměnné neprovedeme, případně můžeme vypsat chybovou hlášku apod. Zkusme tedy naši metodu *NastavVek* upravit tak, abychom do proměnné Vek mohli uložit pouze nezáporné číslo a zároveň omezíme i horní hranici, např. na 150 let.

```
class Osoba
{
    private int Vek;

    Počet odkazů: 1
    public void NastavVek(int v)
    {
        if (v >= 0 && v <= 150)
        {
            Vek = v;
        }
        else
        {
            Console.WriteLine("Zadaná hodnota věku není " + v + " přípustná");
        }
    }
```

Obrázek 2.14: Test na přípustnost hodnoty

Vidíme tedy, že k nastavení hodnoty do proměnné Vek dojde pouze tehdy, pokud hodnota bude větší nebo rovna nule a zároveň nebude větší než 150. V opačném případě metoda vypíše na obrazovku chybovou hlášku. Význam použití metody pro nastavení hodnoty proměnné jsme si už vysvětlili, ovšem nemusí být úplně zřejmé, proč používat metodu i pro vrácení hodnoty proměnné. Samozřejmě, že pro tento účel význam použití metod není

až tak velký, ale mohli bychom tento způsob použít např. pro ochranu dat, kdy k určitým proměnným budou mít přístup pouze někteří uživatelé.

2.4 Vlastnosti

Jak jsme si ukázali, použití metod pro přístup k proměnným nám umožňuje kontrolovat, jaké hodnoty se do proměnných ukládají. Nicméně jste si jistě všimli, že samotný kód potom vypadá poněkud těžkopádně. Proto v jazyce C# existuje jisté zjednodušení, tzv. vlastnosti, které umožňují jednoduše zapouzdřit určitou proměnnou, kdy uvnitř třídy je přístup k proměnným realizován pomocí metod, ale z vnějšku vypadá zápis, jako bychom přímo přiřazovali hodnotu dané proměnné nebo z ní četli. Obecný zápis pro vlastnosti vypadá takto.

```
private datovy_typ promenna;
public datovy_typ Vlastnost
{
        get { return promenna; }
        set { promenna = value; }
}
```

Vidíme, že nejprve deklarujeme proměnnou, kterou chceme zapouzdřit a označíme ji modifikátorem přístupu *private* samozřejmě můžeme použít libovolný modifikátor, ale pak by to poněkud postrádalo smysl. K této proměnné tedy nebude možno přistupovat z vnějšku. Následně deklarujeme vlastnost, která se navenek bude chovat jako obyčejná veřejně přístupná proměnná, ale uvnitř deklarace vlastnosti jsou pro čtení a zápis do proměnné použity metody, které jsou reprezentovány klíčovými slovy *get* a *set*. Jak už jejich název napovídá, metoda *get* slouží pro vrácení hodnoty proměnné a metoda *set* slouží pro přiřazení hodnoty do proměnné. Metody *get* a set můžeme chápat jako zjednodušené metody, které nemají definované žádné parametry ani žádné návratové hodnoty, a dokonce ani nepoužívají kulaté závorky. Je tomu tak proto, že datový typ návratové hodnoty i parametry je pevně dán deklarací vlastnosti. Všimněme si zejména, že pro parametr metody *set* je definované klíčové slovo *value*. Zkusme si nyní přepsat předchozí příklad s využitím vlastností.

```
class Osoba
{
    private int _vek;

    Počet odkazů: 2
    public int Vek
    {
        get { return _vek; }
        set
        {
            if (value >= 0 && value <= 150)
            {
                _vek = value;
            }
        }
    }
}
```

Obrázek 2.15: Vlastnosti

Původní proměnnou *Vek* jsme nahradili proměnnou *_vek*, která je přístupná pouze uvnitř třídy a identifikátor *Vek* je nyní vlastnost, která bude na venek vystupovat jako obyčejná proměnná, tj. z metody *Main* můžeme do vlastnosti *Vek* přímo přiřadit nějakou hodnotu a stejně tak z této vlastnosti přímo číst. Uvnitř však budou volány metody *get* a *set* podle toho, zda půjde o požadavek přiřazení či čtení hodnoty.

Pokud budeme ve svých programech využívat vlastnosti, je dobré vědět, není nutné uvádět obě části *get* i *set* současně. Pokud například u vlastnosti deklarujeme pouze metodu *get*, potom se daná vlastnost bude chovat, jako by proměnná byla pouze pro čtení, tj. pokud se do vlastnosti pokusíme uložit nějakou hodnotu, překladač nám ohlásí syntaktickou chybu. Analogicky pokud u vlastnosti deklarujeme pouze metodu *set*, potom se daná vlastnost bude chovat, jako by byla pouze pro zápis a při pokusu o čtení z této vlastnosti obdržíme opět syntaktickou chybu.

2.5 Statické metody a proměnné

Doposud jsme si říkali, že třídy můžeme chápat jako nějaké šablony a až u jejich instancí můžeme pracovat s jejich proměnnými a metodami. Pokud však příslušnou proměnnou či metodu deklarujeme jako statickou pomocí klíčového slova static, potom se daná proměnná či metoda stane statickou a se statickými členy lze pracovat bez vytvoření instance.

Nyní si ukážeme použití statických metod na jednoduchém příkladu, kdy budeme mít třídu *Vypocty* se dvěma metodami *Soucet* a *Rozdil*, přičemž tyto metody budou statické a každá z nich bude mít dva celočíselné parametry.

```
class Vypocty
{
    Počet odkazů: 0
    public static int Soucet(int a, int b)
    {
        return a + b;
    }

    Počet odkazů: 0
    public static int Rozdil(int a, int b)
    {
        return a - b;
    }
}
```

Obrázek 2.16: Statické metody

Všimněte si klíčového slova *static*, který je umístěn za modifikátorem přístupu. Nyní si ukážeme, jak můžeme tyto statické metody volat z metody *Main*.

```
static void Main(string[] args)
{
    int soucet = Vypocty.Soucet(2, 3);
    Console.WriteLine("Součet: " + soucet);

    int rozdil = Vypocty.Rozdil(2, 3);
    Console.WriteLine("Rozdíl: " + rozdil);

    Vypocty v = new Vypocty();
    int soucet2 = v.Soucet(2, 3);  //Nelze - syntaktická chyba
}
```

Obrázek 2.17: Volání statické metody

Vidíme, že při volání statické metody nevytváříme instanci dané třídy, ale místo toho příslušnou metodu voláme přímo názvem třídy následovaným tečkou a názvem metody.

Ve spodní části příkladu je i demonstrováno, co se stane, pokud se statickou metodu pokusíme zavolat jako standardní metodu – při spuštění obdržíme syntaktickou chybu.

K ZAPAMATOVÁNÍ

Statickou metodu nelze volat z instance třídy.

Nyní si zkusíme demonstrovat využití statických proměnných. Vytvoříme si třídu *Ucet*, která bude mít reálnou proměnnou *Zustatek* uchovávající zůstatek na daném bankovním účtu, dále bude mít celočíselnou statickou proměnnou *PocetUctu* uchovávající informaci o celkovém počtu otevřených účtů, tj. počet vytvořených instancí z třídy *Ucet*. Dále bude mít třída bezparametrický konstruktor, který zvýší hodnotu proměnné *PocetUctu* o 1. Nakonec bude mít třída metodu *Uloz*, která na účet uloží příslušnou částku o předanou v parametru této metody, tj. zvýší o tuto hodnotu proměnnou *Zustatek*. Naše třída bude tedy vypadat následovně.

```
class Ucet
{
    public double Zustatek;
    public static int PocetUctu = 0;

    Počet odkazů: 2
    public Ucet()
    {
        PocetUctu++;
    }

    Počet odkazů: 2
    public void Vloz(double castka)
    {
        Zustatek += castka;
    }
}
```

Obrázek 2.18: Statická proměnná

Nyní si vytvoříme dvě instance třídy *Ucet* a ověříme si, jaká bude výsledná hodnota proměnné *PocetUctu*.

```
static void Main(string[] args)
{
    Ucet ucet1 = new Ucet();
    ucet1.Vloz(500);
    Console.WriteLine("Zůstatek účtu 1: " + ucet1.Zustatek);

    Ucet ucet2 = new Ucet();
    ucet2.Vloz(750);
    Console.WriteLine("Zůstatek účtu 2: " + ucet2.Zustatek);

    Console.WriteLine("Počet účtů: " + Ucet.PocetUctu);
}
```

Obrázek 2.19: Použití statické proměnné

Všimněte si, že ke statické proměnné přistupujeme podobně jako u statických metod pomocí názvu třídy, nikoli její instance. Můžete si ověřit, že po spuštění programu bude hodnota proměnné 2. Podobně bychom pomocí statické proměnné mohli evidovat celkovou částku uloženou na všech účtech.

K ZAPAMATOVÁNÍ

Statická proměnná je společná pro všechny instance.

OTÁZKY

1. Objekt je
    a. Instance třídy
    b. V jazyce C# neexistuje
    c. Druh cyklu
2. Které klíčové slovo použijeme pro vytvoření objektu?
    a. create
    b. new
    c. init
3. Konstruktor se deklaruje pomocí klíčového slova
    a. constructor
    b. const
    c. Konstruktor je definován názvem třídy
4. Který symbol používáme při deklaraci destruktoru?

   a. *
   b. %
   c. ~
5. Který modifikátor přístupu použijeme, pokud chceme, aby proměnná byla přístupná pouze ve třídě, kde je deklarovaná?
   a. void
   b. private
   c. encapsulated
6. Který modifikátor přístupu použijeme, pokud chceme, aby proměnná byla přístupná z celého programu?
   a. public
   b. open
   c. everywhere
7. Co je to Garbage Collector?
   a. Je to odkaz na koš v systému MS Windows, kam se ukládají odstraněné soubory.
   b. Dočasný soubor na disku.
   c. Mechanismus, který zajišťuje automatické odstranění objektů z paměti, pokud již na ně neexistuje žádný odkaz.
8. Vytvoříme-li třídu *Osoba* se statickou proměnnou typu *int*, kolik místa bude tato statická proměnná zabírat v paměti, pokud vytvoříme dvě instance třídy *Osoba*?
   a. 2
   b. 4
   c. 8
9. Musíme vytvořit instanci třídy, pokud chceme zavolat statickou metodu deklarovanou v této třídě?
   a. Ne
   b. Ano
   c. Statické metody nelze zavolat
10. Co může obsahovat třída?
   a. Pouze proměnné
   b. Pouze metody
   c. Proměnné i metody

SHRNUTÍ KAPITOLY

Tato kapitola byla zaměřena na základy objektově orientovaného programování. Seznámili jsme se s pojmem třída, ukázali jsme si, jak vytvořit instanci třídu a k čemu se používá konstruktor. Dále jsme se podívali na možnosti ochrany vnitřního stavu objektu pomocí zapouzdření. Nakonec jsme se zmínili o statických metodách a proměnných a ukázali jsme si příklady jejich použití.

ODPOVĚDI

1. a
2. b
3. c
4. c
5. b
6. a
7. c
8. b
9. a
10. c

3 DĚDIČNOST

RYCHLÝ NÁHLED KAPITOLY

V této kapitole si představíme základní prvek strukturovaného programování, a to členění programu do menších částí, které nám potom umožňují opakované vykonávání částí kódu. Začneme těmi nejjednoduššími metodami, které pokaždé vykonají tutéž činnost, dále si popíšeme způsob, jak mohou metody vracet nějaký výsledek, a nakonec budeme vytvářet metody, které budou měnit své chování v závislosti na hodnotách parametrů, které jim předáme. Ukážeme si rovněž, jaký je rozdíl mezi parametry předávané hodnotou a parametry předávané odkazem.

CÍLE KAPITOLY

Po prostudování této kapitoly budete umět:

- Vytvořit metodu.
- Využívat návratovou hodnotu.
- Modifikovat chování metody pomocí parametrů.
- Rozlišovat mezi parametry volanými hodnotou a odkazem.
- Vracet více než jednu návratovou hodnotu.

KLÍČOVÁ SLOVA KAPITOLY

Metoda, parametr, hodnota, odkaz, návratová hodnota.

3.1 Dědičnost atributů a metod

Dědičnost je jedna ze základních vlastností objektového programování, která umožňuje z jedné třídy vytvořit jinou třídu tak, že ji rozšíří o další metody či atributy. Ukážeme si tento princip na příkladu, kdy budeme mít třídu vozidlo, která bude obsahovat proměnnou *Nazev* typu *string* a celočíselnou proměnnou *Hmotnost*. Dále bude mít třída metodu *Vypis*, která vypíše na obrazovku hodnotu těchto dvou proměnných.

```
class Vozidlo
{
    public string Nazev;
    public int Hmotnost;

    Počet odkazů: 0
    public void Vypis()
    {
        Console.WriteLine(Nazev + " " + Hmotnost);
    }
}
```

Obrázek 3.1: Třída Vozidlo

Nyní bychom si chtěli vytvořit třídu *Auto*, která bude obsahovat stejné proměnné jako třída *Vozidlo* a navíc ještě bude mít novou reálnou proměnnou *PrumerVolantu*. Jednou možností je, že bychom vytvořili novou třídu a v ní definovali všechny proměnné, které jsme definovali ve třídě *Vozidlo*, a k nim bychom přidali proměnnou *PrumerVolantu*. Tento přístup by samozřejmě fungoval, ale sami vidíte, že to vede k jisté duplicitě. Ukážeme si tedy druhý způsob, který využívá dědičnost. To, že nějaká třída dědí z jiné třídy, zapíšeme v kódu velmi jednoduše a to tak, že za deklaraci nové třídy zapíšeme dvojtečku a za ní název třídy, ze které dědíme. Obecný zápis dědičnosti vypadá následovně:

```
class Potomek : Rodic
{
        Deklarace dalších proměnných a metod
}
```

Pro náš konkrétní příklad bude nová třída *Auto* dědící z třídy *Vozidlo* vypadat následovně:

```
class Auto : Vozidlo
{
    public double PrumerVolantu;
}
```

Obrázek 3.2: Třída Auto

Touto deklarací přebírá třída *Auto* všechny proměnné a metody z třídy *Vozidlo* a navíc k nim přidává proměnnou *PrumerVolantu*. Zkusme si nyní vytvořit instanci této nové třídy *Auto*.

```
static void Main(string[] args)
{
    Auto a = new Auto();
    a.Nazev = "Ford";
    a.Hmotnost = 1350;
    a.PrumerVolantu = 45;
    a.Vypis();
}
```

Obrázek 3.3: Instance třídy Auto

Pokud tento program spustíme, uvidíme, že nám vypíše pouze název a hmotnost auta, ale nikoliv průměr volantu. Je to pochopitelné, protože pro výpis na obrazovku voláme metodu *Vypis*, která je definována ve třídě Vozidlo a v této metodě vypisujeme pouze proměnné definované v třídě *Vozidlo*. Abychom tedy docílili vypsání všech proměnných, musíme si definovat ve třídě Auto novou metodu, např. *VypisAuto*, která nám vypíše všechny tři proměnné. Jednou možností je, že tato metoda vypíše tyto tři proměnné přímo pomocí metody *Console.WriteLine*, ale tím vznikne opět jakási duplicita kódu, kdy budeme mít kód pro výpis prvních dvou proměnných definovaný na dvou místech, proto zvolíme přístup, kdy metoda *VypisAuto* nejprve zavolá metodu *Vypis* z třídy *Vozidlo* a potom vypíše pouze novou proměnnou *PrumerVolantu*. Třída *Auto* bude nyní vypadat následovně:

```
class Auto : Vozidlo
{
    public double PrumerVolantu;

    Počet odkazů: 1
    public void VypisAuto()
    {
        Vypis();
        Console.WriteLine("Průměr volantu: " + PrumerVolantu);
    }
}
```

Obrázek 3.4: Třída Auto s metodou VypisAuto

Následně musíme upravit v metodě *Main* volání metody *Vypis* na *VypisAuto*:

```
static void Main(string[] args)
{
    Auto a = new Auto();
    a.Nazev = "Ford";
    a.Hmotnost = 1350;
    a.PrumerVolantu = 45;
    a.VypisAuto();
}
```

Obrázek 3.5: Instance třídy Auto s voláním metody VypisAuto

Nyní nám již tento kód správně vypíše hodnoty proměnných definovaných jak ve třídě *Vozidlo* tak ve třídě *Auto*.

3.2 Konstruktory a dědičnost

U demonstrace dědičnosti jsme si doposud vystačili bez použití konstruktorů. Zkusme si nyní doplnit do obou tříd konstruktory, které nám umožní zadat výchozí hodnoty proměnných. Začne nejprve konstruktorem pro třídu *Vozidlo*.

```
class Vozidlo
{
    public string Nazev;
    public int Hmotnost;

    Počet odkazů: 0
    public Vozidlo(string n, int h)
    {
        Nazev = n;
        Hmotnost = h;
    }

    Počet odkazů: 1
    public void Vypis()
    {
        Console.WriteLine(Nazev + " " + Hmotnost);
    }
}
```

Obrázek 3.6: Třída Vozidlo s konstruktorem

Pokud se nyní pokusíme program spustit, obdržíme syntaktickou chyba. Abychom si objasnili příčinu této chyby, musíme si něco říct o tom, jak funguje volání konstruktorů v případě tříd dědících z jiných tříd. V případě, že vytvoříme instanci potomka, vždy se nejprve zavolá konstruktor rodiče a až poté konstruktor potomka. Pokud ve třídě potomka není určeno, který konstruktor rodiče se má zavolat, potom se vždy volá bezparametrický konstruktor rodiče. Vzhledem k tomu, že jsme ve třídě rodiče definovali náš vlastní konstruktor, tento bezparametrický konstruktor se již automaticky nevytváří, a proto překladač hlásí tuto chybu. Abychom tedy tuto chybu odstranili, musíme ve třídě Vozidlo deklarovat i bezparametrický konstruktor s prázdným tělem:

```
class Vozidlo
{
    public string Nazev;
    public int Hmotnost;

    Počet odkazů: 0
    public Vozidlo(string n, int h)
    {
        Nazev = n;
        Hmotnost = h;
    }

    Počet odkazů: 0
    public Vozidlo()
    {
    }
```

Obrázek 3.7: Třída Vozidlo s bezparametrickým konstruktorem

Nyní již půjde program spustit bez chyb. Podobným způsobem doplníme konstruktor i do třídy *Auto*, který opět umožní zadat výchozí hodnoty všech proměnných. Zde je třeba si jen uvědomit, že třída *Auto* má ve skutečnosti tři proměnné, proto musí mít i konstruktor tři parametry.

```
class Auto : Vozidlo
{
    public double PrumerVolantu;

    Počet odkazů: 1
    public Auto(string n, int h, double p)
    {
        Nazev = n;
        Hmotnost = h;
        PrumerVolantu = p;
    }
```

Obrázek 3.8: Třída auto s konstruktorem

Instanci třídy *Auto* s využitím konstruktoru potom vytvoříme takto:

```
static void Main(string[] args)
{
    Auto a = new Auto("Ford", 1350, 45);
    a.VypisAuto();
}
```

Obrázek 3.9: Instance třídy Auto

Ve třídě *Auto* nemusíme deklarovat bezparametrický konstruktor, protože ho nikde nevyužíváme a nemáme ani žádnou třídu, která by dědila ze třídy *Auto*. Zkusme si ale nyní porovnat konstruktory ve třídě *Vozidlo* a *Auto*. Můžeme si všimnout, že první dva řádky konstruktoru ve třídě *Auto* se zcela shodují s tělem konstruktoru ve třídě *Vozidlo*. Nabízí se tedy otázka, zda bychom z konstruktoru ve třídě auto zavolat konstruktor ve třídě *Vozidlo*. Odpověď je kladná. Samotný zápis je analogií zápisu dědičnosti s tím, že u konstruktoru využíváme klíčové slovo *base*. Obecný zápis konstruktoru s voláním konstruktoru rodiče je následovný:

```
public Potomek(parametry) : base(parametry konstruktoru rodiče)
{
        Tělo konstruktoru potomka
}
```

Třída *Auto* s upraveným konstruktorem bude tedy vypadat takto:

```
class Auto : Vozidlo
{
    public double PrumerVolantu;

    Počet odkazů: 1
    public Auto(string n, int h, double p) : base(n, h)
    {
        PrumerVolantu = p;
    }

    Počet odkazů: 1
    public void VypisAuto()
    {
        Vypis();
        Console.WriteLine("Průměr volantu: " + PrumerVolantu);
    }
}
```

Obrázek 3.10: Volání konstruktoru rodiče

Můžete si všimnout, že v těle konstruktoru třídy *Auto* máme nyní pouze jeden řádek, který nastavuje hodnotu proměnné *PrumerVolantu*, protože proměnné *Nazev* a *Hmotnost* se nastavují v konstruktoru třídy *Vozidlo*. Tímto zápisem jsme rovněž definovali, který konstruktor rodiče se má spustit, takže nyní již není nezbytně nutné, aby ve třídě *Vozidlo* byl explicitně definovaný bezparametrický konstruktor. Můžete si vyzkoušet, zda program bude fungovat i bez tohoto konstruktoru.

3.3 Virtuální metody

Doposud jsme si ukázala, jak může třída dědit proměnné a metody z jiné třídy a jak může metoda potomka přistupovat k proměnným a metodám rodiče. Nabízí se otázka, zda by nějakým způsobem nešlo přistupovat metodám potomka ze svého rodiče. Přímo to nejde, ale existuje určitý mechanismus, který to do jisté míry umožňuje. Tímto mechanismem jsou tzv. virtuální metody. Jejich použití si ukážeme na hierarchii tříd, které budou reprezentovat tělesa, která budou mít rovinnou podstavu a nad podstavou budou vztyčeny kolmé stěny, např. pokud bude podstavou obdélník, tělesem bude kvádr nebo pokud podstavou bude kruh, potom tělesem bude válec. Nejprve si budeme definovat obecnou třídu *Teleso,* která bude mít pouze jednu reálnou proměnnou *Vyska* udávající výšku tělesa, dále metodu *Vypis*, která vypíše rozměry tělesa, tj. pro třídu hodnotu proměnné *Vyska*. Rovněž bude mít třída *Teleso* definovaná konstruktor pro zadání výchozí výšky tělesa. Třída *Teleso* bude tedy vypadat následovně:

```
class Teleso
{
    public double Vyska;

    Počet odkazů: 0
    public Teleso(double v)
    {
        Vyska = v;
    }

    Počet odkazů: 0
    public void Vypis()
    {
        Console.WriteLine("Výška: " + Vyska);
    }
}
```

Obrázek 3.11: Třída Teleso

Nyní si budeme chtít vytvořit třídu *Valec*, která bude dědit z obecné třídy *Teleso*. Třída *Valec* bude navíc obsahovat reálnou proměnnou *Polomer* reprezentující poloměr podstavy válce. Rovněž budeme chtít mít metodu pro výpis všech rozměrů válce, ale nebudeme to realizovat pomocí nové metody *VypisValec*, ale vyžijeme k tomu existující metodu *Vypis* v třídě *Teleso*. Budeme ale samozřejmě požadovat, aby metoda *Vypis* vypsala i hodnotu *Polomer* definovanou ve třídě *Valec*. Aby to bylo možné musíme nejprve upravit třídu *Teleso* a doplnit do ní virtuální metodu *DalsiRozmery*, která bude vracet řetězec popisující další rozměry příslušného tělesa. Tuto metodu potom využijeme v metodě *Vypis* pro vypsání všech rozměrů. Než tedy přistoupíme k deklaraci třídy *Valec*, upravíme třídu *Teleso* následovně:

```
class Teleso
{
    public double Vyska;

    Počet odkazů: 0
    public Teleso(double v)
    {
        Vyska = v;
    }

    Počet odkazů: 0
    public void Vypis()
    {
        Console.WriteLine("Výška: " + Vyska + " " + DalsiRozmery());
    }

    Počet odkazů: 1
    public virtual string DalsiRozmery()
    {
        return "";
    }
}
```

Obrázek 3.12: Třída Teleso s virtuální metodou

Všimněte si, že virtuální metodu označujeme klíčovým slovem *virtual* umístěným mezi modifikátor přístupu a návratový typ. V třídě *Teleso* vrací metoda *DalsiRozmery* pouze prázdný řetězec, protože obecné těleso žádné jiné rozměry, než výšku nemá. Slouží pouze pro to, abychom ji mohli předefinovat v potomkovi a mohli k ní přistupovat v metodě *Vypis*. Rovněž stojí za povšimnutí, že *DalsiRozmery* je metoda, takže při volání v metodě *Console.WriteLine*, musíme za názvem metody napsat kulaté závorky.

Nyní si budeme deklarovat třídu *Valec*, která bude dědit z třídy *Teleso*, navíc bude mít proměnnou *Polomer*, konstruktor pro zadání výchozích hodnot a rovněž předefinuje virtuální metodu *DalsiRozmery* tak, aby generovala řetězec s popisem poloměru válce.

```
class Valec : Teleso
{
    public double Polomer;

    Počet odkazů: 0
    public Valec(double v, double r) : base(v)
    {
        Polomer = r;
    }

    Počet odkazů: 2
    public override string DalsiRozmery()
    {
        return "Polomer: " + Polomer;
    }
}
```

Obrázek 3.13: Třída Valec

Všimněte si, že abychom předefinovali virtuální metodu *DalsiRozmery*, musíme vložit mezi modifikátor přístupu a návratový ty klíčové slovo *override*. Je rovněž důležité, aby metoda měla stejný název jako v rodičovské třídě. Nyní si pro kontrolu vytvoříme instance obou tříd a pro každou z nich zavoláme metodu *Vypis*.

```
static void Main(string[] args)
{
    Teleso t = new Teleso(5);
    t.Vypis();

    Valec v = new Valec(4, 3);
    v.Vypis();
}
```

Obrázek 3.14: Instance tříd Teleso a Valec

Využití virtuálních tříd je samozřejmě mnohem širší než jen pro výpis na obrazovku. Zkusme si nyní ukázat, jak můžeme využít virtuálních metod pro výpočet objemu těles. Určitě byste si vzpomněli na vzoreček objemu válce či kvádru, ale u složitějších těles by už to mohlo být trochu obtížnější. Když se ale pozorněji podíváme na tělesa, která naše hierarchie tříd popisuje, můžeme si definovat obecný vzorec libovolného tělesa, které bude odvozeno z třídy *Teleso*, a to jako součin obsahu podstavy tělesa a výšky tělesa. Výška

tělesa je dána hodnotou proměnné *Vyska*, ale obsah podstavy se bude pro každé těleso lišit. Zde se tedy nabízí využít k výpočtu obsahu podstavy virtuální metodu *ObsahPodstavy*, kterou vždy předefinujeme v konkrétním tělese. Metodu *ObsahPodstavy* pak můžeme využít v třídě *Teleso* pro výpočet objemu tělesa. Pro jednoduchost upravíme metodu *Vypis* tak, aby zobrazovala i vypočtený objem tělesa. Upravená třída těleso bude vypadat takto:

```
public void Vypis()
{
    Console.WriteLine("Výška: " + Vyska + " " + DalsiRozmery() + " objem: " + ObjemTelesa());
}

Počet odkazů: 2
public virtual string DalsiRozmery()
{
    return "";
}

Počet odkazů: 1
public virtual double ObsahPodstavy()
{
    return 0;
}

Počet odkazů: 1
public double ObjemTelesa()
{
    return ObsahPodstavy() * Vyska;
}
```

Obrázek 3.15: Virtuální metoda pro obsah podstavy

Všimněte si, že virtuální metoda *ObsahPodstavy* vrací nulu, protože obecné těleso nemá žádnou podstavu, tedy její obsah musí být roven nule. Pokud náš program spustíme, bude u každého tělesa zobrazen jeho objem, prozatím však u všech roven nule, protože jsme doposud nepředefinovali virtuální metodu *ObsahPodstavy*. To nyní uděláme pro třídu *Valec*.

```
class Valec : Teleso
{
    public double Polomer;

    Počet odkazů: 1
    public Valec(double v, double r) : base(v)
    {
        Polomer = r;
    }

    Počet odkazů: 2
    public override string DalsiRozmery()
    {
        return "Polomer: " + Polomer;
    }

    Počet odkazů: 2
    public override double ObsahPodstavy()
    {
        return Math.PI * Math.Pow(Polomer, 2);
    }
}
```

Obrázek 3.16: Obsah podstavy pro třídu Valec

Metoda *ObsahPodstavy* obsahuje vzoreček pro výpočet obsahu kruhu, kde jsme využili konstanty *Math.PI* pro číslo π a metodu *Math.Pow* pro umocnění poloměru na druhou. Nyní po spuštění programu bychom již měli obdržet správnou hodnotu objemu válce.

V jazyce C# existují i některé předdefinované virtuální metody. Zřejmě nejpoužívanější z nich je metoda *ToString*, která specifikuje, jak se daný objekt převed na datový typ *string*. Zřejmě vás již napadlo, proč v každé třídě deklarujeme nějakou metodu pro výpis na obrazovku, když by se zdálo přímočařejší použít k tomu metodu *Console.WriteLine* a jako parametr uvést název instance příslušné třídy. Tento přístup sice bude fungovat neobdržíme syntaktickou chybu, ale nezobrazí se nám hodnoty proměnných daného objektu ale název třídy. A právě pomocí metody *ToString* můžeme toto výchozí chování změnit. Zkusme si nyní ve třídě *Teleso* předefinovat virtuální metodu *ToString* tak, aby zobrazovala totéž, co metoda *Vypis*.

```
class Teleso
{
    public double Vyska;

    Počet odkazů: 2
    public Teleso(double v)
    {
        Vyska = v;
    }

    Počet odkazů: 2
    public void Vypis()
    {
        Console.WriteLine("Výška: " + Vyska + " " + DalsiRozmery() + " objem: " + ObjemTelesa());
    }

    Počet odkazů: 0
    public override string ToString()
    {
        return "Výška: " + Vyska + " " + DalsiRozmery() + " objem: " + ObjemTelesa();
    }
```

Obrázek 3.17: Metoda ToString

Všimněte si, že v metodě *ToString* se na obrazovku nic nevypisuje, pouze se poskládá řetězec, který vrací jako návratovou hodnotu. Nyní když napíšeme *Console.WriteLine(v)*, zobrazí se nám totéž, co při zavolání metody *Vypis*. Vzhledem k tomu, že třída *Valec* je potomkem třídy *Teleso*, dědí rovněž i tuto virtuální metodu *ToString* a tudíž můžeme tento mechanismus používat i pro instance třídy *Valec*, resp. libovolné instance tříd odvozených z třídy *Teleso*.

Když porovnáme metody *Vypis* a *ToString*, můžeme vidět, že jsou téměř identické, nabízí se tedy otázka, zda by nešlo tuto duplicitu odstranit. Zkusíme tedy upravit metody *Vypis* tak, aby využívala metodu *ToString*.

```
class Teleso
{
    public double Vyska;

    Počet odkazů: 2
    public Teleso(double v)
    {
        Vyska = v;
    }

    Počet odkazů: 2
    public void Vypis()
    {
        Console.WriteLine(ToString());
        //Console.WriteLine("Výška: " + Vyska + " " + DalsiRozmery() + " objem: " + ObjemTelesa());
    }

    Počet odkazů: 1
    public override string ToString()
    {
        return "Výška: " + Vyska + " " + DalsiRozmery() + " objem: " + ObjemTelesa();
    }
```

Obrázek 3.18: Využití metody ToString

Nyní již je výpis na obrazovku specifikován pouze na jednom místě v metodě *ToString* a v případě jakýchkoli úprav, stačí tuto úpravu provést pouze na jednom místě. Jediný detail, který by nám ještě mohl vadit je, že voláme explicitně metodu *ToString*, kdežto pokud máme k dispozici instanci třídy, stačí uvést instanci této třídy jako parametr metody *Console.WriteLine*. Metoda *Vypis* se ovšem nachází uvnitř třídy, a proto samozřejmě zde ještě její instance neexistuje. Nicméně právě z tohoto důvodu bylo zavedeno klíčové slovo *this*, které reprezentuje budoucí instanci dané třídy. S využitím klíčového slova *this* bude metoda *Vypis* vypadat následovně:

```
class Teleso
{
    public double Vyska;

    Počet odkazů: 2
    public Teleso(double v)
    {
        Vyska = v;
    }

    Počet odkazů: 2
    public void Vypis()
    {
        Console.WriteLine(this);
    }

    Počet odkazů: 0
    public override string ToString()
    {
        return "Výška: " + Vyska + " " + DalsiRozmery() + " objem: " + ObjemTelesa();
    }
```

Obrázek 3.19: Použití klíčového slova this

3.4 Modifikátory tříd

V souvislosti s dědičností je možno modifikovat chování tříd z hlediska dědičnosti. Když jsme vytvářeli obecnou třídu těleso, tak poněkud postrádá smysl, abychom z takovéto třídy vytvářeli instanci. Veškeré výpočty, které jsme v této třídě definovali začnou mít smysl až tehdy, když vytvoříme potomka z této třídy a předefinujeme potřebné virtuální metody. Pokud chceme, aby z třídy nebylo možno vytvořit instanci, použijeme k tomu klíčové slovo *abstract*, která umístíme před deklaraci třídy. Takovou třídu potom označujeme jako abstraktní třída. Abstraktní třída *Teleso* potom bude vypadat takto:

```
abstract class Teleso
{
    public double Vyska;

    Počet odkazů: 2
    public Teleso(double v)
    {
        Vyska = v;
    }

    Počet odkazů: 2
    public void Vypis()
    {
        Console.WriteLine(this);
    }

    Počet odkazů: 0
    public override string ToString()
    {
        return "Výška: " + Vyska + " " + DalsiRozmery() + " objem: " + ObjemTelesa();
    }
```

Obrázek 3.20: Abstraktní třída

Pokud se nyní pokusíme vtvořit instanci třídy *Teleso*, obdržíme syntaktickou chybu. Abstraktní třídy slouží především pro zpřehlednění kódu, aby omylem nedošlo k vytvoření instancí u tříd, kde to nemá smysl.

Pokud bychom naopak chtěli zamezit vytvoření potomka z dané třídy, potom k tomuto účelu používáme klíčové slovo *sealed*, které zapíšeme před deklaraci třídy. Takovou třídu nazýváme uzavřená třída. Zkusíme si nyní použití uzavřené třídy demonstrovat na třídě *Valec*, u které chceme zakázat vytvoření potomka z této třídy.

```
sealed class Valec : Teleso
{
    public double Polomer;

    Počet odkazů: 1
    public Valec(double v, double r) : base(v)
    {
        Polomer = r;
    }

    Počet odkazů: 2
    public override string DalsiRozmery()
    {
        return "Polomer: " + Polomer;
    }

    Počet odkazů: 2
    public override double ObsahPodstavy()
    {
        return Math.PI * Math.Pow(Polomer, 2);
    }
}
```

Obrázek 3.21: Uzavřená třída

Pokud se nyní pokusíme vytvořit novou třídu, která bude dědit ze třídy *Valec*, obdržíme rovněž syntaktickou chybu.

OTÁZKY

1. Jaký symbol použijeme, pokud chceme vytvořit třídu, která dědí z jiné třídy?
   a. ^
   b. :
   c. >
2. Dědit lze
   a. Pouze metody
   b. Pouze proměnné
   c. Proměnné i metody

3. Jaké klíčové slovo použijeme pro zavolání konstruktoru rodičovské třídy?
    a. base
    b. super
    c. parent
4. Každá třída musí mít
    a. Alespoň jeden konstruktor.
    b. Nejvýše jeden konstruktor.
    c. Právě jeden konstruktor.
5. Jaké klíčové slovo použijeme pro deklaraci virtuální metody?
    a. virtual
    b. override
    c. abstract
6. Jaké klíčové slovo použijeme pro předefinování virtuální metody?
    a. virtual
    b. override
    c. abstract
7. Jaké klíčové slovo použijeme pro instanci třídy uvnitř třídy?
    a. inst
    b. instance
    c. this
8. Co je to abstraktní třída?
    a. Třída, ze které nelze vytvořit instanci.
    b. Třída, ze které nelze dědit.
    c. Neexistující třída.
9. Co je to uzavřená třída?
    a. Třída, ze které nelze vytvořit instanci.
    b. Třída, ze které nelze dědit.
    c. Neexistující třída.
10. Jaké klíčové slovo použijeme pro deklaraci uzavřené třídy?
    a. abstract
    b. sealed
    c. closed

SHRNUTÍ KAPITOLY

V této kapitole jsme se naučili vytvářet metody a ukázali jsme si, jak je můžeme z programu opakovaně volat. Zároveň jsme si demonstrovali, jak můžeme modifikovat chování metody v závislosti na hodnotách parametrů. Ukázali jsme si rovněž, jak může metoda vracet návratovou hodnotu a s využitím parametrů předávaných odkazem jsme si předvedli, jak můžeme vracet více než jednu návratovou hodnotu.

ODPOVĚDI

1. b
2. c
3. a
4. a
5. a
6. b
7. c
8. a
9. b
10. b

4 POLYMORFISMUS

RYCHLÝ NÁHLED KAPITOLY

Nyní umíte vytvořit program, který postupně vykoná všechny příkazy, které jste zapsali. Představte si ale situaci, že budete chtít napsat program, který od uživatele načte dvě čísla a poté podle volby uživatele vypíše jejich součet nebo rozdíl. Jinými slovy budete chtít napsat takový program, který vykoná jednu sadu příkazů, pokud uživatel zvolí možnost součtu, a druhou sadu příkazů v případě, že uživatel zvolí možnost rozdílu. Toto se právě v této kapitole naučíte.

CÍLE KAPITOLY

Po prostudování této kapitoly budete umět:

- Rozdělit běh programu na více větví podle určitých podmínek.
- Jak pracovat s logickými výrazy.
- Vyhodnotit základní logické operátory.
- Poznat případy kdy nahradit příkaz *if* příkazem *switch*
- Nahradit jednoduchou podmínku ternárním operátorem.

KLÍČOVÁ SLOVA KAPITOLY

Podmínka, logický výraz, logická proměnná, ternární operátor.

4.1 Polymorfismus metod

Polymorfismus je důležitá vlastnost objektového programování umožňující, aby metoda se stejným názvem měnila své chování v závislosti na počtu nebo typu parametrů. Zkusme si nyní demonstrovat tuto vlastnost na třídě *Matemetika*, v níž bychom chtěli mít metody pro součet dvou reálných čísel, tří reálných čísel a rovněž metodu pro součet dvou čísel, která jsou ale zadána jako řetězce. Pokud bychom nevěděli o polymorfismu, museli bychom každou metodu pojmenovat jinak, např. *Soucet2*, *Soucet3*, *SoucetString*. S takto jednoduchou třídou o třech metodách by to jistě nebyl velký problém, ale v případě, že bychom chtěli mít metody pro součet až deseti čísel, navíc různého datového typu, bylo by obtížné nejen vymyslet vhodné názvy metod, ale zejména potom zvolit správnou metodu při jejich

volání. Se znalostí polymorfismu nám ovšem tyto starosti odpadají, protože všechny metody můžeme pojmenovat stejně a překladač sám pozná podle počtu a typu parametrů, kterou metodu má použít. Třída *Matematika* s výše uvedenými metodami bude vypadat následovně:

```
class Matematika
{
    Počet odkazů: 0
    public double Soucet(double a, double b)
    {
        return a + b;
    }

    Počet odkazů: 0
    public double Soucet(double a, double b, double c)
    {
        return a + b + c;
    }

    Počet odkazů: 0
    public double Soucet(string a, string b)
    {
        return Convert.ToDouble(a) + Convert.ToDouble(b);
    }
}
```

Obrázek 4.1: Polymorfismus

A takto potom můžeme výše uvedené metody volat z hlavního programu:

```
static void Main(string[] args)
{
    Matematika m = new Matematika();

    double a = m.Soucet(2, 3);
    Console.WriteLine(a);

    double b = m.Soucet(2, 3, 4);
    Console.WriteLine(b);

    double c = m.Soucet("2", "3");
    Console.WriteLine(c);
}
```

Obrázek 4.2: Použití polymorfismu

4.2 Přetěžování operátorů

Matematické operátory jako + nebo – jsme zvyklí používat pouze pro práci s čísly, ovšem jazyk C# umožňuje použít tyto operátory pro libovolné třídy. Samozřejmě, že pokud bychom se pokusili sečíst dva objekty třídy *Valec* z předchozí kapitoly, došlo by k syntaktické chybě, protože překladač by nevěděl, jak má takové objekty sečíst. Nejprve tedy musíme definovat jakým způsobem se mají objekty příslušné třídy sečíst. Tento mechanismus se nazývá přetěžování operátorů. Zkusme si tento mechanismus demonstrovat na příkladu komplexních čísel. Vytvoříme si tedy třídu *KomplexniCislo*, která bude obsahovat dvě reálné proměnné *RealnaCast* a *ImaginarniCast*, dále bude obsahovat konstruktor, metody pro výpis komplexního čísla na obrazovku a kód, který umožní použít operátor + pro sečtení dvou komplexních čísel. Pro připomenutí součet dvou komplexních čísel se vypočítá tak, že sečteme zvlášť reálné složky a zvlášť imaginární složky.

```
class KomplexniCislo
{
    public double RealnaCast;
    public double ImaginarniCast;
    Počet odkazů: 1
    public KomplexniCislo(double r, double i)
    {
        RealnaCast = r;
        ImaginarniCast = i;
    }
    Počet odkazů: 0
    public void Vypis()
    {
        Console.WriteLine(RealnaCast + "+" + ImaginarniCast + "i");
    }
    Počet odkazů: 0
    public static KomplexniCislo operator+(KomplexniCislo a, KomplexniCislo b)
    {
        double r = a.RealnaCast + b.RealnaCast;
        double i = a.ImaginarniCast + b.ImaginarniCast;
        return new KomplexniCislo(r, i);
    }
```

Obrázek 4.3: Přetížení operátoru +

Pokud se podíváme na kód, který se stará o přetížení operátoru +, můžeme si všimnout, že jeho deklarace se velmi podobá deklaraci statické metodě s tím, že název metody je vždy tvořeno klíčovým *operator* a symbolem operátoru, který přetěžujeme. Počet parametrů je dán počtem operandů příslušného operátoru. Návratový typ je obvykle stejný jako samotné operandy, ale může být obecně libovolný, např. výsledkem skalárního součinu dvou vektorů je reálné číslo.

Nyní se podívejme, jak můžeme využít přetíženého operátoru pro součet dvou komplexních čísel:

```
static void Main(string[] args)
{
    KomplexniCislo a = new KomplexniCislo(2, 3);
    KomplexniCislo b = new KomplexniCislo(4, 5);
    KomplexniCislo c = a + b;
    c.Vypis();
}
```

Obrázek 4.4: Využití přetíženého operátoru

Nemusíme ovšem sčítat pouze dvě komplexní čísla, ale i komplexní číslo s číslem reálným, protože reálné číslo je pouze speciální případ čísla komplexního, kdy imaginární

složka je rovna nule. Abychom tedy mohli sečíst číslo komplexní s číslem reálným, musíme znovu přetížit operátor +, ale tentokrát s parametry typu *KomplexniCislo* a *double*.

```
public static KomplexniCislo operator+(KomplexniCislo a, KomplexniCislo b)
{
    double r = a.RealnaCast + b.RealnaCast;
    double i = a.ImaginarniCast + b.ImaginarniCast;
    return new KomplexniCislo(r, i);
}

Počet odkazů: 0
public static KomplexniCislo operator +(KomplexniCislo a, double b)
{
    return a + new KomplexniCislo(b, 0);
}
```

Obrázek 4.5: Přetížení operátoru + pro KomplexniCislo a double

Všimněte si, že pro výpočet vlastního součtu využíváme již dříve přetížený operátor + pro dvě komplexní čísla, kdy druhý operand převedeme na komplexní číslo vytvořením nové instance třídy *KomplexniCislo* s využitím konstruktoru, kde druhý parametr, tedy imaginární složku nastavíme na nulu. Nyní již tedy můžeme sečíst komplexní číslo a reálné číslo, ovšem pokud se pokusíme sečíst stejné operandy, ale pouze v opačném pořadí, tj. reálné číslo a komplexní číslo, obdržíme syntaktickou chybu, že operátor není přetížen. Problém je totiž v tom, že je důsledně dodržováno pořadí operandů, protože např. 3-2 není totéž co 2-3. Samozřejmě pro součet komplexních čísel to neplatí a nezáleží v jakém pořadí sčítáme, takže aby nám součet reálného a komplexního čísla fungoval musíme znovu přetížit operátor + s parametry v tomto pořadí.

```
public static KomplexniCislo operator+(KomplexniCislo a, KomplexniCislo b)
{
    double r = a.RealnaCast + b.RealnaCast;
    double i = a.ImaginarniCast + b.ImaginarniCast;
    return new KomplexniCislo(r, i);
}

Počet odkazů: 2
public static KomplexniCislo operator +(KomplexniCislo a, double b)
{
    return a + new KomplexniCislo(b, 0);
}

Počet odkazů: 0
public static KomplexniCislo operator +(double a, KomplexniCislo b)
{
    return b + a;
}
```

Obrázek 4.6: Přetížení operátoru + pro double a KomplexniCislo

Všimněte si, že pro vlastní výpočet využíváme již přetíženého operátoru + pro *KomplexniCislo* a double a ten zase využívá přetížený operátor pro *KomplexniCislo* a *KomplexniCislo*. Nyní si zkusíme analogickým způsobem přetížit operátor – pro dvě komplexní čísla a kombinace komplexního čísla a reálného čísla. Podobně jako součet se rozdíl dvou komplexních čísel se vypočítá tak, že odečteme zvlášť reálné složky a zvlášť imaginární složky.

```
public static KomplexniCislo operator -(KomplexniCislo a, KomplexniCislo b)
{
    double r = a.RealnaCast - b.RealnaCast;
    double i = a.ImaginarniCast - b.ImaginarniCast;
    return new KomplexniCislo(r, i);
}

Počet odkazů: 0
public static KomplexniCislo operator -(KomplexniCislo a, double b)
{
    return a - new KomplexniCislo(b, 0);
}

Počet odkazů: 0
public static KomplexniCislo operator -(double a, KomplexniCislo b)
{
    return new KomplexniCislo(a, 0) - b;
}
```

Obrázek 4.7: Přetížení operátoru -

Zde si jen všimněte, že u posledního přetížení nemůžeme využít již přetíženého operátoru – pro *KomplexniCislo* a double, protože u rozdílu záleží na pořadí, v jakém odčítáme. Operátor – má však jedno specifikum, že může figurovat jako unární operátor, tedy kdybychom napsali např. *-a*. Jedná se tedy o opačnou hodnotu daného čísla, kdy zaměníme znaménka u obou složek. Nicméně, abychom toto unární – mohli používat, musíme znovu přetížit operátor –, ale tentokrát bude mít pouze jeden parametr.

```
public static KomplexniCislo operator -(KomplexniCislo a, KomplexniCislo b)
{
    double r = a.RealnaCast - b.RealnaCast;
    double i = a.ImaginarniCast - b.ImaginarniCast;
    return new KomplexniCislo(r, i);
}
Počet odkazů: 0
public static KomplexniCislo operator -(KomplexniCislo a, double b)
{
    return a - new KomplexniCislo(b, 0);
}
Počet odkazů: 1
public static KomplexniCislo operator -(double a, KomplexniCislo b)
{
    return new KomplexniCislo(a, 0) - b;
}
Počet odkazů: 0
public static KomplexniCislo operator -(KomplexniCislo a)
{
    return 0 - a;
}
```

Obrázek 4.8: Přetížení operátoru unární -

Velkou výhodou přetěžování operátorů je, že mezi operátory fungují veškerá matematická pravidla, jako přednost operátorů, či používání závorek. Podívejme se nyní na malou ukázku, jak můžeme s těmito přetíženými operátory pracovat.

```
static void Main(string[] args)
{
    KomplexniCislo a = new KomplexniCislo(2, 3);
    KomplexniCislo b = new KomplexniCislo(4, 5);
    KomplexniCislo c = a + b;
    c.Vypis();
    KomplexniCislo d = a + 2;
    d.Vypis();
    KomplexniCislo e = 2 + a;
    e.Vypis();
    KomplexniCislo f = a - b;
    f.Vypis();
    KomplexniCislo g = a - 2;
    g.Vypis();
    KomplexniCislo h = 2 - a;
    h.Vypis();
    KomplexniCislo i = -a;
    i.Vypis();
    KomplexniCislo j = -b + c - (3 + g) + 5;
    j.Vypis();
}
```

Obrázek 4.9: Použití přetížených operátorů

PRO ZÁJEMCE

Zkuste si přetížit operátory pro násobení a dělení komplexních čísle. Pozor jen na to, že součin či dělení komplexní čísle není tak přímočarý jako u součtu/rozdílu. Pokud si vzorečky nepamatujete, vyhledejte je na internetu.

OTÁZKY

1. Co znamená polymorfismus?
    a. Můžeme deklarovat více tříd se stejným názvem, ale jiným počtem metod.

   b. Můžeme deklarovat více metod se stejným názvem, ale jiným počtem nebo typem parametrů.
   c. Můžeme deklarovat více proměnných se stejným názvem, ale jiným datovým typem.
2. Přetížení operátoru lze nahradit
   a. podmínkou
   b. cyklem
   c. metodou
3. Pokud přetížíme operátor +, můžeme zároveň s tímto operátorem používat závorky?
   a. ano
   b. ne
   c. jen někdy
4. Jaké klíčové slovo používáme pro přetížení operátoru
   a. operator
   b. virtual
   c. override
5. Přetížení operátoru je realizováno pomocí
   a. abstraktní metody
   b. virtuální metody
   c. statické metody
6. Příkaz switch se používá zejména
   a. Pokud testujeme jednu proměnnou na více hodnot
   b. Pokud se nám nelíbí příkaz *if*
   c. Když chceme nahradit cyklus *for*
7. Kolik operandů má ternární operátor?
   a. Jeden
   b. Dva
   c. Tři
8. Který příkaz nemůžeme použít u podmínky *if*?
   a. else
   b. case
   c. else if
9. Které klíčové slovo použijeme v příkazu *switch* pro možnost, která nevyhovuje žádné z uvedených možností?
   a. default
   b. else
   c. other
10. Kterým příkazem ukončujeme v příkazu switch každou větev programu?
   a. end
   b. stop
   c. break

SHRNUTÍ KAPITOLY

V této kapitole jsme se seznámili se základními příkazy pro podmínečné vykonávání části programu v závislosti na určitých podmínkách. Pomocí příkazu if můžeme specifikovat libovolnou podmínku, oproti tomu příkaz switch je specializovaný pro testování jedné proměnné na více hodnot. Dále jsme si ukázali, jak můžeme nahradit jednoduchou podmínku pomocí ternárního výrazu a tím tak podstatně zkrátit příslušný kód. Nakonec jsem se vysvětlili základy práce s logickými výrazy a operátory.

ODPOVĚDI

1. b
2. c
3. a
4. c
5. b
6. a
7. c
8. b
9. a
10. c

5 KNIHOVNY TŘÍD

RYCHLÝ NÁHLED KAPITOLY

Velmi často se setkáte s tím, že budete potřebovat, aby program určité příkazy vykonal opakovaně. Představte si situaci, kdy budete chtít vytvořit program, který uživateli umožní sečíst libovolné množství čísel. S obdobnou úlohou byste si již poradili za předpokladu, že by bylo pevně dáno, kolik čísel budete sčítat. Pokud počet čísel neznáte a chcete, aby program fungoval tak, že se po každém zadaném čísle zeptá, zdali chce uživatel zadat další, musíte se naučit další konstrukce jazyka C# umožňující určitou sadu příkazů vykonat opakovaně. Tyto konstrukce nazýváme cykly. Zároveň se naučíte používat nový datový typ pole, který vám umožní si do proměnné uložit v podstatě libovolné množství dat.

CÍLE KAPITOLY

Po prostudování této kapitoly budete umět:

- Vyjmenovat základní druhy cyklů a způsob jejich použití.
- Používat cykly pro opakované vykonání kódu.
- Vyřešit situaci, kdy potřebujete, aby cyklus proběhl za všech okolností alespoň jednou
- Deklarovat pole.
- Vypsat prvky pole.
- Provádět různé operace s prvky pole.
- Správně indexovat prvky pole.

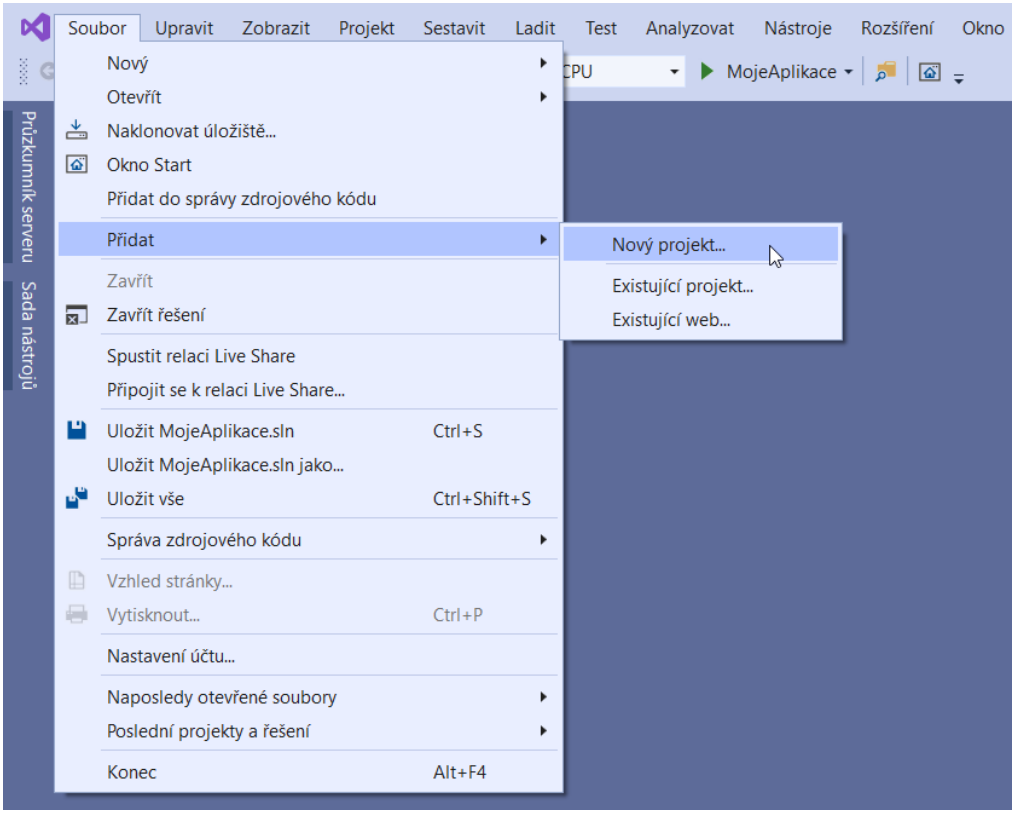
KLÍČOVÁ SLOVA KAPITOLY

Cyklus, iterace, iterační proměnná, pole, index.

5.1 Vytváření knihoven tříd

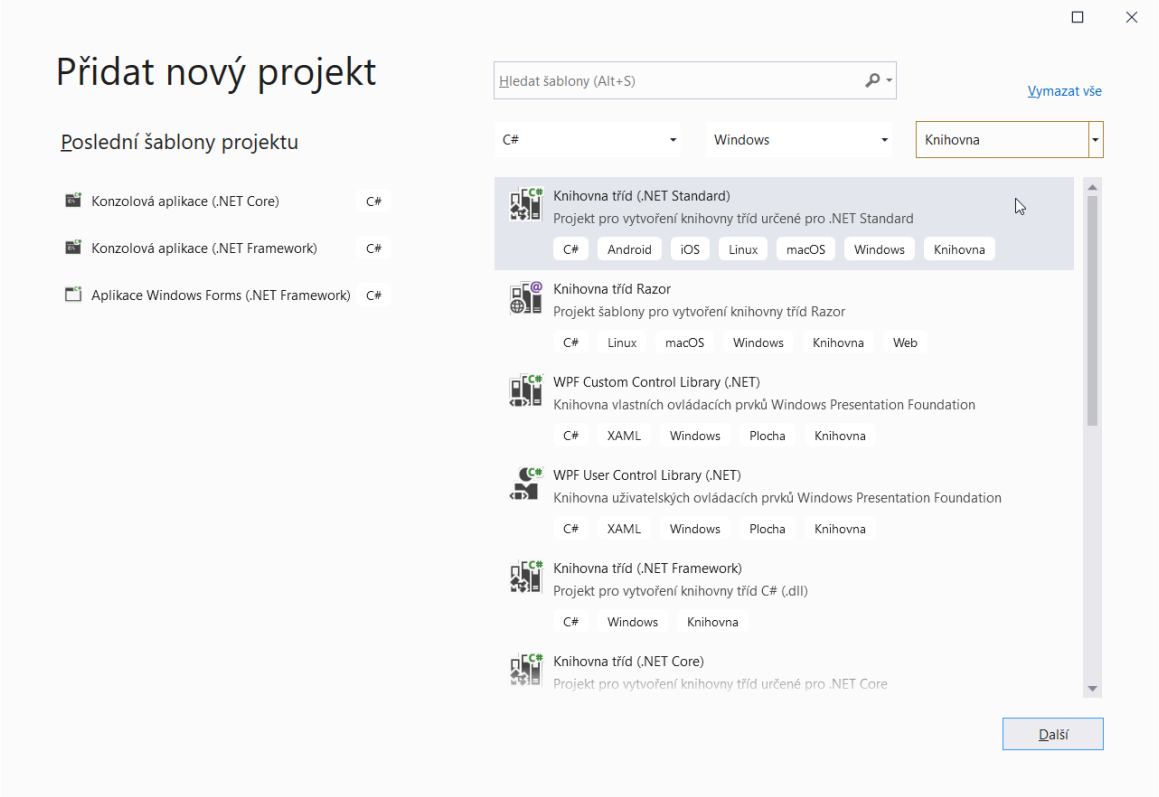
Prozatím jsme celý kód měli vždy pouze v jednom projektu. Pokud ale budeme pracovat na větším programu nebo budeme chtít určitou část našeho projektu opakovaně, je vhodné

umístit tuto část kódu do tzv. knihovny tříd. Knihovnu tříd vytvoříme v menu *Soubor→Přidat→Nový projekt*:



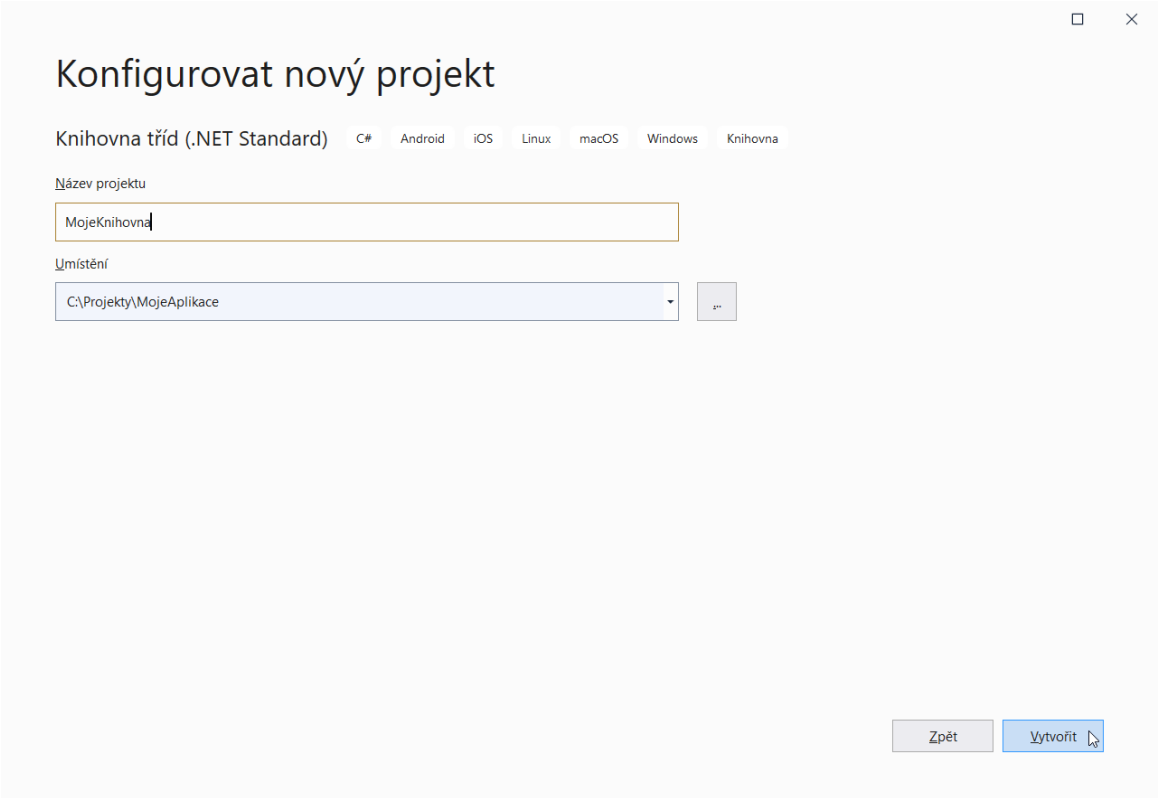
Obrázek 5.1: Nový projekt

Po potvrzení se nám zobrazí formulář, kde vybereme typ projektu, který chceme přidat. Ve filtrech zvolíme Windows, C# a Knihovna a ze seznamu nabízených typů vybereme Knihovna tříd .NET Standard:



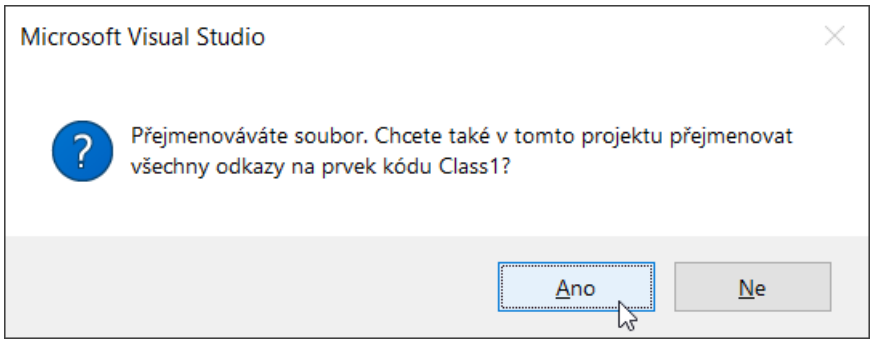
Obrázek 5.2: Knihovna tříd

Na dalším formuláři potom zvolíme název knihovny tříd, případně umístění na disku:



Obrázek 5.3: Název projektu

Po potvrzení tlačítkem Vytvořit se nám do řešení přidá tento nový projekt, což si můžeme zkontrolovat v Průzkumníku řešení. Vytvořená knihovna tříd obsahuje ve výchozím stavu prázdnou třídu s názvem *Class1*. Pro lepší orientaci je vhodné tuto třídu buď smazat a vytvořit novou s vhodným názvem, nebo přejmenovat. My zvolíme druhý způsob, tedy třídu přejmenujeme kliknutím pravým tlačítkem myši na název třídy v Průzkumníku řešení, zde uvedeme název *Kalkulacka*. Tímto dojde k přejmenování souboru, kde se tato třída nachází. Po potvrzení se nás ještě MS Visual Studio zeptá, zda chceme přejmenovat i samotnou třídu, kde zvolíme ano:



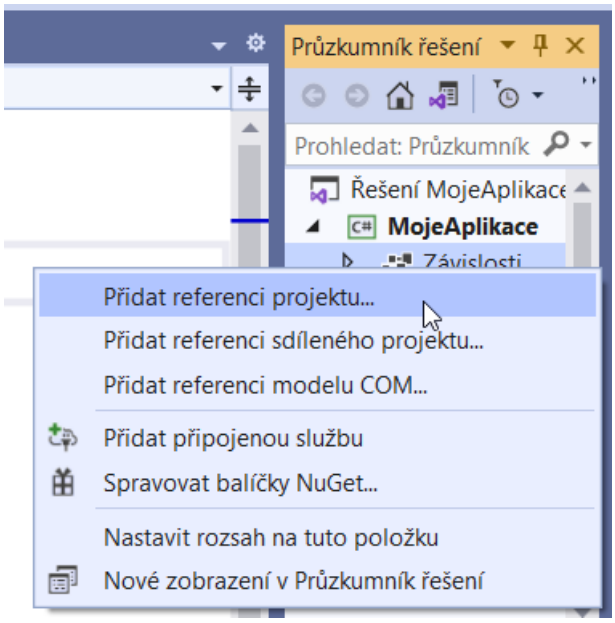
Obrázek 5.4: Přejmenování třídy

Ve třídě *Kalkulacka* si vytvoříme jednoduchou metodu pro součin dvou reálných čísel:

```
namespace MojeKnihovna
{
    Počet odkazů: 0
    public class Kalkulacka
    {
        Počet odkazů: 0
        public double Soucin(double a, double b)
        {
            return a * b;
        }
    }
}
```

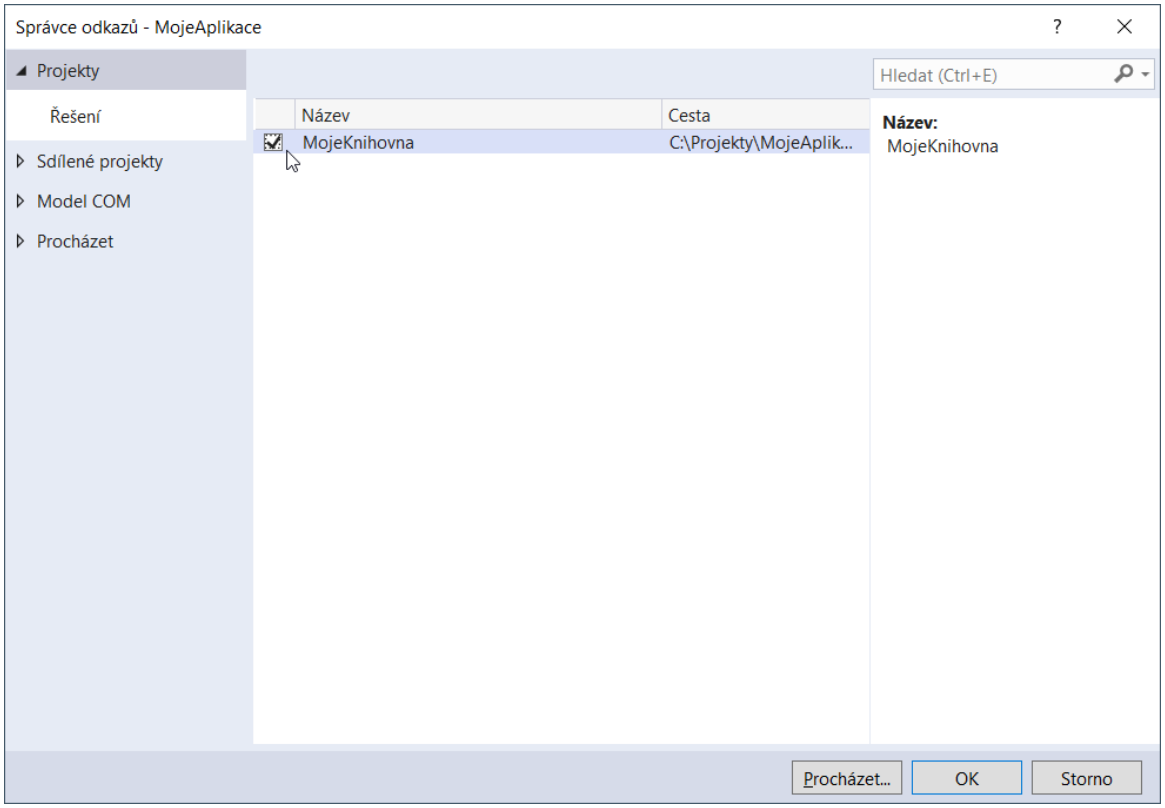
Obrázek 5.5: Třída Kalkulacka

Nyní bychom chtěli vytvořit instanci této třídy v hlavním programu. Pokud se však o to pokusíme, obdržíme syntaktickou chybu, že identifikátor *Kalkulacka* neexistuje. Je tomu tak proto, že třída *Kalkulacka* se nachází v jiném projektu a standardně třídy jednoho projektu nemají přístup k třídám jiného projektu, byť jsou součástí stejného řešení. Nejprve tedy musíme tedy tyto projekty tak, aby projekt *MojeAplikace* měl přístup k projektu *MojeKnihovna*. To uděláme tak, že v Průzkumníku řešení klikneme pravým tlačítkem na složku Závislosti v projektu *MojeAplikace* a zde volíme *Přidat referenci projektu*:



Obrázek 5.6: Přidání reference

Poté se nám zobrazí formulář, který nám umožní do projektu přidat referenci na jiný projekt nebo externí knihovnu. V levé části zvolíme typ projekty, jehož referenci chceme přidat, v našem případě to bude položka *Řešení* a poté vybereme projekt *MojeKnihovna*:



Obrázek 5.7: Správce odkazů

Po potvrzení si můžeme zkontrolovat, že odkaz na projekt by přidán do složky *Závislosti→Projekty*. Pokud se ovšem pokusíme spustit náš projekt, pořád obdržíme stejnou syntaktickou chyby. Problém je totiž ještě v tom, že každý z projektů se nachází v jiném jmenném prostoru. Musíme tedy pomocí příkazu *using* zpřístupnit jmenný prostor, v němž se nachází třída *Kalkulacka*, tedy *MojeKnihovna*:

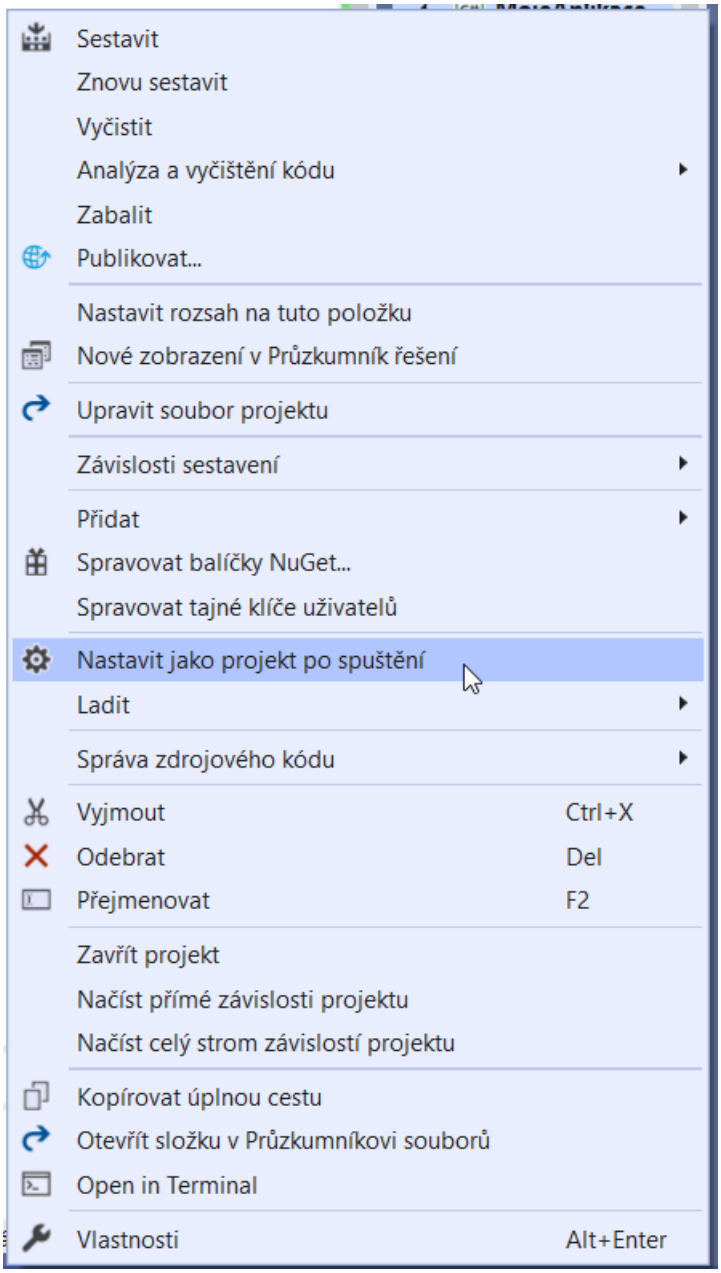
```
using System;
using MojeKnihovna;

namespace MojeAplikace
{
    Počet odkazů: 0
    class Program
    {
        Počet odkazů: 0
        static void Main(string[] args)
        {
            Kalkulacka k = new Kalkulacka();
            double vysledek = k.Soucin(2, 3);
            Console.WriteLine(vysledek);
        }
    }
}
```

Obrázek 5.8: Import jmenného prostoru

Nyní již bude program fungovat bez chyb. A nyní tak budeme mít přístup ke všem třídám nacházejících se v projektu *MojeKnihovna*. Je třeba si jen dát pozor, že pokud budeme do projektu *MojeKnihovna* přidávat novou třídu, musíme před deklaraci třídy umístit modifikátor přístupu *public* tak, jak je tomu u třídy *Kalkulacka*. Bez uvedení modifikátoru přístupu jsou totiž třídy přístupné pouze v rámci projektu, kde jsou deklarovány.

V rámci jednoho řešení můžeme mít libovolný počet projektů a nemusí to být pouze knihovna tříd, ale obecně libovolný typ, tedy můžeme mít i více spustitelných projektů. V jednu chvíli však může být jako spustitelný nastavený pouze jeden projekt v řešení. Ten poznáme tak, že je v Průzkumníku řešení zvýrazněn tučně. Pokud potřebujeme jako spustitelný označit jiný projekt, provedeme to kliknutím pravým tlačítkem myši na projekt, který chceme nastavit jako spustitelný a zvolíme *Nastavit jako projekt po spuštění*:



Obrázek 5.9: Nastavení spustitelného projektu

Takto můžeme spouštěný projekt měnit opakovaně.

5.2 Dokumentace

Již jsme si ukázali, jak můžeme dokumentovat náš kód pomocí komentářů, a to buď jednořádkových nebo víceřádkových. Nevýhodu těchto komentářů je, že slouží pouze pro naši potřebu a MS Visual Studio je ignoruje. Existují však i XML komentáře, které slouží nejen pro naši potřebu, ale které dokáže přečíst i MS Visual Studio a zobrazí nám kontextovou nápovědu při psaní identifikátoru, jemuž jsme přiřadili XML komentář. XML komentář vytvoříme tak, že umístíme kurzor na prázdný řádek nad deklarací příslušné metody, třídy či proměnné a napíšeme za sebou tři lomítka ///. To způsobí že se automaticky

vygeneruje kostra XML komentáře s předvyplněnými prvky, které mají pro daný identifikátor smysl. Zkusíme si tyto komentáře demonstrovat na okomentování třídy *Kalkulacka* v knihovně tříd *MojeKnihovna*. Kurzor tedy umístíme na prázdný řádek nad řádkem „*class Kalkulacka*“ a napíšeme zde tří lomítka, tím se nám vygeneruje XML kostra s prvkem *<summary></summary>*. Mezi značky *<summary>* a *</summary>* umístíme samotný komentář k třídě *Kalkulacka*:

```
namespace MojeKnihovna
{
    /// <summary>
    /// Třída s metodami pro základní matematické operace
    /// </summary>
    Počet odkazů: 1
    public class Kalkulacka
    {
        Počet odkazů: 1
        public double Soucin(double a, double b)
        {
            return a * b;
        }
    }
}
```

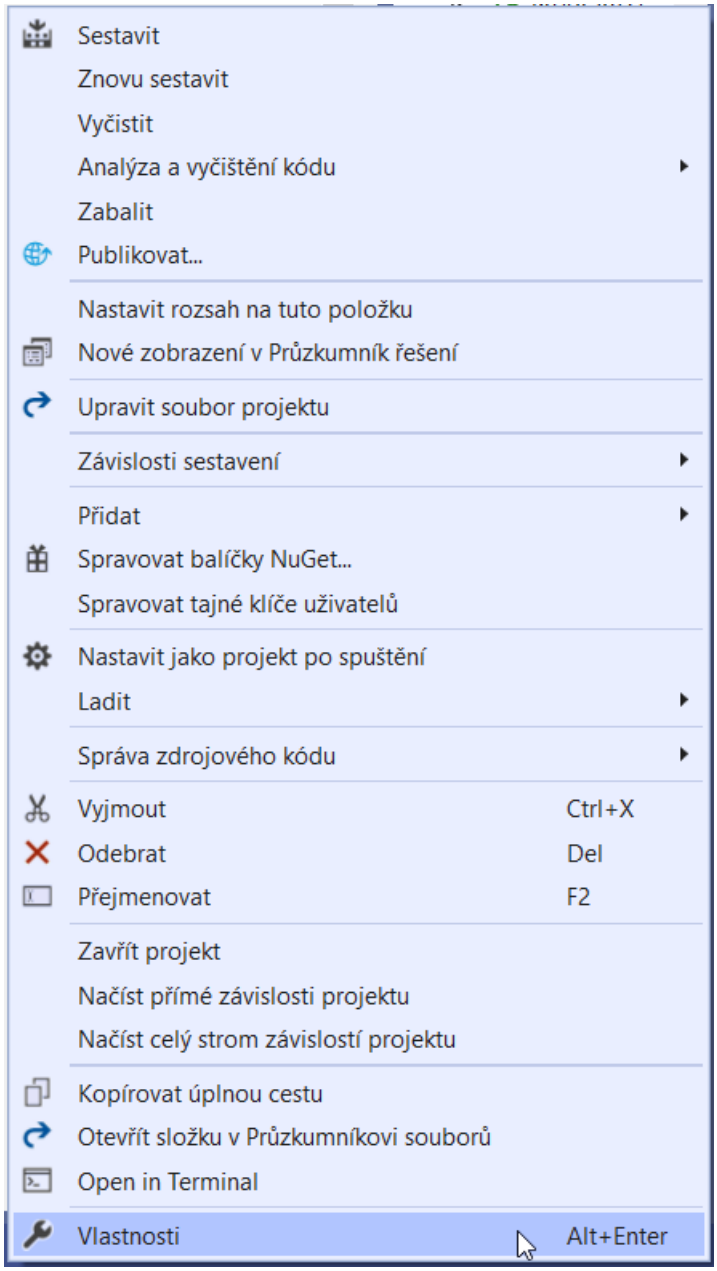
Obrázek 5.10: XML komentář třídy

Analogickým způsobem vložíme komentář pro metodu *Soucin*, umístíme kurzor opět na prázdný řádek nad deklarací metody a napíšeme tři lomítka. Tentokrát se nám vygeneruje poněkud delší kostra XML komentáře obsahující nejen prvek *summary*, ale i *param* umožňující popsat jednotlivé parametry a prvek *returns*, který popisuje návratovou hodnotu metody:

```
/// <summary>
/// Třída s metodami pro základní matematické operace
/// </summary>
Počet odkazů: 1
public class Kalkulacka
{
    /// <summary>
    /// Vypočítá součin dvou reálných čísel
    /// </summary>
    /// <param name="a">První číslo součinu</param>
    /// <param name="b">Druhé číslo součinu</param>
    /// <returns>Vrací výsledek součinu</returns>
    Počet odkazů: 1
    public double Soucin(double a, double b)
    {
        return a * b;
    }
}
```

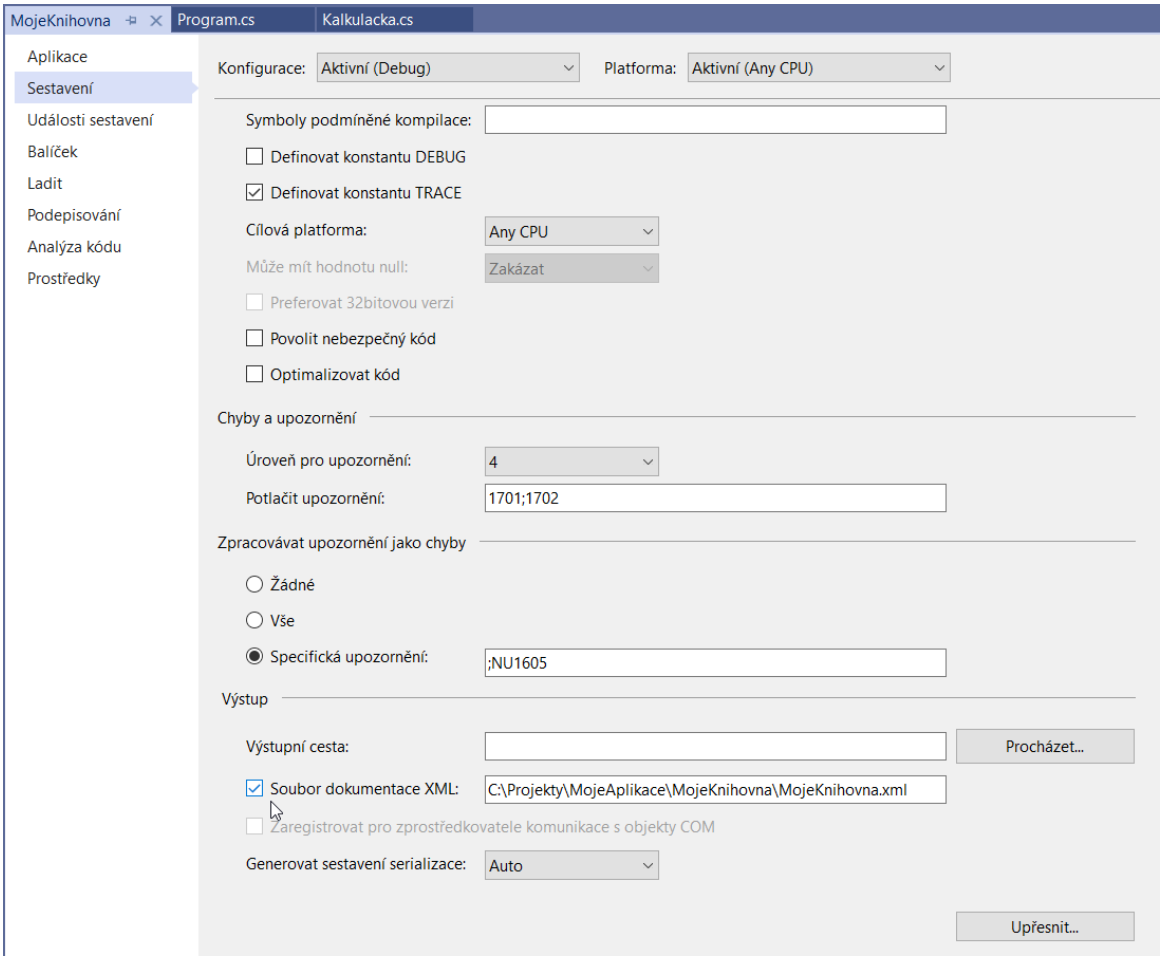
Obrázek 5.11: XML komentář metody

Nyní kdekoliv v kódu napíšeme název třídy *Kalkulacka* nebo metody *Soucin*, zobrazí se nám kontextová nápověda k této třídě a metodě. Tuto nápovědu můžeme rovněž vyvolat najetím myši nad daný identifikátor. To však není jediná výhoda XML komentářů. Pokud je budeme používat důsledně, můžeme pak velmi snadno automaticky vygenerovat dokumentaci k celému projektu. To provedeme tak, že klikneme pravým tlačítkem v Průzkumníku řešení na projekt MojeKnihovna a vybereme Vlastnosti:



Obrázek 5.12: Vlastnosti projektu

Zobrazí se nám formulář s různými vlastnostmi projektu. Nás bude zajímat vlevo karta *Sestavení*, kde potom dole zaškrtneme *Soubor dokumentace XML*:



Obrázek 5.13: Soubor dokumentace XML

Nyní program znovu spustíme, zobrazí se nám v průzkumníku řešení ve složce projektu *MojeKnihovna* nový soubor MojeKnihovna.xml. Po jeho otevření se nám zobrazí XML se všemi komentáři v našem projektu:

```xml
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>MojeKnihovna</name>
    </assembly>
    <members>
        <member name="T:MojeKnihovna.Kalkulacka">
            <summary>
            Třída s metodami pro základní matematické operace
            </summary>
        </member>
        <member name="M:MojeKnihovna.Kalkulacka.Soucin(System.Double,System.Double)">
            <summary>
            Vypočítá součin dvou reálných čísel
            </summary>
            <param name="a">První číslo součinu</param>
            <param name="b">Druhé číslo součinu</param>
            <returns>Vrací výsledek součinu</returns>
        </member>
    </members>
</doc>
```

Obrázek 5.14: XML dokumentace

Pokud byste potřebovali dokumentaci v jiném formátu, existuje celá řada nástrojů, např. *SandCastle* nebo *Doxygen*, které umožňují převést tento XML soubor do jiného formátu.

OTÁZKY

1. Který cyklus použijeme, pokud chceme, aby se provedl alespoň jednou?
    a. for
    b. while
    c. do-while
2. Který cyklus nepoužívá ukončující podmínku?
    a. foreach
    b. for
    c. while
3. První prvek pole má hodnotu indexu rovnu
    a. 1
    b. 0
    c. U deklarace pole můžeme definovat hodnotu počátečního indexu pole
4. Co znamená, když v cyklu *for* napíšeme *i*++ ?
    a. Hodnota proměnné *i* se zvýší o 1
    b. Hodnota proměnné *i* se zvýší o 2
    c. Jedná se o syntaktickou chybu
5. Cyklus *while* se bude provádět, dokud je podmínka
    a. Pravdivá

   b. Nepravdivá
   c. Cyklus *while* proběhne vždy pouze jednou
6. Jednotlivé části cyklu *for* oddělujeme
   a. Čárkou
   b. Středníkem
   c. Mezerou
7. Které klíčové slovo používáme v cyklu *foreach*?
   a. in
   b. inside
   c. with
8. Je-li *cisla* proměnná typu pole celých čísel, jak zjistíme maximální počet čísel, které můžeme do tohoto pole uložit?
   a. cisla.Size
   b. cisla.Width
   c. cisla.Length
9. Máme-li cyklus *for* s iterační proměnnou *i* s výchozí hodnotou 1, co uvedeme do poslední částí cyklu, pokud chceme projít pouze liché hodnoty proměnné *i*?
   a. i++
   b. i++2
   c. i+=2
10. Máme-li deklarované pole celých čísel *cisla* o kapacitě 5 prvků a napíšeme příkaz *cisla*[5]=3, co se stane?
   a. Poslednímu prvku se přiřadí hodnota 3
   b. Program se nespustí
   c. Program se spustí, ale skončí chybou

SHRNUTÍ KAPITOLY

V této kapitole jsme si ukázali, jak opakovat vykonání určité části kódu. U každého cyklu jsme si uvedli základní charakteristiky a srovnali z jinými druhy cyklů. Dále jsme si ukázali nový datový typ pole, které se používá pro uložení více hodnot stejného typu, ke kterým můžeme přistupovat pomocí indexu. Nakonec jsme se seznámili s cyklem *foreach*, který je specializovaný pro procházení polí.

ODPOVĚDI

1. c
2. a
3. b

4. a
5. a
6. b
7. a
8. c
9. a
10. c

6 GRAFICKÉ UŽIVATELSKÉ ROZHRANÍ

RYCHLÝ NÁHLED KAPITOLY

Program, který bude dělat pořád totéž, nebude zřejmě úplně užitečný. Každý programovací jazyk proto nabízí možnosti, jak komunikovat s okolím a na základě toho měnit své chování. Zde si ukážeme, jaké jsou základní možnosti načtení vstupu od uživatele a jak potom získat výstupy z programu. Seznámíme se zejména s formátovaným výstupem na obrazovku, načítáním vstupu od uživatele z klávesnice, ukážeme si základy práce se soubory a nakonec si ukážeme, jak jednoduše můžeme předávat parametry na příkazovém řádku.

CÍLE KAPITOLY

Po prostudování této kapitoly budete umět:

- Zobrazit výsledek programu na obrazovku.
- Zobrazený výstup vhodně zformátovat.
- Načíst a zpracovat vstup od uživatele z klávesnice.
- Vytvořit soubor a zapsat do něho text.
- Načíst data z existujícího souboru.
- Ověřit, zda soubor existuje na disku.
- Používat parametry na příkazovém řádku.

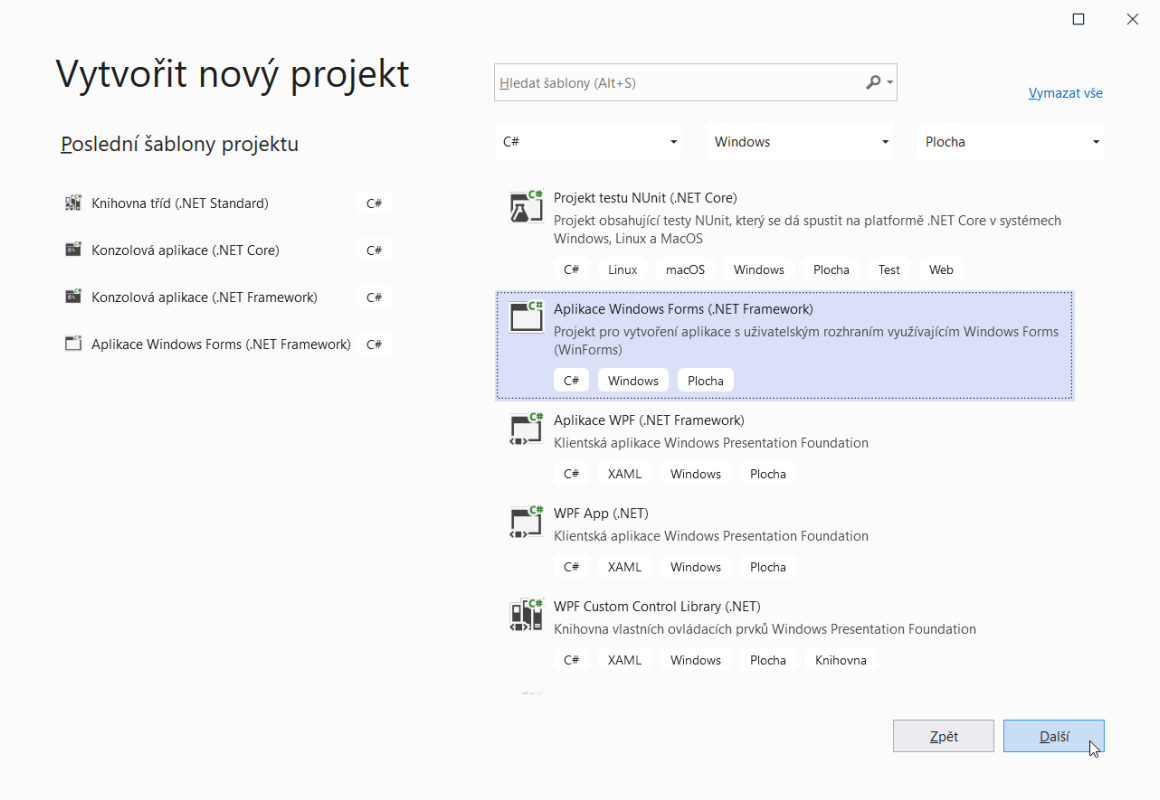
KLÍČOVÁ SLOVA KAPITOLY

Vstup, výstup, konzole, formát, soubor, parametr.

6.1 Vytvoření formulářové aplikace

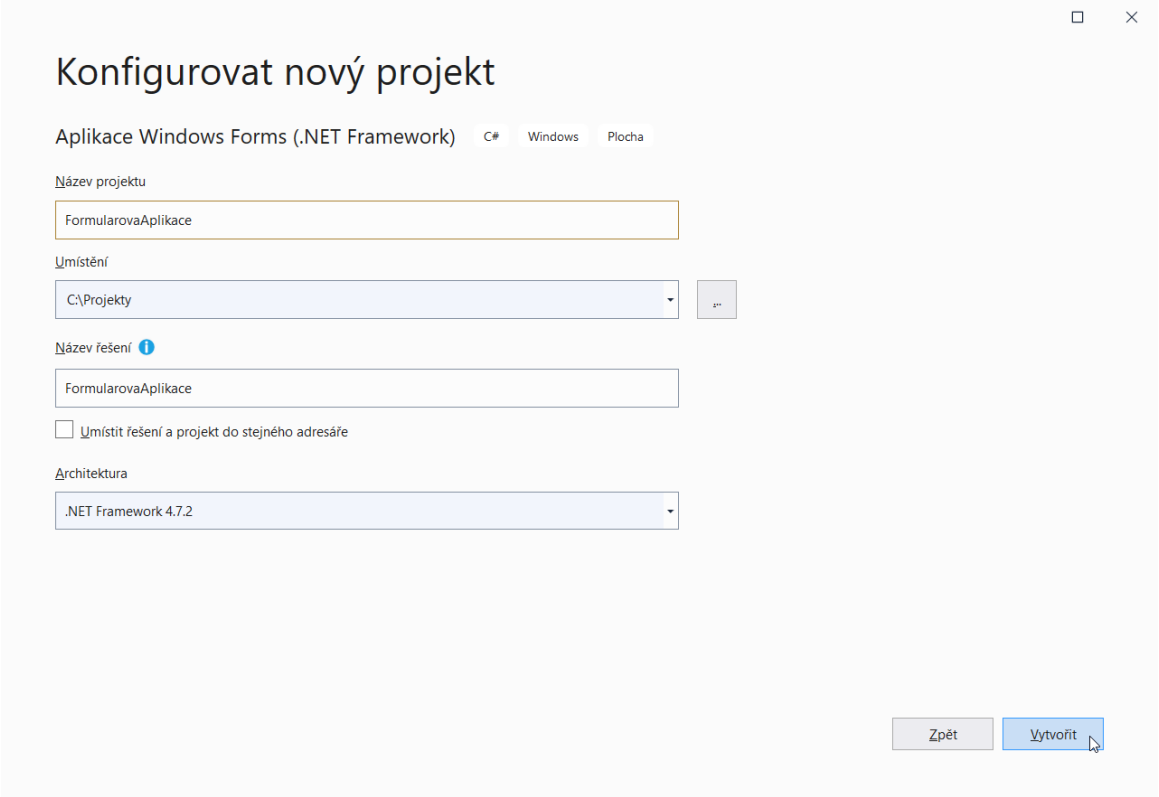
Dosud jsme všechny programy vytvářeli jako konzolové aplikace, tedy veškerý vstup a výstup byl pouze textový. Konzolové aplikace se hodí zejména pro systémové aplikace nebo pro aplikace běžící na pozadí. Pro praktické použití je obvykle je vhodné, aby aplikace byla uživatelsky přívětivá a měla nějaké grafické rozhraní pro zadávání vstupů a zobrazování výstupů. Nejprve si ukážeme, jak vytvořit formulářovou aplikaci v MS Visual Studio.

V menu zvolíme *Soubor→Nový→Projekt* a vybereme typ projektu *Aplikace Windows Forms (.NET Framework)*:



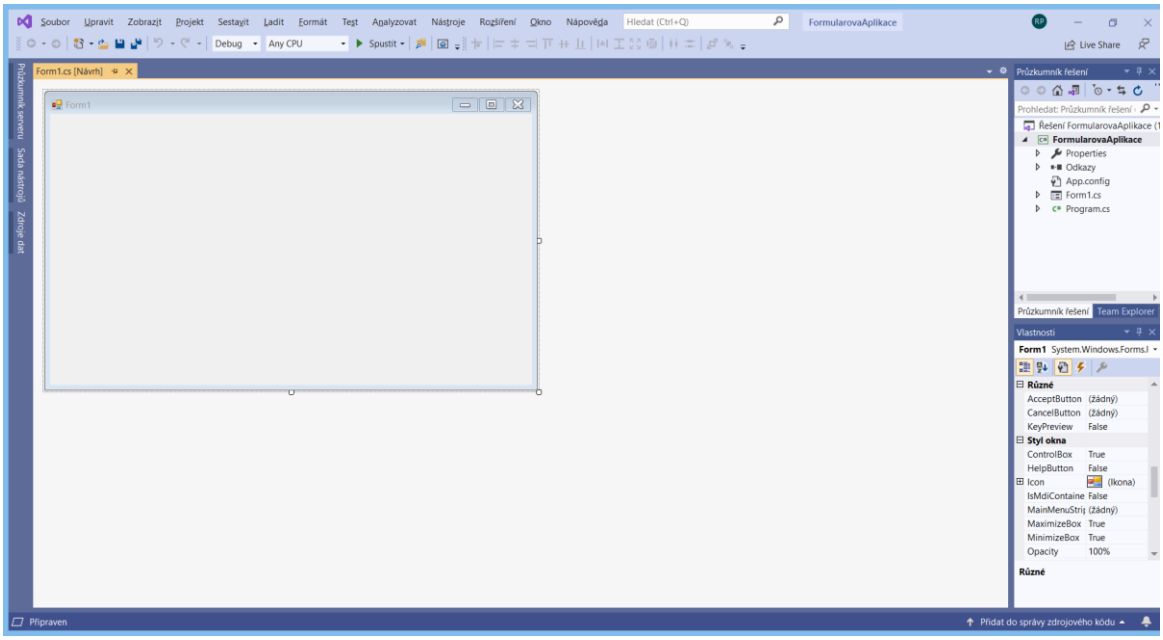
Obrázek 6.1: Nová formulářová aplikace

Na dalším formuláři potom zadáme název projektu, případně jeho umístění na disku:



Obrázek 6.2: Název projektu

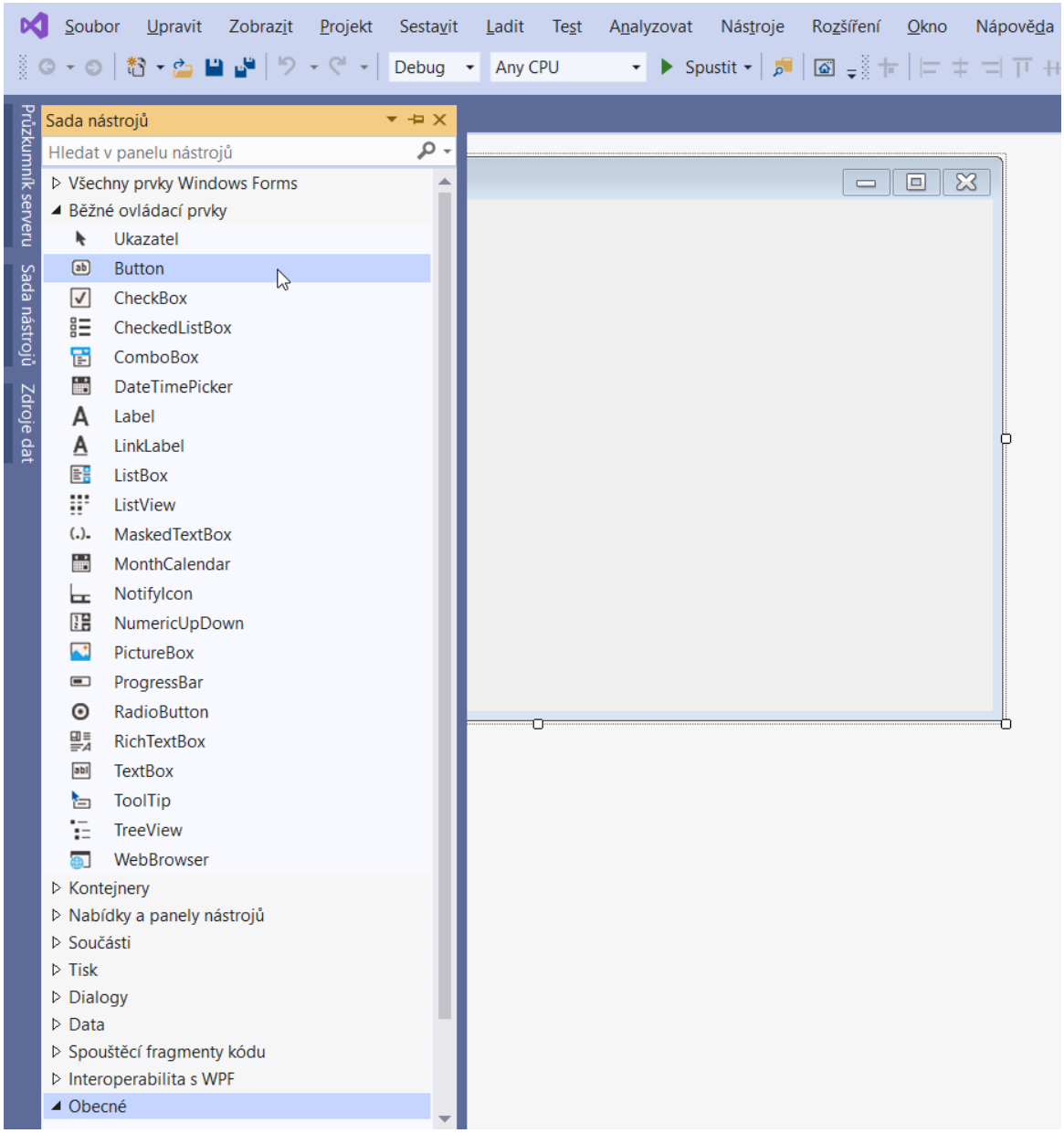
Po potvrzení se nám vytvoří formulářová aplikace s prázdným formulářem, na který můžeme vkládat různé ovládací prvky jako např. tlačítko, textové pole, popisek apod.



Obrázek 6.3: Nová formulářová aplikace

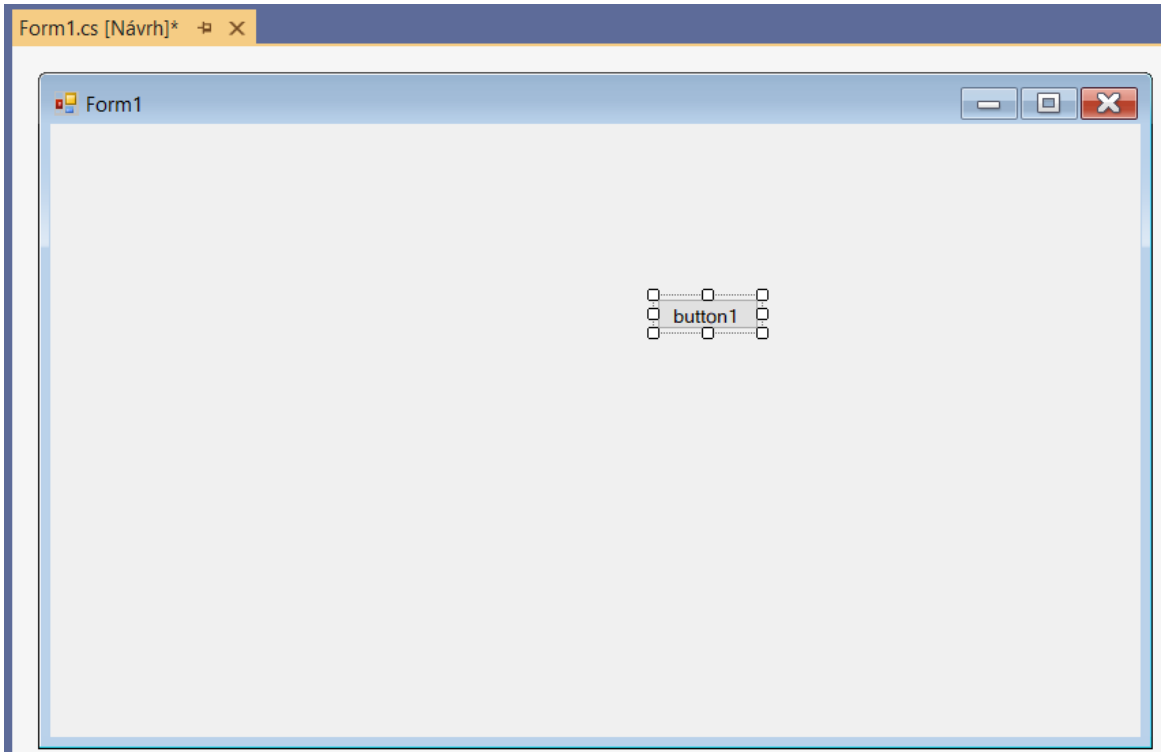
Podobně jako jsme začínali u konzolové aplikace i u formulářové aplikace si ukážeme, jak vypsat na obrazovku text „Hello World“. Nejprve vložíme na formulář tlačítko, které

najdeme na kartě Sada nástrojů umístěné vlevo nahoře. Po jejím rozkliknutí se nám zobrazí všechny dostupné ovládací prvky, které jsou umístěny do kategorií. Pro naše účely si vystačíme s kategorií Běžné ovládací prvky, kde najdeme i tlačítko – Button:



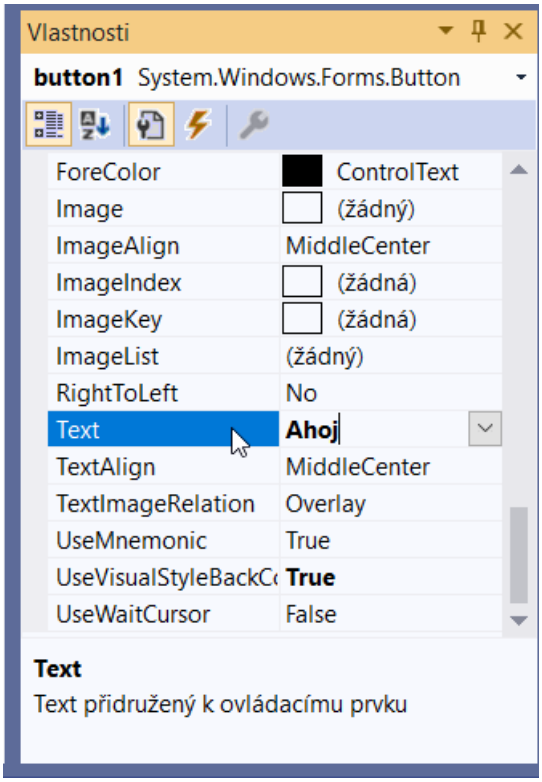
Obrázek 6.4: Ovládací prvky

Následně tlačítko přetáhneme na libovolné místo ve formuláři:



Obrázek 6.5: Vložení tlačítka

Vložené tlačítko můžeme libovolně přesouvat, či měnit jeho velikost. Každý ovládací prvek má řadu vlastností, které můžeme měnit. Mezi ty nejpoužívanější patří *Text* – popisek, *ForeColor* – barva písma, *BackColor* – barva pozadí, *Font* – písmo, *Enabled* – zda je prvek aktivní. Tyto a ostatní vlastnosti můžeme nastavit v okně *Vlastnosti*. V našem případě změníme popisek tlačítka na „Ahoj“:



Obrázek 6.6: Vlastnosti

Pokud nyní aplikaci spustíme, uvidíme sice na formuláři tlačítko s popiskem Ahoj, nicméně po kliknutí se nic nestane. Aby tlačítko mělo nějakou funkcionalitu, musíme tomuto tlačítku přiřadit událost, tj. metodu která se spustí, při kliku na toto tlačítko. Nejjednodušší způsob přiřazení události na kliknutí na tlačítko je v návrhovém formuláři dvakrát kliknout na tlačítko, čímž se nám vygeneruje prázdná metoda *button1_Click*:

```
namespace FormularovaAplikace
{
    Počet odkazů: 2
    public partial class Form1 : Form
    {
        Počet odkazů: 1
        public Form1()
        {
            InitializeComponent();
        }

        Počet odkazů: 1
        private void button1_Click(object sender, EventArgs e)
        {

        }
    }
}
```

Obrázek 6.7: Událost button1_Click

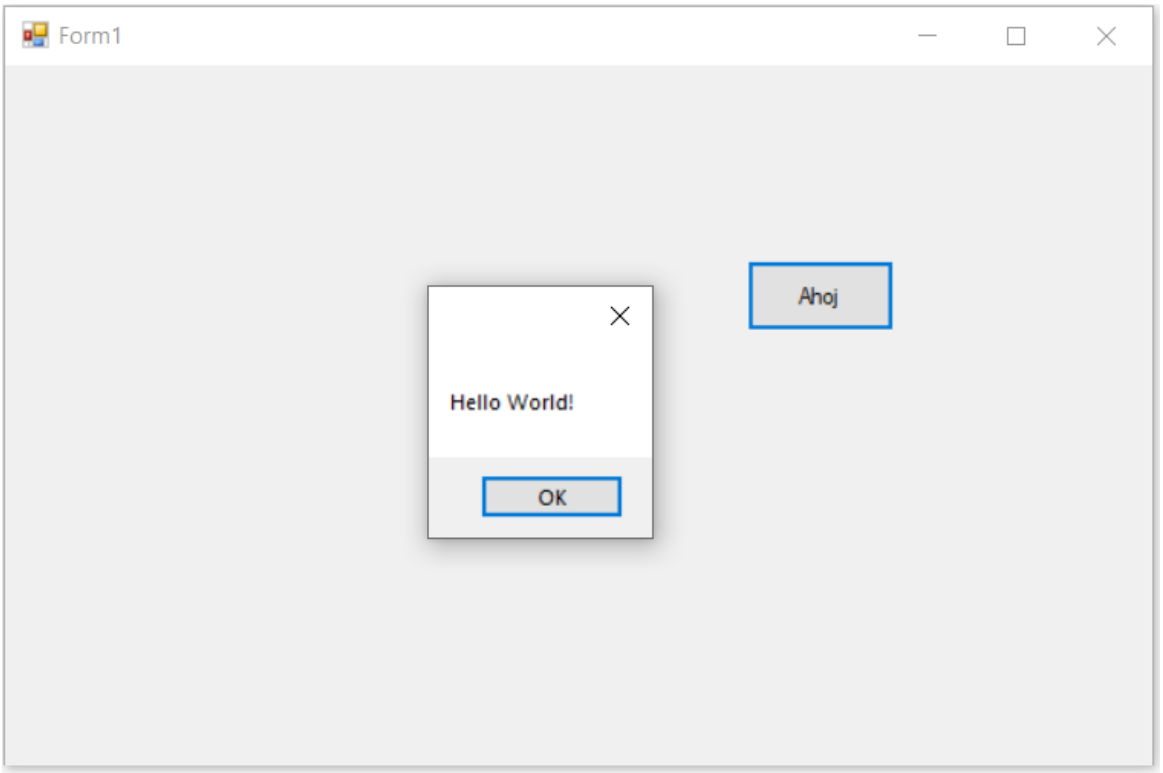
Jméno metody je tvořeno názvem ovládacího prvku a druhem události. Dále si můžeme všimnout, že metoda má dva parametry. Parametr *sender* obsahuje objekt, který událost vyvolal a druhý parametr *e* obsahuje další doplňující informace o události. Prozatím tyto parametry budeme ignorovat a zaměříme se pouze na tělo metody, kde doplníme kód, který se má po stisku tlačítka vykonat. Pokud bychom pro výpis na obrazovku použili metodu *Console.WriteLine*, program by sice neohlásil žádnou chybu, ale ani by nic nezobrazil. *Console.WriteLine* slouží totiž pouze pro konzolové aplikace. K vypisování stručných informací používáme pro konzolové aplikace metodu *MessageBox.Show*. Tělo metody *button1_click* bude tedy vypadat následovně:

```
public partial class Form1 : Form
{
    Počet odkazů: 1
    public Form1()
    {
        InitializeComponent();
    }

    Počet odkazů: 1
    private void button1_Click(object sender, EventArgs e)
    {
        MessageBox.Show("Hello World!");
    }
}
```

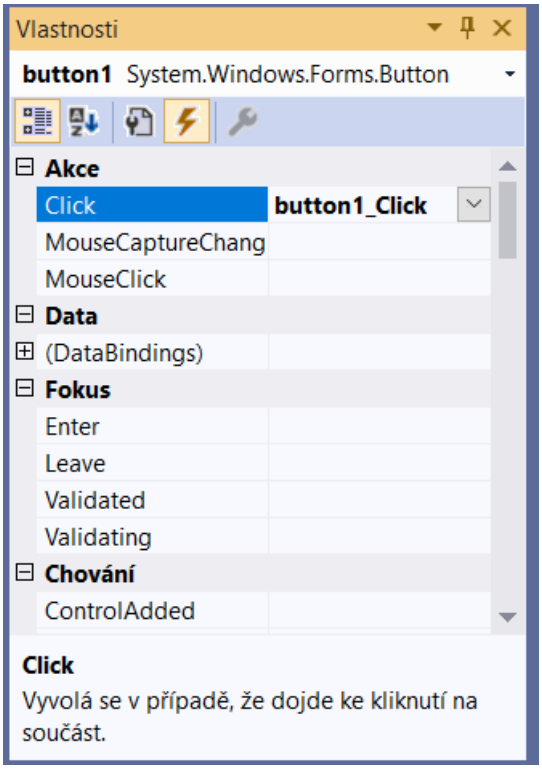
Obrázek 6.8: Výpis na obrazovku

Po spuštění aplikace a kliknutí na tlačítko obdržíme:



Obrázek 6.9: Hello World aplikace

Zkusme se ještě podrobněji podívat na vytváření událostí. Událost na kliknutí je pouze jednou z mnoha událostí. Další události můžeme vidět, pokud v okně Vlastnosti klikneme na ikonu blesku:



Obrázek 6.10: Události

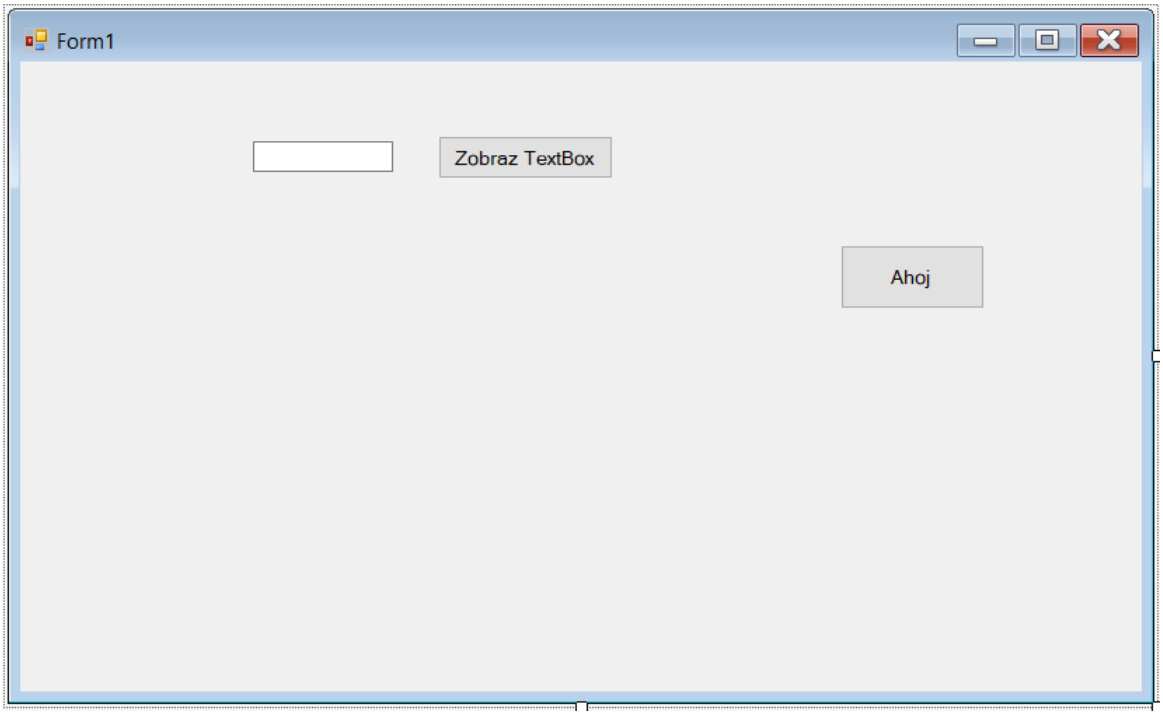
Vidíme, že pro akci *Click* je definována metoda *butto1_Click*, zároveň ve spodní části okna vidíme nápovědu k aktuálně vybrané události. Pokud tedy hledáme nějakou událost, jejímž názve si nejsme jisti, můžeme projít tímto seznamem a podívat se na popis příslušné události. Některé často používané události si ukážeme v následující části při popisu jednotlivých ovládacích prvků. Pokud potřebujeme přiřadit některou událost, potom stačí dvakrát kliknout do prázdného místa vpravo od názvu události. Pokud se potřebujeme dostat zpátky k seznamu vlastností daného ovládacího prvku, provedeme to pomocí ikony klíče vlevo od ikony blesku.

6.2 Základní ovládací prvky

Nyní si ukážeme, některé často používané ovládací prvky a vždy si ukážeme jednoduchou ukázku, jak je můžeme v aplikaci využít.

6.2.1 TextBox

Tento ovládací prvek použijeme, pokud chceme od uživatele načíst nějaký vstupní text. Zkusíme si nyní demonstrovat použití textového pole na příkladu, kdy budeme mít na formuláři textové pole a tlačítko a po stisknutí tlačítka se nám zobrazí řetězec zadaný v textovém poli. Nejprve tedy přetáhneme na formulář textové pole a tlačítko z okna Sada nástrojů:



Obrázek 6.11: Textové pole

Dále dvojklikem na tlačítko vygenerujeme událost, která se vykoná po stlačení tlačítka a do těla metody vložíme kód, kterým vypíšeme obsah textového pole na obrazovku:

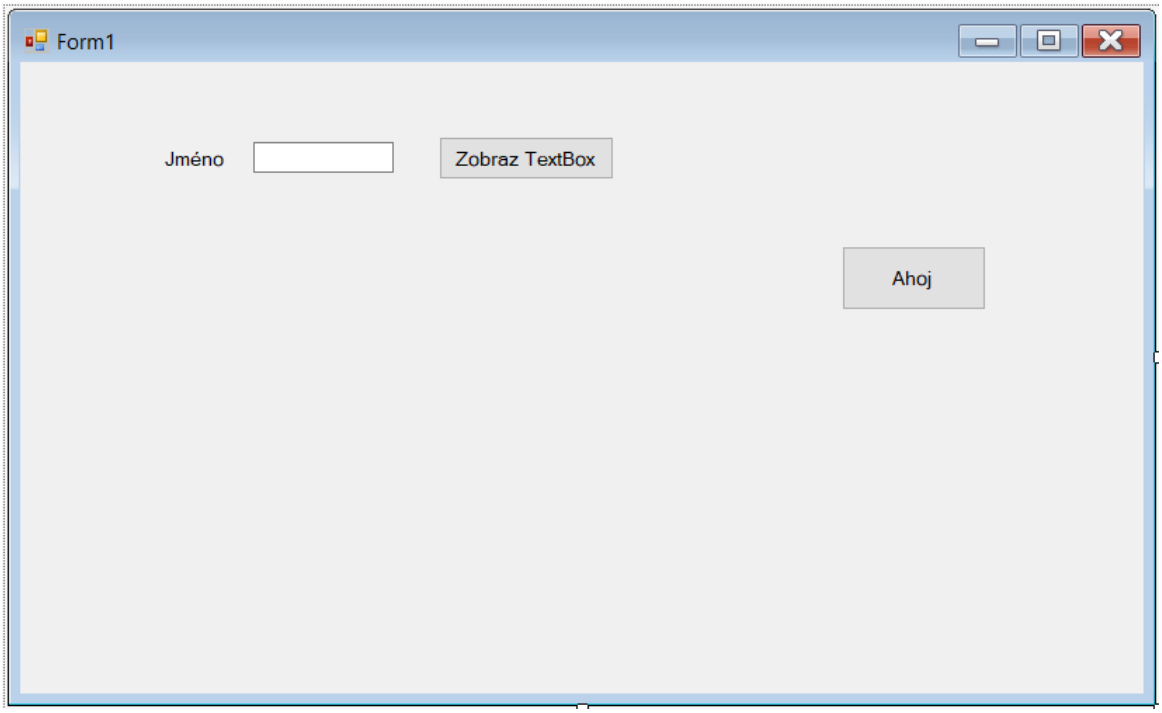
```
private void button2_Click(object sender, EventArgs e)
{
    MessageBox.Show(textBox1.Text);
}
```

Obrázek 6.12: Vypsání obsahu textového pole

Všimněte si, že obsah textového pole je uložen ve vlastnosti *Text*.

6.2.2 LABEL

Tento ovládací prvek používáme pro popis jednotlivých ovládacích prvků. Pokud by se například na formuláři nacházelo více textových polí, bylo by obtížné rozlišit, k čemu příslušné textové pole slouží. Zkusíme si tedy vložit k textovému poli na formuláři popisek „Jméno“. Ze sady nástrojů tedy přetáhneme Label na formulář vedle textového pole. Jako výchozí popisek se nám zobrazí label1, který přepíšeme ve vlastnosti Text na „Jméno“:

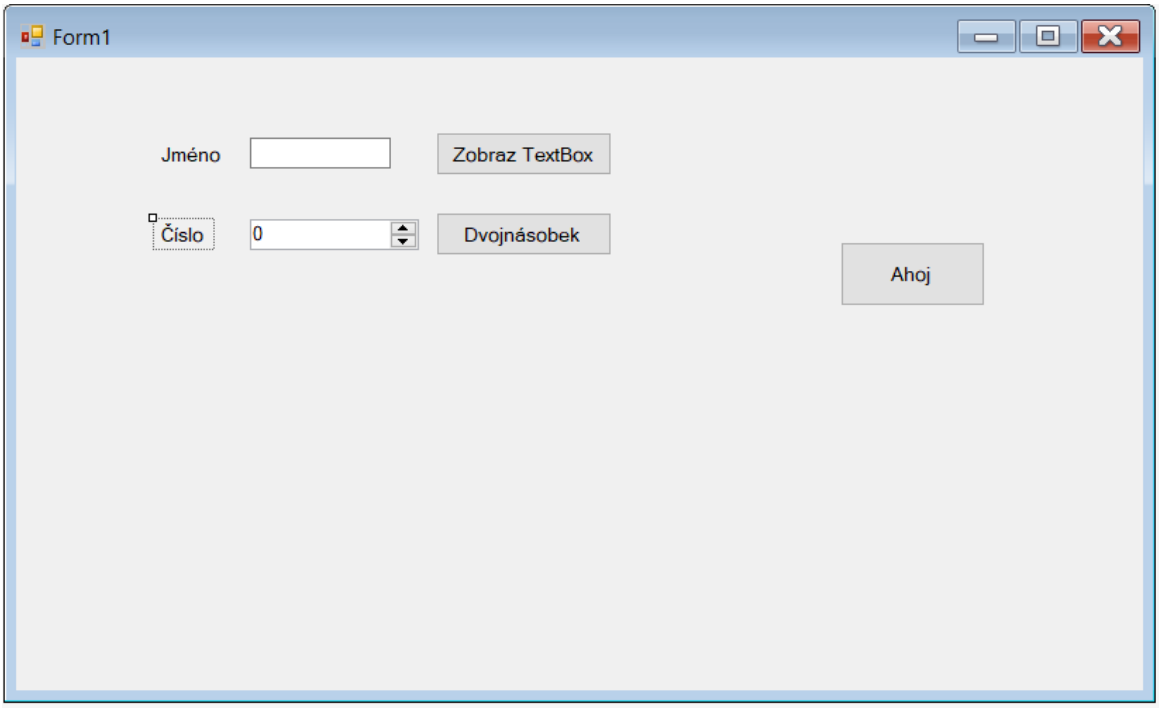


Obrázek 6.13: Popisek

U popisku už ze své podstaty nepoužíváme žádné události, i když smysl může mít například zobrazení nápovědy při kliknutí na popisek.

6.2.3 NUMERICUPDOWN

Tento ovládací prvek je velice podobný textovému poli, ale je určený pro zadávání číselných hodnot. Odpadá nám tím potřeba konverze z řetězce na číslo nebo kontrola, zda zadaný řetězec je číslo. Zkusíme si nyní demonstrovat použití numerického pole na příkladu pro zobrazení dvojnásobku zadané hodnoty. Na formulář si přetáhneme numerické pole s popiskem a tlačítko:



Obrázek 6.14: Numerické pole

Dále dvojklikem na tlačítko vygenerujeme událost, která se vykoná po stlačení tlačítka a do těla metody vložíme kód, který načte hodnotu z numerického pole do pomocné proměnné, vypočítá dvojnásobek této hodnoty a vypíše výsledek na obrazovku:

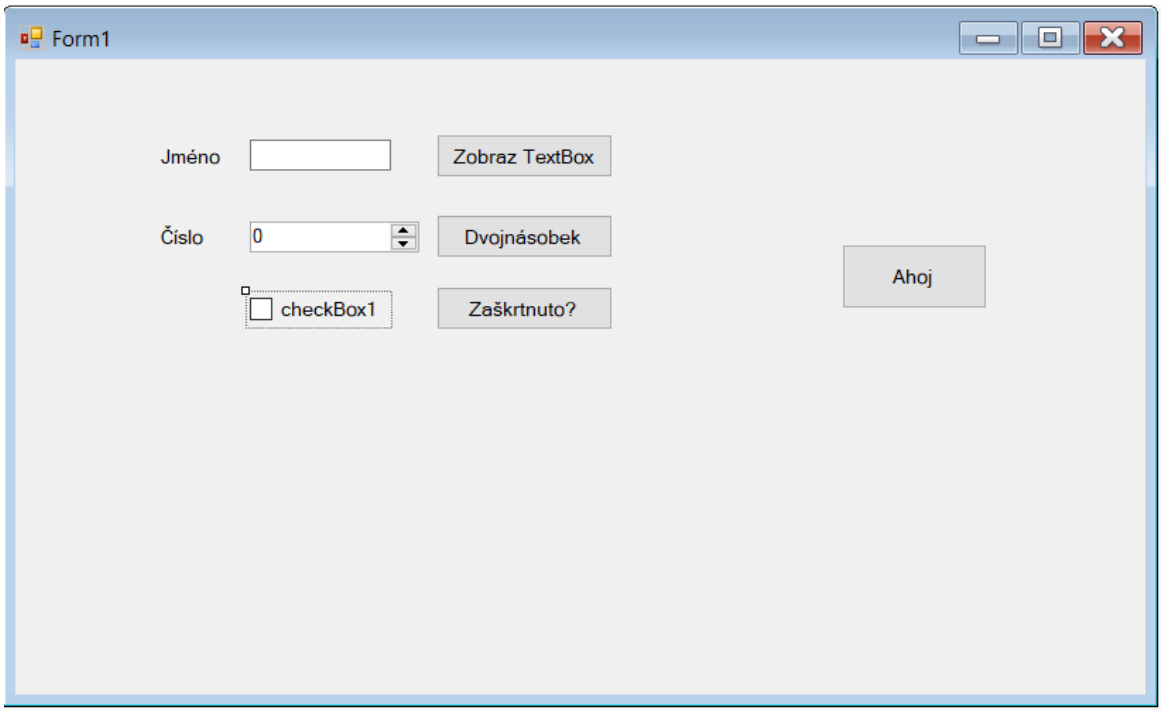
```
private void button3_Click(object sender, EventArgs e)
{
    double cislo = (double)numericUpDown1.Value;
    double vysledek = 2 * cislo;
    MessageBox.Show(vysledek.ToString());
}
```

Obrázek 6.15: Přístup k numerickému poli

Zadaná hodnota je u numerického pole uložena ve vlastnosti *Value*. Tady je třeba si jen dát pozor na to, že je datového typu *decimal*, takže pokud potřebujeme pracovat s typem *int* nebo *double*, musíme provést přetypování.

6.2.4 CheckBox

Tento ovládací prvek používáme pro zadávání vstupních dat, které mohou nabývat pouze dvou hodnot, typicky ano/ne. Jeho použití si ukážeme na příkladu, kdy po kliknutí na tlačítko se nám zobrazí informace, zda *CheckBox* je zaškrtnuto či nikoli.



Obrázek 6.16: CheckBox

Dále dvojklikem na tlačítko vygenerujeme událost, která se vykoná po stlačení tlačítka a do těla metody vložíme kód, který zjistí, zda je zaškrtávací políčko zaškrtnuto či nikoli a tuto informaci vypíše výsledek na obrazovku:

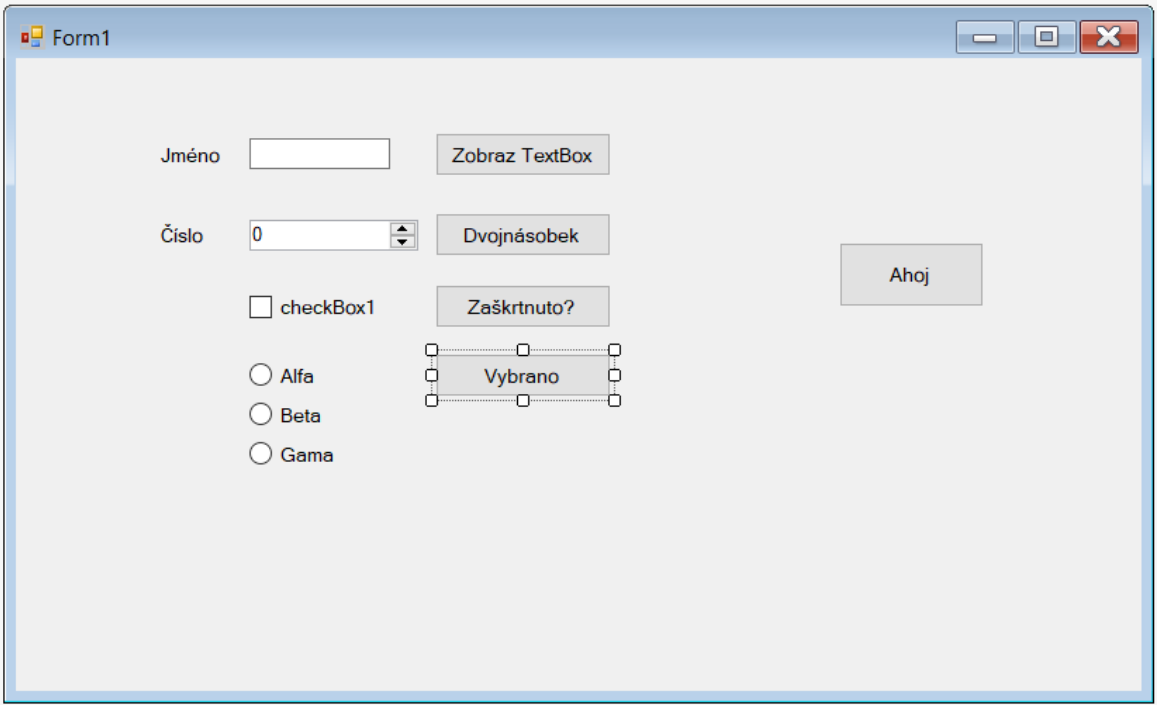
```
private void button4_Click(object sender, EventArgs e)
{
    if (checkBox1.Checked)
    {
        MessageBox.Show("Zaškrtnuto");
    }
    else
    {
        MessageBox.Show("Nezaškrtnuto");
    }
}
```

Obrázek 6.17: Přístup k zaškrtávacímu políčku

Informace o zaškrtnutí je uložena ve vlastnosti *Checked*, která je typu *bool*.

6.2.5 RadioButton

Tento ovládací prvek je jakýsi přepínač, který nám umožňuje vybrat jednu z předdefinovaných hodnot. Opět si jeho použití ukážeme na příkladu, kdy budeme mít nabídku tří možností Alfa, Beta a Gama a po kliknutí na tlačítko se nám zobrazí vybraná položka. Nejprve na formulář přetáhneme 3x *RadioButton* (pro každou možnost jeden *RadioButton*) a potom také tlačítko:



Obrázek 6.18: RadioButton

Dále dvojklikem na tlačítko vygenerujeme událost, která se vykoná po stlačení tlačítka. Oproti jiným ovládacím prvkům je zde každá položka samostatným ovládacím prvkem, takže abychom zjistili, která položka je vybrána, musíme otestovat všechny ovládací prvky a na základě toho vypsat popis vybrané položky:

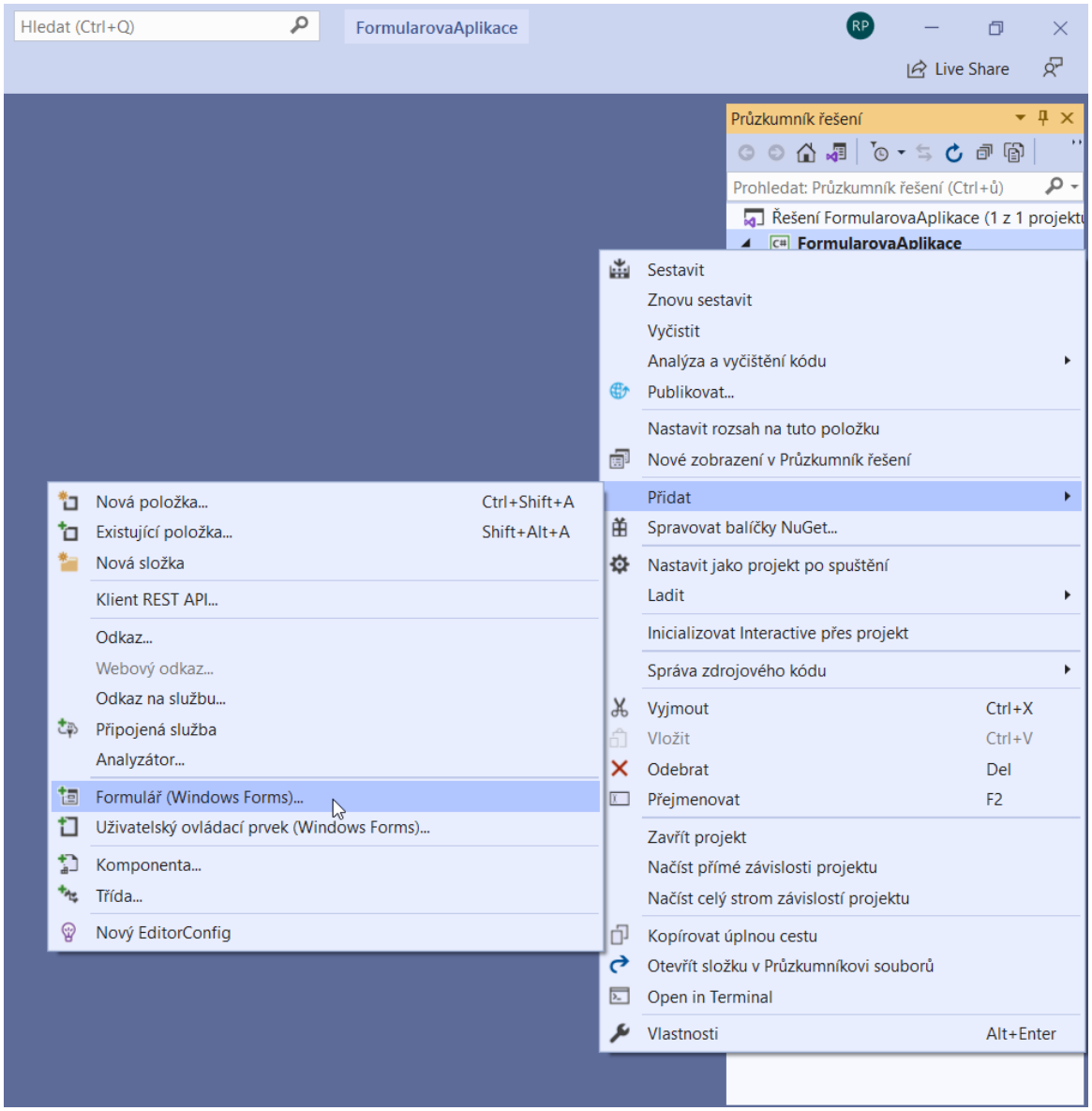
```
private void button5_Click(object sender, EventArgs e)
{
    string vybranaPolozka = "";
    if (radioButton1.Checked) vybranaPolozka = "Alfa";
    if (radioButton2.Checked) vybranaPolozka = "Beta";
    if (radioButton3.Checked) vybranaPolozka = "Gama";
    MessageBox.Show(vybranaPolozka);
}
```

Obrázek 6.19: Práce s přepínačem

Informace o vybrání příslušné položky je podobně jako u zaškrtávacího políčka uložena ve vlastnosti *Checked*, která je typu *bool*.

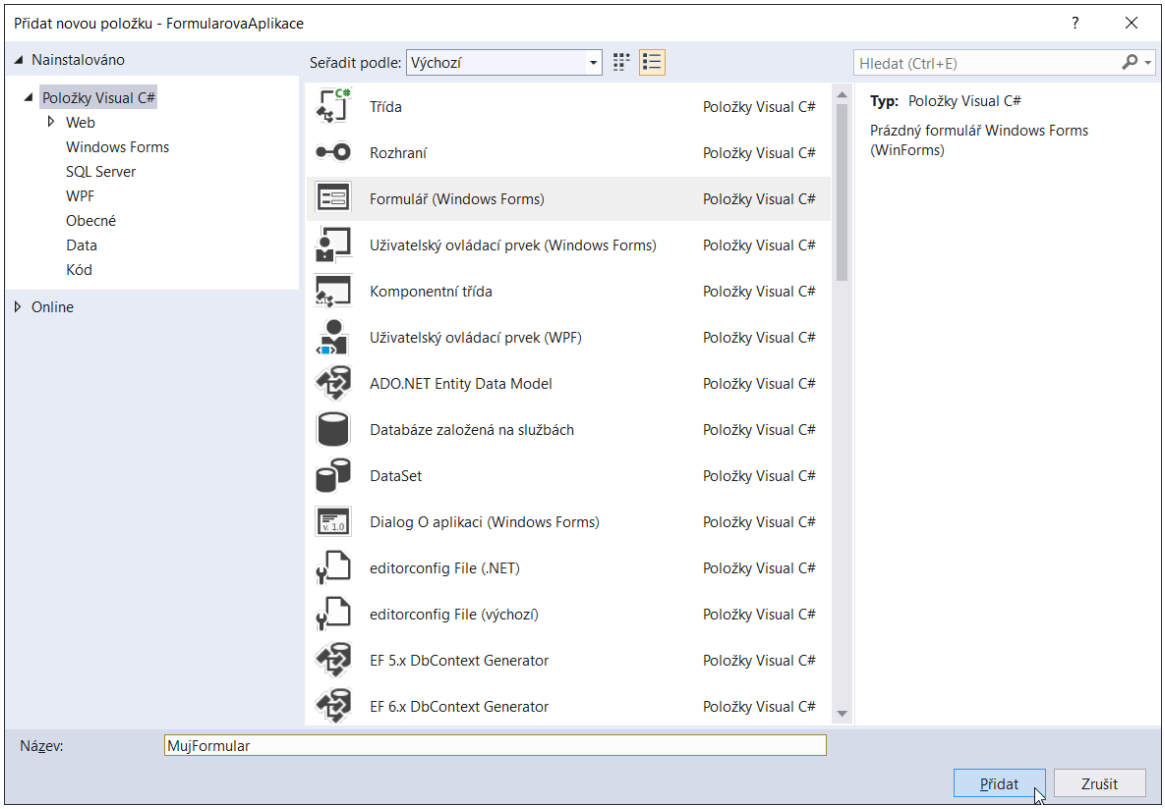
6.3 Práce s formuláři

Pokud se nebude jednat o velmi jednoduchou aplikaci, potom si obvykle s jedním formulářem nevystačíme. Pokud tedy potřebujeme mít v aplikaci další formulář, klikneme pravým tlačítkem myši v Průzkumníku řešení na složku projektu a zvolíme Přidat→Formulář (Windows Forms):



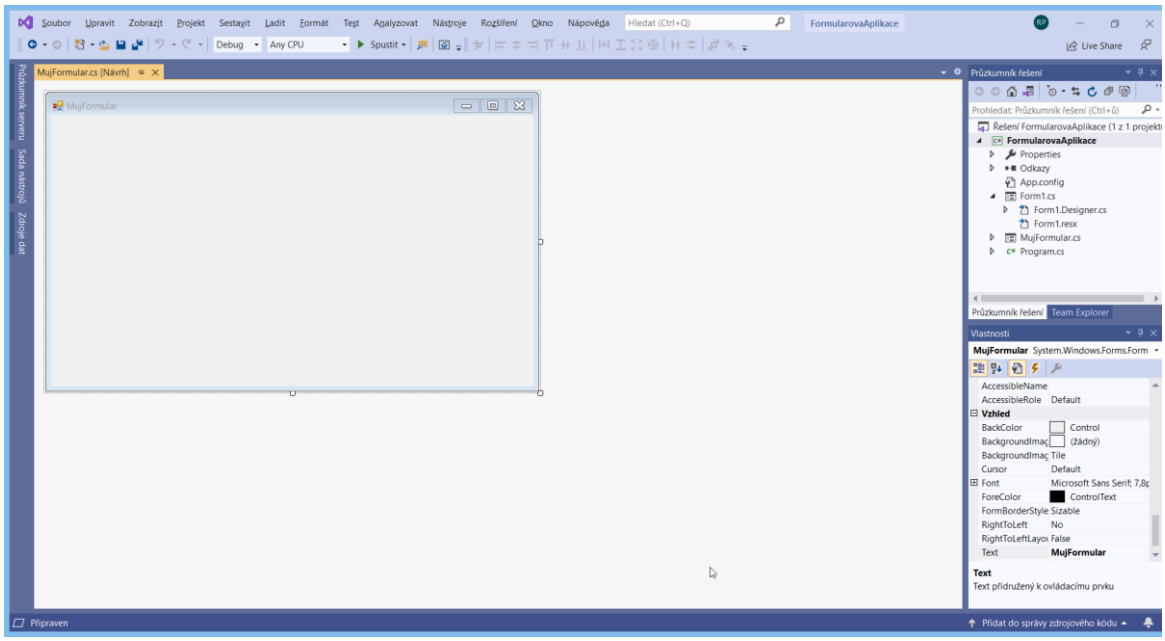
Obrázek 6.20: Přidání formuláře do projektu

Následně se nám zobrazí dialogové okno pro zadání názvu formuláře:



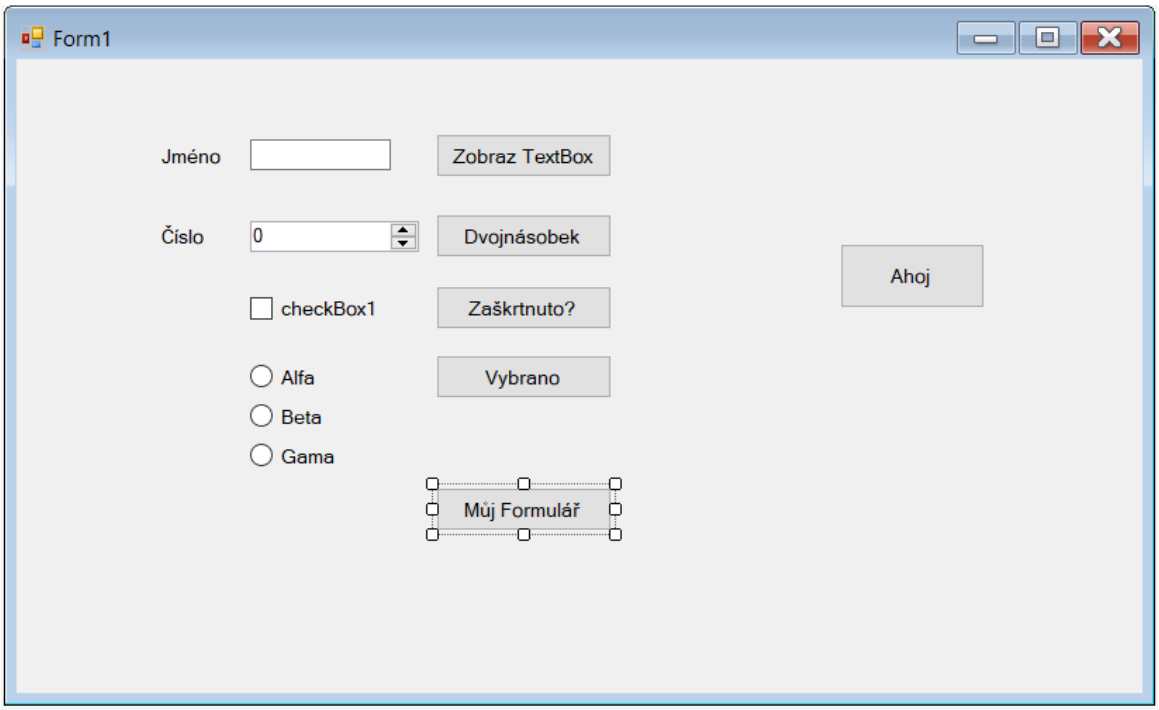
Obrázek 6.21: Název formuláře

Po potvrzení tlačítkem přidat se nám do projektu vloží nová položka – formulář MujFormular, na které můžeme vkládat ovládací prvky úplně stejným způsobem jako na výchozí formulář



Obrázek 6.22: Formulář MujFormular

Pokud aplikaci spustíme, uvidíme pouze pořád původní výchozí formulář. *MujFormular*, který jsme právě do projektu vložili, je sice součástí projektu, ale v projektu je pouze ve tvaru třídy. K tomu abychom s ním mohli pracovat, musíme ve výchozím formuláři vytvořit jeho instanci a poté zavolat metodu *ShowDialog*, která provede vykreslení na obrazovku. Nejprve tedy na výchozí formulář vložíme tlačítko s popiskem „Můj Formulář“:



Obrázek 6.23: Tlačítko Můj Formulář

Nakonec dvojklikem na tlačítko vygenerujeme událost, která se vykoná po stlačení tlačítka. Zde vytvoříme instanci třídy *MujFormular* a zavoláme metodu *ShowDialog*:

```
private void button6_Click(object sender, EventArgs e)
{
    MujFormular mf = new MujFormular();
    mf.ShowDialog();
}
```

Obrázek 6.24: Zobrazení nového formuláře

Nyní již po spuštění aplikace a kliknutí na tlačítko „Můj Formulář“ se nám zobrazí nově vložený formulář.

OTÁZKY

1. Pokud máme dvě proměnné typu *string* obsahující hodnoty „Alfa“ a „Beta“, potom po aplikaci operátoru plus na tyto dvě proměnné, bude výsledkem
    a. Syntaktická chyba
    b. Řetězec „Alfa Beta“
    c. Řetězec „AlfaBeta“
2. Který příkaz vypíše na obrazovku hodnotu celočíselné proměnné *cislo*?
    a. Console.WriteLine(cislo)
    b. Console.WriteLine(“cislo”)
    c. Console.WriteLine(“{1}”, cislo)
3. Který příkaz načte textový vstup od uživatele z klávesnice?
    a. Console.Get()
    b. Consloe.GetLine()
    c. Console.ReadLine()
4. Který příkaz použijeme, pokud aby se kurzor po vypsání textu přesunul na další řádek?
    a. Console.WriteLine()
    b. Consloe.Write()
    c. Console.WriteLineNoEnter()
5. Pokud se pokusíme zapsat text do souboru pomocí metody *File.WriteAllText* a tento soubor již existuje
    a. Existující soubor bude nejprve smazán
    b. Text se připojí na konec souboru
    c. Program skončí chybou
6. Pokud se pokusíme zapsat text do souboru pomocí metody *File.AppendAllText* a tento soubor již existuje
    a. Existující soubor bude nejprve smazán
    b. Text se připojí na konec souboru
    c. Program skončí chybou
7. Pokud se pokusíme přečíst text ze souboru pomocí metody *File.ReadAllText* a tento soubor neexistuje
    a. Program po spuštění oznámí, že soubor neexistuje a zeptá se na název nového souboru
    b. Metoda vrátí prázdný řetězec
    c. Program skončí chybou
8. Metoda *File.Exists* vrací návratovou hodnotu s datovým typem
    a. int
    b. bool
    c. string
9. Jak se nazývá parametr metody *main*, který slouží k předávání parametrů na příkazovém řádku?
    a. args

b. params
c. cmdln

10. První parametr příkazového řádku má index
    a. 0
    b. 1
    c. Parametry nejsou indexovány

SHRNUTÍ KAPITOLY

V této kapitole jsme si ukázali základní možnosti, jak může program komunikovat s vnějším světem. Nejprve jsme se zabývali zobrazením výstupu programu na obrazovku, přičemž jsme se zaměřili na možnosti formátování výstupu tak, aby odpovídal potřebám uživatele. Následně jsme si ukázali, jak můžeme načíst vstup od uživatele z klávesnice. Potom jsme přešli na ukázku práce s textovými soubory, a to jak pro načtení dat, tak i pro zápis dat do souboru. Kapitolu jsme ukončili demonstrací možností předávání vstupních dat pomocí parametrů příkazového řádku včetně ukázky nastavení těchto parametrů v MS Visual Studiu pro účely ladění.

ODPOVĚDI

1. c
2. a
3. c
4. b
5. a
6. b
7. c
8. b
9. a
10. a

7 DATABÁZE

RYCHLÝ NÁHLED KAPITOLY

Tato kapitola již nepřináší žádné nové konstrukce jazyka C#, ale je zaměřena spíše prakticky, abychom dokázali aplikovat nabyté znalosti pro vytvoření základních algoritmů pro řešení reálných úloh. Začneme od těch jednodušších, kdy budeme vyhledávat určité prvky v poli, potom se pokusíme setřídit pole prvků, zkusíme si vygenerovat prvočísla. Velká pozornost bude věnována rekurzivním algoritmům, které využívají vlastnosti, že metoda volá sebe sama. Rekurzi budeme demonstrovat na výpočtu součtu řady, faktoriálu, nalezení členů Fibonacciho posloupnosti, největšího společného dělitele dvou čísel a kapitolu zakončíme ukázkou řešení starého hlavolamu Hanojské věže.

CÍLE KAPITOLY

Po prostudování této kapitoly budete umět:

- Samostatně vytvořit jednoduchý algoritmus.
- Zjistit, zda určité číslo existuje v poli.
- Setřídit pole čísel.
- Najít všechna prvočísla do zadané hranice.
- Používat rekurzi.
- Vypočítat součet řady celých čísel pomocí rekurze.
- Vypočítat faktoriál.
- Nalézt členy Fibonacciho posloupnosti.
- Najít největšího společného dělitele dvou čísel.
- Vyřešit hlavolam Hanojské věže.

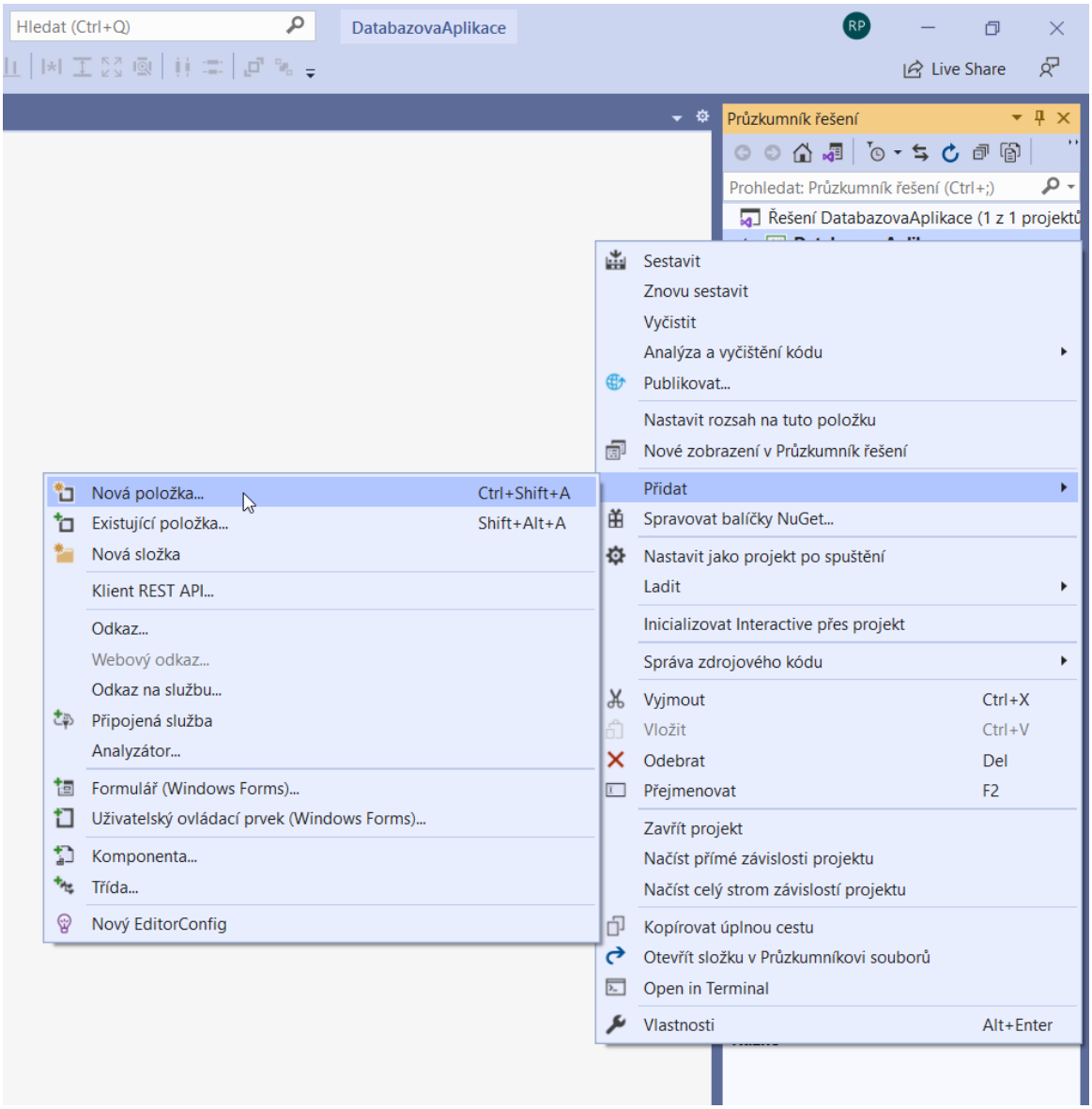
KLÍČOVÁ SLOVA KAPITOLY

Algoritmus, vyhledávání, třídění, prvočíslo, rekurze, řada, posloupnost, největší společný dělitel, faktoriál, hanojské věže.

7.1 Vytvoření databáze

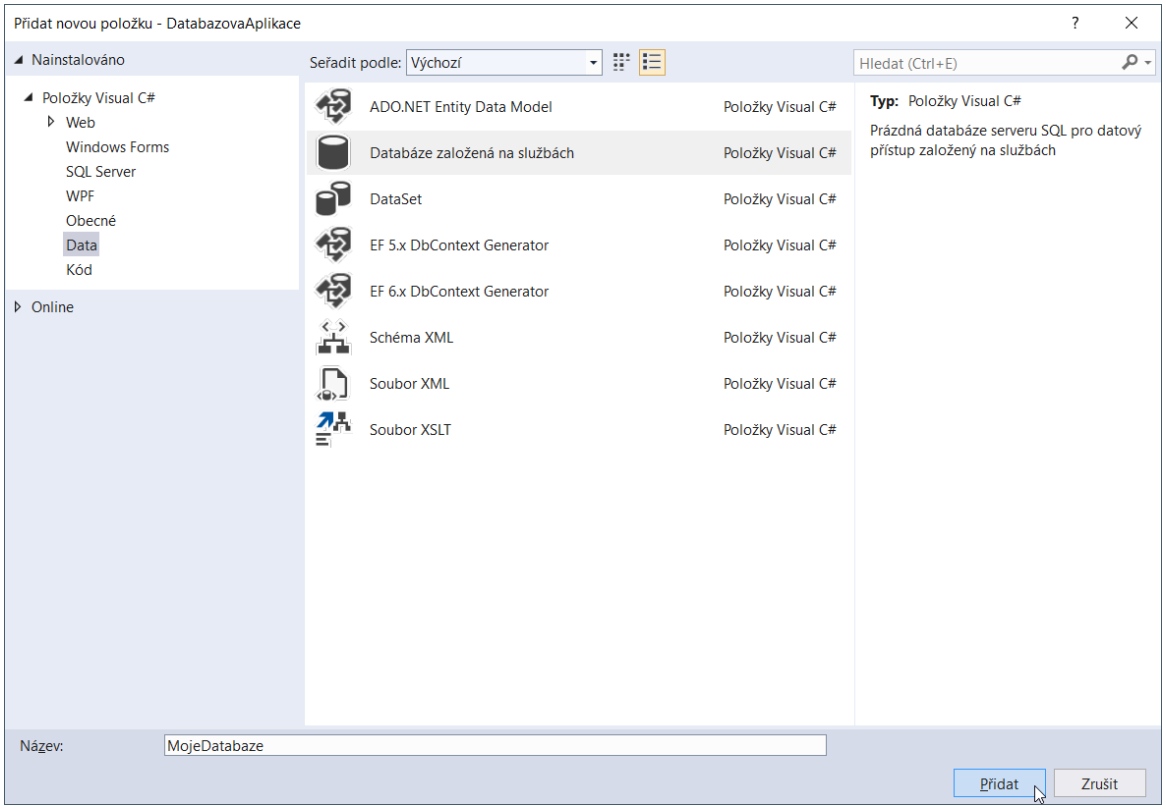
Prvním krokem je vytvoření samotné databáze. Máme několik možností, v praxi budeme obvykle používat nějakou databázi na samostatném SQL serveru. Při menším rozsahu dat nebo pro testování však MS Visual Studio nabízí jednoduchou souborovou databázi, která však používá stejné rozhraní jako MS SQL Server. Takže jakmile vytvoříme aplikaci s využitím této testovací souborové databáze, můžeme velmi jednoduše aplikaci přepnout, aby využívala MS SQL Server, případně jinou SQL databázi.

Začneme tedy vytvořením formulářové aplikace úplně stejným způsobem, jako jsme to provedli v předchozí kapitole. Potom v Průzkumníku řešení klikneme pravým tlačítkem na složku projektu a zvolíme *Přidat→Nová položka*:



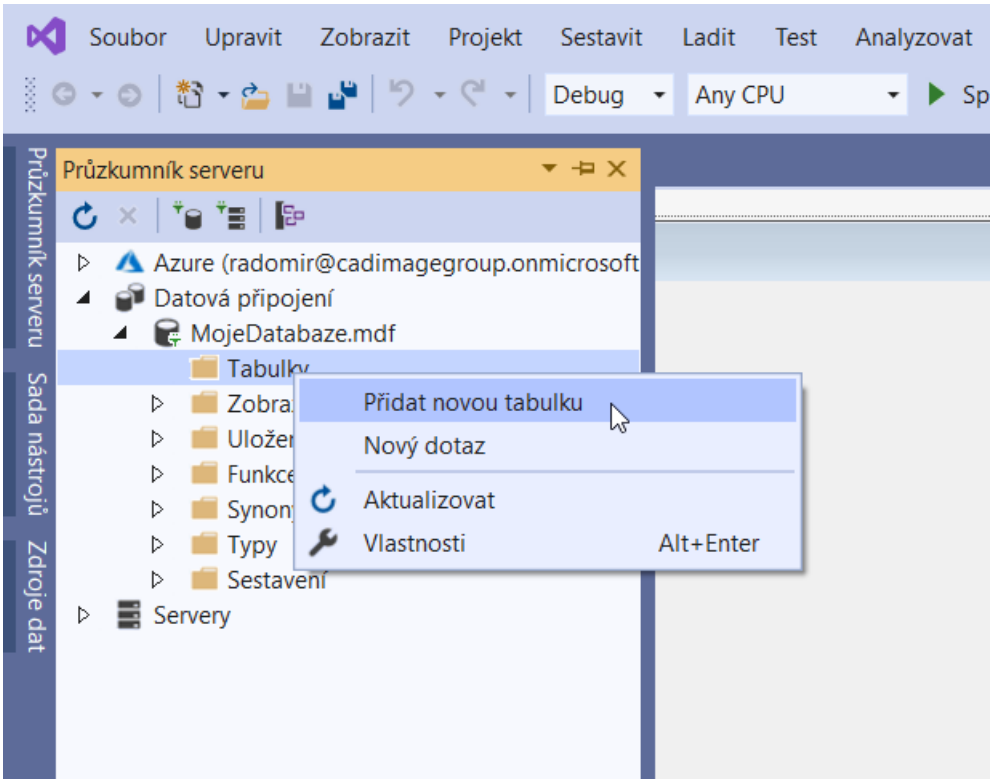
Obrázek 7.1: Přidání nové položky do projektu

Poté se nám zobrazí formulář pro výběr typu a názvu položky. V levé části zvolíme *Data* a potom uprostřed vybereme „Databáze založená na službách“. Dole potom ještě zadáme název databáze, např. *MojeDatabaze*:



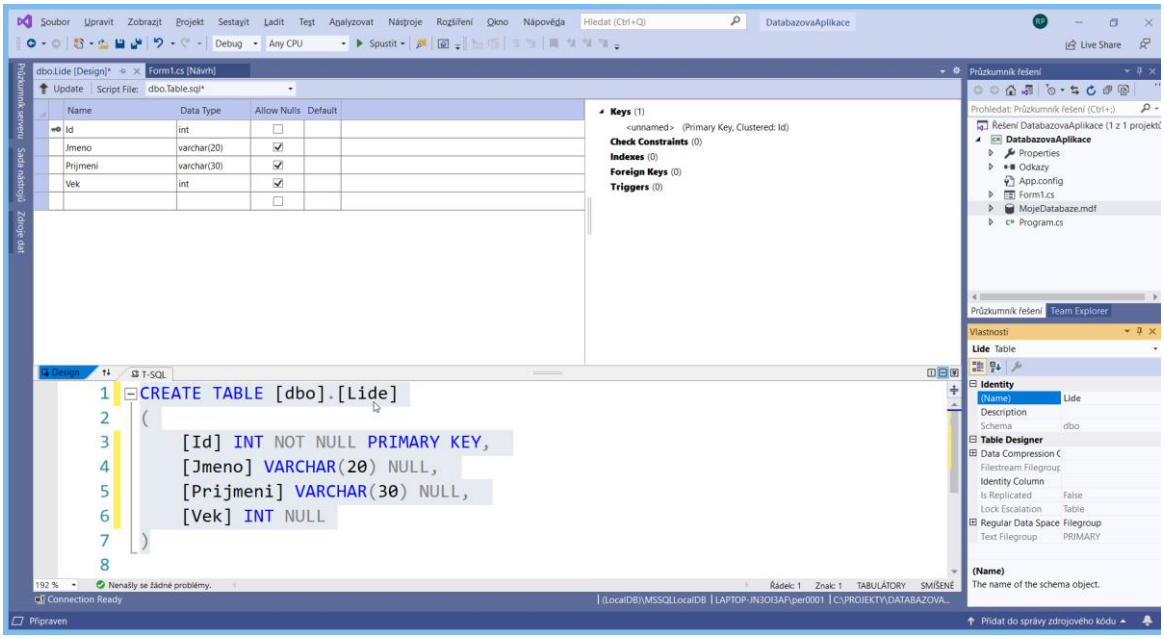
Obrázek 7.2: Vložení databáze

Po potvrzení se přidá databáze do našeho projektu, což si můžeme zkontrolovat v Průzkumníku řešení, kde by se měla objevit nová položka *MojeDatabaze.mdf*. Vytvořená databáze je samozřejmě prázdná, takže abychom s ní mohli pracovat, musíme si v ní vytvořit nějaké tabulky. Na ukázku si vytvoříme jednoduchou tabulku *Lide*, která bude obsahovat tři sloupce *Jmeno*, *Prijmeni* a *Vek*. To uděláme tak, že v Průzkumníku řešení dvakrát klikneme na položku *MojeDatabaze.mdf*. Alternativní možností je kliknutí na kartu Průzkumník serveru vlevo nahoře. Rozbalíme složku *Datová připojení* a klineme pravým tlačítkem na položku *Tabulky* pod *MojeDatabaze.mdf*, kde vybereme položku *Přidat novou tabulku*:



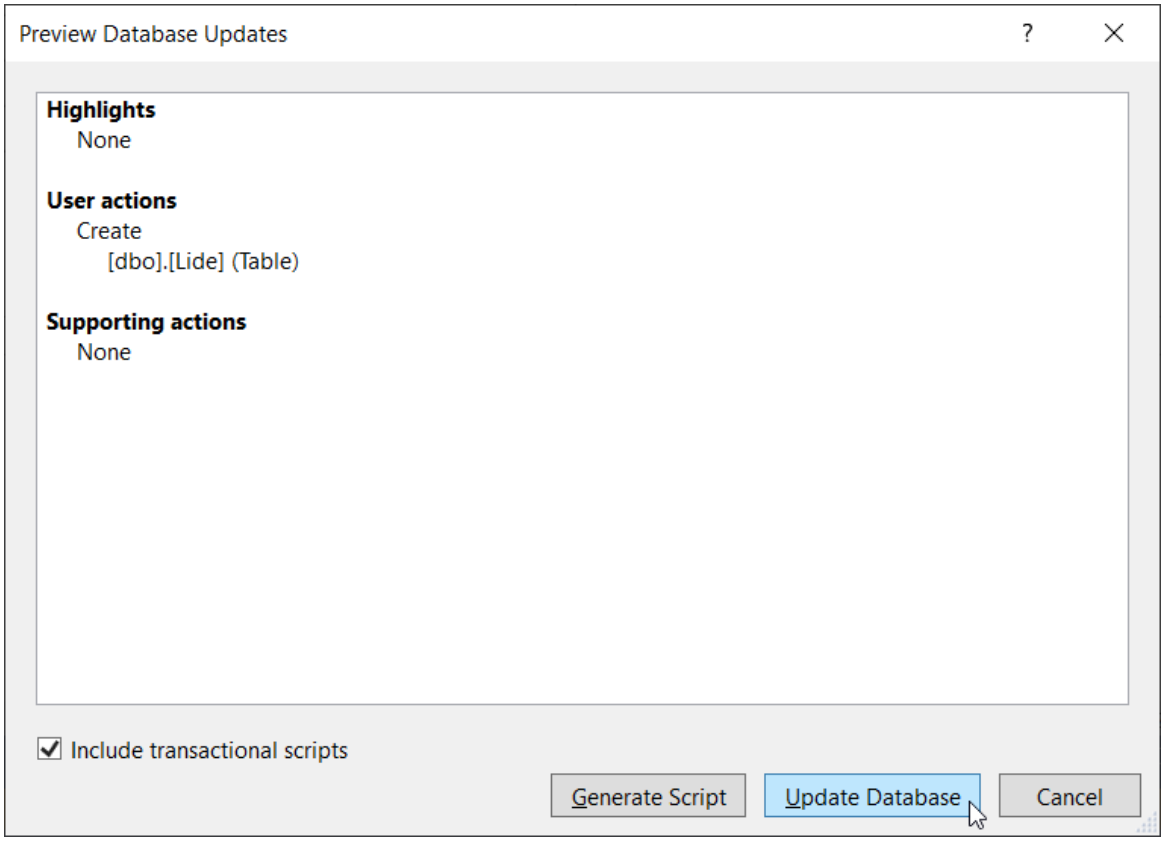
Obrázek 7.3: Přidání nové tabulky

Následně se nám zobrazí formulář, kde můžeme definovat strukturu tabulky, tj. definovat názvy a typy jednotlivých sloupců. Všimněte si, že první sloupec tabulky je již předvyplněn položkou Id, která slouží jako jednoznačný identifikátor řádku, aby bylo možno jednotlivé řádky rozlišit i v případě, že dvě osoby budou mít stejné jméno i příjmení. Dále zadáme další tři sloupce, tj. *Jmeno*, *Prijmeni* a *Vek*. U každé položky musíme specifikovat datový typy. Databázové datové typy jsou podobné datovým typům v C#, ale existuje jich více, aby bylo možno do databáze efektivně uložit široké spektrum dat. Pro Vek použijeme identický databázový datový typ jako v C#, tj. *int*. Pro řetězce však použijeme databázový datový typ *varchar(20)*, který označuje řetězce s variabilní délkou, maximální počet znaků je dán číslem v závorce. Výsledná struktura tabulky bude vypadat takto:



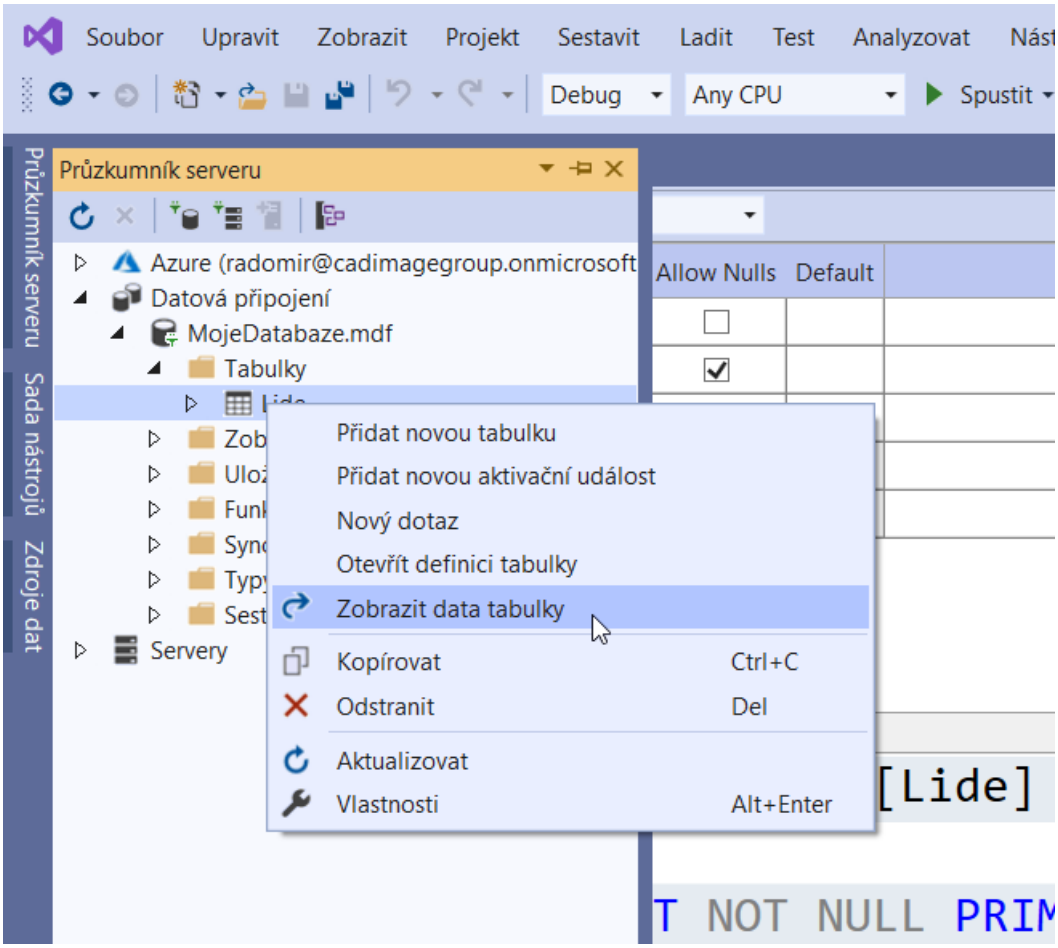
Obrázek 7.4: Návrh tabulky

Podívejme se nyní na spodní část obrazovky, která obsahuje SQL (Structured Query Language) kód pro vytvoření naší tabulky. Doposud jsme totiž tabulku v databázi nevytvořili, pouze jsme navrhli tabulku a MS Visual Studio tento návrh automaticky převedlo na SQL kód. Tento kód lze editovat a provést i speciální nastavení, které nelze provést v návrháři v horní části obrazovky. Jednu věc budeme v SQL kódu měnit vždy, a to název tabulky, který poněkud překvapivě nelze změnit v návrháři. Název tabulky je v SQL kódu je uveden na prvním řádku za *CREATE TABLE*. Po provedení všech změn klikneme na tlačítko Update, poté se zobrazí dialogové okno s dotazem, zda chceme pouze vygenerovat SQL skript nebo přímo upravit databázi, zvolíme tedy *Update Database*:



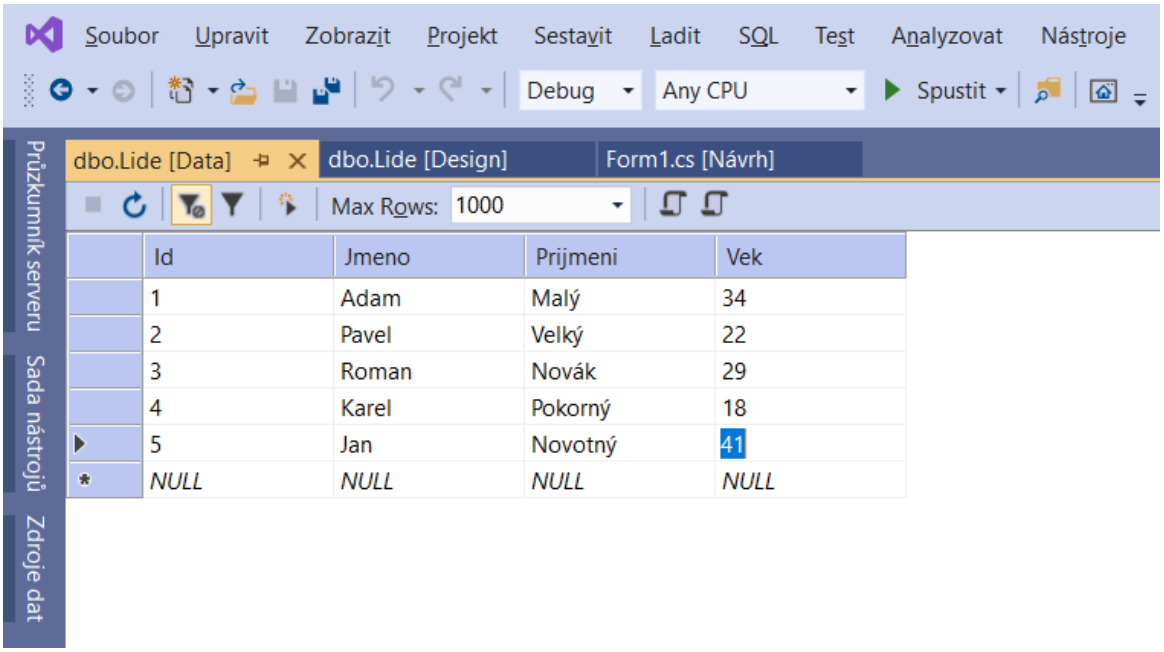
Obrázek 7.5: Potvrzení vytvoření tabulky

Tímto dojde k vytvoření tabulky v databázi, což si můžeme ověřit v Průzkumníku serveru. Dalším krokem bude vložení záznamů do tabulky, což provedeme v Průzkumníku serveru kliknutím pravým tlačítkem myši na *Zobrazit data tabulky*:



Obrázek 7.6: Zobrazit data tabulky

Nyní již vidíme klasické tabulkové zobrazení, do kterého můžeme vkládat jednotlivé záznamy. Zkusíme si tedy přidat do tabulky několik testovacích záznamů:



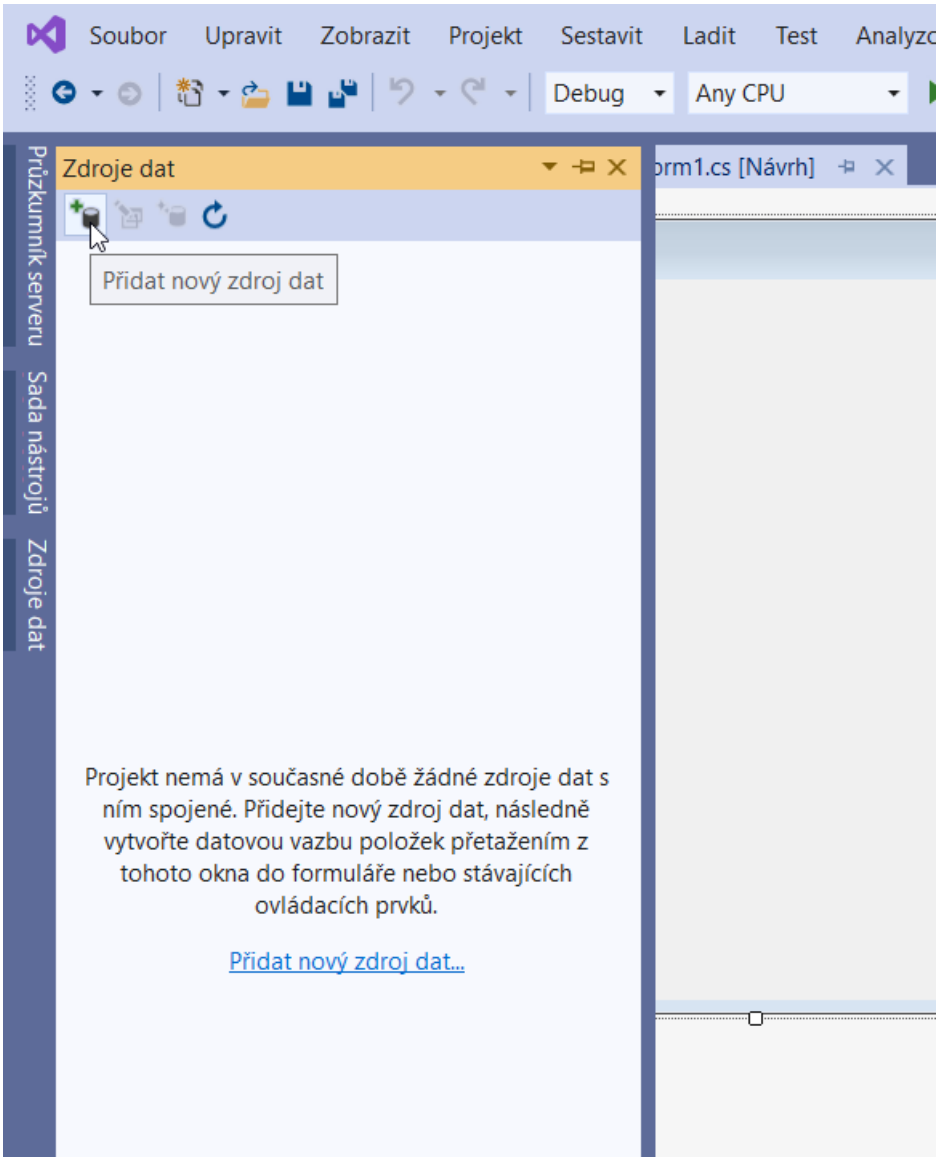
Id	Jmeno	Prijmeni	Vek
1	Adam	Malý	34
2	Pavel	Velký	22
3	Roman	Novák	29
4	Karel	Pokorný	18
5	Jan	Novotný	41
NULL	*NULL*	*NULL*	*NULL*

Obrázek 7.7: Vložení záznamů do tabulky

Na rozdíl od samotné struktury tabulky není nutné záznamy ukládat, protože při zadání celého řádku a přesunu na nový se daný řádek automaticky uloží do databáze.

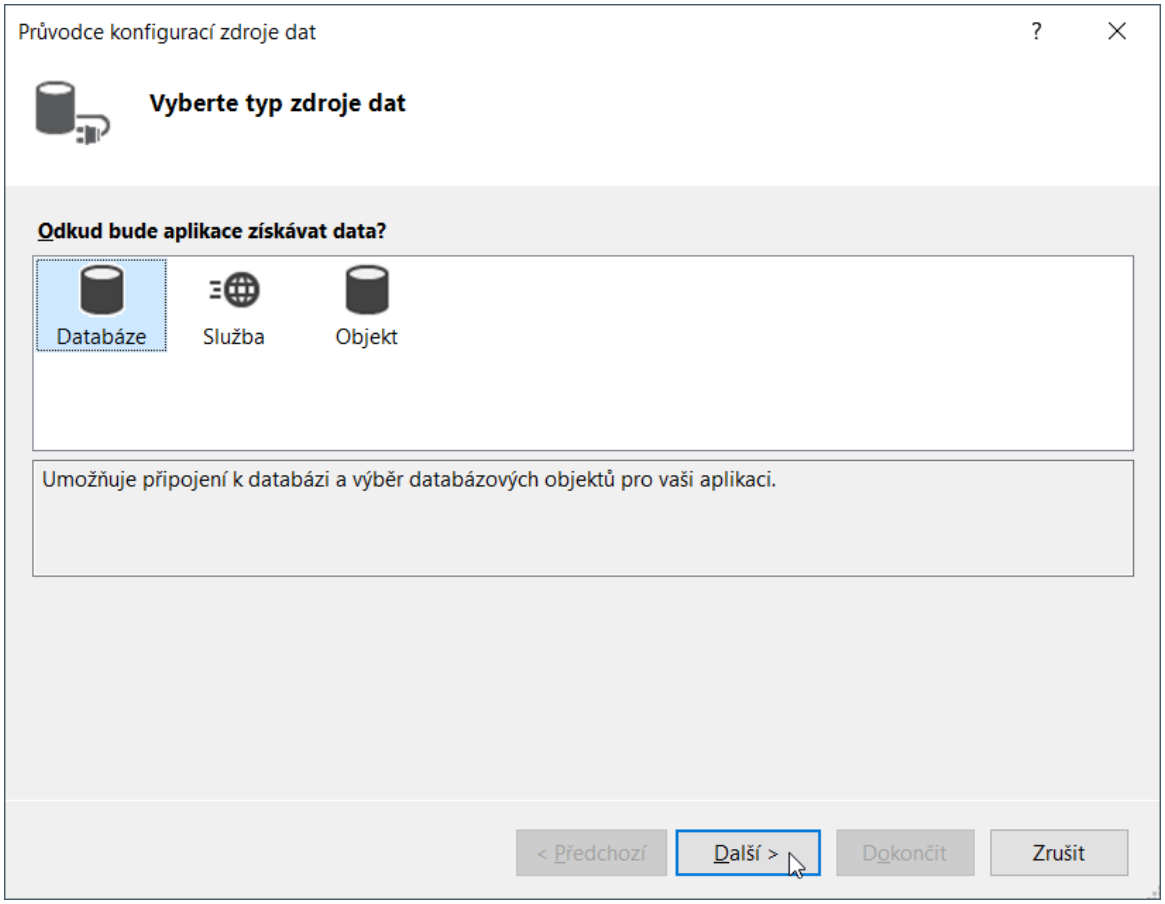
7.2 Práce s databází

Nyní bychom si zkusili ukázat, jak můžeme zobrazit a editovat data v naší testovací tabulce *Lide*. Vlevo na kartě *Zdroje dat* klikneme na ikonu se symbolem plus:



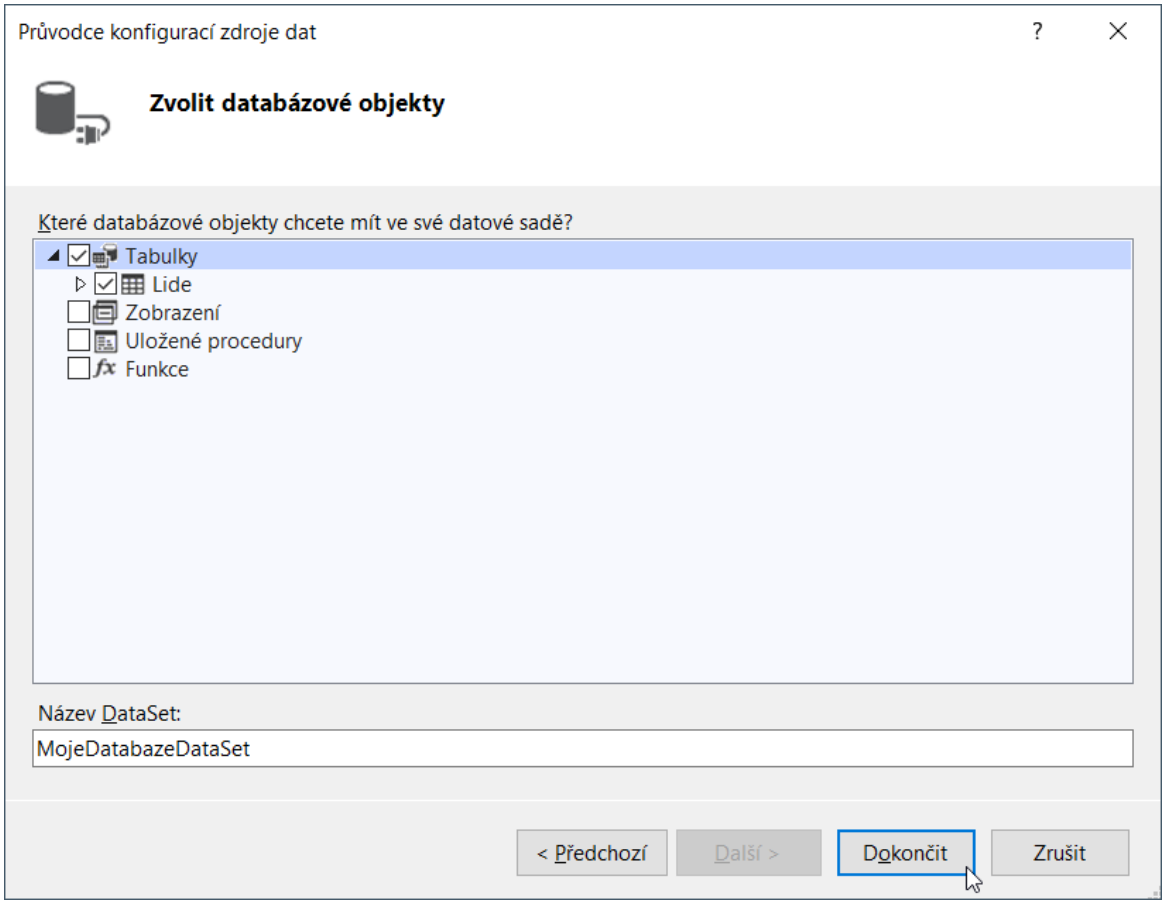
Obrázek 7.8: Zdroje dat

V dalším kroku se nám zobrazí typ zdroje dat, kde vybereme *Databáze*:



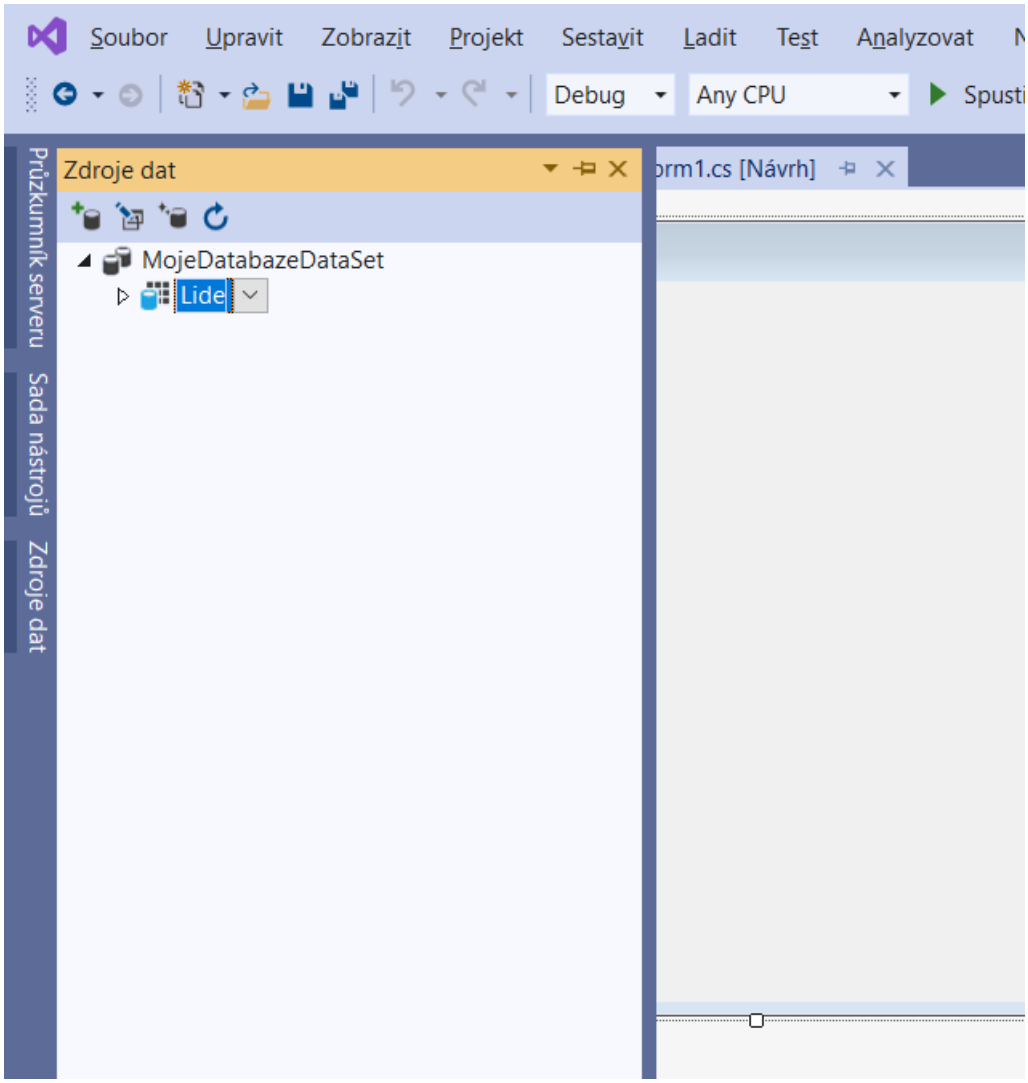
Obrázek 7.9: Typ zdroje dat

Dále se zobrazí několik potvrzujících obrazovek, které jen potvrdíme, až se dostaneme na krok, kde vybíráme příslušnou tabulku. Tam vybereme tabulku *Lide* a potvrdíme tlačítkem *Dokončit*:



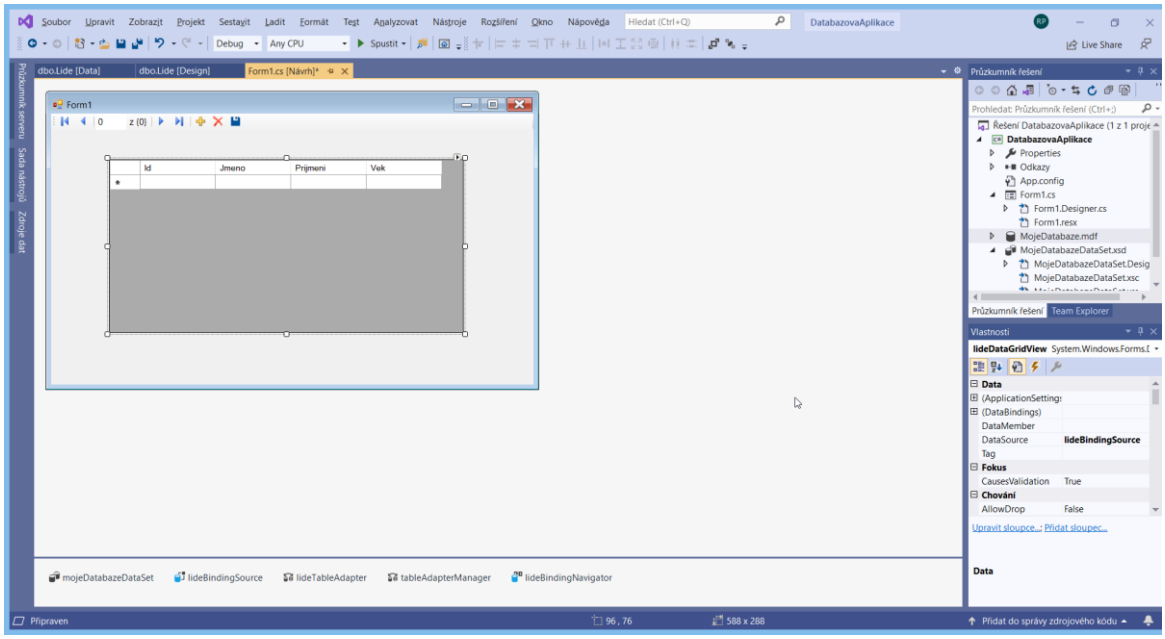
Obrázek 7.10: Výběr tabulky

Nyní se nám v okně Zdroje dat objeví nová položka *ModeDatabazeDataSet* s tabulkou *Lide*:



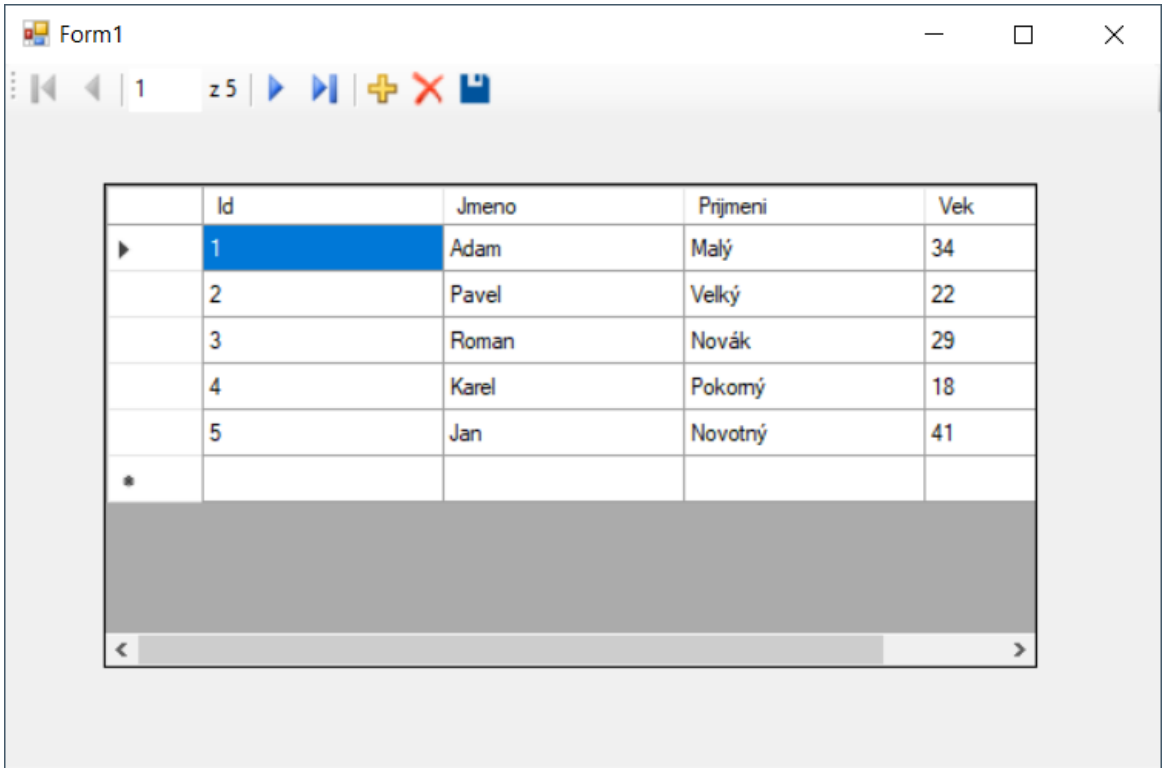
Obrázek 7.11: Zdroj dat

Tuto tabulku *Lide* poté přetáhneme na formulář, čímž se nám na formulář vloží veškeré ovládací prvky, které jsou potřeba pro zobrazení dat v tabulce:



Obrázek 7.12: Vložení tabulky na formulář

Po spuštění aplikace již vidíme všechny záznamy v tabulce:



Obrázek 7.13: Zobrazení tabulky v aplikaci

OTÁZKY

1. Eratosthenovo síto je algoritmus pro nalezení
    a. Lichých čísel
    b. Sudých čísel
    c. Prvočísel
2. Euklidův algoritmus je algoritmus pro nalezení
    a. Největšího společného dělitele dvou čísel
    b. Prvočísel
    c. Nejmenšího společného násobku dvou čísel
3. Fibonacciho posloupnost je definována jako
    a. Každý člen je roven součtu dvou předchozích členů posloupnosti
    b. Každý člen je roven dvojnásobku předchozího členu posloupnosti
    c. Každý člen je roven přirozenému logaritmu předchozího členu posloupnosti
4. Která realizace algoritmu pro nalezení členů Fibonacciho posloupnosti je efektivnější?
    a. Jsou přibližně stejně efektivní
    b. Pomocí cyklu
    c. Pomocí rekurze
5. K čemu slouží příkaz *continue* v cyklu?
    a. Předčasně ukončí celý cyklus
    b. Předčasně ukončí aktuální iteraci cyklu, ale samotný cyklus dále pokračuje
    c. Přeskočí se kód, který je před příkazem *continue*
6. Co je to rekurze?
    a. Neexistuje
    b. Jiný výraz pro cyklus
    c. Druh algoritmu, kdy v průběhu výpočtu metoda volá sebe sama
7. Lze rekurzi nahradit pomocí cyklu?
    a. Ano
    b. Ne
    c. Ano, ale jen pokud je počet volání rekurentního algoritmu menší než deset
8. Celkový počet iterací vnořených cyklů je roven
    a. Součtu iterací jednotlivých cyklů
    b. Součinu iterací jednotlivých cyklů
    c. Součtu přirozeného logaritmu iterací jednotlivých cyklů
9. Jaký návratový typ může mít metoda počítající faktoriál 20 tak, aby výsledek byl správný?
    a. short
    b. int
    c. long

10. Jak dlouho bude přibližně trvat vyřešení původní verze hlavolamu Hanojské věže pro 64 disků na běžném osobním počítači?
    a. Jeden den
    b. Jeden rok
    c. Běžný počítač tento hlavolam nevyřeší ani za celý lidský život

SHRNUTÍ KAPITOLY

V této kapitole jsme se zaměřili na praktickou aplikaci nabytých znalostí z předchozí kapitol. Začali jsme ukázkou jednoduchého algoritmu pro nalezení indexu prvku v poli, potom jsme si ukázali použití vnořených cyklů pro setřídění pole čísel. Rovněž jsme si ukázali, jak vypsat všechna prvočísla do zadané horní meze. Následně jsme se seznámili s takzvanou rekurzí, kdy metoda při výpočtu volá opakovaně sebe sama. Použití rekurze jsme si demonstrovali na výpočtu součtu řady celých čísel, výpočtu faktoriálu, vypsání členů Fibonacciho posloupnosti, nalezení největšího společného dělitele dvou čísel a kapitolu jsme uzavřeli rekurzivním algoritmem pro řešení známého hlavolamu Hanojské věže.

ODPOVĚDI

1. c
2. a
3. a
4. b
5. b
6. c
7. a
8. b
9. c
10. c

8 SPECIFIKA PROGRAMOVACÍHO JAZYKA JAVA

RYCHLÝ NÁHLED KAPITOLY

Tato kapitola již nepřináší žádné nové konstrukce jazyka C#, ale je zaměřena spíše prakticky, abychom dokázali aplikovat nabyté znalosti pro vytvoření základních algoritmů pro řešení reálných úloh. Začneme od těch jednodušších, kdy budeme vyhledávat určité prvky v poli, potom se pokusíme setřídit pole prvků, zkusíme si vygenerovat prvočísla. Velká pozornost bude věnována rekurzivním algoritmům, které využívají vlastnosti, že metoda volá sebe sama. Rekurzi budeme demonstrovat na výpočtu součtu řady, faktoriálu, nalezení členů Fibonacciho posloupnosti, největšího společného dělitele dvou čísel a kapitolu zakončíme ukázkou řešení starého hlavolamu Hanojské věže.

CÍLE KAPITOLY

Po prostudování této kapitoly budete umět:

- Samostatně vytvořit jednoduchý algoritmus.
- Zjistit, zda určité číslo existuje v poli.
- Setřídit pole čísel.
- Najít všechna prvočísla do zadané hranice.
- Používat rekurzi.
- Vypočítat součet řady celých čísel pomocí rekurze.
- Vypočítat faktoriál.
- Nalézt členy Fibonacciho posloupnosti.
- Najít největšího společného dělitele dvou čísel.
- Vyřešit hlavolam Hanojské věže.

KLÍČOVÁ SLOVA KAPITOLY

Algoritmus, vyhledávání, třídění, prvočíslo, rekurze, řada, posloupnost, největší společný dělitel, faktoriál, hanojské věže.

8.1 Jazyk Java

Java je objektově orientovaný programovací jazyk, který vyvinula firma Sun Microsystems a představila 23. května 1995. Jde o jeden z nejpoužívanějších programovacích jazyků na světě. Podle TIOBE indexu byla Java nejpopulárnější programovací jazyk. V roce 2020 jazyk Java v žebříčku TIOBE index předběhly jazyky C a Python.[3] Díky své přenositelnosti je používán pro programy, které mají pracovat na různých systémech počínaje čipovými kartami (platforma JavaCard), přes mobilní telefony a různá zabudovaná zařízení (platforma Java ME), aplikace pro desktopové počítače (platforma Java SE) až po rozsáhlé distribuované systémy pracující na řadě spolupracujících počítačů rozprostřené po celém světě (platforma Java EE). Tyto technologie se jako celek nazývají platforma Java. Dne 8. května 2007 Sun uvolnil zdrojové kódy Javy (cca 2,5 miliónů řádků kódu) a Java bude dále vyvíjena jako open source.

Java používá automatickou správu paměti pro řízení paměti v object lifecycle. Programátor určuje, kdy bude objekt vytvořen a Java runtime zodpovídá za obnovu paměti jakmile se objekty přestanou používat. Když nezůstanou žádné odkazy na objekt, nepřístupná paměť se stává přístupnou pro garbage collector, který ji automaticky uvolní. Něco podobného úniku paměti přesto může vzniknout, jestliže programátorův kód obsahuje referenci na objekt, který již není potřeba.

Jedním z účelů pro automatickou správu paměti Java bylo zbavit programátora nutnosti lámat si hlavu nad manuální správou paměti. V některých programovacích jazycích je paměť nutná pro vytvoření objektů implicitně přidělena do zásobníku, anebo explicitně přidělena a uvolněna. V takovém případě odpovědnost za správu paměti padá na programátora. Pokud program objekt neuvolní, může dojít k úniku paměti. Pokud se program pokusí uvolnit paměť, která jíž byla uvolněna, výsledek není definován a je obtížné ho odhadnout, takže program se pravděpodobně stane nestabilní a/nebo spadne. To může být částečně odstraněno použitím chytrých ukazatelů, ale to také zvětšuje náročnost a složitost.

Syntaxe Javy je do značné míry odvozena z C++. Na rozdíl od C++, které kombinuje syntaxi pro strukturované, generické, a objektově orientované programování, Java byla postavena téměř výhradně jako objektově orientovaný jazyk. Veškerý kód je napsán uvnitř třídy a všechno je objekt, s výjimkou primitivních datových typů (tj. celá čísla, desetinná čísla, logické hodnoty a znaky), které nejsou třídy z výkonnostních důvodů. Na rozdíl od C# Java nepodporuje přetěžování operátorů nebo vícenásobnou dědičnost pro třídy. To zjednodušuje jazyk a pomáhá při prevenci potenciálních chyb.

8.2 Ukázka programu v jazyce Java

Podobně jako jsme u jazyka C# začínali s ukázkou jednoduchého programu, zkusíme si vytvořit *Hello World* aplikaci v jazyce Java:

```
public class Program {
```

```
        public static void main(String[] args)
        {
                System.out.println("Hello World!");
        }
}
```

Můžeme si všimnout, že struktura programu je identická jako v jazyce C#. Jediný rozdíl je v příkazu k výpisu na obrazovku.

8.3 Odlišnosti od jazyka C#

Nyní se zkusme podívat na nejdůležitější rozdíly mezi jazyky C# a Java.

8.3.1 DĚDIČNOST

Mechanismus dědičnosti je stejný jako v jazyce C#, rozdíl je pouze v samotné syntaxi, kyd místo symbolu dvojtečky používáme klíčové slovo *extends*. Ukažme si to na příkladu, kdy budeme chtít deklarovat třídu *Pes*, která dědí z třídy *Zvire*:

```
public class Pes extends Zvire
{
        //deklarace třídy Pes
}
```

8.3.2 KONSTRUKTOR

V deklaraci ani mechanismu fungování samotného konstruktoru není žádný rozdíl. Odlišnost však zpozorujeme, pokud budeme chtít z konstruktoru potomka zavolat konstruktor rodiče. V jazyce C# jsme k tomu využívali klíčové slovo *base*, kdežto v jazyce Java používáme klíčové slovo *super*. Rozdíl je také v tom, že v jazyce Java není volání nadřazeného konstruktoru definováno v samotné deklaraci konstruktoru, ale v jeho těle. Zkusme si toto demonstrovat opět na příkladu tříd *Zvire* a *Pes*, kde třída zvíře má deklarovaný konstruktor s parametrem *jmeno* typu *String* a ve třídě *Pes* budeme deklarovat konstruktor, který bude mít parametry *jmeno* a *hmotnost*, přičemž budeme chtít volat konstruktor třídy *Zvire* s parametrem *jmeno*:

```
public class Pes extends Zvire
{
        public Pes(String jmeno, int hmotnost)
        {
                Super(jmeno);
                //další kód konstruktoru třídy Pes
        }
```

}

8.3.3 VIRTUÁLNÍ METODY

Pokud jsme v jazyce C# chtěli používat virtuální metody, museli jsme nejprve takovou metodu v rodičovské třídě označit klíčovým slovem *virtual* a v potomkovi potom klíčovým slovem *override*. V jazyce Java jsou však všechny metody automaticky virtuální, takže pro jejich využití stačí v obou třídách deklarovat danou metodu se stejným názvem a samozřejmě i počtem a typem parametrů. Na jednu stranu může být toto chování považováno za výhodu, na druhou stranu je třeba dávat pozor, abychom omylem nevytvořili metodu se stejným názvem jako v rodičovské třídě a tím nezměnili její chování.

8.3.4 PŘETĚŽOVÁNÍ OPERÁTORŮ

Na rozdíl od jazyka C jazyk Java nepodporuje přetěžování operátorů. Pokud potřebujeme definovat nějaké speciální operace pro námi vytvořené třídy, musíme toto realizovat pomocí metod. Nevýhodou je samozřejmě těžkopádnější zápis a také nemožnost využití matematických pravidel pro práci s operátory, tedy přednosti jednotlivých operátorů a použití závorek.

OTÁZKY

1. Eratosthenovo síto je algoritmus pro nalezení
   a. Lichých čísel
   b. Sudých čísel
   c. Prvočísel
2. Euklidův algoritmus je algoritmus pro nalezení
   a. Největšího společného dělitele dvou čísel
   b. Prvočísel
   c. Nejmenšího společného násobku dvou čísel
3. Fibonacciho posloupnost je definována jako
   a. Každý člen je roven součtu dvou předchozích členů posloupnosti
   b. Každý člen je roven dvojnásobku předchozího členu posloupnosti
   c. Každý člen je roven přirozenému logaritmu předchozího členu posloupnosti
4. Která realizace algoritmu pro nalezení členů Fibonacciho posloupnosti je efektivnější?
   a. Jsou přibližně stejně efektivní
   b. Pomocí cyklu

   c. Pomocí rekurze
5. K čemu slouží příkaz *continue* v cyklu?
   a. Předčasně ukončí celý cyklus
   b. Předčasně ukončí aktuální iteraci cyklu, ale samotný cyklus dále pokračuje
   c. Přeskočí se kód, který je před příkazem *continue*
6. Co je to rekurze?
   a. Neexistuje
   b. Jiný výraz pro cyklus
   c. Druh algoritmu, kdy v průběhu výpočtu metoda volá sebe sama
7. Lze rekurzi nahradit pomocí cyklu?
   a. Ano
   b. Ne
   c. Ano, ale jen pokud je počet volání rekurentního algoritmu menší než deset
8. Celkový počet iterací vnořených cyklů je roven
   a. Součtu iterací jednotlivých cyklů
   b. Součinu iterací jednotlivých cyklů
   c. Součtu přirozeného logaritmu iterací jednotlivých cyklů
9. Jaký návratový typ může mít metoda počítající faktoriál 20 tak, aby výsledek byl správný?
   a. short
   b. int
   c. long
10. Jak dlouho bude přibližně trvat vyřešení původní verze hlavolamu Hanojské věže pro 64 disků na běžném osobním počítači?
   a. Jeden den
   b. Jeden rok
   c. Běžný počítač tento hlavolam nevyřeší ani za celý lidský život

SHRNUTÍ KAPITOLY

V této kapitole jsme se zaměřili na praktickou aplikaci nabytých znalostí z předchozí kapitol. Začali jsme ukázkou jednoduchého algoritmu pro nalezení indexu prvku v poli, potom jsme si ukázali použití vnořených cyklů pro setřídění pole čísel. Rovněž jsme si ukázali, jak vypsat všechna prvočísla do zadané horní meze. Následně jsme se seznámili s takzvanou rekurzí, kdy metoda při výpočtu volá opakovaně sebe sama. Použití rekurze jsme si demonstrovali na výpočtu součtu řady celých čísel, výpočtu faktoriálu, vypsání členů Fibonacciho posloupnosti, nalezení největšího společného dělitele dvou čísel a kapitolu jsme uzavřeli rekurzivním algoritmem pro řešení známého hlavolamu Hanojské věže.

ODPOVĚDI

1. c
2. a
3. a
4. b
5. b
6. c
7. a
8. b
9. c
10. c

LITERATURA

BORY, P., 2016. *C# – Bez předchozích znalostí*. Brno: Computer Press. ISBN 978-80-251-4686-6.

ČADA, O., 2009. *Objektové programování*. Praha: Grada. ISBN 978-80-247-2745-5.

FORD, S., 2009. *266 tipů a triků pro Microsoft Visual Studio*. Brno: Computer Press. ISBN 978-80-251-2554-0.

JOYCE, F., 2017. *Microsoft Visual C#: An Introduction to Object-Oriented Programming*. Boston: Cengage Learning. ISBN 9781337102100.

NAGEL, C., 2007. *C# 2005 Programujeme profesionálně*. Brno: Computer Press. ISBN 80-251-1181-4.

ROUBALOVÁ, E., 2015. *Java – Bez předchozích znalostí*. Brno: Computer Press. ISBN 978-80-251-4572-2.

SCHILDT, H., 2014. *Mistrovství – Java*. Brno: Computer Press. ISBN 978-80-251-4145-8.

VIRIUS, M., 2002. *C# pro zelenáče*. Praha: Neokortex. ISBN 8086330117.

SHRNUTÍ STUDIJNÍ OPORY

Tato studijní opora se vás pokusila seznámit se základy algoritmizace a základy programovacího jazyka C# ve vývojovém prostředí Microsoft Visual Studio. Po jejím přečtení byste měli být schopni samostatně implementovat jednoduchý i středně složitý algoritmus v jazyce C# s využitím pravidel strukturovaného programování.

Pokud vás programování zaujalo, mohu vám doporučit studium dalších knih uvedených v Literatuře. Pro dokonalé zvládnutí však nebude stačit studium jakéhokoliv množství literatury. Je třeba zejména vše zkoušet implementovat v příslušném programovacím jazyce.

Po úspěšném zvládnutí základů strukturovaného programování je vhodné se poté poohlédnout po objektovém programování, které je v současné době standardem v programování.

PŘEHLED DOSTUPNÝCH IKON


Čas potřebný ke studiu


Cíle kapitoly


Klíčová slova


Nezapomeňte na odpočinek


Průvodce studiem


Průvodce textem


Rychlý náhled


Shrnutí


Tutoriály

Definice


K zapamatování


Případová studie


Řešená úloha

Věta


Kontrolní otázka


Korespondenční úkol


Odpovědi


Otázky


Samostatný úkol


Další zdroje


Pro zájemce


Úkol k zamyšlení

Název: **Objektové programování**

Autor: **Ing. Radomír Perzina, Ph.D.**

Vydavatel: Slezská univerzita v Opavě

Obchodně podnikatelská fakulta v Karviné

Určeno: studentům SU OPF Karviná

Počet stran: 138