

OBJEKTOVO ORIENTOVANÉ PROGRAMOVANIE

**Peter SCHREIBER
Michal KEBÍSEK
Michal ELIÁŠ**

2013



**SLOVENSKÁ TECHNICKÁ
UNIVERZITA V BRATISLAVE
MATERIÁLOVOTECHNOLOGICKÁ
FAKULTA SO SÍDLOM V TRNAVE**

© Doc. Ing. Peter Schreiber, CSc., Ing. Michal Kebísek, PhD.,
Ing. Michal Eliáš, PhD.

Recenzenti: doc. Ing. Pavel Važan, PhD.
Ing. Miroslav Božík, PhD.

Jazyková korektúra: Mgr. Valéria Krahulcová

Schválilo Vedenie Materiálovotechnologickej fakulty STU ako vysokoškolské
skriptá dňa 24. januára 2012 pre študijný program Aplikovaná informatika
a automatizácia v priemysle

ISBN 978-80-8096-185-5
EAN 9788080961855

ZOZNAM SKRATIEK

BCL	Basic Class Library
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CLS	Common Language Specification
CTS	Common Type System
GAC	Global Assembly Cache
GC	Garbage Collector
IL	Intermediate Language
JIT	Just In Time
LINQ	Language Integrated Query
ODBC	Open Database Connectivity
OOP	Object-Oriented Programming
XML	Extensible Markup Language

ÚVOD

Skriptá Objektovo orientované programovanie sú určené pre študentov Materiálovotechnologickej fakulty Slovenskej technickej univerzity v Bratislave so sídlom v Trnave. Sú vhodným študijným materiálom pre študentov bakalárskeho štúdia študijných programov B-AIA – Aplikovaná informatika a automatizácia v priemysle a B-PPT – Počítačová podpora výrobných technológií, ktorí študujú predmet OBJEKTOVO ORIENTOvané PROGRAMOVANIE. Skriptá sú vhodné aj pre záujemcov, ktorí sa chcú oboznámiť so základmi objektovo orientovaného programovania a so základmi programovania v programovacom jazyku C#. Za účelom lepšieho pochopenia prezentovanej problematiky sú vhodné základné znalosti programovacích jazykov C alebo C++ a základy objektovo orientovaného programovania.

Objektovo orientované programovanie je rozšírená programovacia technika, ktorá je podporovaná vo väčšine moderných programovacích jazykov. Úlohou týchto skrípt je prezentovať postupy tvorby objektovo orientovaných programov, princípy a zásady udalost'ami riadeného programovania.

Skriptá sa skladajú z dvoch základných častí:

- základy objektovo orientovaného programovania,
- objektovo orientované programovanie a jazyk C#.

Prvá časť je venovaná objektovo orientovanému programovaniu, jeho základným princípom a zásadám. Druhá časť obsahuje opis základných vlastností programovacieho jazyka C#, jeho dátové a riadiace štruktúry a implementáciu objektovo orientovaného programovania v tomto programovacom jazyku. Skriptá obsahujú viacero príkladov, ktoré slúžia na prezentáciu riešenej problematiky a pomáhajú k jej lepšiemu porozumeniu. Implementácia programov písaných v programovacom jazyku C# je realizovaná vo vývojovom prostredí Microsoft Visual Studio.

Autori skrípt by na tomto mieste chceli poďakovať recenzentom doc. Ing. Pavlovi Važanovi, PhD. a Ing. Miroslavovi Božíkovi, PhD. za pozorné prečítanie textu a ich cenné rady a pripomienky.

1 ZÁKLADY OBJEKTIVO ORIENTOVANÉHO PROGRAMOVANIA

História vzniku objektovo orientovaného programovania (OOP) siaha do 60. rokov minulého storočia. Do syntaxe programovacieho jazyka bolo po prvýkrát implementované v jazyku Simula 67. Ako napovedá jeho názov, išlo o programovací jazyk vyvinutý na programovanie diskretných simulácií. Neskôr si vývojári uvedomili, že každý program je v skutočnosti simuláciou nejakého reálneho alebo virtuálneho sveta, a preto by mohlo byť užitočné aplikovať tieto myšlienky na všetky programy. Oprávnenosť tejto myšlienky potvrdzuje súčasná prevaha objektovej paradigmy.

Na začiatku 70. rokov vznikla v laboratóriách firmy Xerox v Palo Alto skupina vývojárov, ktorá začala vyvíjať koncepčne úplne nový programovací jazyk, nazvaný Smalltalk. Tento jazyk bol postavený na myšlienkach jazyka Simula 67 a ďalej ich rozvíjal. Simula 67 priniesla základné myšlienky, ale objektovo orientovaná paradigma rovnako, ako aj termín objektovo orientované programovanie priniesli až autori jazyka Smalltalk. Vďaka jazyku Smalltalk sa OOP rozšírilo na univerzity a medzi vedcov zaoberajúcich sa počítačovou vedou.

Programátori pracujúci na softvérových zákazkách ich však používali iba výnimočne. Zmena nastala na konci 70. rokov, keď Bjarne Stroustrup dostal v Bellových laboratóriách za úlohu naprogramovať simuláciu rozsiahlych telefónnych sústav. Poznal jazyk Simula, ale ten mu pripadal príliš pomalý na taký rozsiahly projekt. Na druhej strane, jazyk C, ktorý sa pomaly stával štandardným jazykom systémových programátorov, bol síce dostatočne rýchly, ale bol zasa príliš nízkoúrovňový, takže vývoj takého rozsiahleho systému by bol zbytočne časovo náročný.

Stroustrup sa rozhodol pre dvojfázové riešenie: definoval upravenú verziu jazyka C, ktorú nazval „*C with classes*“ (C s triedami) a vytvoril sústavu makier, ktoré prevádzali program z jazyka Simula do jazyka C. Získal tak silu jazyka Simula a rýchlosť jazyka C. Tento jazyk sa ukázal ako veľmi efektívny, takže sa začal rýchlo rozširovať. V roku 1983 bol jazyk premenovaný na C++ a bol vytvorený vlastný prekladač, ktorý prekladal programy bez predchádzajúceho medzistupňa.

Jazyk C++ sa stával veľmi obľúbeným programovacím jazykom. Umožňoval totiž programátorom pracovať aj naďalej vo svojom obľúbenom jazyku C a pritom kedykoľvek využívať výhody, ktoré poskytovala objektovo orientovaná nadstavba. Tieto konštrukcie

umožňovali výrazne zefektívniť vývoj mnohých programov, takže ich programátori začali používať stále častejšie, až si bez nich onedlho nedokázali predstaviť vývoj programov.

V 80. rokoch sa preto objektovo orientované programovanie začalo šíriť ako lavína. Ako je však známe, nič nie je dokonalé. Programátori používajúci objektovo orientované konštrukcie veľmi často pochopili iba syntax týchto konštrukcií ale nie ich skutočný význam. Často ich preto používali vo svojich programoch nevhodne a vo chvíli, keď program nadobudol určitú veľkosť, začali sa dostávať do problémov. A ako je zvykom ľudí, neobviňovali z týchto problémov vlastnú neschopnosť, ale princípy OOP.

Z toho dôvodu sa na konci 80. rokov rozhodla časť z nich opustiť OOP a vrátiť sa k „starému dobrému“ štruktúrovanému programovaniu. V tej dobe sa ale na odborných konferenciách začali množiť príspevky, ktoré poukazovali na toto nesprávne chápanie zásad OOP a ukazovali, ako by mal programátor „rozmyšľať“ a programovať tak, aby bol jeho program skutočne objektovo orientovaný a aby mu používanie objektovo orientovanej paradigmy prinieslo sľubovaný nárast efektivity vývoja a spoľahlivosti vytvorených programov.

Tieto hlasy sa ale objavovali väčšinou iba na konferenciách a spôsob práce radových programátorov ovplyvňovali iba sporadicky. Zásadnú zmenu prinieslo až vydanie knihy autorov Ericha Gamma, Richarda Helma, Ralpha Johnsona a Johna Vlissidesa „*Design Patterns – Elements of Reusable Object-Oriented Software*“ vydavateľstvom Addison–Wesley v roku 1995. Táto kniha ukázala, ako je potrebné rozmyšľať pri vývoji objektovo orientovaných programov a súčasne priniesla niekoľko overených odporúčaní na riešenie typických programátorských problémov.

1.1 ZÁKLADNÉ POJMY A CIELE OOP

Objektovo orientované programovanie predstavuje programovanie pomocou objektov. Objekt má svoju identitu (predstavuje jeho jednoznačnú identifikáciu), stav (zahŕňa všetky vlastnosti objektu a ich hodnoty) a správanie sa (predstavuje konanie objektu pri zmenách stavu a aktivácii operácií, ktoré poskytuje).

Objektovo orientovaný program sa uskutočňuje ako interakcia objektov. Správanie objektu v objektovo orientovaných jazykoch so statickými typmi definuje jeho typ. Typ objektu sa označuje ako trieda (class). V zdrojových kódach programov sa najčastejšie definujú triedy a objekty pritom predstavujú ich inštancie. Preto sa niekedy zdá, že ide skôr o programovanie pomocou tried. Iný prístup je v dynamických objektovo orientovaných jazykoch, v ktorých

niekedy úplne absentuje pojem triedy ako šablóny na tvorbu objektov a objekty sa vytvárajú priamo.

1.1.1 Základné pojmy

Trieda (class)

- trieda predstavuje predpis (šablónu) pre skupinu inštancií, ktoré nazývame objekty,
- predpis popisuje vnútornú štruktúru objektu,
- objekty rovnakej triedy majú rovnako definované operácie, atribúty a metódy,
- triedy sa využívajú na vytváranie objektov,
- niektoré programovacie jazyky majú schopnosť modifikovať triedy počas činnosti alebo čítať ich štruktúru, táto schopnosť sa nazýva reflexia.

Objekt (object)

- objekt je zoskupenie dát a funkcionalít, ktoré sú spojené za účelom plnenia spoločnej množiny zodpovedností,
- objekty sú organizované v triedach, ktoré združujú ich spoločné vlastnosti,
- objekt má svoju identitu, vlastnosti, správanie a zodpovednosť,
- vlastnosti sú atribúty objektu a správanie je realizované jeho metódami,
- objekt môže predstavovať skutočný objekt z reálneho sveta, ale nie je to podmienkou.

Inštancia (instance)

- inštancia triedy je dátová štruktúra v pamäti počítača, ktorá je vytvorená na základe konkrétnej triedy,
- jednotlivé inštancie sa od sebe líšia umiestením (adresou) v pamäti a hodnotami svojich atribútov,
- adresu aktuálnej inštancie má programátor v objektovo orientovaných jazykoch k dispozícii pod špeciálnym kľúčovým identifikátorom (napr. this alebo self).

1.1.2 Základné ciele OOP

Robustnosť

Robustný program by sa mal správať korektne za každej situácie. Program by mal vracieť správne výstupy pre správne vstupy a vhodne reagovať na nesprávne vstupy.

Prispôsobivosť

Vzhľadom na dĺžku životného cyklu programov by malo byť umožnené tieto programy modifikovať a vylepšovať s minimálnymi nákladmi a rizikom. Ak je to možné, tak je vhodné programy jednoducho prenášať aj na iné platformy.

Znovupoužiteľnosť

Použitie rovnakého kódu vo viacerých častiach programu znižuje jeho zložitosť a robí ho ľahšie pochopiteľným pre programátorský tím. Znižujú sa tak aj celkové náklady na jeho vývoj. Taktiež je vhodné prenášať zdrojový kód medzi viacerými projektmi.

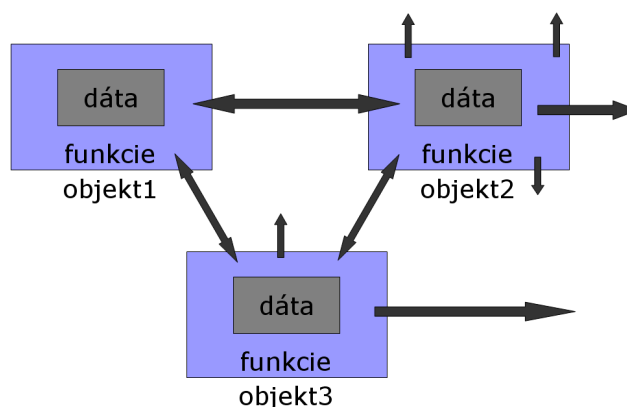
1.2 ZÁKLADNÉ ČRTY OOP

1.2.1 Zapuzdrenie

Objektovo orientované programovanie umožňuje definovať operácie súčasne s dátovým typom, a to tak, že objektový dátový typ obsahuje okrem dátových položiek (atribútov) aj funkčné položky (metódy), ktoré implementujú operácie s dátami. K atribútom by mal byť umožnený prístup iba pomocou metód. Spoločná definícia dátových položiek a operácií s nimi sa nazýva zapuzdrenie (encapsulation).

Zapuzdrenie spája dáta a programový kód, ktorý je od nich závislý. Prináša to viacero výhod pri programovaní.

Zapuzdrením je zabezpečená väčšia bezpečnosť práce s dátami a kódom programu. Ak vznikne v programe chyba, tak jej vyhľadanie by malo byť ľahšie, nakoľko zo správania sa programu je možné zistiť, ktorá časť programu túto chybu spôsobila, a tak túto chybu rýchlejšie lokalizovať. Ak počas programovania je potrebné rozšíriť jeho funkčnosť, je možné vykonať aj zásadné dodatočné zmeny programu bez nutnosti prepisovania celého zdrojového kódu. Je to možné vykonať úpravou, prípadne pridaním požadovanej funkcionality do príslušnej časti triedy. Zapuzdrenie taktiež umožňuje ľahšie rozvrhnutie programu medzi viacero programátorov.



Obr. 1 Zapuzdrenie

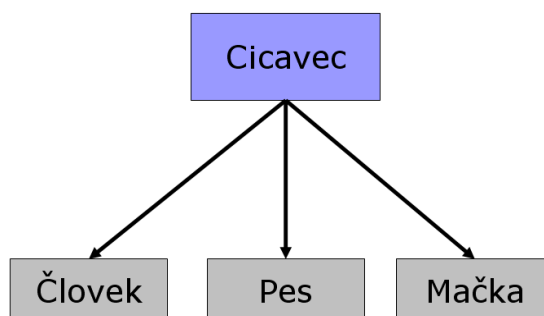
1.2.2 Dedičnosť

Objektovo orientované programovanie ponúka možnosť znovupoužitia už existujúceho programového kódu, a to prostredníctvom dedičnosti (inheritance).

Dedičnosť predstavuje spôsob ako z existujúceho objektového typu odvodiť typ nový. Odvodený typ obsahuje všetky vlastnosti typu, od ktorého bol odvodený a súčasne umožňuje pridať nové dátové položky a metódy. Taktiež je možné zmeniť implementáciu niektorých metód.

Trieda, z ktorej sa bude vytvárať nová trieda, sa zvyčajne nazýva predok, rodič alebo základná trieda. Trieda, ktorá vznikne dedením, sa najčastejšie nazýva potomok, dcérska trieda alebo odvodená trieda.

Prístup k jednotlivým častiam základnej triedy (dátovým položkám a metódam) z odvodenej triedy je riadený prístupovými právami jednotlivých častí základnej triedy. To môže spôsobiť neprístupnosť niektorých metód z odvodenej triedy, ale nemá to vplyv na funkčnosť základnej triedy.



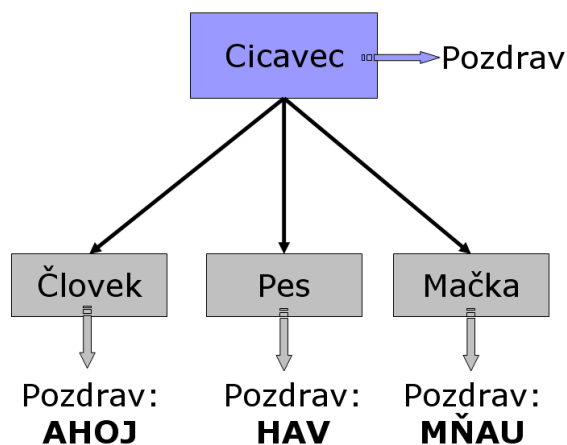
Obr. 2 Dedičnosť

1.2.3 Polymorfizmus

Polymorfizmus (polymorphism) umožňuje používať jednotné rozhranie na prácu s rôznymi typmi objektov. Rôzne objekty reagujú na tú istú správu (udalosť) rôzne. Polymorfizmus umožňuje prepísať metódy rozhrania konkrétnej inštancie tak, aby boli schopné vykonať to, čo je špecifické pre konkrétnu inštanciu, ale zároveň, aby zostalo zachované spoločné rozhranie a programátor nemusel pracovať s viacerými názvami metód.

Polymorfizmus je možné zabezpečiť nasledujúcimi spôsobmi:

- Prekrývanie metód – základom prekrývania je použitie rovnakého mena na označenie viacerých metód. Konkrétna metóda sa zavolá na základe aktuálneho typu vstupného parametra volanej metódy.
- Pretypovávanie – oproti predchádzajúcemu spôsobu realizácie polymorfizmu sa pri pretypovávaní vyžaduje vykonať operáciu konvertovania vstupnej premennej na typ požadovaný metódou, ktorá je volaná.
- Inkluzívny polymorfizmus – objekt môže patriť do viacerých typov, ktoré sú usporiadané podľa inklúzie, tzn. objekt odvodenej triedy sa môže vyskytovať všade tam, kde sa vyžaduje objekt niektorej z jej základných tried. Inkluzívny polymorfizmus je jeden z najčastejšie používaných spôsobov realizácie polymorfizmu v objektovo orientovanom programovaní.
- Parametrický polymorfizmus – podľa aktuálnej hodnoty vstupného parametra metódy sa určujú zvyšné vstupné parametre a celkové správanie sa metódy. Parametrický polymorfizmus sa najčastejšie realizuje využívaním šablón (template), kedy hodnotu špecializovaného parametra nemožno priradiť v čase vytvárania programu, ale musí ju poznať kompilátor.

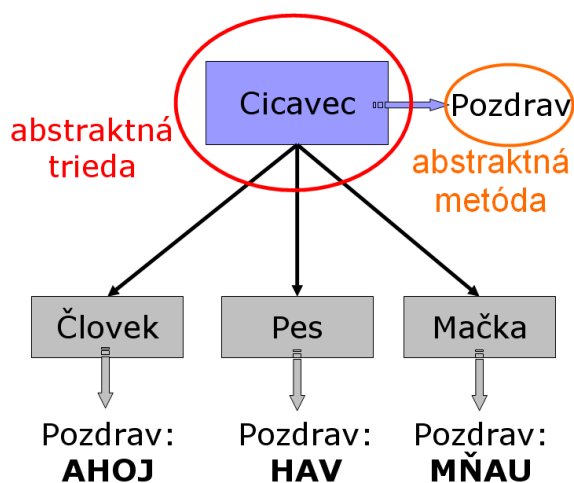


Obr. 3 Polymorfizmus

1.2.4 Abstraktná trieda a metóda

Abstraktná trieda (abstract class) je špeciálna trieda, ktorá je navrhnutá tak, aby tvorila spoločný základ pre iné triedy (pri použití dedičnosti). Z abstraktnej triedy sa nevytvára objekt.

Abstraktné metódy (abstract method) predstavujú iba hlavičky metód bez ich implementácie. Obsahujú ich abstraktné triedy. Neabstraktní potomkovia abstraktnej triedy musia tieto metódy povinne implementovať.



Obr. 4 Abstraktná trieda a metóda

2 OBJEKTOVO ORIENTOvané PROGRAMOVANIE A PROGRAMOVACÍ JAZYK C#

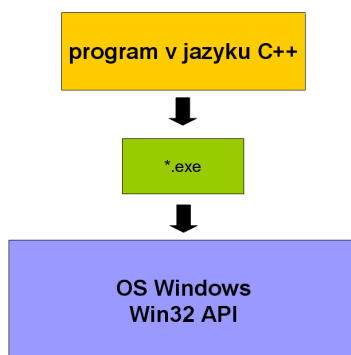
2.1 CHARAKTERISTIKA .NET FRAMEWORKU

.NET Framework je softvérová platforma vyvinutá spoločnosťou Microsoft. Je primárne určená pre operačné systémy Microsoft Windows, aj keď existujú verzie pre iné operačné systémy. Obsahuje viaceré knižnice tried a zabezpečuje možnosť vytvárať programy písané vo viacerých programovacích jazykoch. Programy napísané pre .NET Framework sú spravované v softvéri prostredí, ktoré je nazývané Common Language Runtime (CLR). CLR poskytuje služby ako napr. správu pamäte, bezpečnosť, zachytávanie a obsluhovanie výnimiek a pod. Knižnica tried a CLR tvoria spoločný základ .NET Frameworku.

.NET Framework tvorí vývojové a aplikačné prostredie pre vývoj a správu programov, ktoré môžu byť napísané vo viacerých programovacích jazykoch, ako sú napr. C#, J#, F#, Managed C++, Visual Basic, Object Pascal (Delphi), IronRuby, IronPython, atď. Vývoj takýchto aplikácií, ich správa a integrácia s inými systémami je jednoduchšia ako v „štandardných“ programovacích jazykoch. Umožňuje vývoj konzolových aplikácií, ale aj aplikácií určených na prácu v grafickom prostredí operačných systémov Microsoft Windows a aj aplikácií určených pre webové prostredie.

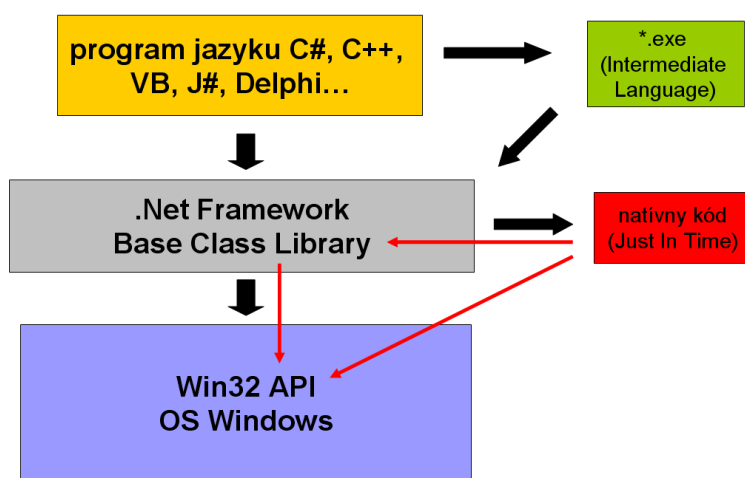
Kompilácia a správa aplikácií pracujúcich v prostredí .NET Framework je odlišná od správy aplikácií priamo v operačnom systéme.

Program napísaný vo zvolenom programovacom jazyku (napr. C++) je prekompilovaný a prelinkovaný kompilátorom a linkerom a výsledkom je spustiteľný súbor v strojovom kóde s koncovkou *.exe. Takto vytvorený súbor je možné spúšťať priamo v operačnom systéme.



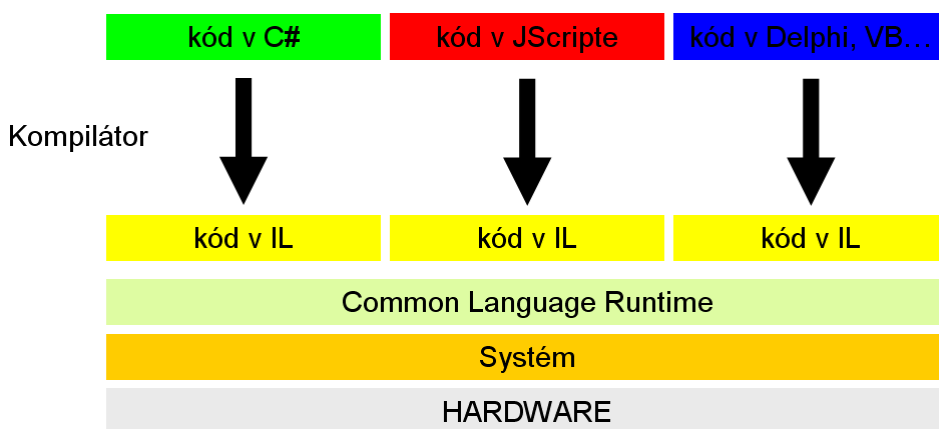
Obr. 5 Vývoj a správa Win32 aplikácie

Na rozdiel od predchádzajúceho príkladu je proces kompilácie aplikácie napísanej pre .NET Framework výrazne odlišný. Aplikácia sa neprekladá priamo do strojového kódu, ale do medzijazyka označovaného IL (Intermediate Language). Ide o analógiu bajtového kódu Java, teda o akýsi assembler virtuálneho počítača. Na rozdiel od Javy sa jazyk IL neinterpretuje, ale vždy sa prekladá do strojového kódu. O to sa starajú JIT (Just In Time) prekladače, ktoré sú v prostredí .NET Framework celkovo tri. Jeden z nich umožňuje preložiť aplikáciu pri inštalácii, druhý v momente jej spustenia. Tretí prekladá jednotlivé metódy vždy v momente, keď sú volané. Preložený kód je uložený v pamäti cache, a tak sú minimalizované straty výkonu.



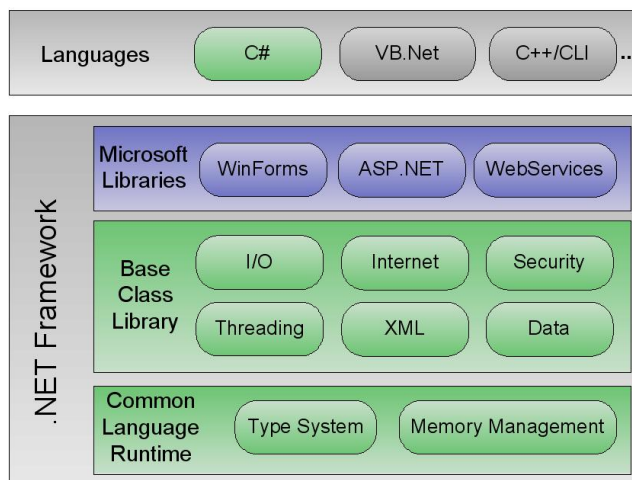
Obr. 6 Vývoj a správa .NET Framework aplikácie

Hierarchia usporiadania jednotlivých častí hardvéru, operačného systému a prostredia .NET Framework je znázornená na nasledujúcom obrázku.



Obr. 7 Hierarchia .NET Framework a OS

Nasledujúci obrázok zobrazuje základnú architektúru .NET Framework.



Obr. 8 Architektúra .NET Framework

Napriek tomu, že .NET Framework je primárne určený pre operačné systémy Microsoft Windows, existuje niekoľko projektov, ktoré umožňujú preniesť niektoré jeho vlastnosti aj na iné operačné systémy.

Projekt Mono, ktorý je produktom nezávislej open source iniciatívy, implementuje .NET runtime na operačné systémy Unixového typu (Linux, MacOS X) a aj na operačný systém Android.

GNU obdoba .NET sa nazýva DotGNU. Jeho časť nazývaná DotGNU Portable.NET umožňuje spúšťať všetky .NET aplikácie na unixových platformách (Linux, BSD, Mac OS X, Solaris, AIX) a dokonca pomocou nástrojov Cygwin a Mingw32 aj na operačných systémoch Windows.

Vo všetkých platformách .NET používa rovnaké základné princípy, čo umožňuje jednoduchší prechod medzi nimi. Problém pri prechode však spôsobuje neúplná implementácia vlastností .NET Frameworku pre operačný systém Windows do verzií pre iné operačné systémy.

2.1.1 Základné prvky .NET Frameworku

.NET Framework je založený na otvorených štandardoch CLI (Common Language Infrastructure) (ECMA 335):

- CLR (Common Language Runtime) – jazykovo neutrálne vývojové a aplikačné prostredie poskytujúce služby pre prebiehajúce .NET aplikácie,

- BCL (Base Class Library) – knižnica základných tried (na prácu s databázami, webovými a sieťovými službami, na prácu s obrázkami...).

CLR – Common Language Runtime

Základom .NET Frameworku je spoločné priebehové prostredie CLR (Common Language Runtime). Predstavuje prostredie, ktoré zabezpečuje beh a vzájomnú spoluprácu aplikácií napísaných v rôznych programovacích jazykoch (napr. C#, J#, Visual Basic, IronRuby, IronPython...) určených na prácu v .NET Frameworku. CLR je založené na spoločnej špecifikácii jazyka CLS (Common Language Specification) a obsahuje pravidlá, ktorými sa riadia vývojári prekladačov ľubovoľných programovacích jazykov určených pre .NET Framework.

Spoločné priebehové prostredie obsahuje Just In Time kompiláciu a zároveň vykonáva verifikáciu zdrojového kódu podľa špecifických pravidiel (kontrola a správa pamäte, kontrola dátových typov, blokovanie poškodeného a nebezpečného kódu...).

IL/MSIL/CIL – Intermediate Language/Microsoft Intermediate Language/Common Intermediate Language

Programy vytvorené pre .NET Framework nie sú prekladané priamo do strojového kódu, ale do jazyka označovaného IL (Intermediate Language). Často býva tento jazyk označovaný aj MSIL (Microsoft Intermediate Language) alebo CIL (Common Intermediate Language).

Základné vlastnosti IL:

- predstavuje obdobu JAVA bytecode,
- je prísne objektový,
- prísne rozlišuje medzi referenčnými a hodnotovými typmi,
- prísna typizácia dát,
- používanie atribútov,
- ošetrovanie chýb pomocou výnimiek,
- nie je interpretovaný,
- Just In Time kompilácia prebieha inkrementálne – prekladá sa iba tá časť kódu, ktorá sa má vykonať,
- je možné optimalizovať JIT kompiláciu pre ľubovoľnú platformu a hardvérovú konfiguráciu.

Garbage Collector – Automatická správa pamäte

Súčasťou priebehového prostredia (CLR) je aj automatická správa pamäte – Garbage Collector (GC). Umožňuje programátorovi vytvárať inštancie objektových typov alebo polí podľa jeho potreby, ale odbreňuje ho od nutnosti starať sa o ich rušenie. Objekty existujú dovtedy, pokiaľ na ne existuje aspoň jeden odkaz. Keď zanikne aj posledný odkaz na ne, tak sa automatická správa pamäte postará o ich korektné uvoľnenie z pamäte. K tomu nemusí prísť hneď pri zániku posledného odkazu na objekt, ale niekedy aj neskôr, napr. v závislosti od toho, či môže dôjsť k vyčerpaniu pamäte pridelenej aplikácii. Môže nastať aj taká situácia, že k uvoľneniu pamäte nedôjde počas priebehu aplikácie, ale až po jej ukončení. Spúšťanie automatickej správy pamäte je nedeterministické – spúšťa sa v závislosti od potrieb a stavu systému.

Assembly – zoskupenie

Zoskupenie je základnou distribučnou jednotkou programov, komponent... v prostredí .NET Framework. Zoskupenie môže pozostávať z viacerých modulov, ktoré sú vzájomne prepojené manifestom. Modul vznikne prekladom zdrojového súboru do IL. Predstavuje analógiu súboru *.obj z prostredia operačných systémov MS DOS a MS Windows.

Manifest je tvorený popisom obsahu daného zostavenia. Zostavenie môže mať označenie verzie, čo umožní rozlíšiť zostavenia s rovnakým názvom, ale rôznych verzií. Ďalej sa skladá z metadát popisujúcich obsiahnuté typy. Keď sa štandardne distribuuje natívna *.dll knižnica, tak sa používa hlavičkový súbor alebo dokumentácia popisujúca exportované funkcie. Metadáta spĺňajú tieto nároky a kompletne popisujú všetky obsiahnuté zostavenia. Zostavenia sú teda, stručne povedané, samopopisné jednotky v prostredí CLR a môžu byť rozličných verzií.

Metadáta predstavujú popis zoskupenia. Generuje ich priamo kompilátor a sú automaticky ukladané do *.exe alebo *.dll súboru (podľa zvoleného typu projektu) a sú zapisované v binárnej forme. Metadáta je možné exportovať do a z XML súboru alebo knižnice typu COM.

Metadáta obsahujú:

- popis distribučnej jednotky (zoskupenia) – meno, verziu, užívateľom nastavený jazyk, verejný verifikačný kľúč...,
- zoznam všetkých dátových typov definovaných v tomto zoskupení,
- zoznam všetkých zoskupení, na ktoré sa programy v danom zoskupení odkazujú,

- zoznam súborov, ktoré tvoria zoskupenie,
- bezpečnostné nastavenia potrebné na spracovanie,
- základné triedy a rozhrania použité týmto zoskupením,
- užívateľské atribúty (vložené programátorom),
- kompilátorom definované atribúty.

V prostredí .NET Framework je možné vytvárať dva základné typy zoskupení:

- súkromné zoskupenie (private assembly),
- spoločne používané zoskupenie (shared assembly).

Súkromné zoskupenie:

- používané jedným programom,
- používajú voľnejšie konvencie pomenovávania,
- bývajú uložené v adresári s aplikáciou.

Spoločne používané zoskupenie:

- používané viacerými programami,
- strong naming – unikátne mená – rieši problém pomenovania zoskupení,
- GAC – Global Assembly Cache – rieši problém používania rôznych verzií zoskupení.

Prostredie .NET Framework umožňuje mať súčasne vedľa seba niekoľko verzií toho istého spoločne používaného zoskupenia. Aplikácia môže mať svoje vlastné privátne verzie zoskupení, ktoré môžu byť prednostne použité pred rovnakými spoločne používanými verziami.

Na rozdiel od štandardných programovacích jazykov podporujúcich objektovo orientované programovanie, slúži zoskupenie používané v prostredí .NET Framework aj ako oblasť, na ktorú je možné definovať prístupové práva.

Namespaces – menné priestory

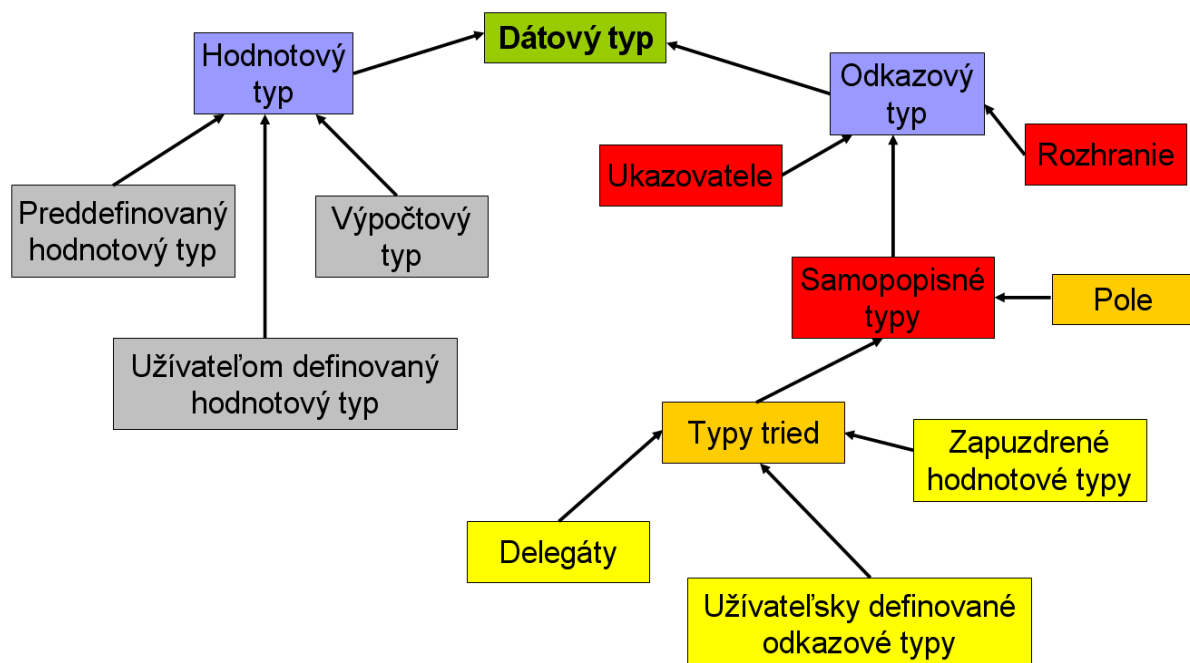
Menné priestory predstavujú nástroj na organizáciu identifikátorov v rozsiahlom programe a v knižniciach. Ich hlavný prínos je:

- pomáhajú organizovať knižnice objektov a hierarchií,
- zjednodušujú odkazy na objekty,
- zabráňujú dvojznačnostiam pri odkazoch na objekty,
- riadia rozsah platnosti identifikátorov objektov,

- deklarujú sa predtým, ako sa používajú,
- deklarácia menného priestoru umožní používať skrátené názvy príkazov.

2.1.2 Spoločný typový priestor – Common Type System

Dôležitou súčasťou priebehového prostredia (CLR) .NET Frameworku je spoločný typový priestor – Common Type System (CTS). Predstavuje súbor pravidiel na definíciu dátových typov a spôsob práce s nimi. Ich dodržiavanie zaručí, že môžeme dátový typ vytvorený v jednom programovacom jazyku napr. C#, používať aj inom programovacom jazyku, ktorý pracuje v rámci .NET Frameworku napr. vo Visual Basicu. Nejde iba o vytvorenie objektových inštancií alebo inštancií iných typov, ale aj o odvodzovanie potomkov a iné operácie. Vzájomná kompatibilita programovacích jazykov umožňuje inicializáciu tried napísaných v rôznych programovacích jazykoch a aj prístup k vlastnostiam tried napísaných v rôznych programovacích jazykoch.



Obr. 9 Common Type System

Základné rozdelenie dátových typov je na:

- hodnotový typ – alokuje sa na zásobníku,
- odkazový typy – na zásobníku má odkaz, v dynamickej pamäti dáta.

Medzi hodnotové dátové typy patria:

- preddefinovaný hodnotový typ – štandardný primitívny dátový typ vyjadrujúci číslo, logickú hodnotu alebo znaky,
- výpočtový typ – množina výpočtových hodnôt,
- užívateľom definovaný hodnotový typ – typ definovaný v zdrojovom kóde a ukladaný ako hodnotový typ (napr. štruktúra).

Medzi odkazové dátové typy patria:

- rozhranie – typ, na ktorý je možné pretypovať inštanciu tried implementujúcu dané rozhranie,
- ukazovatele – ukazovatele (pointery),
- samopopisné typy – typy poskytujúce automatickej správe pamäti informácie o nich samotných,
- polia – typ obsahujúci pole objektov,
- typy tried – typy, ktoré sú samopopisné, ale nie sú polia,
- delegáty – typy navrhnuté na nesenie odkazov na metódy,
- užívateľom definované odkazové typy – typy definované v zdrojovom kóde a ukladané ako odkazové typy (napr. triedy),
- zapuzdrené hodnotové typy – hodnotový typ, ktorý je dočasne zabalený do odkazu tak, že môže byť uložený v dynamickej pamäti.

2.1.3 Verzie .NET Frameworku

.NET Framework 1.0 (2002)

Prvá verejná verzia .NET Frameworku. Priniesol nový programovací jazyk C# vo verzii 1.0. Je súčasťou Microsoft Visual Studio 7.0.

.NET Framework 1.1 (2003)

Vylepšená verzia. Priniesol podporu pre mobile zariadenia, zvýšenie bezpečnosti, podporu protokolu IPv6. Ďalej rozšíril podporu ODBC a Oracle databáz. Je súčasťou Microsoft Visual Studio 2003.

.NET Framework 2.0 (2006)

Verzia priniesla nové verzie jazykov C# 2.0 a VB.NET 8.0. Jeho súčasťou je aj nová verzia CLR 2.0 (Common Language Runtime), ktorá pridala nové triedy do BCL (Base Class Library) vrátane generík (generics) a generických kolekcií (generic collections). Priniesol

plnú podporu 64-bitových aplikácií, zmeny v ASP.NET, vylepšenú prácu s databázami Microsoft SQL Server 2005. Je súčasťou Microsoft Visual Studio 2005.

.NET Framework 3.0 (2006)

Táto verzia .NET Frameworku je viac-menej rovnaká ako predchádzajúca verzia. Jej hlavný prínos je pridanie WPF (Windows Presentation Foundation), WCF (Communication), WF (WorkFlow) a Windows CardSpace. Z tohto dôvodu býva často táto verzia nazývaná aj WinFX. Je súčasťou Microsoft Visual Studio 2008.

.NET Framework 3.5 (2007)

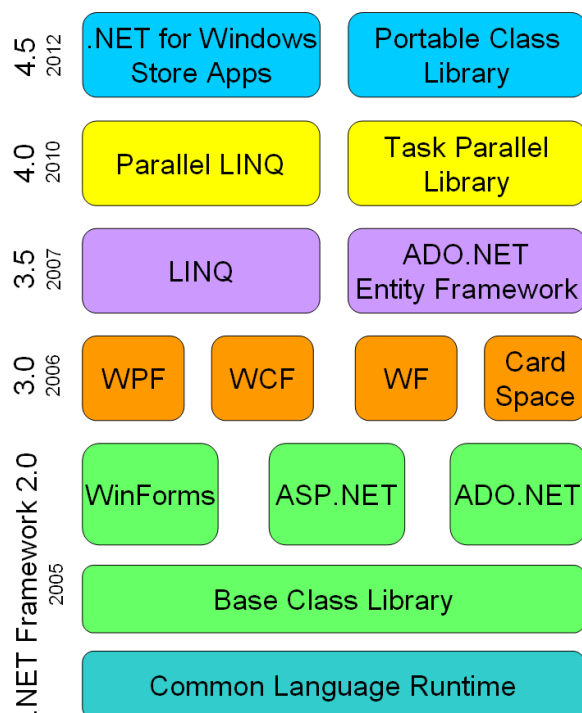
Na rozdiel od predchádzajúcej verzie priniesla verzia 3.5 nové verzie jazykov C# 3.0 a VB.NET 9.0. Stále využíva CLR 2.0, ale bol pridaný LINQ (Language Integrated Query) for Object, vylepšená práca s XML, SQL, podpora ASP.NET AJAX. Je súčasťou Microsoft Visual Studio 2008 SP1.

.NET Framework 4.0 (2010)

Verzia 4.0 priniesla rozšírenie paralelného programovania pre distribuované a multiprocessorové systémy. Obsahuje novú verziu CLR 4.0 (verzia CLR 3.0 nie je), novú verziu jazyka C# 4.0 a plnú podporu programovacích jazykov IronPython, IronRuby a F#. Je súčasťou Microsoft Visual Studio 2010.

.NET Framework 4.5 (2012)

.NET Framework 4.5 pridal podporu pre väčšie dátové objekty (objekty väčšie ako 2GB pre 64-bitové operačné systémy), podpora Windows Store App, background JIT kompiláciu, Portable Class Library, asynchrónne operácie so súbormi, zlepšenie paralelného spracovania, Managed Profile Guided Optimization pre optimalizáciu spúšťania aplikácií a využívania operačnej pamäte. Súčasťou tejto verzie je aj nové CLR 4.5, nová verzia programovacieho jazyka C# 5.0 a vylepšenia ASP.NET, WPF, WCF, WF... . Je súčasťou Microsoft Visual Studio 2012.



Obr. 10 Hlavné časti verzií .NET Framework-u

Jednotlivé verzie .NET Frameworku sú súčasťou príslušnej verzie vývojového prostredia Microsoft Visual Studio, inštalácií operačných systémov Microsoft Windows, prípadne je možné ich voľne stiahnuť do operačného systému zo stránky Microsoft. Do jednotlivých operačných systémov nie je možné inštalovať ľubovoľné verzie .NET Frameworku. Vzájomná kompatibilita jednotlivých verzií .NET Frameworku a rôznych verzií operačného systému MS Windows je uvedená na nasledujúcom obrázku.

OS	.NET verzia	1.0	1.1	2.0	3.0	3.5	4.0	4.5
Windows 95								
Windows NT								
Windows 98, Windows 98 SE								
Windows Me								
Windows 2000								
Windows XP								
Windows Server 2003								
Windows Vista								
Windows Server 2008								
Windows Server 2008 R2								
Windows 7								
Windows 8								

	nepodporovaný
	možnosť doinštalovať
	kompatibilný
	čiasť kompatibilita
	súčasť operačného systému

Obr. 11 Kompatibilita jednotlivých verzií .NET Frameworku a OS MS Windows

2.1.4 Verzie programovacieho jazyka C#

Programovací jazyk C# je súčasťou .NET Frameworku, z toho dôvodu dochádza pri vydaní novej verzie .NET Frameworku k určitým zmenám aj v programovacom jazyku C#. Jednotlivé verzie a najdôležitejšie zmeny v nich sú uvedené v nasledujúcom prehľade.

C# 1.0

Prvá verzia, vydaná ako súčasť .NET Frameworku 1.0 v roku 2002. Obsahovala základnú podporu objektovo orientovaného programovania.

C# 2.0

Druhá verzia programovacieho jazyka C# vyšla spolu s .NET Framework 2.0 v roku 2005. Základne vlastnosti tejto verzie:

- natívna podpora genericity (Generics) vychádzajúca z podpory na úrovni CLI,
- statické (static class) a čiastkové triedy (partial class),
- iterátory,
- anonymné metódy na jednoduchšie používanie delegátov,
- anonymné delegáty,
- nullovateľné hodnotové typy a operátor koalescence,
- možnosť nastavenia rôznych prístupností na čítanie a zápis vlastností triedy (properties).

C# 3.0

Vyšiel na konci roku 2007 spoločne s .NET Frameworkom 3.5. Obsahuje pomerne výrazné zmeny, ale zostáva zachovaná predchádzajúca verzia IL.

Nové vlastnosti jazyka C# 3.0:

- LINQ (Language Integrated Query),
- lambda výrazy,
- rozširujúce metódy,
- inicializátory objektov a kolekcí,
- výrazové stromy (expression trees),
- anonymné triedy (anonymous classes),
- anonymné typy (anonymous types),
- autoimplementačné vlastnosti,

- čiastkové metódy (partial methods).

C# 4.0

Ďalšia verzia tohto jazyka bola vydaná ako súčasť .NET Framework 4.0 v roku 2010. Nová verzia sa zameriava hlavne na spoluprácu s dynamickými aspektmi programovania a frameworky, ako napríklad DLR (Dynamic Language Runtime) a COM. Medzi ďalšie novinky patrí:

- kovariancia a kontravariancia,
- voliteľné parametre a pomenované parametre,
- dynamicky typované objekty (dynamic binding).

C# 5.0

Zatiaľ posledná verzia programovacieho jazyka C# (v čase vydania skript) je verzia 5.0, ktorá je súčasťou .NET Frameworku 4.5, vydaného v auguste 2012.

Jej najväčšie prínosy sú:

- informácie o kóde pri volaní (caller info attributes),
- asynchrónne metódy (asynchronous methods).

2.2 CHARAKTERISTIKA JAZYKA C#

C# je jednoduchý, moderný, objektovo orientovaný jazyk odvodený z C++ a jazyka Java. Spája vysokú produktivitu jazyka VB so silou jazyka C++. Do MS Visual Studio je zaradený od verzie 7.0. Triedy a dátové typy má spoločné pre všetky programovacie jazyky v .NET Framework. Umožňuje vyvíjať aplikácie pre konzolu, grafické prostredie Windows a Web.

Podporuje zapuzdrenie dát, dedičnosť, polymorfizmus, rozhrania a ďalšie vlastnosti OOP. Na rozdiel od C++ bežne nepoužíva ukazovatele (pointer). Používanie ukazovateľov je povolené len obmedzene, a to ako „unsafe“ blok kódu a na manipulovanie so starým kódom. Nepoužíva operátory ":" alebo "->".

Nie je povolené používať operácie pre priamu správu pamäti, ale využíva automatickú správu pamäte a garbage collection.

Kompilátor automaticky inicializuje hodnotové typy (primitive types) na nulu a referenčné typy (objekty a triedy) sú inicializované na null. Polia (arrays) sú indexované od nuly a sú testované ich hranice. Testuje sa pretečenie (overflow) typov. Neumožňuje vykonávať neuskutočniteľné typové konverzie, napr. double na boolean.

Príklad: *Sharp Hello bez použitia using*

Zdrojový kód jednoduchého programu, ktorý vypíše na obrazovku daný text a čaká na stlačenie klávesu. Pri volaní funkcií z menného priestoru System sa na ne odkazujeme celým menom vrátane menného priestoru.

```
namespace SharpHello
{
    public class Program
    {
        public static void Main(string[] args)
        {
            System.Console.WriteLine("Prvy program v C#");
            System.Console.ReadKey();
        }
    }
}
```

Príklad: *Sharp Hello s použitím using*

Zdrojový kód jednoduchého programu, ktorý vypíše na obrazovku daný text a čaká na stlačenie klávesu. V úvode je deklarované predvolené používanie menného priestoru System, čo umožňuje volanie funkcie kratším menom bez príslušného menného priestoru.

```
using System;
namespace SharpHello
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Prvy program v C#");
            Console.ReadKey();
        }
    }
}
```


3 DÁTOVÉ TYPY

V programoch pracujeme s rôznymi druhmi informácií – dát. Tieto dáta sú najčastejšie číselné, napr. počet študentov v učebni, priemerná známka z predmetu, hodnota nameranej veličiny, veľkosť okna a iné. Často sa stretávame aj s textovými údajmi, napr. pri spracovaní zoznamov študentov nás zaujíma meno, priezvisko, adresa a pod. V neposlednom rade používame dáta vyjadrujúce logický stav – pravda (true) alebo nepravda (false), resp. či niečo platí alebo neplatí.

Typ údajov, či ide o celé číslo, reálne číslo, text alebo booleovskú hodnotu, označujeme ako dátový typ. Najčastejšie sa stretneme s typom `int` (celé číslo), `double` (reálne číslo), `bool` (booleovská hodnota), `string` (text), `char` (jeden znak). Dátové typy tiež rozdeľujeme podľa veľkosti hodnoty, ktorú daný dátový typ dokáže uchovať (pozri nasledujúcu tabuľku).

Pri používaní platformy .NET Framework sú základné dátové typy zastúpené ich ekvivalentom. Napr. typ `int` je zastúpený typom `System.Int32` a podobne je to aj s ostatnými dátovými typmi.

DÁTOVÉ TYPY V JAZYKU C#

Tabuľka

Typ v C#	Typ v .NET Framework (System)	±	Bytes	Rozsah
sbyte	System.Sbyte	A	1	-128 ... 127
short	System.Int16	A	2	-32768 ... 32767
int	System.Int32	A	4	-2147483648 ... 2147483647
long	System.Int64	A	8	-9223372036854775808 ... 9223372036854775807
byte	System.Byte	N	1	0 ... 255
ushort	System.UInt16	N	2	0 ... 65535
uint	System.UInt32	N	4	0 ... 4294967295
float	System.Single	A	4	$\pm 1.5 \times 10^{-45}$... $\pm 3.4 \times 10^{38}$
double	System.Double	A	8	$\pm 5.0 \times 10^{-324}$... $\pm 1.7 \times 10^{308}$
decimal	System.Decimal	A	12	$\pm 1.0 \times 10^{-28}$... $\pm 7.9 \times 10^{28}$
char	System.Char	N/A	2	znak Unicode (16 bit)
bool	System.Boolean	N/A	1/2	true alebo false

Každý dátový typ obsahuje atribúty `MinValue`, `MaxValue`, pomocou ktorých je možné zistiť hornú a dolnú hranicu číselného rozsahu daného dátového typu.

```
DatTypNet.MinValue
```

```
DatTypNet.MaxValue
```

napr.:

```
Int32.MinValue
```

```
UInt32.MaxValue
```

3.1 DEFINÍCIA PREMENNÝCH A KONŠTÁNT

V jazyku C# nie je možné definovať globálne premenné ako v C++, preto sú všetky premenné v C# iba „lokálne“. Premenné sa môžu definovať na mieste, kde sú potrebné (priamo v kóde programu), prípadne za účelom prehľadnosti na začiatku funkčných blokov (procedúr, funkcií, tried, hlavného programu...). Definíciu premennej je možné spojiť s jej inicializáciou.

Názvy premenných

Názov premennej by mal vystihovať jej obsah, tzn. ak definujete premennú pre uchovanie dĺžky stola alebo hĺbky bazéna, tak ich nenazvete `abcd` alebo `h`, ale zmysluplným a samopopisným názvom, napr. `DlзкаStola`, `HlbkaBazena`.

Obmedzenia pre názov premennej:

- môže pozostávať len z alfanumerických znakov (A-Z, a-z, 0-9) a podtržníka (`_`),
- nesmie začínať číslou,
- nesmie obsahovať medzery a iné znaky, ako už boli spomínané, napr. bodka (`.`), čiarka (`,`), lomka (`/`), spätná lomka (`\`), otáznik (`?`), zátvorky,
- nesmie byť dlhší ako 255 znakov,
- nesmie sa zhodovať s niektorým z kľúčových slov jazyka (napr. `break`, `int`, `class`, ...),
- v jednom rozsahu platnosti nemôžete nazvať premennú rovnako ako inú premennú.

Odporúča sa tiež dodržiavať konvenciu používania predpony vystihujúcej v skratke dátový typ danej premennej, napr. `d` pre `double`, `i` pre `int`, `b` pre `bool`, `s` pre `string`, príp. tieto predpony môžu mať aj viac písmen, napr. `str` pre `string`, `ch` pre `char` a pod. Ako príklad môžeme uviesť: `iDlзкаStola`, `dHlbkaBazena`, `strPriezvisko`.

Premenné

Premenná je vyhradené miesto v pamäti prístupné pomocou identifikátora (názvu), v ktorom je možné uchovať údaje príslušného dátového typu. Používa sa na prácu s údajmi, ktorých hodnota sa počas priebehu programu mení. Pri premenných definovaných v rámci bloku funkcie sa neuvádza modifikátor prístupu.

```
DatovyTyp Meno;  
DatovyTyp Meno1, Meno2, ..., MenoN;  
DatovyTyp Meno = Hodnota;  
int a;  
float StranaA, StranaB;  
string Meno = "Karol";
```

Konštanty

Konštanta je špeciálny typ premennej, ktorej hodnota sa nedá meniť počas priebehu programu, možno ju len inicializovať, teda hodnota je známa už počas kompilácie programu. Má predponu `const`. Definovanie premennej ako konštanty má aj ochranný charakter, ak nechceme aby ktokoľvek, hoci aj náhodne, mohol zmeniť jej hodnotu.

```
const DatovyTyp Meno = Hodnota;  
const int DPH = 20;  
const string Skola = "MTF STU";
```

Konštanty majú najväčšie uplatnenie pre hodnoty, ktoré sa môžu v zdrojovom kóde zmeniť niekedy v budúcnosti. Ak počítame napr. cenu tovaru s dph, tak číslo 20 (ako hodnotu dph) vo vzorcoch nahradíme konštantou DPH, čím si do budúcnosti výrazne uľahčíme prácu, ak by sa hodnota dph zmenila. Nemusíme robiť úpravu v celom kóde programu, ale iba zmeníme hodnotu konštanty.

Premenné iba na čítanie

Je to špeciálny typ konštanty, ktorej hodnota nemusí byť známa už počas kompilácie programu (pri inicializácii). Je možné ju nastaviť až počas priebehu programu iba jedenkrát, a to iba v konštruktoze triedy. Má predponu `readonly`.

```
readonly int Koeficient;
```

Kontrola pretečenia premennej

Pretečenie nastáva, ak sa program pokúša zapísať do premennej väčšie alebo menšie číslo, ako je maximálna alebo minimálna hodnota daná príslušným dátovým typom premennej. Pretečenie premennej nie je kontrolované implicitne. Z toho dôvodu program nevyhlási chybu, ale začne sa správať „čudne“. Kontrolu pretečenia je možné povoliť pomocou príkazu `checked` a zakázať príkazom `unchecked`.

Enumerátory – vymenované typy

Enumerátor je zoskupenie množiny konštánt rovnakého (alebo podobného) významu. Vnútorne sú jednotlivé enumeráty prezentované pomocou číselných konštánt od 0 (je to možné zmeniť v programe priradeným požadovanej hodnoty konštante),

```
<ModifPristupu> enum MenoEnum
{konst1, konst2, ... , konstN};

public enum Den
{ pondelok, utorok, streda, stvrtok, piatok, sobota, nedela };
```

Výpis enumerátu (hodnota – streda):

```
Den.streda;
```

Výpis vnútornej hodnoty enumerátu (hodnota – 2):

```
(int)Den.streda;
```

Premenné – delenie nulou

Pri delení čísel môže niekedy dôjsť k deleniu nulou. V štandardných programovacích jazykoch ako napríklad C, C++, Pascal a podobne nie je možné deliť nulou a pri pokuse o delenie nulou je vyvolaná výnimka. To isté platí aj v programovacom jazyku C#, avšak iba pri práci s celými číslami. Pre reálne čísla obsahuje jazyk C# definované špeciálne hodnoty výsledku delenia.

Výsledok delenia čísla nulou:

$1/0$ = PositiveInfinity	– kladné nekonečno,
$-1/0$ = NegativeInfinity	– záporné nekonečno.

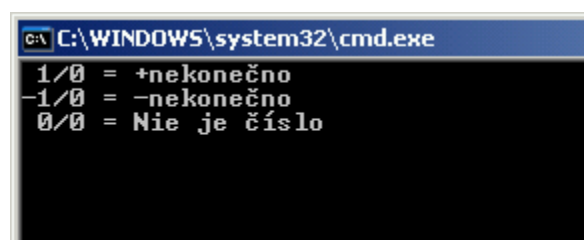
Výsledok delenia nuly nulou:

$0/0 = \text{NaN}$ (Not a Number) – nedefinované číslo.

Príklad: *Delenie nulou – program*

Príklad demonštruje spôsob použitia delenia dvoch reálnych čísel, ak je deliteľ rovný nule a delenec má kladnú, zápornú alebo nulovú hodnotu.

```
using System;
namespace DelenieNulou
{
    class Program
    {
        static void Main(string[] args)
        {
            float cislo1 = 1, cislo2 = 0;
            Console.WriteLine(" 1/0= {0}", cislo1/cislo2);
            Console.WriteLine("-1/0= {0}", -cislo1/cislo2);
            Console.WriteLine(" 0/0= {0}", cislo2/cislo2);
            Console.ReadKey();
        }
    }
}
```



Obr. 12 Delenie nulou – výsledok v konzole

3.2 KONVERZIA DÁTOVÝCH TYPOV

S touto konverziou sa stretávame predovšetkým pri spracovávaní vstupných údajov zadávaných používateľom ako vstup z klávesnice, prípadne pri čítaní údajov z textových polí alebo zo súborov. Pri konverzii je potrebné dopredu vedieť, akého typu a veľkosti majú byť konvertované textové údaje.

Konverzia textu na číslo

Každý preddefinovaný dátový typ má zabudovaný parser, ktorý sa stará o spracovanie vstupného textu, spĺňajúceho určité požiadavky a jeho konverziu na daný dátový typ.

Príklad: Konverzia textu na číslo

Pri zadávaní čísel z klávesnice je nutné zadané znaky skonvertovať na číslo. Toto zabezpečuje metóda `Parse()`, ktorú má každý číselný dátový typ v .NET Framework. Metóde `Parse()` je ako parameter odovzdaná hodnota načítaná z konzoly pomocou metódy `ReadLine()`.

```
using System;
namespace KonverziaTextNaCislo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Zadaj text: ");
            string Retazec = Console.ReadLine();
            Console.Write("Zadaj cislo: ");
            short shortCislo = Int16.Parse(Console.ReadLine());
            int intCislo = Int32.Parse(Console.ReadLine());
            float floatCislo = Single.Parse(Console.ReadLine());
            double doubleCislo = Double.Parse(Console.ReadLine());
            Console.ReadKey();
        }
    }
}
```

Konverzia číselných hodnôt

Pri práci s číselnými hodnotami rôzneho typu dochádza ku konverzii jedného typu na druhý. Konverzia je bezpečná, ak sa hodnota z rozsahovo menšieho dátového typu ukladá do väčšieho. Ak je to ale naopak, tak môže dôjsť k pretečeniu alebo k strate informácie, napr. pri snahe zapísať reálne číslo do celočíselnej premennej, pridáme o desatinné miesta.

Implicitnú konverziu realizuje kompilátor automaticky pri vzájomne kompatibilných dátových typoch. V prípade možných problémov upozorní varovaním a pri nekompatibilitě chybou.

Explicitnú konverziu musí prikázať programátor napr. pretypovaním. V tomto prípade sa už varovania nezobrazia.

Príklad: *Implicitná a explicitná konverzia číselných hodnôt*

Implicitná konverzia je demonštrovaná priradením hodnoty premennej typu `byte` do premennej typu `ushort`. Konverzia bude bezproblémová, pretože rozsahovo menší dátový typ sa konvertoval na väčší.

Explicitná konverzia sa vykoná umiestnením cieľového dátového typu v zátvorke pred konvertovanú premennú iného typu. V tomto prípade priradíme hodnotu premennej väčšieho dátového typu do premennej menšieho dátového typu. Takéto priradenie by bez explicitnej konverzie skončilo varovaním kompilátora. Tým, že pred konvertovanú premennú typu `ushort` zapíšeme `(byte)`, dáme kompilátoru najavo, že o tom vieme a chceme to urobiť napriek nožnej strate časti informácie.

```
using System;
namespace ImplAExplKonverzia
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Implicitna konverzia");
            byte byCislo1 = 100;
            ushort usCislo2 = byCislo1;
            Console.WriteLine("Hodnota byte {0} a ushort {1}",
                              byCislo1, usCislo2);
            Console.WriteLine("\nExplicitna konverzia");
            ushort usCislo3 = 300;
            byte byCislo4 = (byte)usCislo3;
            Console.WriteLine("Hodnota byte {0} a ushort {1}",
                              byCislo4, usCislo3);
        }
    }
}
```

```
C:\ file:///E:/CSharp/CSharpConsole_Konverzia/CSharp
Implicitna konverzia
Hodnota byte 100 a ushort 100

Explicitna konverzia
Hodnota byte 44 a ushort 300
```

Obr. 13 Implicitná a explicitná konverzia číselných hodnôt – výsledok v konzole

V predošlom príklade vidno prípad pretečenia čísla – prekročenie rozsahu dátového typu. Do premennej `byCislo4`, ktorá je typu `byte`, ukladáme číslo s väčšou hodnotou, ako je rozsah dátového typu `byte` (0...255). Výsledná hodnota 44 je časťou 16-bitového čísla 300, ktorá sa zmestila do 8-bitového rozsahu premennej. Pretečenie je najnázornejšie vidieť pri binárnom zápise celej operácie.

00000000 = 0	8-bitová premenná
<u>00000001 00101100 = 300</u>	16-bitové číslo (256 + 44)
00101100 = 44	8-bitová premenná po zápise 16-bitového čísla

Konverzia čísla na text

Všetky preddefinované dátové typy majú metódu `ToString()`, pomocou ktorej možno prekonvertovať číselnú informáciu na textovú v požadovanom formáte.

Varianty metódy `ToString()` sú:

- `ToString()`
- `ToString(string)`
- `ToString(IFormatProvider)`
- `ToString(string, IFormatProvider)`

kde:

`string` – formátovaný reťazec,

`IFormatProvider` – formátovací predpis.

Metódu je možné použiť s triedou `Convert` na konverziu z ľubovoľného dátového typu.

Parsovanie reťazcov

Všetky preddefinované dátové typy majú metódu `Parse()`, ktorá umožňuje prekonvertovať textový vstup požadovaného formátu na daný dátový typ.

Varianty metódy Parse() sú:

- Parse(string)
- Parse(string, IFormatProvider)

kde:

string – formátovaný reťazec,

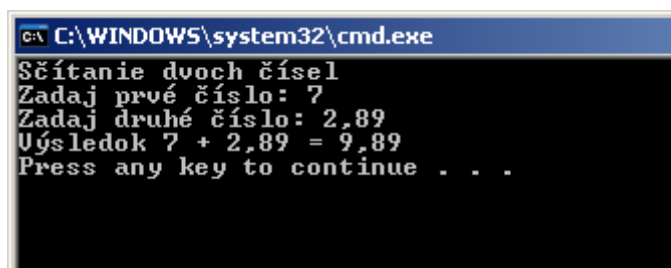
IFormatProvider – formátovací predpis.

Vrátená hodnota je vždy požadovaného typu. Ak nie je možné prekonvertovať hodnotu na požadovaný typ, potom je generovaná výnimka.

Príklad: *Konverzia textu na číslo*

Príklad demonštruje spôsob používania metódy Parse() pri načítavaní číselných hodnôt zadávaných v konzole z klávesnice, ich súčet a výpis výsledku na obrazovku.

```
using System;
namespace KonverziaTextNaCislo
{
    class Program
    {
        static void Main(string[] args)
        {
            int Cislo1;
            float Cislo2, Vysledok;
            Console.WriteLine("Sčítanie dvoch čísel");
            Console.Write("Zadaj prvé číslo: ");
            Cislo1 = Int32.Parse(Console.ReadLine());
            Console.Write("Zadaj druhé číslo: ");
            Cislo2 = Single.Parse(Console.ReadLine());
            Vysledok = Cislo1 + Cislo2;
            Console.WriteLine("Výsledok {0} + {1} = {2}", Cislo1,
                               Cislo2, Vysledok);
        }
    }
}
```



```
C:\WINDOWS\system32\cmd.exe
Sčítanie dvoch čísel
Zadaj prvé číslo: 7
Zadaj druhé číslo: 2,89
Výsledok 7 + 2,89 = 9,89
Press any key to continue . . .
```

Obr. 14 Parsovanie reťazcov – výsledok v konzole

Na konvertovanie je možné použiť triedu `Convert`, ktorá taktiež umožňuje konverziu na požadovaný dátový typ.

Formát je nasledovný:

- `Convert.ToDateTime(hodnota)`
- `Convert.ToDateTime(hodnota, IFormatProvider)`

kde:

`ToDateTime` – dátový typ, do ktorého sa vykoná konverzia,

`hodnota` – hodnota, ktorá sa bude konvertovať,

`IFormatProvider` – formátovací predpis.

Napr.:

```
int Cislo1 = Convert.ToInt32(Console.ReadLine());
Console.WriteLine(Convert.ToString(Vysledok));
```

IFormatProvider

Formátovací predpis `IFormatProvider` sa používa na používateľské prispôsobenie formátu (tvaru) vypisovaných číselných údajov alebo definovanie formátu načítavaných číselných údajov z textového vstupu.

Formátovací predpis môže používať nasledujúce parametre:

- D – decimal – platí pre celočíselné dátové typy,
- C – currency – zobrazenie meny,
- N – number – zobrazenie s oddeľovačom tisícok,
- E – exponential – zobrazenie v exponenciálnom tvare,
- F – fixed point – zobrazenie čísla s desatinným oddeľovačom,
- G – general – vyberie kratší z formátov E (exponential) alebo F (fixed point),
- X – hexadecimal – hexadecimálna reprezentácia čísla (platí iba pre celé čísla),

- 0 – zobrazenie čísla alebo nuly na danej pozícii,
- # – zobrazenie iba platných číslíc na danej pozícii,
- . – oddeľovač desatinnej časti,
- , – oddeľovač tisícok,
- e0 – definovanie exponenciálnej časti,
- 'text' – zadaný reťazec bude zobrazený ako text.

Znak, ktorý bude použitý ako desatinný oddeľovač, oddeľovač tisícok alebo symbol meny, je možné ľubovoľne definovať.

Príklad: *Formátovanie výstupu*

Formát výstupu je zostavený v množinových zátvorkách ako kombinácia poradového čísla vypisovanej hodnoty, dvojbodky a formátovacieho predpisu.

```
using System;
using System.Globalization;
namespace Formatovanie
{
    class Program
    {
        static void Main(string[] args)
        {
            int iCislo = 1234567;
            double fPi = 3.1415;
            Console.WriteLine("{0:D}", iCislo);
            Console.WriteLine("{0:0000000000}", iCislo);
            Console.WriteLine("{0:C}", iCislo);

            string fCenaCZ = string.Format(new CultureInfo("cs-CZ"),
                                           "{0:C}", 12.3);
            Console.WriteLine(fCenaCZ);
            Console.WriteLine("{0:00.00}", fPi);
            Console.WriteLine("{0:#0.00}", fPi);
            Console.WriteLine("{0:'Hodnota PI je '#0.00}", fPi);
        }
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
Formát 0:D -> 1234567
Formát 0:000000000 -> 001234567
Formát 0:C -> 1 234 567,00 Sk

Uýpis ceny s novou menou -> 12,30 Kč

Formát 0:00.00 -> 03,14
Formát 0:#0.00 -> 3,14
Formát s textom -> Hodnota PI je 3,14
```

Obr. 15 Formátovanie výstupu – výsledok v konzole

4 RIADIACE ŠTRUKTÚRY

Riadiace štruktúry, ako už názov napovedá, riadia priebeh vykonávania príkazov v programe. Majú definovanú syntax, ktorej štruktúru je potrebné dodržiavať. Medzi základné riadiace štruktúry patrí:

- **sekvencia** – postupnosť vykonávania príkazov, ktoré sa vykonajú v poradí, v akom sú napísané, ak explicitne nie je stanovené iné poradie,
- **vetvenie** – výber jedného alebo viacerých príkazov podľa toho, či je, alebo nie je splnená nejaká podmienka,
- **opakovanie (cyklus)** – opakovanie jedného alebo viacerých príkazov dovtedy, pokiaľ je splnená nejaká podmienka.

Vhodnou kombináciou riadiacich štruktúr, príkazov a operátorov možno zostaviť algoritmus riešenia definovaného problému a implementovať ho do formy programu.

4.1 OPERÁTORY

Operátory vyjadrujú symbolický zápis určitej operácie aplikovanej na jeden operand alebo častejšie medzi dvoma operandami. Operandom je premenná alebo objekt, s ktorým danú operáciu vykonávame. V nasledujúcej tabuľke môžete vidieť operátory, ktoré budeme používať pri programovaní v C#.

OPERÁTORY A ICH PRIORITA

Tabuľka

Kategória	Operácie
primárne	x.y f(x) a[x] x++ x-- new typeof checked unchecked
unárne	+ - ! ~ ++x --x
násobenie, delenie, modulo	* / %
sčítanie, odčítanie	+ -
bitový posun	<< >>
relačné a testovacie	< > <= >= is as
rovnosť, nerovnosť	== !=
logické AND	&
logické OR	
logické XOR	^
podmienkové AND	&&

podmienkové OR	
podmienka	? :
priradenie	=
kombinované priradenie	*= /= %= += -= <<= >>= &= ^= =

4.1.1 Aritmetické operátory

Používajú sa na bežné aritmetické výpočty s celými a reálnymi číslami. Patrí sem sčítanie, odčítanie, násobenie, delenie a modulo. Pri ich používaní platia rovnaké pravidlá ako v C++, a to vrátane tzv. celočíselného delenia. Ak oba operandy delenia sú celočíselné, tak aj výsledok bude celočíselný. Na zistenie zvyšku po takomto delení používame modulo.

Operátor sčítanie možno použiť napr. aj na spájanie reťazcov.

```
int A = 3, B = 4;
double C = 2.0, D = 4.0;
int iSucet = A + B;           // výsledok = 7
int iRozdiel = A - B;         // výsledok = -1
int iSucin = A * B;           // výsledok = 12
int iPodiel = B / A;          // výsledok = 1
int iZvysok = B / A;          // výsledok = 1
double dPodiel = C / D;       // výsledok = 0.5
string sMeno = "Jan" + "Novak"; // výsledok = "JanNovak"
```

4.1.2 Inkrement a dekrement

Tieto operátory sa používajú na zvyšovanie alebo znižovanie hodnoty premennej o 1. Najčastejšie sa s nimi stretneme v cykloch. Rozlišujeme predponovú a príponovú inkrementáciu a dekrementáciu (prefix a postfix). Rozdiel medzi nimi sa prejaví, ak sa použijú v kombinácii s ďalšími operátormi. Predponová verzia sa aplikuje pred inými operátormi a príponová až po aplikovaní ostatných operátorov, čiže na koniec.

```
int X = 0;
X++;           // výsledok = 1
++X;          // výsledok = 2
--X;          // výsledok = 1
X--;          // výsledok = 0
```

4.1.3 Priradenie a kombinované priradenie

Operátor priradenia je jediným operátorom, ktorý sa spracúva sprava doľava, tzn. že hodnota z pravej strany operátora sa ukladá do premennej na ľavej strane operátora. To isté platí aj pri kombinovanom priradení.

Kombinované priradenie skracuje zápis bežných aritmetických a niektorých ďalších operácií v prípade že sa touto operáciou má zmeniť hodnota premennej na ľavej strane od operátora priradenia, napr.

```
int X = 0, Y = 2;  
X = X + Y;    // k premennej X sa pripočíta Y a výsledok sa uloží do X  
X += Y;       // k premennej X sa pripočíta Y
```

4.1.4 Porovnávacie operátory

Pri vzájomnom porovnávaní dvoch hodnôt používame operátory väčší ako (>), menší ako (<), väčší alebo rovný (>=), menší alebo rovný (<=), rovnosť (==) a nerovnosť (!=). Všetky tieto operátory sú vyhodnocované zľava doprava, napr. či je hodnota vľavo väčšia ako hodnota vpravo. Najčastejšie sa používajú na porovnávanie čísel, ale operátory rovnosť a nerovnosť možno použiť aj na porovnanie reťazcov.

Výsledkom týchto operátorov je logický stav pravda (true) alebo nepravda (false).

Častou chybou býva používanie operátora = na zistenie rovnosti. Operátor = je priradovací operátor, správne treba používať operátor ==, teda „dvojité rovná sa“.

```
int X = 5, Y = 3;  
X < Y           // nepravda  
Y >= 3          // pravda  
X == 5          // pravda  
X != Y          // pravda
```

4.1.5 Logické operátory

Logické operátory sa používajú v prípade súčasného overovania pravdivosti alebo nepravdivosti viacerých podmienok, resp. ich kombinácií. Na vykonanie nejakého príkazu niekedy potrebujeme, aby boli splnené všetky podmienky – vtedy použijeme operátor logického súčinu (AND, a súčasne), inokedy nám stačí splniť len jednu podmienku z niekoľkých – vtedy použijeme operátor logického súčtu (OR, alebo).

Programovací jazyk C# rozlišuje dva druhy logického súčinu a dva druhy logického súčtu. Bežne sa používajú operátory `&&` a `||`. Oba tieto operátory používajú systém skráteného vyhodnocovania. Ak sa vyhodnotením prvého výrazu (vľavo) dosiahne neplatnosť, resp. platnosť celej podmienky, tak sa druhý výraz (vpravo) už nevyhodnocuje. V prípade `&&` teda stačí ak prvý výraz neplatí, to spôsobí neplatnosť celej podmienky. V prípade `||` zase stačí, ak prvý výraz platí, to spôsobí platnosť celej podmienky. V oboch spomínaných prípadoch sa už druhý výraz nebude vyhodnocovať. Operátory `&` a `|` na rozdiel od predchádzajúcich vyhodnotia vždy všetky výrazy v zloženej podmienke.

```
int X = 4, Y = 0;
Y != 0 && X/Y > 3    // prvý výraz neplatí, druhý sa nevyhodnotí
X < 3 || Y == 0      // prvý výraz neplatí, druhý spĺňa podmienku
X > 1 ^ Y == 0       // celá podm. neplatí, lebo oba výrazy platia
```

Špeciálnym operátorom je exkluzívny logický súčet (XOR, exkluzívne alebo), ktorého výsledkom je pravda iba vtedy, keď je pravdivý len jeden z výrazov, a teda nesmú byť pravdivé oba naraz.

4.2 PRÍKAZY SKOKOV

Príkazy skokov slúžia na skokovú zmenu sekvenčného vykonávania jednotlivých príkazov. Podľa spôsobu, akým program preruší alebo ukončí práve realizované činnosti a pokračuje na inom mieste, rozlišujeme:

- **break** – program prejde bezprostredne za cyklus, resp. za príkaz `switch`:
 - ukončí cyklus a pokračuje za telom cyklu,
 - ukončí vetvu `switch` a pokračuje za telom `switch`,
- **continue** – ukončí práve sa vykonávajúci jeden priebeh cyklu, príkazy po koniec tela cyklu sa ignorujú a program prejde na vyhodnotenie podmienky cyklu,
- **return** – ukončí metódu so zadanou návratovou hodnotou a riadenie programu sa odovzdá volajúcej metóde,
- **throw** – slúži na vyvolanie výnimky,
- **goto** – skočí na zadané návestie, resp. v príkaze `switch` skáče na zadanú hodnotu:
 - `goto navestie;`
 - `goto default;`
 - `goto case hodnota;`
 - nie je možné skákať z jednej funkcie do druhej,

- nie je možné skákať do vnútra cyklov,
- z cyklu je povolené vyskočiť,
- za každým cieľom skoku `goto` musí byť príkaz.

4.3 PRÍKAZY VETVENÍ

Príkazy vetvenia umožňujú programátorovi riadiť spracovanie programu. Vetvenie sa vykonáva na základe vyhodnotenia podmienky alebo hodnoty premennej.

Vetvenie programu – IF

Vetvenie `if` používame vtedy, keď potrebujeme vykonať nejaký príkaz len pri splnení určitej podmienky. Súčasťou vetvenia môže byť aj alternatívna vetva `else`, ktorá sa vykoná len vtedy, ak podmienka nebola splnená.

Ak platí podmienka uvedená v oboj zátvorke, tak sa vykoná `prikaz1`, inak sa vykoná `prikaz2`. Podmienkou môže byť ľubovoľný výraz, ktorý vráti hodnotu. Za podmienkou nesmie byť bodkočiarka! Vetva `else` je nepovinná.

```
if (podmienka)
    prikaz1;
else
    prikaz2;
```

Ak sa má pri splnení podmienky spracovať viac ako jeden príkaz, tak je potrebné vytvoriť blok príkazov. Rovnako to platí aj pre vetvu `else`.

```
if (podmienka)
{
    prikaz1;
    ...
    prikazN;
}
else
{
    prikaz2;
    ...
    prikazM;
}
```

Vetvenie programu – SWITCH

Viacnásobné vetvenie programu na základe hodnoty premennej uvedenej v zátvorke. Premenná musí byť celočíselného typu. Každá prípustná hodnota premennej musí mať vlastnú vetvu **case**. Hodnoty, ktoré nevyhovujú žiadnemu **case**, je možné spracovať v časti **default**. Spracovanie príslušnej vetvy musí byť ukončené príkazom **break**.

```
switch (premenna)
{
    case hodnota1: prikaz1; break;
    ...
    case hodnotaN: prikazN; break;
    default: prikazDefault; break;
}
```

Vetvenie programu – kaskáda vetvení IF

Nevýhodou vetvenia **switch** je nutnosť vytvoriť vetvu **case** pre všetky možné hodnoty, ktoré môže nadobúdať kontrolovaná premenná. Nie je možné definovať vetvu **case** s rozsahom hodnôt, príp. s reálnymi hodnotami. Tento problém sa rieši sériou príkazov **if..else**, čím sa vytvorí kaskáda vetvení, pri ktorej sa postupne prejdú všetky podmienky, kým sa nenájde vyhovujúca, príp. sa nedôjde až k záverečnej časti **else**.

Príklad: Kaskáda vetvení if

Na základe hodnoty počtu bodov sa určí známka z písomky. Hodnoty sú v podmienkach zapísané tak, aby podmienky vyhovovali v celom rozsahu možných hodnôt počtu bodov.

```
if (body >= 94)
    znamka = "A";
else if (body >= 86 && body < 94)
    znamka = "B";
else if (body >= 73 && body < 86)
    znamka = "C";
else if (body >= 62 && body < 73)
    znamka = "D";
else if (body >= 50 && body < 62)
    znamka = "E";
else
    znamka = "FX";
```

4.4 PRÍKAZY CYKLOV

Cyklus slúži na opakovanie určitej skupiny príkazov, kým platí ukončovacia podmienka. V tele cyklu musí nastať stav, ktorý spôsobí neplatnosť podmienky, a tým ukončenie cyklu. Špeciálnym prípadom sú „nekonečné“ cykly. Nekonečné cykly majú podmienku nastavenú vždy ako pravdivú (true). Ukončenie nekonečného cyklu sa dosiahne napr. príkazom `break` po splnení určenej podmienky v tele cyklu.

Cyklus FOR

Tento cyklus obsahuje tri výrazy s ľubovoľnými príkazmi, ktoré môžu byť aj vynechané. Nemusí prebehnúť ani raz.

```
for (inicializacia; podmienka; inkrement)
{
    TeloCyklu;
}
```

Inicializácia: Vykoná sa iba raz pred spustením cyklu. Slúži väčšinou na inicializáciu počítadla.

Podmienka: Ukončovacia podmienka musí platiť, aby sa vykonalo telo cyklu. Kontroluje sa pred každým vykonávaním tela cyklu.

Inkrement: Vykonáva sa po každom priebehu tela cyklu. Väčšinou sa používa na zvyšovanie hodnoty počítadla.

Proces vykonávania jednotlivých častí cyklu `for` možno zapísať nasledovne: *inicializácia*, *podmienka* (platí), *TeloCyklu*, *inkrement*, *podmienka* (platí), *TeloCyklu*, *inkrement*, ..., *podmienka* (platí), *TeloCyklu*, *inkrement*, *podmienka* (neplatí), koniec cyklu.

Cyklus WHILE

Cyklus s podmienkou na začiatku. Vykoná sa, ak je podmienka pravdivá. Nemusí prebehnúť ani raz.

```
while (podmienka)
{
    TeloCyklu;
}
```

Cyklus DO..WHILE

Cyklus s podmienkou na konci. Opakuje sa, ak je podmienka pravdivá. Telo cyklu sa vykoná aspoň jedenkrát aj pri neplatnej podmienke.

```
do
{
    TeloCyklu;
} while (podmienka);
```

Príklad: *Výpis čísel od 1 po 10 pomocou cyklov*

V uvedenom príklade sú použité tri cykly **for**, **while** a **do-while**, kde pri každom type cyklu sú na začiatku nastavené počiatočné podmienky a v tele cyklov je vykonaný výpis a súčasne aj zmena premennej cyklu.

```
using System;
namespace Cykly
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Vypis pomocou cyklu FOR");
            for (int i = 1; i <= 10; i++)
                Console.Write("{0,3}", i);

            Console.WriteLine("\nVypis pomocou cyklu WHILE");
            int j = 1;
            while (j <= 10)
            {
                Console.Write("{0,3}", j);
                j++;
            }

            Console.WriteLine("\nVypis pomocou cyklu DO-WHILE");
            int k = 1;
```

```

do
{
    Console.Write("{0,3}", k);
    k++;
}
while (k <= 10);
}
}
}

```

Cyklus FOREACH

Používa sa pri prechádzaní hodnôt uložených v objekte, ktorý implementuje rozhranie **IEnumerable**. Sú to hlavne polia, enumerátory, objekty dátových štruktúr a pod. Pri spracovaní cyklu nadobúda premenná **prem** postupne hodnoty objektu označeného **obj**. Počet opakovaní zodpovedá počtu hodnôt objektu.

```

foreach (DatTyp prem in obj)
{
    TeloCyklu;
}

```

Enumerácia a cyklus foreach

V nasledujúcom príklade je definovaná enumerácia s názvom **Tyzden**, ktorá obsahuje pomenované konštanty s názvami dní. Pomocou metódy **Enum.GetNames()** získame zoznam dní v textovej forme, ktorý potom celý prechádzame pomocou cyklu **foreach** a každý záznam vypíšeme na nový riadok.

Príklad: Cyklus foreach

```

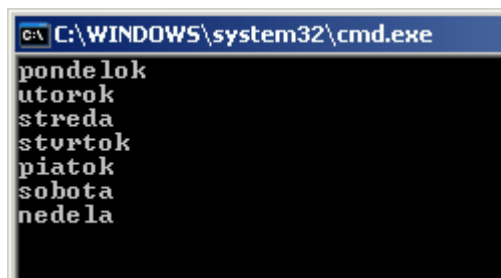
using System;
namespace Enumerata
{
    class Program
    {
        public enum Tyzden
        { pondelok, utorok, streda, stvrtok, piatok, sobota, nedela }
    }
}

```

```

static void Main(string[] args)
{
    foreach (string Den in Enum.GetNames(typeof(Tyzden)))
    {
        Console.WriteLine(Den);
    }
}
}

```



Obr. 16 Cyklus foreach – výsledok v konzole

4.5 POMOCNÉ RIADIACE ŠTRUKTÚRY

Using

Umožňuje deklaráciu konkrétneho menného priestoru (namespace), z ktorého je potom možné používať jeho súčasti bez uvádzania plne kvantifikovaného mena.

```
using namespace;
```

Umožňuje vytváranie aliasov existujúcich menných priestorov (namespace).

```
using meno = namespace;
```

Regióny

Umožňujú prehľadnejšie zobrazovanie zdrojového kódu v editore a nemajú vplyv na funkčnosť programu. Jednotlivé regióny možno „uzatvárať“, vnárať a tiež pomenovať.

```

#region <meno regionu>
.....
#endregion

```

Komentáre

Umožňujú vkladať ľubovoľný text, ktorý kompilátor ignoruje. Komentáre by mali byť zmysluplné a týkať sa zdrojového kódu, v ktorom sa nachádzajú. Používajú sa najmä na vysvetlenie zložitejších častí algoritmu, opis účelu jednotlivých metód tried a ich požadovaných vstupoch a poskytovaných výstupoch. Rozhodne je nevhodné komentovať každý riadok programu. Komentáre treba udržiavať aktuálne, a teda so zmenami v kóde upravovať aj príslušné komentáre. Možno nimi tiež znefunkčniť časť kódu, ale vo finále by sa takéto zakomentované príkazy v kóde nemali vyskytovať.

Rozoznávame nasledovné typy komentárov:

- jednoriadkový komentár, komentár do konca riadku

```
//jedenriadkovy komentar
```

- viacriadkový komentár

```
/* tento dvojriadkovy text bude  
povazovany za komentar */
```

- dokumentačný komentár

```
/// <summary>  
/// dokumentacny komentar v programe  
/// </summary>
```

5 TRIEDY A OBJEKTY V C#

V bežnom živote používame pojmy, pod ktorými si každý vie predstaviť konkrétny predmet, osobu, zviera, vec, príp. ich skupinu s rovnakými či podobnými vlastnosťami a parametrami. Vieme, že sa používajú na nejaké činnosti, resp. nejaké činnosti samy dokážu vykonávať. Činnosť, ktorou tieto skupiny so spoločnými vlastnosťami definujeme, sa nazýva kategorizácia. Ak povieme auto, tak myslíme dopravný prostriedok, ktorý má najčastejšie štyri kolesá, je poháňaný motorom na určitý druh paliva, dokáže odviezť určitý počet osôb a nejaký náklad. Auto vie ísť dopredu, dozadu, zatáča, zrýchľuje, brzdí, atď. Súhrn týchto vlastností a funkcionalít charakterizuje auto a každý vie, čo to znamená.

Pri definovaní tried vychádzame z podobného princípu ako pri kategorizovaní už spomínaných vecí. Zapuzdrujeme atribúty (vlastnosti) a metódy (činnosti), ktoré spolu úzko súvisia do jedného samostatného celku. Zapuzdrenie vyjadruje ešte jeden dôležitý princíp objektovo orientovaného programovania a to, že na používanie triedy nemusí programátor vedieť, ako táto trieda interne funguje. V prípade auta viete, že na zrýchlenie musíte pridať plyn, ale nemusíte vedieť, ako presne celý proces od stlačenia pedálu až po skutočné zvýšenie rýchlosti prebieha. Podobne je to aj s programovaním. Viete, že na výpis do konzoly môžete použiť metódu `Console.WriteLine()`, ale ako sa váš text spracuje a objaví na obrazovke, vás už nemusí zaujímať. Vnútorne dáta a činnosti triedy sú pred používateľom triedy skryté, preto sa zapuzdrowanie označuje tiež ako skrývanie informácií.

5.1 OOP V C# – ZÁKLADNÉ ROZDIELY OPROTI OOP V C++

Implementácia objektovo orientovaného programovania v programovacom jazyku C#, resp. v programovacích jazykoch pracujúcich na platforme .NET Framework, má niektoré špecifiká, ktoré ho odlišujú od implementácie v programovacom jazyku C++.

- všetky triedy v jazyku C# majú spoločného predka – systémovú triedu `Object`,
- vytvárajú sa iba dynamické inštancie tried, ktoré spravuje automatická správa pamäte (Garbage Collector),
- v jazyku C# sa štandardne nepoužívajú deštruktory pri zrušení inštancie triedy. O jej zrušenie a uvoľnenie alokovaných prostriedkov sa opäť postará automatická správa pamäte. Ak ale programátor potrebuje použiť deštruktor, tak tá možnosť tu zostáva zachovaná.
- triedy môžu implementovať rozhrania,

- triedy môžu obsahovať metódy, dátové položky, konštruktory, deštruktory, vlastnosti, udalosti, preťažené operátory a vnorené typy,
- jazyk C# nepodporuje viacnásobnú dedičnosť (súčasné dedenie z viacerých tried), ale umožňuje dedenie z jednej triedy a viacerých rozhraní súčasne,
- jazyk C# nepodporuje virtuálnu dedičnosť,
- je možné zakázať dedenie (zapečatiť triedu).

5.2 TRIEDA V C#

Trieda sa definuje kľúčovým slovom `class`, za ktorým nasleduje názov triedy. Nepovinný údaj `modprist` určuje prístupnosť triedy, prípadne jej špeciálne vlastnosti (abstraktná trieda, zapečatená, ...).

```
<modprist> class MenoTriedy
{
    členské premenné, metódy, typy...
}
```

5.3 PRÍSTUPOVÉ PRÁVA

Prístupové práva vymedzujú možnosti prístupu k prvkom triedy predovšetkým z miest kódu mimo funkcií danej triedy. Rozlišujeme:

- `private` – prístup majú iba metódy implementované v danej triede,
- `protected` – prístup majú iba metódy implementované v danej triede a jej potomkoch,
- `internal` – prístup zo zoskupenia, v ktorom je trieda definovaná,
- `protected internal` – prístup zo zoskupenia, v ktorom je trieda definovaná, a taktiež zo všetkých potomkov triedy bez ohľadu na zoskupenie, v ktorom sú definované,
- `public` – je možné pristupovať z ktorejkoľvek časti programu, zoskupenia (assembly) a ľubovoľnej triedy.

Nasledujúca tabuľka znázorňuje kombinácie prístupových práv a možnosť prístupu z rôznych miest kódu.

	trieda	potomkovia	zoskupenie	ostatné
private	x			
protected	x	x		
internal	x		x	
protected internal	x	x	x	
public	x	x	x	x

Prístupové práva v jazyku C# sa nevzťahujú iba na samotnú triedu, ale je možné definovať prístupové práva samostatne pre jednotlivé časti triedy (premenné, metódy, ...).

5.4 ČLENSKÉ PREMENNÉ A METÓDY TRIEDY

Deklarácia a definícia členskej metódy musí byť vždy zapísaná v tele triedy. V triede nie je možné uviesť iba prototyp metódy a definíciu dať mimo triedy, ako je to možné v C++. V C# tiež nie je ekvivalent C++ špecifikácie [inline](#).

Metódy je možné preťažovať aj na základe rozdielu medzi hodnotovými, vstupnými a výstupnými parametrami. Parametrom nie je možné predpísať implicitné hodnoty. Pri volaní metódy je možné argumenty odovzdávať hodnotou, odkazom, a taktiež pomocou výstupných parametrov.

Členské premenné

Členská premenná je vnútorne zapuzdrená premenná, ktorá je priamo prístupná všetkým členským metódam triedy. Mimo triedy je jej prístupnosť určená modifikátorom prístupu. Zvyčajne bývajú členské premenné neprístupné mimo triedy, okrem jej priamych potomkov. K premennej prístupujeme v rámci triedy (z členskej metódy) priamo pomocou názvu premennej:

```
nazovPremennej;
```

Z programu mimo triedy, teda z inštancie triedy prístupujeme k premennej pomocou operátora `.` (bodka):

```
nazovInstancie.nazovPremennej;
```

Statické členské premenné

Členská premenná môže byť označená modifikátorom `static`, vtedy sa stáva statickou premennou, resp. premennou triedy. Statická premenná je prístupná bez nutnosti vytvoriť inštanciu. Používa sa na využívanie dát medzi viacerými inštanciami jednej triedy. Prístup k statickej premennej:

```
MenoTriedy.nazovPremennej;
```

Statické metódy

Statické metódy sú definované modifikátorom `static` a predstavujú tzv. metódy triedy. Implementujú operácie s triedou ako celkom a možno ich využívať, aj keď neexistuje žiadna inštancia z danej triedy. V statickej metóde nie je možné využívať nestatické dátové zložky bez zadania mena konkrétnej inštancie. Volanie statickej metódy:

```
MenoTriedy.MenoMetody();
```

Nestatické metódy

Nestatické metódy sú všetky metódy triedy, ktoré nemajú definovaný modifikátor `static`. Implementujú operácie s jednotlivými inštanciami triedy, preto ich voláme vždy pre konkrétne inštancie. Volanie nestatickej metódy:

```
MenoInstancie.MenoMetody();
```

Možno v nich použiť kľúčové slovo `this` predstavujúce odkaz na aktuálnu inštanciu:

```
this.MenoMetody();
```

5.5 TRIEDA A INŠTANCIA V JAZYKU C#

Trieda a inštancia vystupujú v podobnom význame ako dátový typ a premenná, a teda definícia triedy je opisom dátového typu a inštancia je premenná typu trieda.

Príklad: Základná definícia triedy

Trieda `Student` obsahujúca len tri členské premenné a funkcia `Main()`, v ktorej sú vytvorené tri inštancie tejto triedy pomocou operátora `new`.

Uvedená trieda `Student` bude v nasledujúcich príkladoch rôznym spôsobom upravovaná a dopĺňaná pre potreby názorného predstavenia rôznych prvkov a vlastností objektovo

orientovaného programovania v jazyku C#. V niektorých príkladoch sú uvádzané funkcie skrátene do zbalenej formy, príp. sú vo výpise triedy úplne vynechané. Pristúpili sme k tomu z dôvodu úspory miesta a aby sa v každom príklade neopakovali dlhé časti kódu, ktoré boli vysvetlené v predchádzajúcich zdrojových kódach. Plný výpis takýchto funkcií vždy nájdete v niektorom z predošlých, príp. nasledujúcich príkladov.

```
class Student
{
    string sMeno;
    int    iKruzok;
    string sZnamka;
} // definícia triedy

class Program
{
    static void Main(string[] args)
    {
        Student Roman = new Student(); // inštancia
        Student Ivana = new Student(); // inštancia
        Student Jozef = new Student(); // inštancia
    }
}
```

Príklad: Definícia triedy a vytvorenie inštancie

Trieda `Student` obsahuje len tri členské premenné a dve členské metódy. Vo funkcii `Main()` je vytvorená jedna inštancia tejto triedy a z nej sú volané obe nestatické členské metódy.

```
class Student
{
    string sMeno; // členske premenne
    int    iKruzok;
    string sZnamka;

    public void prcZadajStudent(){...} // členske metódy
    public void prcVypisStudent(){...}
}
```

```

class Program
{
    static void Main(string[] args)
    {
        Student Std01 = new Student();
        Std01.prcZadajStudent();
        Std01.prcVypisStudent();
    }
}

```

Príklad: Deklarácia triedy s jej metódami

Trieda `Student` s tromi členskými premennými a dvoma členskými metódami vrátane výpisu celého tela funkcií. Metóda `prcZadajStudent()` načíta údaje z konzoly a uloží ich do členských premenných, metóda `prcVypisStudent()` vypíše do konzoly informácie uložené v členských premenných.

```

class Student
{
    string sMeno;
    int    iKruzok;
    string sZnamka;

    public void prcZadajStudent()
    {
        Console.Write("Zadaj meno studenta: ");
        sMeno = Console.ReadLine();
        Console.Write("Zadaj cislo kruzku studenta: ");
        iKruzok = Int32.Parse(Console.ReadLine());
        Console.Write("Zadaj znamku studenta: ");
        sZnamka = Console.ReadLine();
    }

    public void prcVypisStudent()
    {
        Console.WriteLine("Student {0} z kruzku {1} ma
                           znamku {2}.", sMeno, iKruzok, sZnamka);
    }
}

```

```
C:\WINDOWS\system32\cmd.exe
Zadaj meno studenta: Roman
Zadaj cislo kruzku studenta: 12
Zadaj znamku studenta: A
Student Roman je z 12. kruzku a ma znamku A.
Press any key to continue . . .
```

Obr. 17 Výsledok volania metód triedy v konzole

5.6 KONŠTRUKTOR TRIEDY

Konštruktor triedy je špeciálnou metódou, ktorá má rovnaký názov ako trieda a nemá návratovú hodnotu. Nepíše sa ani `void`! Konštruktor je volaný explicitne príkazom `new` pri vzniku inštancie a používa sa hlavne na inicializovanie členských premenných.

Príklad: Trieda s bezparametrickým konštruktorom

Konštruktor triedy `Student` nastavuje počiatočné hodnoty členských premenných. Vo funkcii `Main()` je vytvorená inštancia a následne je zavolaná metóda na výpis hodnôt.

```
class Student
{
    string sMeno;
    int    iKruzok;
    string sZnamka;

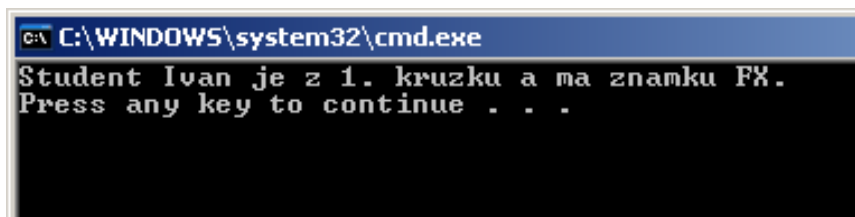
    public Student() //konstruktor bez parametrov
    {
        sMeno    = "Ivan";
        iKruzok  = 1;
        sZnamka  = "FX";
    }

    public void prcZadajStudent(){...}
    public void prcVypisStudent(){...}
}
```

```

class Program
{
    static void Main(string[] args)
    {
        Student Std01 = new Student();
        //Std01.prcZadajStudent();
        Std01.prcVypisStudent();
    }
}

```



Obr. 18 Výsledok volania metódy výpisu v konzole

Konštruktor s parametrami

Parametrický konštruktor môže obsahovať jeden alebo viac parametrov, využíva sa na inicializáciu objektu hodnotami odovzdanými pri vytváraní objektu. Trieda môže obsahovať viac parametrických konštruktorov (preťažené konštruktory).

Príklad: Trieda s parametrickým konštruktorom

Trieda `Student` má definovaný bezparametrický aj parametrický konštruktor. V parametrickom konštruktore sú hodnotami parametrov inicializované členské premenné. Konkrétne hodnoty sú odovzdané konštruktoru pri jeho volaní počas vytvárania inštancie triedy `Student` vo funkcii `Main()`.

```

class Student
{
    string sMeno;
    int    iKruzok;
    string sZnamka;

    public Student(){...} //konštruktor bez parametrov
}

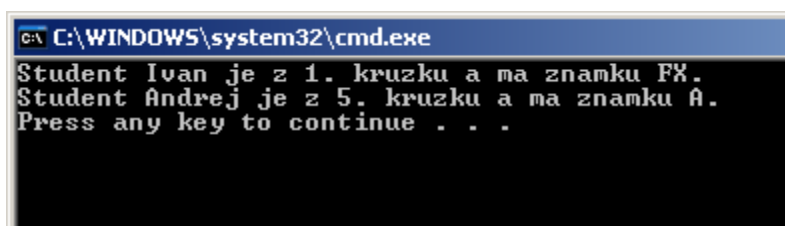
```

```

//konštruktor s parametrami
public Student(string ksMeno, int kiKruzok, string ksZnamka)
{
    sMeno    = ksMeno;
    iKruzok  = kiKruzok;
    sZnamka  = ksZnamka;
}
public void prcZadajStudent(){...}
public void prcVypisStudent(){...}
}
class Program
{
    static void Main(string[] args)
    {
        Student Std01 = new Student();
        //Std01.prcZadajStudent();
        Std01.prcVypisStudent();

        Student Std02 = new Student("Andrej", 5, "A");
        Std02.prcVypisStudent();
    }
}

```



Obr. 19 Výsledok volania metódy výpisu v konzole

Statický konštruktor

Statický konštruktor nie je možné explicitne volať, ale je volaný automaticky pri zavedení triedy do pamäte. Možno v ňom inicializovať len statické dátové zložky s modifikátorom **readonly**. Statický konštruktor sa nededí.

Príklad: *Trieda so statickým konštruktorom*

Trieda `Student` je doplnená o statickú členskú premennú `sFakulta` určenú iba na čítanie a jej hodnota je inicializovaná v statickom konštruktoze.

```
class Student
{
    string sMeno;
    int    iKruzok;
    string sZnamka;
    static readonly string sFakulta; //staticka clenka premenna

    public Student(){...} //konstruktor bez parametrov
    //konstruktor s parametrami
    public Student(string ksMeno, int kiKruzok, string ksZnamka){...}
    static Student() //staticky konstruktor
    {
        sFakulta = "MTF STU Trnava";
    }

    public void prcZadajStudent(){...}
    public void prcVypisStudent(){...}
}
```

5.7 DEŠTRUKTOR TRIEDY

Deštruktor triedy je špeciálna metóda, ktorá má rovnaký názov ako trieda, ale pred menom má ~ (vlnovka). Pri deštruktoze sa neudáva modifikátor prístupu, nemá návratovú hodnotu ani parametre. Deštruktor nie je možné volať explicitne, je volaný automaticky pri zániku inštancie. Slúži hlavne na „upratanie“ po práci inštancie – uvoľnenie alokovaných systémových prostriedkov, ukončenie práce so súbormi, ukončenie spojenia s databázou a pod.

V C# (resp. v .NET Frameworku) sa o zrušenie nepoužívaných inštancií tried štandardne stará Garbage Collector (automatická správa pamäte), a tak nie je potrebné definovať deštruktor.

Príklad: *Trieda s deštruktorom*

Deštruktor pri zániku inštancie vypíše do konzoly informačný text. Vo funkcii `Main()` sú vytvorené dve inštancie triedy `Student`, následne sú vypísané údaje týchto inšancií a výpis informácie o ukončení programu. Môžete si všimnúť, že v kóde nie je žiaden príkaz na zmazanie týchto dvoch inšancií. O uvoľnenie pamäte sa postaral Garbage Collector, čoho dôkazom je výpis informácie z deštruktora triedy `Student`, ktorý bol spustený automaticky pri zániku oboch inšancií.

```
class Student
{
    string sMeno;
    int    iKruzok;
    string sZnamka;
    static readonly string sFakulta;

    public Student(){...}
    public Student(string ksMeno, int kiKruzok, string ksZnamka){...}
    static Student(){...}

    ~Student() //destruktor triedy
    {
        Console.WriteLine("Koniec existencie instancie!");
    }

    public void prcZadajStudent(){...}
    public void prcVypisStudent(){...}
}

class Program
{
    static void Main(string[] args)
    {
        Student Std01 = new Student();
        //Std01.prcZadajStudent();
        Std01.prcVypisStudent();
    }
}
```

```

Student Std02 = new Student("Andrej", 5, "A");
Std02.prcVypisStudent();

Console.WriteLine("\n\nKoniec programu.\n");
}
}

```

```

C:\WINDOWS\system32\cmd.exe
Student Ivan je z 1. kruzku a ma znamku FX.
Student Andrej je z 5. kruzku a ma znamku A.

Koniec programu.

Koniec existencie instance!
Koniec existencie instance!
Press any key to continue . . .

```

Obr. 20 Výsledok volania metódy výpisu v konzole a ukončenie programu

Príklad: Počítanie inštancií triedy pomocou statickej premennej

V triede `Obdlznik` je definovaná statická premenná `iPocetObdlznikov`, ktorá bude automaticky inkrementovaná pri každom vytvorení inštancie triedy `Obdlznik`. Takto budeme vždy vedieť, koľko inštancií bolo celkovo vytvorených. V závere príkladu je vidieť volanie statickej metódy priamo z triedy a nie z inštancie triedy.

```

class Obdlznik
{
    private static int iPocetObdlznikov = 0;
    private int iDlзка;
    private int iVyska;

    public Obdlznik(int a, int b)
    {
        iDlзка = a;
        iVyska = b;
        iPocetObdlznikov++;
    }
}

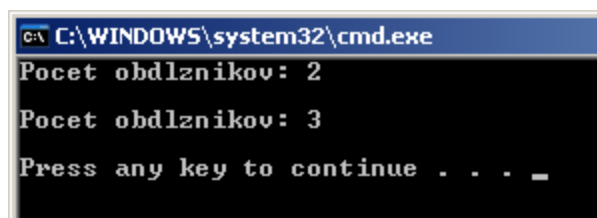
```

```

public static int PocetObdlznikov()
{
    return iPocetObdlznikov;
}
};

class Program
{
    static void Main(string[] args)
    {
        Obdlznik obd1 = new Obdlznik(2, 3);
        Obdlznik obd2 = new Obdlznik(1, 6);
        Console.WriteLine("Pocet obdlznikov: {0}\n",
                           Obdlznik.PocetObdlznikov());
        Obdlznik obd3 = new Obdlznik(4, 9);
        Console.WriteLine("Pocet obdlznikov: {0}\n",
                           Obdlznik.PocetObdlznikov());
    }
}

```



Obr. 21 Počítanie inštancií triedy

5.8 IMPLEMENTÁCIA DEDIČNOSTI V TRIEDE

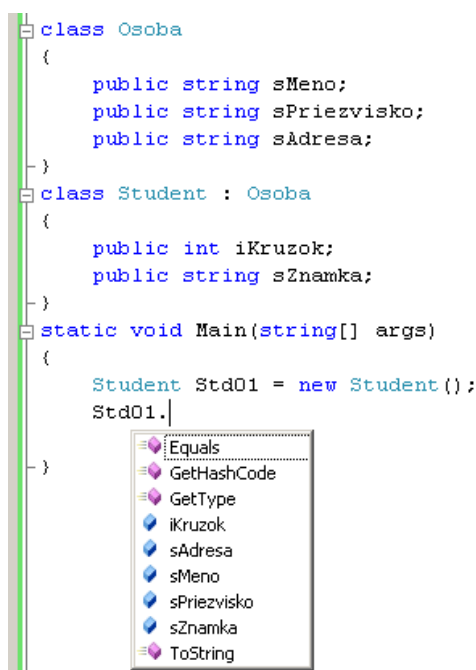
Jazyk C# podporuje iba jednoduché dedenie, preto nie je možné použiť dedenie z viacerých tried súčasne. Ak nie je explicitne uvedený predok triedy, systém automaticky doplní triedu **Object**. Pri dedení je možné využívať abstraktné triedy. Potomok triedy získa všetky vlastnosti predka, pričom potomkovi môžeme pridávať nové vlastnosti, prípadne môžeme nahradzovať pôvodné vlastnosti novými. Dedenie je možné zakázať tzv. zapečatením triedy.

Pri opise hierarchie tried v procese dedenia sa môžeme stretnúť s rôznymi vyjadreniami tohto vzťahu. Najpoužívanejšie sú tieto: základná trieda – odvodená trieda, rodič – potomok, predok – potomok.

Dedičnosť – deklarácia

Pri deklarácii odvodenej triedy uvedieme za názov odvodenej triedy dvojbodku a následne názov základnej triedy.

```
<modpristupu> class Potomok : Predok
{
    členské premenné, metódy, typy...
}
```



Obr. 22 Implementácia dedičnosti

5.8.1 Konštruktory a deštruktory pri dedičnosti

Pri inicializácii objektu z triedy vytvorenej dedením je automaticky spustený implicitný bezparametrický konštruktor predka a následne konštruktor potomka. Pri zrušení takéhoto objektu je volanie deštruktorov v opačnom poradí ako pri konštruktoroch – najskôr je spustený deštruktor potomka a po ňom deštruktor predka.

Príklad: *Dedičnosť – konštruktory a deštruktory triedy*

```
using System;
namespace CA_Dedicnost
{
    class Osoba
    {
```

```

protected string sMeno;
protected string sPriezvisko;
protected string sAdresa;

protected Osoba()
{
    sMeno      = "Nezname";
    sPriezvisko = "Nezadal";
    sAdresa     = "Neuviedol";
}
~Osoba()
{
    Console.WriteLine("\nDestruktor predka\n");
}
}

class Student : Osoba
{
    int iKruzok;
    string sZnamka;

    public Student()
    {
        sAdresa = "Trnava";
        iKruzok = 99;
        sZnamka = "FX";
    }
    ~Student()
    {
        Console.WriteLine("\nDestruktor potomka");
    }
    public void prcVypisStudent()
    {
        Console.WriteLine("Student {0} {1} z kruzku {2}, bytom {3}, ma
znamku {4}.", sMeno, sPriezvisko, iKruzok, sAdresa, sZnamka);
    }
}

```

```

class Program
{
    static void Main(string[] args)
    {
        Student std01 = new Student();
        std01.prcVypisStudent();
    }
}

```

```

C:\WINDOWS\system32\cmd.exe
Student Nezname Nezadal z kruzku 99, bytom Trnava, ma znamku FK.
Destruktor potomka
Destruktor predka
Press any key to continue . . .

```

Obr. 23 Dedičnosť – konštruktory a deštruktory triedy – výsledok v konzole

V tomto príklade je definovaná trieda **Osoba**, ktorá obsluhuje a uchováva zvolené atribúty osoby, ako je meno, priezvisko a adresa. V štandardnom konštruktore sú tieto atribúty inicializované konštantnými reťazcami. Trieda **Student** vznikne odvodením z triedy **Osoba**, pričom pridáva ďalšie atribúty, ktoré sú špecifické pre študenta. Konštruktor potomka inicializuje členské premenné vlastnej triedy (**Student**), ale môže zmeniť aj premenné triedy svojho predka (**Osoba**). Toto je možné aj z toho dôvodu, že pri vytvorení inštancie triedy **Student** bude najskôr spustený konštruktor predka a až následne konštruktor potomka, ktorý môže prepísať hodnoty nastavené konštruktorom predka.

Vo funkcii **Main()** je vytvorená inštancia triedy **Student**, vypíšu sa údaje tejto triedy a po skončení sa inštancia automaticky uvoľní z pamäte. Vo výpise z konzoly je vidno volanie deštruktorov v poradí: deštruktor triedy **Student** (potomok) a následne deštruktor triedy **Osoba** (predok).

5.8.2 Metódy triedy pri dedičnosti

Pri odvodzovaní tried často dochádza k situácii, keď v triede potomka sa nachádza rovnako pomenovaná metóda ako v triede predka. Vtedy hovoríme o prekryvaní metód.

V triede potomka môže byť prekrytá metóda predka využitím modifikátora **new**

```
new <modpristupu> typ menoMetody()
```

Ak chceme prekryť metódy predka, je potrebné v predkovi definovať metódu ako virtuálnu (modifikátorom **virtual**) a v potomkovi túto metódu definovať s modifikátorom **override**

```
<modpristupu> virtual typ menoMetody()
```

```
<modpristupu> override typ menoMetody()
```

Prekrytá metóda predka je v triede potomka aj napriek tomu stále prístupná a je ju možné volať aj z potomka. Aby ju bolo možné volať, nesmie mať prístupové práva **private**. Ak je potrebné volať v triede potomka prekrytú metódu predka, je to možné pomocou operátora **base**

```
base.menoMetodyPredka();
```

Možno využívať iba metódy bezprostredného predka nie vzdialenejších predkov.

Príklad: *Metódy triedy pri dedičnosti*

Trieda **Osoba** má definovanú funkciu **prcZadaj()**, pomocou ktorej sa načítajú dáta do členských premenných. Z nej odvodená trieda **Student** pridáva ďalšie atribúty, a tiež rovnako pomenovanú metódu **prcZadaj()**, ktorá načíta údaje pre členské dáta triedy **Student**. Táto metóda pokrýva pôvodnú zdedenú metódu. Na načítanie údajov zdedených atribútov sa využíva volanie pôvodnej metódy predka pomocou operátora **base**.

```
using System;
namespace CA_Dedicnost
{
    class Osoba
    {
        protected string sMeno;
        protected string sPriezvisko;
        protected string sAdresa;

        protected Osoba(){...}
        ~Osoba(){...}
        public void prcZadaj() //clenska metoda predka
        {
            Console.Write("Zadaj meno: ");
            sMeno = Console.ReadLine();
        }
    }
}
```



```

        Console.Write("Zadaj priezvisko: ");
        sPriezvisko = Console.ReadLine();
        Console.Write("Zadaj adresu: ");
        sAdresa = Console.ReadLine();
    }
}

class Student : Osoba
{
    int iKruzok;
    string sZnamka;

    public Student(){...}
    ~Student(){...}

    new public void prcZadaj() //clenska metoda potomka
    {
        base.prcZadaj(); //volanie clenskej metody predka
        Console.Write("Zadaj cislo kruzku studenta: ");
        iKruzok = Int32.Parse(Console.ReadLine());
        Console.Write("Zadaj znamku studenta: ");
        sZnamka = Console.ReadLine();
    }
    public void prcVypis()
    {
        Console.WriteLine("Student {0} {1} z kruzku {2}, bytom {3}, ma
        znamku {4}.", sMeno, sPriezvisko, iKruzok, sAdresa, sZnamka);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Student std01 = new Student();
        std01.prcZadaj();
    }
}

```

```

        std01.prcVypis();
    }
}
}

```

```

C:\WINDOWS\system32\cmd.exe
Zadaj meno: Jan
Zadaj priezvisko: Novak
Zadaj adresu: Trnava
Zadaj cislo kruzku studenta: 1
Zadaj znamku studenta: A
Student Jan Novak z kruzku 1, bytom Trnava, ma znamku A.
Destruktor potomka
Destruktor predka
Press any key to continue . . .

```

Obr. 24 Metódy triedy pri dedičnosti – výsledok v konzole

5.8.3 Rozdiely v dedičnosti v C++ a C#

Pri deklarácii dedenia sa neuvádza, na rozdiel od jazyka C++, typ dedenia. Typ dedenia je analogický ako pri dedení typu `public` v C++. Zdedené sú všetky zložky predka bez ohľadu na nastavené prístupové práva (nemusia byť v potomkovi prístupné). Konštruktory a deštruktory sa nededia.

5.9 ABSTRAKTNÁ A ZAPEČATENÁ TRIEDA

Abstraktná trieda – abstract class

Z abstraktnej triedy nie je možné vytvoriť objekt, slúži iba ako predok pri dedení. Iba abstraktná trieda môže obsahovať abstraktné metódy, pričom abstraktná metóda nesmie byť typu `private`.

```

<modprist> abstract class MenoTriedy
{
    členské premenné, metódy, typy...
    <modprist> abstract void MenoMetody();
}

```

Zapečatená trieda – sealed class

Od zapečatenej triedy nie je možné vytvoriť potomka, zapečatenú metódu nie je možné v potomkovi predefinovať. Abstraktné triedy a metódy nemôžu byť zapečatené.

```

<modprist> sealed class MenoTriedy
{
    členské premenné, metódy, typy...
}

```

5.10 ROZHRAINIA TRIED (INTERFACES)

Rozhranie je špeciálnym typom triedy. Obsahuje deklarácie metód, vlastností alebo udalostí, ale nie ich implementáciu. Rozhranie je možné využívať pri dedičnosti a na rozdiel od obyčajných tried, je možné dediť od viacerých rozhraní súčasne. Triedy, ktoré zdedia rozhranie, musia implementovať všetky jeho časti.

```

<modpristupu> interface IMenoRozhrania <: IPredkovia>
{
    deklarácia metód, vlastností, udalostí ...
}

```

Príklad: Definícia a použitie rozhrania

Rozhranie `IZadajVypisUdaje` deklaruje funkcie, ktoré bude musieť implementovať trieda, ktorá ho použije pri dedení, v tomto prípade trieda `Student`. Základná trieda `Osoba` obsahuje len atribúty. Metódy, ktoré budú používané na načítanie a výpis hodnôt týchto atribútov, budú implementované až v odvodenej triede, ktorá vznikne zo základnej triedy `Osoba` a rozhrania `IZadajVypisUdaj`.

```

interface IZadajVypisUdaje //rozhranie
{
    void prcZadaj();
    void prcVypis();
}
class Osoba
{
    protected string sMeno;
    protected string sPriezvisko;
    protected string sAdresa;
    protected Osoba(){...}
    ~Osoba(){...}
}

```

```

class Student : Osoba, IZadajVypisUdaje
{
    int    iKruzok;
    string sZnamka;
    public Student(){...}
    ~Student(){...}

    public void prcZadaj () {...} //implementovana metoda rozhrania
    public void prcVypis () {...} //implementovana metoda rozhrania
}

```

Preddefinované rozhrania

C# obsahuje niekoľko štandardizovaných rozhraní, ktoré zahŕňajú metódy určitého typu.

- **ICloneable** – umožňuje klonovanie inštancií tried alebo štruktúr,
- **IComparable** – musia ho implementovať typy, ktoré bude možné usporiadať,
- **IFormattable** – umožňuje formátovanie pri konverzii na textový reťazec,
- **IEnumerable** – musia ho implementovať kontajnery, ktoré budú predchádzané `foreach`,
- **ICollection** – umožňuje kopírovanie časti obsahu kontajnera (napr. pole).

5.11 VLASTNOSTI (PROPERTIES)

Vlastnosti predstavujú syntaktickú skratku pre dvojicu prístupu k členským premenným triedy.

```

<modprist> typ menoVlastnosti
{
    set {...}
    get {...}
};

```

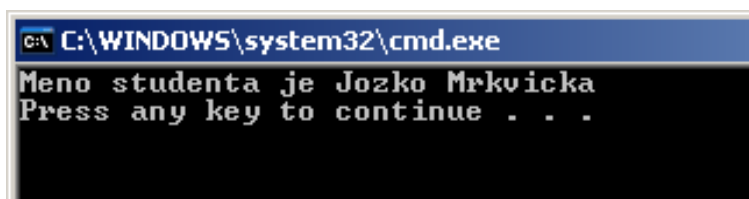
Na rozdiel do jazyka C++ nie sú vlastnosti `get` a `set` považované za metódy. Prístup k premenným pomocou vlastností stále zachováva princíp zapuzdrenia. Členské premenné zostávajú skryté a vlastnosť môže byť verejná. Časť `set` predstavuje prístup na nastavenie hodnoty vlastnosti a časť `get` predstavuje prístup na zistenie hodnoty vlastnosti. Vo vlastnosti nie je povinné uvádzať obidve časti `get` aj `set`, takže vynechaním časti `set` sa vytvorí vlastnosť iba na čítanie a vynechaním časti `get` sa vytvorí vlastnosť iba na nastavenie.

Príklad: Definícia a použitie vlastnosti

Trieda `Student` obsahuje verejnú vlastnosť `MenoProp`, v ktorej sú implementované prístupy `get` a `set` na získanie a zmenu hodnoty skrytej členskej premennej `Meno`.

```
using System;
namespace CA_Property
{
    class Student
    {
        private string Meno = String.Empty;
        public string MenoProp //vlastnost
        {
            get { return Meno; }
            set { Meno = value; }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Student std01 = new Student();
            //pouzitie vlastnosti
            std01.MenoProp = "Jozko Mrkvicka";
            Console.WriteLine("Meno studenta je {0}", std01.MenoProp);
        }
    }
}
```



Obr. 25 Definícia a použitie vlastnosti – výsledok v konzole

Vlastnosť s automatickou implementáciou

Jazyk C# od verzie 3.0 umožňuje vytvoriť vlastnosť, ktorá bude automaticky implementovaná podľa svojho typu. Automatická implementácia bude vykonaná, ak je zadaný iba typ vlastnosti `get` a `set` bez implementácie tela vlastnosti. Vytvorenie vlastnosti iba na čítanie alebo iba na zápis je možné uvedením prístupového práva pred `get` alebo `set` napr. `private set`.

Príklad: Využitie auto-implementačných vlastností

Trieda `Student` obsahuje dve verejné vlastnosti `MenoProp` a `KruzokProp`, v ktorých sa nachádzajú len názvy prístupov `get` a `set`. Implementáciu prístupu `get` a `set` nie je potrebné uvádzať.

```
using System;
namespace CA_Property
{
    class Student
    {
        public string MenoProp //autoimplementacna vlastnost
        {
            get;
            set;
        }
        public int KruzokProp
        {
            get;
            set;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Student std01 = new Student();
            //pouzitie autoimplementacnej vlastnosti
            std01.MenoProp = "Jozko Mrkvicka";
        }
    }
}
```

```

        std01.KruzokProp = 10;
        Console.WriteLine("Meno studenta je {0} a chodi do
        kruzku {1}", std01.MenoProp, std01.KruzokProp);
    }
}
}

```

5.12 DELEGÁTY (DELEGATES)

Delegát je objektovo zapuzdrený ukazovateľ na metódu triedy. Môže obsahovať adresu statickej metódy alebo adresu nestatickej metódy spolu s adresou inštancie, pre ktorú má byť volaná.

```
<modprist> delegate typ menoDelegata(<zoznam_parametrov>);
```

Príklad: Delegát

V triede `Program` je definovaný delegát `delObsah`, ktorý predstavuje typ hlavičiek (prototypov) funkcií s dvoma parametrami typu `int` a návratovou hodnotou typu `int`, ktoré môže delegovať. Statická funkcia `prcObsahObdlznika` má rovnakú hlavičku ako delegát, a teda môže byť použitá pri vytváraní delegáta. V hlavnej časti programu je vytvorený delegát `Obsah` typu `delObsah`, ktorý bude odkazovať na statickú funkciu `prcObsahObdlznika`, tzn. túto funkciu môžeme zavolať nielen jej menom ale aj pomocou delegáta `Obsah`.

```

using System;
namespace CA_Delegates
{
    class Program
    {
        public delegate int delObsah(int stranaa, int stranab);

        private static int prcObsahObdlznika(int a, int b)
        {
            return a * b;
        }

        static void Main(string[] args)
        {
            delObsah Obsah = new delObsah(prcObsahObdlznika);
        }
    }
}

```

```

        Console.WriteLine("Obsah obdlznika 12x47 je {0}", Obsah(12,47) );
    }
}
}

```

Viacnásobné delegáty

Jeden delegát môže súčasne ukazovať na viacero metód. Nové metódy sa do delegáta pridávajú pomocou operátora `+=` a odoberajú sa pomocou operátora `-=`. Pri volaní viacnásobného delegáta sa spúšťajú všetky jeho metódy v poradí, v akom sa do neho pridávali.

5.13 UDALOSTI (EVENTS)

Udalosti umožňujú podporu udalosťami riadeného programovania. Predstavujú činnosti počítača, prebiehajúceho programu, užívateľa a pod. Medzi činnosti počítača patria napr. prekreslenie obrazovky, zmena času... Udalosti generované prebiehajúcim programom môžu byť napr. spustenie programu, otvorenie súboru... Užívateľské udalosti sú napr. kliknutie na tlačidlo, stlačenie klávesu, pohyb myšou, atď.

V C++ neboli udalosti štandardne podporované, ale boli pridané pomocou externých knižníc ako sú MFC a VCL.

Udalosti deklarujeme ako dátové zložky triedy. Je potrebné definovať metódu, ktorá je určená na obsluhu udalosti – event handler. Pomocou delegátov sa volajú potrebné metódy na obsluhu udalosti. Udalosti je možné definovať aj ako vlastnosti.

Princíp publisher → subscriber

Udalosti pracujú na princípe vydavateľ → predplatiteľ. Inštancia, v ktorej môže vzniknúť udalosť (vydavateľ), má zoznam inštancií, ktoré môžu na túto udalosť reagovať (predplatiteľ). Predplatiteľ má u vydavateľa zaregistrovanú metódu, ktorá má reagovať na udalosť. Ak nastane definovaná udalosť, tak vydavateľ na to upozorní predplatiteľov tým, že zavolá ich zaregistrovanú metódu.

Event handler

Je to metóda reagujúca na udalosť. Event handler má návratový typ vždy `void` a najčastejšie má dva vstupné parametre.


```
void Meno(object zdroj, EventArgs e)
```

kde:

`object` je odkaz na inštanciu, ktorá je zdrojom udalosti,
`EventArgs` určuje parametre udalosti.

Event handler býva najčastejšie deklarovaný ako delegát.

Udalosť

Udalosť je deklarovaná ako dátová položka triedy:

```
<modif> event typ Identifikator;
```

Môže byť deklarovaná aj ako vlastnosť triedy:

```
<modif> event typ Meno(add a remove)
```

Príklad: Využitie udalosti

Trieda `Student` má definovanú udalosť `EventNeprospel`, ktorej typ je zhodný s názvom delegáta `Neprospel`, v ktorom sa budú registrovať všetky funkcie, ktoré majú na danú udalosť reagovať. V metóde `prcZadajStudent()` je táto udalosť vyvolaná v prípade, že bude zadaná známka "FX". Pri vyvolaní udalosti sú argumentmi metódy `prcZadajStudent()`, aktuálna inštancia `this` a nový objekt typu `EventArgs`.

V hlavnej časti v triede `Program` je definovaná funkcia `ZnamkaFX()`, ktorá bude reagovať na spomínanú udalosť triedy `Student`. Hlavička tejto funkcie sa parametricky zhoduje s hlavičkou delegáta. Aby inštancia programu mohla reagovať na udalosť v inštancii triedy `Student`, tak v tejto inštancii treba zaregistrovať funkciu `ZnamkaFX()` pomocou delegáta `Neprospel()` nasledujúcim spôsobom:

```
std01.EventNeprospel += new Neprospel(ZnamkaFX);
```

Až do tohto momentu bude fungovať obsluha udalosti príslušnou funkciou. Na overenie funkčnosti je následne zavolaná metóda `prcZadajStudent()`.

```
using System;
namespace CA_Events
{
    //event handler
    public delegate void Neprospel(object zdroj, System.EventArgs e);
```

```

class Student
{
    public event Neprospel EventNeprospel; //udalost - event

    string sMeno;
    int iKruzok;
    string sZnamka;

    //vlastnost iba na citanie
    public string MenoStud
    {
        get { return sMeno; }
    }

    public void prcZadajStudent()
    {
        Console.Write("Zadaj meno studenta: ");
        sMeno = Console.ReadLine();
        Console.Write("Zadaj cislo kruzku studenta: ");
        iKruzok = Int32.Parse(Console.ReadLine());
        Console.Write("Zadaj znamku studenta: ");
        sZnamka = Console.ReadLine();

        if (sZnamka == "FX") //podmienka volania udalosti
            EventNeprospel(this, new EventArgs());
    }

    public void prcVypisStudent()
    {
        Console.WriteLine("Student {0} z kruzku {1} ma znamku {2}",
                           sMeno, iKruzok, sZnamka);
    }
}

```

```

class Program
{
    //metoda, ktora reaguje na udalost
    public static void ZnamkaFX(object zdroj, EventArgs e)
    {
        Console.WriteLine("*** Student {0} neprospel! ***",
            (zdroj as Student).MenoStud);
    }

    static void Main(string[] args)
    {
        Student std01 = new Student();
        //priradenie metody k udalosti
        std01.EventNeprospel += new Neprospel(ZnamkaFX);
        std01.prcZadajStudent();
    }
}

```

```

C:\WINDOWS\system32\cmd.exe
Zadaj meno studenta: Karol
Zadaj cislo kruzku studenta: 6
Zadaj znamku studenta: FX
*** Student Karol neprospel! ***
Press any key to continue . . .

```

Obr. 26 Využitie udalosti – výsledok v konzole

6 VÝNIMKY

6.1 CHARAKTERISTIKA A DEFINÍCIA VÝNIMIEK

Mechanizmus výnimiek slúži na prenos riadenia pri vzniku chyby stavu počas priebehu programu. Výnimka vznikne ako dôsledok chyby v programe (chyba matematických operácií, parametrov metód, práce s premennými, nesprávne pretypovanie, prístup k súborom, operačnej pamäte...).

Výnimky sú zachytávané pomocou handlerov výnimiek a následne je možné vykonať ich ošetrovanie (nie je vždy potrebné). Blok programu, ktorý je ošetrený pri vzniku výnimky, sa nazýva „strážení blok“. V jednom stráženom bloku môže byť zahrnuté odchyťovanie viacerých výnimiek.

Obsluha výnimky, resp. výnimiek sa vykonáva v bloku, ktorý sa nazýva „obsluha výnimky“. Táto časť zdrojového kódu sa vykonáva iba vtedy, ak počas priebehu programu vznikne v stráženom bloku výnimka. Ak výnimka nevznikne, kód obsluhy výnimky bude preskočený.

K týmto dvom blokom je možné pripojiť blok, ktorý sa nazýva „koncovka“. Na rozdiel od bloku obsluhy výnimky, zdrojový kód bloku koncovky sa vykoná vždy. Vykoná sa, ak v stráženom bloku nevznikne výnimka, ale aj v prípade, ak z stráženom bloku vznikla výnimka a bola ošetrená v bloku obsluhy výnimky. Koncovka sa vykoná aj vtedy, ak sa v stráženom bloku vykoná niektorý z príkazov skoku, ktorý vedie mimo stráženého bloku. Koncovka sa vykoná bezprostredne predtým, ako program prejde na miesto, kde prenáša riadenie vykonaný príkaz skoku.

6.1.1 Konštrukcie príkazu try

Príkaz **try** sa skladá z troch základných blokov:

- strážení blok – **try**,
- obsluha výnimky, resp. výnimiek – **catch**,
- koncovka bloku – **finally**.

Z týchto troch blokov je možné vytvoriť štyri varianty príkazu **try**. Každá z nich má svoje špecifické využitie.

try – catch

```
try
{
    prikazy
}
catch (handle_vynimky)
{
    obsluha vynimky
};
```

try – catch

```
try
{
    prikazy
}
catch (handler_vynimky1)
{
    obsluha vynimky 1
}
catch (handler_vynimkyN)
{
    obsluha vynimky N
}
```

try – catch – finally

```
try
{
    prikazy
}
catch (handler_vynimky)
{
    obsluha vynimky
}
finally
{
```

```
        koncovka bloku
    }
```

try – finally

```
try
{
    prikazy
}
finally
{
    koncovka bloku
}
```

Použitie konkrétneho variantu príkazu `try` závisí od špecifickej situácie vo vytváranom programe.

6.2 ŠÍRENIE VÝNIMIEK

Ak vznikne v programe výnimka, začne program hľadať zodpovedajúci handler výnimky. Po vzniku výnimky program preskočí všetky zostávajúce príkazy v aktuálnom bloku. Ak výnimka vznikla v stráženom bloku s jedným alebo viacerými handlermi výnimiek, začne postupne skúmať jednotlivé handlers a hľadať, ktorý z nich môže zachytiť a ošetriť vzniknutú výnimku. O tom, ktorý handler zachytí výnimku, rozhoduje typ handleru. Pri vyhodnocovaní handlerov sa uplatňuje pravidlo o dedičnosti z objektovo orientovaného programovania. Ak budú v časti `catch` použité odvodené výnimky a aj výnimka, z ktorej sú odvodené (ich predok), je potrebné uviesť odvodené výnimky ako prvé.

Ak vznikla výnimka v bloku, ktorý nie je strážený, prípadne ak v stráženom bloku nie je handler výnimky, ktorý by zodpovedal typu vzniknutej výnimky, prejde riadenie do dynamicky nadriadeného bloku. Ak vznikla výnimka v tele metódy, opustí program túto metódu a prejde do metódy, z ktorej bola volaná. Opäť dochádza k preskočeniu zdrojového kódu metódy a k snahe nájsť zodpovedajúci handler výnimky. Ak sa nájde správny handler výnimky, riadenie prechádza do neho a vykonajú sa riadky zdrojového kódu v časti bloku určeného na ošetrovanie výnimky. Ak sa tam nachádzajú aj iné handlers výnimiek, tak sú preskočené a program pokračuje za stráženým blokom. Ak príkaz `try` obsahuje aj koncovku `finally`, tak sa po ošetrovaní výnimky vykoná najskôr koncovka a až potom pokračuje

program za stráženým blokom. Vstupom do príslušného handleru výnimky sa považuje výnimka za ošetrenú, tzn. blok ošetrenia handleru výnimky môže byť prázdny (nemusí obsahovať žiadne príkazy).

Ak sa nenájde zodpovedajúci handler výnimky, prechádza sa do ďalšieho dynamicky nadriadeného bloku až dovtedy, pokiaľ sa riadenie nedostane do metódy `Main()`. Ak ani tu nie je zodpovedajúci handler, prechádza riadenie do .NET Frameworku. Prostredie .NET Framework vypíše štandardné chybové hlásenie, ktoré zodpovedá vzniknutej výnimke a program je korektne ukončený.

Pri šírení výnimky programom (prechodom riadenia do dynamicky nadriadených blokov programu) sú všetky opúšťané bloky korektne ukončené. Ak existujú lokálne premenné definované v opúšťanom bloku, sú tieto premenné odstránené zo zásobníka. Takto môžu zaniknúť aj odkazy na dynamicky alokované inštancie tried a polí. O odstránenie týchto inštancií sa v prípade potreby postará automatická správa pamäte (Garbage Collector).

Vyvolanie výnimky je štandardne spôsobené chybou vo vykonávanom programe. V prípade potreby má programátor možnosť vyvolať výnimku pomocou príkazu `throw`. Priebeh šírenia a ošetrenia takto vzniknutej výnimky je totožný so šíreným a ošetrením štandardne vzniknutej výnimky.

Príklad: Výnimky

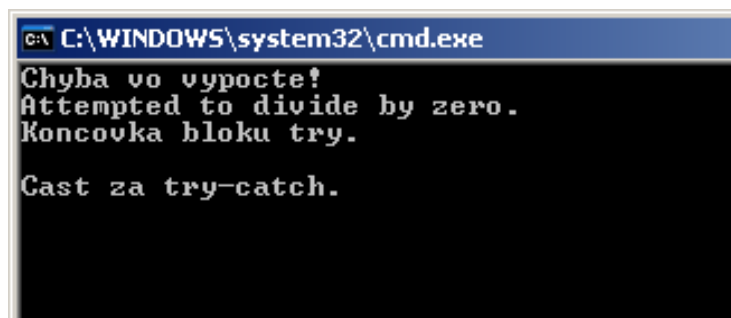
V nasledujúcom príklade je použitý kompletný blok `try-catch-finally`. V časti `try` sú príkazy, v ktorých sa predpokladá vznik chyby počas priebehu programu. V časti `catch` sa zachytia všetky výnimky odvodené zo základného typu `Exception`, pričom sa vypíše aj správa vo výnimke obsiahnutá. Časť `finally` sa vykoná vždy, či výnimka nastane, alebo nenastane.

```
using System;
namespace CA_TryCatch
{
    class Program
    {
        static void Main(string[] args)
        {
            int iVysledok, iCislo1 = 1, iCislo2 = 0;
```

```

try
{
    iVysledok = iCislo1 / iCislo2;
}
catch (Exception e)
{
    Console.WriteLine("Chyba vo vypocte!\n" + e.Message);
}
finally
{
    Console.WriteLine("Koncovka bloku try.\n");
}
Console.WriteLine("Cast za try-catch.");
}
}
}

```



Obr. 27 Výnimky – výsledok v konzole

6.3 TRIEDY PRE VÝNIMKY

Knižnica základných tried (BCL) prostredia .NET Framework obsahuje niekoľko preddefinovaných tried, ktoré je možné použiť na prenos informácií o výnimkách. Tieto je možné doplniť o vlastné, programátorom definované triedy.

Trieda `System.Exception` je spoločný predok pre všetky triedy výnimiek. Obsahuje niekoľko vlastností, ktoré môžu pomôcť programátorovi pri ladení programu. Medzi najdôležitejšie patrí:

- `StackTrace` – vlastnosť, ktorá obsahuje znakový reťazec predstavujúci všetky metódy od vzniku výnimky až po blok `catch`.

- **InnerException** – vlastnosť, ktorá umožní pri vyvolaní vnútornej výnimky (výnimky, ktorá vznikne v bloku ošetrovania výnimky) uchovať informácie o pôvodnej výnimke.
- **Message** – vlastnosť, ktorá obsahuje textový reťazec so stručným opisom chyby (zadaný programátorom v programe alebo priamo z knižnice základných tried z prostredia .NET Framework), vhodným pre programátora, resp. pre užívateľa programu.

Základná trieda **System.Exception** sa pri riadení informácií o výnimkách nepoužíva často. Býva nahradená niektorou z odvodených tried, pomocou ktorých je možné presnejšie identifikovať vzniknutú výnimku.

System.SystemException

Predstavuje spoločného predka tried na prácu s výnimkami, definovanými v mennom priestore **System**.

System.ArithmeticException

Používa sa pri výnimkách, ktoré vzniknú počas matematických výpočtov. Sú z nej odvodené viaceré triedy, ktoré presnejšie identifikujú typ výnimky. Napr.:

System.DivideByZeroException pre odchytenie výnimky pri pokuse o delenie nulou, **System.OverflowException** pre výnimky, ktoré vzniknú pri celočíselnom pretečení.

System.InvalidCastException

Využíva sa na ošetrovanie výnimiek, vznikajúcich pri nesprávnom pretypovaní.

System.IO.IOException

Je spoločným predkom tried určených na zachytávanie a ošetrovanie výnimiek, ktoré môžu vzniknúť počas práce so vstupno/výstupnými operáciami. Ide napríklad o výnimky pri práci s adresármi (**System.IO.DirectoryNotFoundException**), rôznymi typmi súborov (**System.IO.FileNotFoundException**, **System.IO.FileLoadException**), pri práci s prúdmi (**System.IO.EndOfStreamException**) a podobne.

System.IndexOutOfRangeException

Výnimka vznikne pri pokuse o prístup k prvku poľa, ktorého index leží mimo jeho definovaného rozsahu.

6.4 POUŽITIE OŠETRENIA VÝNIMIEK

Odchyťovanie a ošetrovanie výnimiek je možné použiť všade tam, kde je predpoklad vzniku výnimky. Napomáha programátorovi zabezpečiť syntakticky stručným a jednotným spôsobom identifikovanie a ošetrovanie chýb bez nutnosti vykonávať viaceré opatrenia v zdrojovom kóde, ktoré majú za úlohu predísť situáciám, ktoré by mohli byť príčinou vzniku chyby v programe.

Napriek nesporným výhodám, ktoré tento proces prináša, je potrebné zodpovedne pristupovať k umiestňovaniu strážených blokov do zdrojového kódu aplikácie. Samotné generovanie a ošetrovanie výnimiek môže zaberať pomerne veľké množstvo systémových prostriedkov pridelených programu. Ak bude strážený blok umiestnený v metóde, ktorá je často volaná, prípadne vykonáva náročnejšiu činnosť, bude táto metóda menej výkonná ako v prípade, že by v nej nebol umiestnený strážený blok.

Ak je to možné, tak je vhodné vyhnúť sa príčine vzniku výnimky napr. ošetrením vstupu do volanej metódy, zabránením prekročenia rozsahu poľa pri práci s ním a podobne.

7 POLIA

Pole je neusporiadaná postupnosť prvkov. Všetky prvky v poli sú rovnakého typu (na rozdiel od dátových zložiek v štruktúrach alebo triedach, kde sú povolené rozdielne typy). Prvky poľa sú uložené v súvislom bloku pamäte a pre prístup k nim sa používa celočíselný index (pri dátových zložkách v štruktúrach alebo triedach je to ich názov), ktorý sa štandardne začína od čísla 0.

Pole patrí k referenčným dátovým typom nezávisle od toho, akého dátového typu sú jeho prvky. To znamená, že premenná typu pole odkazuje na súvislý blok uchovávajúci pole prvkov v dynamickej pamäti, rovnako ako inštancia triedy odkazuje na objekt v dynamickej pamäti, pričom tento súvislý blok pamäte neudržiava prvky poľa na zásobníku, ako je tomu v prípade štruktúr. Keď je deklarovaná premenná typu trieda, tak je pamäť pre objekt pridelená až pri vytvorení jeho inštancie kľúčovým slovom `new`. Pole podlieha rovnakým pravidlám – pri deklarácii premennej typu pole sa neuvádza jeho veľkosť, tá sa zadáva až pri skutočnom vytvorení inštancie poľa.

Inštancia poľa sa vytvára pomocou kľúčového slova `new`, za ktorým nasleduje dátový typ prvkov poľa a v hranatých zátvorkách veľkosť poľa. Pri vytvorení poľa sú taktiež inicializované jeho prvky štandardnými hodnotami, zodpovedajúcimi zvolenému dátovému typu (0, null alebo false, v závislosti od toho, či ide o dátový typ číselný, referenčný, alebo logický). O zrušenie poľa sa stará automatická správa pamäte (Garbage Collector).

Pole v jazyku C# predstavuje objekt odvodený od triedy `System.Array`. Okrem samotných prvkov poľa obsahuje ešte ďalšie vlastnosti a metódy, ktoré sú zdedené z triedy `System.Array` a umožňujú napr. zistenie veľkosti poľa, jeho dimenzii, zoradiť pole, kopírovať ho a podobne. Keďže trieda `System.Array` implementuje rozhranie `IEnumerable`, je možné použiť cyklus `foreach` na prechádzanie jednotlivých prvkov poľa.

Na rozdiel od polí v jazyku C++ je v jazyku C# kontrolovaný prístup k prvkom poľa, či vybraný prvok leží v rozsahu danom deklaráciou poľa. Ak príde k prekročeniu hranice poľa, tak je vygenerovaná výnimka `System.IndexOutOfRangeException`.

7.1 JEDNOROZMERNÉ POLIA

Jednorozmerné pole predstavuje „vektor“ prvkov daného typu.

Deklarácia jednorozmerného poľa je:

```
DatTyp[] meno;
```

kde:

DatTyp – dátový typ deklarovaného poľa,

meno – meno deklarovaného poľa.

Napríklad:

```
int[] pole1D;
```

Je možné priamo spojiť deklaráciu poľa s jeho inicializáciou:

```
DatTyp[] meno = new DatTyp[velkost];  
DatTyp[] meno = new DatTyp[]{hodnoty};  
DatTyp[] meno = new DatTyp[velkost]{hodnoty};  
DatTyp[] meno = {hodnoty};
```

Napríklad:

```
float[] pole1D = new float[5];  
char[] pole1D= new char[]{'A', 'N'};  
byte[] pole1D = new byte[3]{1,2,3};  
int[] pole1D = {1,2,3,4,5,6,7,8};
```

Pokiaľ je pri inicializácii poľa zadaná veľkosť poľa a súčasne sú zadané aj inicializačné hodnoty jednotlivých prvkov poľa, je potrebné, aby zadaná veľkosť poľa bola zhodná s počtom inicializačných hodnôt.

Napríklad:

```
byte[] pole1D = new byte[2]{1,2,3}; //chyba  
byte[] pole1D = new byte[3]{1,2,3}; //v poriadku
```

Príklad: Práca s jednorozmerným poľom

Vo funkcii Main() je deklarované a zároveň inicializované jednorozmerné pole pole1D piatich prvkov typu int. Následne je na výpis jednotlivých hodnôt z poľa použitý cyklus **for**, kde k jednotlivým položkám pristupujeme pomocou indexu danej položky v hranatej zátvorke. Druhú možnosť predstavuje iterácia prvkami poľa pomocou cyklu **foreach**, kde hodnotu každej položky vráti **foreach** sám.

```

using System;
namespace CA_Pole1D
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] pole1D = new int[5] {1, 2, 3, 4, 5};

            Console.Write("Vypis pola pomocou cyklu FOR: ");
            for (int i = 0; i < 5; i++)
                Console.Write("{0, 3}", pole1D[i]);

            Console.Write("\nVypis pola pomocou cyklu FOREACH: ");
            foreach (int i in pole1D)
                Console.Write("{0, 3}", i);
        }
    }
}

```

7.1.1 Kopírovanie jednorozmerných polí

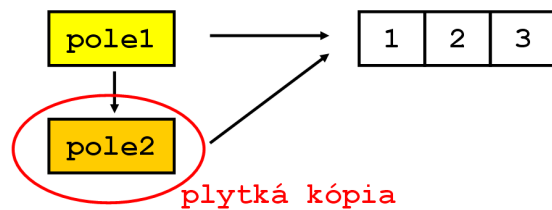
Plytká kópia poľa

Pole v programovacom jazyku C# patrí k referenčným dátovým typom a z toho dôvodu nesie premenná poľa odkaz na pamäťové miesto v dynamickej pamäti, kde sa nachádzajú uložené hodnoty jednotlivých prvkov poľa. Preto pri kopírovaní poľa v nasledujúcom prípade je do premennej pole2 skopírovaný iba odkaz na pole1. Obidve premenné, pole1 a pole2 obsahujú odkaz do rovnakej pamäťovej oblasti. V tomto prípade je vytvorená tzv. plytká kópia poľa.

```

int[] pole1 = new int[3]{1,2,3};
int[] pole2;
pole2 = pole1;

```

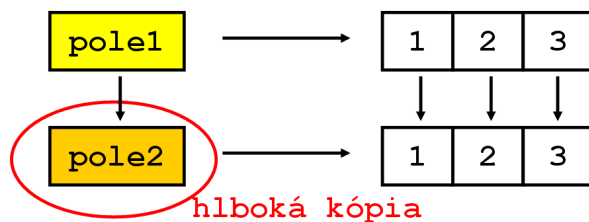


Obr. 28 Plytká kópia poľa

Hlboká kópia poľa

Ak je potrebné vytvoriť identickú kópiu existujúceho poľa, je možné na to použiť metódu `Clone()`, ktorú dedí objekt poľa z triedy `System.Array`. Táto metóda zabezpečí vytvorenie tzv. hlbokéj kópie poľa.

```
int[] pole1 = new int[3]{1,2,3};
int[] pole2;
pole2 = (int[])pole1.Clone();
```



Obr. 29 Hlboká kópia poľa

7.1.2 Vlastnosti a metódy triedy `System.Array`

Systémová trieda `System.Array`, ktorá je predkom polí vytvorených v programovacom jazyku C#, prináša niekoľko vlastností a metód, ktoré môže programátor využiť pri práci s poľom. Väčšina metód triedy `System.Array` je viackrát preťažená. Uvedené sú najčastejšie používané varianty.

Vlastnosť `Length`

Slúži na zistenie veľkosti poľa (počet prvkov).

```
meno_pola.Length;
```

Vlastnosť `Rank`

Slúži na zistenie dimenzie poľa.

```
meno_pola.Rank;
```

Metóda Array.Sort

Metóda `Array.Sort` slúži na utriedenie jednorozmerného poľa.

```
Array.Sort(meno_pola);  
Array.Sort(meno_pola, od, pocet);
```

kde:

meno_pola – meno poľa, ktoré sa má utriediť,

od – index prvku poľa, od ktorého sa má pole utriediť,

pocet – počet prvkov poľa, ktoré sa majú utriediť.

Metóda Array.Reverse

Táto metóda umožní reverzovanie jednorozmerného poľa.

```
Array.Reverse(meno_pola);  
Array.Reverse(meno_pola, od, pocet);
```

Metóda Array.Clear

Metóda `Array.Clear` slúži na vynulovanie prvkov poľa v zadanom intervale. Do prvkov poľa je zadaná hodnota 0, null alebo false, v závislosti od toho, či ide o dátový typ číselný, referenčný alebo logický.

```
Array.Clear(meno_pola, od, pocet);
```

Metóda Array.IndexOf

Metóda `Array.IndexOf` vráti index prvého výskytu zadanej hodnoty v prehľadávanom jednorozmernom poli.

```
Array.IndexOf(meno_pola, hodnota);  
Array.IndexOf(meno_pola, hodnota, od);
```

Metóda Array.BinarySearch

Slúži na vyhľadávanie hodnoty v utriedenom jednorozmernom poli s využitím binary search algoritmu.

```
Array.BinarySearch(meno_pola, hodnota);  
Array.BinarySearch(meno_pola, od, pocet, hodnota);
```

7.2 VIACROZMERNÉ PRAVIDELNÉ POLIA

Deklarácia viacrozmerných pravouhlých polí (pravouhlé viacrozmerné polia majú v jednom rozmere vždy rovnaký počet prvkov) v jazyku C# je podobná deklarácii jednorozmerného poľa. Hlavný rozdiel je v tom, že hranatá zátvorka obsahuje čiarky, ktoré oddelujú jednotlivé rozmery poľa. Na základe nasledujúceho pravidla je daný rozmer poľa počtom čiarok v hranatej zátvorke.

rozmer poľa = počet čiarok + 1

Na rozdiel od programovacieho jazyka C++ v jazyku C# sa pri viacrozmerných poliach píše veľkosť jednotlivých rozmerov do jednej dvojice hranatých zátvoriek.

Deklarácia dvojrozmerného poľa:

```
DatTyp[, ] meno;
```

kde:

DatTyp – dátový typ deklarovaného poľa,

meno – meno deklarovaného viacrozmerného poľa.

Napríklad:

```
byte[, ] pole2D; //dvojrozmenne pole  
int[, , ] pole3D; //trojrozmenne pole
```

Podobne ako pri jednorozmerných poliach je aj pri viacrozmerných poliach možné spojiť deklaráciu poľa s jeho inicializáciou.

```
DatTyp[, ] meno = new DatTyp[, ]{{...},{...}};
```

Napríklad:

```
byte[, ] meno = new byte[, ]{{1,2},{3,4}};
```

Príklad: Dvojrozmerné pravidelné pole

Dvojrozmerné pole sa podobá matici či tabuľke. V príklade vytvárame pole celých čísel s názvom pole2D veľkosti 4x3. Pre pohyb v takomto poli sa najčastejšie používajú dva cykly **for**, kde prvý cyklus mení index riadka a druhý cyklus mení index stĺpca. Na zistenie počtu riadkov či stĺpcov sa používa metóda `GetLength()`, ktorej parametrom je poradové číslo rozmeru viacrozmerného poľa. Pole je naplnené hodnotami z generátora náhodných čísel. Pri výpise je použitý cyklus **foreach**, ktorý prejde síce všetkými prvkami matice, ale ak chceme

vedieť, v ktorom riadku sa nachádzame, tak si to musíme sami odsledovať počítaním počtu spracovaných prvkov a počtu prvkov v riadku.

```
using System;
namespace CA_Pole2D
{
    class Program
    {
        static void Main(string[] args)
        {
            Random generator = new Random();
            int[,] pole2D = new int[4, 3];

            for (int i = 0; i < pole2D.GetLength(0); i++)
                for (int j = 0; j < pole2D.GetLength(1); j++)
                    pole2D[i, j] = generator.Next(10);

            int x = 0;
            foreach(int a in pole2D)
            {
                Console.Write("{0, 3}", a);
                x++;
                if (x == pole2D.GetLength(0))
                {
                    x = 0;
                    Console.WriteLine();
                }
            }
        }
    }
}
```

7.2.1 Vlastnosti a metódy triedy System.Array

Vlastnosť Length

Slúži na zistenie počtu prvkov vo všetkých rozmeroch poľa.

```
meno_pola.Length;
```

Vlastnosť GetLength

Vlastnosť `GetLength` umožní zistenie počtu prvkov poľa v zadanom rozmere.

```
meno_pola.GetLength(rozmer);
```

Vlastnosť Rank

Slúži na zistenie dimenzie poľa.

```
meno_pola.Rank;
```

Metóda Array.Clear

Metóda `Array.Clear` slúži na vynulovanie prvkov poľa v zadanom intervale. Do prvkov poľa je zadaná hodnota `0`, `null` alebo `false`, v závislosti od toho, či ide o dátový typ číselný, referenčný, alebo logický.

```
Array.Clear(meno_pola, od, pocet);
```

7.3 VIACROZMERNÉ NEPRAVIDELNÉ (ZUBATÉ) POLIA

Na rozdiel od pravidelných viacrozmerných polí majú nepravidelné polia (jagged array) rozličný počet prvkov v jednotlivých rozmeroch. Nepravidelné pole predstavuje pole polí a je zložené z viacerých jednorozmerných polí.

Deklarácia nepravidelného poľa sa podobá deklarácii viacrozmerného poľa v programovacom jazyku C++. Jednotlivé rozmery nepravidelného poľa sú umiestnené do samostatných hranatých zátvoriek.

Deklarácia dvojrozmerného nepravidelného poľa:

```
DatTyp[][] meno;
```

kde:

`DatTyp` – dátový typ deklarovaného poľa,

`meno` – meno deklarovaného nepravidelného poľa.

Napríklad:

```
char[][] pole2D;
```

Nepravidelné polia sa deklarujú postupne. Pri dvojrozmernom nepravidelnom poli sa najskôr deklaruje veľkosť poľa v prvom rozmere:

```
DatTyp[][] meno=new DatTyp[2][];
```

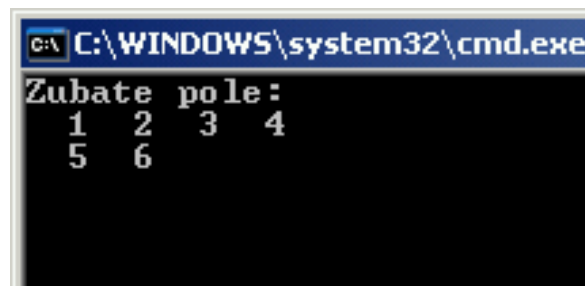
a potom sa deklarujú veľkosti jednotlivých riadkov (jednorozmerných polí) v nepravidelnom poli:

```
meno[0]=new DatTyp[velkost1]{...};  
meno[1]=new DatTyp[velkost2]{...};
```

Príklad: *Dvojrozmerné nepravidelné pole*

V príklade je definované dvojrozmerné pole `poleJagged` bez udania dĺžky druhého rozmeru. Následne je každému riadku priradené pole s rôznym počtom prvkov. Pri spracovaní potom v cykloch `for` zistíme počet riadkov a počet položiek v nich pomocou atribútu `Length`.

```
using System;  
namespace CA_PoleJagged  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            int[][] poleJagged = new int[2][];  
            poleJagged[0] = new int[4]{1, 2, 3, 4};  
            poleJagged[1] = new int[2]{5, 6};  
  
            Console.WriteLine("Zubate pole:");  
            for (int i = 0; i < poleJagged.Length; i++)  
            {  
                for (int j = 0; j < poleJagged[i].Length; j++)  
                    Console.Write("{0, 3}", poleJagged[i][j]);  
                Console.WriteLine();  
            }  
        }  
    }  
}
```



```
C:\WINDOWS\system32\cmd.exe
Zubate pole:
1 2 3 4
5 6
```

Obr. 30 Dvojrozmerné nepravidelné pole – výsledok v konzole

8 PRÁCA S ADRESÁRMÍ A SO SÚBORMI

Programovací jazyk C# poskytuje programátorovi viaceré triedy a metódy na prácu s adresármi a súbormi súborového systému. Adresáre je možné napr. vytvárať, premenovávať, vyhľadávať, mazať, meniť ich parametre. Podobné možnosti sú aj pri práci so súbormi, kde sa navyše umožňuje otvárať súbory, čítať a zapisovať do nich, zatvárať ich, atď. Programovací jazyk C# rozlišuje medzi prácou s textovými súbormi a binárnymi súbormi a k práci s nimi poskytuje triedy a metódy na to určené.

Základné nástroje (triedy, metódy, výpočtové typy...) pre prácu s adresármi a súbormi obsahuje menný priestor `System.IO`.

8.1 PRÁCA S ADRESÁRMÍ

Základné metódy práce s adresármi obsahuje trieda `System.IO.Directory`:

- zistenie existencie adresára:
`Directory.Exists(menoadr);`
- názov aktuálneho (pracovného) adresára – výsledné meno adresára je v plnom formáte napr. `E:\OOP\FileSystem`:
`Directory.GetCurrentDirectory();`
- výpis podadresárov v zadanom adresári:
`Directory.GetDirectories(menoadr);`
- výpis súborov v zadanom adresári:
`Directory.GetFiles(menoadr);`
- výpis podadresárov a súborov v zadanom adresári:
`Directory.GetFileSystemEntries(meno);`
- vytvorenie nového adresára:
`Directory.CreateDirectory(menoadr);`
- presunutie adresára alebo súboru:
`Directory.Move(zdrojadr, cieladr);`
- vymazanie adresára:
`Directory.Delete(menoadr);`

Príklad: *Práca s adresármi a súbormi*

Príklad využitia služieb `System.IO.Directory` na získanie aktuálneho adresára prebiehajúceho programu, a tiež zoznamu len jeho podadresárov, len jeho súborov a nakoniec zoznamu jeho podadresárov aj súborov. Na výpis zoznamov je použitý cyklus `foreach`.

```
using System;
using System.IO;
namespace CA_SuboryAdresare
{
    class Program
    {
        static void Main(string[] args)
        {
            string menoadr = Directory.GetCurrentDirectory();
            string[] adresare = Directory.GetDirectories(menoadr);
            string[] subory = Directory.GetFiles(menoadr);
            string[] adrsb = Directory.GetFileSystemEntries(menoadr);

            Console.WriteLine("Pracovny adresar: {0}", menoadr);
            Console.WriteLine("\nVypis podadresarov:");
            foreach (string s in adresare)
            {
                Console.WriteLine("{0}", s);
            }
            Console.WriteLine("\nVypis suborov:");
            foreach (string s in subory)
            {
                Console.WriteLine("{0}", s);
            }
            Console.WriteLine("\nVypis podadresarov a suborov");
            foreach (string s in adrsb)
            {
                Console.WriteLine("{0}", s);
            }
        }
    }
}
```

8.2 PRÁCA S TEXTOVÝMI SÚBORMI

Základné metódy na prácu so súbormi sa nachádzajú v triedach `System.IO.FileInfo` a `System.IO.FileStream`. Trieda `FileInfo` obsahuje základné metódy určené na prácu so súbormi (zistenie umiestnenia súboru, kopírovanie, mazanie...). Trieda `FileStream` obsahuje

základné metódy prístupu k súborom (vytvorenie nového súboru, otvorenie súboru na čítanie alebo zápis...).

8.2.1 Práca s textovými súbormi – FileInfo

Inicializácia práce sa vykonáva v konštruktore triedy:

```
FileInfo fiMeno = new FileInfo(menosuboru);
```

Na zistenie mena súboru a jeho umiestnenia slúžia nasledujúce vlastnosti:

- `fiMeno.Name` – obsahuje meno súboru,
- `fiMeno.FullName` – obsahuje meno súboru aj s kompletnou cestou.

Ďalej umožňuje zistenie adresárovej štruktúry umiestnenia súboru:

- `fiMeno.Directory` – vráti názov rodičovského adresára,
- `fiMeno.DirectoryName` – vráti reťazec, ktorý obsahuje celú cestu k rodičovskému adresáru.

Na zistenie základných informácií o zadanom súbore slúžia nasledujúce vlastnosti:

- `fiMeno.Length` – vráti veľkosť zadaného súboru,
- `fiMeno.Attributes` – vráti vlastnosti zadaného súboru.

Trieda `FileInfo` poskytuje programátorovi niekoľko metód, ktoré umožňujú vykonávať základné operácie so súborom. Najčastejšie používané sú:

- `fiMeno.CopyTo(umiestnenie)` – slúži na kopírovanie existujúceho súboru do nového súboru.
- `fiMeno.MoveTo(umiestnenie)` – umožňuje presun zadaného súboru na zadané miesto a poskytuje možnosť premenovať presunutý súbor,
- `fiMeno.Delete()` – slúži na vymazanie zvoleného súboru.

8.2.2 Práca s textovými súbormi – FileStream

Tak ako v prípade triedy `FileInfo` aj prípade triedy `FileStream` sa inicializácia vykonáva v konštruktore:

```
FileStream fsMeno = new FileStream(subor, modotvorenia, typpristupu);
```

Na určenie módu otvorenia súboru slúži trieda `FileMode`, ktorá poskytuje nasledujúce módy otvorenia:

- `FileMode.Create` – pokúsi sa vytvoriť nový súbor. Ak súbor v zadanom adresári neexistuje, tak sa vytvorí nový súbor. Ak ale súbor už existuje, tak sa tento súbor otvorí a jeho obsah sa vymaže.
- `FileMode.CreateNew` – pokúsi sa o vytvorenie nového súboru so zadaným názvom. Ak taký súbor v zadanom adresári neexistuje, tak bude vytvorený. Ak súbor so zadaným názvom už existuje, tak sa súbor nevytvorí, ale vznikne výnimka `IOException`.
- `FileMode.Open` – slúži na otvorenie existujúceho súboru. Ak súbor so zadaným názvom neexistuje, tak vznikne výnimka `FileNotFoundException`.
- `FileMode.OpenOrCreate` – umožní otvorenie existujúceho súboru. Ak zvolený súbor existuje, tak je otvorený. Ak požadovaný súbor neexistuje, tak je vytvorený nový súbor so zadaným menom a následne je otvorený.
- `FileMode.Append` – ak zvolený súbor existuje, tak je otvorený a pozícia kurzora je nastavená za posledný znak súboru. Ak zvolený súbor neexistuje, tak sa vytvorí nový súbor so zadaným názvom. Tento mód otvorenia môže byť použitý iba s módom otvorenia na zápis (`FileAccess.Write`). Pokus o posun na pozíciu pred posledný znak súboru spôsobí vznik výnimky `IOException` a pokus o čítanie zo súboru spôsobí vznik výnimky `NotSupportedException`.
- `FileMode.Truncate` – slúži na otvorenie existujúceho súboru. Po otvorení súboru je jeho obsah vymazaný. Pokus o čítanie z takto otvoreného súboru spôsobí vznik výnimky.

Typ prístupu k súboru určuje trieda `FileAccess`. Môže nadobúdať nasledujúce hodnoty:

- `FileAccess.Write` – súbor bude otvorený na zápis,
- `FileAccess.Read` – súbor bude otvorený na čítanie,
- `FileAccess.ReadWrite` – súbor bude otvorený na zápis a aj na čítanie.

Trieda `FileStream` obsahuje niekoľko konštruktorov. Programátor si môže podľa potreby vybrať ten, ktorý riešenej problematike najviac vyhovuje. Podľa toho sa pri inicializácii uvádza požadovaný počet parametrov konšuktora.

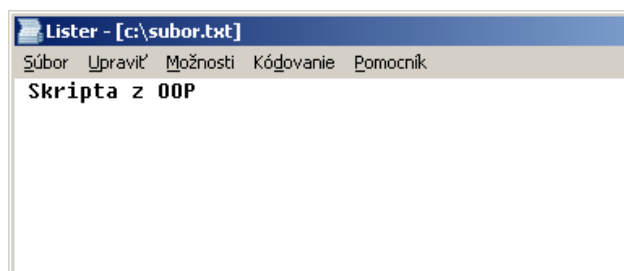
Príklad: Práca s textovými súbormi pomocou triedy *FileStream*

Na začiatku programu je definované a inicializované pole bajtov s názvom `Text` s rôznymi hodnotami. Následne je definovaná inštancia triedy `FileStream`, ktorá otvorí daný súbor v režime `FileMode.Append`, čiže na dopĺňanie na koniec súboru. Do súboru sa následne zapíše blok bajtov `Text` od pozície 0 po jeho koniec. Na konci sa súbor zatvorí.

```
using System;
using System.IO;
namespace CA_FileStream
{
    class Program
    {
        static void Main(string[] args)
        {
            byte[] Text = new byte[] {83, 107, 114, 105, 112, 116, 97, 32,
                                      122, 32, 79, 79, 80};

            FileStream fsZapis = new FileStream("c:\\subor.txt",
                                                FileMode.Append);

            fsZapis.Write(Text, 0, Text.Length);
            fsZapis.Close();
        }
    }
}
```



Obr. 31 Práca s textovými súbormi pomocou triedy *FileStream* – obsah vytvoreného súboru

8.2.3 Práca s text. súbormi – `StreamWriter`

Pomocou triedy `FileStream` je možné zapisovať a čítať zo súboru iba po bytoch. Aby bolo možné čítať a zapisovať do súboru priamo znaky, je výhodné použiť triedy `StreamWriter`

a `StreamReader`. Tieto triedy používajú na komunikáciu so zadaným súborom triedu `FileStream`.

Trieda `StreamWriter` slúži na čítanie z textových súborov. Inicializácia prebieha v jej konštruktoze a jej parametrom je súbor otvorený na zápis pomocou triedy `FileStream`.

```
StreamWriter swMeno = new StreamWriter(fsMeno);
```

Príklad: *Práca s textovými súbormi pomocou triedy StreamWriter*

Súbor otvoríme v režime `FileMode.Create`, čiže sa vytvorí nový súbor alebo sa existujúci prepíše. Na otvorený súbor sa nadviaže nová inštancia `StreamWriter`, ktorá nám umožňuje do súboru zapisovať podobne ako do konzoly. Po zápise textu do súboru je potrebné zatvoriť inštanciu `StreamWriter` aj inštanciu `FileStream`.

```
using System;
using System.IO;
namespace CA_StreamWriter
{
    class Program
    {
        static void Main(string[] args)
        {
            FileStream fsZapis = new FileStream("c:\\subor.txt",
                                                FileMode.Create);

            StreamWriter swZapis = new StreamWriter(fsZapis);
            swZapis.WriteLine("Skripta z OOP");
            swZapis.WriteLine("Zapis do suboru pomocou triedy StreamWriter.");
            swZapis.Close();
            fsZapis.Close();
        }
    }
}
```

8.2.4 Práca s text. súbormi – StreamReader

Pomocou triedy `StreamReader` je možné načítať údaje uložené vo zvolenom súbore. Inicializácia a aj spôsob nadviazania spojenia so súborom je podobná ako pri triede `StreamWriter`.

```
StreamReader srMeno = new StreamReader(fsMeno);
```

Je potrebné mať vytvorený `FileStream` na čítanie zo súboru.

Príklad: *Práca s textovými súbormi pomocou triedy StreamReader*

Na čítanie zo súboru je použitý základ predošlého príkladu. Po zápise a zatvorení súboru tento súbor opäť otvoríme, tentokrát vytvoríme novú inštanciu `StreamReader`, ktorá nám umožní zo súboru údaje čítať. Pri čítaní zo súboru postupujeme podobne ako pri načítavaní údajov z klávesnice. Musíme len zabezpečiť cyklický proces čítania a korektné ukončenie po dosiahnutí konca súboru. Ak nie je čo čítať, tak `ReadLine()` vráti `null`.

```
using System;
using System.IO;
namespace CA_StreamWriter
{
    class Program
    {
        static void Main(string[] args)
        {
            FileStream fsZapis = new FileStream("c:\\subor.txt",
                                                FileMode.Create);

            StreamWriter swZapis = new StreamWriter(fsZapis);
            swZapis.WriteLine("Skripta z OOP");
            swZapis.WriteLine("Zapis do suboru pomocou triedy StreamWriter.");
            swZapis.Close();
            fsZapis.Close();

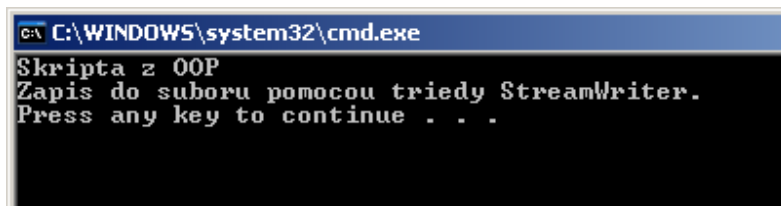
            FileStream fsCitanie = new FileStream("c:\\subor.txt",
                                                  FileMode.Open);

            StreamReader srCitanie = new StreamReader(fsCitanie);
            string Text = null;
```

```

while ((Text = srCitanie.ReadLine()) != null)
    Console.WriteLine(Text);
srCitanie.Close();
fsCitanie.Close();
}
}
}

```



Obr. 32 Práca s textovými súbormi pomocou triedy *StreamReader* – výpis v konzole

8.3 PRÁCA S BINÁRNymi SÚBORMI

Na prácu so súbormi, v ktorých sú údaje uložené v binárnom tvare, je potrebné použiť triedu **BinaryWriter** na zápis a triedu **BinaryReader** na čítanie zo súboru. Podobne ako pri práci s textovými súbormi je aj tomto prípade možné použiť na nadviazanie spojenia so súborom triedu **FileStream**.

```
BinaryWriter brMeno = new BinaryWriter (fsMeno);
```

Trieda **BinaryWriter** resp. **BinaryReader** umožňuje zapisovať alebo načítavať dáta v rôznych formátoch – celé čísla, desatinné čísla, kladné a záporné čísla, textové reťazce...

Binárne súbory uchovávajú dáta v rovnakej forme, ako sú uložené v pamäti, nie sú priamo čitateľné a zaberajú priestor podľa veľkosti konkrétneho dátového typu. Pre správne čítanie z binárneho súboru preto musíme poznať presné poradie a typ hodnôt v ňom uložených.

Príklad: Práca s binárnymi súbormi

V príklade sa do otvoreného súboru pomocou inštancie triedy **BinaryWriter** zapíše postupne celé kladné číslo, reálne číslo, celé záporné číslo a reťazec. Následne po zatvorení a opätovnom otvorení súboru sa pomocou inštancie triedy **BinaryReader** čítajú dáta z tohto súboru. Všimnite si poradie a typ zapisovaných a následne čítaných dát. Ak by sme to nedodrжали, tak načítané hodnoty by sa líšili od zapísaných hodnôt.

```

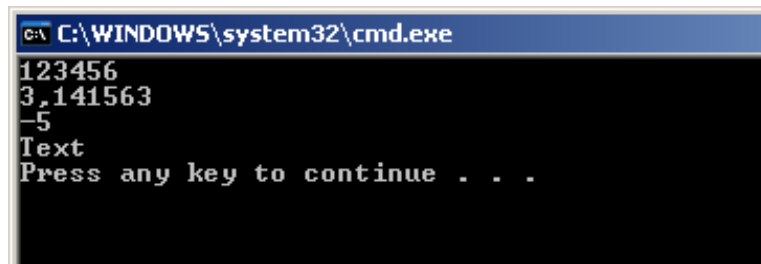
using System;
using System.IO;
namespace CA_Binarne_subory
{
    class Program
    {
        static void Main(string[] args)
        {
            FileStream fsZapis = new FileStream("c:\\subor.dat",
                                                FileMode.Create);

            BinaryWriter bwZapis = new BinaryWriter(fsZapis);
            bwZapis.Write(123456);
            bwZapis.Write(3.1415626f);
            bwZapis.Write(-5);
            bwZapis.Write("Text");
            bwZapis.Close();
            fsZapis.Close();

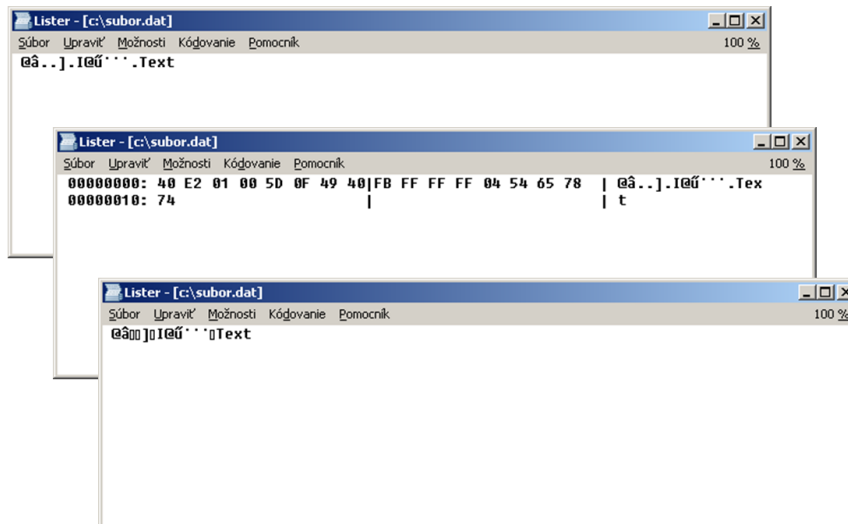
            FileStream fsCitanie = new FileStream("c:\\subor.dat",
                                                  FileMode.Open);

            BinaryReader brCitanie = new BinaryReader(fsCitanie);
            int iCislo1 = brCitanie.ReadInt32();
            Console.WriteLine(iCislo1);
            float fCislo = brCitanie.ReadSingle();
            Console.WriteLine(fCislo);
            int iCislo2 = brCitanie.ReadInt32();
            Console.WriteLine(iCislo2);
            string Text = brCitanie.ReadString();
            Console.WriteLine(Text);
            brCitanie.Close();
            fsCitanie.Close();
        }
    }
}

```



Obr. 33 Práca s binárnymi súbormi – výsledok v konzole



Obr. 34 Práca s binárnymi súbormi – obsah vytvoreného súboru s rôznym kódovaním

8.4 PRESMEROVANIE VSTUPU A VÝSTUPU

Pri práci so súbormi je možné využívať presmerovanie štandardného vstupu, výstupu, prípadne štandardného chybového výstupu pomocou metód triedy `Console`:

- `Console.SetOut(TextWriter)` – slúži na nastavenie štandardného výstupu,
- `Console.SetIn(TextReader)` – slúži na nastavenie štandardného vstupu,
- `Console.SetError(TextWriter)` – umožňuje nastavenie štandardného chybového výstupu.

Na zistenie aktuálneho nastavenia slúžia nasledujúce vlastnosti:

- `Console.Out` – uchováva aktuálne nastavený štandardný výstup,
- `Console.In` – uchováva aktuálne nastavený štandardný vstup,
- `Console.Error` – uchováva aktuálne nastavený štandardný chybový výstup.

Príklad: *Presmerovanie vstupu a výstupu*

Príklad demonštruje presmerovanie všetkých výpisov, ktoré by sa štandardne vypisovali na obrazovku, do textového súboru.

Najskôr je otvorený nový súbor pomocou inštancie `fsZapis` triedy `FileStream`. Do tohto súboru budeme zapisovať pomocou novej inštancie `swZapis` triedy `StreamWriter`. Pre potreby návratu do pôvodného stavu si uložíme aktuálne nastavenie konzolového výstupu `Console.Out` do premennej `twObrazovka`. Následne prestavíme konzolový výstup pomocou metódy `Console.SetOut()`, ktorej parametrom je predtým vytvorená inštancia `swZapis`. Od tohto príkazu bude každý výpis do konzoly presmerovaný do textového súboru.

Na ukončenie presmerovania je potrebné zatvoriť konzolový výstup metódou `Console.Out.Close()`, a tiež inštancie `swZapis` a `fsZapis`. Až potom je možné vrátiť pôvodné nastavenia konzolového výstupu pomocou metódy `Console.SetOut()` a hodnoty uloženej v `twObrazovka`.

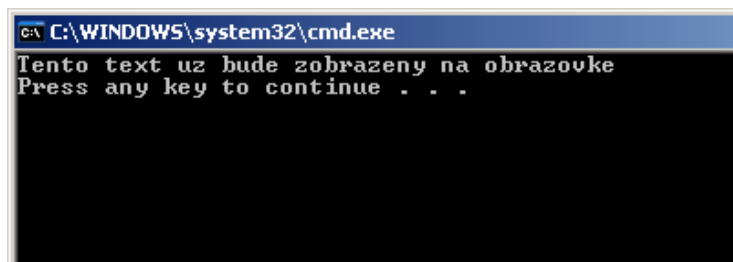
```
using System;
using System.IO;
namespace CA_Presmerovanie_vystupu
{
    class Program
    {
        static void Main(string[] args)
        {
            FileStream fsZapis = new FileStream("c:\\subor.txt",
                                                FileMode.Create);

            StreamWriter swZapis = new StreamWriter(fsZapis);
            TextWriter twObrazovka = Console.Out;

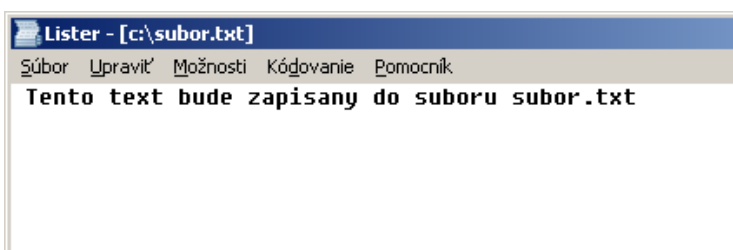
            Console.SetOut(swZapis);
            Console.WriteLine("Tento text bude zapisany do suboru subor.txt");
            Console.Out.Close();
            swZapis.Close();
            fsZapis.Close();

            Console.SetOut(twObrazovka);
            Console.WriteLine("Tento text uz bude zobrazeny na obrazovke");
        }
    }
}
```

```
}  
}  
}
```



Obr. 35 Presmerovanie vstupu a výstupu – výsledok v konzole



Obr. 36 Presmerovanie vstupu a výstupu – obsah vytvoreného súboru

9 WINDOWS FORMS APPLICATION

9.1 CHARAKTERISTIKA WINDOWS FORMS APPLICATION

Umožňuje vytvárať aplikácie prebiehajúce v grafickom operačnom prostredí. Aplikácie využívajú okná a vizuálne a nevizuálne komponenty, ktoré je možné na ne umiestniť. Windows Forms Application je súčasťou .NET Frameworku.

Hlavné komponenty pre prácu obsahuje systémový menný priestor (namespace)

```
System.Windows.Forms
```

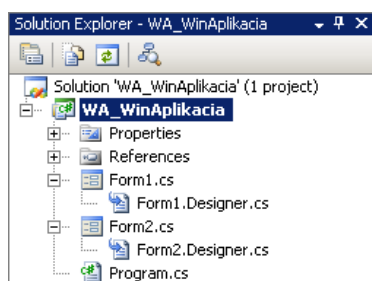
9.1.1 Odlišnosti pri implementácii Windows Forms Application v C++ a C#

Na rozdiel od programovania Win aplikácií v jazyku C++, jazyk C# nevyužíva ako primárny súbor knižníc WinAPI, ale .NET Framework. Nevyužíva sa funkcia WinMain(). Triedu okna nie je potrebné registrovať. Nie je potrebné spravovať front správ. Trieda okna prevedie všetky „svoje“ správy na udalosti, ktoré môže využívať programátor.

9.2 ZÁKLADNÉ PRVKY WINDOWS FORMS APPLICATION

9.2.1 Zloženie projektu

V zdrojovom súbore Program.cs sa nachádza vstupná metóda Main(). V zdrojových súboroch FormX.cs sa nachádzajú metódy určené na prácu s daným formulárom a jeho komponentmi. V zdrojových súboroch FormX.Designer.cs sa nachádzajú metódy určené na nastavenie vlastností formulára a jednotlivých jeho komponentov.



Obr. 37 Zloženie projektu

9.2.2 Trieda aplikácie

Hlavná metóda `static void Main()` obsahuje triedu `Application` a pomocou jej metód je spustená a spravovaná prebiehajúca aplikácia. Základnou metódou je metóda `Run`

```
Application.Run(new Form1());
```

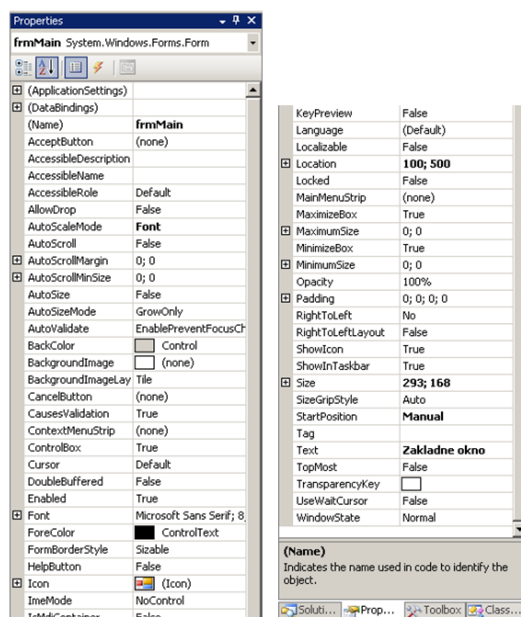
Pomocou nej je aplikácia spustená a obsahuje jedno okno `Form1`. Súčasne je spustená slučka udalostí a aplikácia bude vykonávať obsluhu definovaných udalostí. Pomocou metódy `Application.Exit()` je možné ukončiť slučku udalostí a po jej ukončení ukončiť aj aktuálnu aplikáciu. Ak nastane problém počas priebehu aplikácie, je možné pomocou príkazu `Application.Restart()` zrušiť a znova spustiť aplikáciu.

9.2.3 Okno aplikácie

Trieda `Form` je základná trieda, z ktorej je vytvorený formulár okna. Aplikácia môže zobrazovať okná ako jednoduché samostatné okná alebo okná typu MDI (Multiple–Document Interface). Samostatné okná môžu byť zobrazené ako štandardné okná alebo ako modálne okná. Každé nemodálne okno obsahuje svoju vlastnú slučku udalostí.

Základné vlastnosti triedy Form

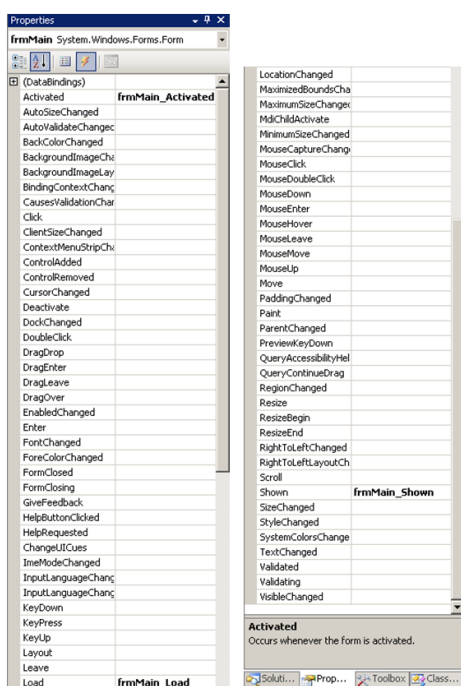
- `Name` – obsahuje hlavné identifikačné meno formulára,
- `Location` – udáva súradnice umiestnenia ľavého horného rohu formulára,
- `StartPosition` – umožňuje nastaviť úvodnú pozíciu formulára (centrované na stred obrazovky, centrované vzhľadom na vlastníka, manual...), ak je zvolená iná možnosť ako `Manual`, má prednosť pred určením pozície vo vlastnosti `Location`,
- `Size` – určuje hlavné rozmery okna (výška a šírka),
- `Text` – obsahuje textový reťazec, ktorý bude zobrazený v záhlaví formulára,
- `IsMdiContainer` – určuje, či ide o jednoduché okno alebo MDI,
- `WindowState` – umožňuje nastaviť základné zobrazenie formulára (normálne, minimalizované, maximalizované),
- `Enabled` – určuje, či je formulár prístupný. Ak nie je prístupný, nie sú prístupné ani objekty umiestnené na ňom.



Obr. 38 Vlastnosti formulára

Základné udalosti triedy Form

- Activated – udalosť vznikne pri každom aktivovaní formulára,
- Shown – udalosť vznikne pri prvom zobrazení formulára,
- Load – udalosť vznikne pri zobrazení formulára,
- Closing – udalosť vznikne pri zatváraní formulára (zatvorenie je ešte možné zrušiť),
- Closed – udalosť vznikne pri definitívnom zatvorení formulára.



Obr. 39 Udalosti formulára

Základné metódy triedy Form

- Show – zobrazí formulár ako jednoduché okno,
- ShowDialog – zobrazí formulár ako modálne okno,
- Activate – aktivuje neaktívny (skrytý) formulár,
- Close – zatvorí formulár,
- Hide – skryje formulár,
- LayoutMdi – ak je typ formulára MDI, usporiada zobrazené formuláre.

Modálne okno

Modálne okná sa používajú na informovanie užívateľa o vykonanej alebo pripravovanej činnosti. Využitím modálneho okna je možné získať odpoveď užívateľa na položenú otázku vzhľadom na vykonávanú činnosť. Modálne okno je umiestnené nad formulár, z ktorého je spustené, a pokiaľ nie je modálne okno zatvorené, nie je možné pracovať s oknom, z ktorého bolo modálne okno spustené. Pri zatvorení modálneho okna je výsledkom jeho činnosti hodnota `DialogResult`, čo je výpočtový typ (enum), ktorý obsahuje nasledujúce hodnoty: OK, No, Yes, Cancel, Abort, Ignore, Retry, None. Hodnoty typu `DialogResult` je možné využívať aj v dialógových oknách typu `MessageBox`.

Príklad: Práca s modálnym oknom

Po vytvorení inštancie dialógového okna `PotvrďKoniec` je toto okno zobrazené zavolaním metódy `ShowDialog()`. Od tohto momentu je zobrazené dialógové okno jediným aktívnym oknom programu, a ten čaká, kým sa dialógové okno nejakým spôsobom nezatvorí. Po zatvorení tohto okna sa vráti výsledok vo forme hodnoty typu `DialogResult` a môže sa vyhodnotiť v podmienke `if`.

```
private void btnKoniec_Click(object sender, EventArgs e)
{
    frmDialog PotvrďKoniec = new frmDialog();

    if (PotvrďKoniec.ShowDialog() == DialogResult.Yes)
        Application.Exit();
}
```

9.2.4 Commons Controls

Na formulár je možné umiestniť vizuálne a nevizuálne komponenty. Medzi vizuálne komponenty patria tlačidlá, textové políčka, rozbaľovacie polia (combo boxy), list boxy, zaškrŕavacie a rádiové tlačidlá, popisové pole, status bar, obrázky, panely, kalendár, záložky, menu, posuvníky, atď. Medzi nevizuálne komponenty patria dátové kontajnery, komponenty na prepojenie s dátovými zdrojmi, časovač, front správ, štandardné dialógové okná – nastavenie farby, písma, vyhľadávanie súborov a pod.

Button (tlačidlo)



Obr. 40 Button (tlačidlo)

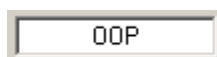
Button – properties

- DialogResult – slúži na priradenie hodnoty DialogResultu pri stlačení tlačidla,
- FlatStyle – určuje grafický vzhľad tlačidla,
- Image – slúži na priradenie obrázka,
- TabIndex – určuje poradie pri prepínaní na formulári pomocou klávesu TAB,
- Text – obsahuje text zobrazený na tlačidle,
- TextAlign – určuje zarovnanie nápisu,
- Visible – slúži na zobrazenie/skrytie tlačidla.

Button – events

- Click – udalosť, ktorá reaguje na kliknutie štandardne ľavým tlačidlom myši,
- MouseClick – udalosť, ktorá reaguje na kliknutie ľubovoľným tlačidlom myši,
- MouseHover – udalosť, ktorá vznikne, ak je kurzor myši určitú dobu nad tlačidlom.

TextBox (textové pole)



Obr. 41 TextBox (textové pole)

TextBox – properties

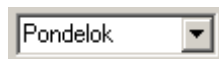
- Lines – slúži na priradenie textového reťazca, ktorý bude zobrazený vo viacerých riadkoch,

- `MaxLength` – určuje maximálny počet zobrazených znakov,
- `MultiLine` – určuje, či bude možné zobraziť text v jednom, alebo vo viacerých riadkoch,
- `PasswordChar` – určuje, aký znak sa bude zobrazovať, ak chceme „skryt“ zadávaný reťazec,
- `ReadOnly` – určuje, či bude možné písať do poľa,
- `Text` – obsahuje text, ktorý bude zobrazený,
- `WordWrap` – automaticky zalomí riadok.

TextBox – events

- `Enter` – udalosť, ktorá vznikne pri vstupe do textového poľa,
- `KeyDown`, `KeyUp`, `KeyPress` – udalosti, ktoré vzniknú, ak je pri písaní do textového poľa stlačené, pustené alebo stlačené a pustené ľubovoľné tlačidlo,
- `TextChanged` – udalosť vznikne, ak je zmenený obsah textového poľa.

ComboBox (rozbaľovacie pole)



Obr. 42 *ComboBox (rozbaľovacie pole)*

ComboBox – properties

- `DataSource` – slúži na priradenie dátového zdroja, z ktorého sa môžu načítavať hodnoty do combo boxu,
- `Items` – obsahuje zoznam hodnôt uložených v combo boxe,
- `MaxDropDownItems` – určuje počet súčasne zobrazených položiek,
- `Sorted` – umožňuje utriedenie hodnôt uložených v combo boxe,
- `Text` – obsahuje aktuálne zobrazenú hodnotu.

ComboBox – events

- `DropDown` – udalosť vznikne pri otvorení zoznamu hodnôt,
- `DropDownClosed` – udalosť vznikne pri zatvorení zoznamu hodnôt,
- `SelectedIndexChanged` – udalosť vznikne, ak sa zmení vybraná hodnota zo zoznamu hodnôt.

CheckBox (zaškrťavacie políčko), RadioButton (rádio políčko)



Obr. 43 *CheckBox (zaškrťavacie políčko), RadioButton (rádio políčko)*

CheckBox, RadioButton – properties

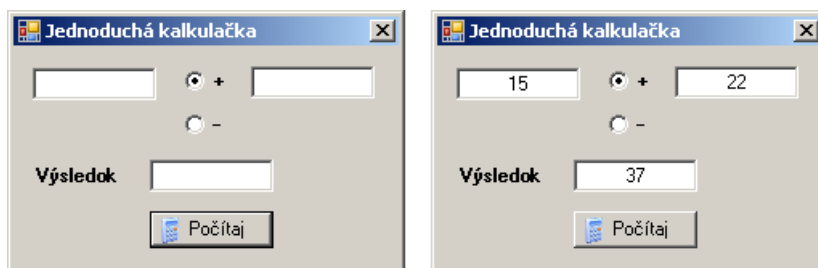
- **CheckAlign** – určuje umiestnenie zaškrťavacieho alebo rádio políčka,
- **Check** – určuje, či je políčko zaškrtnuté, alebo nie,
- **CheckState** (iba CheckBox) – určuje stav zaškrtnutia – zaškrtnuté, odškrtnuté, zaškrtnuté a šedé (indetermina),
- **ThreeState** (iba CheckBox) – umožňuje voľbu medzi dvoma alebo tromi stavmi zaškrťavacieho políčka.

CheckBox, RadioButton – events

- **CheckedChanged** – udalosť pri zmene stavu.

9.3 PRÍKLAD – JEDNODUCHÁ KALKULAČKA

Príklad demonštruje obsluhu niektorých udalostí základných komponentov. Program predstavuje jednoduchú kalkulačku, ktorá z dvoch celých čísel zadanych do vstupných textových polí vypočíta na základe zvolenej operácie ich súčet alebo rozdiel a výsledok zapíše do textového poľa.



Obr. 44 *Okno aplikácie*

Príklad: Základná štruktúra zdrojového kódu okna aplikácie

Základná štruktúra zdrojového kódu okna aplikácie obsahuje metódu `frmMain()`, v ktorej metóda `InitializeComponent()` inicializuje komponenty okna. Ďalej obsahuje tri metódy obsluhy udalostí (`tbCislo1_TextChanged`, `tbCislo1_KeyPress` a `btnPocitaj_Click`), ktoré obsahujú zdrojový kód reagujúci na dané udalosti. Ich podrobnejší opis je uvedený v ďalšom texte.

```

using System;
using System.Windows.Forms;
namespace WA_Okno
{
    public partial class frmMain : Form
    {
        public frmMain()
        {
            InitializeComponent();

            // Zdrojový kód udalosti TextChanged textového políčka tbCislo1
            private void tbCislo1_TextChanged(object sender, EventArgs e)
            {...}

            // Zdrojový kód udalosti KeyPress textového políčka tbCislo1
            private void tbCislo1_KeyPress(object sender, KeyPressEventArgs e)
            {...}

            // Zdrojový kód udalosti Click tlačidla btnPocitaj
            private void btnPocitaj_Click(object sender, EventArgs e)
            {...}

        }
    }
}

```

Príklad: Zdrojový kód udalosti *TextChanged* textového políčka *tbCislo1*

Pri vyvolaní udalosti *TextChanged* sú spracované zadané reťazce z oboch textových polí. Ak textové polia obsahujú nejaký text (v udalosti *tbCislo1_KeyPress* je zabezpečené, že vložený text bude obsahovať iba čísla a nie písmená), je tento text prevedený pomocou metódy *Int32.Parse()* na celé číslo. Na základe vybranej operácie sú tieto čísla sčítané alebo odčítané a následne pomocou metódy *ToString()* prevedené na textový reťazec. Výsledný textový reťazec je uložený do vlastnosti *Text* textového poľa *tbVysledok* slúžiaceho na zobrazenie výsledku. Keďže je tento zdrojový kód uložený v udalosti *TextChanged*, dôjde automaticky k prepočítaniu výsledku počas písania do textových polí.


```

private void tbCislo1_TextChanged(object sender, EventArgs e)
{
    if ((tbCislo1.Text != "") && (tbCislo2.Text != ""))
    {
        if (rbPlus.Checked)
            tbVysledok.Text =
                (Int32.Parse(tbCislo1.Text) +
                 Int32.Parse(tbCislo2.Text)).ToString();
        else if (rbMinus.Checked)
            tbVysledok.Text =
                Convert.ToString(Int32.Parse(tbCislo1.Text) -
                                 Int32.Parse(tbCislo2.Text));
    }
    else
        tbVysledok.Clear();
}

```

Príklad: Zdrojový kód udalosti *KeyPress* textového políčka *tbCislo1*

Udalosť *KeyPress* v textovom poli je vyvolaná pri stlačení akéhokoľvek klávesu, keď je kurzor umiestnený v tomto poli. Slúži napr. na overenie, či používateľ zadáva iba číselné hodnoty, v opačnom prípade sa kláves ignoruje. Táto udalosť je spracovávaná ešte pred udalosťou *TextChanged*, takže napr. pri správnom ošetroení dokáže zabrániť zadávaniu nepovolených znakov do textového poľa.

V tomto prípade sú povolenými znakmi iba čísla a kláves *BackSpace*, ktorý slúži na mazanie čísel v textovom poli.

```

private void tbCislo1_KeyPress(object sender, KeyPressEventArgs e)
{
    if (!Char.IsDigit(e.KeyChar) && e.KeyChar != (char)8)
        e.Handled=true;
}

```

Príklad: Zdrojový kód udalosti *Click* tlačidla *btnPocitaj*

Vyvolanie udalosti *Click* je spôsobené kliknutím na príslušné tlačidlo. Spracovanie dát a výpis výsledku v tomto prípade prebieha rovnako ako pri udalosti *TextChanged* v textovom poli. Rozdiel je v tom, kedy je jedna alebo druhá udalosť vyvolávaná. Udalosť *TextChanged*

je vyvolaná zmenou hodnoty v textovom poli a udalosť Click je vyvolaná kliknutím na príslušné tlačidlo.

```
private void btnPocitaj_Click(object sender, EventArgs e)
{
    if ((tbCislo1.Text != "") && (tbCislo2.Text != ""))
    {
        if (rbPlus.Checked)
            tbVysledok.Text =
                (Int32.Parse(tbCislo1.Text) +
                 Int32.Parse(tbCislo2.Text)).ToString();
        else if (rbMinus.Checked)
            tbVysledok.Text =
                Convert.ToString(Int32.Parse(tbCislo1.Text) -
                                 Int32.Parse(tbCislo2.Text));
    }
    else
        tbVysledok.Clear();
}
```

ZÁVER

V predložených skriptách sú zhrnuté možnosti objektovo orientovaného programovania a možnosti jeho aplikácie v programovacom jazyku C#, ktorý je súčasťou platformy Microsoft .NET Framework. Skriptá poskytujú informácie študentom študujúcim predmet OBJEKTOVO ORIENTOvané PROGRAMOVANIE a aj ďalším záujemcom, ktorí sa chcú oboznámiť s problematikou objektovo orientovaného programovania a s programovacím jazykom C#.

Obsahujú materiál zaoberajúci sa jednotlivými časťami objektovo orientovaného programovania a opis základných vlastností programovacieho jazyka C#, jeho dátových a riadiacich štruktúr. Skriptá sú doplnené množstvom príkladov, na ktorých je názorne vysvetlená prezentovaná problematika.

ZOZNAM BIBLIOGRAFICKÝCH ODKAZOV

1. ALBAHARI, Joseph, ALBAHARI, Ben: *C# 5.0 in a Nutshell: The Definitive Reference*. Sebastopol, Canada: O'Reilly Media, 2012. ISBN 978-1-449-32010-2
2. ECMA International. Standard ECMA-334 –C# Language Specification. 4th edition (June 2006). ECMA International, 2006 [cit. 17.7.2012]. Dostupné na internete: <http://www.ecma-international.org/publications/standards/Ecma-334.htm>
3. DotGNU Project – GNU Freedom for the Net. 2012 [cit. 17.7.2012]. Dostupné na internete: <http://www.dotgnu.org/pnet.html>
4. HANÁK, Ján. *Objektovo orientované programovanie v jazyku C# 3.0*. Brno: Microsoft, 2008. ISBN 978-80-87017-02-9
5. KEBÍSEK, Michal. *Objektovo orientované programovanie*. Trnava: Qintec, 2010. ISBN 978-8-096-98468-8
6. KENT, Jeff. *Visual C# 2005 bez předchozích znalostí. Průvodce pro samouky*. Brno: Computer Press, 2007. ISBN 987-80-251-1574-8
7. MAYO, Joe. *C# Unleashed. With the .NET Framework 3.5. Second edition*. Indianapolis: SAMS, 2008. ISBN 978-0-672-32981-4
8. Microsoft. MSDN – Microsoft Developer Network. Microsoft, 2012. [cit. 17.7.2012]. Dostupné na internete: www.msdn.microsoft.com
9. MONO – Cross platform, open source .NET development framework. 2012. [cit. 17.7.2012]. Dostupné na internete: http://www.mono-project.com/Main_Page
10. MonoDevelop – MonoDevelop. 2012. [cit. 17.7.2012]. Dostupné na internete: <http://monodevelop.com/>
11. NAGEL, Christian, EVJEN, Bill, GLYNN, Jay, SKINNER, Morgan, WATSON, Karli. *Professional C# 2008*. Indianapolis: Wiley Publishing, 2008. ISBN 978-0-470-19137-8
12. NASH, Trey. *C# 2010*. Brno, ČR, Computer Press, 2010. ISBN 978-80-251-3034-6
13. PECINOVSKÝ, Rudolf. Objektově orientované programování? Co to je? In. *Počítač ve škole 2009*. Nové Město na Moravě: GVM, roč. 6, 2009. ISBN 978-80-254-3995-1
14. POKORNÝ, Jan. *Úvod do .NET Framework*. Brno: Mobil Media, 2002. ISBN 80-86593-16-9
15. ROBINSON, Simon, ALLEN, Scott. *C# Programujeme profesionálně*. Brno: Computer Press, 2003. ISBN 80-251-0085-5
16. SHARP, John, JAGGER Jon. *Microsoft Visual C# .NET krok za krokem*. Praha: Microsoft Press, 2002. ISBN 80-865-9327-4
17. SPAANJAARS, Imar. *Beginning ASP.NET 4 in C# and VB*. Indianapolis: Wiley Publishing, 2010. ISBN 978-0-470-50221-1
18. TROELSEN, Andrew. *Pro C# 2010 and the .NET 4 Platform*. New York: Apress, 2010. ISBN 978-1-430-22549-2
19. TROELSEN, Andrew. *Pro C# 5.0 and the .NET 4.5 Framework*. New York: Apress, 2012. ISBN 978-1-430-24233-8
20. VALÁŠEK, Michal: ASPNET.CZ – in technology we trust. 2011. ISSN: 1801-9447. [cit. 17.7.2012]. Dostupné na internete: <http://www.aspnet.cz/>

21. VIRIUS, Miroslav. *C# pro zelenáče*. Praha: Neocortex spol. s r. o., 2002. ISBN 80-86330-11-7
22. VIRIUS, Miroslav. *Od C k C++*, České Budějovice: Kopp, 2004. ISBN 80-723-2110-5
23. VIRIUS, Miroslav. *Od C++ k C#*, České Budějovice: Kopp, 2002. ISBN 80-723-2176-5

OBSAH

Zoznam skratiek	3
Úvod.....	4
1 Základy objektovo orientovaného programovania.....	5
1.1 Základné pojmy a ciele OOP	6
1.1.1 Základné pojmy.....	7
1.1.2 Základné ciele OOP	7
1.2 Základné črty OOP	8
1.2.1 Zapuzdrenie.....	8
1.2.2 Dedičnosť	9
1.2.3 Polymorfizmus	10
1.2.4 Abstraktná trieda a metóda.....	11
2 Objektovo orientované programovanie a programovací jazyk C#	12
2.1 Charakteristika .NET Frameworku	12
2.1.1 Základné prvky .NET Frameworku.....	14
2.1.2 Spoločný typový priestor – Common Type System.....	18
2.1.3 Verzie .NET Frameworku	19
2.1.4 Verzie programovacieho jazyka C#	22
2.2 Charakteristika jazyka C#	23
3 Dátové typy	25
3.1 Definícia premenných a konštánt.....	26
3.2 Konverzia dátových typov	29
4 Riadiace štruktúry	37
4.1 Operátory	37
4.1.1 Aritmetické operátory	38
4.1.2 Inkrement a dekrement.....	38
4.1.3 Priradenie a kombinované priradenie.....	39
4.1.4 Porovnávacie operátory.....	39
4.1.5 Logické operátory	39
4.2 Príkazy skokov.....	40
4.3 Príkazy vetvení.....	41
4.4 Príkazy cyklov	43
4.5 Pomocné riadiace štruktúry.....	46
5 Triedy a objekty v C#.....	48
5.1 OOP v C# – základné rozdiely oproti OOP v C++	48
5.2 Trieda v C#	49
5.3 Prístupové práva.....	49
5.4 Členské premenné a metódy triedy.....	50
5.5 Trieda a inštancia v jazyku C#.....	51
5.6 Konštruktor triedy	54
5.7 Deštruktor triedy	57
5.8 Implementácia dedičnosti v triede	60
5.8.1 Konštruktory a deštruktory pri dedičnosti.....	61

5.8.2	Metódy triedy pri dedičnosti	63
5.8.3	Rozdiely v dedičnosti v C++ a C#	66
5.9	Abstraktná a zapečatená trieda.....	66
5.10	Rozhrania tried (Interfaces)	67
5.11	Vlastnosti (Properties)	68
5.12	Delegáty (Delegates).....	71
5.13	Udalosti (Events)	72
6	Výnimky	76
6.1	Charakteristika a definícia výnimiek	76
6.1.1	Konštrukcie príkazu try.....	76
6.2	Šírenie výnimiek	78
6.3	Triedy pre výnimky.....	80
6.4	Použitie ošetrenia výnimiek	82
7	Polia.....	83
7.1	Jednorozmerné polia	83
7.1.1	Kopírovanie jednorozmerných polí.....	85
7.1.2	Vlastnosti a metódy triedy System.Array.....	86
7.2	Viacrozmerné pravidelné polia	88
7.2.1	Vlastnosti a metódy triedy System.Array.....	89
7.3	Viacrozmerné nepravidelné (zubaté) polia	90
8	Práca s adresármi a so súbormi.....	93
8.1	Práca s adresármi	93
8.2	Práca s textovými súbormi.....	94
8.2.1	Práca s textovými súbormi – FileInfo	95
8.2.2	Práca s textovými súbormi – FileStream.....	95
8.2.3	Práca s text. súbormi – StreamWriter.....	97
8.2.4	Práca s text. súbormi – StreamReader.....	99
8.3	Práca s binárnymi súbormi.....	100
8.4	Presmerovanie vstupu a výstupu.....	102
9	Windows Forms Application.....	105
9.1	Charakteristika Windows Forms Application.....	105
9.1.1	Odlišnosti pri implementácii Windows Forms Application v C++ a C#	105
9.2	Základné prvky Windows Forms Application	105
9.2.1	Zloženie projektu.....	105
9.2.2	Trieda aplikácie	106
9.2.3	Okno aplikácie.....	106
9.2.4	Commons Controls.....	109
9.3	Príklad – Jednoduchá kalkulačka.....	111
	Záver	115
	Zoznam bibliografických odkazov	116

Autori: Doc. Ing. Peter Schreiber, CSc., Ing. Michal Kebísek, PhD.,
Ing. Michal Eliáš, PhD.
Názov: Objektovo orientované programovanie
Object Oriented Programming
Miesto vydania: Trnava
Vydavateľ: AlumniPress
Rok vydania: 2013
Vydanie: prvé
Rozsah : 119 strán
Edičné číslo: 3/AP/2013

ISBN 978-80-8096-185-5
EAN 9788080961855