

Pro verze
2007
2010

Martin Král

Excel VBA

Výukový kurz

Od základů po programování
procedur a vlastních tříd

Ovládací prvky, práce s daty
na listu, formuláře

Shrnutí výkladu,
otázky a odpovědi



CD obsahuje

Kódy všech příkladů z knihy
Užitečné soubory od předních vývojářů

C PRESS

Martin Král

Excel VBA

Výukový kurz

Computer Press
Brno
2012

Excel VBA

Výukový kurz

Martin Král

Obálka: Martin Sodomka

Odpovědný redaktor: Libor Pácl

Technický redaktor: Jiří Matoušek

Objednávky knih:

<http://knihy.cpress.cz>

www.albatrosmedia.cz

eshop@albatrosmedia.cz

bezplatná linka 800 555 513

ISBN 978-80-251-2358-4

Vydalo nakladatelství Computer Press v Brně roku 2012 ve společnosti Albatros Media a. s. se sídlem Na Pankráci 30, Praha 4. Číslo publikace 16 061.

© Albatros Media a. s. Všechna práva vyhrazena. Žádná část této publikace nesmí být kopírována a rozmnožována za účelem rozšiřování v jakékoli formě či jakýmkoli způsobem bez písemného souhlasu vydavatele.

Dotisk 1. vydání.

 **ALBATROS** MEDIA a.s.

Obsah

ÚVOD	11
Komu je kniha určena	11
Co je to VBA a jak se liší od VB	12
VBA pomáhá šetřit čas	14
Jak používat tuto knihu a vůbec, jak se učit VBA.....	16
Otázky k částem lekcí.....	17
Kód není vždy nejefektivnější řešení	17
Shrnutí	20

Lekce 1

Základy jazyka VBA.....	21
Proměnné	22
Nejdůležitější typy proměnných a kdy je použít	23
Proměnná typu Boolean	23
Proměnná typu Long.....	23
Proměnná typu String.....	24
Proměnná typu Double a Currency	25
Proměnná typu Object	27
Proměnná typu Variant.....	28
Pole a jejich využití.....	29
Vlastní kolekce a jejich využití	34
Deklarace, rozsah a platnost proměnných.....	35
Otázky	37
Důležité konstrukce VBA použité v knize	38
With ... End With	38
Blok If ... Else ... End If.....	39
Blok Select Case	41
For ... Next.....	44
For Each ... Next.....	47
Do ... Loop.....	49
Ošetření chyb.....	52
Objekt Err	60

Rozsah platnosti ovladače chyb	60
On Error Goto <návěští>	61
On Error Resume Next	61
Výraz Resume	63
Výraz Resume Next	65
Shrnutí	66
Otázky	66
Text ve VBA.....	67
Jak VBA ukládá text.....	67
Funkce porovnávající text	75
Velká a malá písmena	81
Formátování textu	89
Funkce Space.....	89
Funkce String.....	90
Funkce Format	91
Hledání znaků v řetězci	94
Nalezení délky řetězce – Funkce Len.....	94
Hledání znaků v řetězci – Funkce InStr a InStrRev.....	95
Funkce, vracující upravený řetězec	98
Funkce Left a Right.....	98
Funkce Mid.....	99
Funkce Trim, LTrim a RTrim	102
Funkce Replace	104
Výrazy Mid, LSet a RSet.....	105
Funkce Split, Join a Filter.....	108
Export dat z listu do textového souboru	113
Načtení textového souboru do listu.....	119
Otázky.....	123
Čísla ve VBA.....	124
Jak VBA ukládá čísla.....	124
Formátování čísel ve VBA.....	127
Základní matematické funkce ve VBA	132
Vlastní funkce ve VBA.....	136
Otázky.....	138
Datum ve VBA	139
Jak Excel a VBA ukládá datum a čas.....	139
Formátování data a času.....	142
Získání informace z data	145
Stanovení určitého data.....	146

Datové výpočty	148
Otázky	153

Lekce 2

Základy objektového modelu Microsoft Excel 155

Objekt Application	157
Application.ScreenUpdating	159
Application.DisplayAlerts	160
Application.EnableEvents	160
Application.StatusBar	162
Application.InputBox	162
Application.GoTo	164
Application.OnTime	165
Application.OnKey	167
Application.Run	168
Application.WorksheetFunction	168
Application.Evaluate	169
Application.SendKeys	171
Application.Caller	172
Application.Volatile	172
Otázky	173
Objekty Workbook a Worksheet	174
Kolekce Workbooks	174
Otevírání a ukládání sešitů	176
Kolekce Sheets a Worksheets	180
Kopírování a přesunování listů	183
Objekt Window	185
Otázky	186
Objekt Range	187
Definice objektu Range pomocí vlastnosti Range	187
Metody Union a Intersect	188
Definice objektu Range pomocí vlastnosti Cells	191
Vlastnosti Offset a Resize	193
Otázky	195
Grafy	196
Objekt Chart	197
Využití záznamníku maker pro tvorbu grafu	199
Vytvoření listu s grafem ve VBA	202
Vytvoření grafu na listu ve VBA	203

Editace datové řady ve VBA	205
Úpravy vzhledu grafu ve VBA.....	206
Skupina Typ.....	207
Skupina Data	208
Skupina Rozložení grafu.....	208
Skupina Umístění.....	210
Karta Rozložení	210
Objekt ChartFormat	216
Přidání vlastních popisků dat.....	217
Otázky	219
Kontingenční tabulky.....	220
Vytvoření kontingenční tabulky přes uživatelské rozhraní.....	220
Vytvoření kontingenční tabulky v kódu VBA.....	222
Přístup k prvkům kontingenční tabulky.....	225
Otázky	227

Lekce 3

Uživatelské funkce (UDF) 229

Procedury a funkce	230
Použití vlastních funkcí jako vzorců v listu	236
Příklady uživatelských funkcí.....	239
Funkce vracející výši progresivní daně	239
Funkce sčítající barvy	240
Funkce vracející údaje o souboru a uživateli.....	241
Funkce vracející některá nastavení Windows.....	245
Použití vlastností Application.Volatile a Application.Caller	247
Otázky	248

Lekce 4

Programování ovládacích prvků 251

Rozdíl mezi ovládacími prvky typu ActiveX a Forms.....	252
Otázky	255
Nejpoužívanější ovládací prvky.....	256
Příkazové tlačítko (Commandbutton).....	256
Pole se seznamem (ComboBox).....	257
Zaškrťávací políčko (CheckBox).....	259
Přepínač (OptionButton, Radio Button)	260
Seznam (ListBox).....	261
Posuvník (Scrollbar)	263

Číselník (Spin Button).....	264
Otázky	265
Jak přistupovat k ovládacím prvkům na listu přes VBA	266
Otázky	274

Lekce 5

Programování událostních procedur a vlastních tříd275

Události listu.....	276
Událost Worksheet_Change.....	276
Událost Worksheet_Calculate.....	277
Událost Worksheet_BeforeRightClick	278
Událost Worksheet_BeforeDoubleClick.....	278
Události sešitu.....	278
Událost Workbook_Open.....	279
Událost Workbook_BeforeClose	279
Událost Workbook_BeforeSave.....	280
Událost Workbook_BeforePrint.....	280
Události listu na úrovni sešitu.....	282
Otázky	282
Vytváření vlastních tříd objektů	283
Terminologie tříd.....	283
Vytvoření vlastní třídy pro výpočet hypotéky.....	284
Využití tříd k zapouzdření komplexních API funkcí	287
Jediná procedura pro více ovládacích prvků	288
Otázky	290
Využití tříd k zachycování událostí aplikace	290
Využití tříd k zachycování událostí vloženého grafu	292
Využití tříd k zachycování událostí objektu QueryTable	294
Otázky	295

Lekce 6

Úpravy a psaní kódu297

Základní pravidla navrhování aplikací v Excelu.....	298
Struktura sešitu.....	298
Návrh listu	299
Jak psát robustní kód	302
Otázky	307
Práce s editorem VBA	307
Prostředí editoru VBA.....	308

Pomoc při psaní kódu	309
Nástroje pro ladění kódu	311
Úprava nahraného kódu	315
Otázky	318
Notace R1C1	319
Otázky	322
Psaní rychlého kódu	323
Co zpomaluje vaši aplikaci	323
Optimalizace příkazů VBA	325
Objektové proměnné	325
Funkce Len()	327
Konstanta vbNullString	327
Spojování řetězců	328
Výraz Mid\$	328
IsCharAlphaNumeric	328
Funkce StrComp	329
Použití operátoru Like	329
Správný typ proměnné	329
Celočíselné dělení	330
Logické přiřazení	330
Použití Not jako přepínače	330
For... Next má přednost před Do... Loop	331
Funkce If	331
Volání DoEvents	332
Vhodné použití For Each... Next a For... Next	332
Vymazání prvků kolekce	332
Otázky	333
Efektivní kontrola jiných aplikací	333
Definice odkazů na jiné aplikace	334
Vytváření nové instance objektu	334
Odkaz na existující instanci	337
Časná a pozdní vazba	339
Otázky	342

Lekce 7

Práce s daty na listu..... 343

Odkaz na oblast v listu	344
Poslední řádek sloupce	344
Poslední sloupec v řádku	346

První zaplněná buňka v řádku	346
Použitá oblast.....	348
Prázdné buňky	351
Buňky s určitým obsahem	353
Shrnutí	353
Otázky	354
Práce s vybranými řádky	354
Cykly jsou neefektivní.....	354
Vymazání prázdných řádků.....	355
Práce s řádky splňujícími podmínku.....	356
Kopírování řádků podle seznamu podmínek.....	365
Shrnutí	374
Otázky	374
Transformace dat	375
Převod dat do databázové podoby	375
Hromadné zpracování dat.....	382
Řazení dat.....	390
Otázky	394
Konsolidace dat z několika sešitů.....	394
Obecný algoritmus konsolidace	395
Kopírování dat do jednoho sešitu	398
Otázky	409
Názvy ve VBA.....	409
Použití pojmenovaných vzorců ve VBA.....	410
Otázky	414

Lekce 8

Formulář pro zadávání dat 415

Základní pravidla návrhu formuláře	416
Přidání formuláře do projektu	417
Zobrazení a uzavření formuláře	418
Přidání ovládacích prvků pro datová pole	419
Inicializace formuláře ve VBA	421
Nahrání záznamu do formuláře a uložení změn	423
Zadávání data pomocí Calendar ActiveX kontrol.....	425
Doplnění dalších navigačních prvků.....	428
Otázky	435

ZÁVĚR..... 437

Co jste se v knize naučili 438

Co v knize naopak nebylo 438

Příloha**Řešení k otázkám..... 441**

Proměnné 442

Konstrukce jazyka VBA 444

Text ve VBA..... 445

Čísla ve VBA..... 448

Datum ve VBA 449

Objekt Application 450

Kolekce Workbooks a Sheets..... 453

Objekt Range 456

Grafy 458

Kontingenční tabulky..... 460

Uživatelské funkce (UDF) 463

Ovládací prvky na listu 465

Ovládací prvky ActiveX 466

Přístup k prvkům na listu přes VBA 469

Události objektů Excelu..... 470

Vytváření vlastních tříd..... 472

Události objektů Application a Chart..... 473

Návrh aplikace 474

Úpravy nahraného kódu 476

Notace R1C1 477

Optimalizace kódu VBA 478

Automatizace jiných aplikací..... 480

Vyhledávání oblastí na listu..... 482

Vyhledání řádků podle podmínky..... 484

Transformace dat..... 487

Konsolidace dat z více sešitů..... 488

Názvy ve VBA..... 490

Uživatelský formulář..... 492

Rejstřík 495

ÚVOD

Komu je kniha určena

Tato kniha je především o *času*, kterého nikdo z nás nemá nazbyt. Možná jste manažer, který si chce firemní data pravidelně analyzovat sám, aby si byl jistý, že mu údaje předkládané oddělením controllingu nezkrusují skutečnost. Možná jste pracovník oddělení controllingu a každý měsíc do pátého pracovního dne musíte zkonsolidovat tuny dat a zaplnit standardní tabulku definovanou kdesi v mateřské firmě kýmsi, kdo nemá ponětí o databázovém uspořádání dat. Možná jste pracovník IT oddělení, expert na databáze a všechny možné dialekty SQL, ale váš šéf jednoduše vyžaduje reporty v Excelu. Jako databázový expert a zkušený programátor v SQL a ASP.NET považujete něco takového pod vaši úroveň, nicméně šéf je šéf a chce mít svou zprávu co nejdřív.

Pro vás všechny je tato kniha. Naučí vás, jak vám VBA může ušetřit čas, a rovněž vám ukáže, jak ušetřit čas při samotném psaní kódu. Když hovoříme o kódu, rozumíme tím program, v tomto případě ve VBA. V praxi je možno se často setkat s výrazem „makro“. Toto slovo v této knize nebudu pro program používat, protože se jedná o slovo matoucí. Slovo „Makro“ v Excelu pochází z dob, kdy Excel používal takzvaný XLM jazyk (neplést s XML). Velmi neintuitivní syntaxe byla postupně vytlačena nástupem VBA. Makro příkazy XML mají však své výhody. Ukazuje se, že díky tomu, že jsou přímo součástí jádra Excelu, vykonávají příkazy mnohem rychleji než jejich ekvivalenty v VBA. Samozřejmě, většinou mluvíme v milisekundách, ale co když příkazem zpracováváme jednotlivě 20 000 řádek? A co když chceme dávkově zpracovat 100 takových souborů čekajících na nás ve složce „KeZpracovani“?

Asi vás příliš nepřekvapím, když řeknu, že efektivní kód znamená obvykle kratší kód, čímž ušetříte i čas na samotné programování. Proto, až se naučíte psát efektivní programy, zabijete dvě mouchy jednou ranou. Ušetříte čas jak při vývoji, tak později díky využívání vaší báječné aplikace.

Kniha není primárně určena pro začátečníky v programování, neboť poměrně do hloubky probírá základy VBA a nejpoužívanější části objektového modelu Excelu. V knize je velké množství konkrétních ukázek a řešení běžných problémů, se kterými jsem se setkal ve vlastní praxi a které rovněž vedou k častým dotazům na internetových fórech. Vysvětlení, jak kód pracuje, se spíše týká použitého algoritmu a pastí, než toho, jak ta která klauzule, funkce, metoda nebo vlastnost pracuje. Budu se snažit vysvětlit rovněž netypické a netradiční použití jinak standardních metod, které jsem ve velké míře odkoukal od špičkových mágů ze světa Excelu a v menší míře na ně přišel sám.

Přestože je kniha určena především pro pokročilé uživatele Excelu 2007, většina kódu prošla testem na Excel 2007 CZ, Excel 2007 ENG, Excel 2003 ENG a většina i na Excel 2000 ENG. Operační systémy byly Windows Vista i Windows XP SP2 v anglické verzi s českým místním nastavením. Při extrémnějších pokusech s mezinárodním prostředím jsem používal Windows XP RU a rovněž

Excel v ruské lokalizaci. Pokud bude některý kód nekompatibilní s verzemi nižšími než Excel 2007, výslovně na to upozorním.

Některým oblastem jsem úmyslně věnoval menší místo, než by možná bylo v knize o VBA obvyklé. Jedná se především o uživatelské formuláře. Přestože jim v této knize věnuji celou jednu lekci, ve vhodně navržené aplikaci jsou zřídka kdy potřeba, protože list v Excelu je vlastně výkonný a flexibilní formulář, za který můžeme schovat libovolný kód. Naopak, podle mé zkušenosti neustále vyskakující okna a formuláře vyžadující reakci uživatele spíše obtěžují. Objektový model Excelu je velmi rozvětvený a s každou novou verzí se dále rozšiřuje. V současné době se rozšiřování týká zejména propojení na vnější databáze, těsnější integrace XML a podobně. Tradiční funkce Excelu se od verze 2000 příliš nemění a výraznějšími změnami prošel zejména objektový model grafiky a jejího formátování. Podstatně rozšířen ve verzi 2007 byl i podmíněný formát.

Další „opomenutou“ oblastí je ovládání vestavěného menu Excelu. Za prvé, Excel 2007 menu zásadně změnil a dosavadní objektový model pro něj neplatí, čímž se vaše aplikace stává nekompatibilní směrem dolů. A za druhé, zvládnutí úplného programování nového menu předpokládá znalost jazyka XML a to už je opravdu mimo rámec této knihy.

Obecně se tato kniha drží spíše hesla „Dej člověku rybu, a nakrmíš ho na jeden den. Nauč jej rybařit, a nakrmíš jej na celý život.“ Je prakticky nemožné a hlavně zbytečné snažit se popisovat práci se všemi objekty Excelu do detailů. Autor samozřejmě taky nezná nazpaměť všechny objekty a už vůbec ne jejich metody a vlastnosti. Vtip je v tom, umět méně používané prvky nalézt a správně a hlavně efektivně je použít ve svém kódu. A i o tom je tato kniha, zejména její šestá lekce.

Co je to VBA a jak se liší od VB

Visual Basic (VB) a Visual Basic for Applications (VBA) mají společný základ. Lze říci, že jádro jazyka Visual Basic 6.0 je součástí instalace Microsoft Office (od verze 9.0, neboli 2000) a komunikace s ním probíhá přes moduly kódu a uživatelské formuláře, které se od formulářů Visual Basic liší svou určitou omezeností, avšak pro většinu aplikací postačí.

Pokud mluvíme o VBA, nemluvíme o Excel VBA, Word VBA, nebo Access VBA. Syntaxe jazyka je pro všechny aplikace stejná, liší se pouze *objektový model* aplikace, se kterou pracujeme. Pokud se naučíte syntaxi jazyka VBA a programování aplikace Excel, je relativně jednoduché nabyté znalosti použít na jiné aplikace, které VBA rovněž používají. Jediné, co se musíte naučit, je objektový model aplikace, kterou chcete automatizovat.

Podívejte se na následující ukázky jednoduchého kódu, které vykonávají velmi podobnou činnost, totiž zapisují sekvenci čísel 1 až 10 do nové tabulky Excelu, do nového dokumentu Wordu a konečně do existující tabulky v Accessu.

Microsoft Excel:

```
'// Kód zapíše čísla 1 až 10 do oblasti A1:A10  
'// listu "Sheet1" nového sešitu v Excelu
```

```
Const MAX As Long = 10  
Dim i As Long
```

```
With Workbooks.Add
    For i = 1 To MAX
        Sheets(1).Cells(i, "A") = i
    Next i
End With
```

Microsoft Word:

```
'// Kód zapíše čísla 1 až 10, každé do nového odstavce,
'// do nového dokumentu ve Wordu
```

```
Const MAX As Long = 10
Dim i As Long
```

```
Documents.Add
With Application
    For i = 1 To MAX
        .Selection.TypeText Text:=CStr(i)
        .Selection.TypeParagraph
    Next i
End With
```

Microsoft Access:

```
'// Kód přidá nové záznamy do tabulky "tblMyTable" v databázi Access
'// a zapíše čísla 1 až 10 do pole "MyField"
```

```
Const MAX As Long = 10
Dim i As Long
Dim rs As DAO.Recordset
Set rs = CurrentDb.OpenRecordset("tblMyTable")
```

```
With rs
    For i = 1 To MAX
        .AddNew
        .Fields("MyField").Value = i
        .Update
    Next i
End With
```

```
rs.Close
Set rs = Nothing
```

Objektový model aplikace se skládá z objektů, jejich metod a vlastností. S objekty manipulujeme metodami a jejich charakteristiky stanovujeme a zjišťujeme přes jejich vlastnosti. V běžném jazyce řekneme:

„Natankujeme do automobilu naftu, poté ho nastartujeme, a jestli motor běží, rozjedeme se.“ V pseudokódu by tento proces vypadal takto:

```
Automobil.Natankovat Palivo:=Nafta
Automobil.Nastartovat
If Automobil.Motor.Bezi Then
```

```
...Automobil.RozjetSe  
End If
```

Podobně, pokud chcete „nastartovat“ aplikaci Excel například z aplikace Word, spusíte Word, stisknete Alt+F11, vložíte nový modul, napíšete tento kód, umístíte kurzor kamkoli dovnitř kódu a stisknete F5.

```
Sub NastartovatExcel()  
Dim oAppXL As Object  
Set oAppXL = CreateObject("Excel.Application")  
If Not oAppXL Is Nothing Then  
    oAppXL.Workbooks.Add  
    oAppXL.ActiveWorkbook.Sheets(1).Range("A1") = "MS Word"  
    oAppXL.Visible = True  
End If  
End Sub
```

Nevadí, jestli nevíte, jak tento kód pracuje. Vše se dozvíte v dalších částech knihy. Pro tento moment je důležité, abyste si uvědomili, že je jen jeden jazyk VBA, který vychází z jazyka Visual Basic. Co se liší, jsou objekty dané aplikací, jejich metody a vlastnosti.

VBA pomáhá šetřit čas

Toto není vymyšlený příběh.

Představte si, že jste byli jmenováni manažerem v nové firmě, která provozuje desítky obchodů, obsluhuje desetitisíce zákazníků denně a prodává kolem deseti tisíc položek. Ve firmě existuje několik informačních systémů, včetně vlastní databáze nákupů a prodejů na bázi FoxPro a účetního systému na bázi SAP. Protože jsou výstupy z těchto systémů dosti uživatelsky nepřívětivé, každé oddělení si vytvořilo své standardní tabulky v Excelu, které jsou ovšem zčásti zaplňovány a upravovány ručně, a tudíž jsou náchylné k chybovosti. Na existující zprávy se nemůžete spolehnout, protože nevíte, jaká data zahrnují a jestli jsou informace úplné. Souhrnné přehledy integrující výsledky jednotlivých oddělení neexistují. Navíc neexistuje jednotná databáze obchodů, popisující jejich vybavení, příslušnost do regionů a kvalitativní kategorizaci. Databáze prodejů neumožňuje přímo generovat časový vývoj, srovnání s plánem a podobně. Výsledkem je, že nedokážete odpovědět na základní otázky týkající se silných a slabých stránek firmy a jejího možného ohrožení.

Jako profesionál víte přesně, jaké informace potřebujete, abyste dostal firmu rychle pod kontrolu. Vznese proto dotaz na IT oddělení, jaké budou náklady na vytvoření standardních přehledů a sestav podle vašeho přání. Odpovědí budou pravděpodobně finanční náklady minimálně ve stovkách tisíc a časové náklady v řádu měsíců. Mimoto rozpočet neumožňuje zahrnout takové náklady do letošního roku, a tudíž je možné s pracemi začít nejdříve po schválení rozpočtu na rok příští a i schválení takového projektu je otázkou.

Vy ovšem nemáte čas čekat, protože akcionáři od vás očekávají rychlou analýzu situace a návrhy opatření. Navíc očekávají pravidelnou měsíční zprávu o situaci ohledně prodejů. A to je chvíle, kdy vám znalost programování aplikací Microsoft Office ve VBA může doslova zachránit kůži.

První den si ujasníte, která data budete potřebovat, abyste obdrželi potřebné souhrnné zprávy. Ještě tentýž den navrhnete několik základních tabulek v aplikaci Microsoft Access, které potřebujete pra-

videlně plnit daty, a definujete potřebné relace mezi nimi kvůli integritě dat (toto je velká přednost Accessu oproti Excelu). Uživatelské rozhraní zatím programovat nepotřebujete, protože databáze v Accessu bude sloužit pouze jako sklad dat.

Druhý den si necháte od IT oddělení předvést, jak a v jaké formě můžete data z jednotlivých systémů dostat. Je důležité, abyste dostali data na co nejnižší úrovni, protože nemůžete vědět, jestli agregační zprávy zahrnují všechny potřebné informace. Pokud bude mít předpokládaný objem dat do blízké budoucnosti několik set tisíc až několik málo milionů řádků, nevadí, protože SQL dotazy si s tímto objemem dat relativně hravě poradí. Omezení databáze Access (2GB) by se vás zdaleka týkat nemělo, navíc je možno data rozdělit a databáze propojit.

Když budete mít štěstí (autor ho většinou neměl), IT oddělení vám nainstaluje potřebné databázové ovladače a dá práva čtení k tabulkám ve firemní databázi. Pokud budete mít o něco méně štěstí, obdržíte ze systémů data v tabulární formě s jasně oddělenými sloupci ve formátu Microsoft Excel. Často ale nebudete mít štěstí vůbec, protože se bude jednat o textový soubor obsahující nepotřebné informace v záhlaví a v těžko čitelném formátu. Navíc není možno dostat všechna data v jednom souboru, protože systém umožňuje buď export prodejů po jednotlivých dnech pro jeden obchod, nebo pro všechny obchody pro zadané období. Pokud chcete sledovat denní vývoj, potřebujete obojí, a tudíž každý den exportujete data pro všechny obchody.

Nyní již víte, jaká data jsou k dispozici, a máte v Accessu připravené tabulky, do kterých budete data importovat. Je čas napsat první program ve VBA, jenž data upraví do podoby, kterou lze do tabulek správně naimportovat. Samozřejmě že elegantní je data upravovat přímo interakcí s textovým souborem, ale na psaní takového kódu nemáte čas a ostatně, je to zbytečné, neboť aplikaci Excel máte stejně otevřenou.

Založíte nový sešit, spustíte záznamník maker, otevřete soubor se syrovými daty a data upravíte do přijatelné podoby. Poté záznamník vypnete a napíšete efektivní kód, přičemž jako základ pro psaní složitějších metod a vlastností (`TextToColumns`, `FormulaR1C1`, apod.) použijete řádky nahrané záznamníkem. Během několika minut jste hotovi a kód sám otevře sešit, upraví data a opět jej uloží.

Dalším krokem je kód upravit tak, aby prohledal adresář na všechny podobné soubory, všechny upravil podobným způsobem a data zkopíroval pod sebe do hlavního sešitu. Odladění takového kódu chvíli trvá, ale odměnou vám bude konsolidace dat z několika desítek sešitů během několika sekund.

Dalším krokem bude napsání jednoduchého kódu, který data z konsolidovaného sešitu neimportuje do dočasné tabulky připravené v Accessu a spustí přidávací dotazy, které odpovídající data nahrají do příslušných tabulek. Během řádově několika desítek sekund máte ve vaší databázi nová aktuální data. Navíc jste si jisti, že pokud import proběhl zdárně, není narušena integrita dat (např. neexistují prodeje nepřizávané prodejně).

Nyní je čas daty nakrmit vaše standardní sestavy v Excelu. Na kartě „Data“, skupina „Načíst externí data“ klepnete na „Z aplikace Access“. Postupujete podle instrukcí průvodce a případně nezapomenete přidat do buněk parametry, pokud nechcete do listu načíst všechny data, která dotaz připravený v Accessu poskytuje.

Jako konečný krok propojíte buňky ve vašich sestavách na listu Excelu s importovanými daty a jste hotovi. Pokud změníte parametr v buňce, během řádově několika sekund (podle objemu dat) získáte přehled pro nové období.

Poté se můžete zabývat doplňkovými tabulkami a údaji (adresy, vybavení, kategorie), abyste mohli udělat přesnou analýzu situace, ale s hlavní částí jste hotovi. Každodenní aktualizace dat zabere méně času, než pití ranní kávy a vy máte každý den k dispozici aktuální přehled.

Dále můžete automatizovat automatické rozesílání standardních zpráv na definované adresy přes aplikaci Microsoft Outlook a rovněž automaticky vygenerovat měsíční prezentaci pro akcionáře v aplikaci Microsoft PowerPoint.

Domníváme se, že při investici řádově desítek hodin to není špatný výsledek, a určitě je nyní jasné, proč má smysl se učit VBA. Tato kniha se věnuje téměř výhradně jazyku VBA a objektovému modelu Excel, a proto v ní nenaleznete podrobný návod, jak řešit všechny kroky z našeho příběhu. Přesto ale po nastudování všech příkladů byste měli být schopni podobné programy psát.

Jak používat tuto knihu a vůbec, jak se učit VBA

Tato kniha si neklade za cíl stát se podrobným přehledem jazyka VBA a úplným popisem objektového modelu aplikace Microsoft Excel. O obou tématech byly napsány celé knihy a pouhý popis všech objektů aplikace Excel včetně jejich vlastností a metod by přesáhl obsah této knihy.

Jak jsme již řekli v úvodu, tato kniha je zaměřena na praktické využití jazyka VBA při programování aplikací v Excelu a řešení nejčastějších úloh. Autor je původcem více než 10000 příspěvků na mezinárodních fórech a zodpověděl tisíce dotazů. Samozřejmě že se dotazy často týkají podobných úloh, a přestože není možné kód napsat všeobecně, tato kniha se bude snažit představit techniky programování tak, aby je při rozumné míře přizpůsobení bylo možné použít pro vaše konkrétní potřeby.

I když předpokládáme, že čtenář zná základy programování ve VB nebo VBA, v prvních částech se zaměříme na popis nepoužívanějších konstrukcí jazyka VBA a rovněž popis základních objektů aplikace Microsoft Excel. Zatímco jazyk VBA je obecný a program může být zapsán do modulu kterékoli aplikace, která hostování jazyka podporuje, objektový model Excelu je specifický pro tuto aplikaci a jazyk VBA ho přes standardní objekty, metody a vlastnosti využívá. Všechny důležité funkce jazyka VBA i základních objektů aplikace Excel se budeme vždy okamžitě snažit demonstrovat na příkladech konkrétního použitelného kódu. Někdy uvedeme vedle sebe kód běžný, který je snadno pochopitelný, a kód optimalizovaný pro výkon, který často obsahuje nezvyklé konstrukce. Takové konstrukce vždy hned objasníme.

V dalších částech knihy se zaměříme na praktické použití VBA pro řešení nejčastějších úloh, se kterými se pokročilý uživatel Excelu setkává, například:

1. Hromadná změna dat ve sloupci
2. Transformace dat do databázové podoby
3. Konsolidace dat z více sešitů

Všechny tyto techniky umožňují enormní ušetření času a nákladů na pracovní sílu. Vzhledem k tomu, s jakým množstvím různých sešitů Excel z různých zdrojů se uživatel setkává, je časová úspora prakticky neomezená.

Mocným nástrojem při konsolidaci dat může být jazyk SQL (Structured Query Language), což je obecný jazyk určený pro tvorbu dotazů nad databázovými tabulkami. Přestože jeho použití v Excelu má určitá omezení, ukážeme jeho použití na konkrétních příkladech. Samozřejmě že podrobně

vysvětlení jeho syntaxe není vzhledem k rozsahu této knihy možné. O tomto jazyku a jeho použití v aplikacích Microsoft Office přes technologii ADO (ActiveX Data Objects) byly rovněž napsány celé knihy.

Co se týče použití uživatelských formulářů, tato kniha vás podrobně provede tvorbou uživatelského formuláře pro editaci dat v databázové formě. Formulář bude záměrně navržen tak, aby představil použití všech praktických ovládacích prvků, které budete při návrhu vašeho konkrétního formuláře potřebovat. Pokud budete navrhovat váš vlastní formulář, měli byste v této lekci najít vždy odpověď na otázku, jak konkrétní ovládací prvek použít ve vašem kódu.

Samostatným tématem Excelu jsou kontingenční tabulky (Pivot Tables). Tento nástroj je velmi mocný při analýze databázových dat a jejich agregaci do přehledů. V praxi jsme se příliš často nesetkali s nutností kontingenční tabulku vytvořit od začátku v kódu, ale přesto si tento postup ukážeme, neboť se při něm seznámíte s jejím objektovým modelem.

Vzhledem k tomu, že tato kniha není referencí jazyka, ale jeho praktickou učebnicí, doporučujeme knihu alespoň jednou být jen zběžně přečíst. Až se v budoucnu setkáte s konkrétním problémem, budete vědět, kde v knize najít konkrétní řešení, nebo alespoň návod k němu.

Jazyk VBA (a vlastně žádný programovací jazyk) se nenaučíte čtením popisu jednotlivých funkcí, objektů a možností jejich manipulace. Nejspolehlivější metodou je praxe. Programování je možno přirovnat k umění, jako je hudba. Můžete znát všechny noty a pravidla harmonie, ale to ještě neznamená, že jste schopni napsat skladbu, která se bude líbit. Podobně můžete do podrobností znát příkazy jazyka VBA a celý objektový model Excelu, ale to ještě neznamená, že napíšete program, který náročnou úlohu vyřeší za několik sekund. Přestože programování podobně jako hudba vyžaduje určitou dávku talentu, při určité dávce praxe a studia technik zkušených programátorů je možno za několik měsíců psát efektivní kód prakticky pro jakoukoli úlohu. A právě ke zkrácení této doby vám chce pomoci tato kniha. Najdete v ní mnoho ověřených řešení, která byste jinak dlouho hledali na internetu nebo v literatuře (podobně jako autor).

Otázky k částem lekcí

Za většinou částí lekcí najdete kontrolní otázky, které prověří, jestli jste porozuměli vykládanému textu. I když na konci knihy naleznete řešení, snažte se otázky nejprve zodpovědět samostatně, protože se většinou jedná o základy dané lekce.

Pokud je v řešení uveden příkladový kód, nejedná se samozřejmě většinou o jedinou možnost. Snažili jsme se však napsat vždy kód co nejefektivnější, který splňuje nároky na rychlost a současně jednoduchou upravitelnost.

V řešení jsme vždy danou otázku zopakovali, a tak vlastně příloha slouží i jako rychlá příručka řešení mnoha malých úloh, se kterými se budete v praxi setkávat.

Kód není vždy nejefektivnější řešení

Přestože tato kniha je téměř výhradně o programování aplikace Excel v jazyce VBA, musíme dát důrazné upozornění. Aplikace Microsoft Excel je složitý program a veškeré jeho části jsou zkompilovány (přeloženy) do jazyka, kterému procesor ve vašem počítači přímo rozumí a rychle reagu-

je. Oproti tomu váš kód VBA, zapsaný do normálních modulů nebo do modulů tříd, zůstává i po kompilaci (Debug => Compile Project) jazykem interpretovaným. To znamená, že příkazy nejsou předány procesoru přímo, nýbrž musí být nejprve za běhu programu přeloženy a vykonány. To znamená, že i ta neefektivnější zapsaná uživatelská funkce ve VBA bude v naprosté většině pomalejší, než vestavěný vzorec Excelu.

Pokud programujete aplikaci pro Excel, snažte se vždy o spojení obou světů. Vytvořte list nebo sadu listů, která využívá metody dostupné přes uživatelské rozhraní (vzorce, ověření dat, podmíněné formátování, apod.), a programem psaným ve VBA tuto aplikaci pouze podpořte. V praxi je mnohem jednodušší a rychlejší vytvořit dokonalou šablonu, kterou přes VBA naplníte libovolnými daty, než vytvářet celý sešit přes VBA! A to nemluvíme o údržbě. Pokud si váš šéf vymyslí nový formát zprávy, musíte kód měnit na mnoha místech. Pokud budete mít vytvořenou šablonu, změníte formát jednoduše přes uživatelské rozhraní, případně v kódu upravíte oblasti, kam budete zapisovat hodnoty, a jste hotovi.

Další začátečnickou chybou je duplikace vestavěných dialogů a funkcí. Často se setkáváme s dotazem typu:

„Chci vytvořit formulář, do něhož uživatel zadá písmeno a program vybere první buňku, která začíná tímto písmenem.“

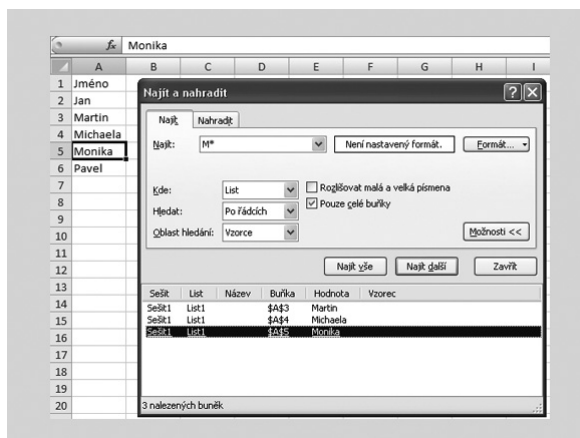
Samozřejmě že Excel má pro tuto úlohu vestavěný poměrně mocný dialog „Najít a vybrat“ ve skupině Úpravy. Tento dialog navíc umožňuje klepnout na tlačítko „Najít vše“ a klikáním na jednotlivé odkazy se pohybovat po zvolených buňkách!

Je určitě jednodušší uživatelům vysvětlit použití tohoto dialogu (případně pokročilé uživatele odkázat na nápovědu), než vytvářet vlastní formulář, který bude funkčně omezený, a navíc budete muset kód měnit při každé změně zadání. Kromě toho lze například původní dialog „Najít“ zobrazit přímo z VBA včetně parametrů:

```
Application.Dialogs(64).Show "M*", . . . . False, True
```

Jedinou výjimkou jsou takzvané diktátorské aplikace (*Dictatorship Applications*). V těchto vysoce profesionálních aplikacích, ušitých klientu na míru, se uživatelské rozhraní nahrazuje rozhraním vlastním a běžné funkce Excelu jsou pro uživatele nedostupné. Důvodem bývá nutnost omezit lidský faktor při zadávání dat, kdy například vložení sloupce nebo změna hlavičky může zcela zmást proceduru importu do jiné aplikace. Důvodem bývá i bezpečnost dat, ale jak se zmíníme v závěru knihy, tato bezpečnost je v Excelu dosti rozporuplná.

Vyvarujte se i přílišného obtěžování uživatele vyskakujícími okny se zprávami (funkce MsgBox). Nejenže takové chování je nestandardní a běžného



Obrázek Ú.1: Hromadné hledání buněk s obsahem odpovídajícím kritériu

uživatelé může bez důkladné nápovědy zcela zmást, ale nevhodné naprogramování může celou aplikaci prakticky paralyzovat!

Příkladem může být tento dotaz:

„Ve sloupci A mám čísla faktur a ve sloupci B jejich lhůtu splatnosti. Chci napsat program ve VBA, který zobrazí upozornění pro každou fakturu po splatnosti.“

Standardní odpověď může obsahovat tento kód, který úlohu vyřeší:



*Soubor 0 Uvod.xlsm
modul Module1*

```
Sub AlertOverdue()  
Const MSG_TITLE As String = "Faktura po splatnosti"  
Dim c1 As Range, rg As Range  
Dim msg As String  
With ActiveSheet  
    Set rg = .Range("B2", Cells(.Rows.Count, "B").End(3))  
End With  
For Each c1 In rg  
    If c1 < Date Then  
        MsgBox c1(, 0) & ": " & c1, vbOKOnly, MSG_TITLE  
    End If  
Next  
End Sub
```

Jeho použití ovšem nedoporučujeme, pokud se – což je pravděpodobné – na seznamu vyskytuje po splatnosti faktur více. Jestliže má seznam několik desítek nebo dokonce stovek takových položek, uživatel by se z kódu nedostal, dokud by neodklepal všechna vyskakující okna se zprávou! Pokud chcete zobrazit všechny problematické položky v jedné zprávě, použijte tento kód:



*Soubor 0 Uvod.xlsm
modul Module1*

```
Sub AlertOverdue2()  
Const MSG_TITLE As String = "Faktury po splatnosti"  
Dim c1 As Range, rg As Range  
Dim msg As String  
With ActiveSheet  
    Set rg = .Range("B2", Cells(.Rows.Count, "B").End(3))  
End With  
For Each c1 In rg  
    If c1 < Date Then  
        msg = msg & c1(, 0) & ": " & c1 & vbLf  
    End If  
Next  
MsgBox msg, vbOKOnly, MSG_TITLE  
End Sub
```

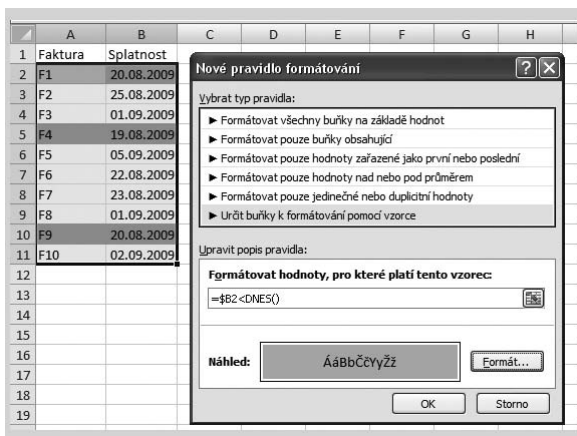
V praxi však zobrazení okna se zprávou mnoho nepomůže. Co s ním uživatel udělá? Opíše si všechna čísla, než klepne na OK? Práce s takovou aplikací by určitě nebylo nic příjemného.

Proto doporučujeme použít podmíněný formát:

1. Vyberte souvislou oblast s daty ve sloupcích A a B tak, aby buňka A2 byla aktivní
2. Na kartě „Domů“ klepněte na „Podmíněné formátování“ ve skupině „Styly“ a vyberte položku „Nové pravidlo“.
3. V následujícím dialogu klepněte na položku „Určit hodnoty k formátování pomocí vzorce“ a do textového pole запиšte vzorec: `=B2<DNES()`. Dialogové okno ještě nezavírejte.
4. Klepněte na tlačítko „Formát“, upravte formát podle potřeby (např. výplň buňky) a potvrďte klepnutím na „OK“.
5. Dialogové okno podmíněného formátu uzavřete rovněž klepnutím na tlačítko „OK“.

V naprosté většině případů je tazatel udiven robustností podmíněného formátu a volí toto řešení. Následujícím dotazem může být, jak implementovat podmíněný formát automaticky, pokud se seznam faktur dynamicky mění. Jeho hlavní problém je ale vyřešen rychle a elegantně, a to i ke spokojenosti ostatních uživatelů.

Chtěli jsme uvést tyto příklady, abyste si uvědomili, že než začnete psát kód, je vždy vhodné si ujasnit, jaké je vlastně zadání. V posledním příkladu bylo přáním upozornit na faktury po splatnosti. Ačkoli je úlohu možné řešit přes VBA, podmíněný formát je o mnoho vhodnější a nabízí větší flexibilitu, například zobrazení různých dob prodlení různými barvami. Mimochodem, podmíněné formátování bylo v Excelu 2007 podstatně rozšířeno a podle nás je jedním z důvodů, proč přejít na vyšší verzi.



Obrázek Ú.2: Použití podmíněného formátu k upozornění na kritické řádky

Shrnutí

Nesnažte se napodobovat standardní funkce Excelu. Ve většině případů je to zbytečné a uživatele to jedine zmate. Navíc se vaše aplikace zpomaluje. Raději vysvětlete, jak se ta která funkce používá, a pokročilí uživatele odkážete na nápovědu.

Pamatujte si, že v případě vyskakovacích oken a zpráv je méně více. Neustálé zobrazování opakujících se zpráv a upozornění nesvědčí o vaší profesionalitě, ale o tom, že neumíte navrhnout list tak, aby uživatele logicky vedl ke správnému zadávání dat.

Standardní vestavěné dialogy můžete zobrazit přímo z VBA. V nápovědě hledejte výraz „XlBuiltinDialog Enumeration“ pro podrobný seznam dialogů. Pro seznam možných parametrů hledejte výraz „Built-In Dialog Box Argument Lists“.

LEKCE 1

Základy jazyka VBA

V této lekci najdete:

◆ Proměnné	22
◆ Důležité konstrukce VBA použité v knize.....	38
◆ Text ve VBA	67
◆ Čísla ve VBA	124
◆ Datum ve VBA.....	139

Jak jsme již řekli v úvodu, je jen jeden jazyk VBA, který může, ale nemusí používat objektové modely jiných aplikací a knihoven. Tato lekce vás seznámí se základy jazyka VBA a zaměří se na problematiku, kterou vývojář aplikací v Excelu hojně využije, totiž manipulaci s textem, čísly a daty. Většina příkladů, pokud nevyužívá pouze knihovnu VBA, využívá objekty aplikace Excel, protože konečnou právě o programování Excelu tato kniha je. Kromě specifických příkladů je ale veškerý text platný i pro vývoj jiných aplikací, které hostují VBA na základě VB6 (od verze Office 9, neboli 2000). Většina kódu by pracovala i v Excelu 8 (97), který hostoval VBA na základě VB5. Nové funkce jsou například funkce `Join`, `Split`, `Filter` a `InStrRev`.

Proměnné

Proměnnou si můžete představit jako pojmenované místo v paměti počítače, na němž se nachází kousek dat, se kterými chcete pracovat. Data mohou mít samozřejmě různou podstatu a podobu, jako je celé číslo, záporný zlomek, iracionální číslo (např. druhá odmocnina ze dvou, číslo π či základ přirozených logaritmů, známý též jako Eulerovo číslo e), částka v domácí měně, seznam hodnot, datum, text, sešit Excelu, graf na listu Excelu, nebo dokonce celá běžící aplikace Microsoft Word.

Proměnných ve VBA je celá řada typů, jak je vidět na seznamu v tabulce 1.1.

Tabulka 1.1: Přehled typů proměnných

Datový typ	Rozsah hodnot	Velikost v paměti
Boolean (Logický)	True nebo False	2 byty
Byte	0-255	1 byte
Currency (Měnový)	-922,337,203,685,477.5808 až 922,337,203,685,477.5807	8 bytů
Date (Datum)	1. leden 100 až 31. prosinec 9999, čas 0:00:00 až 23:59:59	8 bytů
Decimal (Desetinný)	Maximální hodnota +/-79,228,162,514,264,337,593,543,950,335 s desetinnou čárkou v libovolném místě	12 bytů
Double (Dvojitá přesnost)	-1.79769313486231 až -4.94065645841247E-324 pro záporná čísla, 4.94065645841247E-324 do 1.79769313486232E308 pro kladná čísla	8 bytů
Integer (Krátké celé číslo)	-32,768 až 32,767	2 byty
Long (Dlouhé celé číslo)	-2,147,483,648 až 2,147,483,647	4 byty
Object (Objekt)	Uložený ukazatel (pointer) na libovolný objekt v paměti	4 byty
Single (Jednoduchá přesnost)	-3.402823E38 až -1.401298E-45 pro záporné hodnoty a 1.401298E-45 až 3.402823E38 pro hodnoty kladné	4 byty
String (Řetězec, „Text“)	Může obsahovat až 2^{31} znaků, může mít i konstantní délku až do přibližně 64 000 znaků	Různá
User Defined (Uživatelský)	Může obsahovat jeden nebo více různých typů	Různá
Variant	Může obsahovat různé hodnoty a objekty, viz dále v této lekci	Různá

Nejdůležitější typy proměnných a kdy je použít

Protože tato kniha není vyčerpávajícím popisem jazyka VBA, nýbrž praktickou příručkou, zaměříme se pouze na nejpoužívanější typy při práci s aplikací Microsoft Excel.

Proměnná typu Boolean

Proměnná obsahuje buď hodnotu `True` (Pravda) nebo `False` (Nepravda). Použijete ji jako podmínku v rozhodovacích příkazech a také jako návratovou hodnotu mnoha uživatelských funkcí, např.



01-01 Proměnné.xlsm
modul MBoolean

```
'// Funkce vrátí True, pokud list zadaného jména v aktivním sešitu existuje

Public Function WorksheetExists(SheetName As String) As Boolean

On Error Resume Next
    WorksheetExists = Len(Worksheets(SheetName).Name) > 0
    Err.clear
On Error Goto 0

End Function
```

Přičemž tuto funkci použijete například takto:

```
'// Pokud list se zadaným jménem v aktivním sešitu neexistuje,
'// přidej ho, pojmenuj a přesuň na začátek

Const SheetName As String = "List1"
If Not WorksheetExists(SheetName) Then
    With Worksheets.Add
        .Name = SheetName
        .Move Before:=Sheets(1)
    End With
End If
```

Kód přidá list, pokud neexistuje, podle obrázku:

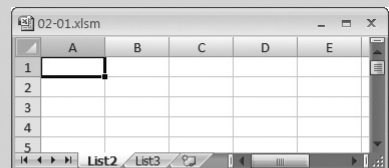
Typické použití proměnné Boolean v kódu je

```
If PromennaBoolean Then
    ' Vykonej, pokud True
Else
    ' Vykonej, pokud False
End If
```

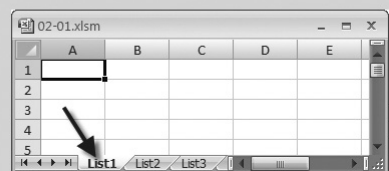
Proměnná typu Long

Kromě proměnných, které ukládají ukazatel (pointer) na objekt, budete tento typ proměnné používat asi nejčastěji. Typické použití je čítač v cyklu a uložení čísla řádku nebo sloupce.

Původní stav



Výsledný stav



Obrázek 1.1: Přidání nového listu na začátek sešitu



01-01 Proměnné.xlsm
modul MLong

```
'// Kód vynásobí každou druhou buňku ve sloupci A
'// aktivního listu dvěma, počínaje druhým řádkem
```

```
Dim i As Long 'Citac
Dim lLastRow As Long 'poslední řádek

lLastRow = Cells(Rows.Count, 1).End(xlUp).Row
```

```
For i = 2 To lLastRow Step 2
    Cells(i, "A") = Cells(i, "A") * 2
Next i
```

Proměnnou typu `Integer` jsme vynechali úmyslně. Ušetření paměti je minimální a s proměnnou typu `Long` zachází dvaatřicetibitová architektura počítače rychleji. Kromě toho je při velikosti listu v Excelu 2007 (a vlastně i v několika předešlých verzích) reálné nebezpečí, že bychom se do proměnné pokusili přiřadit hodnotu mimo její rozsah, což by vyvolalo *run-time* chybu (chybu za běhu) programu. Pokud budete pro celá čísla používat typ `Long`, ve většině případů bude váš kód bezpečný.

Proměnná typu `String`

Tuto proměnnou budete používat pro uložení řetězcových proměnných, čili zjednodušeně řečeno, textu. Mějte však na paměti, že operace s textem jsou jedny z nejpomalejších ve VBA, a použijte je s rozmyslem, zejména pak v cyklech.

Předpokládejme, že buňky A1:A5 obsahují po jednom slovu, které chcete spojit v jeden řetězec, kde budou jednotlivá slova oddělená čárkou.

Obvyklý kód vypadá přibližně takto:



01-01 Proměnné.xlsm
modul MString

```
'//spojení slov v jeden text v cyklu
```

```
Dim cl As Range
Dim strText ' výsledný text
For Each cl In Range("A1:A5")
    strText = strText & cl.Value & ","
Next cl
strText = Left$(strText, Len(strText) - 1)
Debug.Print strText
```

Efektivní kód však vykoná stejnou práci bez opakovaného spojování řetězců:

```
'//spojení slov v jeden text bez cyklu
```

```
Dim y
Dim strText ' výsledný text
y = Application.Transpose(Range("A1:A5"))
strText = Join(y, ",")
Debug.Print strText
```

Jak kód druhého typu pracuje, se dozvíte v následující části této lekce.

Proměnná typu Double a Currency

S těmito typy budete pracovat méně, neboť při programování aplikace Excel jsou hodnoty většinou vypočítávány vzorci na listu. Přesto se jejich použití nevyhnete například při výpočtu degresivní provize v závislosti na prodeji. Proměnná typu Double má jednu velkou výhodu a zároveň nevýhodu. Výhodou je, že rozsah hodnot je skutečně obrovský a kromě mezních astrofyzikálních výpočtů týkajících se budoucnosti vesmíru pojme jakékoli smysluplné číslo. Na druhou stranu, čísel v daném rozsahu je nekonečné množství a proměnná má k dispozici omezený počet bitů. Proto rovněž nekonečné množství čísel nemůže být zobrazeno s absolutní přesností. Již relativně běžná čísla mohou vést k neočekávaným výsledkům:



01-01 Proměnné.xlsm
modul MDouble

```
Dim a1 As Double, a2 As Double, a3 As Double
a1 = 1
a2 = 0.3451
a3 = 0.6549
Debug.Print (a1 - a2 - a3) = 0
Debug.Print a1 = a2 + a3
Debug.Print a1 - a2 - a3
```

Protože $a1 = a2 + a3$, na výpisu v okně Immediate (Ctrl+G, pokud se nacházíme ve VB Editoru) bychom čekali následující údaje:

```
True
True
0
```

Výstup však zobrazí částečně jiné výsledky:

```
False
True
-5,55111512312578E-17
```

Přičemž první výsledek je vyloženě špatně a třetí je sice velmi blízko nule, avšak nula to není. Je nutné si uvědomit, že chyba se takzvaně *propaguje* do výpočtu, neboli její vliv se s nárůstem počtu operací s chybným číslem zvětšuje. Ještě větším nebezpečím je použití proměnné typu Double jako podmínky pro ukončení cyklu, neboť nikdy nemusí nabýt přesné očekávané hodnoty. Toto může váš kód poslat do nekonečného cyklu, ze kterého je vysvobozením pouze <Ctrl>+<Break> (v lepším případě), nebo „sestřelení“ Excelu přes Task Manager (v horším případě).

V naprosté většině nefyzikálních výpočtů je řešením nahradit typ Double typem Currency:

```
Dim a1 As Currency, a2 As Currency, a3 As Currency
a1 = 1
a2 = 0.3451
a3 = 0.6549
Debug.Print (a1 - a2 - a3) = 0
Debug.Print a1 = a2 + a3
Debug.Print a1 - a2 - a3
```

Což vrátí výpis

```
True
True
0
```

čili očekávaný výsledek.

Uvedeme ještě jeden příklad, demonstrující, jak nebezpečné je použít proměnnou typu `Double` v podmínce cyklu:



01-01 Proměnné.xlsm
modul MDouble

```
Sub TestDouble3()

'// Bez ochranného mechanismu by tento kód skončil
'// v nekonečné smyčce

Const MAX_LOOP As Long = 999999
Dim i As Long
Dim dblHodnota As Double
i = 1
Do
    dblHodnota = dblHodnota + 0.0001
    If dblHodnota = 1 Then GoTo ExitSub

    '// Ochranný kód eliminující nekonečný cyklus
    i = i + 1
    If i > MAX_LOOP Then GoTo ExitSub

Loop

ExitSub:
    Debug.Print "Hodnota: " & dblHodnota & vbCrLf & "Pocet cyklu: " & i
End Sub
```

Kód vykonává velmi jednoduchou činnost. Přičítá hodnotu 0,0001 ke stávající hodnotě proměnné `dblHodnota` a po dosažení hodnoty rovné jedné se tato hodnota vypíše společně s počtem cyklů a program se ukončí. Na výstupu bychom tedy očekávali následující text:

```
Hodnota: 1
Pocet cyklu: 10000
```

Ovšem kód po svém ukončení vypíše něco jiného:

```
Hodnota: 99,9999000021961
Pocet cyklu: 1000000
```

Neboli kód se *neukončil* v momentě, kdy `dblHodnota` nabyla hodnoty jedna, protože ona přesně této hodnoty nikdy nenabyla! Přesně řečeno, v 10000. cyklu nabyla hodnoty 0,999899999999906, což je sice hodnota velmi blízká jedné, ale přesně jedna to není.

Opět, pro číslo do čtyř desetinných míst může pomoci použití typu `Currency`:

```
Sub TestCurrency()  
Dim i As Long  
Dim currHodnota As Currency  
i = 1  
Do  
    currHodnota = currHodnota + 0.0001  
    If currHodnota = 1 Then GoTo ExitSub  
    i = i + 1  
Loop  
  
ExitSub:  
    Debug.Print "Hodnota: " & currHodnota & vbCrLf & "Pocet cyklu: " & i  
End Sub
```

Všimněte si, že tento kód nepotřebuje obranný kód proti nekonečné smyčce. Pro profesionální aplikace vám ale jeho použití vřele doporučujeme, neboť určitě nechcete, aby vám váš klient volal, že mu „zamrzl“ Excel po půldenním usilovném zadávání dat. K této problematice se ještě vrátíme v části popisující cyklus `Do...Loop`.

Závěrem shrňme, že ačkoli Excel považuje čísla v buňkách implicitně za typ `Double`, v naprosté většině případů je vhodné je přiřadit do proměnné typu `Currency`. Uvědomte si, že do této proměnné nemusíte ukládat pouze údaje v penězích. Proměnnou můžete použít pro všechna čísla v daném rozsahu, která chcete zobrazit s absolutní přesností na čtyři desetinná místa.

V případě, že proměnnou typu `Currency` použít nemůžete, buďte opatrní při porovnávání hodnot a používejte zaokrouhlení a intervaly.

Proměnná typu `Object`

Proměnná tohoto typu bude pravděpodobně jednou z nejčastějších proměnných, se kterými budete ve VBA kódu pro Excel pracovat. Existují pro to dva důvody, *rychlost* a *čitelnost* kódu.

Existují dva základní druhy použití:

```
Dim oObject As Object  
  
Dim wdApp as Word.Application  
Dim ws As Worksheet  
Dim rg As Range
```

V prvním případě předem nevíme, kterého typu objekt bude a tudíž kterou třídu mu přiřadit. Proto proměnnou deklarujeme s obecnou třídou `Object`. Typicky se u tohoto typu deklarace setkáte při automatizaci jiných s aplikací tzv. *pozdní vazbou* (*Late Binding*).

V ostatních případech známe, jaké třídy objekt bude, a zároveň je knihovna obsahující tento objekt zahrnuta v referencích sešitu, ve kterém je tento kód zapsán (VBE=>Tools=>References). Toto je velmi důležité! Objektové knihovny Excelu, VBA a Office zde budou zaškrtnuty standardně, jiné, např. Word, ADO či Outlook, musíte vyhledat a zaškrtnout, případně dokonce nejprve nainstalovat, pokud chcete jejich objekty explicitně deklarovat a používat. Tento přístup má mnohé výhody, jak se dozvíte v části o automatizaci aplikací v lekci o psaní efektivního kódu.

Proměnná typu Variant

Tuto specialitu jazyka VB a VBA jsme si nechali na konec přehledu nejpoužívanějších typů. `Variant` je flexibilní typ proměnné, který může pojmout jakoukoli hodnotu kromě řetězce o pevně dané délce (fixed-length string), včetně hodnot a objektů `Empty`, `Error`, `Nothing` a `Null`. Ačkoli od verze Excelu 2002 může být do proměnné typu `Variant` přiřazen i uživatelský typ proměnné, toto použití je velmi vzácné.

Ve všech těchto případech bude proměnná `var` typu `Variant`:

```
Dim var As Variant
Dim var, i As Long
Dim var
```

Pokud není v záhlaví modulu uvedeno `Option Explicit`, je možno i deklaraci vynechat a proměnnou použít rovnou v kódu. Bude automaticky typu `Variant`:

```
var = var + 1
```

Poslední přístup se zásadně nedoporučuje, jak si řekneme v části o deklaraci proměnných a jejich rozsahu.

Pro zjištění podtypu proměnné, která se skrývá v proměnné `Variant`, můžete použít funkci `VarType`:

```
If VarType(Var) = vbLong Then
'// kód, který pracuje pouze s celými čísly
End If
```

Jiná funkce, která zjistí typ proměnné, je funkce `TypeName()`.

Přestože použití proměnné typu `Variant` může vypadat lákavě, nepoužívejte ji kromě případů, kdy je to nutné. Jedním takovým případem je nutnost ošetřit chybu vrácenou funkcí za běhu programu. Tato funkce vrátí hodnotu typu `Error`, pokud hledaná hodnota není v seznamu:



01-01 Proměnné.xlsm
modul MVariant

```
Function NajdiMne(arg As Long)
    NajdiMne = Application.Match(arg, Array(1, 3, 5), 0)
End Function
```

Všimněte si, že celá funkce nemá deklarovaný typ, čili je implicitně typu `Variant`, aby mohla vrátit chybu jako typ proměnné. Pokud by byla deklarována jako typ `Long`, došlo by k run-time chybě 13: `Type Mismatch`, neboli kolizi typu (`Error versus Long`).

Vrácenou hodnotu můžete ověřit, než výsledek použijete v dalším kódu:

```
Sub TestujFunkci()
Dim Vysledek
Vysledek = NajdiMne(2)
If TypeName(Vysledek) = "Error" Then
    MsgBox "Hodnota nenalezena"
Else
'// kód pokračuje, hodnota nalezena
End If
End Sub
```

Druhou důležitou vlastností proměnné typu Variant je schopnost pojmut pole o předem neznámé velikosti a typu. Tato vlastnost je pro programátora Excelu natolik důležitá, že ji podrobněji rozvedeme v příští lekci.

Ve všech ostatních případech vám radíme používat konkrétní typ. Je to efektivnější z hlediska využití paměti a výkonu počítače, vyvarujete se některých základních chyb a váš kód bude mnohem čistší a snadněji udržovatelný.



Poznámka

Proměnná typu Variant je jedním z důvodů, proč VBA není „opravdovými programátory“ považován za plnohodnotný programovací jazyk. Visual Basic se jim zdá být příliš benevolentní, což vede ke zvýšenému riziku výskytu těžko odhalitelných chyb. „Opravdový programátor“ není pojem, který jsem vymyslel, nýbrž jím sami sebe označují programátoři v jazycích C, C++, C# a podobně. Určitě je pravda, že VBA má daleko do „strong-typed“ jazyka, tedy jazyka, který umožňuje pouze explicitní deklarace a konverze datových typů. Na druhou stranu ale moudré použití proměnných typu Variant umožňuje psát pro aplikace z rodiny Microsoft Office kód rychle a efektivně, a přesně o tom je tato kniha. Držte se především dvou pravidel: Mějte vždy zapnutou explicitní deklaraci (`Option Explicit`) a proměnné typu Variant používejte jen tehdy, když přesně vyhraněný typ selže. Při psaní kódu pravidelně používejte kompilaci (`Debug=>Compile ...`) a budete mít jistotu, že jste – alespoň co se týče deklarace a následné použití jmen proměnných – nikde neudělali chybu.

Je výborné znát angličtinu nebo němčinu, ale pokud znáte oba jazyky, váš obzor se neskutečně rozšíří. A podobně je to i s jazyky programovacími. Ne náhodou nejlepší vývojáři ve VBA znají i C++ a jeho výhody při vytváření např. knihoven funkcí. Bohužel, toto téma již přesahuje rámec této knihy.

Pole a jejich využití

Pole je indexovaná skupina dat, která se chová jako jedna proměnná. Na jednotlivé prvky pole se odkazujeme *celočíselným* indexem.

Jednorozměrné pole (pole o jedné dimenzi) můžeme deklarovat různým způsobem:

```
Dim Prodeje (2002 To 2009) As Double
Dim Prodeje (1 To 8) As Double
Dim Prodeje (7) As Double
Dim Prodeje() As Double
Dim Prodeje
```

Na prvním příkladu vidíte, že spodní hranice nemusí nutně začínat indexem 0 nebo 1. Pokud do pole ukládáte například roční hodnoty, může být výhodné pro čitelnost použít index korespondující s rokem.

Ve třetím příkladu bude mít pole osm prvků a obvykle je tato deklarace identická s deklarací

```
Dim Prodeje (0 To 7)
```

Klíčové je zde slovo *obvykle*. Pokud totiž uvedete do záhlaví modulu klauzuli

```
Option Base 1
```

bude pole bez deklarované spodní hranice začínat prvkem s indexem 1.

Stejně jako vám doporučujeme *vždy* v modulu uvádět `Option Explicit`, doporučujeme *nikdy* nepoužívat výraz `Option Base 1`. Důvod je ten, že pokud kód zkopírujete do modulu bez této klauzule, kód nemusí pracovat správně. Takovou chybu může být velmi složité odhalit, neboť se typicky jedná o chybu algoritmu, a ne o chybu syntaktickou nebo run-time chybu (chybu za běhu). Tím se zbavíte možnosti kód beze ztráty funkčnosti přenášet do jiných aplikací. Zvykněte si spodní hranici *vždy* explicitně deklarovat a kód bude vždy pracovat očekávaným způsobem.

Čtvrtý příklad deklarace je znám jako Dynamické pole a použijete jej v případě, že předem neznáte počet prvků pole. Příkladem je kód, který naplní pole `SheetNames` jmény listů v aktivním sešitu. Počet listů je předem neznámý:



01-02 Pole.xlsm
modul MArrays

```
Dim SheetNames() As String
Dim i As Long
ReDim SheetNames(1 To Sheets.Count)
For i = LBound(SheetNames) To UBound(SheetNames)
SheetNames(i) = Sheets(i).Name
Next i
```

Ačkoli je možné již zčásti naplněné pole předimenzovat při zachování již existujících prvků za pomoci příkazu `ReDim Preserve`, použijte tento postup, jen pokud je to absolutně nezbytné. VBA musí nejprve vytvořit nové pole v paměti počítače a poté do něj prvky překopírovat, než uvolní paměť zabranou původním polem. Tento proces je (v řeči počítače) velmi pomalý. Pokud je tento příkaz použit v kódu jednou, nezaznamenáte žádné výrazné zpomalení, ale ve smyčce o velkém počtu průchodů může ztráta času exponenciálně narůstat s každým průchodem cyklem.

Jako příklad uvedeme naplnění pole jmény souborů typu Word, které se nacházejí v daném adresáři a jméno souboru začíná řetězcem „Test“.



01-02 Pole.xlsm
modul MArrays

```
Dim FileNames() As String
Dim FileName As String
Dim strFolder As String

'// Nadefinujeme adresář
strFolder = ThisWorkbook.Path & "\TestFiles\"

'// Pokusíme se najít první soubor
FileName = Dir$(strFolder & "Test*.doc")

If Len(FileName) > 0 Then
'// Víme, že alespoň jeden soubor existuje

    ReDim FileNames(1 To 1)
    FileNames(1) = FileName
    FileName = Dir$

    Do While Len(FileName) > 0
'// Pokud nalezen další soubor, předimenzuj pole
```

```

    '// a ulož nové jméno na konec pole
    ReDim Preserve FileNames(1 To UBound(FileNames) + 1)
    FileNames(UBound(FileNames)) = FileName
    FileName = Dir$
Loop
End If

```

Velmi zajímavou možností pro programátora Excelu je naplnit dynamické pole přímo daty z listu. Podmínkou je, že pole musí být typu `Variant`.



Poznámka

Od doby, kdy jsem objevil, že Excel umí oblast buněk načíst přímo do pole, se „quicked–stacked“ pole stala neodmyslitelnou součástí mého stylu psaní kódu. Nevím, jak termín „quicked–stacked“ přesně přeložit. Představte si to jako proces, kdy pole nadimenzujete i naplníte současně. Myslím, že kromě Visual Basicu žádný jiný jazyk tento přístup nepodporuje, takže pokud se učíte programovat například v C#, rozhodně na tento přístup zapomeňte.

Řekněme, že máme oblast buněk, ve kterých chceme odříznout první znak. Můžeme vložit pomocný sloupec a vložit vzorec `=MID(A1,2,LEN(A1))`, poté sloupec okopírovat a vložit jako hodnoty přes původní data, a nakonec jej opět vymazat. Můžeme rovněž napsat kód podobný tomuto:

```

'// Obvyklý Kód

Dim c1 As Range
For Each c1 In Range("A1:A10")
...c1.Value = Mid$(c1.Value,1,Len(c1.Value))
Next c1

```

Přestože na tomto kódu není vůbec nic špatného, jednu drobnou vadu přeci jen má. Při každém průchodu cyklem třikrát interaguje s aktivním listem. Dvakrát hodnotu čte a potřetí ji zapisuje. Samozřejmě, jedno čtení můžeme ušetřit uložením hodnoty do dočasné proměnné, ale výrazného ušetření času nedosáhneme.

Pokud použijeme pole, veškerá kalkulace se odehraje pouze v paměti RAM počítače a konečný výsledek zapíšeme na list najednou:

```

'// Rychlý Kód

Dim y, i As Long
y = Range("A1:A10")
For i = LBound(y) To UBound(y)
y(i, 1) = Mid$(y(i, 1), 2, Len(y(i, 1)))
Next i
Range("A1").Resize(UBound(y)).Value = y

```

První řádek deklaruje dvě proměnné, první typu `Variant` a druhou typu `Long`. VBA neumožňuje deklarovat více proměnných jednou klauzulí `As`. Každá proměnná, která bezprostředně nesousedí s klauzulí `As <TypProměnné>`, bude typu `Variant`.

Kvůli čitelnosti by řada programátorů zvolila explicitní deklaraci:

```
Dim y As Variant
Dim i As Long
```

Doporučuji používat deklaraci na jednom řádku, protože tyto dvě proměnné k sobě logicky patří a rovněž se tím zlepšuje čitelnost kódu.



Poznámka

Osobně příliš nepoužívám tzv. Maďarskou notaci, která stanovuje pravidla pojmenování proměnných. Vedl jsem velmi obsáhlé debaty s vývojáři ve VBA a nikdy jsme se neshodli, jestli standardní pravidla pojmenování proměnných ve VBA používat, či ne. Jediné, na čem jsme se vždy shodli, je důslednost. Každý programátor si časem vyvine určitý systém pojmenování proměnných, modulů apod. Není těžké po několika minutách jeho konvenci pochopit, ale je problém, když autor konvenci mění v každé rutině, kterou napíše. Mimochodem, malý trik: Když chcete zjistit, kterého typu je ta která proměnná, umístěte do ní v kódu kurzor a stiskněte Shift+F2. Editor vás přenese na její deklaraci. Bohužel, opačným směrem to funguje jen omezeně, ale Ctrl+Shift+F2 vás přenese alespoň na začátek řádku s proměnnou, jejíž typ jste zjišťovali.

Další řádek kódu

```
y = Range("A1:A10")
```

vykoná vlastně dva příkazy. Nejprve vytvoří dvojrozměrné pole typu Variant o rozměrech 10×1 (počet řádků \times počet sloupců) a poté jej naplní hodnotami z buněk. Důležité je, že spodní hranice pole je vždy 1, na rozdíl od implicitní spodní hranice polí ve VBA, čímž skvěle koresponduje s adresami buněk v listu.

Alespoň věříme, že příkazy vykonává v tomto pořadí. Pokud by totiž načtl hodnotu, předimenzoval pole, hodnotu uložil a pokračoval až do konce oblasti, vykonával by vlastně příkaz `Redim Preserve` v cyklu. Tento příkaz společně s funkcí `Union` je jednou z nejpomalejších funkcí VBA a v cyklu se zásadně nedoporučuje, jak jsme se již zmínili v předešlé části.

Vzhledem k tomu, že nás zajímá rozměr pouze první dimenze (víme, že druhá je rovna jedné, podle rozměru oblasti), můžeme druhý parametr funkcí `Lbound(po1e, <dimenze>)` a `Ubound(po1e, <dimenze>)` vynechat a spustit cyklus s čítačem `i`, který všechny hodnoty v poli upraví:

```
For i = LBound(y) To UBound(y)
y(i, 1) = Mid$(y(i, 1), 2, Len(y(i, 1)))
Next i
```

Nakonec pomocí metody `Resize` objektu `Range` určíme oblast, která přesně pojme naše pole, a jako parametr předáme velikost první dimenze pole. Pole „vyprázdníme“ na list opět jediným příkazem:

```
Range("A1").Resize(UBound(y)).Value = y
```

Samozeřejmě nemusíme původní rozsah přepisovat, můžeme zaměnit „tvar“ pole na šířku, a dokonce bez jakékoli interakce s listy můžeme pole „vyprázdnit“ na jiný list:

```
Sheets("Sheet2").Range("A1").Resize(,UBound(y)).Value= _
```

```
Application.Transpose(y)
```

Všimněte si, že tentokrát je rozměr pole použit jako druhý parametr metody `Resize` (protože měníme počet sloupců). K transformaci pole jsem si vypůjčil funkci Excelu `TRANSPOSE()`. Je to další příklad toho, že vhodnou kombinací VBA a vestavěných funkcí získáme to nejlepší z obou světů.

Nyní uvedeme praktičtější příklad. Řekněme, že máte prvek `ComboBox` umístěný na formuláři a při otevření formuláře jej chcete naplnit seznamem ve sloupci A listu `Sheet1`. Buňka A1 obsahuje hlavičku a data o neznámém počtu začínají v buňce A2. Vše, co potřebujete, je umístit tento kód do modulu třídy formuláře:



01-02 Pole.xlsm
modul UserForm1

```
Private Sub UserForm_Initialize()
    Dim y
    With Sheets("Sheet1")
        y = .Range("A2", .Cells(.Rows.Count, "A").End(3))
    End With
    Me.cboSeznam.List = y
End Sub
```

Obvyklý kód bez použití polí vypadá takto:

```
Private Sub UserForm_Initialize()
    Dim c1 As Range
    With Sheets("Sheet1")
        For Each c1 In .Range("A2", .Cells(.Rows.Count, "A").End(3))
            Me.cboSeznam.AddItem c1.Value
        Next c1
    End With
End Sub
```

Všimněte si, že kód volá metodu `AddItem` prvku `ComboBox` při každém průchodu cyklem. Elegance uložení seznamu do pole vynikne zejména tehdy, když potřebujeme seznam například setřídít před jeho předáním do ovládacího prvku a nemůžeme nebo nechceme třídít hodnoty na listu.

Tolik tedy ke „quick-stacked arrays“ neboli rychle naplněným polím. V úvodu jsme ale slíbili, že vás budeme učit psát rychlý a efektivní kód. Pokud zůstaneme u našeho příkladu s formulářem, tady je:

```
Private Sub UserForm_Initialize()
With Sheets("Sheet1")
    Me.cboSeznam.List = _
        Application.Transpose(.Range("A2", .Cells(.Rows.Count, "A").End(3)))
End With
End Sub
```

Všimněte si, že celý příkaz je jeden jediný řádek. Osobně ale dáváme přednost kódu využívajícímu pole, protože je minimálně slušnost dbát rovněž na čitelnost kódu. Pokud tedy v knize uvidíte proměnné typu `y` a `z`, vězte, že se jedná o pole typu `Variant`, naplněná hodnotami z oblasti listu. Jestli vás toto téma zaujalo, experimentujte s přiřazováním oblasti do proměnné typu `variant` v kombinaci s funkcí `Transpose` a testujte horní hranici obou rozměrů. Ke „quick-stacked“ polím se ještě jednou vrátíme v lekcích o funkcích `Split`, `Join` a `Filter`.

Vlastní kolekce a jejich využití

Kolekce je soubor objektů stejné třídy. List je objekt třídy Worksheet a patří do kolekce Worksheets, otevřený sešit je objektem třídy Workbooks a patří do kolekce Workbooks. Na rozdíl od polí se na prvky v kolekci můžeme odkazovat jak indexem, tak i klíčem, který bývá shodný se jménem objektu (vlastnost Name):

```
Debug.Print Worksheets("Sheet1").Visible = True
Debug.Print Worksheets(2).Name
```

Kolekce je zároveň objektem, který poskytuje různé metody a vlastnosti, jež umožňují objekty do kolekce přidávat, ubírat, zjišťovat jejich vlastnosti apod. Tyto metody však nejsou v každém objektovém modelu identické. Například, ve většině aplikací Microsoft Office přidáte nový objekt zavoláním metody Add objektu kolekce:

```
Workbooks.Add
Documents.Add
Worksheets.Add
```

Avšak pokud chcete přidat například novou tabulku do databáze za pomoci knihovny ADOX, musíte nejprve vytvořit novou instanci objektu třídy Table, poté nastavit všechny potřebné vlastnosti a nakonec zavolat metodu Append kolekce Tables ADOX katalogu.

VBA umožňuje vytvořit kolekci vlastní. K manipulaci s jejími objekty potřebujete znát pouze tři metody a jednu vlastnost:

- ◆ **Metoda Add** přidá objekt do kolekce. Musíte předat odkaz na objekt a volitelně i jednoznačný identifikátor (např. jméno, kterým se později můžete na prvek odkazovat).
- ◆ **Metoda Remove** odebere objekt z kolekce. Metodě předáte index objektu nebo jednoznačný identifikátor.
- ◆ **Metoda Item** vrátí konkrétní objekt v kolekci, respektive odkaz na něj (pointer). Jako parametr opět předáte index objektu nebo jednoznačný identifikátor.
- ◆ **Vlastnost Count** vrátí počet objektů v kolekci.

Jedním z obvyklých použití kolekcí v Excelu je extrakce jedinečných hodnot z listu. Níže uvedený kód využívá faktu, že žádný objekt (v tomto případě hodnota) nemůže mít stejný jedinečný identifikátor. Proto je příkazem On Error Resume Next a On Error Goto vypnuto a opět zapnuto ošetření chyb (debugging). Výsledkem je, že kód neohlásí chybu, ale stejná hodnota se dvakrát nepřidá.



01-03 Kolekce.xlsm
modul MCollections

```
Dim Uniques As New Collection
Dim cl As Range, rg As Range
Dim i As Long
With ActiveSheet
    Set rg = .Range("A2", .Cells(.Rows.Count, "A").End(xlUp))
    On Error Resume Next
        For Each cl In rg
            Uniques.Add cl.Value, CStr(cl.Value)
        Next cl
    On Error GoTo 0
    .Range("B1") = "Uniques"
```

```

    For i = 1 To Uniques.Count
        .Cells(.Rows.Count, "B").End(xlUp)(2) = Uniques.Item(i)
    Next i
End With

```

Jiným příkladem může být kolekce, obsahující otevřené uživatelské formuláře (dialogy). Na rozdíl od Accessu, který má kolekci `Forms`, Excel žádnou kolekci typu `Userforms`, která by obsahovala otevřené formuláře, nemá. Můžeme ji však vytvořit:

```

Public MyForms As New Collection
'
Sub ShowForm()
    Dim lNr As Long, strUFName As String
    lNr = 1
    strUFName = "Userform" & CStr(lNr)
    VBA.UserForms.Add(strUFName).Show vbModeless
    MyForms.Add UserForms(0), strUFName
End Sub

```

Tento kód přidá do vlastní kolekce formulář `Userform1` (který jste předtím vytvořili) a otevře ho. Poté se můžete na formulář odkazovat podobně jako v Accessu:

```
MyForms("UserForm1").Caption = "Muj formular"
```

Kolekce i pole mohou být velmi užitečné. Kolekce použijte, pokud potřebujete často přidávat a odebrat prvky, nebo pokud se chcete na prvky odkazovat jejich jménem. Pole jsou většinou rychlejší, ale mají své nevýhody při změně velikosti pole. V případě kolekce všechnu práci udělá VBA za vás.

Deklarace, rozsah a platnost proměnných

Existují dvě základní vlastnosti každé proměnné, nezávisle na jejím typu:

- ◆ **Rozsah proměnné** určuje, které procedury mohou tuto proměnnou použít
- ◆ **Životnost proměnné** určuje, jak dlouho si proměnná podrží hodnotu, která byla do ní vložena

Pokud spustíte opakovaně tento kód:



01-01 Proměnné.xlsm
modul MVarScope

```

Sub Zivotnost()
    Dim Hodnota As Long
    Hodnota = Hodnota + 100
    MsgBox "Hodnota je " & Hodnota
End Sub

```

bude hodnota vždy 100, neboť proměnná `Hodnota` si drží svoji hodnotu, pouze dokud kód nedosáhne řádek `End Sub`. Poté je paměť uvolněna.

Životnost můžete prodloužit příkazem `Static`:

```

Sub Zivotnost2()
    Static Hodnota As Long
    Hodnota = Hodnota + 100
    MsgBox "Hodnota je " & Hodnota
End Sub

```

Pokud nyní budete střídavě spouštět oba kódy, v prvním případě bude hodnota 100, zatímco ve druhém případě se bude hodnota po stovkách zvyšovat. Je to proto, že druhá proměnná si podrží svoji hodnotu i po skončení běhu kódu a proměnná Hodnota v proceduře Zivotnost *není* stejná proměnná jako proměnná hodnota v proceduře Zivotnost2, protože jsou proměnné deklarovány *uvnitř* těla kódu. S oběma proměnnými je nakládáno *nezávisle*.

Pokud proměnnou nadeklaruje v záhlaví modulu (sekce deklarací, která se nachází pod příkazy typu `Option Explicit`), můžete ji použít ve více procedurách:

```
Dim mHodnota As Long
Sub Zivotnost3()
    mHodnota = mHodnota + 10
    MsgBox "Hodnota je " & mHodnota
End Sub
Sub Zivotnost4()
    mHodnota = mHodnota + 100
    MsgBox "Hodnota je " & mHodnota
End Sub
```

Pokud nyní spouštíte oba kódy, proměnná změní svoji hodnotu o 10 nebo o 100 podle toho, kterou proceduru spustíte. Rozsah proměnné jsou všechny procedury v modulu, kde je proměnná deklarována, a proměnná si podrží svoji hodnotu po celou dobu, kdy je sešit s tímto modulem otevřený.

Pokud chcete, aby proměnnou „viděly“ i procedury v jiných modulech, nadeklaruje proměnnou jako „veřejnou“ (klíčové slovo `Public`):

```
Public gHodnota As Long
```



Tip

I když podobně jako autor nedodržíte vždy tzv. Maďarskou konvenci pojmenovávání proměnných, zvykněte si proměnným dávat příponu „m“, pokud je platná pro modul, a „g“, pokud je platná pro celou aplikaci.

Jako odstrašující příklad se podívejte na tento kód, který se výše uvedeným tipem neřídí:

```
Dim Hodnota As Long

Sub Zivotnost()
    Dim Hodnota As Long
    Hodnota = Hodnota + 100
    MsgBox "Hodnota je " & Hodnota
End Sub
```

Proměnná deklarovaná na úrovni modulu bude pro proceduru neviditelná, protože proměnná s tímto jménem je deklarována uvnitř těla procedury a tato má prioritu! Pokud užijete předponu, na první pohled budete vědět, jestli se odkazujete na lokální, modulovou, nebo globální proměnnou.

Otázky

Otázka 1.1:

Pokud potřebujete uložit do proměnné číslo posledního řádku jenž obsahuje data jenž typ proměnné použijete a proč?

Otázka 1.2:

Proč je následující kód nebezpečný a jak ho můžete upravit, aby plnil zamýšlený účel? Uvedte alespoň dvě možnosti.

```
Dim dblHodnota As Double  
  
Do  
    dblHodnota = dblHodnota + 0.01  
    If dblHodnota = 1 Then Exit Do  
Loop
```

Otázka 1.3:

Ve kterých případech je nutné použít proměnnou typu Variant? A ve kterém, pro Excel specifickém případě je její použití výhodné?

Otázka 1.4:

Proč je nebezpečné použití výrazu Option Base 1 v záhlaví modulu?

Otázka 1.5:

Napište co nejrychlejší kód, který v buňkách A1:A1000 odřízne poslední znak.

Otázka 1.6:

Napište kód, který do prvku typu listbox ve formuláři při jeho otevření nahraje data z prvního listu, od buňky A2 dolů, o neznámém počtu prvků.

Otázka 1.7:

Proč je výhodné využít vlastní kolekci k vygenerování jedinečných hodnot ze seznamu?

Otázka 1.8:

Jak nadeklarujete proměnnou, aby si svou hodnotu uchovala i poté, co kód doběhne?

Důležité konstrukce VBA použité v knize

With ... End With



01-04 VBA Konstrukce.xlsm
modul MConstructions

Pro příklad předpokládejme, že chcete ve VBA sečíst hodnoty na listu „Prodeje“ ve sloupci C od druhého řádku po poslední buňku podle obrázku 1.2.

Pokud je list „Prodeje“ aktivní, následující kód bude pracovat správně,

```
Dim Soucet As Double
Soucet = Application.Sum(Range(Cells(2, "C"), Cells(Rows.Count, "C").End(xlUp)))
Debug.Print Soucet
```

a vrátí hodnotu 2700 do okna Immediate:

```
Sub SectiAktivniList()
Dim Soucet As Double
Soucet = Application.Sum(Range(Cells(2, "C"), Cells(Rows.Count, "C").End(xlUp)))
Debug.Print Soucet
End Sub
```

Immediate

2700

Obrázek 1.3: Výpis výsledku procedury v okně Immediate

Problém ale nastane, pokud list „Prodeje“ není aktivní. Potom musíme vlastnost Cells (a raději i Rows) kvalifikovat správným objektem:

```
Dim Soucet As Double
Soucet = Application.Sum(Range(Sheets("Prodeje").Cells(2, "C"), _
    Sheets("Prodeje").Cells(Sheets("Prodeje").Rows.Count, "C") _
    .End(xlUp)))
Debug.Print Soucet
```

Tento kód se však velmi rychle čte a navíc, pokud byste jméno listu změnili, musíte kód upravit na třech (!) různých místech. Zdůrazňujeme, že se jedná o jeden řádek kódu, pro snazší čitelnost v editoru VBE zalomený mezerou a podtržítkem do více řádků. Právě v tomto případě je vhodné použít konstrukci With...End With:

```
Dim Soucet As Double
With Sheets("Prodeje")
    Soucet = Application.Sum(.Range(.Cells(2, "C"), _
        .Cells(.Rows.Count, "C").End(xlUp)))
End With
Debug.Print Soucet
```

Když nyní změníte jméno listu, musíte změnit kód na jednom jediném místě. V praxi by bylo ještě lepší jméno uložit do konstanty a umístit hned na začátek kódu. Případná úprava pak bude provedena velmi rychle a spolehlivě. Rovněž jsme si mohli dovolit umístit dodatečnou kvalifikaci před

	A	B	C	D
1	Město	Měsíc	Prodeje	
2	Praha	Leden	150	
3	Praha	Únor	350	
4	Praha	Březen	300	
5	Brno	Leden	400	
6	Brno	Únor	500	
7	Brno	Březen	100	
8	Ostrava	Leden	200	
9	Ostrava	Únor	450	
10	Ostrava	Březen	250	
11				

Obrázek 1.2: Úkolem je sečíst čísla ve sloupci C

objekt Range, což sice v tomto případě není nutné, ale kód je čistší, protože je zřejmé, že vrácená oblast bude patřit do listu jmenovaného za klíčovým slovem With.

Doporučujeme vám nikdy nepsat kód, který se implicitně spoléhá na aktivní list, protože to může vést k těžko odstranitelným chybám a navíc, kód vám může například **nenávratně přepsat a uložit data na aktivním listu!!!** Pokud má kód skutečně pracovat pro kterýkoli aktivní list, můžete vždy použít explicitní definici:

```
Dim Soucet As Double
With ActiveSheet
    Soucet = Application.Sum(.Range(.Cells(2, "C"), _
        .Cells(.Rows.Count, "C").End(xlUp)))
End With
Debug.Print Soucet
```

Pokud se později rozhodnete pro omezení, pro které listy má kód pracovat, změníte pouze jméno listu, případně zkontrolujete jméno aktivního listu v bloku If - End If. Zbytek kódu můžete nechat beze změny.

Na závěr uvedme, že bloky With...End With je možno i vnořovat:

```
With Sheets("List1")
    With .Cells(.Rows.Count, "A").End(xlUp).Resize(, 3)
        .Offset(1).FormulaR1C1 = "=SUM(R2C:R[-1]C)"
        .Borders(xlEdgeBottom).LineStyle = xlDouble
    End With 'poslední řádek
End With ' List1
```

Uvedený kód dvakrát podtrhne poslední řádek na listu „List1“ a vloží vzorce sčítající sloupce:

Všimněte si komentářů za klíčovými slovy End With. Pokud je počet řádků uvnitř bloku With...End With dlouhý, je vhodné označit, který z bloků se uzavírá. Na výsledek to samozřejmě nemá vliv, ale usnadňuje to čitelnost kódu.

	A	B	C	D
1	A	B	C	
2	15	22	21	
3	23	11	8	
4	3	5	17	
5	12	18	14	
6	1	20	2	
7	10	9	16	
8	7	13	4	
9	19	24	6	
10				
11				



	A	B	C	D
1	A	B	C	
2	15	22	21	
3	23	11	8	
4	3	5	17	
5	12	18	14	
6	1	20	2	
7	10	9	16	
8	7	13	4	
9	19	24	6	
10	90	122	88	
11				

Obrázek 1.4: Vložení součtů a podtržení posledního řádku, jehož pozici předem neznáme

Blok If ... Else ... End If

Tento blok je jedním z nejdůležitějších ve VBA a má několik variant.

Nejjednodušší varianta je zápis na jednom řádku:

```
If Intersect(Selection, Rows(1)) Is Nothing Then Exit Sub
```

Výše uvedený kód zajistí, že pokud je vybrána jakákoli oblast kromě oblasti náležející do prvního řádku aktivního listu, program se okamžitě ukončí.

Nenechte se zmást případným rozdělením řádku mezerou a podtržítkem na dva řádky. Jedná se stále o jeden řádek kódu:


```
If IsEmpty(c1.Offset(.1)) Then _
    c1.Value = "No Data"
```

Čili pokud je buňka nalevo od buňky přiřazené do objektové proměnné prázdná, do buňky je vložen text „No Data“.

Běžnější je použití `If` v bloku, kdy příkazy nacházející se uvnitř těla bloku budou vykonány, pouze je-li splněna podmínka mezi klíčovými slovy `If` a `Then`:

```
If Err.Number = 0 Then
    Call ImportData
    Call FormatSheet
End If
```

Pokud nedošlo k chybě, kód zavolá procedury, které naimportují data a zformátují list.

Blok `If` může vykonat jednu sadu příkazů, pokud je podmínka splněna, a jinou sadu příkazů, pokud je podmínka nesplněna:

```
If Plat > 100000 Then
    Dan = 30000 + (Plat-100000) * 0.40
Else
    Dan = Plat * 0.30
End If
```

Program spočítá daň, která je nad 100000 progresivní a stoupne z 30 % na 40 %.

Blok `If` může mít rovněž jednu nebo více sekcí `ElseIf`, pokud potřebujeme rozlišit mezi více podmínkami. Typickou úlohou je podbarvení buněk podle obsahované hodnoty (ačkoli v praxi tuto úlohu lépe řeší Podmíněný formát).



01-04 VBA Konstrukce.xlsm
modul MConstructions

```
Dim c1 As Range
For Each c1 In Range("A:A").SpecialCells(xlCellTypeConstants)
If c1.Value > 8 And c1.Value <= 10 Then
    c1.Interior.Color = vbGreen
ElseIf c1.Value > 5 And c1.Value <= 8 Then
    c1.Interior.Color = vbYellow
ElseIf c1.Value > 0 And c1.Value <= 5 Then
    c1.Interior.Color = vbRed
Else
    c1.Interior.Color = xlNone
End If
Next
```

Kód změní barvu pozadí buňky na červenou pro hodnoty mezi 8 a 10 včetně, na žlutou pro hodnoty od 5 do 8 včetně, a červenou pro hodnoty mezi 0 a 5 včetně. Pro ostatní buňky barvu pozadí změní na průhlednou (standardně bílou).

Pokud je klíčové slovo `If` následováno jedním nebo několika testy `ElseIf`, VBA provádí test, dokud nenajde splněnou podmínku. Vykoná příkazy v tomto bloku a pokračuje příkazem, který se nachází pod řádkem `End If`. Pokud není splněna ani jedna podmínka, jsou vykonány příkazy v sekci `Else`, pokud existuje.

Pokud není splněna ani jedna podmínka a sekce `Else` chybí, blok `If` nevykoná nic.

Bloky `If` můžete do sebe vnořovat:



01-04 VBA Konstrukce.xlsm
modul MConstructions

```
Dim Ans
Do
    Ans = InputBox("Zadejte číslo od 1 do 5")
    If Len(Ans) <> 0 Then
        If IsNumeric(Ans) Then
            If Ans < 1 Or Ans > 5 Then
                'číslo přesahuje hranice
                MsgBox "Číslo musí být od 1 do 5"
            Else
                'testy splněny, pokračuj
                Exit Do
            End If
        Else
            'Nesplněn test na číslo
            MsgBox "Musíte zadat číslo"
        End If
    Else
        'Žádné zadání, ukončit
        Exit Sub
    End If
Loop
```

Kód zavolá funkci VBA `Inputbox` a testuje zadání. Pokud není zadáno nic (klepnuto na tlačítko „Cancel“), kód se ukončí. Pokud není zadáno číslo, je zobrazeno varovné hlášení a dialogové okno se znovu otevře. Pokud číslo nesplňuje podmínku zadání, zobrazí se jiné varovné hlášení a dialogové okno se znovu otevře. Pokud jsou podmínky splněny, program pokračuje příkazem `Exit Do` a řádkem následujícím za klíčovým slovem `Loop`. V praxi se v podobném případě často ještě zavádí čítač, umožňující omezený počet pokusů.

Blok `Select Case`

Blok `Select Case` má mnoho společného s blokem `If...ElseIF...Else...End If`, ale je podle našeho názoru elegantnější. Předpokládejme, že máme v listu sloupec s údajem o věku a chceme spočítat jízdné, které je 7 korun pro děti od 4 do 7 let včetně, 13 korun pro děti od 8 do 15 let, 26 korun pro osoby nad 15 let, a jízdné neplatí děti do tří let a senioři nad 66 let.

Funkce bude vypadat takto:

```
Function Jizdne(Vek As Long)
    Select Case Vek
        Case 0 To 3, Is > 66
            Jizdne = 0
        Case 4 To 7
            Jizdne = 7
        Case 8 To 15
            Jizdne = 13
    End Select
End Function
```

```

Case Is > 15
    Jizdne = 26
Case Else
    Jizdne = CVErr(xlErrNA)
End Select
End Function

```

V listu ji použijeme jako každou jinou vestavěnou funkci Excelu:

Funkce rozhodne o výši jízdného, a pokud žádná podmínka není splněna, vrátí chybovou hodnotu. Pokud máte pro každou podmínku jeden příkaz, můžete vložit dvojtečku mezi podmínku a příkaz a zapsat je na jeden řádek:



01-04 VBA Konstrukce.xlsm
modul MFunctions

```

Function Jizdne(Vek As Long)
    Select Case Vek
        Case 0 To 3, Is > 66: Jizdne = 0
        Case 4 To 7: Jizdne = 7
        Case 8 To 15: Jizdne = 13
        Case Is > 15: Jizdne = 26
        Case Else: Jizdne = CVErr(xlErrNA)
    End Select
End Function

```

Pro větší přehlednost jsme příkazy oddělili tabulátorem.

VBA považuje příkazy oddělené dvojtečkou za samostatné řádky. Tento způsob zápisu může váš kód zkrátit a zpřehlednit (na rychlost provádění samozřejmě vliv nemá).

Tento způsob zápisu se často používá u rutinních příkazů, které následují za sebou, například tento kód zavře sadu záznamů a odstraní objektové proměnné z paměti

```
rs.Close: Set rs = Nothing
```

a tento kód ukončí automatizovanou aplikaci a odstraní objektové proměnné z paměti:

```
objWord.Quit: Set objWord = Nothing
```

Zajímavým a užitečným příkladem je automatické přihlášení na webovou stránku, např. na seznam.cz:



01-04 VBA Konstrukce.xlsm
modul MConstructions

```

Sub OpenSeznamAndLogin()
    Dim objIE As Object
    On Error GoTo Err_Handler
    Set objIE = CreateObject("InternetExplorer.Application")
    With objIE
        .Navigate "http://www.seznam.cz"
    End With

```

	A	B	C	D	E
1					
2		Školní výlet			
3					
4			Věk	Jízdné	
5		Ředitel	67	0	
6		Učitelka	25	26	
7		Pepík	3	0	
8		Honza	4	7	
9		Martin	6	7	
10		Věra	8	13	
11		Šárka	7	7	
12		Bohouš	8	13	
13		David	3	0	
14		Robin	7	7	
15		Alena	?	#HODNOTA!	
16					

Obrázek 1.5: Použití vlastní funkce jako vzorce v listu

```

Do While .Busy Or .ReadyState <> 4: DoEvents: Loop

.Visible = True
With .document
    .all.Item("login").Value = "VaseJmeno"
    .all.Item("password").Value = "VaseHeslo"
    .forms("login-fom").Submit
End With
.Visible = True
End With
ExitHere:
Set objIE = Nothing
Exit Sub
Err_Handler:
MsgBox "Unexpected Error, I'm quitting."
objIE.Quit
Resume ExitHere
End Sub

```

Všimněte si standardní smyčky `Do...Loop`, která čeká na úplné načtení stránky, než zapíše uživatelské jméno a heslo a formulář odešle na server k přihlašovacímu procesu. Je nutno podotknout, že automatizace přihlášení a vůbec interakce s HTML kódem webových stránek velmi závisí na designu stránky. Pokud seznam.cz stránku změnil (testovali jsme v srpnu 2009), je možné, že tento příklad již fungovat nebude. Příklad obsahuje standardní postup ošetření chyb, o kterém pojednáme později.

Zajímavým způsobem využití bloku `Select Case` je jeho obrácené použití. Místo, abychom za klíčová slova `Select Case` zapsali výraz, který chceme testovat, zapíšeme za ně očekávanou hodnotu a testujeme jednotlivé výrazy na rovnost.

Typické použití je při testování ovládacích prvků typu Přepínač (OptionButton) na listu. Tyto prvky umožňují většinou vybrat jen jednu hodnotu z několika, které se vzájemně vylučují. Proto se tento ovládací prvek také někdy nazývá Radio Button jako vzpomínka na staré radiopřijímače, kdy zmačknutí mechanického knoflíku pro jedno pásmo přinutilo knoflík ostatních pásem vyskočit.

V našem příkladu jsme na list umístili dva ovládací prvky typu Přepínač. Jména jsme změnil na `optMale` (označuje muže) a `optFemale` (označuje ženu). Úkolem je zobrazit zprávu o pohlaví osoby zapsané v buňce C2.

Úkol vyřeší tento kód:

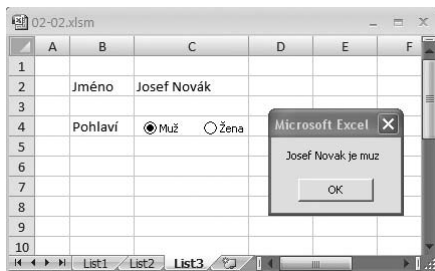


01-04 VBA Konstrukce.xlsm
modul MConstructions

```

With ActiveSheet
    Select Case 1
        Case .Shapes("optMale").ControlFormat
            MsgBox "[C2] & " je muz"

```



Obrázek 1.6: Získání hodnoty podle stisknutého přepínače

```

Case .Shapes("optFemale").ControlFormat
    MsgBox "[C2] & " je žena"
End Select
End With

```

Pokud je přepínač v poloze „Zapnuto“, vrací vlastnost `Value` hodnotu 1. Po nalezení prvku, který tuto hodnotu splňuje, se vypíše zpráva a program se ukončí, nebo pokračuje dál příkazem na řádku následujícím za řádkem `End Select`.

For ... Next

Smyčka `For...Next` je velmi běžným cyklem v procedurách VBA a jeho použití musíte bezpečně ovládat. Obsahuje vnitřní čítač, který je automaticky při každém průchodu zvyšován o zvolený krok, standardně o jedničku. Příkazy uvnitř těla cyklu jsou vykonávány tak dlouho, dokud čítač nepřekročí definovanou maximální hodnotu. Typické použití ve VBA pro Excel je procházení přes všechny prvky pole. Následující příklad načte do pole hodnoty v buňkách A2:A10, vynásobí je samy sebou (neboli vypočítá druhou mocninu) a výsledek zapíše do buněk B2:B10:



01-04 VBA Konstrukce.xlsm
modul MConstructions

```

Sub Mocnina()
    Dim y, i As Long
    y = Range("A2:A10")
    For i = LBound(y, 1) To UBound(y, 1)
        y(i, 1) = y(i, 1) ^ 2
    Next i
    Range("B2:B10") = y
End Sub

```

Pole ve VBA mohou mít obecně jakýkoli celočíselný spodní i horní index (horní hranice je v rámci omezení VBA, samozřejmě). Spodní hranice záleží na případné klauzuli `Option Base` v záhlaví modulu, ale i na způsobu načtení pole. Například pokud načteme do pole hodnoty z listu, bude pole vždy dvourozměrné a prvek bude mít indexy 1,1.

Proto doporučujeme při iteraci přes všechny prvky pole *vždy* používat funkce `LBound()` a `UBound()` pro zjištění spodního i horního indexu, abyste první nebo poslední prvek nevynechali.

Další použití je procházení přes prvky, které *nejsou* součástí kolekce. Pro procházení prvků v kolekci je vhodnější cyklus `For...Each`, kterému se budeme věnovat v další části této lekce. Následující příklad není příliš užitečný, ale demonstruje procházení přes jednotlivé znaky v buňce. Liché znaky mění na velká písmena a sudá na malá. Navíc font lichých znaků změní na tučný. Pokud tedy buňka A1 obsahuje slovo `Velbloud`, výsledkem bude `VeLbLoUd`:

```

Sub Velbloud()
    Dim i As Long
    With Range("A1")
        For i = 1 To .Characters.Count
            With .Characters(i, 1)
                If i Mod 2 = 1 Then
                    .Text = UCase(.Text)
                    .Font.Bold = True
                End If
            End With
        Next i
    End With
End Sub

```

```

Else
    .Text = LCase(.Text)
End If
End With
Next i
End With
End Sub

```

Pokud v cyklu `For...Next` vynecháme klíčové slovo `Step`, bude se čítač zvyšovat o jedničku. Můžeme však definovat krok jiný, například pokud chceme ve sloupcích změnit barvu každého druhého řádku, počínaje druhým řádkem listu (prvním řádkem dat) a konče řádkem posledním (předem neznámým):

```

Sub AlternateRows()
Dim lLastRow As Long, i As Long
lLastRow = Cells(Rows.Count, "A").End(xlUp).Row
For i = 2 To lLastRow Step 2
    With Cells(i, "A").Resize(, 2)
        .Interior.Color = RGB(127, 127, 127)
        .Font.Color = vbWhite
    End With
Next i
End Sub

```

Čítač se nemusí nutně zvyšovat inkrementálně. Můžeme definovat i krok záporný. Následující příklad, který mění pořadí znaků v buňce A1, rovněž není velmi praktický:

```

Sub Shenanigan()
Dim i As Long
Dim tmp As String
With Range("A1")
    For i = Len(.Value) To 1 Step -1
        tmp = tmp & Mid$(.Value, i, 1)
    Next i
    .Value = tmp
End With
End Sub

```

Častějším případem, kdy je nutné použít zpětnou smyčku, je mazání vybraných řádků. Algoritmus nemůže postupovat dopředu, protože by při vymazání řádku byl další řádek přeskočen. Pokud by čítač dosáhl hodnoty 3 a řádek splňoval podmínku, řádek by byl vymazán a původní čtvrtý řádek by postoupil na místo třetí. Čítač však mezitím nabude hodnoty čtyři a testuje původně pátý řádek. Čtvrtý řádek bude při testu přeskočen.

Přestože pro mazání vybraných řádků existují účinnější techniky, které podrobně představíme ve čtvrté lekci, pro malé objemy dat je možno použít následující kód:

Původní stav			Výsledný stav		
	A	B		A	B
1	Hodnota	Mocnina	1	Hodnota	Mocnina
2	11	121	2	11	121
3	12	144	3	12	144
4	13	169	4	13	169
5	14	196	5	14	196
6	15	225	6	15	225
7	16	256	7	16	256
8	17	289	8	17	289
9	18	324	9	18	324
10	19	361	10	19	361

Obrázek 1.7: Střídané vybarvení řádků

```

Sub DeleteEmptyRows()
Dim lLastRow As Long, i As Long
lLastRow = Cells.SpecialCells(xlLastCell).Row
For i = lLastRow To 2 Step -1
    If Application.CountA(Rows(i)) = 0 Then
        Rows(i).Delete
    End If
Next i
End Sub

```

Kód využívá vestavěný vzorec POČET2() (v angličtině COUNTA), který sečte neprázdné buňky v předané oblasti.



Tip

Přestože to není zcela v souladu se zásadami strukturovaného programování, někdy je vhodné cyklus předčasně ukončit, například proto, že kód narazil na buňku s chybovou hodnotou, nebo aby se neprováděly nadbytečné příkazy. Bývá dobrým zvykem, aby cyklus obsahoval pouze jeden příkaz předčasného opuštění cyklu `Exit <typ cyklu>`.

Následující funkce vrátí `True`, pokud se řetězec předaný do proměnné `Zakaznik` vyskytuje v poli „Zakaznici“:

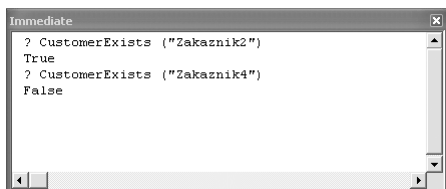


01-04 VBA Konstrukce.xlsm
modul MFunctions

```

Function CustomerExists(Zakaznik As String) As Boolean
Dim Zakaznici
Dim i As Long
Zakaznici = Array("Zakaznik1", "Zakaznik2", "Zakaznik3")
For i = LBound(Zakaznici) To UBound(Zakaznici)
    If Zakaznici(i) = Zakaznik Then
        'zákazník nalezen, není nutné hledat dále
        CustomerExists = True
        Exit For
    End If
Next i
End Function

```



Obrázek 1.8: Použití okna Immediate pro testování výsledku vlastní funkce

Funkci můžete ověřit v okně Immediate:

Po napsání řádku začínajícího otazníkem (ekvivalent příkazu `Debug.Print`) stiskněte `Enter`.



Tip

Patří k dobrým programátorským praktikám nepoužívat proměnnou, obsahující čítač cyklu `For . . . Next` k žádnému jinému účelu, než právě k řízení cyklu. Přestože se chová jako kterákoli jiná proměnná a je možné jí přiřadit novou hodnotu dokonce i uvnitř cyklu, tento postup zásadně odporuje principům strukturovaného programování a váš kód se stává prakticky nečitelným i pro vás jako autora. Navíc je nutné si uvědomit, že čítač mění hodnotu při každém průchodu přes klíčové slovo `Next` a novou hodnotu porovnává s horní hranicí. Pokud je hodnota čítače vyšší (nebo nižší v případě záporného kroku), cyklus je ukončen a program pokračuje dalším řádkem pod klíčovým slovem `Next`. Čítač tedy po opuštění cyklu není roven horní hranici!

For Each ... Next

Cyklus `For Each . . . Next` je v procedurách pro Excel zřejmě ještě běžnější než cyklus `For . . . Next`. Je tomu tak proto, že v objektovém modelu Excel se většina běžných objektů vyskytuje v *kolekcích* a cyklus `For Each . . . Next` je ušit na míru právě procházení prvků v kolekci.

Nejenže je použití cyklu `For Each . . . Next` pro kolekce bezpečnější, ale je i o něco rychlejší, jak se můžeme přesvědčit například na testu těchto nic zvláštního nedělajících procedur:



01-04 VBA Konstrukce.xlsm
modul MConstructions

```
Sub TestForEach()

Const NR_LOOPS = 1000
Dim lCounter As Long

Dim ws As Worksheet
For lCounter = 1 To NR_LOOPS
    For Each ws In ThisWorkbook.Sheets
        Debug.Print ws.Name
    Next ws
Next lCounter

End Sub

Sub TestForNext()

Const NR_LOOPS = 1000
Dim lCounter As Long

Dim i As Long
For lCounter = 1 To NR_LOOPS
    For i = 1 To ThisWorkbook.Sheets.Count
        Debug.Print Sheets(i).Name
    Next i
Next lCounter

End Sub
```


Procedury vypíše tisíckrát jména všech listů do okna Immediate. Přestože časový rozdíl není v absolutní hodnotě velký, druhá procedura trvá zhruba o 25 % déle.

Okno Immediate lze vymazat umístěním kurzoru do něj (aby okno získalo fokus), stisknutím Ctrl+A (vybere veškerý text) a následným stisknutím klávesy Delete. Objektový model VBA projektu bohužel neumožňuje přímou manipulaci s obsahem tohoto okna, ale uvedenou kombinaci kláves můžeme nasimulovat pomocí funkce SendKeys:

```
Sub ImmediateWindowClear()
    Application.VBE.Windows.Item("Immediate").SetFocus
    Application.SendKeys "^a"
    Application.SendKeys "{Del}"
End Sub
```

Následující příklad prochází kolekci souborů ve zvoleném adresáři a vypíše jejich jména a velikost v kilobytech:

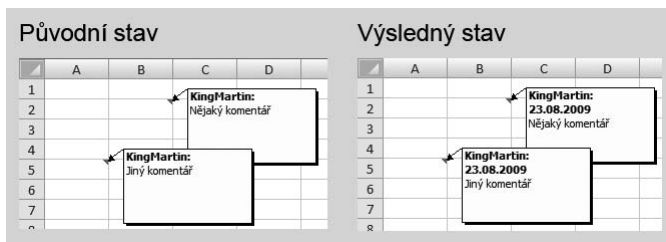


```
Sub FileList()
    Const FOLDER As String = "C:\"
    Dim fso As Object, fld As Object, f As Object
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set fld = fso.GetFolder(FOLDER)
    For Each f In fld.Files
        Debug.Print f.Name & vbTab & Format$(f.Size \ 1024, "#,##0")
    Next
    Set fso = Nothing
End Sub
```

Kód využívá knihovnu VB skriptu, která leží mimo objektový model Excelu. Více se o ní dozvíte v sedmé lekci, kde se budeme zabývat hromadným zpracováním dat z více sešitů.

Jako poslední příklad uvedeme proceduru, která za jméno autora komentáře vloží dnešní datum:

```
Sub Comments()
    Dim cmt As Comment
    For Each cmt In ActiveSheet.Comments
        cmt.Text vbLf & Date, Len(cmt.Author) + 2, False
    Next cmt
End Sub
```



Obrázek 1.9: Vložení aktuálního data do všech komentářů

Obecným pravidlem je používat cyklus `For...Next` pro procházení prvky pole a cyklus `For Each...Next` pro procházení prvků kolekce. Pro prvky v kolekcích použijte cyklus `For...Next` tehdy, pokud potřebujete jiný krok než jedna, nebo krok záporný.

Podobně jako cyklus `For...Next`, i cyklus `For Each...Next` lze předčasně ukončit. Následující procedura prochází buňky v oblasti A1:A10 aktivního listu. Pokud buňka obsahuje hodnotu, interpretovatelnou jako číslo, vynásobí se dvěma. Pokud hodnota číselná není, zobrazí se varovné hlášení, smyčka se předčasně ukončí a kód pokračuje řádkem `End Sub`.

```
Sub ExitForEach()
Dim c1 As Range
For Each c1 In Range("A1:A10")
    If IsNumeric(c1) Then
        c1 = c1 * 2
    Else
        MsgBox "Bunka " & c1.Address & " neobsahuje číslo!"
        Exit For
    End If
Next
End Sub
```

Do ... Loop

Cyklus `Do...Loop` je ve VBA pro Excel používána méně než cykly, které jsme popsali v předešlých částech lekce. Je tomu tak proto, že v Excelu a vůbec ve všech aplikacích Microsoft Office se většina objektů nachází v kolekcích o konkrétním počtu (vlastnost `Count`), případně lze horní hranici (například poslední řádek) snadno nalézt.

Konstrukce `Do...Loop` se používá v případech, kdy je počet průchodů předem neznámý. Proto je její použití obvyklé zejména při manipulaci s objekty, které se nacházejí mimo objektový model Microsoft Office.

V Excelu se cyklus `Do...Loop` vyskytuje většinou ve spojení s metodou `Find` objektu `Range`, kdy je počet nalezených buněk předem neznámý. Následující příklad předpokládá, že na aktivním listu jsou vzorce odkazující na list „List1“, typu „=List1!A1“. Úkolem je vzorce „přepojit“ na list „List2“, čili původní vzorec bude změněn na „=List2!A1“. Toto může být velmi důležitá technika, která vám může ušetřit ohromné množství času při konsolidaci dat.



01-04 VBA Konstrukce.xlsm
modul MConstructions

```
Sub PrepojList()
Const OldSheet As String = "List1"
Const NewSheet As String = "List2"

Dim c1 As Range
With ActiveSheet
    Set c1 = .Cells.Find(OldSheet, LookIn:=xlFormulas, LookAt:=xlPart)
    If Not c1 Is Nothing Then
        'alespoň jedna buňka nalezena
        Do
            c1.Formula = Replace$(c1.Formula, OldSheet, NewSheet)
        Loop
    End If
End With
```

```

        Set c1 = .Cells.FindNext
    Loop Until c1 Is Nothing
End If
End With

End Sub

```

Procedura nejprve zavolá poprvé metodu `Find` a pokusí se nalézt buňku, která obsahuje hledaný výraz. Všimněte si, že je metoda doplněna o parametry určující, že má hledat ve vzorcích, a ne ve vypočítaných hodnotách, a rovněž že se má vyhledávat část vzorce, a ne obsah celé buňky.

Pokud je alespoň jedna buňka nalezena, procedura vstoupí do smyčky `Do...Loop`, a protože je podmínka opuštění cyklu až za výrazem `Loop`, příkazy se vykonají alespoň jednou. Funkce `Replace()` provede nahrazení starého řetězce řetězcem novým a poté se zavolá metoda `FindNext`, která se pokusí nalézt následující buňku. Podmínka za klíčovým slovem `Loop` zkontroluje, zda byla další buňka nalezena. Pokud ano, program pokračuje řádkem pod klíčovým slovem `Do`. Výraz

```
Loop Until c1 Is Nothing
```

volně přeloženo znamená „pokračuj ve smyčce, *dokud* není proměnná `c1` prázdná, jinak řečeno, další buňka nebyla nalezena“.

Jinou variantou na tutéž podmínku by bylo použití podmínky `While`:

```
Loop While Not c1 Is Nothing
```

Což znamená, „pokračuj ve smyčce, *pokud* je proměnná `c1` neprázdná“. Čeština má pro oba případy podobnou gramatiku, ale angličtina mezi oběma případy zřetelně odlišuje. V každém případě je jedno, jestli použijete variantu `Until` nebo variantu `While`. Použijte tu, jejíž logika je vám bližší.

Zajímavé je, že nápověda Microsoft Excel uvádí tento příklad použití metody `Find()`:

```

With Worksheets(1).Range("A1:A500")
    Set c = .Find(2, LookIn:=xlValues)
    If Not c Is Nothing Then
        firstaddress = c.Address
        Do
            c.Value = 5
            Set c = .FindNext(c)
        Loop While Not c Is Nothing And c.Address <> firstaddress
    End If
End With

```

Kód má za úkol vyhledat buňky, obsahující hodnotu 2, a nahradit ji hodnotou 5. Pokud definovaná oblast žádnou hodnotu 2 neobsahuje, vše proběhne normálně a kód dle očekávání nevykoná nic.

Pokud ovšem alespoň jedna taková buňka existuje, kód vstoupí do smyčky a pokračuje v hledání a nahrazování tak dlouho, až byly všechny buňky nahrazeny a výraz

```
Set c = .FindNext(c)
```

vrátí objekt `Nothing`, neboli „nic“. Takový objekt ovšem nemůže mít vlastnost `Address`, a protože se u složených podmínek vždy vyhodnocují všechny jejich části, výraz

```
c.Address <> firstaddress
```

způsobí chybu za běhu programu! Navíc je tato podmínka zbytečná. Vzhledem k tomu, že hledané hodnoty nahrazujeme, víme, že dříve nebo později už nová hodnota nebude nalezena, a podmínka

```
While Not c Is Nothing
```

stačí ke spolehlivému ukončení smyčky.

Bylo by ovšem chybou si myslet, že porovnávání první nalezené buňky s buňkou aktuálně nalezenou je zbytečná technika. Metoda `Find` a `FindNext` se sama od sebe neukončí, jak se sami můžete přesvědčit v dialogu „Najít a nahradit“. Zatímco například aplikace Word vám po nalezení posledního výskytu oznámí, že „Aplikace Word dokončila prohledávání dokumentu“, aplikace Excel po nalezení poslední buňky nalezne znovu buňku první a cyklus se opakuje donekonečna.

V předešlém kódu se nalezené hodnoty nahrazovaly hodnotami jinými, a proto po nalezení poslední buňky už první buňka opět nalezena nebyla a cyklus se ukončil. Pokud ovšem nalezené hodnoty nenahrazujeme, musíme adresu nalezené buňky porovnávat s adresou buňky, která byla nalezena jako první, a cyklus při splnění rovnosti ukončit.

Následující příklad vyhledává ve sloupci A hodnotu 2, a pokud je nalezena, odsadí text v sousední buňce. Toto může být užitečné například pro zpřehlednění hierarchie výrobků:

Do kódu můžeme „pro jistotu“ vložit podmínku, že při žádné další nalezené buňce se cyklus okamžitě ukončí. Patří k dobrým programátorským praktikám předvídat i mimořádné okolnosti, i když, jak jsme vysvětlili, by kód na tomto příkladu dat nenalezenou buňkou skončit neměl.

Původní stav			Výsledný stav		
	A	B		A	B
1	Úroveň	Název	1	Úroveň	Název
2	1	Skupina1	2	1	Skupina1
3	2	Výrobek1	3	2	Výrobek1
4	2	Výrobek2	4	2	Výrobek2
5	1	Skupina2	5	1	Skupina2
6	2	Výrobek3	6	2	Výrobek3
7	2	Výrobek4	7	2	Výrobek4
8	1	Skupina1	8	1	Skupina1
9	2	Výrobek5	9	2	Výrobek5
10	2	Výrobek6	10	2	Výrobek6

Obrázek 1.10: Automatické odsazení textu v buňkách druhé úrovně



01-04 VBA Konstrukce.xlsm
modul MConstructions

```
Sub IndentLevel()
Const LEVEL As Long = 2
Dim c1 As Range
Dim FirstFound As String
With ActiveSheet
Set c1 = .Range("A:A").Find(LEVEL, LookIn:=xlValues, LookAt:=xlWhole)
If Not c1 Is Nothing Then
FirstFound = c1.Address
Do
c1.Offset(, 1).InsertIndent 2
Set c1 = .Range("A:A").FindNext(c1)
If c1 Is Nothing Then Exit Do
Loop Until c1.Address = FirstFound
End If
End With
End Sub
```

Procedura se pokusí najít první buňku s hodnotou 2. Pokud není hodnota nalezena, kód pokračuje řádkem `End With` a ukončí se. Pokud je první buňka nalezena, uloží se její adresa do proměnné `FirstFound`. Kód poté pokračuje vstupem do smyčky `Do...Loop`, uvnitř které vykonává vlastní odsazení, pokusí se nalézt další buňku a pro jistotu se smyčka okamžitě ukončí příkazem `Exit Do`, pokud buňka nebyla nalezena. Adresa nově nalezené buňky se porovná s uloženou adresou, a pokud je různá, smyčka pokračuje.

Ošetření chyb

V předešlých částech lekce jsme se zabývali standardními strukturami jazyka VBA. Jednalo se o konstrukce

- ◆ `With...End With`
- ◆ `If...Else...Then`
- ◆ `Select Case...Case...End Select`
- ◆ `For...Next`
- ◆ `For Each...Next`
- ◆ `Do...Loop`

Všimněte si, že každá z těchto konstrukcí má jasnou strukturu a je zřetelně patrné, kde je její začátek a konec. Jak jsme již vysvětlili, cykly je možno předčasně opustit, ale to by mělo být jen ve výjimečných případech. Strukturované konstrukce mají svůj jasný vstup, daný konkrétním klíčovým slovem, a po vykonání příkazů kód pokračuje prvním řádkem následujícím za klíčovým slovem, které strukturu ukončuje.

Existuje ovšem ještě další sada příkazů, která řídí běh kódu:

- ◆ `GoTo Návěští`
- ◆ `Resume`
- ◆ `Resume Návěští`
- ◆ `Resume Next`

Tyto příkazy (a platí to zejména pro příkaz `GoTo`) mohou provádět odskoky na libovolné místo programu, označené návěští, což je řádek s libovolným textem následovaným dvojtečkou. V běžném programu je toto velmi špatná praktika, neboť program se stává velice špatně čitelným a odladění chyb je čirou noční můrou. Příkaz `GoTo` pochází z dob, kdy ještě neexistovaly moderní programovací struktury, a program byl řízen příkazy `If` a odskoky na návěští při splnění podmínce.

Ve VBA by se tyto příkazy měly používat pouze ve výjimečných případech a takovou výjimkou je právě ošetření chyb za běhu programu (Run-time errors). Chyby v programu je třeba rozlišovat na chyby syntaktické, algoritmické a run-time chyby.

Chyby syntaktické jsou ve velké většině odchyceny již při zápisu programu, při jeho kompilaci, při prvním krokování pomocí klávesy F8, či při pokusu o spuštění. Editor VBA vám neumožní zkompileovat a spustit program, který bude obsahovat jakékoli syntaktické chyby, odkazy na knihovny, jež nejsou zahrnuty v referencích projektu, ani který bude zahrnovat nedeklarované proměnné, pokud máte v záhlaví modulu `Option Explicit` (doporučujeme).

Chyby algoritmické jsou chyby, kdy program sice funguje technicky správně, avšak vykonává nesprávnou činnost. Vzpomeňte si na příklad zpětného kroku smyčky, mazající vybrané řádky. Pokud bychom

použili kladný krok, algoritmus by některé řádky vůbec nekontroloval. Na tomto místě opět zdůrazňujeme, jak je důležité psát dobře strukturovaný program bez umělých čítačů, odskoků ven ze struktur, manipulace s čítačem cyklu a podobně. Chyba v algoritmu se v takovém programu hledá nesrovnatelně snadněji, než když musíte luštit, jak je běh programu vlastně řízen a kterou hodnotu má ve kterém momentu ten který čítač.

Chyby za běhu programu jsou chyby, kdy program narazí na příkaz, který neumí z nejrůznějších důvodů vykonat anebo mu v daném kontextu nerozumí.

Podívejte se na tento krátký kód, který má za úkol změnit barvu pozadí buněk A1:A10 na listu „List1“ na žlutou:

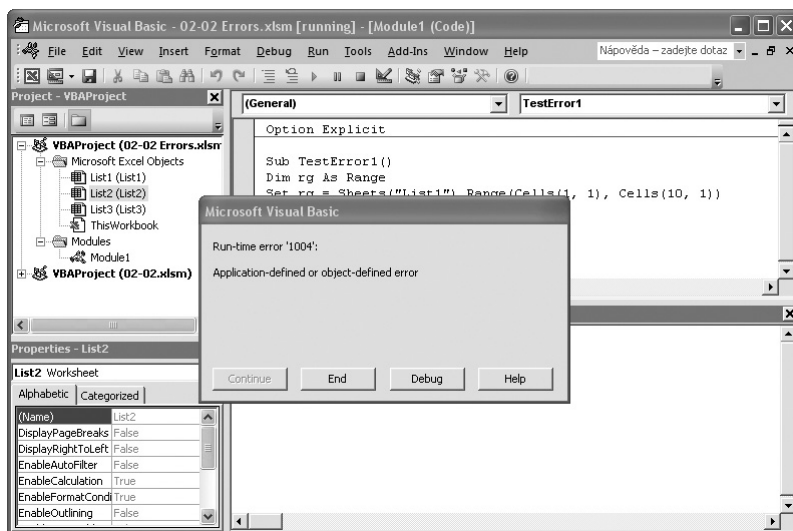


01-05 Chyby.xlsm
modul MErrorHandling

```
Sub TestError1()  
Dim rg As Range  
Set rg = Sheets("List1").Range(Cells(1, 1), Cells(10, 1))  
rg.Interior.Color = vbYellow  
End Sub
```

Ujistěte se, že je List1 aktivní, a kód spusťte (Karta „Vývojář“, skupina „Kód“, položka „Makra“). Vše proběhne, jak má, a buňky změní barvu.

Nyní vyberte List2 (nebo jakýkoli jiný list kromě listu List1) a kód spusťte znovu:



Obrázek 1.11: Chyba za běhu programu jako důsledek nedůsledného kódu

Kód skončí příliš nepomáhajícím hlášením a navíc uživatele přepne do okna Editoru VBA. Pokud uživatel stiskne tlačítko Debug, řádek, který nebylo možno vykonat, se žlutě podbarví a kód čeká na jeho úpravu. Poté je možno běh programu obnovit (F5), nebo programem krokovat (F8). Pokud byla úprava neúčinná, řádek zůstane podbarven a zobrazí se nové okno s chybovým hlášením. V tomto režimu lze kurzorem přejíždět po proměnných, aby bylo možno vidět jejich momentální hodnoty,

a rovněž lze ověřovat výrazy a hodnoty proměnných v okně Immediate, což je velmi účinná technika pro ladění!

V našem příkladu je náprava jednoduchá. Problém kódu je v tom, že spoléhá na to, že List1 bude aktivní. Vlastnosti Cells nejsou nijak kvalifikovány rodičovským objektem, a tudíž se odkazují na svůj rodičovský objekt ze třídy Globals, konkrétně Activesheet. Samozřejmě, že kód neumí určit oblast na jednom listu (definovaný List1) pomocí buněk z listu druhého (aktivní List2).

Kód musíme napsat takto:

```
Dim rg As Range
With Sheets("List1")
    Set rg = .Range(.Cells(1, 1), .Cells(10, 1))
End With
rg.Interior.Color = vbYellow
```

V tomto případě chybu odstranila oprava programu, ale tak tomu zdaleka vždy být nemusí.

Následující kód se pokusí otevřít sešit se zadaným názvem, který by se měl nacházet ve stejné složce, jako sešit, z něhož proceduru spouštíme:

```
Sub TestError2()
Const FileName = "Nothing.xlsx"
Workbooks.Open ThisWorkbook.Path & "\ " & FileName
End Sub
```

Tento kód skončí chybou, protože ve vašem adresáři s vysokou pravděpodobností takový sešit neexistuje. Řešením je nejprve zkontrolovat, jestli daný sešit existuje:

```
Sub NoError2()
Const FileName = "Nothing.xlsx"
Dim FullName As String
FullName = ThisWorkbook.Path & "\ " & FileName
If Len(Dir$(FullName)) > 0 Then
    Workbooks.Open ThisWorkbook.Path & "\ " & FileName
Else
    MsgBox "Soubor " & FullName & " nebyl nalezen."
End If
End Sub
```

I v tomto případě pomohla úprava programu.



Tip

Na tomto místě si zapamatujte důležité pravidlo: Nikdy nepředávejte do metod a funkcí neprovořené parametry. Pokud předáváte jméno objektu, nejdříve ověřte, že objekt existuje. Pokud tomu tak není, uživatele upozorněte pomocí zprávy a dejte mu na výběr, jak pokračovat, případně kód ukončete.

Zatím jsme úmyslně uváděli příklady, kdy bylo předvídatelným chybám možno zabránit pečlivým programováním. Nyní předpokládejme, že máte obsáhlý sešit s mnoha listy, z nichž je pro uživatele důležitý list „Report“. Kód má za úkol tento list zkopírovat do nového sešitu a tento sešit pod jménem „Report_yyyyymmdd.xlsx“ (datum ve formátu rok, měsíc, den) uložit na uživatelův USB disk,

který se pravidelně připojuje pod písmenem H (neboť je to první volné písmeno na vašem počítači – jedná se samozřejmě o příklad):

```
Sub TestError3()  
Const SHEET_NAME As String = "Report"  
Const SAVE_DIR As String = "H:\"  
Dim FullName As String  
Application.ScreenUpdating = False  
FullName = SAVE_DIR & "Report_" & Format$(Date, "yyyymmdd") & ".xlsx"  
ThisWorkbook.Sheets(SHEET_NAME).Copy  
With ActiveWorkbook  
    With .Sheets(SHEET_NAME).UsedRange  
        .Value = .Value  
    End With  
    Application.DisplayAlerts = False  
        .SaveAs FullName  
        .Close False  
    Application.DisplayAlerts = True  
End With  
End Sub
```

Pokud je USB disk připojen pod písmenem H, vše proběhne jak má. Pokud ovšem disk připojen není (nebo se připojuje pod jiným písmenem), dostanete po spuštění přibližně takovouto zprávu o chybě za běhu programu vinou nepřipojeného USB disku.

Nejenže kód sešit nemůže uložit, ale navíc nově vytvořený sešit zůstane otevřený v Excelu a musíte ho ručně zavřít bez uložení.

Potřebujeme tedy kód ošetřit tak, aby dal uživateli možnost opomenutí napravit a připojit USB disk. Pokud se uživatel rozhodne sešit neukládat (například proto, že žádný volný USB disk po ruce nemá), měl by kód automaticky vytvořený sešit zavřít bez uložení a tím aplikaci vrátit do původního stavu.

Řešením je tento kód:



*01-05 Chyby.xlsm
modul MErrorHandler*

```
Sub NoError3()  
Const SHEET_NAME As String = "Report"  
Const SAVE_DISK As String = "H:\"  
Dim NewWB As Workbook  
Dim FullName As String  
Dim CurrentDir As String  
Dim msg As String  
  
On Error GoTo Err_Handler  
  
    CurrentDir = CurDir$  
    ChDrive SAVE_DISK  
    ChDrive CurrentDir  
  
    Application.ScreenUpdating = False
```



```

FullName = SAVE_DISK & "Report_" & Format$(Date, "yyyymmdd") & ".xlsx"
ThisWorkbook.Sheets(SHEET_NAME).Copy
Set NewWB = ActiveWorkbook
With NewWB
    With .Sheets(SHEET_NAME).UsedRange
        .Value = .Value
    End With
    Application.DisplayAlerts = False
    .SaveAs FullName
    Application.DisplayAlerts = True
End With

```

Cleanup:

```

'v této části si už nemůžeme dovolit zachycování chyb
On Error Resume Next
NewWB.Close False

```

Exit Sub

Err_Handler:

```

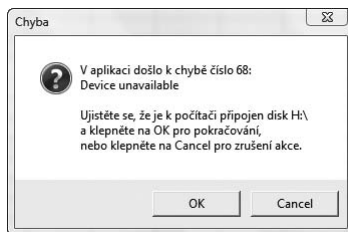
msg = "V aplikaci došlo k chybě číslo " & Err.Number & ":" & vbCrLf
msg = msg & Err.Description & String(2, vbCrLf)
msg = msg & "Ujistěte se, že je k počítači připojen disk " & _
    SAVE_DISK & vbCrLf
msg = msg & "a klepněte na OK pro pokračování, " & vbCrLf
msg = msg & "nebo klepněte na Cancel pro zrušení akce."
If MsgBox(msg, vbQuestion + vbOKCancel, "Chyba") = vbOK Then
    Resume
Else
    Resume Cleanup
End If

```

End Sub

Tento způsob kódu dá uživateli šanci napravit opomenutí a vyzve ho k nápravě. Program se nejprve pokusí změnit aktivní disk příkazem `ChDrive`. Pokud disk pod tímto písmenem neexistuje, vzniklá chyba přinutí tok programu skočit na návěští `Err_Handler`, což je běžné označení části kódu, která ošetřuje run-time chyby. Pokud uživatel připojí disk a klepne na tlačítko OK, kód vykoná znovu příkaz `ChDrive`, a pokud je úspěšný, kód pokračuje dále. Pokud uživatel klikne na tlačítko Cancel, kód skočí na návěští `Cleanup` (anglický výraz pro „úklid“). Tato závěrečná část kódu by měla aplikaci uvést do původního stavu, a to ať již k chybě došlo, či ne. V této části už si nemůžeme dovolit zachytávat chyby, protože by došlo k nekonečné smyčce, kdy vzniklá chyba v závěrečné části kódu by to programu vrátila přes sekci `Err_Handler` znovu na návěští `Cleanup`. Proto je zachytávání chyb potlačeno příkazem `On Error Resume Next`.

Kód předpokládá, že může nastat jediná chyba, a to když daný disk není připojen (nebo není disketa v diskové jednotce). Když se ovšem na kód pozorně podíváte, zjistíte, že selže, pokud list „Report“ neexistuje. Jak jsme si již řekli, do metod a funkcí by



Obrázek 1.12: Uživatelsky přizpůsobené hlášení o chybě

se nikdy neměly předávat neproověřené parametry, ale v tomto případě si ukážeme, jak vypadá procedura zahrnující ošetření dvou konkrétních chyb a jedné obecné:



01-05 Chyby.xlsm
modul MErrorHandler

```
Sub NoError3b()
Const SHEET_NAME As String = "Report"
Const SAVE_DISK As String = "H:\\"
Dim NewWB As Workbook
Dim FullName As String
Dim CurrentDir As String
Dim msg As String

On Error GoTo Err_Handler

    CurrentDir = CurDir$
    ChDrive SAVE_DISK
    ChDrive CurrentDir

    Application.ScreenUpdating = False

    FullName = SAVE_DISK & "Report_" & Format$(Date, "yyyymmdd") & ".xlsx"
    ThisWorkbook.Sheets(SHEET_NAME).Copy
    Set NewWB = ActiveWorkbook
    With NewWB
        With .Sheets(SHEET_NAME).UsedRange
            .Value = .Value
        End With
        Application.DisplayAlerts = False
        .SaveAs FullName
        Application.DisplayAlerts = True
    End With

Cleanup:
    'v této části si už nemůžeme dovolit zachycování chyb
    On Error Resume Next
    NewWB.Close False

Exit Sub

Err_Handler:
    Select Case Err.Number
    Case 9
        msg = "V aplikaci došlo k chybě číslo " & Err.Number & ":" & vbCrLf
        msg = msg & Err.Description & String(2, vbCrLf)
        msg = msg & "List s názvem " & SHEET_NAME & " nebyl nalezen."
        MsgBox msg, vbOKOnly + vbCritical, "Chyba"
        Resume Cleanup
    Case 68
        msg = "V aplikaci došlo k chybě číslo " & Err.Number & ":" & vbCrLf
        msg = msg & Err.Description & String(2, vbCrLf)
```

```

msg = msg & "Ujistěte se, že je k počítači připojen disk " & _
    SAVE_DISK & vbLf
msg = msg & "a klepněte na OK pro pokračování, " & vbLf
msg = msg & "nebo klepněte na Cancel pro zrušení akce."
If MsgBox(msg, vbQuestion + vbOKCancel, "Chyba") = vbOK Then
    Resume
Else
    Resume CleanUp
End If
Case Else
msg = "V aplikaci došlo k chybě číslo " & Err.Number & ":" & vbLf
msg = msg & Err.Description & String(2, vbLf)
msg = msg & "Neočekávaná chyba. Není možné pokračovat."
MsgBox msg, vbOKOnly + vbCritical, "Chyba"
Resume CleanUp
End Select
End Sub

```

V tomto mnohem robustnějším kódu je při vzniku chyby za běhu programu předáno ošetření chyby bloku `Select Case`. Pokud chybí list s daným jménem, vyvolá se chyba číslo 9, „Subscript out of range.“ Tento nic moc nevypovídající popis s nejvyšší pravděpodobností znamená, že se program pokusil jménem zavolat neexistující prvek kolekce (naš případ), nebo zavolal prvek pole indexem, který přesahuje hranice pole či neodpovídá počtu dimenzí. Je to jedna z nejčastějších run-time chyb.

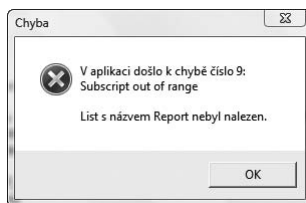
Po zobrazení této zprávy kód pokračuje skokem na návěští `CleanUp`, vykoná „úklidové práce“ a poté se ukončí. Všimněte si, že zpráva je zobrazena s ikonou `vbCritical`, která naznačuje předčasné ukončení programu.

Pokud USB disk buď není připojen, nebo se připojuje pod jiným písmenem, nebo není v disketové jednotce disketa, vyvolá se chyba číslo 68, *Device unavailable*, a zobrazí se dialog s ikonou `vbQuestion`, umožňující volbu mezi pokračováním a ukončením skokem na návěští `CleanUp`.

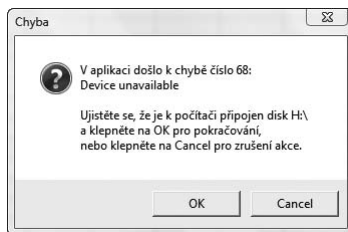
Pokud číslo chyby neodpovídá ani jednomu případu v bloku `Select Case`, vykoná se jeho část `Case Else` a zobrazí se obecné hlášení o neočekávané chybě s ikonou `vbCritical` (a následný odskok na sekci `CleanUp`).

V našem příkladu by neočekávaná chyba měla nastat pouze ve velmi výjimečných případech, například pokud je disk zaplněn nebo nějakým způsobem chráněn proti zápisu.

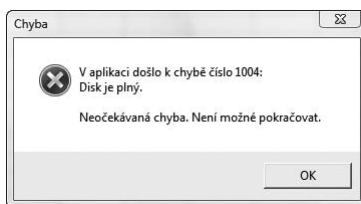
Obvyklá obecná procedura ošetřující chyby za běhu programu tedy vypadá takto:



Obrázek 1.13: Hlášení o chybějícím listu



Obrázek 1.14: Hlášení o nepřipojeném disku



Obrázek 1.15: Hlášení o zaplněném disku

```

Sub ObecnaProcedura()

'V případě chyby bude proveden odskok na návěští Err_Handler
On Error GoTo Err_Handler

' V tomto místě bude vlastní kód procedury

Cleanup:
' Tato část bude v každém případě vykonána

' V této části si už nemůžeme dovolit zachycování chyb
  On Error Resume Next

' Slouží k úklidu a uvedení aplikace do původního stavu, například:
  Application.EnableEvents = True      ' zapnutí událostí
  rst.Close: Set rs = Nothing           ' uzavření sady záznamů
  WorkbookObject.Close False          ' zavření sešitu bez uložení
  objWord.Quit: Set objWord = Nothing  ' uzavření cizí aplikace
Exit Sub

Err_Handler:
  Select Case Err.Number
  Case Cisl01
    MsgBox "Zpráva1", vbOKOnly + vbCritical, "Chyba"
    Resume Cleanup
  Case Cisl02
    MsgBox "Zpráva2", vbOKOnly + vbCritical, "Chyba"
    Resume Cleanup
  ' zde pokračují předvídané chyby
  Case Else
    'nepředvídaná chyba
    MsgBox "Zpráva3", vbOKOnly + vbCritical, "Chyba"
    Resume Cleanup
  End Select
End Sub

```

Všimněte si důležité vlastnosti této obecné ukázky. Ať již k chybě dojde nebo ne, sekce `Cleanup` je vždy vykonána a kód i při nezdárném průběhu po sobě uklidí. Navíc tuto část píšete jen jednou a nemusíte úpravy provádět na více místech v blocích `Case`. Do těchto bloků umísťujete pouze příkazy, které napravují změny související s konkrétní chybou. V našem předchozím příkladu bylo možno umístit příkaz

```
NewWB.Close False
```

do sekce `Case 68` bloku `Select Case`.

Pomocí příkazů `On Error Goto Návěští` a `Resume Návěští` v kombinaci s blokem `Select Case` můžeme tedy ošetřit většinu běžných případů. Každá procedura by měla mít jediný bod výstupu, což je zpravidla několik příkazů před řádkem `Exit Sub`.

Po prostudování uvedeného příkladu je vám již určité jasné, jak se run-time chyby (chyby vzniklé za běhu programu) ošetřují. Podívejme se tedy podrobněji na jednotlivé součásti.

Objekt Err

Když dojde za běhu programu k chybě, objekt `Err` je naplněn informacemi o vzniklé chybě. Prakticky každá technika ošetření chyb tento objekt využívá, a proto je užitečné znát základní metody a vlastnosti tohoto objektu.

`Err.Clear` anuluje všechny vlastnosti objektu `Err` a tím ruší vzniklou chybu.

`Err.Description` vrací krátký popis chyby.

`Err.Number` vrací číslo spojené s naposledy vzniklou chybou. Tato vlastnost je užitečná při rozhodování, jak se vzniklou chybou naložit.

`Err.Raise` je metoda, která umožňuje vyvolávání vlastních chyb. Toto je užitečné např. v modulech tříd, nebo při odlaďování kódu zachytávajícího chyby.

`Err.source` vrací informaci o zdroji chyby. V běžném programu tato vlastnost není příliš užitečná, neboť vrací prázdný řetězec nebo obecné jméno VB projektu, avšak při vyvolávání vlastních chyb bychom do této vlastnosti měli přiřadit jméno procedury nebo třídy, která tuto chybu vyvolala.

Rozsah platnosti ovladače chyb

Část kódu, která se vykonává po vzniku chyby, je platná pro proceduru, v níž je umístěna, jakož i pro všechny podprocedury touto procedurou volané, *kteří nemají vlastní kód ovládající chyby*. Pokud hlavní procedura obsahuje ovladač chyb, avšak podřízené procedury ne, bude tento ovladač aktivován i z těchto volaných procedur. Zde je příklad takové kaskády:

```
Sub CascadedError()
    On Error GoTo Err_Handler
    Call SubProcedure1
ExitHere:
Exit Sub
Err_Handler:
    MsgBox Err.Description, vbCritical, "Error " & Err.Number
    Resume ExitHere
End Sub
```

```
Sub SubProcedure1()
    'vyvoláme chybu dělení nulou
    Err.Raise 11
End Sub
```

Chyba vzniklá ve volané proceduře provede odskok směrem zpět na první ovladač chyb, který najde. V tomto případě se jedná o ovladač v proceduře o jednu úroveň výše. Ovladač vypíše varovné hlášení, pokračuje návěstí `ExitHere` a program se ukončí. Sekce `Err_Handler` tedy obsluhuje chyby vzniklé v jakékoli proceduře, ať již volané nebo volající, pokud nemá vlastní kód ovládání chyb (neobsahuje výraz `On Error...`).

V některých jednoduchých případech toto může být rozumná a účinná technika, ale obvykle ji pro profesionální aplikace doporučit nelze. Za prvé, ztrácíte informaci, kde chyba vlastně nastala. Za druhé, při vzniku chyby musí procedura po sobě většinou uklidit a tento úklid může být velmi specifický. Může se jednat o zavření objektů, odstranění objektů z paměti, ukončení spojení s data-

bází, inicializace globálních proměnných, obnovení vzhledu aplikace apod. Pokud procedura neobsahuje vlastní ošetření chyb, tento úklidový kód nebude vykonán a to může vést až ke zhroutil počítače (lépe řečeno, operačního systému). Chyba se propaguje směrem zpět do prvního ovladače chyb, který v kaskádě volajících procedur najde, a neexistuje způsob, jak tok kódu vrátit zpátky do procedury, která chybu vyvolala. Proto je většinou lepší psát procedury, které obsahují svůj vlastní kód ovládání chyb.

On Error Goto <návěští>

Tento příkaz inicializuje ovladač chyb, který se nachází za řádkem <návěští>. Každá vzniklá run-time chyba provede odskok na toto návěští a je na programátorovi, jak se mají chyby ošetřit, jaké hlášení se má zobrazit a kde má kód poté pokračovat.

On Error Resume Next

V případě tohoto příkazu doslova platí přísloví o dobrém sluhovi, ale zlém pánu, protože je velmi užitečný i velmi nebezpečný zároveň. V podstatě říká programu VBA, aby ignoroval jakoukoli chybu, která nastane, a pokračoval dalším řádkem. Na tomto místě je užitečné vědět, že pokud program narazí na více příkazů `On Error Goto ...`, zůstává v platnosti poslední.

Příkaz

```
On Error Resume Next
```

nikdy nepoužívejte pro potlačení neočekávaných chyb, které mohou ve vašem programu nastat. V žádném případě to není náhražka za špatně napsaný kód! Máte-li program, který nepoběží, pokud na jeho začátek neumístíte tento výraz, téměř určitě není tento kód dobře napsán.

Ukážeme si však případy, kdy je tento výraz velmi užitečný. Předtím se ale ještě podívejte na výraz

```
On Error Goto 0
```

Tento výraz, ať je jeho syntaxe jakkoli zvláštní (neboť provádí odskok na neexistující návěští, na které by kód neodskočil, ani kdyby existovalo), neprovádí nic jiného, než že anulují předchozí výrazy `On Error` a kontrolu nad chybami vrací zpět vlastnímu prostředí VBA. Tyto výrazy ruší ovšem jen v případě, že se nacházejí ve stejné proceduře jako výraz `On Error Goto 0`. Pokud je uveden ve volané proceduře a ovladač chyb se nachází v proceduře volající, příkaz `On Error` anulován nebude a chyba vrátí tok kódu do sekce ovladače chyb.

Jak jsme již řekli, příkaz `On Error Resume Next` je velmi nebezpečný, a proto jeho platnost doporučujeme omezit na nejmenší možný rozsah příkazem `On Error Goto 0`:

```
Sub SilentError()
```

```
On Error Resume Next
```

```
    'jakékoli chyby v této sekci budou ignorovány
```

```
    Err.Raise 11
```

```
On Error GoTo 0
```

```
    'vyčistíme objekt Err od možné informace o předešlé chybě
```

```
    Err.Clear
```

```
    'tato chyba vyvolá standardní hlášení o výjimce
```

```
Err.Raise 13
```

```
End Sub
```

Sekce, ve které se chyby ignorují, může být samozřejmě ukončena i výrazem, předávajícím ovládací chybu zpět ovladači:



01-05 Chyby.xlsm
modul MErrorHandler

```
Sub OpenIfNotOpened()
Const WB_NAME As String = "Nothing.xlsm"
Dim wb As Workbook
On Error GoTo Err_Handler

'zde bude nějaký kód s chybami zachycovanými ovladačem

'otestujeme, jestli je sešit se zadaným názvem otevřen
On Error Resume Next
    Set wb = Workbooks(WB_NAME)
On Error GoTo Err_Handler

'pokud je objektová proměnná prázdná, sešit otevřeme
'pokud sešit neexistuje, chyba bude zachycena ovladačem
If wb Is Nothing Then
    Set wb = Workbooks.Open(ThisWorkbook.Path & "\" & WB_NAME)
End If

'zde bude další část kódu

ExitHere:

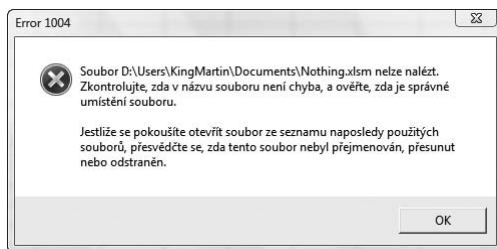
Exit Sub

Err_Handler:
    MsgBox Err.Description, vbCritical, "Error " & Err.Number
    Resume ExitHere
End Sub
```

V tomto výpisu je řízení chyb nejprve předáno ovladači. Poté následuje sekce ohraničená příkazy

```
On Error Resume Next
    Set wb = Workbooks(WB_NAME)
On Error GoTo Err_Handler
```

Pokud sešit není otevřený, bude vyvolána chyba. Proto ji potlačíme a po pokusu o přiřazení objektu sešitu do objektové proměnné opět ovládací chybu zapneme. Pokud selže i metoda `Open`, kód ohlásí chybu definovaným způsobem, provede se odskok na návěští `ExitHere` a program skončí.



Obrázek 1.16: Hlášení o neexistujícím souboru

Zapnutí ignorování chyb může být velmi užitečným pomocníkem. Použití je běžné ve dvou případech. Jedním je testování existence objektu, podobně jako v předěšlém případě. Jiným běžným případem je test na existenci listu v sešitu, který jsme uvedli již dříve:

```
'// Funkce vrátí True, pokud list zadaného jména v aktivním sešitu existuje
```

```
Public Function WorksheetExists(SheetName As String) As Boolean
```

```
On Error Resume Next
```

```
WorksheetExists = Len(Worksheets(SheetName).Name) > 0
```

```
Err.Clear
```

```
On Error Goto 0
```

```
End Function
```

Druhým použitím je ignorování chyb v závěrečné části kódu (Cleanup), který má za úkol pokusit se aplikaci uvést do původního stavu, uzavřít spojení s databází, zničit objekty v paměti počítače a podobně. Zachycení chyb v této části již příliš užítka nepřinese.

V profesionálních aplikacích se většinou používá centrální procedura ovládání chyb, do které jsou všechny chyby v aplikaci předány, včetně informace o proceduře, která chybu vyvolala, jejím umístění, otevřených sešitech, typu chyby a podobně. Tato procedura často využívá externí textový soubor, do kterého chyby zapisuje, aby byl zachován přehled chování aplikace. Centrální procedura pro zachycování a registraci chyb se většinou používá pro neočekávané chyby, a ne pro chyby očekávané a specifické pro danou proceduru.

Výraz Resume

Tento výraz je rovněž velmi nebezpečný, protože ukončuje běh ovladače chyb a tok programu vrací na řádek, který chybu způsobil. Předpokládá se, že buď program sám (např. otevřením zavřeného sešitu) nebo uživatel (např. připojením externího disku) chybu odstraní a příkaz bude opětovně vykonán bez chyby. Pokud chyba trvá, bude tok programu vrácen do ovladače a lehce může dojít k nekonečné smyčce, ze které je jediným únikem stisknutí kláves Ctrl+Break, nebo v horším případě Ctrl+Alt+Del a násilné ukončení aplikace Microsoft Excel bez uložení dat.

Přes toto varování může být výraz `Resume` velmi užitečný. Pokud se například pokoušíte připojit ke vzdálené databázi, je obvyklé, že se spojení na první pokus nepodaří. Pokud je spojení neúspěšné, tok programu je předán ovladači chyb a tento ovladač může po zjištění čísla chyby tuto chybu individuálně zpracovat. Například může zvýšit hodnotu specifikovaného čítače, a pokud hodnota nepřesáhla maximální definovaný počet pokusů, může pokus o připojení opakovat. Takový kód může vypadat například takto:



01-05 Chyby.xlsm
modul MErrorHandler

```
Sub ZkusPripojeni()
```

```
Const DB_NAME = "Northwind 2007.accdb"
```

```
Dim cn As ADODB.Connection
```

```
Dim FullName As String
```

```
Dim lPokus As Long
```



```

Dim strConn As String

On Error GoTo Err_Handler
FullName = "\\ServerName\Path\" & DB_NAME

If Val(Application.Version) < 12 Then
    strConn = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
        "Data Source=" & FullName & ";"
Else
    strConn = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & FullName & ";"
End If

'Zkusíme otevřít připojení
Application.StatusBar = "Pokus o připojení k databázi..."
Set cn = CreateObject("ADODB.Connection")
cn.Open strConn
'pokud se kód dostal sem, spojení se zdařilo

'pokračuje kód pracující s objekty databáze

Cleanup:
On Error Resume Next
cn.Close: Set cn = Nothing
Application.StatusBar = False
Exit Sub

Err_Handler:

'Pokusíme se o připojení třikrát
If lPokus < 3 And cn.Errors.Count > 0 Then
    If cn.Errors(0).NativeError = 17 Then
        Application.StatusBar = "Opakovaný pokus o připojení..."
        lPokus = lPokus + 1
        Resume
    End If
End If

'jinak chybu budeme považovat za kritickou
MsgBox Err.Description, vbCritical, "Error " & Err.Number
Resume Cleanup

End Sub

```

Jiný příklad použití výrazu Resume, kdy uživatel dostane šanci připojit opomenutý externí disk, jsme již uvedli.

Výraz Resume Next

Výraz Resume Next provede první vykonatelný řádek následující za řádkem, který způsobil chybu. Oproti očekávání ale ovladač chyb umístěný v nadřizené proceduře nevrátí tok kódu do procedury podřizené (volané), nýbrž pokračuje dalším řádkem následujícím za řádkem, který chybnou proceduru volal. Toto můžete s výhodou využít, pokud má kritická chyba ukončit vykonávání podprocedury, avšak kód hlavní procedury má pokračovat dále:



01-05 Chyby.xlsm
modul MErrorHandler

```
Sub ProceedAfterError()
    On Error GoTo Err_Handler
    Call SubProcedureA
    Call SubProcedureB
Exit Sub
Err_Handler:
    MsgBox Err.Description, vbCritical, "Error " & Err.Number
    Resume Next
End Sub
```

```
Sub SubProcedureA()

    'vyvoláme chybu dělení nulou
    Err.Raise 11

    'Toto hlášení zobrazeno nebude
    MsgBox "Procedura A vykonána s chybou."

End Sub
```

```
Sub SubProcedureB()
    'toto hlášení se zobrazí
    MsgBox "Procedura B vykonána."
End Sub
```

Hlavní procedura nejdříve zavolá podproceduru A, která vygeneruje chybu. Tato chyba předá tok kódu ovladači chyb (Err_Handler) v nadřizené proceduře (protože sama žádný ovladač nemá). Výraz Resume Next nevrátí tok programu zpět do podprocedury A, nýbrž pokračuje dalším řádkem ve svém vlastním těle, tedy Call SubProcedureB, a podprocedura B vrátí hlášení o svém úspěšném dokončení. Řádky v podproceduře A, následující za řádkem, který vyvolal chybu, vykonány nebudou!

Tuto techniku lze využít například tehdy, pokud generujeme zprávu z obnovených dat. Pokud se nezdaří obnovení dat z databáze na jednom listu, stále můžeme obnovit všechna ostatní data a zprávu s příslušným upozorněním odeslat.

Resume <návěští>

Tento výraz přinutí program pokračovat na dalším řádku za uvedeným návěštím. Toto návěští musí být definováno v té proceduře, která obsahuje výraz Resume <návěští>. Typicky se využívá při odskoku na závěrečnou, „úklidovou“ část programu, jak jsme si již několikrát uvedli. Příklady takových výrazů jsou:

Resume ExitHere
 Resume CleanUp
 Resume TidyUp

Každý váš program by měl mít jediný bod výstupu. Nepoužívejte výraz Exit Sub v dané proceduře více než jednou. Místo toho opatřete výstupní bod návěštím a mezi návěští a výraz Exit Sub případně umístíte „úklidový“ kód. Takových příkladů jsme uvedli několik.

Shrnutí

1. Každá netriviální procedura by měla obsahovat svůj vlastní ovladač chyb.
2. Výraz `On Error Resume Next` používejte s rozmyslem. Nenahrazujte jím nedbale napsaný kód.
3. V každé proceduře definujte pouze jeden bod výstupu, opatřený návěštím.
4. Opuštění procedury by měl předcházet úklidový kód, případně začínající výrazem `On Error Resume Next`.
5. Buďte opatrní při použití výrazu `Resume`. Vždy ošetřete případ, kdy chybu není možné odstranit, aby nedošlo k nekonečné smyčce.
6. Výraz `Resume Next` používejte rovněž po pečlivém zvážení. Způsobí pokračování v kódu, i když v něm nastala chyba.

Otázky

Otázka 1.9:

Napište kód, který spustí proceduru `Procedura1`, pokud je v buňce A1 liché číslo, a proceduru `Procedura2`, pokud je číslo sudé nebo nula. Je-li buňka prázdná nebo údaj nenumerní, nevykoná se nic.

Otázka 1.10:

Napište kód, který skryje v aktivním sešitu všechny listy, jejichž jméno začíná podtržítkem.

Otázka 1.11:

Na co musíte dát pozor, když používáte metodu `Find` pro změnu hodnoty ve vybraných buňkách v dané oblasti?

Otázka 1.12:

Pokud procedura, která obsahuje ovládání chyb, volá další procedury, co se stane, když v těchto procedurách nastane chyba? Na čem záleží?

Text ve VBA

Zatímco čísla jsou při psaní programů ve VBA pro Excel většinou uložena v buňkách a zpracovávána vzorci v buňkách, a tudíž se o jejich formu nemusíte příliš starat, zpracovávání textu se nevyhnete. Typickými úlohami jsou nahrazování části textu, rozdělení textu na části, spojení částí do jednoho řetězce, sestavení cesty k souboru, vytvoření nového jména souboru, převod písmen a podobně.

Importované soubory často přicházejí ve velmi nepřívětivé formě, kdy je data třeba nejprve podstatným způsobem upravit, než je můžeme analyzovat nebo případně importovat do databáze. Pro všechny tyto úkoly má VBA řadu zajímavých a výkonných funkcí.

Jak VBA ukládá text

Textový řetězec (angl. *String*) ve VBA není nic jiného, než sekvence bytů. Aby jazyk VBA mohl s řetězcí pracovat rychleji, nese každý řetězec s sebou rovněž informaci o své délce. Na rozdíl od některých jiných jazyků spravuje řetězcové buffery (vyhrazená místa v paměti počítače) VBA automaticky, a vy se tudíž nemusíte starat, jak VBA řetězec vytvoří, jak jej v paměti najde a jestli se místo v paměti uvolní při zrušení řetězcové proměnné.

Veškerý kód z této části lekce najdete v souboru 01-06 Text.xlsm, v modulu MStrings.

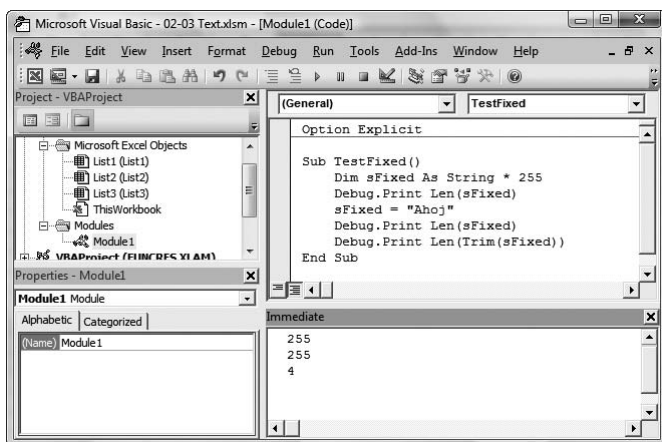


01-06 Text.xlsm
modul MStrings

Řetězcové proměnné nabízí VBA ve dvou druzích: řetězce o *pevné délce* a řetězce *dynamické*. Řetězce o pevné délce jsou ve VBA méně běžné, ale v některých případech je jejich použití nutné, např. při předávání parametrů do funkcí API, které očekávají řetězec o pevné délce (většinou 255 bytů). Jejich zpracování je o něco rychlejší, ale na druhou stranu je pro běžné použití výstup nutné očistit o přebytečné mezery, čímž se výkon vyrovnává.

Řetězcová proměnná o pevné délce se deklaruje takto:

```
Dim strFixed As String * 255
```



Obrázek 1.17: Testování délky řetězcové proměnné o pevném počtu znaků

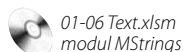
Jak je vidět, i na anglickém systému má český čtenář štěstí, protože celá česká abeceda včetně speciálních typů „ň“ a „ů“ je v základní stránce obsažena.

ANSI kód vrací funkce `Asc()`, jež může mít dvě další varianty, `AscB()`, která se používá pro bytová (bajtová) data v řetězcích a vrací první byte, a `AscW()`, která vrací kód podle standardu Unicode.

S rozmachem internetu a aplikací, které musely být funkční v mnohajazyčných prostředích, bylo stále obtížnější zobrazovat potřebné znaky pro nás exotických jazyků, jako je čínština, arabština, ale i mnohem bližší ruština. Proto byl vytvořen standard Unicode, který pro každý znak využívá dva byty, a tedy může zobrazit $256 * 256 = 65536$ znaků. Tím umožňuje zahrnout prakticky všechny známé znaky, používané lidstvem, do jednoho souboru, adresovaného jedním číslem (či přesněji řečeno, dvěma čísly 0 až 255).

Bez ohledu na to, který systém Windows na svém PC (ne na Macintoshi!) používáte, text ve VBA je vždy ukládán ve formátu Unicode, a tudíž každý znak zabírá dva byty. V podstatě vás nemusí zajímat, jak VBA znaky ukládá, pokud nechcete prozkoumat každý byte řetězce. Pro tento účel VBA nabízí velmi zajímavou vlastnost. Pokud nadeklarujete dynamické pole typu `Byte`, VBA vám umožní do tohoto pole přiřadit řetězec, který je automaticky konvertován na pole s jednotlivými byty. A co více, bytové pole je možné (např. po úpravě) jediným příkazem přiřadit zpět do textové proměnné!

V následujícím příkladu prozkoumáme jednotlivé byty řetězce „Buňka“ a nakonec přiřadíme bytové pole zpět do řetězcové proměnné a vypíšeme výsledek:



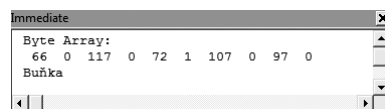
01-06 Text.xlsm
modul MStrings

```
Sub TestUnicode()
    Dim sText As String
    Dim aByt() As Byte
    Dim i As Long
    sText = "Buňka"
    aByt() = sText
    Debug.Print "Byte Array:"
    For i = LBound(aByt) To UBound(aByt)
        Debug.Print aByt(i);
    Next
    sText = aByt()
    Debug.Print
    Debug.Print sText
End Sub
```

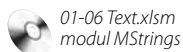
Výsledkem je tento výpis:

První dva znaky („B“ a „u“) daného řetězce jsou standardními znaky latinské abecedy, a proto je druhý byte roven nule. Třetí znak („ň“) je znakem nestandardním, a proto se jeho druhý byte od nuly liší. Jeho Unicode hodnota je $72 + 1 * 256 = 328$, jak vzápětí uvidíme. Kód správně vypsal slovo „Buňka“, které vzniklo přiřazením bytového pole přímo zpět do textové proměnné.

Následující kód vrátí ANSI kódy jednotlivých znaků podle aktivní kódové stránky a z těchto kódů řetězec znovu sestaví pomocí funkce `Chr()`, která vrací znak na základě jeho kódu:



Obrázek 1.18: Výpis hodnoty jednotlivých bytů řetězce Unicode s hodnotou „Buňka“



```

Sub TestANSI()
    Dim sText As String
    Dim sZnak As String
    Dim sNewText As String
    Dim i As Long
    sText = "Buňka"
    Debug.Print ; "ANSI:"
    For i = 1 To Len(sText)
        sZnak = Mid$(sText, i, 1)
        Debug.Print Asc(sZnak);
        sNewText = sNewText & Chr(Asc(sZnak))
    Next
    Debug.Print
    Debug.Print sNewText
End Sub

```

Výpis bude vypadat takto:

Náš speciální znak „ň“ má kód 242 a leží blízko konce aktuální kódové stránky. Podívejte se ještě jednou výše na výpis všech znaků. Najdete ho na posledním řádku téměř u konce.



Obrázek 1.19: Výpis hodnoty jednotlivých hodnot ANSI řetězce „Buňka“

Nakonec ještě pro úplnost ukážeme výpisy funkcí AscW() a AscB():

```

Sub TestANSI_W()
    Dim sText As String
    Dim i As Long
    sText = "Buňka"
    Debug.Print ; "ANSI_W:"
    For i = 1 To Len(sText)
        Debug.Print AscW(Mid$(sText, i, 1));
    Next
    Debug.Print
End Sub

```

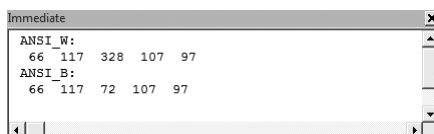
```

Sub TestANSI_B()
    Dim sText As String
    Dim i As Long
    sText = "Buňka"
    Debug.Print ; "ANSI_B:"
    For i = 1 To Len(sText)
        Debug.Print AscB(Mid$(sText, i, 1));
    Next
    Debug.Print
End Sub

```

Tyto funkce vrátí výpisy podle obrázku.

Kód znaku ve formátu Unicode úzce souvisí s výpisem jednotlivých bytů metodou bytového pole. Například



Obrázek 1.20: Výpisy kódů AnsW a AnsB řetězce „Buňka“

první a druhý byte (66 a 0) odpovídají kódu prvního znaku vráceného funkcí `AscW()`, $66 + 0 * 256 = 66$, čili znak „B“. Pátý a šestý byte (72 a 1) odpovídají Unicode kódu znaku „ň“, $72 + 1 * 256 = 328$.

Funkce `AscB()` vrací pouze první byte kódu Unicode a v mezinárodním prostředí nevíme o žádném smysluplném použití.

Uznáváme, že tato látka může být poněkud nezáživná, avšak jsou situace, kdy je dobré znát, jak VBA řetězce ukládá. Jedním z důvodů je fakt, že pole ve VBA jsou velmi rychlá a zpracování řetězce manipulací s jeho byty může být rychlejší, než manipulace standardními metodami.

Následující příklad jistě není nijak zvláště užitečný, neboť pouze vytváří řetězec z pěti počátečních písmen abecedy, „abcde“ a opakováním vytvoří text o délce 10000 znaků:

```
Sub TestConcatenate()  
Dim str As String  
Dim i As Long, j As Long  
For i = 1 To 2000  
    For j = 97 To 101  
        str = str & Chr(j)  
    Next  
Next  
Debug.Print Len(str)  
End Sub
```

Následující příklad využívá podobný algoritmus, pouze nejprve запиše kódy znaků do bytového pole, a poté bytové pole přiřadí do textové proměnné:

```
Sub TestByteArray()  
Dim str As String  
Dim aByt(0 To 9999) As Byte  
Dim i As Long, j As Long  
For i = LBound(aByt) To UBound(aByt) Step 2  
    aByt(i) = 97 + j Mod 5  
    j = j + 1  
Next i  
str = aByt  
Debug.Print Len(str)  
End Sub
```

Všimněte si, že protože každý druhý byte bude nulový (jedná se o standardní znaky), stačí krok cyklu 2. Výsledky časového testu mohou překvapit:

```
10000  
TestConcatenate took      0,020 seconds  
10000  
TestByteArray took      0,002 seconds
```

Pokud máte problémy s rychlostí běhu programu, mohou vám bytová pole pomoci. Čím více je úprava řetězců závislá na matematických vztazích mezi kódy znaků, tím bude tato pomoc výraznější. Spojování řetězců pomocí operátoru „&“ (ampersand) je jednou z nejpomalejších operací ve VBA. Mějte tento fakt na paměti při návrhu vašeho kódu.

Na závěr této části lekce vám ještě ukážeme, jak se výše uvedený kód chová v prostředí diametrálně odlišném od českého, a sice pod Windows XP Pro RU.

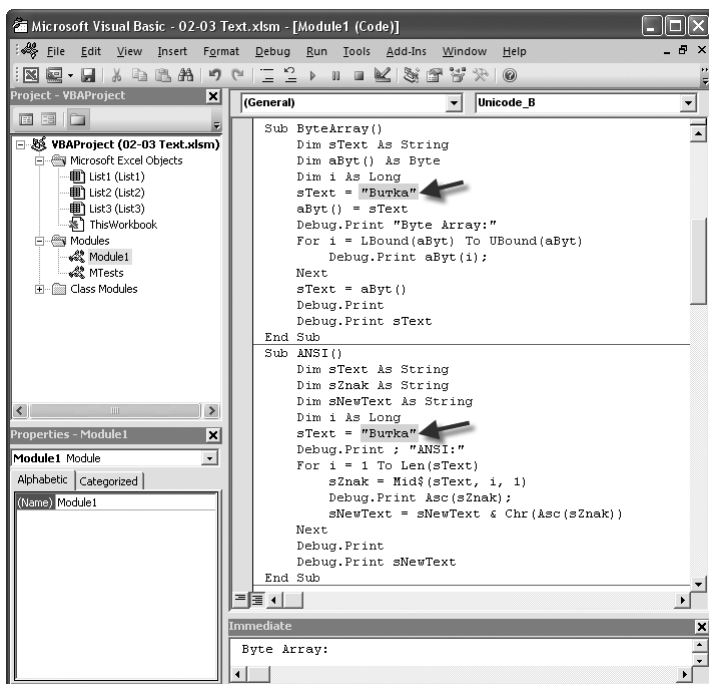


Poznámka

Vzhledem k tomu, že pracuji na Ukrajině a část knihy píší na ruském systému, je většina kódu v této knize otestována i v tomto prostředí. Domnívám se, že to není na škodu, protože při jeho funkčnosti na tak odlišných systémech existuje rozumná pravděpodobnost, že bude fungovat kdekoli v Evropě a Severní Americe. Většinu kódů v této knize není třeba zkusit na jiných systémech, neboť jsou na jazyku operačního systému nezávislé. Ale například v lekci o ukládání textu je takový test více než na místě. Nikdy není na škodu vědět, co se s vaším sešitem stane, až ho elektronickou poštou pošlete třeba právě obchodnímu partnerovi na východ.

Na tomto místě je na místě omluva za nedůslednou českou interpunkci v kódu. Jak za chvíli uvidíte, nečeské systémy si české znaky vykládají všelijak, protože VB Editor pracuje pouze se základní kódovou stránkou. Proto je lepší na nich české znaky vůbec nepoužívat, a vzhledem ke lhůtě vydavatele jsem nestačil všechny komentáře a řetězce v kódech počesit.

Již při otevření tohoto kódu v cizím prostředí vidíme, že něco není v pořádku:



Obrázek 1.21: Ukázka nesprávné interpretace českého znaku na ruském systému

Vzhledem k tomu, že editor VB pracuje s kódovou stránkou ANSI o 256 znacích, převedl nestandardní znaky podle jejich ANSI kódů. Proto se na místě původně českého „ř“ objevilo ruské „т“, v abuzce „т“. Ve VBA editoru se vám na ruských Windows český znak zapsat nepodaří, i když máte ve Windows zapnutou českou klávesnici (jako autor). Museli byste změnit kódovou stránku, ale to by zase mělo neblahé důsledky na spoustu dalších lokálních programů, které v daném systému běží. Stejně tak se vám na českém nebo anglickém systému nepodaří zapsat znak ruský, ať se snažíte sebevíc. Problémů s VBA v mezinárodním prostředí je více. Například se vám nepodaří přinutit standardní

funkci MsgBox, aby zobrazovala hlášení v různých abecedách. Pro odlišná prostředí musíte vytvořit vlastní formulář, který dialog MsgBox nahradí, což je při jeho poměrně vysoké variabilitě použití docela tvrdý oříšek, vyžadující použití vlastních tříd a metody zvané *subclassing*.

Po spuštění výše uvedených testovacích kódů na ruském systému dostaneme tento výsledek:

Modul VBA obsahující kód není ničím jiným, než obyčejným textovým souborem, kódovaným formátem ANSI (jak se sami můžete přesvědčit exportem modulu a otevřením např. v Poznámkovém bloku). Znak s kódem ANSI 242, který na českém systému odpovídá znaku „ň“, bude na jiném systému odpovídat znaku úplně jinému. Tento znak sice sdílí stejný ANSI kód aktivní stránky, ale z kódu Unicode je zřejmé, že se jedná o znak ruské abecedy (druhý byte je roven 4), jehož kód 1090 je roven $4 * 256 + 66$.

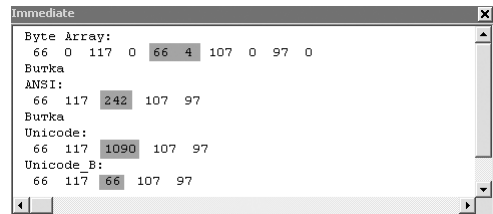
Všimněte si, že bez znalosti kódové stránky, na níž byl kód původně napsán, není možné žádným způsobem odvodit, jak zněl původní text!

Jedná se o další příklad toho, jak je nebezpečné (a neefektivní) do kódu VBA pro Excel uvádět literály, neboli natvrdo vepsané textové konstanty. Protože tato kniha pojednává o programování Excelu, měli byste řetězce zapsat do buněk a kódem VBA je načíst do proměnných. Jak list Excelu, tak i proměnná VBA uloží text správně ve formátu Unicode a žádná informace se neztratí.

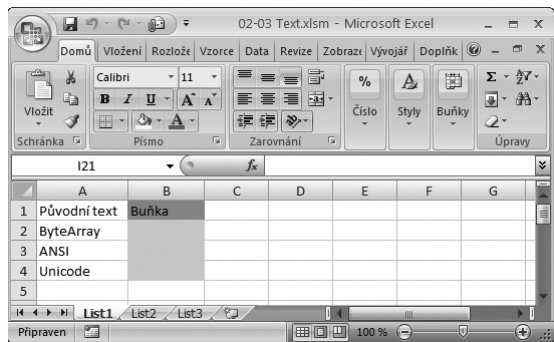
Následující příklad kódu předpokládá, že v buňce B1 aktivního listu je zapsán text „Buňka“ a výsledný text chceme zapsat do buňky B2 pro proceduru `ByteArray`, do buňky B3 pro proceduru `ANSI` a do buňky B4 pro proceduru `Unicode`, která používá funkce pracující s kódem Unicode.

Všechny tři procedury jsme mírně upravili, aby hodnoty načítaly z listu a výsledky na list opět zapisovaly:

```
Sub ReadWriteByteArray()
    Dim sText As String
    Dim aByt() As Byte
    Dim i As Long
    sText = [B1]
    aByt() = sText
    Debug.Print "Byte Array:"
    For i = LBound(aByt) To UBound(aByt)
        Debug.Print aByt(i);
    Next
End Sub
```



Obrázek 1.22: Písmeno „ň“ vrací na ruském systému zcela jiné hodnoty, neboť je interpretováno jako jiný znak



Obrázek 1.23: List připravený pro testování procedur pracujících s řetězcem načteným z listu

```

sText = aByt()
Debug.Print
[B2] = sText
End Sub

Sub ReadWriteANSI()
Dim sText As String
Dim sZnak As String
Dim sNewText As String
Dim i As Long
sText = [B1]
Debug.Print ; "ANSI:"
For i = 1 To Len(sText)
    sZnak = Mid$(sText, i, 1)
    Debug.Print Asc(sZnak);
    sNewText = sNewText & Chr(Asc(sZnak))
Next
Debug.Print
[B3] = sNewText
End Sub

Sub ReadWriteUnicode()
Dim sText As String
Dim sZnak As String
Dim sNewText As String
Dim i As Long
sText = [B1]
Debug.Print ; "Unicode:"
For i = 1 To Len(sText)
    sZnak = Mid$(sText, i, 1)
    Debug.Print AscW(sZnak);
    sNewText = sNewText & ChrW(AscW(sZnak))
Next
Debug.Print
[B4] = sNewText
End Sub

```

Po spuštění obou procedur obdržíme tento výsledek na listu:



Obrazek 1.24: Výsledek různých procedur, které přečtou a zpětně zapíší řetězec na list

A v okně Immediate dostaneme tento výstup:

Jak se můžete přesvědčit, v prvním a posledním případě byly všechny znaky interpretovány správně a rovněž kód odpovídá kódu ve výpisu v českém prostředí. Podle očekávání selhala metoda používající funkce Asc() a Chr(), pracující s ANSI kódem. Všimněte si chybějícího háčku nad „n“ a odlišného kódu ve výpisu. Zatímco 328 a 72 + 1 * 256 jsou kódy identické, druhý výpis je vzhledem k originálu nesprávný.

```

Immediate
Byte Array:
66 0 117 0 72 1 107 0 97 0
ANSI:
66 117 110 107 97
Unicode:
66 117 328 107 97
  
```

Obrázek 1.25: Výpis kódů jednotlivých znaků v okně Immediate



Tip

Snažte se vždy textové konstanty načítat z listu a nevpisujte je do kódu. Při použití znaků specifických pro daný jazyk to platí sto procentně!

Při práci v prostředí Windows dávejte pro práci s jednotlivými znaky a jejich kódy přednost funkcím AscW() a ChrW(), které pracují s kódy Unicode, před jejich základními variantami Asc() a Chr(), které mohou v mezinárodním prostředí přinést neočekávané výsledky.

Do řetězcových proměnných nikdy neukládejte binární data, jak bývalo dříve zvykem. Protože konverze ANSI–Unicode provádí VBA poněkud samovolně, téměř jistě o svá data přijdete.

Funkce porovnávající text

Po poněkud složitém (ale užitečném!) úvodu se nyní podíváme na funkce VBA, které vám umožňují měnit obsah textových proměnných podle vašich potřeb. Na varování z předchozí lekce jsme nezapomněli, ale pro většinou příkladů budeme používat jednoduché řetězce vepsané přímo do kódu VBA. V praxi načítejte raději text z listu do proměnných a ty teprve použijte v kódu.

VBA poskytuje tři způsoby porovnávání textů. Jsou to porovnávací operátory (=,<,> a podobně), operátor Like a konečně funkce StrComp(). Navíc můžete způsob porovnávání ovlivnit výrazem Option Compare v deklarační části modulu:

Option Compare Binary – Srovnání je vykonáno na základě binární reprezentace znaku. Srovnání je citlivé na velká a malá písmena. Pokud v záhlaví modulu výraz Option Compare nepoužijete, bude porovnávání provedeno na základě Option Compare Binary.

Option Compare Text – Srovnání je vykonáno na základě lokálního třídění textu, platného pro aktuální jazyk. Srovnání (alespoň pro běžné jazyky) nebere v úvahu velká a malá písmena.

```

(General) Test
Option Explicit
Option Compare Binary
Sub Test()
Debug.Print "Alpha" = "alpha"
End Sub
  
```

Immediate: False

Obrázek 1.26: Binární porovnání řetězců lišících se velikostí písmen

```

(General) Test
Option Explicit
Option Compare Text
Sub Test()
Debug.Print "Ř" < "ř"
End Sub
  
```

Immediate: True

Obrázek 1.27: Textové porovnávání řetězců

Option Compare Database – Tento výraz uvádíme pouze pro úplnost, neboť je použitelný pouze v modulech Microsoft Access. Srovnání proběhne na základě lokálního nastavení (Locale ID) dané databáze.

Nejběžnější metodou porovnávání řetězců je použití logických operátorů, například:

```
If String1 < String2 Then
    MsgBox "String1 předchází String2"
End If
```

Porovnávání bude závislé na znění výrazu `Option Compare` v záhlaví modulu. Výchozí metodou je porovnávání binární.

Můžete použít všechny platné kombinace srovnávacích operátorů:

Tabulka 1.2: Srovnávací operátory VBA

Operátor	Popis
<	menší než
<=	menší nebo rovno
=	rovno
>=	větší nebo rovno
>	větší než
<>	různé od (nerovná se)

Pokud máte například v buňkách seznam jmen, můžete děti rozřadit do tříd podle jména:



01-06 Text.xlsm
modul MComparisons

```
Sub RozradDoTrid()
    Dim cl As Range
    For Each cl In Range("A1:A10")
        Select Case cl.Value
            Case Is < "K"
                cl.Offset(, 1) = "A"
            Case Is < "R"
                cl.Offset(, 1) = "B"
            Case Else
                cl.Offset(, 1) = "C"
        End Select
    Next
End Sub
```

V „áčku“ budou děti se jménem začínajícím na A až J, v „béčku“ děti od K po Q, a v „céčku“ děti se jménem začínajícím na R až Z. Aby třídění proběhlo správně podle české abecedy, musí být v záhlaví výraz `Option Compare Text`, jinak by například Čenda patřil nesprávně do třídy „C“!

Pro porovnání dvou řetězců je ve VBA k dispozici i operátor `Like`. Tento operátor je velmi robustní, neboť umožňuje použití tzv. divokých karet (*wildcards*), neboli zástupných znaků. Pomocí těchto znaků můžeme definovat obecný vzor, se kterým řetězec porovnáváme.

Operátor Like přijímá zástupné znaky podle tabulky 1.3.

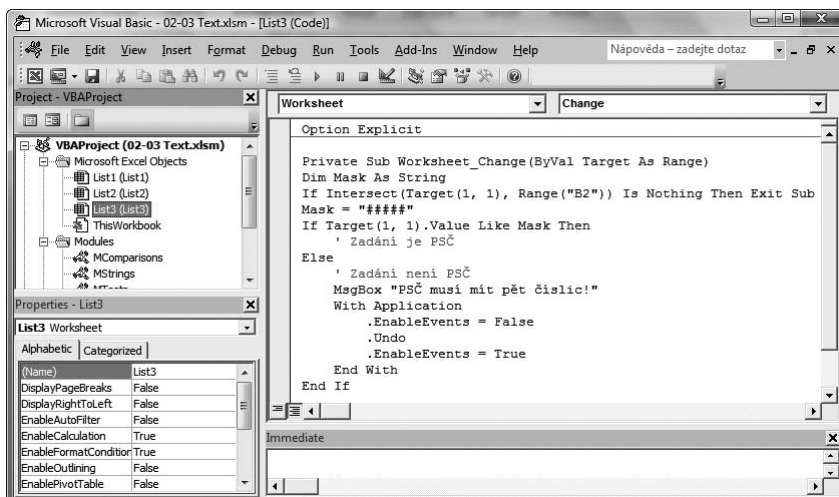
Tabulka 1.3: Znaky platné pro operátor Like

Znak ve vzoru	Vyhovující znak v řetězci
?	Jákýkoli znak
*	Libovolný počet znaků (i žádný)
#	Jakákoli číslice (0-9)
[seznam]	Jákýkoli znak s daného seznamu
[!seznam]	Jákýkoli znak kromě znaků ze seznamu

Seznam může být definován i rozsahem, např. [a-e]. Chování operátoru Like závisí na výrazu Option Compare v záhlaví modulu. Pokud tento výraz chybí, použije se binární srovnávání (v úvahu se berou velká a malá písmena).

Pokud má platné poštovní směrovací číslo pět číslic a chceme jej zapsat do buňky B2, můžeme napsat vlastní proceduru, která vyhodnotí vložený údaj, a pokud údaj nevyhovuje, aplikuje se akce Zpět (Undo), stejná jako klepnutí na tlačítko Zpět na panelu nástrojů Rychlý přístup. O programování událostí si řekneme více dále v této knize. Nyní si jen zapamatujte, že následující kód musí přijít do modulu třídy listu, který obsahuje vyhodnocovanou buňku:

```
Private Sub Worksheet_Change(ByVal Target As Range)
Dim Mask As String
If Intersect(Target(1, 1), Range("B2")) Is Nothing Then Exit Sub
Mask = "#####"
If Target(1, 1).Value Like Mask Then
' Zadání je PSČ
Else
' Zadání není PSČ
MsgBox "PSČ musí mít pět číslic!"
End If
End Sub
```



Obrázek 1.28: Událostní procedura ověřující platnost vstupních dat v listu „List3“

```

With Application
    .EnableEvents = False
    .Undo
    .EnableEvents = True
End With
End If
End Sub

```

Kód se nenachází v běžném modulu, nýbrž v modulu třídy daného listu (Obrázek 1.28).

Pokud provedeme nesprávné zadání, zobrazí se varovné hlášení a aplikuje se akce Zpět (metoda Undo), která buňku vrátí do původního stavu:

Podobně můžeme vyhodnotit údaj, zda se jedná o SPZ vozidla. Předpokládejme, že SPZ musí mít tvar 1A2-34-56, tedy tři skupiny znaků, oddělené pomlčkou, přičemž druhý znak v první skupině musí být velké písmeno. Maska, se kterou budeme porovnávat, bude mít tvar

```
Mask = "#[A-Z]#-##-##"
```

Pokud budeme pokračovat v předchozím příkladu, můžeme kód upravit tak, aby buňka B2 byla vyhodnocována na PSČ a buňka B3 na SPZ. Je to velmi užitečný příklad, který ukazuje, jak můžete napsat vlastní funkci ověření dat, a to mnohem mocnější, než funkci vestavěnou, dostupnou přes Data => Datové nástroje => Ověření dat.



01-06 Text.xlsm
modul listu Verifications

```

Private Sub Worksheet_Change(ByVal Target As Range)
Dim Mask As String

```

```

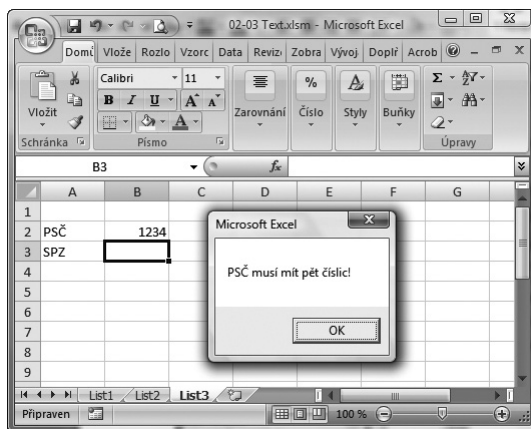
Select Case Target(1, 1).Address

```

```

' Sekce PSČ
Case Range("B2").Address
    Mask = "#####"
    If Target(1, 1).Value Like Mask Then
        ' Zadání je PSČ
    Else
        ' Zadání není PSČ
        MsgBox "PSČ musí mít pět číslic!"
        With Application
            .EnableEvents = False
            .Undo
            .EnableEvents = True
        End With
    End With

```



Obrázek 1.29: Reakce ověřovací procedury při zadání neplatného údaje

```

End If

'Sekce SPZ
Case Range("B3").Address
Mask = "#[A-Z]#-##-##"
If Target(1, 1).Value Like Mask Then
' Zadání je SPZ
Else
' Zadání není SPZ
MsgBox "SPZ musí mít tvar 1A2-34-56!"
With Application
.EnableEvents = False
.Undo
.EnableEvents = True
End With
End If

Case Else
'byla změněna neověřovaná buňka, nedělej nic

End Select

End Sub

```

Událostní procedura nyní ověřuje buňku B2 na platné PSČ a buňku B3 na platnou SPZ vozidla:

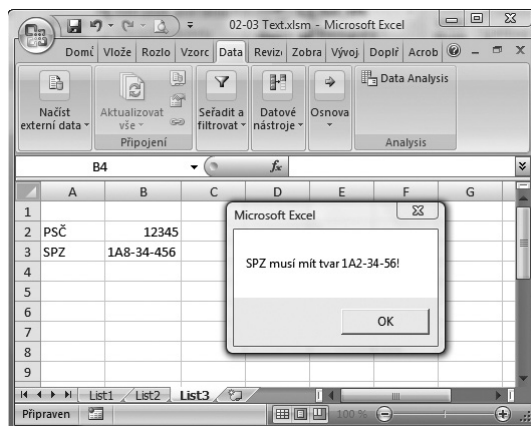
Pokud vás událostní procedury zaujaly, více o jejich programování se dozvíte dále v této knize, konkrétně v páté lekci.

Zatímco logické operátory i operátor Like v sobě skrývají potenciální nebezpečí, neboť jsou závislé na deklaraci `Option Compare`, třetí metoda VBA, porovnávající řetězce, funkce `StrComp()`, tuto překážku překonává.

Funkce `StrComp()` má následující obecnou syntaxi:

```
1 Return = StrComp(Text1, Text2, CompareOption)
```

`Text1` a `Text2` mohou být jakékoli dva řetězce. Hodnota `CompareOption` může být jedna z hodnot v následující tabulce, nebo `LocaleID` identifikátor, označující jazyk, kterým se má porovnávání řídit.



Obrázek 1.30: Reakce procedury ověřující dvě různé buňky při nesprávném zadání SPZ

Konstanta	Hodnota	Ekvivalent v záhlaví modulu
vbBinaryCompare	0	Option Compare Binary
vbTextCompare	1	Option Compare Text
vbDatabaseCompare	2	Option Compare Database (pouze pro MS Access)

Konstanta `CompareOption` může být vynechána. V takovém případě se porovnávání řídí deklarací `Option Compare`. Pokud i toto chybí, bude porovnání provedeno na základě `Compare Option Binary`, tedy s ohledem na malá a velká písmena.



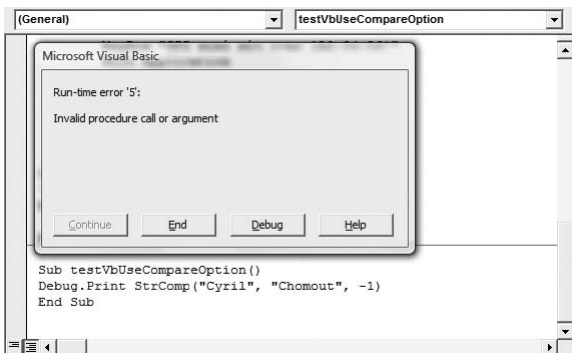
Tip

Nápověda VBA nabízí ještě konstantu `vbUseCompareOption` o hodnotě -1. Tato konstanta by dle nápovědy měla funkci `StrComp()` přinutit chovat se podle tvaru výrazu `Option Compare` v záhlaví modulu. Bohužel, tato konstanta nefunguje, jak se můžete přesvědčit z obrázku 1.31.

Ostatně, pokud napíšete do okna Immediate výraz

```
? vbUseCompareOption
```

a stisknete `Enter`, VBA zobrazí prázdný řetězec, z čehož je patrné, že uvedenou konstantu vůbec nezná. Tato varianta tady pravděpodobně pracovat nebude, ať už nápověda tvrdí cokoli (není to jediný příklad nepřesnosti v nápovědě).



Obrázek 1.31: Použití konstanty `vbUseCompareOption` nefunguje tak, jak uvádí nápověda

Funkce `StrComp()` vrací hodnotu podle následující tabulky:

Tabulka 1.4: Hodnoty, vrácené funkcí `StrComp`

Výsledek porovnání	Návratová hodnota
Text1 je menší než (předchází) Text2	-1
Text1 a Text2 jsou si rovny	0
Text1 je větší než Text2	1

Zde je několik příkladů v okně Immediate (za každým řádkem začínajícím otazníkem byla stisknuta klávesa Enter):

```
? Strcomp("Alpha","alpha")
-1
? Strcomp("Alpha","alpha",vbTextCompare)
0
? Strcomp("Beta","Alpha")
1
? Strcomp("Řeřicha","Sedmikráska")
1
? Strcomp("Řeřicha","Sedmikráska",vbTextCompare)
-1
```

Zajímavou možností je pomocí LocaleID funkci StrComp() přinutit používat pro porovnání specifický jazyk. Například angličtina nezná písmeno „Ch“ a s následujícím porovnáním může být problém:

```
? strcomp("Cyril","Chomout")
1
? strcomp("Cyril","Chomout",vbtextcompare)
1
```

Pro českou abecedu VBA nesprávně řadí řetězec „Cyril“ za řetězec „Chomout“. V češtině je ale písmeno „Ch“ až za písmenem „C“!

Řešením může být použití identifikátoru češtiny (1029) jako konstanty CompareOption:

```
? strcomp("Cyril","Chomout",1029)
-1
```

Pokud chcete zjistit LocaleID kódy pro jiné jazyky, najdete je například zde:

<http://msdn.microsoft.com/en-us/global/bb964664.aspx>

Jazyk musí být na vašem systému samozřejmě k dispozici, aby tato metoda fungovala.

Velká a malá písmena

Pro konverzi textu na velká, malá, či velká začáteční písmena nabízí VBA funkci StrConv(). Funkce má následující syntaxi:

```
strOut = StrConv(strIn, Conversion, LocaleID)
```

Argumenty *strIn* (vstupní řetězec) a *Conversion* (způsob konverze) jsou povinné. Argument *LocaleID* je nepovinný a může specifikovat, podle kterého národního standardu má být konverze provedena. Pokud je vynechán (a obvykle je), potom se použije aktuální nastavení Windows.

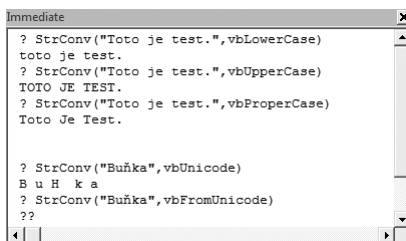
Konstanta *Conversion* musí být z následující tabulky:

Tabulka 1.5: Hodnoty konstanty *Conversion*, která určuje typ převodu

Konstanta	Hodnota	Ekvivalent v záhlaví modulu
vbUpperCase	1	Option Compare Binary
vbLowerCase	2	Option Compare Text
vbProperCase	3	Option Compare Database (pouze pro MS Access)

Nápověda uvádí ještě další možnosti, které jsou určeny pouze pro orientální jazyky. Za zmínku stojí pouze konstanty `vbFromUnicode` a `vbUnicode`, které převádějí text z nebo do kódu Unicode podle aktuální kódové stránky.

Jak funkce pracuje, je jasné z obrázku 1.32.



```

? StrConv("Toto je test.",vbLowerCase)
toto je test.
? StrConv("Toto je test.",vbUpperCase)
TOTO JE TEST.
? StrConv("Toto je test.",vbProperCase)
Toto Je Test.

? StrConv("Buřka",vbUnicode)
B u ř k a
? StrConv("Buřka",vbFromUnicode)
??
  
```

Obrázek 1.32: Různá použití funkce VBA StrConv



Tip

Méně známý je fakt, že kromě mezery rozeznává funkce `StrConv()` i jiné oddělovače slov, např. `vbLf` (Line Feed), znak `Chr$(10)`, `vbCr` (Carriage Return), znak `Chr$(13)` a jejich kombinaci, `vbCrLf`.

Pro převod písmen z velkých na malá a z malých na velká nabízí VBA ještě dvě standardní funkce, `LCase()` a `UCase()`. Jejich použití je jednoduché a rovněž akceptují i jiné oddělovače, než mezery:

```

strIn = "Toto" & vbLf & "je" & vbLf & "test."
? LCase(strIn)
toto
je
test.
  
```

```

? UCase(strIn)
TOTO
JE
TEST.
  
```

VBA nemá funkci ekvivalentní funkcím `LCase()` a `UCase()`, která by převáděla počáteční písmena na velká. Protože ale programujeme ve VBA pro Excel, můžeme si vypomoci funkcemi listu:

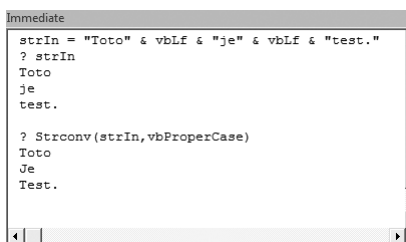
```

strIn = "Toto" & vbLf & "je" & vbLf & "test."
? Application.Proper(strIn)
Toto
Je
Test.
  
```

`Application.Proper` je zkrácený zápis, kompatibilní s Excelem 95. Jeho výhodou je kratší zápis, avšak funkce se neobjeví v seznamu, který získáme díky technologii Intellisense po zapsání tečky za mateřským objektem.

V seznamu se funkce objeví tehdy, pokud použijeme úplný zápis, tedy

```
Application.WorksheetFunction.Proper(strIn)
```

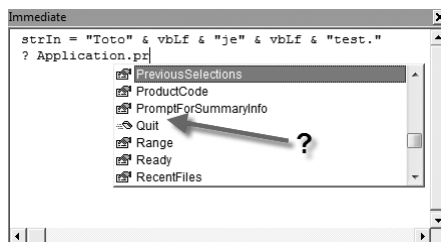


```

strIn = "Toto" & vbLf & "je" & vbLf & "test."
? strIn
Toto
je
test.

? StrConv(strIn,vbProperCase)
Toto
Je
Test.
  
```

Obrázek 1.33: Funkce StrConv se úspěšně vypořádá i s jinými oddělovači slov, než je mezera



```

strIn = "Toto" & vbLf & "je" & vbLf & "test."
? Application.Pr
PreviousSelections
ProductCode
PromptForSummaryInfo
Quit
Range
Ready
RecentFiles
  
```

Obrázek 1.34: Funkce listu nejsou standardně viditelné jako členy třídy Application

Můžete psát také (protože je objekt `WorksheetFunction` členem třídy `<Globals>`):

```
WorksheetFunction.Proper(strIn)
```

Obecně platí, že funkce, které jsou ve VBA přirozené k dispozici, nejsou k dispozici přes objekt `WorksheetFunction` a naopak. Proto například následující funkce listu k dispozici jsou,

```
Application.Max(range)
Application.Min (range)
Application.Cosh(0)
```

neboť VBA nemá přirozený ekvivalent.

Naproti tomu následující pokusy v okně Immediate ohlásí po stisknutí klávesy Enter chybu:

```
? Application.Cos(3.14159/180)
? Application.Upper("test")
? Application.Abs(-1)
```

Všechny tyto funkce mají své ekvivalenty ve VBA, a proto jsou přes objektový model Excelu nedostupné. Největší zmatek panuje patrně u funkcí matematických a finančních. Pokud si chcete být na první pohled jisti tím, jestli používáte v kódu vzorec listu nebo vestavěnou funkci listu, používejte plnou kvalifikaci objektem `VBA` nebo `WorksheetFunction`, protože některé funkce nejsou tím, čím se zdají být:

```
? VBA.Log(2.71828)
0,999999327347282
? WorksheetFunction.Log(2.71828)
0,434294189773888
```

Zatímco funkce `Log`, která je součástí knihovny VBA, vrací přirozený logaritmus o základu Eulera čísla (e), metoda `Log` objektu `WorksheetFunction` očekává dva parametry, jak je vidět z obrázku 1.35.

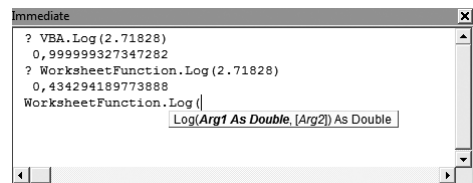
Pokud druhý parametr chybí, metoda vrací dekadický logaritmus. Protože podobné případy jsou zdrojem častých zmatků, doporučujeme méně obvyklé funkce vždy kvalifikovat slovem `VBA` nebo `WorksheetFunction` a raději jejich syntaxi ověřit v nápovědě a otestovat v okně Immediate.

Formátování textu bude věnována následující část lekcce. Aby byl ale přehled možností konverze velkých a malých písmen úplný, uvedeme ještě jednu, málo známou možnost:

```
strIn = "Toto je test."
? strIn
Toto je test.

? Format$(strIn,">@")
TOTO JE TEST.

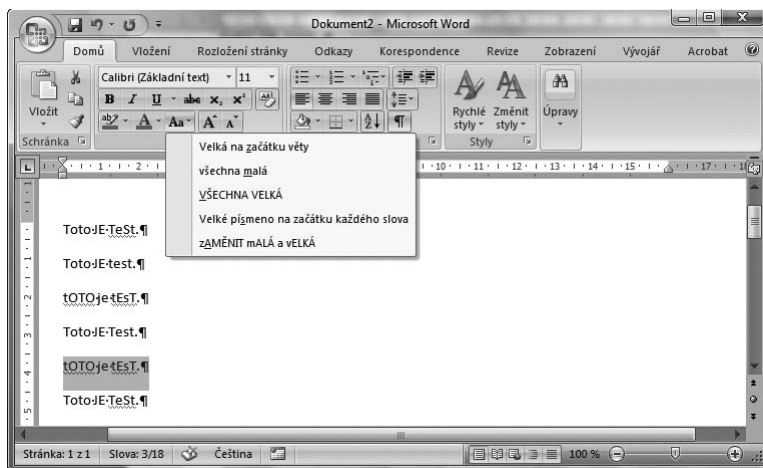
? Format$(strIn,"<@")
toto je test.
```



Obrázek 1.35: Použití funkce listu LOG v kódu VBA

Všechny metody, které jsme uvedli, jsou při správném použití ekvivalentní. Používejte tu, která se vám nejvíce zamlouvá, ale snažte se být konzistentní. Pouze v případě převodu prvních písmen na velká neexistuje ve VBA přímá metoda a musíte si „vypůjčit“ vzorec listu `PROPER()`. V české verzi Excelu bude mít na listu podobu `VELKÁ2()`.

A protože se v této knize učíte psát programy ve VBA především pro Excel, je čas na praktičtější příklad. V aplikaci Microsoft Word existuje vestavěné uživatelské rozhraní, které umožňuje rychlý převod vybraného textu. Některé možnosti jsou praktičtější, než jiné, ale pro většinu situací vyhovují:



Obrázek 1.36: Ukázka možností rychlého převodu písmen v aplikaci Word

Podobná funkce v Excelu neexistuje. Do listu musíte napsat odpovídající vzorce, výsledky vzorců zkopírovat a vložit přes původní data jako hodnoty.

Do Excelu však můžeme podobnou funkci doplnit pomocí kódu VBA. Navíc bude tato funkce obsahovat i možnost převést text na tzv. kapitálky (doufáme, že jsme se neprohřešili proti typografické terminologii), což znamená převod všech písmen na velká a změna velikosti fontu prvních písmen slov, případně všech ostatních. Toto je velmi elegantní efekt u zpráv. Naše procedura bude mít různé možnosti, podle výběru uživatele.

Úlohu vyřeší následující procedura:



01-06 Text.xlsm
modul MConversion

```
Sub ConvertCase()
```

```
Dim rg As Range, c1 As Range
```

Původní stav	Převod na velká	Převod na malá												
<table border="1"> <tr><th>A</th></tr> <tr><td>1 Toto je test</td></tr> <tr><td>2 TOTO JE TEST</td></tr> <tr><td>3 toto JE TeSt</td></tr> </table>	A	1 Toto je test	2 TOTO JE TEST	3 toto JE TeSt	<table border="1"> <tr><th>A</th></tr> <tr><td>1 TOTO JE TEST</td></tr> <tr><td>2 TOTO JE TEST</td></tr> <tr><td>3 TOTO JE TEST</td></tr> </table>	A	1 TOTO JE TEST	2 TOTO JE TEST	3 TOTO JE TEST	<table border="1"> <tr><th>A</th></tr> <tr><td>1 toto je test</td></tr> <tr><td>2 toto je test</td></tr> <tr><td>3 toto je test</td></tr> </table>	A	1 toto je test	2 toto je test	3 toto je test
A														
1 Toto je test														
2 TOTO JE TEST														
3 toto JE TeSt														
A														
1 TOTO JE TEST														
2 TOTO JE TEST														
3 TOTO JE TEST														
A														
1 toto je test														
2 toto je test														
3 toto je test														
Převod na větu	Převod na titulky	Převod na kapitálky												
<table border="1"> <tr><th>A</th></tr> <tr><td>1 Toto je test</td></tr> <tr><td>2 Toto je test</td></tr> <tr><td>3 Toto je test</td></tr> </table>	A	1 Toto je test	2 Toto je test	3 Toto je test	<table border="1"> <tr><th>A</th></tr> <tr><td>1 Toto Je Test</td></tr> <tr><td>2 Toto Je Test</td></tr> <tr><td>3 Toto Je Test</td></tr> </table>	A	1 Toto Je Test	2 Toto Je Test	3 Toto Je Test	<table border="1"> <tr><th>A</th></tr> <tr><td>1 TOTO JE TEST</td></tr> <tr><td>2 TOTO JE TEST</td></tr> <tr><td>3 TOTO JE TEST</td></tr> </table>	A	1 TOTO JE TEST	2 TOTO JE TEST	3 TOTO JE TEST
A														
1 Toto je test														
2 Toto je test														
3 Toto je test														
A														
1 Toto Je Test														
2 Toto Je Test														
3 Toto Je Test														
A														
1 TOTO JE TEST														
2 TOTO JE TEST														
3 TOTO JE TEST														

Obrázek 1.37: Různé převody textu pomocí vlastní procedury VBA

```

Dim msg As String, ans As String
Dim lFirstCapSize As Long, lFontSize As Long
Dim sTemp As String, i As Long

msg = "Zadejte písmeno pro převod textu ve vybrané oblasti:" & _
                                           String(2, vbCrLf)
msg = msg & "[U]" & _
        vbCrLf & "Převod na velká písmena" & vbCrLf
msg = msg & "[L]" & _
        vbCrLf & "Převod na malá písmena" & vbCrLf
msg = msg & "[V]" & _
        vbCrLf & "Převod prvního písmena na velké (věta)" & vbCrLf
msg = msg & "[T]" & _
        vbCrLf & "Převod prvních písmen na velké (titulek)" & vbCrLf
msg = msg & "[K]" & _
        vbCrLf & "Změna na kapitálky" & String(2, vbCrLf)
InputMe:
'Získáme zadání uživatele o způsobu konverze

ans = Application.InputBox(msg, "Zadejte písmeno", Type:=2)
If ans = "False" Then Exit Sub
If InStr(1, "ULVTK", UCase(ans), vbTextCompare) = 0 Or Len(ans) > 1 Then
    MsgBox "Nesprávné zadání!", vbInformation
    GoTo InputMe
End If

'z vybrané oblasti získáme buňky s textem
On Error GoTo Err_Handler
    If Selection.Cells.Count = 1 Then
        Set rg = Selection
    Else
        Set rg = Selection.SpecialCells(xlConstants, 2)
    End If
On Error GoTo 0

For Each cl In rg
    Select Case UCase(ans)
    Case "U": cl = UCase(cl.Text)
    Case "L": cl = LCase(cl.Text)
    Case "V"
        cl = UCase(Left$(cl.Text, 1)) & _
            LCase(Right$(cl.Text, Len(cl.Text) - 1))
    Case "T": cl = Application.Proper(cl.Text)
    Case "K"
        lFirstCapSize = cl.Font.Size
        lFontSize = Int(0.8 * cl.Font.Size)
        cl = UCase(cl.Text)
        cl.Font.Size = lFontSize

        ' do dočasného řetězce uložíme text,
        ' který bude obsahovat velká písmena na správných místech
        sTemp = Application.Proper(cl.Text)

```

```

    For i = 1 To Len(sTemp)
        If Mid$(sTemp, i, 1) = UCase(Mid$(sTemp, i, 1)) Then

            ' jedná se o velké písmeno
            cl.Characters(i, 1).Font.Size = lFirstCapSize

        End If
    Next

End Select
Next cl

ExitHere:
Exit Sub
Err_Handler:
MsgBox "Ve výběru pravděpodobně není žádná buňka, " & _
    "obsahující text.", vbCritical

Resume ExitHere
End Sub

```



Tip

Pokud tuto proceduru vložíte do modulu Personal.xlsx, bude dostupná vždy, když nastartujete Excel.

Pro začínajícího programátora může být v této proceduře příliš mnoho nové látky ke strávení. Proto vysvětlíme po částech, jak procedura pracuje.

Program začíná deklarací proměnných:

```

Dim rg As Range, cl As Range
Dim msg As String, ans As String
Dim lFirstCapSize As Long, lFontSize As Long
Dim sTemp As String, i As Long

```

Všimněte si, že proměnné, které patří logicky k sobě, jsou uvedeny na jednom řádku. Je to např. oblast a její jedna buňka, či textová proměnná a čítač, přes který budeme procházet textem znak po znaku.

Řádky

```

msg = "Zadejte písmeno pro převod textu ve vybrané oblasti:" & _
    String(2, vbCrLf)

msg = msg & "[U]" & _
    vbCrLf & "Převod na velká písmena" & vbCrLf
msg = msg & "[L]" & _
    vbCrLf & "Převod na malá písmena" & vbCrLf
msg = msg & "[V]" & _
    vbCrLf & "Převod prvního písmena na velké (věta)" & vbCrLf
msg = msg & "[T]" & _
    vbCrLf & "Převod prvních písmen na velká (titulek)" & vbCrLf
msg = msg & "[K]" & _
    vbCrLf & "Změna na kapitálky" & String(2, vbCrLf)

```

sestavují zprávu, která vyzve uživatele k zadání písmena ze seznamu. Celý řetězec by bylo možno uložit do proměnné `msg` jediným příkazem, ale z důvodu přehlednosti kódu a jeho snadné pozdější editace jsme zvolili tento způsob. V lekci o mikrooptimalizaci kódu VBA se dozvíte, že spojování řetězců není příliš efektivní. Protože se zde jedná o jednorázovou úlohu a ne o smyčku, bude vliv na rychlost kódu zanedbatelný.

Následuje část kódu, jež zobrazí dialog, který uživatele vyzve k zadání písmena, označujícího způsob konverze:

```
InputMe:
    'Získáme zadání uživatele o způsobu konverze

    ans = Application.InputBox(msg, "Zadejte písmeno", Type:=2)
    If ans = "False" Then Exit Sub
    If InStr(1, "ULVTK", UCASE(ans), vbTextCompare) = 0 Or Len(ans) > 1 Then
        MsgBox "Nesprávné zadání!", vbInformation
        GoTo InputMe
    End If
```

Dialog se sestaveným textem vypadá zhruba takto:

Přesný vzhled samozřejmě závisí na operačním systému a jeho nastavení. Zde jsme použili Windows Vista v základním nastavení.

Po uživatelské zadání a klepnutí na tlačítko OK nebo Storno se provádí kontrola. Pokud uživatel klepl na tlačítko Storno, bude mít proměnná `ans` hodnotu „False“ a kód se ukončí.

Pokud zadaný znak není na seznamu, nebo je počet znaků vyšší než 1, kód skočí na návěští `InputMe` a dialog se po zobrazení varovného hlášení zobrazí znovu. Použití funkce `UCASE()` v ověřování způsobí, že je zadání necitlivé na velká a malá písmena. Uživatel tedy může zadat jak „K“, tak i „k“ a procedura správně porozumí, že chce převést text na kapitálky.

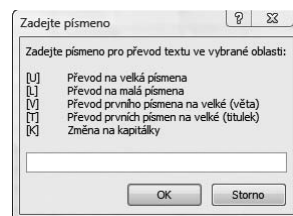
Následuje přiřazení oblasti, kterou budeme zpracovávat, do objektové proměnné:

```
'z vybrané oblasti získáme buňky s textem
On Error GoTo Err_Handler
    If Selection.Cells.Count = 1 Then
        Set rg = Selection
    Else
        Set rg = Selection.SpecialCells(xlConstants, 2)
    End If
On Error GoTo 0
```

Tato část kódu vám může připadat zbytečná. Proč přiřazovat oblast do proměnné, když můžeme procházet přímo přes buňky ve vybrané oblasti pomocí příkazu `For Each c1 in Selection.Cells?`

Později v této knize si řekneme něco o psaní robustního a rychlého kódu. V našem případě by si programátor měl položit následující otázky:

1. Co když má uživatel vybraný jiný objekt, než oblast buněk, například graf?
2. Co když uživatel vybere celý sloupec, který obsahuje jen několik buněk s textem?



Obrázek 1.38: Vlastní dialogové okno vyzývající uživatele k zadání volby

3. Co když uživatel vybere rozsáhlou oblast, která obsahuje několik titulků (jež chceme měnit) a jinak obsahuje pouze vzorce a čísla?

První případ by skončil chybou za běhu programu (run-time chybou), protože objekt `ChartObject` nemá vlastnost `Cells`. Druhé dva případy by pravděpodobně chybou neskončily, ale kód by naprosto zbytečně prohledával prázdné buňky nebo buňky, které změnit nechceme nebo nemůžeme!

V kódu bylo nutné ošetřit případ, kdy je vybrána pouze jedna buňka, protože při jediné vybrané buňce by metoda `SpecialCells` prohledávala celý list (v souladu s chováním uživatelského rozhraní).

Po ověření zadání a platnosti vybrané oblasti kód vstupuje do cyklu, kdy prochází každou buňku v oblasti textových hodnot (`For Each cl in rg`) a mění její obsah v závislosti na uživatelské zadání (`Select Case`).

Pro převod na velká, malá a velká počáteční písmena jsou použity standardní funkce VBA a Excelu:

```
Case "U": cl = UCase(cl.Text)
Case "L": cl = LCase(cl.Text)
Case "T": cl = Application.Proper(cl.Text)
```

Pro převod prvního písmena na velké a zbytek na malá (věta) jsme použili dvě funkce VBA, které si podrobněji představíme později v této lekci, `Left$()` a `Right$()`. Vrací část textu zleva nebo zprava o zadaném počtu znaků:

```
Case "V"
    cl = UCase(Left$(cl.Text, 1)) & _
        LCase(Right$(cl.Text, Len(cl.Text) - 1))
```

Nejsložitější částí procedury je úsek, který převádí text na velká písmena a zvětší font počátečních písmen slov. Nejprve uložíme velikost původního fontu do proměnné, vypočítáme novou velikost, převedeme text na velká písmena a font zmenšíme:

```
lFirstCapSize = cl.Font.Size
lFontSize = Int(0.8 * cl.Font.Size)
cl = UCase(cl.Text)
cl.Font.Size = lFontSize
```

Posledním úkolem je najít počáteční písmena slov a pomocí vlastnosti `Characters` objektu `Range` změnit velikost jejich fontu. Formátování jednotlivých znaků bylo zavedeno v Excelu 2000, kdy byl zaveden pro buňky formát *Rich Text*. Pro určení pozice počátečních písmen by bylo možno použít více způsobů. V našem příkladu jsme se spolehli na relativní robustnost funkce `Proper()` a s její pomocí vytvořili dočasný řetězec (`sTemp`). Pomocí porovnání znaku s jeho „UCase verzí“ zjistíme, jestli má být velikost jeho fontu změněna:

```
sTemp = Application.Proper(cl.Text)

For i = 1 To Len(sTemp)
    If Mid$(sTemp, i, 1) = UCase(Mid$(sTemp, i, 1)) Then

        ' jedná se o velké písmeno
        cl.Characters(i, 1).Font.Size = lFirstCapSize

    End If
Next
```

Poslední částí kódu je ovladač očekávané chyby, která nastane za předpokladu, že ve výběru nelze najít buňky či buňky s textem:

```
Err_Handler:
    MsgBox "Ve výběru pravděpodobně není žádná buňka, " & _
        "obsahující text.", vbCritical
    Resume ExitHere
End Sub
```

V takovém případě se zobrazí varovné hlášení, kód odskočí na návěští `ExitHere` a okamžitě se ukončí.

Formátování textu

Na úvod této části lekce uvedeme dvě funkce, které umějí textový řetězec vytvářet.

Funkce Space

Funkce `Space()` má syntaxi:

```
strOut = Space(1Number)
```

Tato funkce vytvoří řetězec o daném počtu mezer. Pokud známe počet mezer předem, můžeme takový řetězec vytvořit již v sekci deklaraci proměnných:

```
Dim strOut As String * 255
```

nebo později v programu, zejména pokud počet mezer není předem znám:

```
Dim strOut As String, strIn As String
strIn = "Martin"
strOut = Space(Len(strIn))
Debug.Print Len(strOut)
```

Tento kód vypíše číslo 6, neboť nový řetězec obsahuje šest mezer.

Funkce `Space()` se používá nejčastěji v případě, kdy je nutné vytvořit zásobník (buffer) pro budoucí text. Jedním takovým případem je volání API (Application Programming Interface) funkcí Windows. Tyto funkce jsou součástí standardních knihoven Windows a jejich použití – jak si později ukážeme – má přesná pravidla, jejichž porušení může vést k pádu Windows.



01-06 Text.xlsm
modul M_APls

```
Private Declare Function GetUserName _
    Lib "advapi32.dll" Alias "GetUserNameA" _
    (ByVal lpBuffer As String, _
    nSize As Long) _
    As Long

Public Function WindowsUserName() As String
Dim sName As String
sName = Space(255)
If GetUserName(sName, 255) Then
    sName = Left$(sName, 244)
```

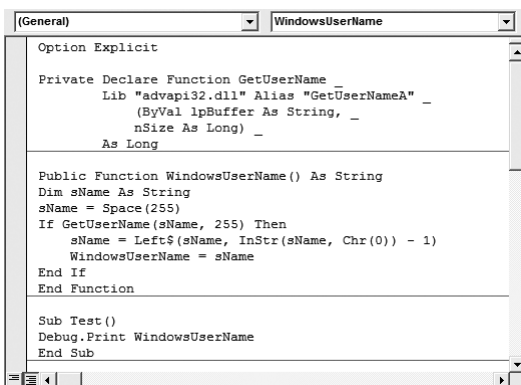
```
WindowsUserName = sName
End If
End Function
```

Jakmile tuto funkci napíšete, její použití je již triviální:

```
Sub Test()
Debug.Print WindowsUserName
End Sub
```

Deklarace funkce API musí být vždy v záhlaví modulu, v jeho deklarační části:

Praktické použití může být například v modulu `ThisWorkbook`, kdy můžete při otevření sešitu zkontrolovat přihlášeného uživatele a při neshodě jmen sešit uzavřít.



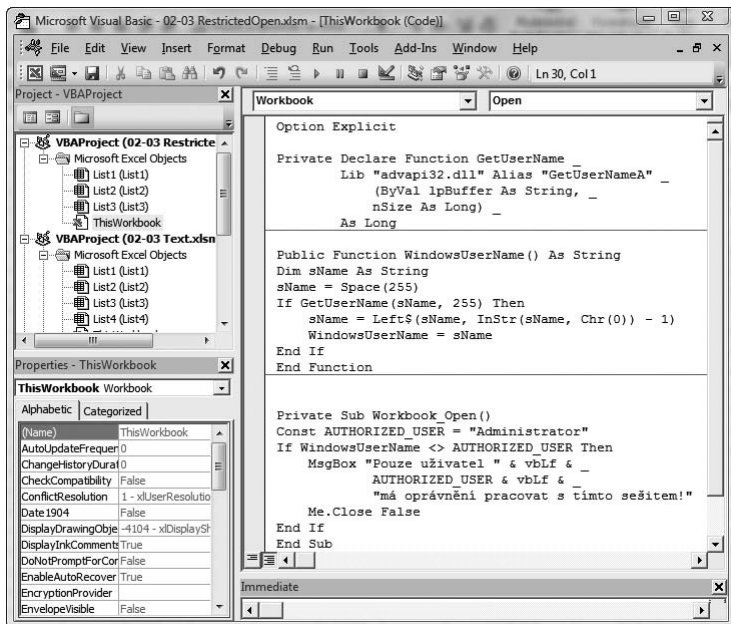
```
(General) WindowsUserName
Option Explicit

Private Declare Function GetUserName _
    Lib "advapi32.dll" Alias "GetUserNameA" _
    (ByVal lpBuffer As String, _
    nSize As Long) _
    As Long

Public Function WindowsUserName() As String
Dim sName As String
sName = Space(255)
If GetUserName(sName, 255) Then
    sName = Left$(sName, InStr(sName, Chr(0)) - 1)
    WindowsUserName = sName
End If
End Function

Sub Test()
Debug.Print WindowsUserName
End Sub
```

Obrázek 1.69: Deklarace a použití funkce API v modulu VBA



Obrázek 1.40: Praktické použití funkce API k ověření oprávnění otevřít sešit

Funkce String

Druhou funkcí, která vytváří řetězec obsahující libovolné stejné znaky, je funkce `String()`, která má dvě verze:

```
strOut = String(lChars, sChar)
```

kde lChars je počet znaků a sChar je znak, kterým se má řetězec naplnit. Druhou verzí je

```
strOut = String(lChars, lANSI)
```

kde lChars je počet znaků a lANSI je kód znaku podle aktuální kódové stránky.

Jak funkce pracuje, je zřejmé z následujících výpisů v okně Immediate:

```

? String(10, "9")
9999999999
? String(10, "a")
aaaaaaaaaa
? String(5, 65)
AAAAA
? String(5, 63)
?????
? string(1, 242)
ě

```

Obrázek 1.41: Různá použití funkce String

Funkci String() jsme již použili v kódu například pro vložení jednoho prázdného řádku do zobrazené zprávy:

```
msg = msg & "[K]" & vbCrLf & "Změna na kapitálky" & String(2, vbCrLf)
```

Funkce Format

Základem formátování řetězců je funkce Format() a její řetězcová varianta, Format\$(). Všimněte si, že se většina funkcí pracujících s řetězci vyskytuje ve dvou „příchutích“, se znakem „\$“ na konci a bez. Takovými funkcemi jsou vedle funkce Format() i funkce Left(), Right(), Mid(), InStr() a další. Rozdíl je v tom, že funkce s příponou „\$“ vracejí typ String (řetězec), zatímco funkce bez této přípony vracejí typ Variant. Protože při práci s řetězci v naprosté většině ukládáme data do řetězcových proměnných, bude varianta s příponou „\$“ rychlejší, protože VBA nemusí provádět dodatečný převod datového typu.

Proč se tedy vůbec zabýváme variantami funkcí bez této přípony, vracející typ Variant? Je to proto, že VBA zná zvláštní stav proměnné typu Variant, kdy obsahuje hodnotu Null. Zatímco funkce bez přípony „\$“ umí tuto hodnotu zpracovat (výsledkem je opět hodnota Null), funkce řetězcové vrátí run-time chybu.



01-06 Text.xlsm
modul MFormat

```

Sub TestStrings ()
Dim var As Variant
Dim str As String
Dim nil As Variant

nil = Null
var = Format(nil, "#,##0.00")
str = Format$(nil, "#,##0.00")
var = Left(nil, 1)

'tento řádek ohlásí run-time chybu
str = Left$(nil, 1)
End Sub

```

Obrázek 1.42: Řetězcová varianta funkce Left() neumí pracovat s hodnou Null

Hodnota Null má velmi zvláštní vlastnosti. Nejlepším českým ekvivalentem je asi slovo Nic. Pro programátora Excelu tato hodnota valný význam nemá, ale například když programujeme databáze Access, případně pokud z Excelu zpracováváme data v Accessu, musíme dávat velký pozor. „Buňky“ v tabulce Access, které neobsahují žádný údaj, mají totiž obecně hodnotu Null. Hodnota Null nemůže být rovna žádné jiné hodnotě či řetězci a *nemůže být dokonce rovna ani sama sobě*.

Předpokládejme, že buňka A1 aktivního listu je prázdná a chceme ji otestovat:

První řádek výpisu se vztahuje k předešlému kódu a dokazuje, že proměnná `var` obsahuje hodnotu `Null`, jako výsledek výrazu `var = Left(nil, 1)`, kde proměnná `nil` obsahuje hodnotu `Null`.

Z výpisu je vidět, že pro testování buňky na „prázdnotu“ se hodí porovnání s takzvaným nulovým řetězcem neboli řetězcem o nulové délce. Vhodnější je volání tohoto řetězce konstantou `vbNullString`, protože tato konstanta zavolá přímo funkci C Windows, která na prázdný řetězec ukazuje, aniž by ho kód musel vytvářet. Kromě toho se domníváme, že je zápis `vbNullString` přehlednější než „“.

Porovnání s prázdným řetězcem ovšem vrátí hodnotu `True` i tehdy, pokud je v buňce vzorec, jehož výsledkem může být prázdný řetězec, např. `=KDYŽ(C1;D1/C1;““)`. Pro testování, zdali je buňka skutečně prázdná, se proto nejlépe hodí funkce `IsEmpty()`. Z výpisu je vidno, že testování na hodnotu `Null` skončí v Excelu neúspěchem.

Protože v této knize programujeme především pro Excel, budeme se u textových funkcí většinou držet syntaxe s příponou „\$“.

Funkce `Format$()` má následující obecnou syntaxi:

```
strOut = Format$(expression[, format[, firstdayofweek[, firstweekofyear]])
```

Popis všech možných hodnot, které lze předat jako parametry, je zbytečné uvádět, protože si je čtenář snadno najde v nápovědě. Stačí napsat v editoru VBA kamkoli do modulu klíčové slovo „Format“, umístit do něj kurzor a stisknout klávesu F1.

- ◆ *Expression* je jakýkoli výraz nebo hodnota, kterou chceme formátovat
- ◆ *Format* je řetězec, sdělující funkci `Format()`, jak má formátování vypadat.

O formátování data a času si řekneme více později v této lekci.

Řetězec `Format` je jakousi maskou, která může pro formátování textu obsahovat následující znaky:

Tabulka 1.6: Přijatelné znaky masky pro formátování textu pomocí funkce `Format`

Znak	Popis
@	Zástupný znak pro jakýkoli znak nebo mezeru. Pokud se na daném místě ve formátovaném řetězci vyskytuje znak, funkce <code>Format\$()</code> jej zobrazí. Jinak na tomto místě vrátí mezeru. Zástupné znaky jsou zaplňovány zprava doleva.
&	Zástupný znak pro jakýkoli znak nebo nic. Pokud se na daném místě ve formátovaném řetězci vyskytuje znak, funkce <code>Format\$()</code> jej zobrazí. Jinak na tomto místě nevrátí nic (míněno řetězec o nulové délce). Zástupné znaky jsou zaplňovány zprava doleva.
<	Všechny znaky budou převedeny na malá písmena. Na umístění v masce nezáleží.
>	Všechny znaky budou převedeny na velká písmena. Na umístění v masce nezáleží.
!	Zástupné znaky budou zaplňovány zleva doprava.

```

Immediate
? var
Null

? [A1] = ""
True
? [A1] = vbNullString
True
? isempty([A1])
True
? isnull([A1])
False
? [A1] = Null
Null
? Null = Null
Null

```

Obrazek 1.43: Testování obsahu buňky a hodnoty `Null`

Pro formátování čísel, data a času se používají další rezervované znaky. Pokud má řetězec obsahovat nějaký konkrétní znak, jednoduše se do masky zapíše. Pokud se jedná o znak rezervovaný, např. m, s, d, musí ho předcházet zpětné lomítko (*backslash*).

Podívejte se na následující výpis:

Všimněte si formátování PSČ. První pokus má dva nedostatky. Za prvé, maska obsahuje znak „s“, který je vyhrazen pro formátování sériového čísla na sekundy. Protože se jedná o text, na tomto místě je vrácena nula. Rovněž dvojtečka je rezervovaný znak, používaný pro formátování času. Proto dalším znakům za dvojtečkou funkce `Format$()` nerozumí. Řešením je použití zpětného lomítka.

Protože rezervovaných znaků je poměrně velké množství, někteří programátoři používají zpětné lomítko před každým znakem, který chtějí zobrazit ve výsledném řetězci:

```
? Format$("ABCDE", "& \N\i\c")
ABCDE
? Format$(vbNullstring, "& \N\i\c")
Nic
```

Ukázka obsahuje rovněž příklad náhradního formátu, pokud je předán nulový řetězec. Náhradní formát je umístěn za znakem středníku.

Se změnou písmen z malých na velká a naopak jsme se již setkali. Tato funkce vrátí velká písmena a mezi znaky vloží mezery:

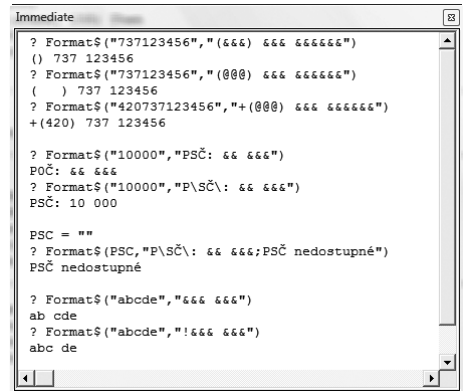
```
? Format$("Ahoj", "> @ @ @ @")
A H O J
```

Pokud počet znaků předem neznáme, můžeme napsat takovou jednoduchou funkci, která mezi znaky vloží libovolné znaky (při vynechání znaku bude vložena mezera):

```
Function InsertChars(str As String, _
    Optional sChar As String = " ")
    Dim i As Long
    Dim Mask As String
    For i = 1 To Len(str)
        Mask = Mask & "@" & sChar
    Next i
    Mask = Left$(Mask, Len(Mask) - 1)
    InsertChars = Format$(str, Mask)
End Function
```

Zde jsou příklady použití:

```
? InsertChars("Ahoj")
A h o j
? InsertChars("Ahoj", "-")
A-h-o-j
```



```
Immediate
? Format$("737123456", "(### ###)")
( ) 737 123456
? Format$("737123456", "(###) ###")
( ) 737 123456
? Format$("420737123456", "+(###) ###")
+(420) 737 123456
? Format$("10000", "PŠČ: ##")
PŠČ: ##
? Format$("10000", "PŠČ: \##")
PŠČ: 10 000
PŠČ = ""
? Format$(PŠČ, "PŠČ: \##; PŠČ nedostupné")
PŠČ nedostupné
? Format$("abcde", "###")
ab cde
? Format$("abcde", "!###")
abc de
```

Obrázek 1.44: Formátování řetězce pomocí funkce `Format`

```
? InsertChars("123", "_")
1_2_3
```

Stále častěji v našem textu používáme funkce, které manipulují s řetězcovými proměnnými. V následující části lekce se na tyto funkce podíváme blíže.

Hledání znaků v řetězci

Nalezení délky řetězce – Funkce Len

Velmi důležitou funkcí je funkce Len(), která vrací počet znaků v daném řetězci. Její syntaxe je jednoduchá:

```
lLength = Len(String)
```

kde String je libovolný řetězec. Funkce Len() má jednu velmi důležitou vlastnost: *je rychlá!* Vzhledem k tomu, že VBA ukládá řetězce včetně údaje o jejich délce, je tato informace získána velmi rychle. Proto, pokud potřebujete otestovat, zda je řetězec prázdný (má nulovou délku), nepoužívejte tento zápis:

```
If str = vbNullString
```

a už vůbec ne tento:

```
If str = "" Then
```

K testu používejte jeden z těchto příkazů:

```
If Len(str) > 0 Then
```

nebo

```
If CBool(Len(str)) Then
```

Druhá varianta využívá skutečnosti, že funkce CBool() převede jakékoli číslo různé od nuly na hodnotu True.

Použití funkce Len pro běžný řetězec je triviální:

```
? Len("Ahoj, světe!")
12
```

Pokud je funkci předána hodnota Null, funkce vrací hodnotu Null:

```
? Len("")
0
? Len(vbNullString)
0
? Len(Null)
Null
```

Pro úplnost dodejme, že se funkce Len() vyskytuje i ve variantě LenB(), jež vrací počet bytů, které řetězec reprezentují v paměti počítače. Jak víme, VBA ukládá řetězce ve formátu *Unicode*, kdy je každý znak reprezentován dvěma byty:

```
? LenB("Ahoj")
8
```

```
? LenB(StrConv("Ahoj",vbFromUnicode))
4
```

Hledání znaků v řetězci – Funkce InStr a InStrRev

Vyhledávání pozice znaků v řetězci a zjištění, zda daný znak v řetězci vůbec existuje, je jednou z nejčastějších úloh v procedurách VBA pro Microsoft Excel, a proto tyto funkce musíte bezpečně ovládat! Naštěstí má VBA k dispozici dvě poměrně uživatelsky přívětivé funkce, InStr() a InStrRev(). Funkce InStrRev() byla zavedena ve VB6, a tedy od verze Excel 2000. Buďte proto při jejím používání opatrní.

Funkce InStr() má tuto obecnou syntaxi:

```
lPosition = InStr([lStart, ]strSearched, strSought[, CompareOption])
```

Parametry strSearched (prohledávaný text) a strSought (hledaný text) jsou povinné. Parametr lStart je volitelný a může označovat pozici, od které má hledání začít. Pokud tento parametr chybí, je řetězec prohledávaný od začátku.

Parametr CompareOption má stejná pravidla použití jako u funkce StrComp(). Určuje, jak se má nakládat s velkými a malými písmeny, a rovněž může určit, podle kterého nainstalovaného jazyka má srovnávání proběhnout. Pokud chybí, je použita definice Option Compare v záhlaví modulu, a pokud chybí i ta, proběhne srovnání binární (citlivé na velikost písmen).

Pokud je uveden parametr OptionCompare, musí být uveden i parametr lStart (i když je výchozí, tedy roven jedné).

Zde je několik příkladů, které chování funkce ozřejmí:

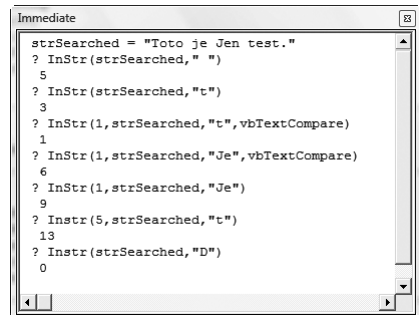
Pokud je parametr lStart větší než délka prohledávaného řetězce, funkce vrátí nulu:

```
? InStr(6,"abcde","e")
0
```

Pro praxi je užitečné znát i chování funkce, pokud je jeden z řetězců roven hodnotě Null, nebo má nulovou délku:

```
? Instr(vbNullString,"a")
0
? Instr("abcde",vbNullString)
1
? Instr(Null,"a")
Null
? InStr("abcde",Null)
Null
```

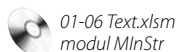
V praxi se funkce InStr() většinou používá společně s funkcemi Left() a Right(), kdy nejprve nalezneme pozici hledaného znaku a potom řetězec na tomto znaku uřízneme. Pro zjednoduše-



Obrázek 1.45: Vyhledávání pozice znaku pomocí funkce InStr

ní předpokládejme, že v buňkách A2 až A5 máme emailové adresy a jediným kritériem platnosti adresy je, zda obsahuje znak „@“ (zavináč, anglicky „at“), který nesmí být na začátku ani na konci.

Úlohu řeší tento kód:



01-06 Text.xlsm
modul MInStr

```
Sub ValidateEmail()
```

```
    Dim rg As Range, cl As Range
    Dim lPos As Long
```

```
    Set rg = ActiveSheet.Range("A2:A5")
    For Each cl In rg
        lPos = InStr(cl, "@")
```

```
        'zavináč nesmí být na začátku ani na konci
```

```
        If lPos > 1 And lPos < Len(cl) Then
```

```
            cl.Offset(, 1) = "Ano"
```

```
        Else
```

```
            cl.Offset(, 1) = "Ne"
```

```
        End If
```

```
    Next
```

```
End Sub
```

Samozřejmě že v praxi toto pro ověření alespoň teoretické platnosti emailové adresy nestačí, ale jako praktická ukázka použití funkce `InStr()` je příklad názorný.

Pro úplnost uvedme, že podobně jako funkce `Len()` má i funkce `InStr()` svou variantu, která vrací pozici nalezeného bytu v řetězci Unicode:

```
? InStrB("Ahoj", "h")
3
```

Pokud chcete prohledávání začít od konce, použijte (ve verzi Excel 2000 a vyšší) funkci `InstrRev`, která má velmi podobnou syntaxi. Jediným rozdílem je, že prohledávání začíná od konce řetězce:

```
lPosition = InStrRev(strSearched, strSought [, lStart[, CompareOption]])
```

Pokud neudáte parametr `lStart`, nebo předáte `-1`, hledání začne od konce řetězce. Můžete specifikovat i jiný začátek hledání:

```
strSearched = "Toto je Test"
```

```
? InStrRev(strSearched, " ")
8
```

```
? InStrRev(strSearched, "o")
```

```
4
```

Původní stav			Výsledný stav		
	A	B		A	B
1	Adresa	Platnost	1	Adresa	Platnost
2	john@hotmail.com		2	john@hotmail.com	Ano
3	martin@seznam.cz		3	martin@seznam.cz	Ano
4	honza		4	honza	Ne
5	maria@		5	maria@	Ne
6			6		

Obrázek 1.46: Zjednodušené ověření platnosti emailové adresy

```
? InStrRev(strSearched,"T")
9
? InStrRev(strSearched,"T",8)
1
? InStrRev(strSearched,"T",8,vbTextCompare)
3
```

Funkce má běžnější použití, než se na první pohled zdá. Běžným úkolem je získat jméno adresáře z jeho úplné cesty, vrácené například přes objekty knihovny Windows Scripting. K tomu potřebujeme zjistit pozici posledního zpětného lomítka a vrátit všechny znaky, následující za tímto znakem.

V následujícím příkladu předpokládáme, že chceme do sešitu vložit nový list, zapsat hlavičku „Folders“ a pod tuto hlavičku zapsat podadresáře nacházející se v daném adresáři, včetně jejich velikosti v kilobytech:



01-06 Text.xlsm
modul MInStr

```
Sub ListSubfolders()

    Const MY_FOLDER As String = _
        "D:\Users\KingMartin\Documents\Visual Studio 2008"
    Const SHEET_NAME As String = "Folders"
    Dim FullName As String, FolderName As String
    Dim lPos As Long
    Dim lFreeRow As Long

    Dim fso As Object, fld As Object, f As Object
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set fld = fso.GetFolder(MY_FOLDER)
    If fld.subfolders.Count = 0 Then
        MsgBox "No subfolders in directory" & _
            String(2, vbLf) & _
            MY_FOLDER
        GoTo ExitHere
    End If

    'tiše odstranit list, pokud existuje
    With Application
        .DisplayAlerts = False
        On Error Resume Next
        Sheets(SHEET_NAME).Delete
        On Error GoTo 0
        .DisplayAlerts = True
    End With

    With Worksheets.Add
        .Name = SHEET_NAME
        .Range("A1:B1") = Array(SHEET_NAME, "Size")
        For Each f In fld.subfolders
            FullName = f.Name
            lPos = InStrRev(FullName, "\")
```

```

FolderName = Right$(FullName, Len(FullName) - 1Pos)
1FreeRow = .Cells(.Rows.Count, "A").End(3).Row + 1
.Cells(1FreeRow, "A") = FolderName
On Error Resume Next
.Cells(1FreeRow, "B") = Format$(f.Size \ 1024, "#,##0 kB")
On Error GoTo 0
Next
.Range("A:B").EntireColumn.AutoFit
End With

```

```

ExitHere:
Set fso = Nothing
End Sub

```

Výsledek může vypadat například takto:

Funkce, vracející upravený řetězec

V příkladech v předešlých částech lekce jsme často naráželi na nutnost vrátit část řetězce, která byla použita dále v kódu. Je tedy nejvyšší čas, abychom se na tyto funkce podívali blíže. V této části lekce si představíme následující funkce a výrazy VBA:

	A	B	C	D	E	F
1	Folders	Size				
2	Code Snippets	0 kB				
3	Projects	17.177 kB				
4	Settings	226 kB				
5	Templates	0 kB				
6						
7						

Obrázek 1.47: Výpis podsložek dané složky včetně velikostí

Tabulka 1.7: Funkce modifikující řetězec

Funkce	Popis
Left(), Left\$()	Vrací specifikovaný počet znaků zleva
Right(), Right\$()	Vrací specifikovaný počet znaků zprava
Mid(), Mid\$()	Vrací specifikovaný počet znaků, počínaje danou pozicí
Trim(), LTrim(), RTrim()	Očistí text od přebytečných mezer na začátku a/nebo na konci
Replace(), Replace\$()	Nahradí specifikovaný řetězec jiným
Výrazy LSet, RSet	Položí nový řetězec přes starý, přebytečná místa jsou nahrazena mezerami.
Výraz Mid	Nahradí specifikovaný počet znaků jinými

Funkce Left a Right

Funkce Left() a Left\$() mají následující syntaxi:

```
strOut = Left$(strIn, Length)
```

kde strIn je vstupní řetězec a Length udává počet znaků zleva, které má funkce vrátit. Pokud je Length rovno nule, bude vrácen prázdný řetězec, Pokud bude řetězec obsahovat hodnotu Null, vrátí funkce Left() hodnotu Null a funkce Left\$() vyvolá chybu za běhu programu. Pokud bude Length rovna délce řetězce nebo větší, funkce vrátí celý řetězec.

```
strIn = "Toto je test"
```

```
? Left$(strIn,4)
```

Toto

```
? Left$(strIn,InStr(strIn," ")-1)
Toto
```

```
? Left$(strIn,InStrRev(strIn," ")-1)
Toto je
```

```
? Left(Null,4)
Null
```

Funkce Right() a Right\$() jsou obdobou funkcí Left() a Left\$() s tím, že vracejí specifikovaný počet znaků zprava:

```
strOut = Right$(strIn,Length)
```

kde strIn je vstupní řetězec a Length udává počet znaků zprava, které má funkce vrátit. Jinak platí pravidla stejná jako u funkce Left\$().

```
strIn = "Toto je test"
```

```
? Right$(strIn,4)
Test
```

```
? Right(strIn,Len(strIn)-InStrRev(strIn," "))
Test
```

```
? Right$(strIn,Len(strIn)-InStr(strIn," "))
je test
```

Funkce Mid

Funkce Mid() a její řetězcová varianta jsou flexibilnější než funkce Left(), neboť mohou vrátit libovolný počet znaků počínaje libovolným znakem.

```
strOut = Mid$(strIn,Start[,Length])
```

Argument Length je u této funkce nepovinný. Pokud chybí, nebo pokud jeho velikost přesahuje délku řetězce, která zbývá od pozice Start, je vrácen celý zbytek řetězce počínaje pozicí Start, jak je vidět z posledního příkladu:

```
strIn = "Toto je test"
```

```
? Mid$(strIn,6,2)
je
```

```
? Mid$(strIn,Instr(strIn,"")+1,InstrRev(strIn," ")-Instr(strIn," ")-1)
je
```

```
? Mid$(strIn,InstrRev(strIn,"")+1)
test
```

Funkce `Mid()` je cenným pomocníkem, pokud potřebujeme iterovat přes všechny znaky v řetězci. Například následující funkce vrátí `True`, pokud vstupní data neobsahují ani jeden numerický znak:



```
Function NoNumbers(strIn As String) As Boolean
    Dim i As Long
    NoNumbers = True
    For i = 1 To Len(strIn)
        If Mid$(strIn, i, 1) Like "[0-9]" Then
            NoNumbers = False
            Exit For
        End If
    Next i
End Function
```

Použití vlastní funkce `NoNumbers()` je jednoduché:

```
? NoNumbers(123)
False
```

```
? NoNumbers("123")
False
```

```
? NoNumbers("A1?")
False
```

```
? NoNumbers("Abc?")
True
```

```
? NoNumbers("Toto je test")
True
```

Pokud chceme zjistit, jestli je řetězec alfabetycký (obsahující pouze písmena), pro většinu jazyků včetně češtiny je riskantní jednoduše kontrolovat všechny znaky pomocí operátoru `Like`:

```
? "ú" Like "[a-z]"
False
```

```
? "ú" Like "[a-ž]"
True
```

Lepší způsob je použít API funkci `Windows`, `IsCharAlphaA()`, která vezme do úvahy lokální jazyk a určí, jestli je znak písmeno:

```
Private Declare Function IsCharAlphaA Lib "USER32" _
    (ByVal bytChar As Byte) As Long
```

Deklarace této funkce musí být uvedena v záhlaví modulu. Jedno z možných použití je na obrázku.

Toto je pouze náhled elektronické knihy. Zakoupení její plné verze je možné v elektronickém obchodě společnosti eReading.