Kapitola 3. Pole

Touto kapitolou se vracíme k tématu proměnných. Proměnné, jak víte, slouží v programech k uchovávání nejrůznějších údajů. Proměnné mohou být lokální nebo členské. Každou proměnnou je třeba deklarovat v nějakém datovém typu, který popisuje druh hodnot do proměnné ukládaných.

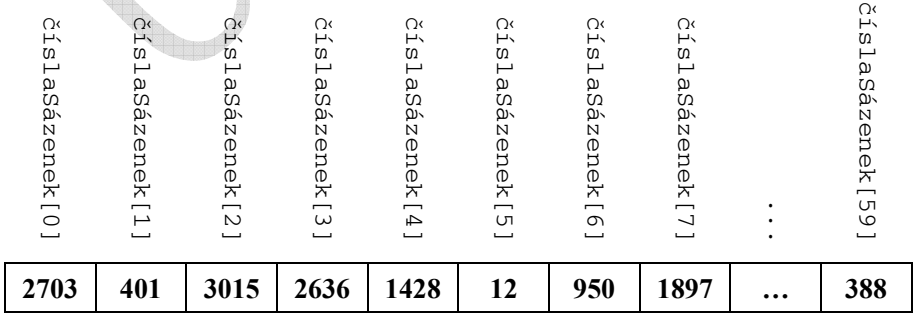
V našich dosavadních programech bylo proměnných vždy jen několik málo. Pamatovali jsme si třeba jednu barvu, jeden obrázek, dvě souřadnice ap. Řada aplikací však klade náročnější požadavky na paměť. Kupříkladu je třeba uložit desítky vylosovaných čísel, stovky dílčích obrazců tvořících kresbu, tisíce jmen zákazníků ap. A to vše v operační paměti (za účelem okamžitého přístupu), nikoli v souboru na disku.

Pro aplikace uvedeného druhu je právě určena datová struktura pole, s níž se v této kapitole seznámíme. Oproti většímu počtu samostatně deklarovaných proměnných má tyto výhody:

- Jediným příkazem lze vyhradit paměť pro uložení třeba tisíců jednotlivých údajů;
- K údajům lze snadno přistupovat na základě jejich pořadových čísel;
- Všechny uložené údaje lze snadno pomocí cyklu jednotným způsobem zpracovat.

DEFINICE POLE

Pole je datová struktura skládající se ze složek stejného typu, jež se navzájem rozlišují pomocí indexu. Toliko definice, podívejme se, co to znamená prakticky. Příklady polí ukazují Obr. 15, Obr. 16. Můžete vidět, že pole je takový řádek, příp. sloupec[1] většího počtu paměťových buněk, přičemž v každé buňce je uložena jedna hodnota – jedno číslo, jeden textový řetězec atd.



Obr. 15 Pole s čísly vylosovaných sázenek

[1] Záleží, jak se vám to lépe představuje.

`zaměstnanci[0]`	**Kolář Miloš**
`zaměstnanci[1]`	**Žváček Jindřich**
`zaměstnanci[2]`	**Černá Bohdana**
`zaměstnanci[3]`	**Bledý Norbert**
`zaměstnanci[4]`	**Karlická Markéta**
`zaměstnanci[5]`	**Mařík Zdislav**
`zaměstnanci[6]`	**Zoubek Čestmír**
`zaměstnanci[7]`	**Křídová Zuzana**
`...`	**…**
`zaměstnanci[213]`	**Vojáčková Sofie**

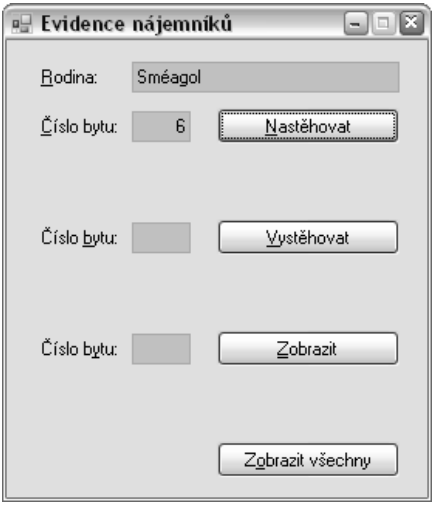
Obr. 16 Pole se jmény zaměstnanců

Pole z Obr. 15 se jmenuje `číslaSázenek` a obsahuje 60 buněk pro uložení 60 čísel. Pole z Obr. 16 se jmenuje `zaměstnanci` a obsahuje 214 paměťových buněk pro uložení 214 textových řetězců. S jednotlivými buňkami daného pole lze manipulovat nezávisle na sobě. Pro odkaz na konkrétní buňku se vždy uvede název celého pole a za ním v hranatých závorkách tzv. ***index***, což je pořadové číslo buňky v poli. Na rozdíl od běžného života, kde se čísluje od jedničky, začínají indexy v jazyce C# od nuly. Index poslední buňky je pak, jak je vám asi jasné, roven počtu buněk minus jedna.

ÚLOHA EVIDENCE NÁJEMNÍKŮ

Základy práce s polem si ukážeme na programu, který bude evidovat nájemníky v činžovním domě se 13 byty (v přízemí byt č. 0, v prvním patře byty č. 1, 2, 3, …), jak jej ukazuje Obr. 17. Program bude poskytovat tyto služby:

- Nastěhování – provede se záznam, že daná rodina, příp. jednotlivec bydlí v bytě číslo to a to;
- Vystěhování – byt se specifikovaným číslem se označí jako prázdný;
- Výpis nájemníků – zobrazí se nájemník daného bytu, příp. všech bytů v domě.



Obr. 17 Program *Evidence nájemníků*

S ohledem na to, že zatím neumíme pracovat se soubory, bude se veškerá evidence provádět zatím pouze v operační paměti. Za účelem následných referencí poznamenám, že jednotlivé aktivní ovládací prvky shora dolů pojmenuji takto: `poleRodina`, `poleČísloNastěhovat`, `tlačítkoNastěhovat`, `poleČísloVystěhovat`, `tlačítkoVystěhovat`, `poleČísloZobrazit`, `tlačítkoZobrazit`, `tlačítkoZobrazitVšechny`.

Když budete nad programem chvíli přemýšlet, asi dojdete k tomu, že ve své podstatě jde o to, zapamatovat si třináct textových řetězců. S prostředky, se kterými jsme dosud pracovali, by se pak podrobnosti řešily nějak takto:

1. Deklarovali bychom třináct samostatných proměnných, například:

```
string nájemník0, nájemník1, nájemník2, nájemník3, ...
```

2. Při nastěhování nájemníka např. do bytu č. 6 bychom jeho jméno zapsali do příslušné proměnné:

```
nájemník6 = "Sméagol";
```

3. Při vystěhování nájemníka např. z bytu č. 6 bychom do příslušné proměnné přiřadili hodnotu `null`:

```
nájemník6 = null;
```

První bod by byl sice trochu zdlouhavý, ale pokud je bytů třináct, tak ještě dejme tomu. Problém ale nastává při nastěhování a vystěhování. Číslo bytu totiž dopředu neznáme. Číslo bytu zadá až uživatel v textovém poli. Nastěhování by se tedy vlastně muselo řešit nějak tak, jak ukazuje Výpis 13, a to už tedy moc pěkné není[1]. Stejně tak s vystěhováním.

Šikovné řešení těchto problémů nabízí pole. Pojďme se na něj podívat.

```
private void tlačítkoNastěhovat_Click(...)
{
  string rodina = poleRodina.Text;
  int čísloBytu = Convert.ToInt32(poleČísloNastěhovat.Text);
  switch (čísloBytu)
  {
     case 0:
        nájemník0 = rodina;
        break;
     case 1:
        nájemník1 = rodina;
        break;
     case 2:
        nájemník2 = rodina;
        break;
     ... a tak dále ...
```

Výpis 13 Nastěhování při třinácti samostatných proměnných

[1] Možná si vzpomenete, že tento postup jsem použil v programu *Ruleta* v závěrečné kapitole učebnice pro začátečníky. Z důvodu, že to není úplně ono, jsem původní hru s čísly 0–36 omezil na rozsah 0–6.

ŘEŠENÍ S POLEM

Deklarace (postavení domu): V úvodu kapitoly bylo řečeno, že s použitím pole lze jediným příkazem vyhradit paměť pro větší množství údajů. V programu *Evidence nájemníků* se to provede takto:

```
string[] nájemníci = new string[13];
```

Jedná se vlastně o běžnou deklaraci proměnné s inicializací. V části nalevo od rovnítka říkáme, že budeme používat proměnnou `nájemníci` typu **pole řetězců.**[1] V části napravo se jako počáteční hodnota této proměnné vytvoří nové pole se 13 paměťovými buňkami pro uložení 13 textových řetězců.

Práce s hodnotami (první nájemníci): Při vytvoření pole je každá jeho buňka naplněna nedefinovanou hodnotou `null`, jak ukazuje Obr. 18.[2] Po vytvoření lze s každou buňkou samostatně pracovat. Například můžeme do některých bytů přiřadit první nájemníky, jak ukazuje Výpis 14 a Obr. 19.

nájemníci[0]	**null**
nájemníci[1]	**null**
nájemníci[2]	**null**
nájemníci[3]	**null**
nájemníci[4]	**null**
nájemníci[5]	**null**
nájemníci[6]	**null**
nájemníci[7]	**null**
nájemníci[8]	**null**
nájemníci[9]	**null**
nájemníci[10]	**null**
nájemníci[11]	**null**
nájemníci[12]	**null**

Obr. 18 Pole ihned po vytvoření

nájemníci[0]	**Křepelkovi**
nájemníci[1]	**Pytlík F.**
nájemníci[2]	**null**
nájemníci[3]	**null**
nájemníci[4]	**Bral**
nájemníci[5]	**Pytlík B.**
nájemníci[6]	**null**
nájemníci[7]	**null**
nájemníci[8]	**Brandorádovi**
nájemníci[9]	**null**
nájemníci[10]	**null**
nájemníci[11]	**Aragornovi**
nájemníci[12]	**null**

Obr. 19 Stav po prvních přiřazeních

```
private void oknoProgramu_Load(...)
{
   nájemníci[ 0] = "Křepelkovi";
   nájemníci[ 1] = "Pytlík F.";
   nájemníci[ 4] = "Bral";
   nájemníci[ 5] = "Pytlík B.";
   nájemníci[ 8] = "Brandorádovi";
   nájemníci[11] = "Aragornovi";
}
```

Výpis 14 Nastěhování prvních nájemníků

[1] Všimněte si páru hranatých závorek za slovem `string`.

[2] Pokud by šlo o pole čísel, byly by buňky na začátku zaplněny nulami.

Index jako proměnná (nastěhování, vystěhování, zobrazení): U výchozích nájemníků jsme si indexy neboli pořadová čísla buněk pole stanovili dopředu a bylo je tudíž možno do programu napevno uvést. V typičtějším případě však index dopředu neznáme, např. nevíme, do jakého bytu uživatel nastěhuje nového nájemníka. O tom už řeč byla výše. V případě pole to ale vůbec nevadí, **indexem** totiž **může být proměnná**. Do ní si nejprve uložíme číslo bytu z textového pole a následně ji použijeme pro přístup ke konkrétní buňce. Nastěhování je pak snadné, jak ukazuje Výpis 15.

```
private void tlačítkoNastěhovat_Click(...)
{
   int čísloBytu = Convert.ToInt32(poleČísloNastěhovat.Text);
   nájemníci[čísloBytu] = poleRodina.Text;
}
```

Výpis 15 Nastěhování s použitím pole

Toto je naprosto klíčové místo programu, kde se v maximální míře projevuje síla polí. Schválně, porovnejte toto řešení s nemotorným řešením bez pole, které naznačil Výpis 13. Použití proměnné jakožto indexu je ta nejdůležitější věc okolo polí, je proto třeba, abyste mu důkladně porozuměli. Představujte si, jak:

1. Uživatel zadává v textovém poli šestku;
2. Ta se převádí do číselné podoby a ukládá do proměnné `čísloBytu`;
3. Pročež se sahá na buňku s pořadovým číslem daným hodnotou této proměnné, neboli na buňku č. 6;
4. A v této a jen v této buňce se mění hodnota.

PRŮCHOD POLEM

Zatímco operace vystěhování, resp. zobrazení nájemníka zadaného bytu se provedou analogicky k nastěhování, je otázka zobrazení všech nájemníků něco jiného. Dá se říct, že jde o velmi typickou úlohu výpisu celého pole. **Úloha** výpisu (v obecném případě jakéhokoli jiného **zpracování**) **celého pole se vždy řeší postupným** výpisem (**zpracováním**) **všech jednotlivých buněk pole**.

S využitím indexace proměnnou lze tento výpis, resp. zpracování provést snadno pomocí cyklu. Cykly procházející všechny buňky pole vypadají tak, že se jejich řídicí proměnná mění od nuly až do počtu prvků pole minus jedna (pořadové číslo poslední buňky) a tato řídicí proměnná slouží jako index do procházeného pole. V programu *Evidence nájemníků* tento cyklus ukazuje Výpis 16. Ve výpisu si také

```
private void tlačítkoZobrazitVšechny_Click(...)
{
   string zpráva = "";
   for (int čísloBytu = 0; čísloBytu < nájemníci.Length; čísloBytu++)
       zpráva += čísloBytu.ToString() + ": " +
                 nájemníci[čísloBytu] + Environment.NewLine;
   MessageBox.Show(zpráva);
}
```

Výpis 16 Průchod polem v programu *Evidence nájemníků*

všimněte, že počet prvků pole se lze kdykoli dovědět od pole samotného dotazem na hodnotu jeho vlastnosti `Length`.

Abyste si činnost kódu z výpisu ještě lépe představili, naznačím, jak vlastně počítač celý cyklus otrocky provede (Výpis 17). Můžete vidět, že pro sestavení zprávy je použit princip postupného střádání mezivýsledku, jak jsem jej vysvětloval při výpočtu faktoriálu v jedenácté kapitole učebnice pro začátečníky.

```
private void tlačítkoZobrazitVšechny_Click(...)
{
   string zpráva = "";
   zpráva += "0: " + nájemníci[0] + Environment.NewLine;
   zpráva += "1: " + nájemníci[1] + Environment.NewLine;
   zpráva += "2: " + nájemníci[2] + Environment.NewLine;
   zpráva += "3: " + nájemníci[3] + Environment.NewLine;
     ... a tak dále ...
   zpráva += "12: " + nájemníci[12] + Environment.NewLine;
   MessageBox.Show(zpráva);
}
```

Výpis 17 Představa, jak počítač vykoná cyklus průchodu polem

Tolik vše k programu *Evidence nájemníků*, kompletní projekt vás čeká na webu.

POLE JAKO KONTEJNER VÝSTUPNÍCH DAT

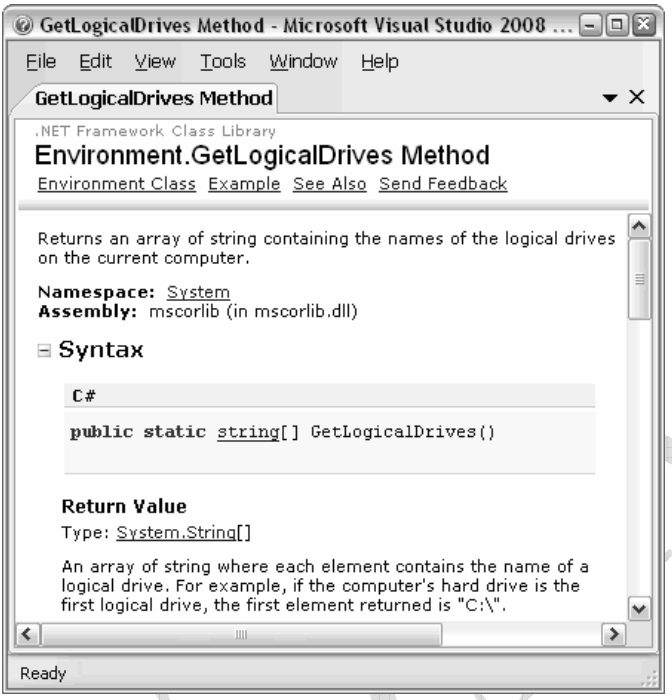
Přestože program *Evidence nájemníků* ukázal všechny základní postupy práce s poli, pro důkladnější pochopení bude dobré, když se na věc podíváme ještě z jiných úhlů. Zajímavé je například všimnout si, že různá pole slouží jako kontejnery pro uložení vstupních a výstupních dat metod knihoven platformy .NET. Soustřeďme se nejprve na jednodušší případ výstupních dat.

Za tím účelem si připomeňme pojem návratové hodnoty metody. Jak víte, kromě metod, jejichž hlavním smyslem je provedení nějaké akce (např. `Close`, `Refresh` ap.), existují metody, jejichž smyslem je stanovení nějaké hodnoty (např. `Sin`, `ToInt32` ap.). Tato tzv. návratová hodnota se po provedení metody jakoby dosadí za její volání. Chápe se tedy jako výsledek tohoto volání.

Výsledkem metod, které jsme doposud používali, byla pokaždé jedna hodnota – jedno číslo, jeden řetězec, jedna barva, … Často je však potřeba, aby metoda vrátila několik hodnot stejného druhu. Ideálním řešením je zabalit všechny tyto hodnoty do pole. Formálně tedy bude mít metoda stále jednu návratovou hodnotu, jak je třeba, ale tou návratovou hodnotou bude pole, které v sobě může obsahovat libovolné množství buněk stejného datového typu[1].

Příkladem může sloužit statická metoda `GetLogicalDrives` třídy `Environment`. Tato metoda umí zjistit, všechny diskové jednotky počítače. Výsledky svého zjištění předává, jak jinak, jako svou návratovou hodnotu. Jelikož ale diskových jednotek bývá obvykle na počítači více, je tato návratová hodnota navržena jako pole textových řetězců (Obr. 20).

[1] Pokud jsou výsledkem hodnoty různých typů, použije se pro jejich agregaci vlastní objekt. S vytvářením vlastních objektů se začnete seznamovat v osmé kapitole.



Obr. 20 Dokumentace k metodě `GetLogicalDrives`

K praktické demonstraci si připravte jednoduchý programek s tlačítkem spouštějícím akci a textovým polem zobrazujícím získané informace (Obr. 21). Ovládací prvky pojmenujte `tlačítkoZjisti` a `poleVýsledek`. Vložte kostru obslužné metody tlačítka a její tělo upravte do podoby, již zachycuje Výpis 18.

Jak vidíte, výsledek volání `GetLogicalDrives` se v programu ukládá do proměnné `jednotky`, deklarované jako pole řetězců. Toto pole je následně procházeno cyklem řízeným proměnnou `i`, která představuje index do pole. V cyklu se postupně sahá na `jednotky[0]`, `jednotky[1]`, `jednotky[2]`, … a z hodnot v těchto buňkách se sestavuje informační text pro uživatele.

Obr. 21 Test metody `GetLogicalDrives`

```
private void tlačítkoZjisti_Click(...)
{
    string[] jednotky = Environment.GetLogicalDrives();
    string zpráva = "";
    for (int i = 0; i < jednotky.Length; i++)
        zpráva += jednotky[i] + Environment.NewLine;
    poleVýsledek.Text = zpráva;
}
```

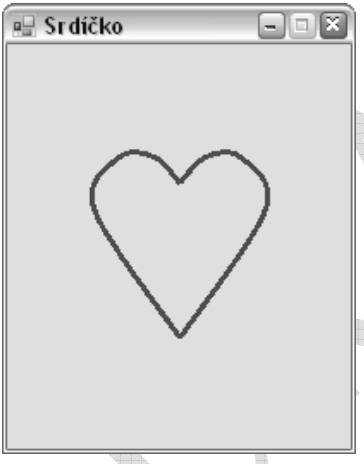
Výpis 18 Testování metody `GetLogicalDrives`

POLE JAKO KONTEJNER VSTUPNÍCH DAT

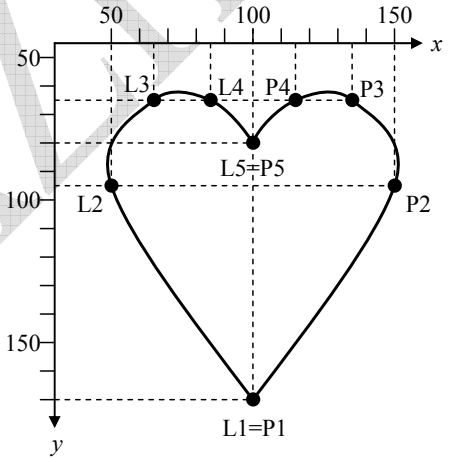
Věnujme se nyní druhému případu, a to použití pole pro specifikaci vstupních dat metody. Jak víte, vstupní data se metodám předávají jako parametry. Ve všech našich dosavadních programech byla sada parametrů (jejich počet a typy) vždy pevně dána. Kupříkladu při kreslení úsečky metodou `DrawLine` jsme pokaždé předávali pět parametrů – pero a dvakrát *x*, *y*, jednou pro počáteční, jednou pro koncový bod úsečky.

Existují však situace, kdy je třeba předat někdy více, někdy méně parametrů. Příkladem budiž metoda `DrawCurve` sloužící ke kreslení křivek. Zatímco úsečka je vždy dána svými dvěma koncovými body, bývají křivky určovány obecně různým počtem bodů. Jednodušší křivka může být dána menším počtem bodů nežli křivka komplikovanější. Jako řešení této situace přebírá metoda `DrawCurve` své definiční body formou pole. Pro každou křivku se tak může formálně předat pokaždé stejný počet parametrů, konkrétně dva, pero a pole definičních bodů.

Prakticky si předávání parametrů metodě `DrawCurve` v poli ukážeme na programu, který kreslí srdíčko ze dvou souměrně umístěných křivek (Obr. 22). Než tento program budeme moci napsat, je třeba obrazce zakreslit do souřadné sítě a odečíst jednotlivé body. Já jsem tak učinil na Obr. 23 a v Tab. 5, Tab. 6.



Obr. 22 Program kreslící křivky



Obr. 23 Geometrie křivek srdíčka

Název bodu	L1	L2	L3	L4	L5
Souřadnice	[100, 170]	[50, 95]	[65, 65]	[85, 65]	[100, 80]

Tab. 5 Definiční body levé křivky

Název bodu	P1	P2	P3	P4	P5
Souřadnice	[100, 170]	[150, 95]	[135, 65]	[115, 65]	[100, 80]

Tab. 6 Definiční body pravé křivky

Ve *Visual C#* nyní vytvořte nový projekt, pomocí Editoru vlastností a událostí vložte obslužnou metodu události `Paint` okna programu a tuto metodu upravte do podoby, kterou ukazuje Výpis 19.

Pro reprezentaci každého bodu je použit jeden objekt `Point`, který v sobě sdružuje údaje o *x*-ové a *y*-ové souřadnici. Objekt se nejlépe vytvoří konstruktorem, kterému se *x* a *y* předají jako dva parametry. Na tyto souřadnice se lze kdykoli později odvolat odkazem na vlastnosti `X`, `Y` objektu `Point`.

Po vytvoření všech bodů se vždy pět a pět sdruží do polí `Point[]`, která jsou pak předávána metodě `DrawCurve`. Pro vytvoření těchto polí jsem oproti programu *Evidence nájemníků* použil odlišnou syntaxi. V hranatých závorkách za `new Point` jsem neuvedl počet prvků. Namísto toho jsem za těmito hranatými závorkami uvedl ve složených závorkách **počáteční hodnoty buněk pole**. Zápis

```
new Point[] { L1, L2, L3, L4, L5 }
```

tak představuje pokyn k vytvoření pole s pěti buňkami[1] pro uložení pěti objektů `Point`, přičemž tyto buňky nebudou při vytvoření plněny hodnotami `null` jako na Obr. 18, nýbrž připravenými objekty `L1`, `L2`, `L3`, `L4` a `L5`.

```
private void oknoProgramu_Paint(...)
{
   Graphics kp = e.Graphics;

   // Body pro levou část srdíčka
   Point L1 = new Point(100, 170);
   Point L2 = new Point( 50,  95);
   Point L3 = new Point( 65,  65);
   Point L4 = new Point( 85,  65);
   Point L5 = new Point(100,  80);

   // Body pro pravou část srdíčka
   Point P1 = L1;
   Point P2 = new Point(150,  95);
   Point P3 = new Point(135,  65);
   Point P4 = new Point(115,  65);
   Point P5 = L5;

   // Sestavení polí bodů pro obě křivky
   Point[] poleLBodů = new Point[] { L1, L2, L3, L4, L5 };
   Point[] polePBodů = new Point[] { P1, P2, P3, P4, P5 };

   // A konečně vykreslení obou křivek
   Pen peroProKřivku = new Pen(Color.Red, 3);
   kp.DrawCurve(peroProKřivku, poleLBodů);
   kp.DrawCurve(peroProKřivku, polePBodů);
}
```

Výpis 19 Kreslení srdíčka dvěma křivkami

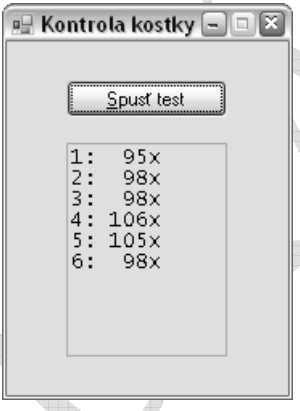
[1] Počet buněk si počítač sám zjistí podle toho, kolik je zadáno výchozích hodnot.

TABULKA ČETNOSTÍ JAKO APLIKACE POLE

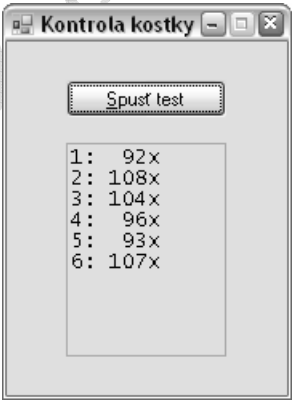
Velice pěkné použití nachází pole v úloze sestavení tabulky četností. S tabulkami četností se setkáte např. v novinách, když se dočtete, že tolik a tolik procent lidí má plat do 10 000,– Kč, tolik a tolik procent mezi 10 a 15 tisíci Kč, tolik a tolik procent v rozmezí 15 až 20 tisíc Kč atd. Nebo se dočtete v knížce, kolik bylo za posledních 50 let zemětřesení o síle 3–4 stupně, kolik bylo zemětřesení o síle 4–5 stupňů, kolik o síle 5–6 stupňů a tak dále. Tabulka četností zkrátka udává počty výskytů hodnot či událostí rozčleněných do řady kategorií.

V programu se úloha sestavení tabulky četností řeší jako vícenásobná úloha počítání. Každá kategorie hodnot má své počitadlo, které je na začátku vynulováno a které je zvětšeno o jedničku pokaždé, když hodnota na vstupu padne do příslušné kategorie. Právě pro uložení počitadel všech kategorií se použije pole.

Prakticky si úlohu sestavení tabulky četností ukážeme na programu, který bude kontrolovat férovost kostky. V učebnici pro začátečníky jsme vytvářeli program, jenž simuloval hrací kostku generováním náhodného čísla z rozmezí 1–6. Je-li tato kostka férová, musí každé z čísel 1–6 padat přibližně stejně často. Připravíme tedy program, který 600× hodí kostkou a na závěr řekne, kolikrát padla jednička, kolikrát dvojka atd. (Obr. 24). Zhruba by každé číslo mělo padnout 100×, ale samozřejmě náhoda je náhoda, většinou to přesná stovka nebude, spíš stovka plus mínus něco (porovnejte Obr. 24, Obr. 25).



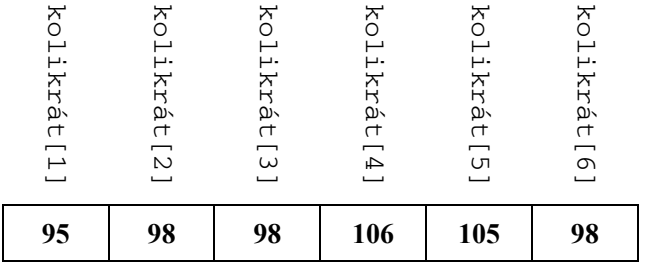
Obr. 24 Program, který 600× hodí kostkou



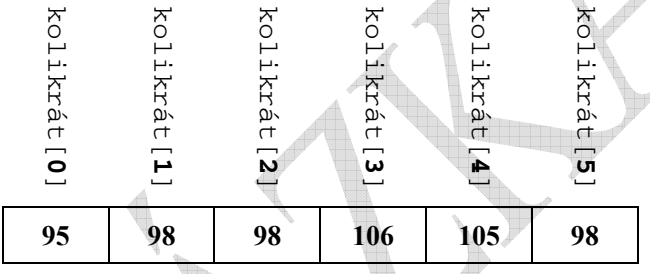
Obr. 25 Opakovaný běh testu

Jak tedy bylo řečeno, vytvoříme pole počitadel pro všechna možná hozená čísla. Pole můžeme nazvat `kolikrát` a bude vypadat podobně jako na Obr. 26. V obrázku je však určitý rozpor v tom, že zobrazené pole začíná na indexu 1, třebaže, jak bylo řečeno výše, pole v C# začínají nulou. Tento rozpor lze řešit dvěma způsoby:

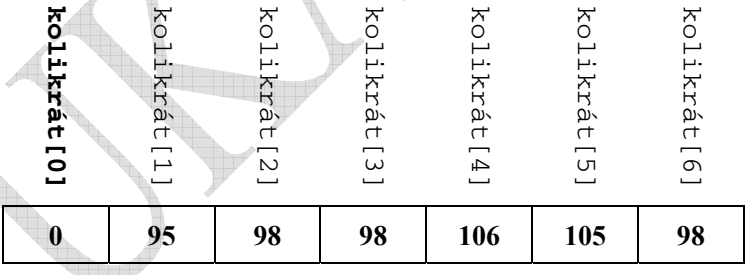
1. Posunout indexy o jedničku, neboli pole indexovat `[0]` až `[5]` a když padne např. čtyřka, připočíst jedničku ne do `kolikrát[4]`, ale do `kolikrát[3]` (Obr. 27);
2. Přidat nepoužívanou buňku `[0]` (Obr. 28).



Obr. 26 Pole počitadel pro čísla 1–6



Obr. 27 Posunutí indexu o 1



Obr. 28 Buňka navíc

Jelikož přidání jediné zbytečné buňky není zas takové plýtvání, přijde mi lepší druhá varianta. To posouvání indexu je totiž takové trochu nepřirozené, ve všech situacích na to člověk musí myslet, snadno udělá chybu. Zvolíme tedy řešení podle Obr. 28.

Jak se s počitadly bude pracovat? Pokud padne např. čtyřka, měli bychom přistoupit k buňce `kolikrát[4]` a zvětšit jí o jedničku. Připomenete-li si stěhování nájemníků v prvním programu této kapitoly, jistě vás napadne řešení: Číslo, které padne, uložíme do proměnné a tato proměnná následně poslouží jako index do pole.

Princip řešení jsme si vysvětlili. Ve *Visual C#* vytvořte nový projekt, do okna přetáhněte tlačítko a textové pole a nazvěte je `tlačítkoSpusťTest` a `poleVýsledky`. Vložte obslužnou metodu tlačítka a její tělo upravte do podoby, kterou ukazuje Výpis 20.

K výpisu učiním ještě poznámku ohledně volání `PadLeft(3)` – to zleva doplní mezery tak, aby řetězec měl vždy tři znaky. Výsledky budou pak zarovnány pěkně pod sebou bez ohledu na to, jsou-li dvojciferné nebo trojciferné. Samozřejmě, to vše pouze pokud ve vlastnosti `Font` textového pole nastavíte neproporcionální písmo, např. „Courier New“, „Lucida Console“ ap.

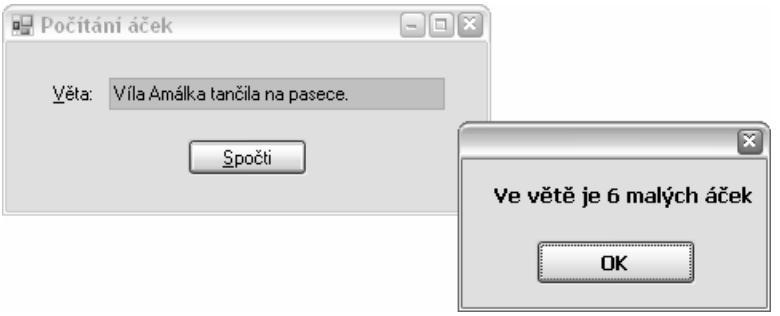
```
private void tlačítkoSpusťTest_Click(...)
{
   // Počitadla [0] až [6] ([0] se nepoužije)
   // Po vytvoření budou automaticky vynulována
   int[] kolikrát = new int[7];

   // Opakovaně házej kostkou a počítej, kolikrát co padlo
   int početHodů = 600;
   for (int čísloHodu = 1; čísloHodu <= početHodů; čísloHodu++)
   {
       int hozenéČíslo = náhoda.Next(1, 6+1);
       kolikrát[hozenéČíslo]++;
   }

   // Zobrazení výsledků
   string zpráva = null;
   for (int číslo = 1; číslo <= 6; číslo++)
   {
       zpráva += číslo.ToString() + ": " +
           kolikrát[číslo].ToString().PadLeft(3) + "x" +
           Environment.NewLine;
   }
   poleVýsledky.Text = zpráva;
}
```

Výpis 20 Kontrola kostky

ŘETĚZCE JAKO POLE ZNAKŮ

Tématiku polí zakončíme pohledem na textové řetězce. Na ty lze totiž za určitých okolností nahlížet jako na pole jednotlivých znaků. Můžete je pak procházet znak po znaku, vyhledávat v nich ap.

Pro ilustraci tohoto pohledu vytvoříme program, který bude počítat, kolik písmen „a“ se vyskytlo ve větě zadané uživatelem (Obr. 29). Pro jednoduchost si nebudeme všímat velkých „A“, ani „á“ s čárkou, jak můžete vidět ze zobrazeného výsledku.



Obr. 29 Program počítající výskyty malého „a“

Program je vcelku jednoduchý, proto rovnou uvedu zdrojový kód obslužné metody tlačítka. V něm upozorním na jedinou věc, a to použití jednoduchých apostrofů namísto dvojitých uvozovek při ohraničení písmene „a“. Hodnota `věta[čísloZnaku]` se chová jako znak, nikoli jako textový řetězec.[1] Musíme ji proto porovnávat se znakem, nikoli s textovým řetězcem. A znaky se, dle syntaxe jazyka C#, ohraničují apostrofy.

```
private void tlačítkoSpočti_Click(...)
{
    // Výchozí řetězec
    string věta = poleVěta.Text;
    int početZnaků = věta.Length;

    // Projdi všechny znaky řetězce a spočítej áčka
    int početÁček = 0;
    for (int číslo Znaku = 0; číslo Znaku < početZnaků; číslo Znaku++)
        if (věta[číslo Znaku] == 'a')
            početÁček++;

    // Zobraz výsledek
    MessageBox.Show("Ve větě je " + početÁček.ToString() +
                    " malých áček");
}
```

Výpis 21 Počítání výskytů písmene „a“

Na začátku tohoto oddílu jsem zmínil, že na řetězce lze nahlížet jako na pole jednotlivých znaků „za určitých okolností“. Těmito okolnostmi myslím především situaci, kdy chcete jednotlivé znaky číst. Nelze je totiž tímto postupem měnit. Pokud byste potřebovali znaky měnit, použijte instanci třídy `StringBuilder` nebo skutečné pole znaků `char[]`. O obou se můžete více dovědět v dokumentaci nebo v některých úlohách doprovodné sbírky.

[1] Řečeno přesně, jde o hodnotu typu `char` a nikoli o hodnotu typu `string`.

SHRNUTÍ KAPITOLY

- K aplikacím, kdy je třeba v paměti uložit větší množství dat stejného druhu, slouží datová struktura pole. Pole je řádek nebo sloupec buněk, které mají dohromady společné jméno a navzájem se rozlišují pořadovým číslem, tzv. indexem. Tento index v jazyce C# začíná na hodnotě 0.
- Při vytváření pole se specifikuje typ hodnot, které se budou ukládat do jednotlivých buněk. Například pole nazvané `nájemníci` s 13 buňkami pro uložení 13 textových řetězců se vytvoří příkazem:

```
string[] nájemníci = new string[13];
```

- Standardně jsou buňky pole při vytvoření naplněny hodnotami `null`, resp. nulami. Zaplnění jinými počátečními hodnotami se např. pro pole se třemi řetězci provede takto:

```
string[] kamarádi = new string[] {"Jirka", "Bára", "Kubrt"};
```

- K jednotlivým složkám pole se přistupuje uvedením jména pole, za nímž následuje index v hranatých závorkách. Index může být hodnotou proměnné, čehož se cíleně využívá v řadě programů.
- Častou úlohou je jednotné zpracování všech prvků pole. To se provádí vždy prvek za prvkem, například pomocí cyklu `for`. Za účelem zjištění počtu prvků lze s výhodou využít dotaz na hodnotu vlastnosti `Length` pole.
- Textové řetězce se chovají jako pole jednotlivých znaků bez možnosti zápisu (změny).