Kapitola 10

Používáme pole a kolekce

Po přečtení této kapitoly budete schopni:

- Deklarovat, inicializovat a používat proměnné typu `pole`
- Deklarovat, inicializovat a používat proměnné různých typů kolekcí

Nyní již víte, jak vytvářet a používat proměnné mnoha různých typů. Příklady proměnných, které jste až dosud mohli vidět, však měly jedno společné – obsahovaly informace o jedné položce (typu `int`, `float`, `Kruh`, `Datum` apod.). Co když ale potřebujete manipulovat najednou s celou sadou položek? Jedním z možných řešení je vytvoření zvláštní proměnné pro každou položku v sadě, ale to před nás staví spoustu dalších otázek: Kolik proměnných je potřeba? Jak je pojmenovat? Pokud je nutné provádět stejnou operaci se všemi položkami (například inkrementovat všechny celočíselné proměnné v sadě), jak se vyhnout nadměrnému opakování kódu? Toto řešení také předpokládá, že při psaní programu víte, kolik položek budete potřebovat – ovšem jak často to skutečně víte? Jestliže píšete například aplikaci načítající záznamy z databáze, které jsou postupně zpracovány, je nutné znát počet záznamů v databázi, který se navíc může měnit.

Pro manipulaci se sadou položek jsou v jazyku C# k dispozici pole a kolekce.

Co je pole

Pole je neuspořádaná posloupnost prvků. Všechny prvky v poli jsou stejného typu (na rozdíl od datových složek ve strukturách nebo třídách, kde jsou povoleny rozdílné typy). Prvky pole jsou uloženy v souvislém bloku paměti a pro přístup k nim slouží celočíselný index (u datových složek ve strukturách nebo třídách je to jejich název).

Deklarace proměnné typu pole

Proměnná typu `pole` se deklaruje názvem datového typu pro všechny prvky pole, za kterým následuje dvojice hranatých závorek, mezera, jméno proměnné a středník. Hranaté závorky signalizují, že proměnná představuje pole. Deklarace pole proměnných typu `int` pro uložení osobních identifikačních čísel tedy vypadá takto:

```
int[] piny;  // PIN = osobní identifikační číslo
```

Programátoři ve Visual Basicu si poznamenají, že v deklaracích polí se používají hranaté závorky, nikoli kulaté. Programátoři v C a C++ vezmou na vědomí, že velikost pole není součástí deklarace. Programátoři v Javě nesmí zapomínat na to, že hranaté závorky patří před název proměnné, ne za něj.

Poznámka: Datové typy prvků nejsou omezeny na primitivní typy, vytvářet lze i pole struktur, výčtů nebo tříd. Pro vytvoření pole struktur typu `Čas` byste mohli použít následující deklaraci:

```
Čas[] časy;
```

Tip: Často je užitečné dávat polím názvy v množném čísle, například `místa` (pole prvků typu `Místo`), `lidé` (pole prvků typu `Člověk`) nebo `časy` (pole prvků typu `Čas`).

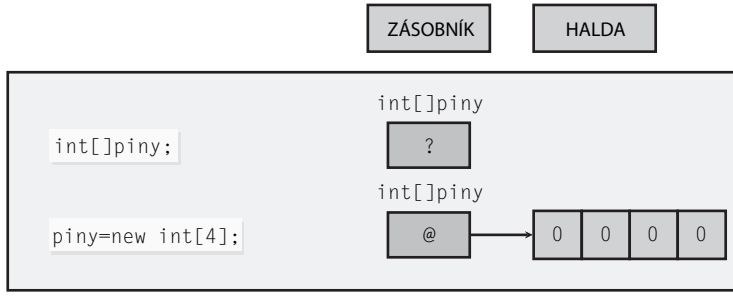
Vytvoření instance pole

Pole patří k referenčním typům nezávisle na tom, jakého typu jsou jeho prvky. To znamená, že proměnná typu `pole` odkazuje na souvislý blok v paměti uchovávající pole prvků v haldě, stejně jako instance třídy odkazuje na objekt v haldě, přičemž tento souvislý blok paměti neudržuje prvky pole na zásobníku, jako je tomu v případě struktur. (Hodnotové a referenční typy, včetně rozdílů mezi zásobníkem a haldou, najdete v kapitole 8.) Když deklarujete proměnnou typu `třída`, je paměť pro objekt přidělena až při vytvoření jeho instance klíčovým slovem `new`. Pole podléhají stejným pravidlům: při deklaraci proměnné typu `pole` neuvádíte její velikost, to se děje až při skutečném vytvoření instance daného pole.

Instance pole se vytváří pomocí klíčového slova `new`, za kterým následuje typ prvků a v hranatých závorkách velikost pole. Při vytvoření pole jsou také inicializovány jeho prvky (nám již známými) výchozími hodnotami, odpovídajícími zvolenému typu (`0`, `null` nebo `false`, v závislosti na tom, jde-li o typ číselný, referenční nebo logický). Vytvoření a inicializace pole čtyř celých čísel pro proměnnou `piny` z předchozí ukázky tedy vypadá takto:

```
piny = new int[4];
```

Výsledek tohoto příkazu vidíte na následujícím schématu:



Velikost instance pole nemusí být stanovena pomocí konstanty, dá se vypočítat za běhu, jak ukazuje další příklad:

```
int velikost = int.Parse(Console.ReadLine());
int[] piny = new int[velikost];
```

Je možné vytvořit pole s nulovou velikostí. Vypadá to sice jako dosti bizarní nápad, ale hodí se to v situacích, kdy je velikost pole určována dynamicky a může nabýt i nulové hodnoty. Pole o velikosti 0 není prázdné pole (tj. neodpovídá hodnotě `null`).

Inicializace proměnných typu pole

Když vytvoříte proměnnou typu `pole`, budou všechny prvky instance inicializovány výchozími hodnotami použitého typu. Toto implicitní chování se dá změnit a inicializovat prvky přesně zadanými hodnotami, které zapíšete mezi dvojici složených závorek jako seznam hodnot oddělených čárkami. Inicializace pole `piny` se čtyřmi proměnnými typu `int` hodnotami `9`, `3`, `7` a `2` tedy vypadá takto:

```
int[] piny = new int[4]{9, 3, 7, 2};
```

Hodnoty uvnitř složených závorek nemusí být zadávány konstantami, ale dají se vypočítat až za běhu programu:

```
Random r = new Random();

int[] piny = new int[4]{r.Next() % 10, r.Next() % 10,
                        r.Next() % 10, r.Next() % 10};
```

Poznámka: Třída `System.Random` je generátor pseudonáhodných čísel. Metoda `Next` standardně vrátí náhodné nezáporné celé číslo v rozsahu `0` až `Int32.MaxValue`. Metoda `Next` je přetížená, přičemž další verze umožňují zadávat minimální a maximální hodnoty požadovaného rozsahu. Implicitní konstruktor třídy `Random` zasadí do generátoru náhodných čísel časově závislou hodnotu semínka, což snižuje možnost vygenerování duplicitní posloupnosti náhodných čísel. Díky přetížené verzi konstruktoru můžete zadat vlastní hodnotu semínka. Tímto způsobem můžete generovat opakovatelné posloupnosti náhodných čísel pro účely testování.

Počet hodnot mezi složenými závorkami musí přesně odpovídat velikosti vytvářené instance pole:

```
int[] piny = new int[3]{9, 3, 7, 2};   // chyba při kompilaci
int[] piny = new int[4]{9, 3, 7};      // chyba při kompilaci
int[] piny = new int[4]{9, 3, 7, 2};   // v pořádku
```

Při inicializaci proměnné typu `pole` explicitně zadanými hodnotami je ve skutečnosti možné vynechat klíčové slovo `new` a velikost pole. Kompilátor v takovém případě vypočítá velikost pole z počtu inicializačních prvků a vygeneruje kód pro vytvoření pole:

```
int[] piny = {9, 3, 7, 2};
```

Při vytváření pole struktur lze každou strukturu v poli inicializovat voláním jejího konstruktoru:

```
Čas rozvrh = {new Čas(12, 30), new Čas(5, 30)};
```

Tvorba implicitně typovaných polí

Typ prvku při deklaraci pole musí odpovídat typu prvků, které budete do pole ukládat. Když například deklarujete `piny` jako pole typu `int`, jak jsme si ukázali v předchozích příkladech, nemůžete do něj ukládat desetinná čísla, řetězce, struktury nebo cokoliv, co není typ `int`. Pokud při deklaraci pole uvedete seznam inicializačních prvků, můžete nechat kompilátor jazyka C# odvodit skutečný typ prvků v tomto poli:

```
var jména = new[]{"Jan", "Katka", "Jakub", "Iveta"};
```

V tomto případě kompilátor jazyka C# stanoví, že proměnná jména je pole řetězců. Povšimněte si několika syntaktických zajímavostí ve výše uvedené deklaraci. Za prvé, u typu

jsme vynechali hranaté závorky, takže proměnnou `jména` nedeklarujeme jako `var[]`, ale jen jako `var`. Za druhé, před seznamem inicializačních prvků je nutné uvést operátor `new` a hranaté závorky.

Pokud tuto syntaxi použijete, musíte dbát na to, aby všechny inicializační prvky byly téhož typu. Následující příklad způsobí chybu při kompilaci:

```
var špatně = new[]{"Jan", "Katka", 99, 100};
```

Nicméně, v některých smysluplných případech dojde k tomu, že kompilátor převede prvky na jiný typ. V následujícím kódu je pole `čísla` polem typu `double`, protože konstanty `3.5` a `99.999` jsou typu `double` a protože kompilátor jazyka C# může převést celočíselné hodnoty `1` a `2` na typ `double`:

```
var čísla = new[]{1, 2, 3.5, 99.999};
```

Obecně vzato, nejlepší je typy vůbec nesměšovat a doufat, že kompilátor je za vás převede. Implicitně typovaná pole jsou nejužitečnější při práci s anonymními typy, které jsme si popsali v kapitole 7. Následující kód vytvoří pole anonymních objektů, z nichž každý obsahuje dvě datové složky specifikující jméno a věk členů nějaké rodiny:

```
var jména = new[] {new {Jméno = "Jan", Věk = 42},
                   new {Jméno = "Katka", Věk = 43},
                   new {Jméno = "Jakub", Věk = 15},
                   new {Jméno = "Iveta", Věk = 13}};
```

Datové položky v anonymních typech musí být stejné pro všechny prvky daného pole.

Přístup k jednotlivým prvkům pole

Pro přístup k jednotlivým prvkům pole je nutné uvést index vyjadřující, který prvek požadujete. Obsah prvku `2` z pole `piny` lze načíst do proměnné typu `int` následujícím příkazem:

```
int můjPin;
můjPin = piny[2];
```

Obdobně lze změnit obsah pole přiřazením hodnoty jeho indexovanému prvku:

```
můjPin = 1645;
piny[2] = můjPin;
```

Indexy pole začínají nulou. První prvek pole má tedy index 0, nikoli 1. Index 1 patří druhému prvku. Při každém přístupu k prvkům pole se kontroluje, zda hodnota indexu leží v mezích daného pole. Jestliže uvedete celočíselný index menší než nula nebo větší či rovný velikosti pole, vyvolá program výjimku typu `IndexOutOfRangeException` – jako v následujícím příkladu:

```
try
{
   int[] piny = {9, 3, 7, 2};
   Console.WriteLine(piny[4]); // chyba, 4. prvek má index 3
}
catch (IndexOutOfRangeException ex)
{
   ...
}
```

Procházení pole

Všechna pole jsou instancemi třídy `System.Array` z rozhraní Microsoft .NET Framework. Tato třída definuje množství užitečných vlastností a metod. Pomocí vlastnosti `Length` se například dá zjistit, kolik prvků dané pole obsahuje. Tuto vlastnost může programátor použít k procházení (iteraci) všemi prvky pole v cyklu `for`. Následující ukázkový kód vypíše do okna příkazového řádku hodnoty všech prvků pole `piny`:

```
int[] piny = {9, 3, 7, 2};
for (int index = 0; index < piny.Length; index++;)
{
   int pin = piny[index];
   Console.WriteLine(pin);

}
```

Poznámka: `Length` je vlastnost, nikoli metoda, a proto se při jejím volání nepíšou závorky. O vlastnostech se dozvíte více v kapitole 15.

Začínající programátoři často zapomínají, že index prvků pole začíná nulou a že číslo posledního prvku je `Length - 1`. Jazyk C# nabízí pro průchod prvky pole také příkaz `foreach`, díky němuž se těmto problémům snadno vyhneme. Předchozí příkaz `for` bychom mohli přepsat pomocí příkazu `foreach` takto:

```
int[] piny = {9, 3, 7, 2};
foreach (int pin in piny)
{
   Console.WriteLine(pin);
}
```

Příkaz `foreach` deklaruje iterační proměnnou (v tomto příkladu `int pin`), která automaticky nabývá v každém průchodu cyklem hodnoty dalšího prvku v poli. Typ této proměnné musí odpovídat typu prvků v procházeném poli. Konstrukce příkazu `foreach` přímo vyjadřuje, co chceme v kódu provádět. V některých případech je však příkaz `for` vhodnější nebo jediný možný:

- Příkaz `foreach` prochází vždy celé pole. Pokud potřebujete zpracovat jen určitou, známou, část pole (třeba první polovinu) nebo některé prvky vynechat (například každý třetí), je jednodušší použít příkaz `for`.
- Příkaz `foreach` vždy začíná iteraci od indexu `0` po index `Length - 1`. Pokud chcete pole projít opačným směrem nebo v jiné posloupnosti, musíte použít příkaz `for`.
- Pokud v těle cyklu musí kód znát aktuální hodnotu indexu, musíte použít příkaz `for`. V cyklu `foreach` je známa jen hodnota aktuálního prvku.
- Jestliže je nutné měnit hodnoty prvků pole, musíte použít příkaz `for`. Důvodem je to, že iterační proměnná v cyklu `foreach` je jen kopií každého prvku pole, kterou nelze měnit.

Iterační proměnnou můžete deklarovat také jako `var` a nechat kompilátor jazyka C#, aby odvodil typ této proměnné z typu prvků v procházeném poli. To je užitečné zvláště tehdy, pokud typ prvků pole ve skutečnosti neznáte, tedy například když toto pole obsahuje anonymní objekty. Následující příklad demonstruje, jak můžete procházet polem členů rodiny, které jsme si ukázali již dříve:

```
var jména = new[] {new {Jméno = "Jan", Věk = 42},
                   new {Jméno = "Katka", Věk = 43},
                   new {Jméno = "Jakub", Věk = 15},
                   new {Jméno = "Iveta", Věk = 13}};
foreach (var člen in jména)
{
   Console.WriteLine("Jméno: {0}, Věk: {1}", člen.Jméno, člen.Věk);
}
```

Kopírování polí

Pole patří mezi referenční typy (nezapomeňte, že pole je instancí třídy `System.Class`). Proměnná typu `pole` proto obsahuje odkaz na instanci pole. Když tedy zkopírujete proměnnou typu `pole`, obdržíte ve skutečnosti dva odkazy na stejnou instanci pole:

```
int[] piny = {9, 3, 7, 2};
int[] alias = piny; // alias a piny ukazují na stejnou instanci pole
```

V tomto případě platí, že po změně hodnoty `piny[1]` se změna projeví i při použití hodnoty `alias[1]`.

Pokud chcete fyzicky kopírovat instanci pole (data v haldě), na kterou ukazuje proměnná typu `pole`, musíte to udělat ve dvou krocích. Nejdříve je nutné vytvořit novou instanci pole stejného typu a stejné velikosti jako kopírované pole a poté zkopírovat data po jednotlivých prvcích z původního do nového pole:

```
int[] piny = {9, 3, 7, 2};
int[] kopie = new int[piny.Length];
for (int i = 0; i < kopie.Length; i++;)
{
   kopie[i] = piny[i];
}
```

Všimněte si, že v tomto kódu používáme pro stanovení velikosti nového pole vlastnost `Length` původního pole.

Kopírování pole je ve skutečnosti docela běžným požadavkem řady aplikací. Tak běžným, že třída `System.Array` byla vybavena několika užitečnými metodami, které lze použít ke kopírování polí místo psaní vlastního kódu. Například metoda `CopyTo` zkopíruje obsah jednoho pole do jiného, počínaje zadaným počátečním indexem:

```
int[] piny = {9, 3, 7, 2};
int[] kopie = new int[piny.Length];
piny.CopyTo(kopie, 0);
```

Další možností, jak kopírovat hodnoty pole, je volání statické metody `Copy` třídy `System.Array`. Obdobně jako u metody `CopyTo` musí být i zde cílové pole inicializováno před voláním metody:

```
int[] piny = {9, 3, 7, 2};
int[] kopie = new int[piny.Length];
Array.Copy(piny, kopie, kopie.Length);
```

Další alternativou je metoda instance třídy `System.Array` s názvem `Clone`, která vytvoří nové pole a zkopíruje do něj obsah zadaného pole pomocí jediného příkazu:

```
int[] piny = {9, 3, 7, 2};
int[] kopie = (int[])piny.Clone();
```

Poznámka: Metoda `Clone` ve skutečnosti vrátí proměnnou typu `object`, kterou je nutné přetypovat na pole odpovídajícího typu. Všechny tři metody dále vytvářejí *mělkou* kopii, což znamená, že pokud kopírované pole obsahuje reference, zkopírují se jen tyto odkazy, nikoli však samotné odkazované objekty. Po kopírování se obě pole odkazují na stejnou sadu objektů. Pokud potřebujete vytvořit *hloubkovou* kopii pole, musíte v cyklu `for` použít náležitý kód.

Vícerozměrná pole

Pole, která jsme si dosud ukázali, se skládají z jediného rozměru a lze na ně pohlížet jako na jednoduché seznamy hodnot. Můžete ovšem vytvářet také pole s více než jedním rozměrem. Kupříkladu pro vytvoření dvourozměrného pole stačí specifikovat pole vyžadující dva celočíselné indexy. Následující kód vytváří dvourozměrné pole obsahující 24 celých čísel s názvem `prvky`. Dvourozměrné pole si můžete představit jako tabulku, u níž první rozměr udává počet řádků a druhý rozměr počet sloupců.

```
int[,] prvky = new int[4, 6];
```

Pro přístup k prvkům tohoto pole musíte uvést dvě indexové hodnoty udávající „buňku“, která uchovává požadovaný prvek. (Buňka je průsečíkem řádku a sloupce.) Následující kód obsahuje několik příkladů použití pole `prvky`:

```
prvky[2, 3] = 99;          // nastav prvek v buňce(2,3) na 99
prvky[2, 4] = prvky [2,3]; // zkopíruje prvek v buňce(2, 3) do buňky(2, 4)
prvky[2, 4]++;             // inkrementuj celočíselnou hodnotu v buňce(2, 4)
```

Počet rozměrů vytvářeného pole není nijak omezen. V níže uvedeném kódu vytváříme a používáme pole s názvem `krychle`, které obsahuje tři rozměry. Všimněte si, že pro přístup k prvkům tohoto pole je nutné uvést tři indexy:

```
int[, ,] krychle = new int[5, 5, 5];
krychle[1, 2, 1] = 101;
krychle[1, 2, 2] = krychle[1, 2, 1] * 3;
```

V souvislosti s vytvářením polí s více než třemi rozměry je třeba upozornit na důležitou skutečnost. Pole totiž mohou být velmi náročná na paměť. Pole `krychle` obsahuje 125 prvků (5 * 5 * 5). Čtyřrozměrné pole, kde každý rozměr uchovává 5 prvků, obsahuje 625 prvků. Obecně lze říci, že při práci s vícerozměrnými poli byste měli být vždy připraveni na zachycení a obsloužení výjimky typu `OutOfMemoryException`.

Použití polí pro hraní karet

V následujícím cvičení použijete pole pro implementaci aplikace, která simuluje rozdávání hracích karet v rámci nějaké karetní hry. Tato aplikace zobrazí formulář WPF (Windows Presentation Foundation) ukazující čtyři ruce s kartami náhodně rozdanými z běžného balíčku (52 karet) hracích karet. Vaším úkolem bude dokončit kód, který rozdává karty do každé ruky.

Implementace karetní hry pomocí polí

1. Spusťte Microsoft Visual Studio 2010, pokud již neběží.
2. Otevřete projekt Karty umístěný ve složce *\Dokumenty\Visual CSharp 2010 Krok za krokem\Kapitola 10\Karty s použitím polí*.

3. V nabídce *Debug* klepněte na příkaz *Start Without Debugging*, čímž dojde k sestavení a spuštění aplikace.

   Objeví se formulář WPF se záhlavím *Karetní hra*, čtyřmi textovými poli (označenými jako *Sever*, *Jih*, *Východ* a *Západ*) a tlačítkem *Rozdej*.

4. Klepněte na tlačítko *Rozdej*.

   Objeví se okno se zprávou „RozdejKartuZBalíčku“, což znamená, že je kód rozdávající karty není dosud implementovaný.

5. Klepněte na tlačítko *OK*, poté zavřete okno *Karetní hra* a vraťte se do Visual Studia 2010.

6. V okně editoru zobrazte soubor *Hodnota.cs*.

   Tento soubor obsahuje výčet s názvem `Hodnota`, který představuje různé hodnoty karet seřazené vzestupně:

```
enum Hodnota {Dva, Tři, Čtyři, Pět, Šest, Sedm, Osm,
              Devět, Deset, Kluk, Dáma, Král, Eso}
```

7. Zobrazte v okně editoru soubor *Barva.cs*.

   Tento soubor obsahuje výčet s názvem `Barva`, který představuje barvu karet v běžném balíčku:

```
enum Barva {Kříže, Káry, Srdce, Piky}
```

8. Zobrazte v okně editoru soubor *HracíKarta.cs*.

   Tento soubor obsahuje třídu `HracíKarta`, která modeluje jednu hrací kartu.

```
class HracíKarta
{
   private readonly Barva barva;
   private readonly Hodnota hodnota;

   public HracíKarta(Barva s, Hodnota v)
   {
      this.barva = s;
      this.hodnota = v;
   }

   public override string ToString()
   {
      string výsledek = string.Format("{0} : {1}", this.hodnota,
                                                    this.barva);
      return výsledek;
   }

   public Barva BarvaKarty()
   {
      return this.barva;
   }

   public Hodnota HodnotaKarty()
   {
      return this.hodnota;
   }
}
```

Tato třída má dvě datové složky určené pouze pro čtení, které představují hodnotu a barvu karty. Tyto datové složky inicializujeme v konstruktoru.

Poznámka: Datová složka označená jako `readonly` (pouze pro čtení) je užitečná pro modelování dat, která by se po své inicializaci již neměla měnit. Do datové složky určené pouze pro čtení můžete přiřadit hodnotu inicializací při deklaraci nebo v konstruktoru, pak už ji ale nemůžete změnit.

Tato třída obsahuje dvojici metod nazvaných `HodnotaKarty` a `BarvaKarty`, které vracejí příslušnou informaci, a dále přepisuje metodu `ToString` tak, aby vracela řetězcovou reprezentaci katry.

Poznámka: Metody `HodnotaKarty` a `BarvaKarty` je ve skutečnosti lepší implementovat jako vlastnosti. Jak se to provádí, se dozvíte v kapitole 15.

9. V okně editoru otevřete soubor *Balíček.cs*.

Tento soubor obsahuje třídu `Balíček`, která modeluje balíček hracích karet. Na začátku třídy `Balíček` jsou dvě veřejné konstantní datové složky typu `int` s názvy `PočetBarev` a `KaretNaBarvu`. Tyto dvě datové složky určují počet barev v balíčku karet a počet karet v každé barvě. Soukromá proměnná `balíčekKaret` je dvourozměrné pole objektů typu `HracíKarta`. (První rozměr použijete pro stanovení barvy a druhý pro stanovení hodnoty karty v dané barvě.) Proměnná `náhodnýVýběrKarty` je objektem typu `Random`. Třída `Random` představuje generátor pseudonáhodných čísel, přičemž proměnnou `náhodnýVýběrKarty` použijete pro zamíchání karet před rozdáním do každé ruky.

```
class Balíček
{
   public const int PočetBarev = 4;
   public const int KaretNaBarvu = 13;
   private HracíKarta[,] balíčekKaret;
   private Random náhodnýVýběrKarty = new Random();
   ...
}
```

10. Vyhledejte výchozí konstruktor pro třídu `Balíček`. V současné chvíli je tento konstruktor kromě komentáře prázdný. Vymažte komentář a přidejte následující zvýrazněný příkaz, který vytvoří instanci pole `balíčekKaret` se správným počtem prvků:

```
public Balíček()
{
   this.balíčekKaret = new HracíKarta[PočetBarev, KaretNaBarvu];
}
```

11. Přidejte níže uvedený kód do konstruktoru třídy `Balíček`. Vnější cyklus prochází seznamem hodnot ve výčtu `Barva` a vnitřní cyklus prochází hodnoty, které může mít každá karta v každé barvě. Vnitřní cyklus vytváří nový objekt typu `HracíKarta` dané barvy a hodnoty a ukládá jej do příslušného prvku v poli `balíčekKaret`:

```
for (Barva barva = Barva.Kříže; barva <= Barva.Piky; barva++)
{
   for (Hodnota hodnota = Hodnota.Dva; hodnota <= Hodnota.Eso; hodnota++)
```

```
    {
        this.balíčekKaret[(int)barva, (int)hodnota] =
            new HracíKarta(barva, hodnota);
    }
}
```

Poznámka: Jako indexy do pole je nutné použít některý z celočíselných typů. Proměnné `barva` a `hodnota` jsou sice výčtového typu, avšak výčty jsou založeny na celočíselných typech, a proto je můžeme bezpečně přetypovat na typ `int`, jako ve výše uvedeném kódu.

12. Vyhledejte ve třídě `Balíček` metodu `RozdejKartuZBalíčku`. Účelem této metody je vybrat náhodnou kartu z balíčku, vydat ji a poté ji z balíčku odebrat, aby tak nedošlo k jejímu opětovnému vydání.

Prvním úkolem v této metodě je náhodně vybrat barvu. Vymažte z této metody komentář a příkaz, který vyvolává výjimku typu `NotImplementedException`, a nahraďte jej následujícím zvýrazněným kódem:

```
public HracíKarta RozdejKartuZBalíčku()
{
    Barva barva = (Barva)náhodnýVýběrKarty.Next(PočetBarev);
}
```

V tomto příkazu používáme metodu `Next` pseudonáhodného generátoru `náhodnýVýběrKarty` pro získání náhodného čísla odpovídajícího určité barvě. Parametr metody `Next` udává výlučnou horní hranici rozsahu, který se má použít. Vybíraná hodnota je tedy mezi 0 a touto hodnotou minus jedna. Všimněte si, že získaná hodnota je typu `int`, a proto ji musíme před přiřazením do proměnné typu `Barva` přetypovat.

Vždy existuje možnost, že v balíčku vybrané barvy již nejsou žádné další karty. Tuto situaci tedy musíme ošetřit, a je-li to nutné, vybrat jinou barvu.

13. Vyhledejte metodu `JeBarvaPrázdná`. Účelem této metody je vzít parametr typu `Barva` a vrátit logickou hodnotu signalizující, zda jsou v balíčku se zadanou barvou ještě nějaké další karty. Z metody vymažte komentář a příkaz, který vyvolává výjimku typu `NotImplementedException`, a přidejte následující zvýrazněný kód:

```
private bool JeBarvaPrázdná(Barva barva)
{
    bool výsledek = true;

    for (Hodnota hodnota = Hodnota.Dva; hodnota <= Hodnota.Eso; hodnota++)
    {
        if (!JeKartaJižRozdaná(barva, hodnota))
        {
            výsledek = false;
            break;
        }
    }

    return výsledek;
}
```

V tomto kódu procházíme všechny možné hodnoty karet a pomocí metody `JeKartaJižRozdaná`, kterou dokončíme v následujícím kroku, zjišťujeme, zda v poli `balíčekKaret` zůstala nějaká karta, která má danou barvu a hodnotu. Pokud v cyklu nalezneme kartu, nastavíme hodnotu proměnné `výsledek` na `false` a pomocí příkazu `break` cyklus přerušíme. Pokud se cyklus dokončí bez nalezení karty, zůstane proměnná `výsledek` nastavená na svoji původní hodnotu `true`. Hodnotu proměnné `výsledek` nakonec předáme zpět jako návratovou hodnotu metody.

14. Vyhledejte metodu `JeKartaJižRozdaná`. Účelem této metody je zjistit, zda karta se zadanou barvou a hodnotou již byla rozdána a odebrána z balíčku. Později uvidíte, že když metoda `RozdejKartuZBalíčku` rozdá kartu, odebere ji z pole `balíčekKaret` tak, že odpovídající prvek nastaví na hodnotu `null`. Nahraďte v této metodě komentář a příkaz, který vyvolává výjimku typu `NotImplementedException`, následujícím zvýrazněným kódem:

```
private bool JeKartaJižRozdaná(Barva barva, Hodnota hodnota)
{
   return (this.balíčekKaret[(int)barva, (int)hodnota] == null);
}
```

Tento příkaz vrátí hodnotu `true`, je-li prvek v poli `balíčekKaret` odpovídající zadané barvě a hodnotě `null`. V opačném případě vrátí `false`.

15. Vraťte se do metody `RozdejKartuZBalíčku`. Za kód, který náhodně vybírá barvu, přidejte následující cyklus `while`. V tomto cyklu voláme metodu `JeBarvaPrázdná` pro zjištění, zda v balíčku zůstaly nějaké karty s danou barvou. Pokud ne, vybereme náhodně jinou barvu (ve skutečnosti se může stát, že dojde opět k výběru stejné barvy) a kontrolujeme znovu. Cyklus se opakuje, dokud nenalezneme barvu s alespoň jednou zbývající kartou.

```
public HracíKarta RozdejKartuZBalíčku()
{
   Barva barva = (Barva)náhodnýVýběrKarty.Next(PočetBarev);
   while (this.JeBarvaPrázdná(barva))
   {
      barva = (Barva)náhodnýVýběrKarty.Next(PočetBarev);
   }
}
```

16. Nyní jsme náhodně vybrali barvu s alespoň jednou zbývající kartou. Dalším úkolem je náhodně vybrat kartu v této barvě. Pro výběr hodnoty karty můžeme použít generátor pseudonáhodných čísel, avšak stejně jako předtím nemáme jistotu, že karta se zvolenou hodnotou dosud nebyla rozdána. Můžeme ovšem i nyní použít stejný postup a zavolat metodu `JeKartaJižRozdaná` pro zjištění, zda byla daná karta již dříve rozdána, a pokud ano, vybrat náhodně jinou kartu a provést kontrolu znovu. To vše opakujeme tak dlouho, dokud nenalezneme vhodnou kartu. Přidejte tedy následující příkazy do metody `RozdejKartuZBalíčku`:

```
public HracíKarta RozdejKartuZBalíčku()
{
   ...
   Hodnota hodnota = (Hodnota)náhodnýVýběrKarty.Next(KaretNaBarvu);
   while (this.JeKartaJižRozdaná(barva, hodnota))
   {
      hodnota = (Hodnota)náhodnýVýběrKarty.Next(KaretNaBarvu);
```

```
    }
}
```

17. Nyní máme vybranou náhodnou hrací kartu, která dosud nebyla rozdána. Přidejte následující kód pro vydání této karty a pro nastavení odpovídajícího elementu v poli `balíčekKaret` na hodnotu `null`:

```
public HracíKarta RozdejKartuZBalíčku()
{
   ...
   HracíKarta karta = this.balíčekKaret[(int)barva, (int)hodnota];
   this.balíčekKaret[(int)barva, (int)hodnota] = null;
   return karta;
}
```

18. Dalším krokem je přidání vybrané karty do ruky. Otevřete soubor *Ruka.cs* a zobrazte jeho kód v okně editoru. Tento soubor obsahuje třídu `Ruka`, která implementuje ruku s kartami (tj. všechny karty rozdané jednomu hráči).

Tento soubor obsahuje konstantní veřejnou datovou složku `VelikostRuky` typu `int`, která je nastavena na velikost ruky s kartami (13). Dále obsahuje pole objektů typu `HracíKarta`, které je inicializované pomocí konstanty `VelikostRuky`. Datovou složku `početHracíchKaret` používáme ve svém kódu pro sledování počtu karet, které se aktuálně nacházejí v hráčově ruce.

```
class Ruka
{
   public const int VelikostRuky = 13;
   private HracíKarta[] karty = new HracíKarta[VelikostRuky];
   private int početHracíchKaret = 0;
   ...
}
```

Metoda `ToString` generuje řetězcovou reprezentaci karet v ruce. Pomocí cyklu `foreach` prochází prvky v poli `karty` a volá metodu `ToString` na každém nalezeném objektu typu `HracíKarta`. Tyto řetězce kvůli formátování vzájemně spojujeme znakem nového řádku (znak `"\n"`).

```
public override string ToString()
{
   string výsledek = "";
   foreach (HracíKarta karta in this.karty)
   {
      výsledek += karta.ToString() + "\n";
   }

   return výsledek;
}
```

19. Ve třídě `Ruka` vyhledejte metodu `PřidejKartuDoRuky`. Účelem této metody je přidat zadanou hrací kartu do ruky. Přidejte do této metody následující zvýrazněné příkazy:

```
public void PřidejKartuDoRuky(HracíKarta rozdávanáKarta)
{
   if (this.početHracíchKaret >= VelikostRuky)
   {
```

```
            throw new ArgumentException("Příliš mnoho karet");
        }
        this.karty[this.početHracíchKaret] = rozdávanáKarta;
        this.početHracíchKaret++;
    }
```

V tomto kódu nejdříve ověříme, že ruka dosud není plná, a pokud je, tak vyvoláme výjimku typu `ArgumentException`. V opačném případě přidáme zadanou kartu do pole `karty` na indexovou pozici stanovenou proměnnou `početHracíchKaret`, kterou následně inkrementujeme.

20. V okně *Solution Explorer* rozbalte uzel *Hra.xaml* a poté otevřete v okně editoru soubor *Hra.xaml.cs*. Jedná se o kód pro okno *Karetní hra*. Vyhledejte metodu `rozdejClick`. Tato metoda se spustí v okamžiku, kdy uživatel klepne na tlačítko *Rozdej*. Její kód vypadá takto:

```
private void rozdejClick(object sender, RoutedEventArgs e)
{
    try
    {
        balíček = new Balíček();

        for (int čísloRuky = 0; čísloRuky < PočetRukou; čísloRuky++)
        {
            ruce[čísloRuky] = new Ruka();
            for (int početKaret = 0; početKaret < Ruka.VelikostRuky;
                 početKaret++)
            {
                HracíKarta rozdávanáKarta = balíček.RozdejKartuZBalíčku();
                ruce[čísloRuky].PřidejKartuDoRuky(rozdávanáKarta);
            }
        }

        sever.Text = ruce[0].ToString();
        jih.Text = ruce[1].ToString();
        východ.Text = ruce[2].ToString();
        západ.Text = ruce[3].ToString();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, "Chyba", MessageBoxButton.OK,
                        MessageBoxImage.Error);
    }
}
```

První příkaz v bloku `try` vytváří nový balíček karet. Vnější cyklus vytváří z tohoto balíčku karet čtyři ruce a ukládá je do pole s názvem `ruce`. Vnitřní cyklus naplňuje každou ruku pomocí metody `RozdejKartuZBalíčku` pro získání náhodné karty z balíčku a metody `PřidejKartuDoRuky` pro přidání této karty do ruky.

Po rozdání všech karet se každá ruka zobrazí v samostatném textovém poli ve formuláři. Tato textová pole se jmenují `sever`, `jih`, `východ` a `západ`. Pro naformátování výstupu používáme metodu `ToString` každé ruky.

Pokud v jakémkoli okamžiku dojde k výjimce, zobrazí obsluha `catch` okno se zprávou obsahující chybovou zprávu pro vzniklou výjimku.

21. V nabídce *Debug* klepněte na příkaz *Start Without Debugging*. Jakmile se objeví okno *Karetní hra*, klepněte na tlačítko *Rozdej*. Karty v balíčku by se měly do každé ruky náhodně rozdat a karty v každé ruce by se měly zobrazit ve formuláři tak, jak to ukazuje následující obrázek:

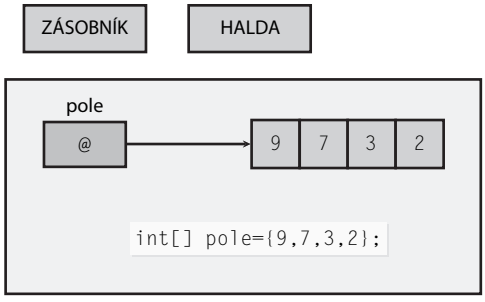
22. Klepněte znovu na tlačítko *Rozdej*. Dojde k novému rozdání karet, takže se karty v každé ruce změní.

23. Zavřete okno *Karetní hra* a vraťte se do Visual Studia.

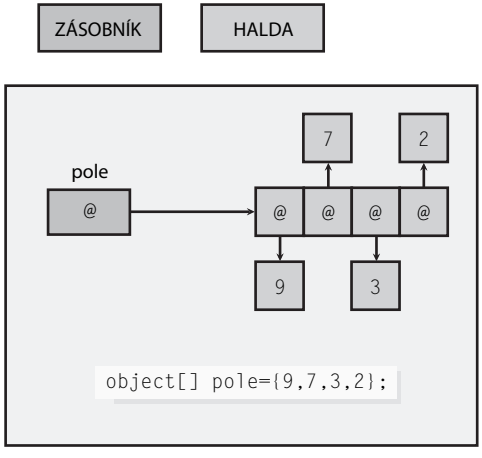
Co jsou kolekce

Pole jsou velmi užitečným nástrojem, i ona však mají svá omezení, z nichž nejzřetelnější je to, že pro přístup k prvkům pole je nutné používat celočíselný index. Naštěstí představují jen jednu z možností, jak shromažďovat prvky stejného typu do jednoho celku. Rozhraní Microsoft .NET Framework nabízí několik dalších tříd, které rovněž umí shromažďovat prvky, ale jinými, specializovanými, způsoby. Jedná se o třídy kolekcí, které se nacházejí v oboru názvů `System.Collections` a jemu podřízených oborech názvů.

Kromě problémů s indexy je zde ještě jeden základní rozdíl mezi polem a kolekcí. Pole totiž může uchovávat hodnotové typy. Základní třídy kolekcí přijímají, uchovávají a vracejí své prvky jako typy `object`, což znamená, že prvky v kolekci jsou typu `object`. Důsledky z toho vyplývající pochopíte lépe, když si porovnáme pole proměnných typu `int` (tedy hodnotového typu) a pole objektů (`object` je referenční typ). V poli proměnných typu `int` jsou hodnoty uloženy přímo, jak je vidět na následujícím schématu:



Nyní se podíváme na pole objektů. Do takového pole lze také přidávat celočíselné hodnoty (ve skutečnosti do něj lze přidávat hodnoty libovolného typu). Přidané celé číslo je automaticky zabaleno a prvek pole (odkaz na objekt) ukazuje na zabalenou kopii celočíselné hodnoty. Podobně je nutné prvek odstraňovaný z pole objektů rozbalit pomocí přetypování. (Zabalování jsme se věnovali v kapitole 8.) Následující schéma zachycuje pole s prvky typu `object` naplněné celočíselnými hodnotami:



V následujících částech této kapitoly najdete velmi stručný přehled čtyř nejužitečnějších tříd kolekcí. Další údaje o těchto třídách najdete v dokumentaci ke knihovně tříd rozhraní Microsoft .NET Framework.

Poznámka: Ve skutečnosti existují třídy kolekcí, u kterých nemusí být jen prvky typu `object`, ale mohou kromě prvků referenčního typu obsahovat i typy hodnotové. Než se však na ně podíváme, musíte o jazyku C# vědět trochu víc. S těmito typy tříd kolekcí se setkáte v kapitole 18.

Třída ArrayList (pole)

Třída `ArrayList` se hodí pro přesouvání prvků v poli. V některých případech je totiž klasické pole příliš omezující:

- Pokud potřebujete změnit velikost pole, musíte vytvořit pole nové, zkopírovat do něj prvky z pole starého (a některé přitom vynechat, pokud je nové pole menší) a nakonec aktualizovat všechny odkazy na původní pole tak, aby ukazovaly na pole nové.
- Je-li nutné odstranit nějaký prvek z pole, musíte přesunout všechny následující prvky o jednu pozici „dopředu". Tento postup má navíc jednu zásadní chybu: poslední prvek bude poté v poli dvakrát.
- Chcete-li vložit dovnitř pole nový prvek, musíte odsunout stávající prvky o jeden prvek „dozadu", abyste uvolnili potřebné místo. Tím však přijdete o poslední prvek pole!

Třída `ArrayList` poskytuje následující prvky, které vám pomohou tato omezení snadněji obejít:

- Prvek z proměnné typu `ArrayList` lze odstranit pomocí metody `Remove`. Kolekce typu `ArrayList` si pak zbylé prvky automaticky uspořádá.
- Prvek na konec kolekce typu `ArrayList` přidáte metodou `Add`, které předáte přidávaný prvek. Kolekce typu `ArrayList` v případě potřeby sama upraví svou velikost.
- Doprostřed kolekce typu `ArrayList` se dá nový prvek vložit metodou Insert. Také v tomto případě upraví kolekce typu `ArrayList` v případě potřeby sama svou velikost.
- Na existující prvek objektu typu `ArrayList` se můžete odkazovat pomocí běžného zápisu s indexem v hranatých závorkách.

Poznámka: Podobně jako u polí nemůžete při použití příkazu `foreach` pro průchod kolekcí typu `ArrayList` použít iterační proměnnou k úpravě obsahu této kolekce. Kromě toho nemůžete v rámci cyklu `foreach` procházejícího kolekci typu `ArrayList` volat žádnou z metod `Remove`, `Add` či `Insert`.

V následující ukázce uvidíte vytvoření kolekce typu `ArrayList`, její naplnění, manipulaci s jednotlivými prvky a průchod jejím obsahem:

```
using System;
using System.Collections;
...
ArrayList čísla = new ArrayList();
...
// naplň kolekci typu ArrayList
foreach (int číslo in new int[12]{10,9,8,7,7,6,5,10,4,3,2,1}
{
   čísla.Add(číslo);
}
...
// vlož prvek na předposlední pozici v seznamu
// a přesuň poslední položku vzhůru
// (prvním parametrem je pozice, druhým vkládaná hodnota)
čísla.Insert(čísla.Count-1, 99);
...
// odeber první prvek, jehož hodnota je 7 (čtvrtý prvek, index 3)
čísla.Remove(7);
// odeber prvek na sedmé pozici, index 6 (hodnota 10)
čísla.RemoveAt(6);
...
```

```
// projdi zbývajících 10 prvků v příkazu for
for (int i = 0; i < čísla.Count; i++)
{
   int číslo = (int)čísla[i]; // všimněte si přetypování
   Console.WriteLine(číslo);
}
...
// projdi zbývajících 10 prvků v příkazu foreach
foreach (int číslo in čísla) // zde bez přetypování
{
   Console.WriteLine(číslo);
}
```

Výstup tohoto kódu vypadá takto:

```
10
9
8
7
6
5
4
3
2
99
1
10
9
8
7
6
5
4
3
2
99
1
```

Poznámka: Vlastnost `Count` vrací počet prvků uvnitř kolekce. Tím se kolekce liší od polí, kde tento údaj obsahuje vlastnost `Length`.

Třída Queue (fronta)

Třída `Queue` implementuje mechanismus FIFO (first-in – first-out, první dovnitř – první ven). Prvky jsou zařazovány na konec fronty (metoda `Enqueue`) a odebírány z jejího začátku (metoda `Dequeue`).

Podívejme se na ukázku fronty a jejích metod:

```
using System;
using System.Collections;
...
Queue čísla = new Queue();
...
```

```
// naplň frontu
foreach (int číslo in new int[4]{9, 3, 7, 2})
{
   čísla.Enqueue(číslo); // zařaď do fronty
   Console.WriteLine("Číslo " + číslo + " vstoupilo do fronty");
}
...
// projdi frontu
foreach (int číslo in čísla)
{
   Console.WriteLine(číslo);
}
...
// vyprázdni frontu
while (čísla.Count != 0)
{
   int číslo = (int)čísla.Dequeue(); // odeber z fronty
   Console.WriteLine("Číslo " + číslo + " opustilo frontu");
}
```

Výstup tohoto kódu vypadá takto:

```
Číslo 9 vstoupilo do fronty
Číslo 3 vstoupilo do fronty
Číslo 7 vstoupilo do fronty
Číslo 2 vstoupilo do fronty
9
3
7
2
Číslo 9 opustilo frontu
Číslo 3 opustilo frontu
Číslo 7 opustilo frontu
Číslo 2 opustilo frontu
```

Třída Stack (zásobník)

Třída `Stack` implementuje mechanismus LIFO (last-in – first out, poslední dovnitř – první ven). Prvek je vložen na vrchol zásobníku (metoda `Push`) a odebírá se také z vrcholu zásobníku (metoda `Pop`). Můžeme to přirovnat ke sloupci talířů: talíře je možné přidávat jen navrch a shora je také možné je odebírat, takže poslední přidaný talíř ve sloupci (zásobníku) je prvním talířem, který může být ze sloupce odebrán. Podívejme se na ukázku:

```
using System;
using System.Collections;
...
Stack čísla = new Stack();
...
// naplň zásobník
foreach (int číslo in new int[4]{9, 3, 7, 2}
{
   čísla.Push(číslo); // vlož do zásobníku
   Console.WriteLine("Číslo " + číslo + " vloženo do zásobníku");
}
```

```
...
// projdi obsah zásobníku
foreach (int číslo in čísla)
{
   Console.WriteLine(číslo);
}
...
// vyprázdni zásobník
while (čísla.Count != 0)
{
   int číslo = (int)čísla.Pop(); // odeber ze zásobníku
   Console.WriteLine("Číslo " + číslo + " odebráno ze zásobníku");
}
```

Výstup tohoto programu vypadá takto:

```
Číslo 9 vloženo do zásobníku
Číslo 3 vloženo do zásobníku
Číslo 7 vloženo do zásobníku
Číslo 2 vloženo do zásobníku
2
7
3
9
Číslo 2 odebráno ze zásobníku
Číslo 7 odebráno ze zásobníku
Číslo 3 odebráno ze zásobníku
Číslo 9 odebráno ze zásobníku
```


Třída Hashtable (hashovací tabulka)

Pole a třída `ArrayList` dovolují očíslovat prvky pole celočíselnými indexy. Tento index zapíšete do hranatých závorek (například `[4]`) a získáte zpět prvek s určitým indexem (`4`, ve skutečnosti pátý prvek). V některých případech je však zapotřebí jiný způsob práce, při němž prvkům kolekce přiřazujete hodnoty jiného typu, jako třeba `string`, `double` nebo `Čas`. V jiných programovacích jazycích se takové konstrukci říká asociativní pole. Třída `Hashtable` tuto funkčnost poskytuje tak, že si interně udržuje dvě objektová pole. Jedno je určené pro přístupové klíče a druhé pro jim odpovídající hodnoty. Když do objektu typu `Hashtable` vložíte dvojici klíč/hodnota, bude klíč automaticky přiřazen dané hodnotě a vy budete moci rychleji a snadněji získat nějakou hodnotu z tabulky pomocí jejího klíče. Z povahy třídy `Hashtable` vyplývají následující důsledky:

- Klíče se v hashovací tabulce nesmějí opakovat. Volání metody `Add`, kterým se pokusíte do hashovací tabulky přidat nějaký klíč podruhé, skončí výjimkou. Můžete však také pro přidání páru klíč/hodnota použít zápis s hranatými závorkami (jak ukazuje následující příklad) bez nebezpečí vzniku výjimky, pokud byl daný klíč již dříve přidán. Pomocí metody `ContainsKey` můžete otestovat, zda se daný klíč v hashovací tabulce již nachází.
- Třída `Hashtable` je interně implementována jako řídká datová struktura, která funguje nejlépe tehdy, má-li velké množství paměti. Její velikost v paměti může při přidávání dalších prvků poměrně rychle růst.

- Když budete příkazem `foreach` procházet objekt typu `Hashtable`, obdržíte objekty typu `DictionaryEntry`. Třída `DictionaryEntry` poskytuje přístup ke klíčům a hodnotám v obou polích prostřednictvím vlastností `Key` a `Value`.

  V následující ukázce bude jménům členů rodiny přiřazen jejich věk a všechny dvojice údajů budou poté vypsány do okna příkazového řádku:

```
using System;
using System.Collections;
...
Hashtable stáří = new Hashtable();
...
// naplň kolekci typu Hashtable
stáří["Jan"] = 42;
stáří["Katka"] = 43;
stáří["Jakub"] = 15;
stáří["Iveta"] = 13;
...
// projdi hashovací tabulku pomocí příkazu foreach
// iterátor generuje objekt DictionaryEntry,
// obsahující pár klíč/hodnota
foreach (DictionaryEntry prvek in stáří)
{
   string jméno = (string)prvek.Key;
   int věk = (int)prvek.Hodnota;
   Console.WriteLine("Jméno: {0}, Věk: {1}", jméno, věk);
}
```

  Výstup výše uvedeného programu vypadá takto:

```
Jméno: Jakub, Věk: 15
Jméno: Jan, Věk: 42
Jméno: Iveta, Věk: 13
Jméno: Katka, Věk: 43
```

Třída SortedList (seřazený seznam)

Třída `SortedList` se velmi podobá třídě `Hashtable` v tom, že umožňuje přiřadit klíče hodnotám. Rozdílem je, že klíče pole jsou vždy seřazeny (proto se třída jmenuje `SortedList` neboli seřazený seznam).

Když do kolekce typu `SortedList` vložíte pár klíč/hodnota, bude klíč vložen do pole klíčů na správnou pozici (index) tak, aby pole zůstalo seřazeno. Poté bude hodnota vložena do pole hodnot tak, aby měla stejný index jako klíč. Třída `SortedList` automaticky zajišťuje, aby při přidávání a odebírání dvojic byly všechny klíče a hodnoty správně synchronizovány. To znamená, že do kolekce typu `SortedList` můžete vkládat páry klíč/hodnota v libovolném pořadí a vždy budou seřazeny podle klíčů.

Podobně jako kolekce `Hashtable`, ani kolekce `SortedList` nesmí obsahovat opakující se klíče. Při průchodu kolekcí typu `SortedList` v příkazu `foreach` obdržíte objekty typu `DictionaryEntry`, které však budou vraceny v pořadí daném vlastností `Key`.

V další ukázce bude členům rodiny přiřazen jejich věk a všechny dvojice pak budou vypsány do okna příkazového řádku. Jedná se o obdobu předchozího příkladu, tentokrát upravenou pro třídu `SortedList`:

```
using System;
using System.Collections;
...
SortedList stáří = new SortedList();
...
// naplň kolekci typu SortedList
stáří["Jan"] = 42;
stáří["Katka"] = 43;
stáří["Jakub"] = 15;
stáří["Iveta"] = 13;
...
// projdi kolekci pomocí příkazu foreach
// iterátor generuje objekt DictionaryEntry,
// obsahující pár klíč/hodnota
foreach (DictionaryEntry prvek in stáří)
{
   string jméno = (string)prvek.Key;
   int věk = (int)prvek.Hodnota;
   Console.WriteLine("Jméno: {0}, Věk: {1}", jméno, věk;
}
```

Výstup výše uvedeného programu vypadá takto:

```
Jméno: Iveta, Věk: 13
Jméno: Jakub, Věk: 15
Jméno: Jan, Věk: 42
Jméno: Katka, Věk: 43
```


Inicializační prvky kolekcí

Příklady v předchozích částech vám ukázaly, jak přidávat jednotlivé prvky do kolekce pomocí metody nejvhodnější pro danou kolekci (např. `Add` pro `ArrayList`, `Enqueue` pro `Queue` či `Push` pro `Stack`). Některé typy kolekcí můžete inicializovat také při jejich deklaraci pomocí syntaxe velmi podobné té, která se používá pro pole. Kupříkladu následující příkaz vytvoří a inicializuje objekt `čísla` typu `ArrayList` jiným způsobem nežli voláním metody `Add`:

```
ArrayList čísla = new ArrayList(){10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1};
```

Kompilátor jazyka C# ve skutečnosti tuto inicializaci převede na sérii volání metody `Add`. Tuto syntaxi tedy můžete použít pouze pro kolekce, které podporují metodu `Add` (např. třídy `Stack` a `Queue` ji nemají).

Pro složitější kolekce, jako je kupříkladu kolekce typu `Hashtable` přijímající dvojici klíč/hodnota, můžete zadat každou dvojici klíč/hodnota v seznamu inicializačních prvků jako anonymní typ:

```
Hashtable stáří = new Hashtable(){{"Jan", 42}, {"Katka", 43},
                                  {"Jakub", 15}, {"Iveta", 13}};
```

První položka v každé dvojici je klíč a druhá je hodnota.

Srovnání polí a kolekcí

Důležité rozdíly mezi poli a kolekcemi se dají shrnout do tří bodů:

- Pole deklaruje typ svých prvků, kolekce ne, protože ukládají své prvky jako objekty.

- Instance pole má pevnou velikost a nemůže se zvětšit ani zmenšit. Kolekce dokáže svou velikost podle potřeby dynamicky měnit.
- Pole může mít více než jednu dimenzi, zatímco kolekce je lineární. Nicméně prvky v kolekci mohou být samy kolekcemi, takže vícerozměrné pole můžete simulovat jako kolekci kolekcí.

Implementace karetní hry pomocí kolekcí

1. Vraťte se do projektu Karty z předchozího cvičení.

Poznámka: Kompletní verzi tohoto projektu pro předchozí cvičení najdete ve složce *\Dokumenty\Visual CSharp 2010 Krok za krokem\Kapitola 10\Karty s použitím polí – Hotové*.

2. V okně editoru zobrazte soubor *Balíček.cs*. Všimněte si následujícího příkazu `using` v horní části souboru:

```
using System.Collections;
```

Třídy kolekcí jsou umístěny v tomto oboru názvů.

3. Ve třídě `Balíček` změňte dle níže uvedeného kódu definici dvourozměrného pole `balíčekKaret` na objekt typu `Hashtable`:

```
class Balíček
{
...
private Hashtable balíčekKaret;
...
}
```

Vzpomeňte si, že třída `Hashtable` definuje kolekci s prvky typu `object`, takže nikde neuvádíme typ `HracíKarta`. Kromě toho původní pole mělo dva rozměry, zatímco objekt typu `Hashtable` má pouze jeden. Dvourozměrné pole budete simulovat pomocí objektů kolekce typu `SortedList` ve formě prvků hashovací tabulky.

4. Vyhledejte konstruktor třídy `Balíček`. Upravte první příkaz v tomto konstruktoru tak, aby se proměnná `balíčekKaret` neinicializovala jako pole, ale jako objekt typu `Hashtable`:

```
public Balíček()
{
   this.balíčekKaret = new Hashtable();
   ...
}
```

5. Ve vnějším cyklu deklarujte objekt kolekce typu `SortedList` s názvem `karetVBarvě`. Změňte kód vnitřního cyklu tak, aby se objekt typu `HracíKarta` přidával do kolekce, a ne do pole. Po vnitřním cyklu přidejte objekt typu `SortedList` do hashovací tabulky `balíčekKaret`, přičemž jako klíč tohoto prvku uveďte hodnotu proměnné `barva`. (Tento objekt typu `SortedList` obsahuje všechny karty v balíčku pro zadanou barvu, přičemž hashovací tabulka obsahuje kolekci těchto objektů typu `SortedList`.)

Následující kód ukazuje kompletní konstruktor se zvýrazněnými změnami:

```
public Balíček()
{
   this.balíčekKaret = new Hashtable();
```

```
    for (Barva barva = Barva.Kříže; barva <= Barva.Piky; barva++)
    {
       SortedList karetVBarvě = new SortedList();
       for (Hodnota hodnota = Hodnota.Dva; hodnota <= Hodnota.Eso;
            hodnota++)
       {
          karetVBarvě.Add(hodnota, new HracíKarta(barva, hodnota));
       }
       this.balíčekKaret.Add(barva, karetVBarvě);
    }
}
```

6. Vyhledejte metodu `RozdejKartuZBalíčku`. Vzpomeňte si, že tato metoda náhodně vybere z balíčku kartu, odebere ji z balíčku a vrátí ji. Logiku pro výběr karty není nutné nijak měnit, avšak příkazy na konci metody, které získávají kartu a odebírají ji z pole, musíme aktualizovat tak, aby používaly kolekci typu `Hashtable`.

Dle následujícího zvýrazněného kódu upravte kód za uzavírací složenou závorkou druhého cyklu `while`:

```
public HracíKarta RozdejKartuZBalíčku()
{
   Barva barva = (Barva)náhodnýVýběrKarty.Next(PočetBarev);
   while (this.JeBarvaPrázdná(barva))
   {
      barva = (Barva)náhodnýVýběrKarty.Next(PočetBarev);
   }

   Hodnota hodnota = (Hodnota)náhodnýVýběrKarty.Next(KaretNaBarvu);
   while (this.JeKartaJižRozdaná(barva, hodnota))
   {
      hodnota = (Hodnota)náhodnýVýběrKarty.Next(KaretNaBarvu);
   }

   SortedList karetVBarvě = (SortedList) balíčekKaret[barva];
   HracíKarta karta = (HracíKarta)karetVBarvě[hodnota];
   karetVBarvě.Remove(hodnota);
   return karta;
}
```

Tato hashovací tabulka obsahuje kolekci objektů typu `SortedList`, jeden pro každou barvu karet. Tento nový kód vezme z hashovací tabulky objekt typu `SortedList` pro danou kartu náhodně zvolené barvy a poté vezme z tohoto objektu typu `SortedList` kartu s vybranou hodnotou. Poslední nový příkaz odebere kartu z objektu typu `SortedList`.

7. Vyhledejte metodu `JeKartaJižRozdaná`. Tato metoda zjišťuje, zda již byla daná karta rozdána, což provádí testování, má-li odpovídající prvek v poli hodnotu `null`. Tuto metodu nyní musíte upravit, protože je nutné zjistit, zda se karta s uvedenou hodnotou nachází v kolekci typu `SortedList` uložené pro zadanou barvu v hashovací tabulce `balíčekKaret`. Metodu aktualizujte podle následujícího zvýrazněného kódu:

```
private bool JeKartaJižRozdaná(Barva barva, Hodnota hodnota)
{
   SortedList karetVBarvě = (SortedList)this.balíčekKaret[barva];
   return (!karetVBarvě.ContainsKey(hodnota));
}
```

8. V okně editoru zobrazte soubor *Ruka.cs*. Tato třída používá pole pro uchovávání hracích karet v ruce. Upravte definici pole `karty` tak, aby používala kolekci typu `ArrayList`:

```
class Ruka
{
   public const int VelikostRuky = 13;
   private ArrayList karty = new ArrayList();
   ...
}
```

9. Vyhledejte metodu `PřidejKartuDoRuky`. Tato metoda v současnosti kontroluje, je-li ruka plná, a pokud ne, tak přidá zadanou kartu do pole `karty` na index stanovený proměnnou `početHracíchKaret`.

   Metodu aktualizujte tak, aby používala metodu `Add` třídy `ArrayList`. Díky této změně není nutné explicitně sledovat počet karet v kolekci, protože k tomuto účelu lze použít vlastnost `Count`. Použijte ji tedy v příkazu `if`, který kontroluje, zda je ruka plná, a proměnnou `početHracíchKaret` vymažte ze třídy.

   Celá metoda by nyní měla vypadat takto:

```
public void PřidejKartuDoRuky(HracíKarta rozdávanáKarta)
{
   if (this.karty.Count >= VelikostRuky)
   {
      throw new ArgumentException("Příliš mnoho karet");
   }
   this.karty.Add(rozdávanáKarta);
}
```

10. V nabídce *Debug* klepněte na příkaz *Start Without Debugging*, čímž sestavíte a spustíte aplikace.
11. Jakmile se otevře okno *Karetní hra*, klepněte na tlačítko *Rozdej*. Zkontrolujte, že se jako dříve karty rozdají a ruce se naplní kartami. Klepněte znovu na tlačítko *Rozdej* pro vygenerování další pseudonáhodné sady rozdaných karet.
12. Zavřete formulář a vraťte se do Visual Studia 2010.

V této kapitole jste se naučili vytvářet pole a používat je pro manipulaci se sadami dat. Kromě toho jste viděli, jak se dají data v paměti ukládat odlišným způsobem, a to pomocí některých základních tříd kolekcí.

- Pokud chcete pokračovat další kapitolou, nechte Visual Studio 2010 běžet a nalistujte kapitolu 11.
- Pokud chcete Visual Studio 2010 nyní ukončit, v nabídce *File* klepněte na příkaz *Exit*. Pokud se objeví dialogové okno s dotazem na uložení změn, klepněte na tlačítko Yes a uložte projekt.

Stručné shrnutí kapitoly 10

Pro...	... učiňte následující:
Deklarování proměnné typu `pole`	Napište jméno typu uchovávaných prvků, za ním hranaté závorky, název proměnné a středník: `bool[] příznaky;`
Vytvoření instance pole	Napište klíčové slovo `new`, název typu uchovávaných prvků a za něj velikost pole do hranatých závorek: `bool[] příznaky = new bool[10];`

Inicializaci prvků pole (či kolekcí podporujících metodu `Add`) určitými hodnotami	V případě pole napište požadované hodnoty mezi složené závorky a oddělte je čárkami:
	`bool[] příznaky = {true, false, true, true};`
	Pro kolekci použijte operátor `new` a typ kolekce se specifickými hodnotami oddělenými čárkou uvnitř složených závorek:
	`ArrayList členové = new ArrayList(){10, 9, 8, 7, 6, 5};`

Zjištění počtu prvků v poli	Použijte vlastnost `Length`:
	`bool[] příznaky = ...;`
	`...`
	`int početPrvků = příznaky.Length;`

Zjištění počtu prvků v kolekci	Použijte vlastnost `Count`:
	`ArrayList příznaky = new ArrayList();`
	`...`
	`int početPrvků = příznaky.Count;`

Přístup k jednomu prvku pole	Napište název proměnné typu `pole` a za něj mezi hranaté závorky celočíselný index prvku. Indexy začínají nulou, nikoli jedničkou: `bool prvníPrvek = příznaky[0];`

Průchod prvky pole nebo kolekce	Použijte příkaz `for` nebo `foreach`, například:
	`bool[] příznaky = {true, false, true, true};`
	
	`for (int i = 0; i < příznaky.Length; i++)`
	`{`
	`  Console.WriteLine(příznaky[i]);`
	`}`
	
	`foreach (bool příznak in příznaky)`
	`{`
	`  Console.WriteLine(příznak);`
	`}`