Strukturované datové typy

Strukturované datové typy, objekt, pole, kolekce, generické kolekce

Joseph Soukup – P4

Strukturované datové typy

- Slouží k agregaci (spojení) různých dat a různých funkcí do jednoho objektu
- Nejznámější jsou struct a class
- Oba typy se liší pouze ve výchozích přístupových právech
- Objekty, které se skládají z několika komponent (členů)
- Přístup k jednotlivým členům je možný pomocí speciálních operací (selektory)
- Dělí se na:
    - **Homogenní –** komponenty jsou stejného typu
    - **Heterogenní –** komponenty jsou rozdílného typu

Objekt

- Definice třídy nebo struktury je jako podrobný plán, který určuje, co může typ provádět
- Objekt je v podstatě blok paměti, který byl přidělen a nakonfigurován podle podrobného plánu
- Program může vytvořit mnoho objektů stejné třídy
- Objekty e nazývají instance a mohou být uloženy buď v pojmenované proměnné nebo v poli nebo v kolekci.
- **Instance struktury vs. Instance třídy:**
    - Vzhledem k tomu, že třídy jsou odkazové typy, proměnná objektu třídy obsahuje odkaz na adresu objetu na spravované haldě
    - Pokud je druhý objekt stejného typu přiřazen k prvnímu objektu, pak obě proměnné odkazují na objekt na dané adrese
    - Instance třídy jsou vytvořeny pomocí operátoru new
        -
```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
    // Other properties, methods, events...
}

class Program
{
    static void Main()
    {
        Person person1 = new Person("Leopold", 6);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

        // Declare new person, assign person1 to it.
        Person person2 = person1;

        // Change the name of person2, and person1 also changes.
        person2.Name = "Molly";
        person2.Age = 16;

        Console.WriteLine("person2 Name = {0} Age = {1}", person2.Name, person2.Age);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
    Output:
    person1 Name = Leopold Age = 6
    person2 Name = Molly Age = 16
    person1 Name = Molly Age = 16
*/
```
    - Vzhledem k tomu, že struktury jsou datové typy hodnot, proměnná objektu struc obsahuje kopii celého objektu
    - Instance struktur lze také vytvořit pomocí new operátoru, ale to není vyžadováno:

```
public struct Person
{
    public string Name;
    public int Age;
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

public class Application
{
    static void Main()
    {
        // Create  struct instance and initialize by using "new".
        // Memory is allocated on thread stack.
        Person p1 = new Person("Alex", 9);
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

        // Create  new struct object. Note that  struct can be initialized
        // without using "new".
        Person p2 = p1;

        // Assign values to p2 members.
        p2.Name = "Spencer";
        p2.Age = 7;
        Console.WriteLine("p2 Name = {0} Age = {1}", p2.Name, p2.Age);

        // p1 values remain unchanged because p2 is  copy.
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
  Output:
    p1 Name = Alex Age = 9
    p2 Name = Spencer Age = 7
    p1 Name = Alex Age = 9
*/
```

- **Identita objektu vs. Hodnota rovnosti**
  - Když porovnáte dva objekty pro rovnost, musíte nejprve rozlišovat, zda chcete zjistit, zda tyto proměnné představují stejný objekt v paměti nebo zda jsou hodnoty jednoho nebo více jejich polí ekvivalentní
  - Pokud hodláte porovnat hodnoty, je nutné vzít v úvahu, zda jsou objekty instance typů hodnot (struktury) nebo typy odkazů (třídy, delegáti, pole)
    - Stejné umístění v paměti – Equals metodu
    - Stejné hodnoty (ValueType.Equals)

```
// Person is defined in the previous example.

//public struct Person
//{
//    public string Name;
//    public int Age;
//    public Person(string name, int age)
//    {
//        Name = name;
//        Age = age;
//    }
//}

Person p1 = new Person("Wallace", 75);
Person p2;
p2.Name = "Wallace";
p2.Age = 75;

if (p2.Equals(p1))
    Console.WriteLine("p2 and p1 have the same values.");

// Output: p2 and p1 have the same values.
```

Struktura

- Typ proměnné, který umožňuje sdružování dat různých datových typů do jedné proměnné
- Definuje se pomocí klíčového slova struct
- Můžou obsahovat i metody, indexery, konstruktor
- Struct je hodnotový typ (třída je referenční)
- Nepodporují dědičnost a neobsahují defaultní konstruktor
-
```
struct Books {
   public string title;
   public string author;
   public string subject;
   public int book_id;
};
```

Pole

- Pomocí operátoru new, který určuje typ prvku pole a počet prvků
-
```
int[] array = new int[5];
```
- Toto pole obsahuje prvky z array[0] do array[4] (Prvky jsou inicializovány na výchozí hodnotu typu prvku 0 pro celá čísla)
- Možnost ukládat jakýkoliv typ prvku
- Inicializace:
  - Možnost inicializovat prvky pole
    -
```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```
- Homogenní strukturovaný datový typ
- Každá položka je označena indexem a pomocí tohoto indexu je přístupná její hodnota
-

indexy	0	1	2	3	4	5	6	7
	15	3	21	8	3	12	5	3

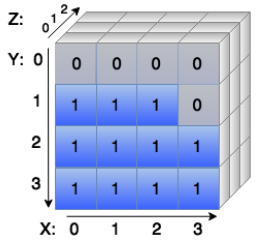
- Pole využíváme, pokud chceme uložit údaje o více prvcích stejného typu – například 50 čísel
-
```
int[] pole = new int[10]; //deklarace pole datového typu int, o velikosti 10 prvků
pole[0] = 4; //přiřazení hodnoty na index 0
int prvniPolozka = pole[0]; //přečtení hodnoty na indexu 0
```

Vícerozměrná pole

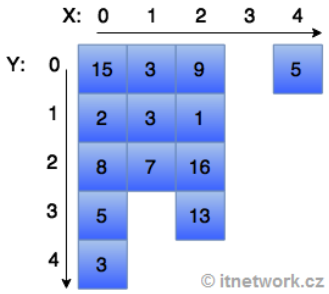
- **Dvou rozměrná**:
  - Například kino sál:
    -

Y: \ X:	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	1	0	0
3	0	1	1	1	0
4	1	1	1	1	1

- Deklaruje následujícím způsobem:
  - ```
    int[,] kinosal = new int [5, 5];
    ```
  - První číslice počet sloupců a druhá počet řádků
  - Po deklaraci automaticky inicializována samými nulami (v paměti tabulka plných nul)
- Naplnění daty:
  - Kinosál naplníme jednička jak na výše uvedeném obrázku
  - ```
    kinosal[2, 2] = 1; // Prostředek
    for (int i = 1; i < 4; i++) // 4. řádek
    {
        kinosal[i, 3] = 1;
    }
    for (int i = 0; i < 5; i++) // Poslední řádek
    {
        kinosal[i, 4] = 1;
    }
    ```
- Výpis:
  - Length – celkový počet prvků v poli (25)
  - GetLength() - přijímá jako parametr dimenzi (0 - sloupce, 1 - řádky) - počet prvků v dimenzi
  - ```
    for (int j = 0; j < kinosal.GetLength(1); j++)
    {
        for (int i = 0; i < kinosal.GetLength(0); i++)
        {
            Console.Write(kinosal[i, j]);
        }
        Console.WriteLine();
    }
    ```
  - ```
    00000
    00000
    00100
    01110
    11111
    ```

- **N – rozměrná**
  - Pole o ještě víc dimenzích
  - Př. Budova kinosálu s více partami (nebo více kinosálů)
  - 

  - 3D:
    - ```
      int[, ,] kinosaly = new int [5, 5, 3];
      ```
    - Přistupování k němu přes indexer (již musíme zadat 3 souřadnice)
    - ```
      kinosaly[3, 2, 1] = 1; // Druhý kinosál, třetí řada, čtvrtý sloupec
      ```
    - GetLength() - přidáme parametr 2, který nám vrátí počet pater
- **Pole polí**

- 2D pole není nic jiného než pole polí
- Pole o 5 prvcích (1. řádek) a každá buňka v tomto řádku v sobě bude obsahovat další pole, reprezentující sloupeček
- ```
  int[][] kinosal = new int[5][];
  ```
- Výhodou je, že si do každého řádku/sloupce můžeme uložit jak velké pole chceme (obejdeme se zbytečného plýtvání pamětí)
- „Zubaté“ (jagged)
  - 

- Nevýhodou je, že se musí pole nepříjemně inicializovat samo
- Původní řádek s pěti buňkami sice existuje, ale jednotlivé sloupečky se do něho musí vkládat sami (zatím vložíme sloupečky o 5 prvcích)
  - ```
    for (int i = 0; i < kinosal.Length; i++)
    {
        kinosal[i] = new int[5];
    }
    ```
- Velikost pole získáme takto:
  - ```
    int sloupcu = kinosal.Length;
    int radku = 0;
    if (sloupcu != 0)
        radku = kinosal[0].Length;
    ```
- Přístup pomocí 2 indexerů:
  - ```
    kinosal[4][2] = 1; // Obsazujeme sedadlo v 5. sloupci a 3. řadě
    ```
- Zkrácená inicializace:
  - ```
    int[,] kinosal = new int[,] {
        { 0, 0, 0, 0, 1 },
        { 0, 0, 0, 1, 1 },
        { 0, 0, 1, 1, 1 },
        { 0, 0, 0, 1, 1 },
        { 0, 0, 0, 0, 1 }
    };
    ```
  - ```
    int[][] zubatePole = new int[][] {
        new int[] {15, 2, 8, 5, 3},
        new int[] {3, 3, 7},
        new int[] {9, 1, 16, 13},
        new int[] {},
        new int[] {5}
    };
    ```

Kolekce

- Soubor dat, které jsou většinou stejného typu a slouží ke specifickému účelu
- Cílem kolekce je sloužit jako úložiště objektů a zajišťovat k nim přístup
- Různé typy – list, pole, seznam, strom, slovník, zásobník, fronta

Obecné kolekce (Negenrické)

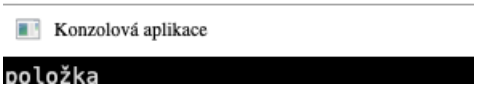
- Všechny datové typy mají předchůdce Object, můžeme prvky v naší kolekci ukládat právě do tohoto datového typu
- Do kolekce můžeme uložit v podstatě cokoliv
- Nevýhoda:
    - Sama kolekce skutečný datový typ nezná a proto umí prvky navracet jen jako obecné objekty -> po získaní prvku si je musíme přetypovat

```
ArrayList list = new ArrayList();
list.Add("položka");

string polozka = (string)list[0];

Console.WriteLine(polozka);
```

Výstup programu:



- Podpora jen pro zásobník, frontu, listy a hash tabulky

Generické kolekce

- Zavádí tzv. genericitu
- Možnost specifikovat datový typ až ve chvíli vytvoření instance
- Ve třídě samotné kolekce se poté pracuje s generickým typem, který slouží jako zástupce pro budoucí datový typ
- Pokud se generická třída změní na string až když vytvoříme její instanci (parametrizování)

```
List<string> list = new List<string>();
list.Add("položka");

string polozka = list[0];

Console.WriteLine(polozka);
```

Přistupování k prvkům kolekce

- LINQ (jazykově intergovaný dotaz) -> možnost filtrování, řazení a seskupování

```
private static void ShowLINQ()
{
    List<Element> elements = BuildList();

    // LINQ Query.
    var subset = from theElement in elements
                 where theElement.AtomicNumber < 22
                 orderby theElement.Name
                 select theElement;

    foreach (Element theElement in subset)
    {
        Console.WriteLine(theElement.Name + " " + theElement.AtomicNumber);
    }

    // Output:
    //  Calcium 20
    //  Potassium 19
    //  Scandium 21
}

private static List<Element> BuildList()
{
    return new List<Element>
    {
        { new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        { new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        { new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        { new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}
```

- **Řazení kolekce:**
    - Řadí instanci Car (implementuje IComparable<T> rozhraní, které vyžaduje CompareTo implementace metody)
    - CompareTo -> porovnání, které se používá k řazení

```
private static void ListCars()
{
    var cars = new List<Car>
    {
        { new Car() { Name = "car1", Color = "blue", Speed = 20}},
        { new Car() { Name = "car2", Color = "red", Speed = 50}},
        { new Car() { Name = "car3", Color = "green", Speed = 10}},
        { new Car() { Name = "car4", Color = "blue", Speed = 50}},
        { new Car() { Name = "car5", Color = "blue", Speed = 30}},
        { new Car() { Name = "car6", Color = "red", Speed = 60}},
        { new Car() { Name = "car7", Color = "green", Speed = 50}}
    };

    // Sort the cars by color alphabetically, and then by speed
    // in descending order.
    cars.Sort();

    // View all of the cars.
    foreach (Car thisCar in cars)
    {
        Console.Write(thisCar.Color.PadRight(5) + " ");
        Console.Write(thisCar.Speed.ToString() + " ");
        Console.Write(thisCar.Name);
        Console.WriteLine();
    }

    // Output:
    //   blue  50 car4
    //   blue  30 car5
    //   blue  20 car1
    //   green 50 car7
    //   green 10 car3
    //   red   60 car6
    //   red   50 car2
}

public class Car : IComparable<Car>
{
    public string Name { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public int CompareTo(Car other)
    {
        // A call to this method makes a single comparison that is
        // used for sorting.

        // Determine the relative order of the objects being compared.
        // Sort by color alphabetically, and then by speed in
        // descending order.

        // Compare the colors.
        int compare;
        compare = String.Compare(this.Color, other.Color, true);

        // If the colors are the same, compare the speeds.
        if (compare == 0)
        {
            compare = this.Speed.CompareTo(other.Speed);

            // Use descending order for speed.
            compare = -compare;
        }

        return compare;
    }
}
```

- **Definice vlastní kolekce:**
  - Implementace IEnumerable<T> IEnumerable rozhraní
  - Příklad implementuje IEnumerable rozhraní, které vyžaduje GetEnumerator implementaci
  - GetEnumerator metoda vrací ColorEnumerator třídy -> implementuje IEnumerator rozhraní, které vyžaduje, aby byla Current MoveNext implementována vlastnost, metoda a Reset metoda

```
private static void ListColors()
{
    var colors = new AllColors();

    foreach (Color theColor in colors)
    {
        Console.Write(theColor.Name + " ");
    }
    Console.WriteLine();
    // Output: red blue green
}

// Collection class.
public class AllColors : System.Collections.IEnumerable
{
    Color[] _colors =
    {
        new Color() { Name = "red" },
        new Color() { Name = "blue" },
        new Color() { Name = "green" }
    };

    public System.Collections.IEnumerator GetEnumerator()
    {
        return new ColorEnumerator(_colors);

        // Instead of creating a custom enumerator, you could
        // use the GetEnumerator of the array.
        //return _colors.GetEnumerator();
    }

    // Custom enumerator.
    private class ColorEnumerator : System.Collections.IEnumerator
    {
        private Color[] _colors;
        private int _position = -1;

        public ColorEnumerator(Color[] colors)
        {
            _colors = colors;
        }

        object System.Collections.IEnumerator.Current
        {
            get
            {
                return _colors[_position];
            }
        }

        bool System.Collections.IEnumerator.MoveNext()
        {
            _position++;
            return (_position < _colors.Length);
        }

        void System.Collections.IEnumerator.Reset()
        {
            _position = -1;
        }
    }
}

// Element class.
public class Color
{
    public string Name { get; set; }
}
```

- **Iterátory**
  - Používá se k provedení vlastní iterace v kolekci
  - Může být metoda nebo get přistupující objekt
  - Používá yield return k vrácení každého prvku kolekce po jednom
  - Zavoláte iterátor pomocí příkazu foreach
  - Každá iterace foreach smyčky volá iterátor

- Při yield return dosažení příkazu v iterátoru je vrácen a aktuální umístění v kódu je uchováno
- Spuštění je restartováno z tohoto umístění při příštím volání iterátoru
- Následující příklad používá metodu iterátoru
- Metoda iterátoru obsahuje yield return příkaz, který uvnitř smyčky for.
- V ListEvenNumbers metodě Každá iterace foreach těla příkazu vytvoří volání metody iterátoru, která pokračuje k dalšímu yield return příkazu.
-

```
private static void ListEvenNumbers()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    Console.WriteLine();
    // Output: 6 8 10 12 14 16 18
}

private static IEnumerable<int> EvenSequence(
    int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (var number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```