Programovanie v jazyku C#

Údajové štruktúry a typy v C#

prednáška 6

Ing. Ján Magyar, PhD.

ak. rok. 2023/2024 ZS

Anonymné typy

trieda bez mena (iba kompilátor pozná), dedí od `object`

definícia je odvodená z inicializácie

pomocou `var` a `new`

```
var captain = new
{
    FirstName = "James",
    MiddleName = "Tiberius",
    LastName = "Kirk"
};
```

priradenie do premennej je možné iba pri rovnakých property

Records (záznamy)

nemeniteľný referenčný typ

porovnávanie na základe hodnôt

kľúčové slovo `record`

definícia podobná triede

dva typy definície:

- nominálny záznam
- pozičný záznam

Nominálny záznam

```
public record Book
{
    public string Title { get; init; } = string.Empty;
    public string Publisher { get; init; } = string.Empty;
}
```

```
Book book1 = new() {
    Title = "The Lord of the Rings",
    Publisher = "Allen & Unwin"

};
```

`init` naznačuje, že hodnotu vieme nastaviť iba pri vytvorení objektu, neskôr ju zmeniť nemôžeme

Pozičný záznam

```
public record Book(string Title, string Publisher)
{
    // your code can go here
}


Book b2 = new("The Lord of the Rings", "Allen & Unwin");
```

Funkcionalita záznamov

porovnávanie na základe hodnôt: == a `!=`

porovnávanie na základe referencií: `object.ReferenceEquals()`

kopírovanie cez `with`:

```
var newBook = book1 with { Title = "The Hobbit" };
```

Structs (štruktúry)

kľúčové slovo `struct` namiesto `class`

```
public struct Dimensions
{
    public Dimensions(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Length { get; }
    public double Width { get; }
}
```

uložené v zásobníku, bez potreby použitia garbage collector

Štruktúry a triedy

štruktúry nepodporujú dedičnosť, ale môžu implementovať rozhranie

štruktúry vždy majú defaultný konštruktor, nevieme zadefinovať konštruktor bez parametra

v štruktúrach vieme presnejšie definovať manažment pamäti

štruktúry sa ukladávajú v zásobníku alebo v halde ak sú v rámci objektu

ak štruktúru odovzdávame ako parameter, zaboxuje sa a kopíruje sa do haldy

Enums (enumeračné typy)

hodnotový typ so zoznamom menovaných konštánt

```
public enum Color
{
   Red,
   Green,
   Blue
}
```

Zgrupovanie enumerácií

```
[Flags] // generate string representation
public enum DaysOfWeek
{
    Monday = 0x1,
    Tuesday = 0x2,
    Wednesday = 0x4,
    Thursday = 0x8,
    Friday = 0x10,
    Saturday = 0x20,
    Sunday = 0x40
}

DaysOfWeek weekend = DaysOfWeek.Saturday | DaysOfWeek.Sunday;
```

Pretypovanie enumeračných typov

```
public enum Color : short
{
   Red = 1,
   Green = 2,
   Blue = 3
}


Color c1 = (Color)2;
short number = (short)c1;
```

Transformácia enumerácií

parsovanie:

```
if (Enum.TryParse<Color>("Red", out Color red))
{
   Console.WriteLine($"successfully parsed {red}");
}
```

`Enum.TryParse` vráti `true/false`, hodnota sa uloží do premennej po kľúčovom slove `out`

stringové reprezentácie: `Enum.GetNames(typeof(Color))`

hodnoty: `Enum.GetValues(typeof(Color))`

Tuples (n-tice)

zgrupovanie objektov rôznych typov bez nutnosti definovať vlastný typ

pre podporu viacerých návratových hodnôt

```
(Book Book, int Number, double discount) tup1 = (new Book("The Hobbit", "Allen Unwin"), 42, 0.1);
```

```
(Book book, int count, double discnt) = tup1;
```

kopírovanie možné pri rovnakých typoch členov, na názve nezáleží

Viaceré návratové hodnoty

```
static (int result, int remainder) Divide(int
dividend, int divisor)
{
   int result = dividend / divisor;
   int remainder = dividend % divisor;
   return (result, remainder);
}


(int result, int remainder) = Divide(7, 2);
```

Arrays (polia)

ukladávanie viacerých inštancií rovnakého typu, veľkosť počas behu nemôžeme meniť

podpora triedenia, filtrovania a enumerácie

referenčný typ

```
int[] myArray = new int[10];
int[,] twoDim = new int[3, 3];

int[] myArray = new int[4] {1, 2, 3, 4};
int[] myArray = new int[] {1, 2, 3, 4};
int[] myArray = {1, 2, 3, 4};
```

Jagged array

každý riadok môže byť rôznej dĺžky

```
int[][] jagged =
{
   new[] { 1, 2 },
   new[] { 3, 4, 5, 6, 7, 8 },
   new[] { 9, 10, 11 }
};
```

Práca s poliami

indexovanie (iba cez `int`) - `IndexOutOfRangeException` pre nesprávnu hodnotu

```
myArray[1] = 42;
twoDim[0, 1] = 2;
```

dĺžka poľa: `Length` property

```
myArray.Length
```

vytváranie kópií

- `Clone()` - shallow kópia (pre hodnotové typy) - vytvorí nové pole
- `Copy()` - shallow kópia (pre referenčné typy) - kopíruje obsah
- deep kópia - potrebujeme vytvoriť kópie členov cez iterátor

Triedenie

pomocou quicksort algoritmu

typ členov musí implementovať `IComparable` (metóda `CompareTo`)

volanie `Array.Sort(myArray)`

vieme odovzdávať aj komparátor, ktorý môže implementovať iný spôsob porovnávania členov

alternatívne vieme použiť delegate

Slicing

výber niekoľkých prvkov naraz

cez span:

```
Span<int> span1 = new(myArray);
Span<int> span2 = new(myArray, start: 3, length: 4);
Span<int> span3 = span1.Slice(start: 2, length: 4);
```

`length` je nepovinný parameter

pre nemeniteľné hodnoty používajte `ReadOnlySpan`

generuje `ArgumentOutOfRangeException`

Práca so span objektmi

`Clear()` - nastaví defaultnú hodnotu

`Fill()` - nastaví zadanú hodnotu

`CopyTo()` - kopírovanie do existujúceho span objektu

`TryCopyTo()` - vráti `false` ak kopírovanie nie je úspešné

Indexovanie

rozšírené možnosti od C# 8

```
int[] data = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int last1 = data[data.Length - 1];
int last2 = data[^1];
Index lastIndex = ^1;
```

výber intervalu cez `range`

```
var slice = data[3..5];
```

Collections (kolekcie)

pre zoskupenia s dynamickým počtom prvkov

rovnaká základná funkcionalita, rôzne rozhrania

Základné operácie

pridanie na koniec

pridávanie

načítanie

vyhľadávanie

odstránenie

Lists (zoznamy)

kolekcia s dynamicky sa meniacou dĺžkou

kapacia sa defaultne nastaví na 0, 4, následne sa zdvojnásobuje

`List<int> intList = new();` - prázdny zoznam

`List<int> intList = new(10);` - nastavená počiatočná kapacita

`Capacity` vs `Count`

Pridávanie prvkov

pri inicializácii

```
List<int> intList = new() {1, 2};
List<int> intList = new(new int[] {1, 2});
```

pomocou `Add`/`AddRange` - na koniec zoznamu

```
intList.Add(3);
intList.AddRange(new int[] {4, 5, 6});
```

pomocou `Insert`/`InsertRange` - na istú pozíciu

```
intList.Insert(3, 3);
```

Načítanie a vyhľadávanie

pristupovanie je umožnené cez indexy:

```
int x = intList[4];
```

`Exists` - `true`/`false` na základe prítomnosti prvku s istou vlastnosťou

`IndexOf`/`LastIndexOf` - vráti index konkrétneho objektu alebo -1

`FindIndex`/`FindLastIndex` - vráti index objektu s istou vlastnosťou

`Find`/`FindLast`/`FindAll` - vráti objekt s istou vlastnosťou

Odstránenie zo zoznamu

`RemoveAt` - podľa indexu (rýchlejšie)

```
intList.RemoveAt(3);
```

`Remove` - podľa objektu

```
intList.Remove(5);
```

`RemoveRange` - niekoľko objektov

```
intList.Remove(index, count);
```

`AsReadOnly` - kópia, z ktorej nemôžeme vymazať prvky, ani aktualizovať ich

Triedenie

metóda `Sort` (quicksort)

prvky musia byť porovnateľné (`IComparable`)

defaultné porovnávanie

vieme odovzdávať aj komparátor, ktorý môže implementovať iný spôsob porovnávania členov

Queues (fronty)

prvky sa spracúvajú na základe princípu first in, first out

vidíme iba začiatok a koniec frontu

`Queue<T>`

`Count` - počet prvkov
`Enqueue` - pridá prvok na koniec
`Dequeue` - vráti a vymaže prvok zo začiatku
`Peek` - vráti prvý prvok
`TrimExcess` - prispôsobí kapacitu

Stacks (zásobníky)

prvky sa spracúvajú na základe princípu last in, first out

vidíme iba vrch zásobníka

`Stack<T>`

`Count` - počet prvkov
`Push` - pridá prvok na vrch zásobníka
`Pop` - vráti a vymaže prvok z vrchu
`Peek` - vráti prvý prvok
`Contains` - vyhľadáva prvok

Linked lists (spojkové zoznamy)

obojsmerné prepojenia

rýchle pridávanie, pomalé vyhľadávanie

`LinkedList<T>` sa skladá z uzlov typu `LinkedListNode<T>`

uzol obsahuje:
- `List` - vráti zoznam, ktorého je súčasťou
- `Next` - smerník na ďalší uzol
- `Previous` - smerník na predošlý uzol
- `Value` - hodnota

Metódy LinkedList

`First/Last`

`AddAfter/AddBefore/AddFirst/AddLast`

`Remove/RemoveFirst/RemoveLast`

`Find/FindLast`

SortedList

oddeľuje hodnoty od kľúča triedenia

iba jedna hodnota na jeden kľúč

`SortedList<TKey, TValue>` sa skladá z `KeyValuePair` objektov

`KeyValuePair` obsahuje:

- `Key` - kľúč
- `Value` - hodnota

Dictionaries (mapy)

prístup k hodnotám cez ľubovoľný kľúč

rýchle čítanie pomocou hashovania

```
Dictionary<int, string> dict = new()
{
    [3] = "three",
    [7] = "seven"
};
```

kapacita by malo byť prvočíslo

Hashovateľnosť

rovnakému objektu musí byť vždy pridelená rovnaká hash hodnota

rôzne objekty môžu mať rovnakú hash hodnotu

nemôže generovať chyby

má využiť aspoň jednu členskú premennú

hash hodnota sa nemá meniť počas života objektu

vlastná definícia cez metódu `GetHashCode` (ideálne rýchle vykonávanie a rovnomerné rozdelenie na vybranom intervale čísel `int`)

Metódy Dictionary

```
Add(TKey, TValue)/TryAdd(TKey, TValue)

TryGetValue(TKey, TValue)

Clear()

ContainsKey(TKey)/ContainsValue(TValue)

Remove(TKey)/Remove(TKey, TValue)
```

Lookup<TKey, TElement>

v rámci `System.Linq`

viaceré hodnoty pod jedným kľúčom

nevieme vytvoriť priamo inštanciu, musíme zavolať `ToLookup`

generovanie z rôznych kolekcií, napríklad zo zoznamov

SortedDictionary<TKey, TValue>

binárny vyhľadávací strom, kde hodnoty sú zotriedené na základe kľúča

kľúč musí implementovať `IComparable`, alebo potrebujeme zadefinovať `IComparer`

podobný ako `SortedList`, avšak vyžaduje viac pamäte, ale rýchlejšie vykonáva pridávanie a vymazanie

Sets

množina jedinečných hodnôt

funkcionalita: únia, prienik, určenie podmnožín a nadmnožín

dva varianty:

- `HashSet<T>`
- `SortedSet<T>`

Výkonnosť

Kolekcia	**Pridanie na koniec**	**Pridávanie**	**Načítanie**	**Vyhľadávanie**	**Odstránenie**
List	*O(1) / O(n)*	*O(n)*	*O(1)*	*O(n)*	*O(n)*
Stack	*O(1) / O(n)*	*n/a*	*n/a*	*n/a*	*O(1)*
Queue	*O(1) / O(n)*	*n/a*	*n/a*	*n/a*	*O(1)*
HashSet	*O(1) / O(n)*	*O(1) / O(n)*	*n/a*	*n/a*	*O(1)*
SortedSet	*O(1) / O(n)*	*O(1) / O(n)*	*n/a*	*n/a*	*O(1)*
LinkedList	*O(1)*	*O(1)*	*n/a*	*O(n)*	*O(1)*
Dictionary	*O(1) / O(n)*	*n/a*	*O(1)*	*n/a*	*O(1)*
SortedDictionary	*O(log n)*	*n/a*	*O(log n)*	*n/a*	*O(log n)*
SortedList	*O(log n) / O(n)*	*n/a*	*O(log n) / O(n)*	*n/a*	*O(n)*

otázky?