Hodnotové a referenční typy

Kde se ukládají proměnné při běhu programu?
Proměnné se mohou ukládat na haldě nebo na zásobníku.
Zásobník

- je opravdu zásobník (last in first out datová struktura)
- přístup je rychlý
- není možné si vybrat kam se co uloží (ukládá se na vršek)

Halda

- přístup je pomalý
- ukládá se nejen na konec

Na zásobníku bydlí hodnotové typy (value types)

- čísla - int, long, uint, decimal..
- boolean, char...
- struct
- odkazy na referenční typy

Na haldě bydlí referenční typy

- string
- pole
- další vlastní třídy a objekty

Proč to tak je?

O referenčních typech se obecně předpokládá, že mohou být velké nebo se jejich velikost mění (např prodlužujeme string či pole, uložíme si do proměnné typu Object nějakou obludnost atd...). V proměnné referenčního typu ve skutečnosti na zásobník jen uložíme adresu, která říká "tady na tom místě na haldě leží tenhle objekt".

Hodnotové typy naopak bývají malé a jednoduché, práce s nimi je intenzivní (čísla většinou často zvětšujeme, zmenšujeme, sčítáme atd.. Referenční typy dost často jen někam přiřazujeme, nepotřebujeme přistupovat ke všem jejich částem naráz atd..)

Stringy jsou sice referenční typy, ale chovají se jako hodnotové, protože se v jistém smyslu jedná o základní typ, používá se podobně často jako hodnotové typy atd..

Jak se chovají

Doporučuji vyzkoušet si zdrojáky ke cvičení (a případně i videa). Oba druhy se chovají různě při:

- kopírování
- upravování v nějaké metodě, kam je předáme jako parametr
- upravování kopií

Kopírování

Jaká hodnota je v a a b na konci programu?

1. Čísla

```
int a = 8; //na tomto řádku se vytvoří nový chlíveček na int a zapíše se tam 8
int b = a; //na tomto řádku se vytvoří nový chlíveček na int a zapíše se tam 8

a = 4; //v chlívečku pro a se změní číslo na 4, chlíveček b s číslem 8 leží
jinde a nikdo po něm nic nechce
```

Výsledek:

```
a = 4
b = 8
```

2. Stringy

```
string s = "retezec1"; //vytvořím si nový chlíveček a uložím tam "retezec1"
string t = s; //vytvořím si nový chlíveček a uložím tam "retezec1"
s = "uplnenecojineho"; //do chlívečku s uložím "uplnenecojineho", na chlívečke
t se nikdo neptá
t = t + "retezec2"; //do chlívečku t uložím "retezec1retezec2", na chlíveček s
se nikdo neptá
```

Výsledek:

```
s = "uplnenecojineho"
t = "retezec1retezec2"
```

3. cokoliv jiného, třeba pole

```
int[] pole1 = new int[] { 4, 4 };
// na předchozím rřádku jsem vytvorila chlivecky na pole intu delky 2 a mám
tam čtyřky
```

```
// v pole1 si pamatuji, kde ty chlívečky leží
int[] pole2 = pole1;
// na předchozím řádku jsem si vytvořila novou proměnnou pole2 a do ní jsem
uložila kde leží chlívečky se čtyřkami, na které ukazuje pole1

pole1[0] = 1;
pole1[1] = 1;

//chlívečky na adrese, kterou si pamatuje pole1, jsem přepsala na 1
// to je stejná adresa, jakou si pamatuje pole2!!
//tj. např pole2[0] = 1

pole2[0] = 2;
pole2[1] = 2;

//chlívečky na adrese, kterou si pamatuje pole2, jsem přepsala na 2
// to je stejná adresa, jakou si pamatuje pole1!!
// tj. např pole1[0] = 2
```

Pro složitější příklady viz zdrojáky a cvičení

Předávání do metody

1. čísla a stringy

Když předáme metodě jako parametr číslo či string a něco s ním uděláme, po ukončení metody se to nezmění.

Metoda si vytvoří vlastní kopii předávaného parametru, tu upraví a na konci běhu metody smaže.

```
public void mojeMetoda(int a)
{
        a = a + 6;
}
```

```
int x = 9;
mojeMetoda(x)
```

Výsledek: x je stále 9.

2. referenční typy, třeba pole

```
public int mojeMetoda(int[] pole)
{}
```

Klíčová slova ref a out

Abyste donutili hodnotové typy chovat se referenčně (tj třeba aby změny provedené v metodě přepsali původní proměnnou), můžete použít klíčové slovo `ref`:

```
public void mojeMetoda(ref int a)
{
        a = a + 6;
}
```

```
int a = 9;
mojeMetoda(ref a);

//nyní je v proměnné a uložena hodnota 15
```

V podstatě stejně se používá klíčové slovo `out`, až na nějaké další problémy. Například lze použít i v případě, že předávaná proměnná ještě není inicializovaná, takže jí musíte v metodě vždy znova přiřadit nějakou hodnotu atd..

Další špíčky

Pokud chcete obejít problém s tím, že v případě kopírování polí máte jen dva ukazatele (pointery) na stejné pole, můžete použít metodu `copy` :

```
static void kopirovanaPole()
        {
            int[] pole1 = new int[] { 0, 0 };
            int[] pole2 = new int[pole1.Length];
            pole1.CopyTo(pole2, 0);

            Console.WriteLine("[{0}]", string.Join(", ", pole1));
            Console.WriteLine("[{0}]", string.Join(", ", pole2));

            pole1[0] = 1;
            pole1[1] = 1;

            Console.WriteLine("[{0}]", string.Join(", ", pole1));
            Console.WriteLine("[{0}]", string.Join(", ", pole2));
        }
```

Nyní se chová kopírování hezky, tj změny v poli1 mění opravdu jen pole 1.

Problém je, že toto je takzvaná **shallow copy** (narozdíl od **deep copy**). V případě že máme referenční typ obsahující další referenční typy, tak se sice vytvoří nový objekt, ale ostatní referenční položky schované uvnitř se nekopírují!

```
        static void polePoli()
         {
             int[] vnitrni1 = new int[] { 0, 0 };
             int[] vnitrni2 = new int[] { 0, 0 };

             int[][] pole1 = new int[][] { vnitrni1, vnitrni2 };
             int[][] pole2 = new int[pole1.Length][];
             pole1.CopyTo(pole2, 0);


             vypisPolePoli(pole1);
             vypisPolePoli(pole2);

             pole1[0][0] = 1;
             pole1[1][0] = 1;

             vypisPolePoli(pole1);
             vypisPolePoli(pole2);
```

Při kopírování se vytvořily nové chlívečky, ale do nich se v obou polích uložily stejné adresy - odkazující na místo kde leží *vnitrní1* a *vnitrni2*.