DATOVÉ TYPY – POKRAČOVÁNÍ

Hodnotové typy – pokračování

Znaky

Pro práci se znaky slouží typ `char`. Jedná se o dvoubajtový typ a uchovává znaky v kódování UNICODE. Je tedy ekvivalentem typu `wchar_t` v jazyce C++.

Typ	Struktura	Velikost [byte]	Hodnoty
`char`	`System.Char`	2	znak v kódování UNICODE

Typ `char` se chová jako celočíselný typ bez znaménka. Jeho hodnoty lze implicitně konvertovat na typ `ushort` a na všechny čtyřbajtové a osmibajtové celočíselné typy, dále na reálné typy a typ `decimal`.

Znakové hodnoty lze používat ve výrazech jako celá čísla – jsou pro ně definovány tytéž operátory jako pro celá čísla.

Žádný z číselných typů nelze implicitně konvertovat na typ `char`, lze však použít explicitní přetypování.

Znakové konstanty lze zapsat:

- mezi dva apostrofy, např. `'A'` – dtto C++,
- pomocí řídicí posloupnosti (escape characters),
- ve tvaru `'\uHHHH'` nebo `'\xHHHH'`, kde `HHHH` hexadecimální kód UNICODE.

Řídicí posloupnosti jsou stejné jako v jazyce C/C++. Jejich přehled je v následující tabulce.

Znak	UNICODE	Význam
`\"`	`\u0022`	Uvozovky
`\'`	`\u002C`	Apostrof
`\\`	`\u005C`	Obrácené lomítko
`\n`	`\u000A`	Nový řádek
`\r`	`\u000D`	Návrat na počátek řádku
`\t`	`\u0009`	Tabulátor
`\v`	`\u000B`	Vertikální tabulátor
`\b`	`\u0008`	Návrat o jeden znak
`\f`	`\u000C`	Nová stránka
`\a`	`\u0007`	Zvukový signál
`\0`	`\u0000`	Znak s kódem 0

Reálná čísla

Existují 2 reálné typy.

Typ	Struktura	Velikost [byte]	Rozsah	Přesnost
`float`	`System.Single`	4	±1.5e−45 až ±3.4e38	7
`double`	`System.Double`	8	±5.0e−324 až ±1.7e308	15 – 16

Tyto typy vyhovují specifikaci IEEE 754, takže umožňují pracovat i se speciálními hodnotami NaN (Not a Number), kladné a záporné nekonečno, pro které v datových typech `float` a `double` existují konstanty `NaN`, `PositiveInfinity`, `NegativeInfinity` a statické metody `IsNaN`, `IsInfinity`, `IsPositiveInfinity`, `IsNegativeInfinity`. Kladné/záporné nekonečno je výsledkem dělení kladného/záporného čísla nulou. Kladné nekonečno je dále výsledkem přetečení, záporné nekončeno je výsledkem podtečení.

Příklad

```
class Program
{
   static void Main(string[] args)
   {
      float f = float.MaxValue;
      float g = f * 2; // výsledkem přetěčení je kladné nekonečno
      if (g == float.PositiveInfinity) Console.WriteLine("Přetečení");
      f = float.MinValue;
      g = f * 2; // výsledkem podtečení je záporné nekonečno
      if (g == float.NegativeInfinity) Console.WriteLine("Podtečení");
      Console.ReadKey();
   }
}
```

Výstup programu bude následující:

```
Přetečení
Podtečení
```

Hodnota NaN je výsledkem operace, která nemá smysl, např. nekonečno minus nekonečno, dělení nula/nula. Je-li jedním z operandů relačních nebo aritmetických operátorů (včetně operátoru `==`) hodnota NaN, výsledkem je opět hodnota NaN. Pro zjištění, zda proměnná `a` typu `float` obsahuje hodnotu NaN nelze použít výraz `(a == float.NaN)`, protože jeho výsledkem je vždy hodnota NaN. Musí se použít výraz `float.IsNaN(a)`.

Reálné konstanty se zapisují stejně jako v C++ – v semilogaritmickém tvaru nebo s pevnou desetinnou tečkou. Za desetinou tečkou se musí uvést alespoň jedna číslice. Před desetinnou tečkou nikoliv.

- správné konstanty: `1.0`, `0.1` a `.1`
- nesprávná konstanta: `1.` (v C++ lze).

Reálná konstanta je automaticky typu `double`. Přípony:

- `F` nebo `f` – typ `float`
- `D` nebo `d` – typ `double`.

Pro reálná čísla jsou definovány následující operátory:

- unární aritmetické operátory `+`, `–`
- binární aritmetické operátory `+`, `-`, `*`, `/`, `%`
- operátory inkrementace a dekrementace `++` a `--`

- relační operátory <, >, <=, >=, ==, !=.

Navíc v C# lze pro reálná čísla použít operátor zbytku po dělení `%`.

Konverze z hodnoty typu `float` na typ `double` je implicitní. Opačná konverze se musí předepsat explicitně.

Nejčastěji používané matematické funkce a konstanty jsou deklarovány jako statické metody a statické datové složky třídy `System.Math`.

Třída `System.Math` obsahuje následující konstanty:

- `E` – Eulerovo číslo (cca 2,72)
- `PI` – Ludolfovo číslo (cca 3,14).

Některé matematické funkce jsou přetíženy i pro celočíselné typy, jako např. `Max`, `Min`. Některé jsou určeny pouze pro práci s reálnými čísly, jako např. `Sin` (sinus), `Log` (logaritmus).

Pro generování pseudonáhodných čísel slouží třída `System.Random`.

Desítková čísla

Datový typ `decimal` neexistuje v C++. Slouží pro finanční výpočty – číslo je uloženo v desítkové soustavě.

Typ	Struktura	Velikost [byte]	Rozsah	Přesnost
`decimal`	`System.Decimal`	16	±1.0 × 10e–28 až ±7.9 × 10e28	28 - 29

Např. číslo 0,1 je v dvojkové soustavě periodické (má tvar 0,00011001100...) a v proměnné typu `float` nebo `double` ho nelze uchovat bez zaokrouhlení. V proměnné typu `decimal` se uloží zcela přesně.

Typ `decimal` neumožňuje pracovat se speciálními hodnotami NaN, kladné a záporné nekonečno.

Konstanty se zapisují jako reálná čísla s příponou `M` nebo `m`.

Pro desítková čísla jsou definovány stejné operátory jako pro reálná čísla.

Proměnné typu `decimal` lze přiřadit hodnotu libovolného celočíselného typu bez přetypování. Konverze reálných typů na typ `decimal` a naopak je nutné provést explicitním přetypováním.

Logické hodnoty

Logické hodnoty `true` a `false` jsou uchovávány v typu `bool`.

Typ	Struktura	Velikost [byte]	Hodnoty
`bool`	`System.Boolean`	1	`true`, `false`

Na rozdíl od C++ neexistuje implicitní ani explicitní konverze mezi celočíselnými typy a typem `bool`.

Pro logické hodnoty jsou definovány následující operátory:

- konjunkce s neúplným vyhodnocením `&&`
- disjunkce s neúplným vyhodnocením `||`
- negace `!`
- konjunkce s úplným vyhodnocením `&`

- disjunkce s úplným vyhodnocením `|`
- nonekvivalence `^` – výsledek je pravda, mají-li operandy různou hodnotu
- relační operátory `!=` a `==`.

Na rozdíl od C++ nejsou definovány relační operátory <, >, <= a >=. Navíc však v C# existují operátory konjunkce a disjunkce s úplným vyhodnocením.

Např. výraz

```
(x >= 0.0) & (Math.Sqrt(x) < y)
```

může způsobit běhovou chybu, protože se v něm vždy vyhodnotí oba operandy a pro záporné `x` není druhá odmocnina definována (funkce `Sqrt` vrátí NaN). Naproti tomu výraz

```
(x >= 0.0) && (Math.Sqrt(x) < y)
```

problémy nezpůsobí, neboť pro záporná `x` se druhý výraz nevyhodnotí.

Typ void

Klíčové slovo `void` nevyjadřuje plnohodnotný datový typ. Lze ho použít pouze jako typ návratové hodnoty metody.

Na rozdíl od C++ nelze slovo `void` použít ke specifikaci prázdného seznamu formálních parametrů ani k přetypování. Následující použití jsou chybná:

```
int f(void) { }
(void)f();
```

Typu `void` odpovídá struktura `System.Void`.

Výčtové typy

Syntaxe:

deklarace výčtového typu:

modifikátor$_{nep}$ **enum** *jméno bázový_typ*$_{nep}$ **{** *seznam_výčtových_konstant* **}** **;**$_{nep}$

bázový_typ:

: *celočíslený_typ*

Modifikátor – přístupová práva

Jméno – identifikátor výčtového typu

Celočíselný_typ – jeden z typů uvedených v kapitole „Celá čísla".

Bázový typ vyjadřuje datový typ, který se má použít pro ukládání výčtových konstant. Pokud se vynechá, použije se typ `int`.

Výčtové konstanty mohou obsahovat i definici hodnoty. Středník za deklarací je nepovinný.

Např.

```
enum Dny { Pondělí, Úterý, Středa, Čtvrtek, Pátek, Sobota, Neděle };
```

Tento výčtový typ má 7 konstant, proto se vejde do typu `byte`. Deklaraci lze upravit následovně

```
enum Dny:byte { Pondělí, Úterý, Středa, Čtvrtek, Pátek, Sobota, Neděle };
```

Na rozdíl od C++ se musí výčtové konstanty při použití kvalifikovat jménem výčtového typu. Musí se např. psát `Dny.Pondělí`.

Konstanta `Pondělí` má hodnotu 0, `Úterý` 1 atd.

Výčtovým konstantám lze přiřadit hodnotu jako v C++, např.

```
enum Dny : byte { Pondělí = 1, Úterý, Středa, Čtvrtek, Pátek, Sobota,
Neděle = 0 };
```

Konstanta `Úterý` má hodnotu 2, `Středa` 3 atd.

Pro výčtové typy jsou definovány operátory:

- relační operátory <, >, <=, >=, ==, !=
- bitové operátory ~, ^, &, |
- operátory inkrementace a dekrementace ++ a --.

Operátor inkrementace zvýší hodnotu výčtového typu numericky o 1, nikoli na následující výčtovou konstantu. Obdobně funguje operátor dekrementace.

Pro převod hodnoty jednoho výčtového typu na jiný výčtový typ, převod hodnoty výčtového typu na celé číslo nebo naopak se musí použít explicitní konverze. Výjimkou je hodnota 0, kterou lze implicitně konvertovat na hodnotu jakéhokoli výčtového typu.

K deklaraci výčtového typu lze připojit atribut `Flags`. Překladač se tím upozorňuje, že se konstanty budou používat jako příznaky, které se budou kombinovat pomocí bitových operátorů, a že s tímto výčtovým typem má podle toho zacházet v ladících nástrojích a při výstupu na konzolu.

Pokud by se výčtové konstanty dnů měly chovat jako příznaky, výčtový typ by vypadal takto

```
[Flags]
enum Dny { Pondělí = 0x01, Úterý = 0x02, Středa = 0x04, Čtvrtek = 0x08,
Pátek = 0x10, Sobota = 0x20, Neděle = 0x40 };
```

Cyklus

```
for (Dny d = Dny.Pondělí; d < Dny.Středa; d++) {
   Console.WriteLine(d);
}
```

vypíše následující texty

```
Pondělí
Úterý
Pondělí, Úterý
```

Pro hodnotu 3 vypsal program text `Pondělí, Úterý`, který vznikl jako výsledek operace `Dny.Pondělí | Dny.Úterý`.

Pokud se vynechá atribut `[Flags]`, uvedený cyklus vypíše tyto texty:

```
Pondělí
Úterý
3
```

Pro uložení kombinace příznaků do proměnné se používá operátor bitové disjunkce |. K zjištění, zda proměnná obsahuje zadaný příznak se používá operátor bitové konjunkce &. Příkazy

```
Dny a = Dny.Pátek | Dny.Neděle;
Console.WriteLine(a);
if ((a & Dny.Pátek) == Dny.Pátek)
   Console.WriteLine("Proměnná obsahuje příznak " + Dny.Pátek);
```

způsobí následující výstup:

```
Pátek, Neděle
Proměnná obsahuje příznak Pátek
```

Pro výčtový typ deklarovaný s atributem `[Flags]` se doporučuje deklarovat i výčtovou konstantu s hodnotou nula s názvem `None`. Tuto konstantu lze použít ve výrazech pro zjištění, zda daná proměnná výčtového typu obsahuje nějaký příznak. Např.

```
[Flags]
enum Dny { None, Pondělí = 0x01, Úterý = 0x02, Středa = 0x04, Čtvrtek =
0x08, Pátek = 0x10, Sobota = 0x20, Neděle = 0x40 };

if (a == Dny.None)
   Console.WriteLine("Proměnná neobsahuje žádný příznak");
```

Pro výčtový typ deklarovaný bez atributu `[Flags]` se doporučuje deklarovat výčtovou konstantu s hodnotou nula, která bude reprezentovat implicitní hodnotu. Pokud daný výčtový typ nemá zřejmou implicitní výčtovou konstantu, doporučuje se zavést výčtovou konstantu `None` s hodnotou nula. Je to proto, že datové složky struktury a třídy jsou inicializovány hodnotou nula a pokud by datová složka výčtového typu neobsahovala nulovou výčtovou konstantu, měla by po inicializaci nedefinovanou hodnotu.

Struktury

V C++ je rozdíl mezi strukturami (`struct`) a třídami (`class`) minimální – odlišné pojetí implicitních přístupových práv pro složky a při dědění. V C# je rozdíl mezi strukturami a třídami větší.

Syntaxe:

deklarace struktury:

modifikátor$_{nep}$ **struct** *jméno specifikace_rozhraní*$_{nep}$ **{** *složky*$_{nep}$ **}** **;**$_{nep}$

Modifikátor – přístupová práva.

Jméno – identifikátor datového typu dané struktury, který tato deklarace zavádí.

Specifikace_rozhraní – rozhraní, která tato struktura implementuje (rozhraní – viz dále).

Složky – seznam složek struktury. Může se jednat o datové složky, metody, přetížené operátory, konstruktory, vlastnosti, události a vnořené typy.

V C++ se pro datové složky struktury a třídy používá termín atribut. V C# se ale pod označením atribut rozumí něco jiného – jakási doplňující specifikace vlastností datového typu, složky struktury nebo třídy apod. Slovo atribut se proto v těchto přednáškách bude používat pouze ve smyslu, v jakém ho zavádí C#, a pro datové složky se bude používat pouze pojem *datové složky*. V normě C# se datová složka nazývá *field*.

Deklarace struktury se na rozdíl od C++ nemůže objevit v metodě (lokální typ), ale jen v prostoru jmen (včetně globálního) nebo uvnitř třídy nebo struktury (vnořený typ).

Rozdíly mezi třídami a strukturami:

- Struktury patří mezi hodnotové typy, třídy mezi referenční typy. Hodnotové typy se vytvářejí v zásobníku, referenční na haldě.
- Všechny struktury (včetně struktur vestavěných hodnotových typů, např. `int`) jsou potomkem třídy `System.ValueType`, která je odvozena od třídy `object`. V deklaraci nelze specifikovat předka a od struktury nelze odvodit potomka.
- Struktury nemohou mít destruktor.

- Struktury nemohou mít uživatelem definovaný konstruktor bez parametrů.
- Konstruktor struktury musí inicializovat všechny její datové složky.

Překladač pro strukturu vždy vytvoří konstruktor bez parametrů, tzv. *implicitní konstruktor* (angl. *default constructor*), který inicializuje všechny datové složky hodnotou 0 případně `false` nebo `null` (viz dále).

Instance struktury lze vytvořit dvěma způsoby:

- pomocí operátoru `new`,
- inicializací všech datových složek struktury.

Např. je dána struktura `Bod` s následující deklarací:

```
struct Bod
{
   public double x, y;
   public Bod(double x, double y)
   {
      this.x = x;
      this.y = y;
   }
}
```

Struktura `Bod` obsahuje pouze veřejně přístupný konstruktor se dvěma parametry a veřejně přístupné datové složky `x` a `y`. Novou instanci pomocí operátoru `new` lze vytvořit jedním z uvedených způsobů:

```
Bod a = new Bod(); // a.x = 0, a.y = 0
Bod b = new Bod(30, 40); // b.x = 30, b.y = 40
```

V prvním případě se volá konstruktor bez parametrů v druhém případě konstruktor s parametry. Hodnota např. složky `x` bodu `a` se získá zápisem `a.x`.

Identifikátory `a`, `b` nejsou ukazatelé (operátor `new` u struktury nepředstavuje dynamickou alokaci paměti). Jedná se o instance. Pokud se vytvoří instance bodu `c` zápisem

```
Bod c = b;
```

a provede se přiřazení

```
c.x = 12;
```

změní se složka `x` proměnné `c`, nikoli složka `x` proměnné `b`.

Při přiřazení jedné instance struktury do jiné dojde ke zkopírování všech datových složek struktury. Zápisem

```
Bod d; // nevolá se konstruktor
int u = d.x; // Chyba
```

se vytvoří neinicializovaná proměnná `d`. Žádný konstruktor se nezavolá. Složky instance `d` nejsou inicializovány a nelze je tudíž použít.

Datové složky struktury lze inicializovat i jednotlivě po vytvoření instance bez volání konstruktoru. Datovou složku lze však použít až po její inicializaci:

```
Bod e;
e.x = 10;
```

```
e.y = 20;
int v = e.x + e.y; // OK
```

Nulovatelné typy

V databázích mohou datové typy obsahovat i hodnotu NULL (žádná hodnota). V jazyce C# standardní hodnotové typy nemohou obsahovat hodnotu `null`.

```
bool a = null; // chyba - nelze přiřadit null
```

Z důvodu lepší práce s databázemi, byly od verze .NET 2.0 zavedeny hodnotové typy, které mohou obsahovat hodnotu `null`, tzv. *nulovatelné typy* (*nullable types*).

Nulovatelný typ může reprezentovat všechny hodnoty svého podkladového typu plus hodnotu `null`.

Syntaxe:

nulovatelný_typ:

nenulovatelný_typ **?**

Nenulovatelný_typ – jakýkoliv hodnotový typ kromě nulovatelného typu. Jedná se o tzv. *podkladový typ* (*underlying type*).

Nulovatelný typ je ve skutečnosti generická struktura `System.Nullable<T>`, kde `T` reprezentuje podkladový hodnotový typ. Následující deklarace proměnné jsou ekvivalentní.

```
int? a = null;
System.Nullable<int> a = null;
```

Struktura nulovatelného typu kombinuje hodnotu podkladového typu s logickou hodnotou, indikující, zda instance nulovatelného typu obsahuje hodnotu `null`. Nulovatelný typ má dvě vlastnosti určené pouze pro čtení:

- `bool HasValue` – je `true`, pokud instance nulovatelného typu neobsahuje hodnotu `null`.
- `T Value` – hodnota podkladového typu. Pokud instance nulovatelného typu obsahuje hodnotu `null` (vlastnost `HasValue` má hodnotu `false`), čtení hodnoty této vlastnosti vyvolá výjimku `System.InvalidOperationException`.

Libovolný nenulovatelný typ lze implicitně konvertovat na jeho odpovídající nulovatelný typ. Opačnou konverzi lze provést explicitním přetypováním. Pokud instance nulovatelného typu obsahuje hodnotu `null`, při explicitní konverzi na podkladový typ vznikne výjimka `System.InvalidOperationException`. Např.

```
int? a = 10;
int? b = null;
int c = (int)a;
int d = (int)b; // Výjimka System.InvalidOperationException
if (a.HasValue) Console.WriteLine(a); // vypíše hodnotu
if (b.HasValue) Console.WriteLine(b); // nevypíše hodnotu
```

Pokud je definována konverze (implicitní nebo explicitní) mezi podkladovými typy, je definována i mezi jejich nulovatelnými typy. Hodnota `null` jednoho typu se konvertuje na hodnotu `null` druhého typu. Jiná hodnota jednoho typu se konvertuje na odpovídající hodnotu druhého typu pomocí předdefinované konverze mezi podkladovými typy. Např.

```
int? a = 10;
int b = 20;
double? c = 10.5;
double? d = a; // int? --> double?
```

```
double? e = b; // int --> double?
int? f = c; // Chyba: double? -> int?
int? g = (int?)c; // double? -> int?
```

Pokud je definován nerelační operátor pro nenulovatelný typ, existuje tento operátor i pro jemu odpovídající nulovatelný typ. Pokud jeden z operandů má hodnotu `null`, je výsledkem takového operátoru hodnota `null`, jinak je výsledkem odpovídající operace provedená na podkladovém typu (u unárního operátoru) resp. typech (u binárního operátoru). Např.

```
int? a = 10;
int? b = 20;
int? c = null;
int? ab = a + b; // #1 ab = 30
int? ac = a + c; // ac = null
```

Příkaz #1 odpovídá příkazu

```
int? ab = a.HasValue && b.HasValue ? a.Value + b.Value : (int?)null;
```

Protože existuje implicitní konverze z nenulovatelného typu na jemu odpovídající nulovatelný typ, lze použít binární operátor, který je definován pro nenulovatelný typ, pro jeden operand nenulovatelného typu a druhý operand nulovatelného typu. Výsledkem je hodnota nulovatelného typu. Např.

```
int? a = null;
int? b = 10;
int? c =  a + 1; // c = null
int? d = b + 1; // d = 11
```

Pro relační operátory (==, !=, <, >, <=, >=) definované pro nenulovatelný typ, jejichž oba operandy jsou tohoto nenulovatelného typu a návratová hodnota je typu `bool` se převádějí pro jemu odpovídající nulovatelný typ následovně.

- Operátory ==, !=: hodnoty `null` se rovnají, hodnota `null` se nerovná hodnotě různé od `null`. Na hodnoty operandů různé od `null` se použije operátor pro podkladové typy.
- Operátory <, >, <=, >=: pokud jeden nebo oba operandy mají hodnotu `null`, výsledkem je hodnota `false`, jinak je výsledkem hodnota operátoru pro podkladové typy.

Hodnotu nulovatelného typu lze porovnávat s konstantou `null` pomocí operátorů == a !=. Při použití jiného relačního operátoru překladač oznámí varování, že hodnota výrazu je vždy `false`.

```
int? a = 10;
if (a == null) Console.WriteLine("a je null");
if (a < null) Console.WriteLine(a je menší než null); // nemá smysl
```

Pro nulovatelné typy lze použít *operátor nulového sjednocení* (*null coalescing operator*) `??`. Slouží k definování implicitní hodnoty výsledku v případě, kdy má nulovatelný typ hodnotu `null`. Výsledkem výrazu `(a ?? b)` je hodnota `a`, jestliže `a` není rovno `null`, jinak je výsledkem hodnota `b`. Hodnota výrazu je standardně podkladového typu operandu `a`. Operand `b` musí být podkladového typu operandu `a`, resp. typu, který lze implicitně konvertovat na podkladový typ operandu `a`. Přesná definice typu výsledku uvedeného výrazu je složitější – viz [4], kapitola 14.12.

Příklad

```
int? a = null;
int? b = 10;
int c = a ?? 0; // c = 0
int d = b ?? 0; // d = 10
```

Operátor nulového sjednocení také pracuje s referenčními typy.

Referenční typy

Referenční typy jsou:

- pole,
- třídy,
- delegáty,
- rozhraní.

Instance referenčních typů se vytváří pomocí operátoru `new` a pracuje se s nimi pomocí *referencí* (odkazů).

Reference neboli odkazy

Reference jsou vlastně ukazatele, které se automaticky dereferencují. V C++ lze prostřednictvím ukazatelů pracovat s proměnnými jakéhokoli typu. V C# lze pomocí referencí pracovat pouze s proměnnými referenčních typů. Podobně jako ukazatele v C++ jsou i reference vázány na určitý typ instance (nebo rozhraní).

Samotné reference jsou vlastně hodnotové typy, které zpřístupňují instance referenčních typů. Přiřazení reference do jiné reference je totéž co přiřazení ukazatele do jiného ukazatele – dvě reference se budou odkazovat na stejnou instanci.

Proměnné typu „reference na typ `T`“ lze přiřadit pouze hodnotu jednoho z těchto typů:

- „reference na typ `T`“
- „reference na potomka typu `T`“.

Z toho vyplývá, že proměnné typu „reference na typ `object`“ lze přiřadit referenci na instanci jakéhokoliv typu (včetně hodnotového typu – viz dále *zabalení*).

Reference, která neodkazuje na žádnou platnou instanci, obsahuje hodnotu vyjádřenou klíčovým slovem `null`.

Porovnávání referencí – pomocí operátorů `==` a `!=`. Avšak v třídě lze operátory `==` a `!=` přetížit tak, že potom mohou porovnávat obsah instancí dané třídy. Nelze porovnávat reference pomocí operátorů <, > atd. Na reference nelze aplikovat adresovou aritmetiku.

Zanikne-li poslední odkaz na instanci referenčního typu, automatická správa paměti se postará o její zrušení. Automatická správa paměti přitom zavolá metodu `Finalize()`, zděděnou po třídě `object`. To ovšem nemusí nastat ihned v okamžiku zániku poslední reference, ale kdykoli později. Nemusí k tomu dojít za celou dobu běhu programu.

Třídy

V této kapitole je uveden pouze stručný přehled. Podrobněji jsou třídy vysvětleny později.

Novou instanci třídy lze vytvořit pouze pomocí operátoru `new`, stejným způsobem jako u struktur.

Např.

```
class TridaBod
{
   public int x, y;
   public TridaBod(int x, int y)
   {
      this.x = x;
      this.y = y;
   }
}
```

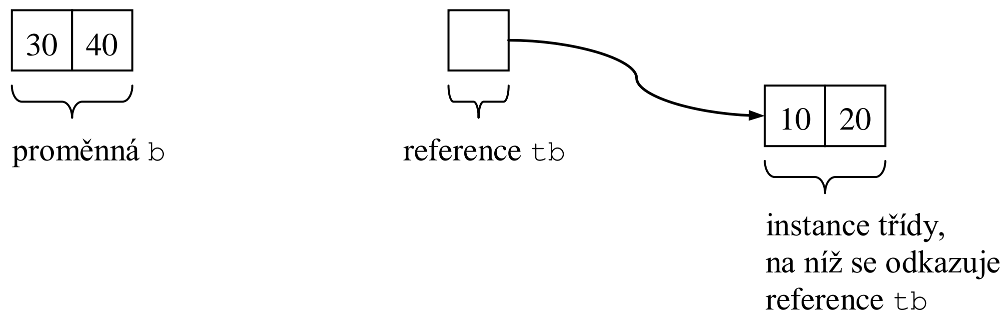
```
TridaBod tb = new TridaBod(10, 20);
```

Třída `TridaBod` obsahuje pouze veřejně přístupný konstruktor se dvěma parametry a veřejně přístupné datové složky `x` a `y`, podobně jako dříve deklarovaná struktura `Bod`. Proměnná `tb` přestavuje referenci na instanci třídy `TridaBod` vytvořenou na haldě voláním konstruktoru se dvěma parametry.

Deklarace s inicializací reference `tb` se podobá deklaraci

```
Bod b = new Bod(30, 40);
```

Rozdíl je v tom, že proměnná `b` označuje přímo místo v paměti, kde je uložena instance struktury `Bod`, zatímco proměnná `tb` označuje místo v paměti, kde je uložen odkaz na instanci třídy `TridaBod`. Viz následující obrázek.



Přiřazení referencí nezpůsobí kopírování instance. Totéž platí i pro použití reference v inicializaci jiné reference:

```
TridaBod tc = tb;
tc.x = 50; // dtto tb.x = 50
```

Reference `tc` a `tb` se odkazují na tutéž instanci třídy `TridaBod`.