**Datové typy, Generika, Výčtové datové typy, Struktury, Delegáti, Eventy**

*Datové typy jsou něco s čím pracujeme denně. Generika jsme tolik neopakovali, ale stačí jeden pohled a máte to. Myslím si, že pokud budete schopni dostatečně dlouho obkecávat tyhle dvě věci tzn. Jak a kdy je použít, tak se nemusíte učit ani moc kódu a na tu čtyřku to máte.*

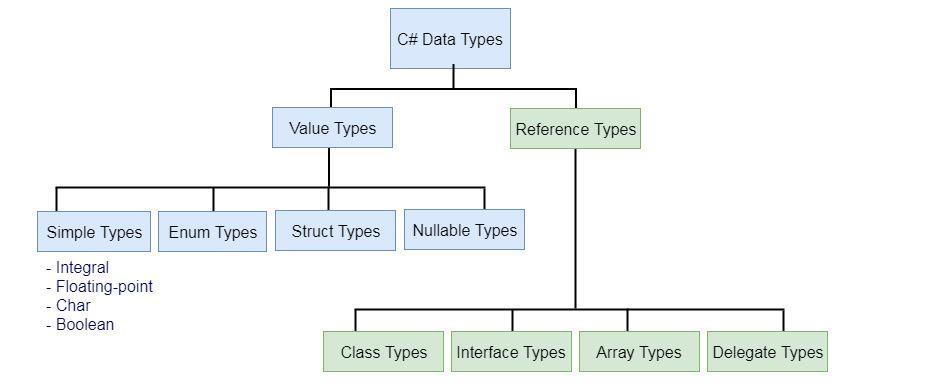
*Je to hodně jednoduchých témat, které jsou vzájemně provázané. Když budete prostě mluvit spojitě, snadno shrnete všechny a ještě se můžete opakovat.*

*Ve všech příkladech jsem používal viditelnost public, abych zbytečně neprotahoval kód gettery a settery.*

*Vypracoval jsem to pro C#, ale mělo by to být většinou stejné třeba i v javě.*

*Karel Muroň*

**Datové typy**



Úplně v základu je můžeme rozdělit na typy, do kterých se dosazuje hodnota a typy referenční, které obsahují referenci na haldu. Ty, které pak nejčastěji používáme se nazývají primitivní.

Je velice důležité vybírat vhodné datové typy. A nemyslím tím jenom triviality ve smyslu, že do intu nenacpu text. Vhodně vybraný datový typ může ušetřit spoustu místa v paměti, což je zásadní.

Např:

* Pro sčítání do dvaceti nebudu používat 64-bitový int (long) - tak velký rozsah zkrátka není potřeba a zbytečně zabírá místo v paměti.
* U metody je zbytečné používat jako návratový typ int, když může vracet jen 0/1. Stejně tak nemusím vracet celý string, když dopředu vím, že budu vracet jen jeden znak (char).

Druhy:

* Pro práci s **textovými řetězci či znaky** slouží char - právě jeden znak a string (de facto pole charů - viz. Samostatná otázka pro práci se stringy).
* Sám o sobě je tu **boolean**, který má dva stavy (true/false). I když je to vlastně jen jeden bit, **zabírá v paměti celý byte**!
* A nyní **čísla**. Je podstatné si promyslet:
  + Jaký rozsah potřebuji
  + Jestli potřebuji desetinné číslo
  + Jestli může být dané číslo záporné

Za největší základ se pokládá 32-bit int, pokud potřebujete desetinné číslo, pak float.

Nesmíme zapomenout, že operace float f = 1/10 **vrátí nulu**! Pokud chceme desetinné číslo musí být za operací písmeno f tzn.

float f = 1/10f

V případě double pak

double d = 1/10.0

Další dělení je možné dle plovoucí desetinné čárky či přesnosti. Na to už je ale otázka v TVy.

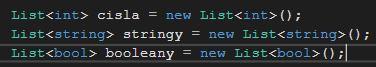
Přehled typů:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Represents** | **Range** | **Default Value** |
| bool | Boolean value | True or False | False |
| byte | 8-bit unsigned integer | 0 to 255 | 0 |
| char | 16-bit Unicode character | U +0000 to U +ffff | '\0' |
| decimal | 128-bit precise decimal values with 28-29 significant digits | (-7.9 x 1028 to 7.9 x 1028) / 100 to 28 | 0.0M |
| double | 64-bit double-precision floating point type | (+/-)5.0 x 10-324 to (+/-)1.7 x 10308 | 0.0D |
| float | 32-bit single-precision floating point type | -3.4 x 1038 to + 3.4 x 1038 | 0.0F |
| int | 32-bit signed integer type | -2,147,483,648 to 2,147,483,647 | 0 |
| long | 64-bit signed integer type | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | 0L |
| sbyte | 8-bit signed integer type | -128 to 127 | 0 |
| short | 16-bit signed integer type | -32,768 to 32,767 | 0 |
| uint | 32-bit unsigned integer type | 0 to 4,294,967,295 | 0 |
| ulong | 64-bit unsigned integer type | 0 to 18,446,744,073,709,551,615 | 0 |
| ushort | 16-bit unsigned integer type | 0 to 65,535 | 0 |

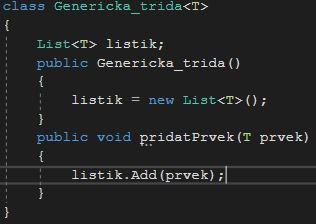
**Generika**

Nejjednodušší definicí generik je **“obecný datový typ”**. Když chceme dát uživateli volnost, aby si mohl vybrat libovolný datový typ, používají se právě generika. Zejména se tak hodí jako vstupní proměnné do metod či pro listy.

Př.: Běžně užíváme listy. Listy nemají předem nadefinovaný datový typ, ale používají generikum. Při vytváření listu, tak musíme přesně určit ve špičatých závorkách, co chceme ukládat:



Pro vytváření takové generické třídy se taktéž používají špičaté závorky. Třída pro práci s listem tak může vypadat třeba následovně:



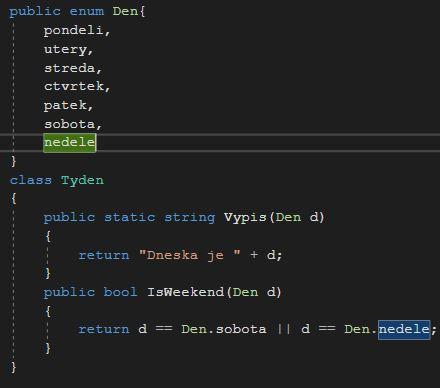
Tzn. Vytvořil jsem si generickou třídou pomocí špičatých závorek u jména třídy a dále nějakou příkladovou ukázku použití - vlastní třída pro práci s listem.

Obecně nezáleží, jaké písmeno do špičatých závorek zvolíme. Nejčastěji se používá T.

**Výčtové datové typy**

Výčtové datové typy nefungují jako ty klasické. Používáme je ve chvíli, kdy chceme dát na výběr z **pevně daných možností, které se nemění**. Neslouží tak pro dynamické ukládání hodnot, jako klasické proměnné.

Např.: Chci dát na výběr mezi dny v týdnu. Je jasné, že názvy i počet dnů je neměnný. V takovém případě se v C# použije typ enum:



Vytvořil jsem si metodu, která vypisuje den podle enumu. Enum stojí vždy **mimo samotnou třídu**.

V Mainu pak použití vypadá následovně:



Dá se taktéž enumy porovnávat či používat ve switch-case viz. IsWeekend().

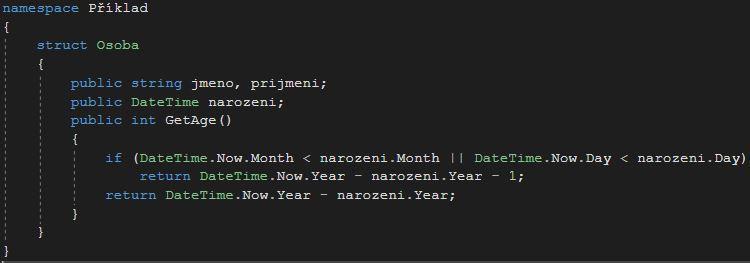
Jednotlivé prvky v enumu jsou očíslovány od 0 do n. Pokud chceme změnit číslování stačí napsat pondeli=1, čímž změníme index pondělí.

**Struktury**

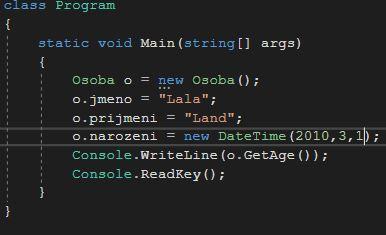
Struktury jsou podobné třídám. Používají se pro ukládání menších kolekcí dat, přičemž žádná z jeho vlastností nesmí být referenční datový typ. Na rozdíl od tříd:

* **Nejsou referenční datový typ**. Tzn. data se ukládají přímo na zásobníku => pokud nepotřebujeme používat metody dané struktury, nemusí se použít new při inicializaci
* Nemohou dědit od jiných tříd ani být děděny
* Nemůže se změnit konstruktor, který nemá vstupní parametry

Př. Struktury:



V mainu:

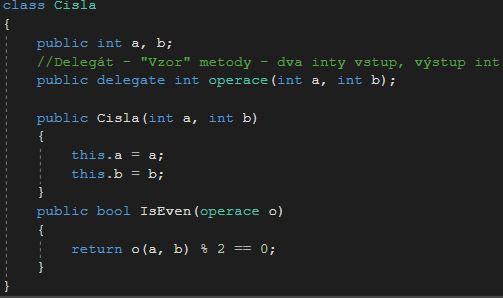


**Delegáti a Eventy**

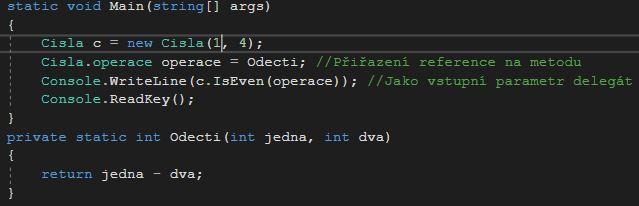
Delegát obsahuje **referenci metody**. Při jeho vytváření je nutné určit návratový typ a vstupní parametry metody. Když pak vkládáme referenci na metodu, metoda musí odpovídat tomuto vzoru. S využitím jsme se již setkali v případě vláken, kdy se odkazuje na námi vytvořenou metodu, aby byla spuštěna v samostatném vlákně.

Zkrátka, když chceme dát uživateli naší metody volnost, aby mohl využít libovolnou svoji metodu, jako vstupní parametr zvolíme delegát a s ním pak dále pracujeme.

Př. Chceme provést uživatelem vybranou funkci s čísly a zjistit, jestli je výsledek sudý:



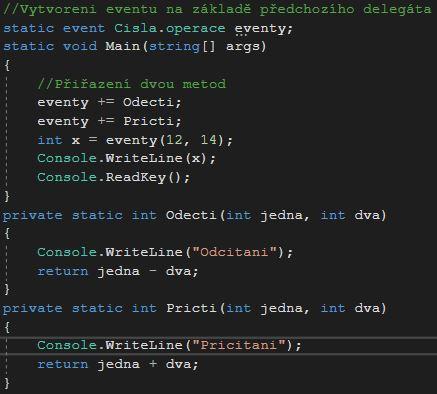
Do metody IsEven() vstupuje delegát operace, který pak volám o(a,b). To vrátí výsledek. Následně to porovná, jestli zbytek po dělení dvěma je nula.



Eventy fungují podobně jako delegáti. V eventu může být více metod. Když pak takový event zavolám jako metodu, spustí se všechny metody do něj vložené.

Často se s nimi můžeme setkat např. v Unity nebo jim přiřazujeme metody v okenních aplikacích (metoda, která se spustí po stisknutí tlačítka apod.).

Př.:



K vytvoření eventu **potřebujeme delegáta**. Toho jsem použil z předchozího příkladu. K eventu jsem pomocí += přidal metody, odpovídající delegátovi. Když pak takový event zavolám, spustí se po sobě metody, jak jsem je tam vložil. Tzn. V proměnné x bude uložený výsledek poslední metody (v tomto případě sčítání). Výpisy jsou tam jen pro ukázku, že metody proběhly obě. Konzole tedy vypadá takto:

