**Datové typy, Generika, Výčtové datové typy, Struktury, Anotace, Operátory**

**C# PŘÍKLADY !!!**

**Datový typ**

* Definuje druh nebo význam hodnoty (proměnná)
* Určen oborem hodnot a operacemi, které lze provádět

**Základní datové typy**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | **C#** |  |
| **Datový typ** | **Velikost** | **Hodnoty** | **Základní hodnota** |
| int | 4 byty | Celá čísla od -2 147 483 648 do 2 147 483 647 | 0 |
| long | 8 bytů | Celá čísla od  -9,223,372,036,854,775,808 do 9,223,372,036,854,775,807 | 0L |
| float | 4 byty | Desetinná čísla s 6 až 7 desetinnými místy | 0.0F |
| double | 8 bytů | Desetinná čísla s až 15 desetinnými místy | 0.0D |
| bool | 1 bit | Hodnota true/false | False |
| char | 2 byty | Jedno písmeno, ohraničené ‘ ’ | ‘\0’ |
| string | 2 byty/písmeno | Několik písmen ohraničené „ “ | null |
| decimal | 16 bytů | Desetinné číslo s 28-29 desetinnými místy | 0.0M |
| byte | 1 byte | Celá čísla od 0 do 255 | 0 |
| sbyte | 1 byte | Celá čísla od -128 do 127 | 0 |
| uint | 4 byty | Celá čísla od 0 do 4 294 967 295 | 0 |
| ulong | 8 bytů | Celá čísla od 0 do 18,446,744,073,709,551,615 | 0 |
| ushort | 2 byty | Celá čísla od 0 do 65 535 | 0 |

|  |  |  |
| --- | --- | --- |
|  | **Python** | |
| **Typy datových typů** |  | **Název** |
| Číselné datové typy |  | int, float, complex |
| Textové datové typy |  | str |
| Sekvenční datové typy |  | list, tuple, range |
| Mapové datové typu |  | dict |
| Datové typy set |  | set, frozenset |
| Datové typy boolean | | bool |
| Binární datové typy | | bytes, bytesarray, memoryview |

# Generické(obecné) datové typy

* Generika, neboli parametrizované datové typy.
* Důvodem jejich existence je zobecnění tříd, metod a interface pro více datových typů.
  + Toto umožňuje vývojáři vytvořit metodu která funguje například jak pro String, Int, …
* K zobecnění metod nebo tříd se samozřejmě dá použít datový typ „Object“
  + mohlo by se stát, že v průběhu metody jen tak změní svůj datový typ
  + Toto se s generiky stát nemůže protože při volání metody, nebo vytváření objektu ze třídy specifikujeme datový typ, který má používat.

## Konvence pojmenovávání generických typů

|  |  |
| --- | --- |
| **Pojmenování** | **Hodnota** |
| T | Typ (String, int, …) |
| E | Element |
| K | Key (například v dictionary) |
| N | Číslo |
| V | Hodnota |

## Typy Generik v jazyce Java|C#

* Generika v Jazycích Java a C# jsou prakticky identická
* **Generické Metody** o Generické metody, stejně jako normální metody mohou přijímat parametry, provedou nějaký úkol a poté vrátí nějakou hodnotu
* Generická hodnota má však i tzv. „Type parametr“, který určuje datový typ, se kterým má metoda pracovat.
  + Díky tomuto se metoda dá použít pro více datových typů bez toho aniž bychom ji pro každý datový typ museli psát znovu.
  + Příklad jednoduché generické metody:

Text

Description automatically generated

* + Tato metoda pouze vypíše datový typ vloženého parametru a vrátí jej

A screenshot of a computer

Description automatically generated with medium confidence

**Generické třídy** o Generická třída se implementuje stejným způsobem jako normální třída, tudíž vytvořením jejího objektu. o Jediným rozdílem je že má „parameter section“ (<>), toto může obsahovat více parametrů, oddělených čárkou.

* + Příkladem třídy s jedním parametrem je například ArrayList, zatímco s více parametry zase HashMap

Text

Description automatically generated

* + Příklad vytvoření Generické třídy:

Text

Description automatically generated

* + Vytváření objektu z generické třídy:

Text

Description automatically generated with medium confidence

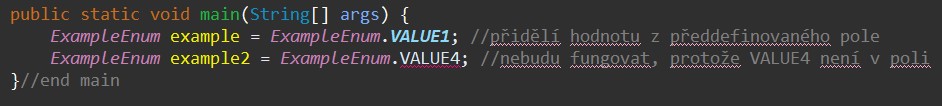
# Výčtové datové typy

* Výčtové datové typy, nebo také „enumy“, jsou datové typy, které umožňují dosadit jako sovou hodnotu jen hodnotu, která se nachází v předdefinovaném listu hodnot.
* Příklad vytvoření Enumu **v jazyce Java**:

Text

Description automatically generated

* Vytvoření proměnné z enumu:



Záznamu v poli se také dá přidělit jeho hodnota (poté je ale třeba vytvořit enumu konstruktor):

Text

Description automatically generated

* Hodnotu poté můžeme zobrazit:

Graphical user interface

Description automatically generated

* **V jazyce C#** je vytvoření enumu ve směs stejné:

Text

Description automatically generated

* Uložení hodnot do enumu je však jiné:

Text

Description automatically generated

* A vypsání hodnoty také:

Text

Description automatically generated

**V jazyce Python** enum nemá svůj vlastní datový typ, ale je třeba vytvořit třídu, která dědí od enum.Enum a do ní poté zadat požadované hodnoty. Těmto hodnotám je na rozdíl od Javy a C# povinné přidělit číselné hodnoty:

Text

Description automatically generated

* Vypsání je poté podobné jako v ostatních jazycích:

Text

Description automatically generated

# Struktury

## Rozdělení

### Primitivní datové struktury

* Základní datové struktury, používané pro základní operace.
* Příklady: Integer, Float, String, …

### Složité datové struktury

* Komplexní datové struktury, které provádějí složité operace s daty.
* Jsou odvozeny z primitivních datových struktur.
* Dají se rozdělit na dvě kategorie:
  + Lineární: Array, Linked List, Queue, Stack, …
  + Nelineární: stromy, grafy, …

## Podrobnější rozdělení

### Pole (Array)

* Lze jej definovat jako kolekci homogenních prvků, má fixní velikost a je to statická datová struktura.
* Každý záznam v poli je nazýván prvek (element)
* Všechny prvku v poli jsou identifikovány stejným jménem proměnné, ale rozlišují se pomocí indexu.
* Pole může mít více dimenzí (2D pole – pole v poli, atp…)

### LinkedList

* Linked list je list v paměti, který se skládá z prvků zvaných „nodes“.
* Je dynamický, tudíž nemá fixní velikost.
* Každý linked list má tzv. „head node“ a „tail node“ to je první a poslední prvek v listu.
* Každý prvek (node) má v sobě obsažen odkaz na prvek před ním a prvek za ním.

### Stack (zásobník)

* Dynamická datová struktura, do které se dá vkládat a odebírat z ní jen na jednom konci.
* Navržena podle standardu LIFO (Last in First Out)

### Queue (fronta)

* Dynamická struktura, do které se vkládá z jednoho konce a z druhého se odebírá.
* Navržena podle standardu FIFO (First in Firs Out)

### Tree (strom)

* Struktura založena na vztahu rodič-dítě.
* Víceúrovňová struktura tvořená z kolekcí prvků. Těmto kolekcím se říká „nodes“
* „Nodes“ ve stromu mezi sebou udržují hierarchické vztahy.
* Nejvyšší „node“ ve stromu je nazýván „root node“ a nejspodnější „node“ se nazývá „leaf node“.
* Každý „node“ obsahuje ukazatel na „nodes“ okolo něj.
* Každý „node“ ve stromu může mít více jak jedno dítě, pokud není „leaf node“
* Každý „node“ má jen jednoho rodiče, kromě „root node“, který je rodičem všech

### Grafy

* Tato datová struktura obsahuje prvky založené na obrazné reprezentaci.
* Každý prvek je reprezentován „vrcholem“ grafu.
* Prvky v grafu jsou spojeny tzv. hranami.
* Hlavním rozdílem mezi grafem a stromem je že graf může obsahovat cyklus, zatímco strom nikoliv.

# Anotace

* Anotace se hlavně používají v jazyce **Java**
* V jazyce **C#** se jsou nazývány jako „atributy“.
* Představují nějakou značku, která reprezentuje metadata spojené s classou, interfacem, metodou
  + Označuje dodatečné informace, které kompilátor může použít.
* Anotace i atributy mohou být nad každou deklarací
  + většina anotací a atributů však většinou má omezení na co může být použita (proměnné, metody, třídy, …)
* Příklad atributu v Jazyce **C#:**

Text

Description automatically generated

# Operátory

• Operátory jsou symboly používané k provádění určitých operací (např. +, -, \*, /)

## Rozdělení operátorů

|  |  |  |
| --- | --- | --- |
| **Typ operátoru** | **Kategorie** | **Použití** |
| Unární | postfix | proměnná++; proměnná-- |
| prefix | ++proměnná --proměnná +proměnná; -proměnná ~ ! |
| Aritmetické | Multiplikativní | \* / % |
| aditivní | + - |
| Posuvné | posuv | <<; >>; >>> |
| Relační | porovnání | < > <= >= |
| rovnost | ==; != |
| Bitové | Bitové AND | & |
| Bitové výlučné OR | ^ |
| Bitové zahrnující OR | | |
| Logické | Logické AND | && |
| Logické OR | || |
| Ternární | ternární | ? : |
| Přiřazovací | přiřazení | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

## Příklady využití operátorů

### Unární

A screenshot of a computer

Description automatically generated

### Aritmetické

A screenshot of a computer

Description automatically generated

### Posuvné

**Posun do leva**

Graphical user interface, text

Description automatically generated

**Posun doprava**

Text

Description automatically generated with medium confidence

**Posun pomocí >>>**

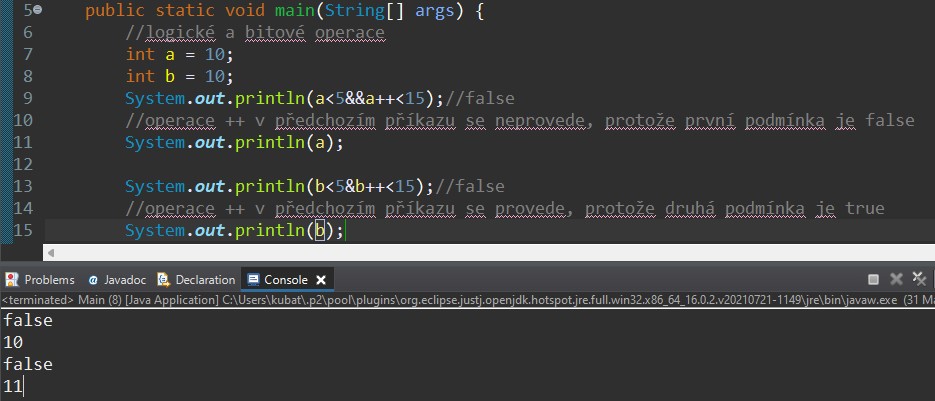
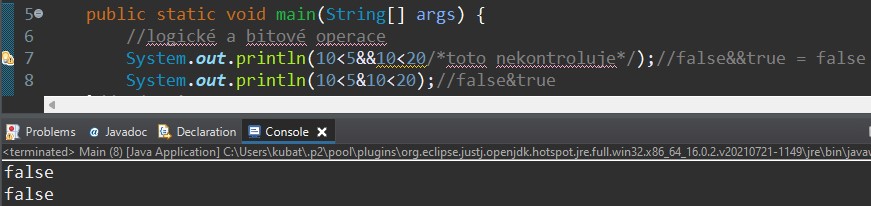
Text

Description automatically generated

## Logické a bitové

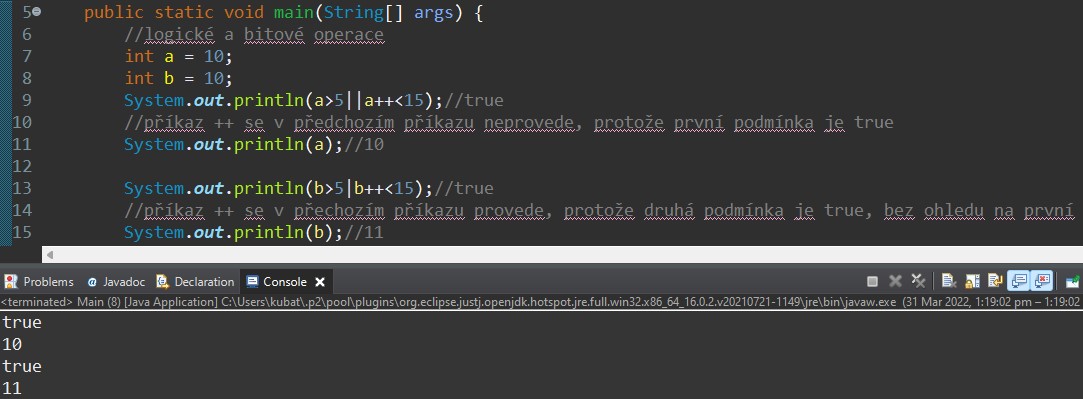
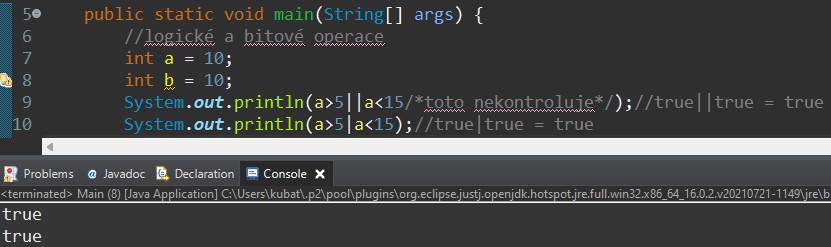
*Logické a bitové AND*

* Rozdíl mezi bitovými a logickými operacemi AND je, že když je více podmínek za sebou, tak bitový operátor zkontroluje obě podmínky, bez ohledu na to jestli je první false nebo true, logický kontroluje druhou jen když je první true



*Logické a bitové OR*

* Logický operátor nekontroluje obě podmínky, pokud je první true, bitový kontroluje obě bez ohledu na první podmínku



### Ternární

* Ternární operátory se využívají jako jednořádkové náhrady za if-else příkazy

A screenshot of a computer

Description automatically generated with medium confidence

* Rozborka příkazu:

Text

Description automatically generated

### Přiřazovací

