Algoritmy a algoritmizace

Algoritmus je nějaký schématický postup k řešení nějakého problému. Každý algoritmus má jeden začátek (vždy začíná na stejném místě) a minimálně jeden konec (někdy skončí, ale může skončit v různých stavech). Algoritmus může do sebe brát různé vstupy a může vracet různé výstupy.

Nějaký problém může mít různé způsoby řešení a tedy i algoritmy. Algoritmy se liší zejména rychlostí, tedy tím, kolik kroků potřebují k vyřešení problému.

Každý algoritmus by měl splňovat následující:

- **konečnost**, tedy, že bude problém algoritmu vyřešen v konečném počtu kroků. Jedinou výjimku tvoří reaktivní procesy, které z principu očekávají vstup a na něj reagují.
- **obecnost**, tedy, že algoritmus řeší obecný problém, místo aby řešil konkrétní konečnou podskupinu. Místo počítání dvou čísel 1 + 1, bude algoritmus sčítat dvě (či nekonečně mnoho) jakýchkoliv čísel.
- **determinovanost**, tedy, že algoritmus pro stejné vstupy vygeneruje stejný výstup. Výjimku tvoří náhodné algoritmy, které mají vstupem taktéž náhodná čísla (či jiné bity).
- **jednoznačnost**, tedy, že algoritmus je přesné definovaný pro každý krok. Z této vlastnosti vznikli programovací jazyky, které vytvářejí syntaxi.
- **efektivnost**, tedy, že algoritmus zbytečně nebude zabírat prostředky počítače. Některé problémy nejsou na aktuální počítačích řešitelné (resp. o takových algoritmech nejsme obeznámeni).

Algoritmy lze reprezentovat mnoha způsoby:

- **textově**, kde jsou jednotlivé kroky napsány slovně. Slovní popisy jsou velmi náchylné na chyby (překlepy, přepisy) a na interpretaci čtenáři. Některé věci nelze zcela dobře popsat a text obsahuje velmi mnoho informací na více (hodně výplňových slov, ...). Je nutné znát daný jazyk, ve kterém byl text napsán. Nelze se dobře orientovat na jaké části ukazujeme např. větvením.
- **struktogramem**, které jsou velmi podobné tabulkám. Při přidání větvení jsou velmi velké a nepřehledně se značí.
- **vývojovým diagramem**, kde má každá operace svůj přiřazený obrazec. Je velmi jednoduché na použití.
- **pseudokódem**, kde se algoritmus píše již velmi blízko programovacímu jazyku, ale jsou vypuštěny implementační detaily (vytvoření listu, inicializace, ...).

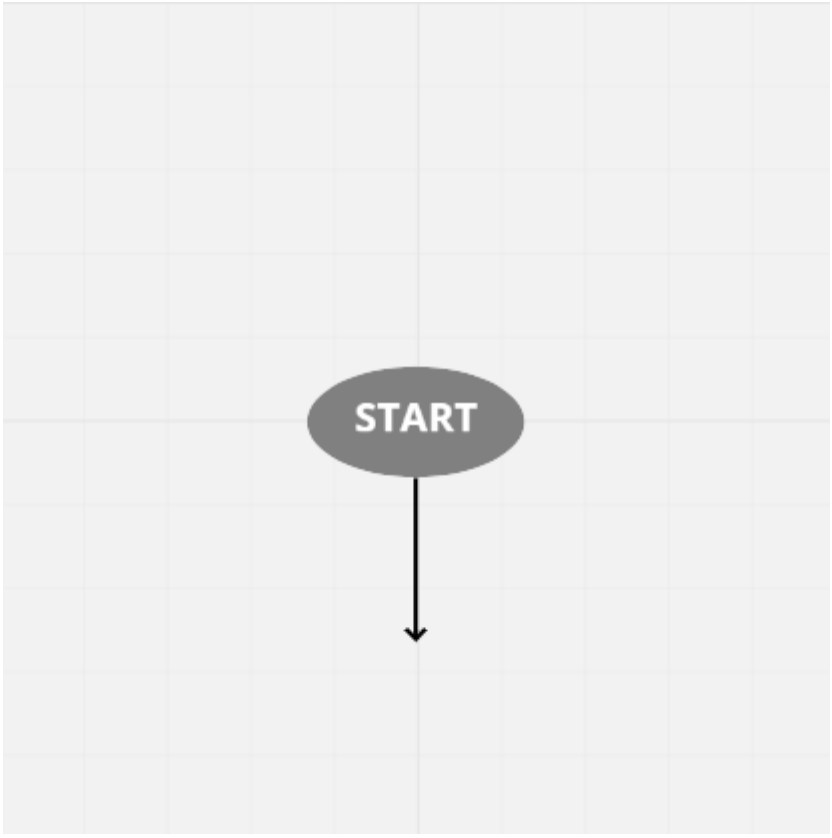
Vývojové diagramy

Vývojové diagramy jsou dobrý způsob pro reprezentaci algoritmů. Mají předefinované různé tvary, které mají vždy svůj účel. Tvary můžeme na svojí potřebu přidávat nebo používat jiné standardy. Pro orientaci se používají šipky, aby se vědělo kudy "teče" průběh algoritmu.

Jednotlivé tvary mají různé vstupy a výstupy na různých hranách. Je nutné je sledovat a nepřipojovat šipkami na špatné vstupy.

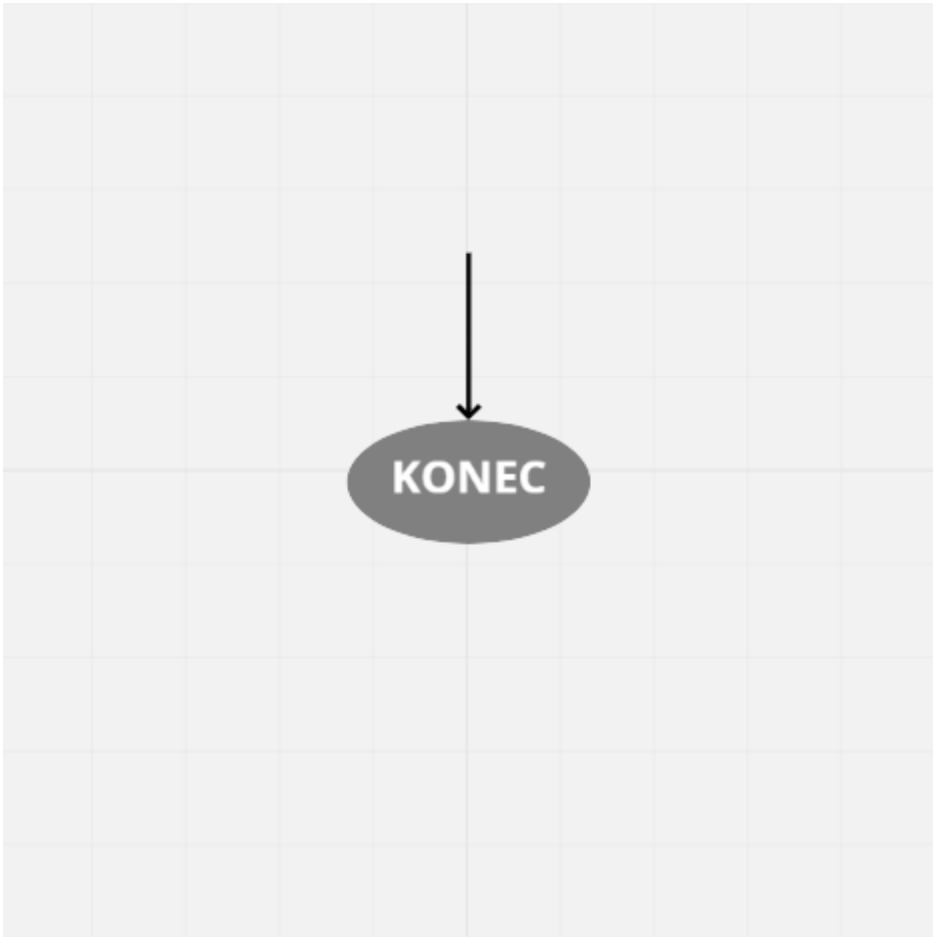
Začátek

V diagramu musí být právě jeden, v tomto bodu začíná diagram.



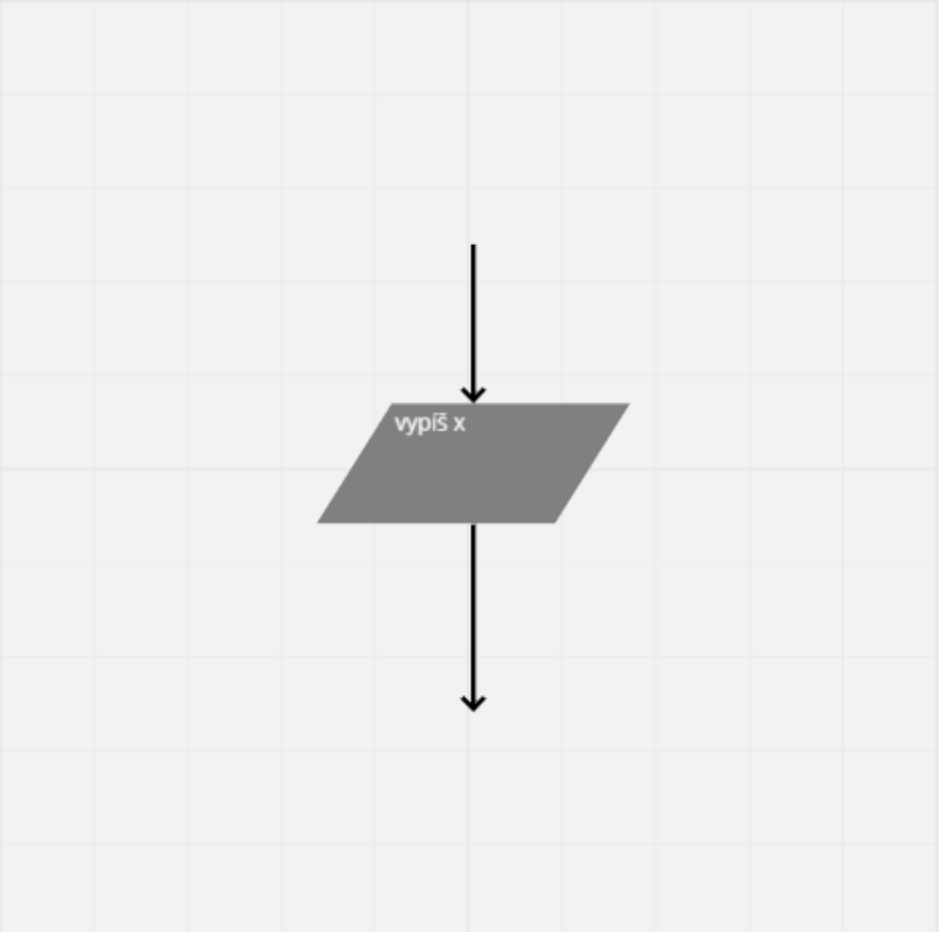
Konec

Označuje, že algoritmus končí. V diagramu jich může být více.



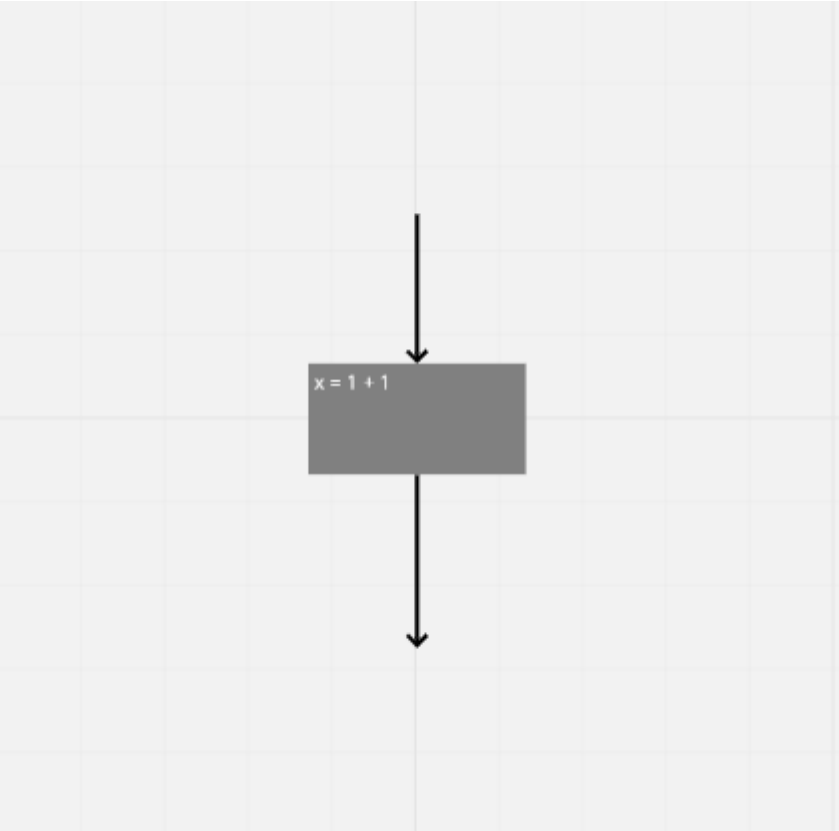
Vstup a výstup

V případě příchodu do tvaru můžeme vypsat či přečíst nějaké data. Často označujeme čtení jako `přečti` a výpis jako `vypiš`. Můžeme zde používat proměnné a další věci.



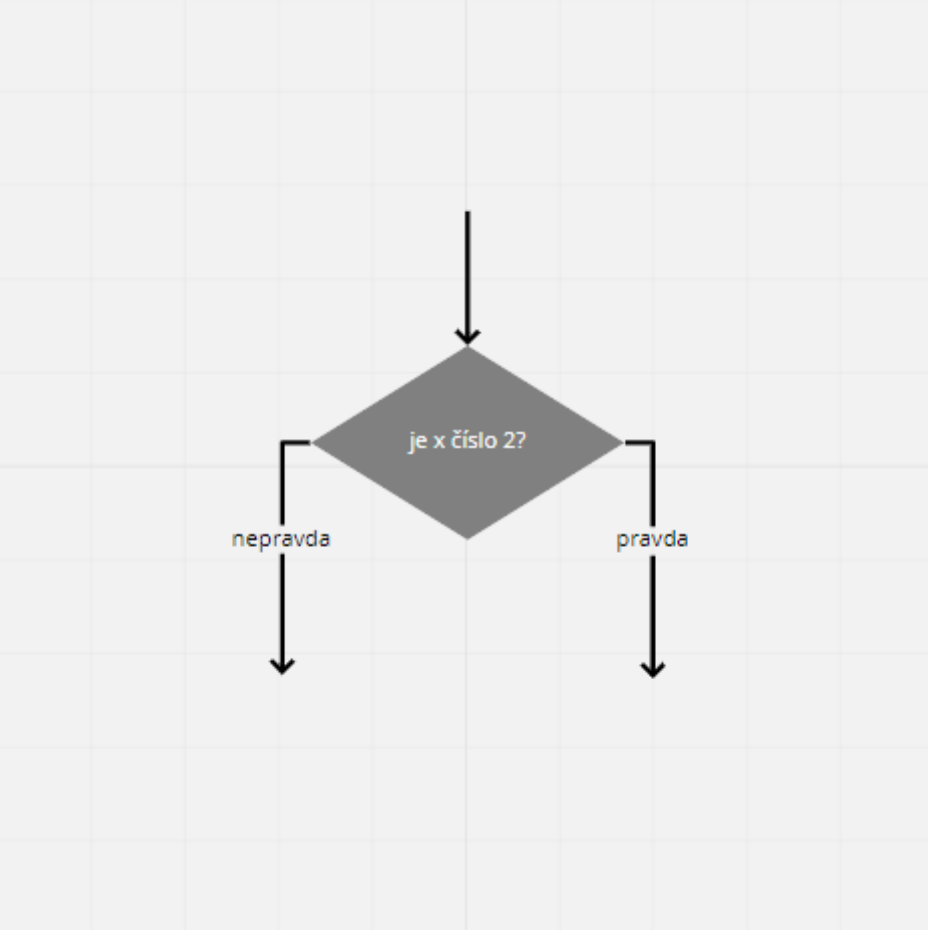
Příkazy

Při přístupu vykoná různé výpočty a hodnoty může např. přiřadit do proměnné.



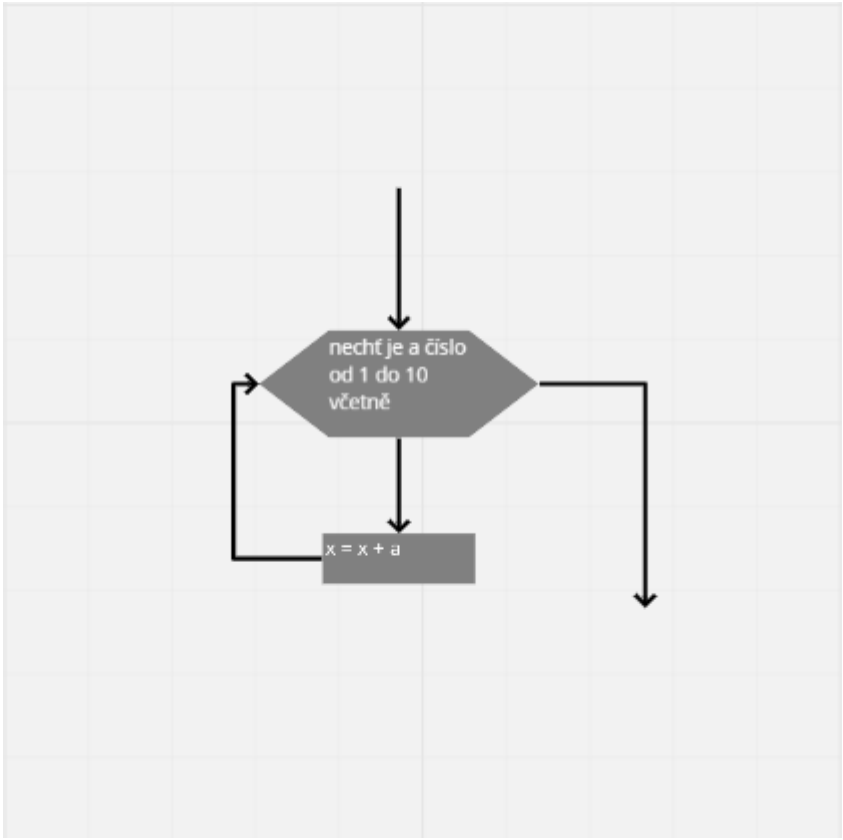
Podmínka

Při přístupu se podmíní nějaká část diagramu. V případě, že zadaná podmínka je pravdivá, pokračuje se ve větvi, u které je napsané `pravda`, a naopak tam, kde je napsáno `nepravda`. Bez pojmenování větví je diagram špatně.



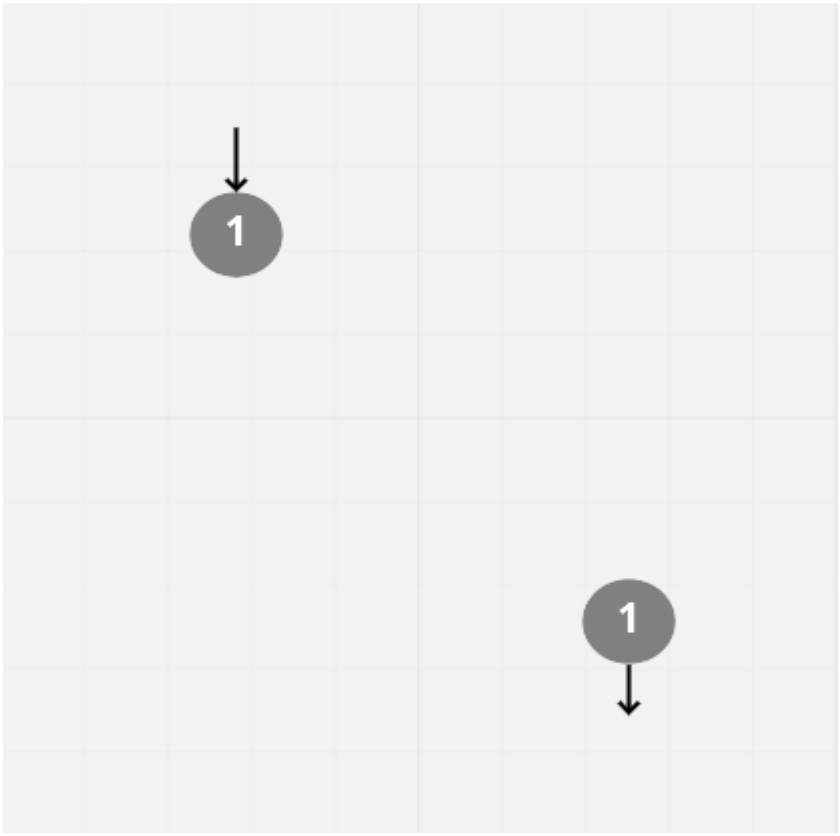
Cyklus

Při přístupu prochází určitými cykly dokud platí podmínka uvnitř zadaná. Zespoda je k dispozici výstup právě průchodu cyklu, který následuje do levého bloku. Pravý výstup se vykoná tehdy, když cyklus skončil (podmínka neplatí).



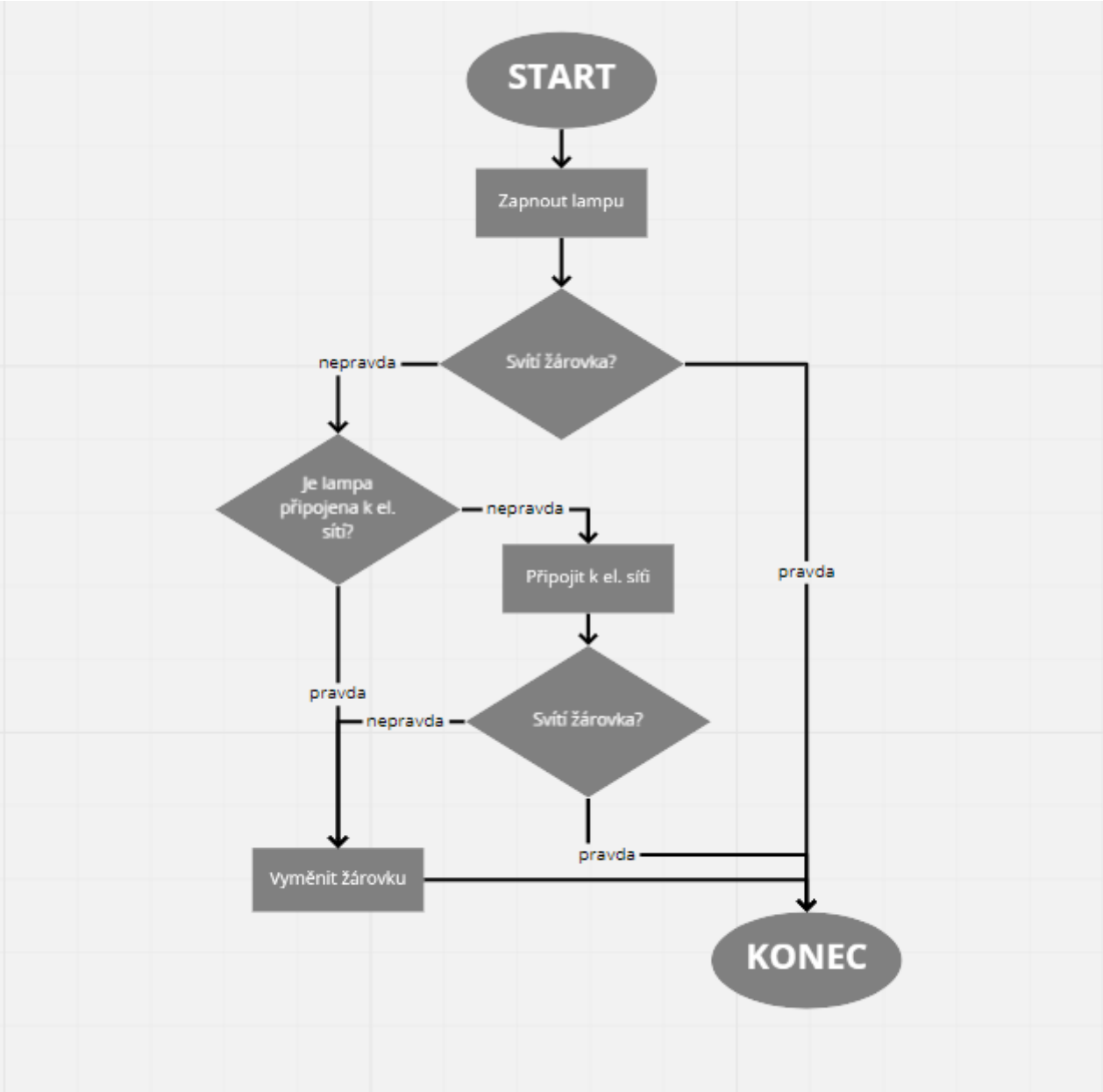
Tunel

Propojuje dvě části diagramu. Vždy se čísluje a v diagramu může existovat s jedním číslem pouze jeden vstup (šipka dovnitř) a jeden výstup (šipka ven). Pokud se diagram dostane do stavu tohoto tunelu (resp. vstupu tunelu), tak přeskočí právě do výstupu.

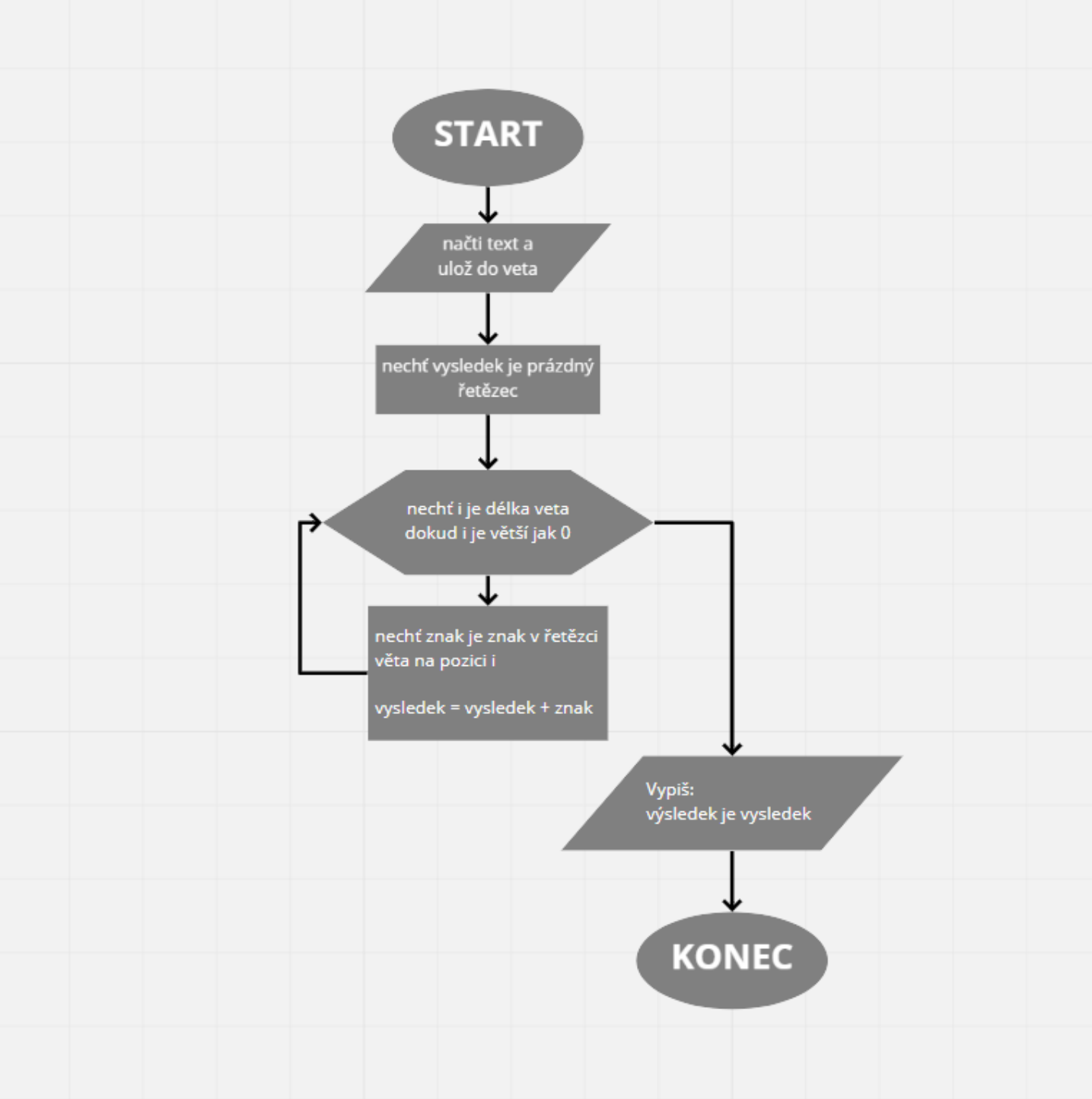


Ukázky

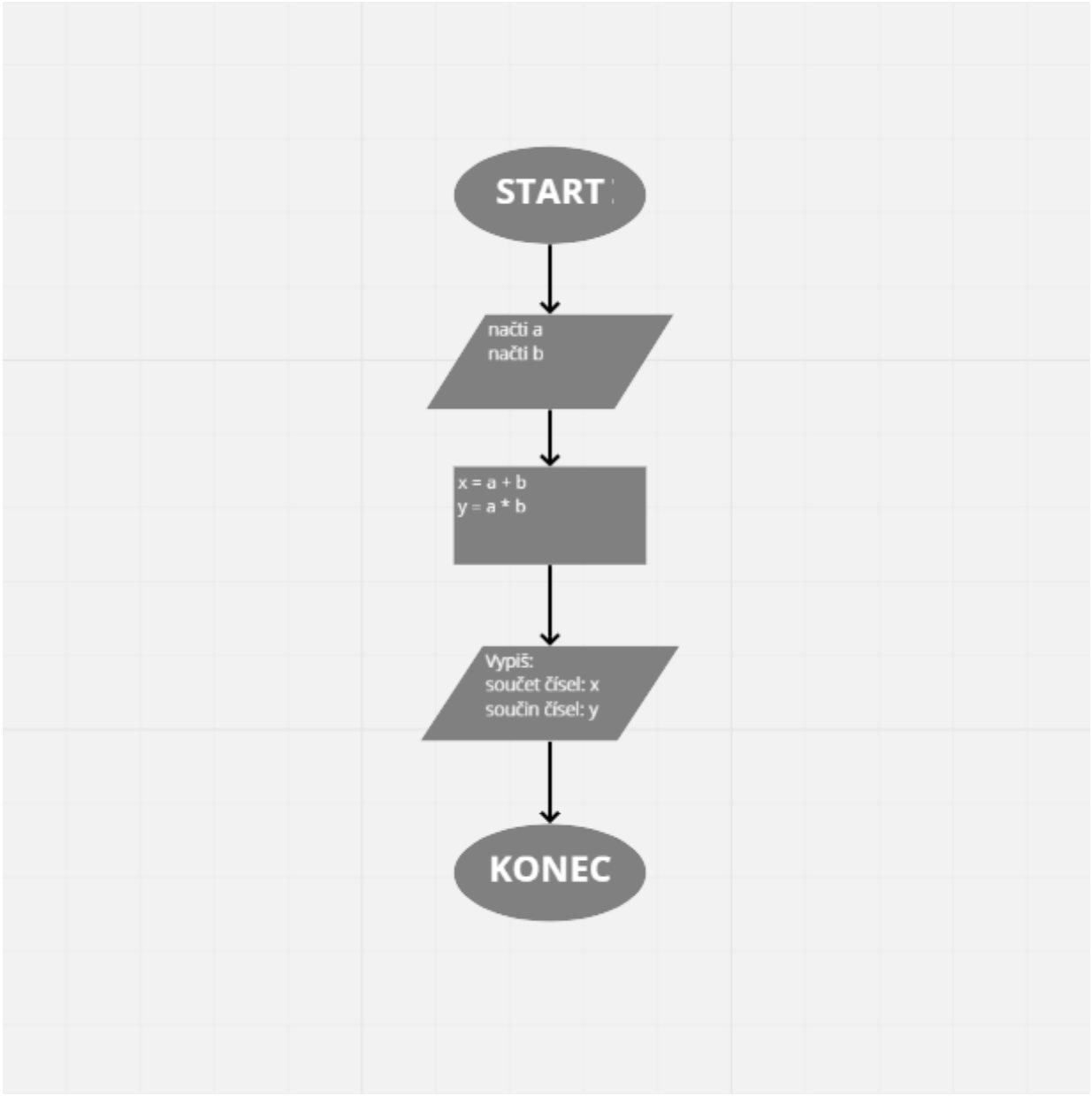
Ukázka, velmi nerealistického (kdo by zapínal lampu a ihned měnil žárovku?), zapnutí lampy.



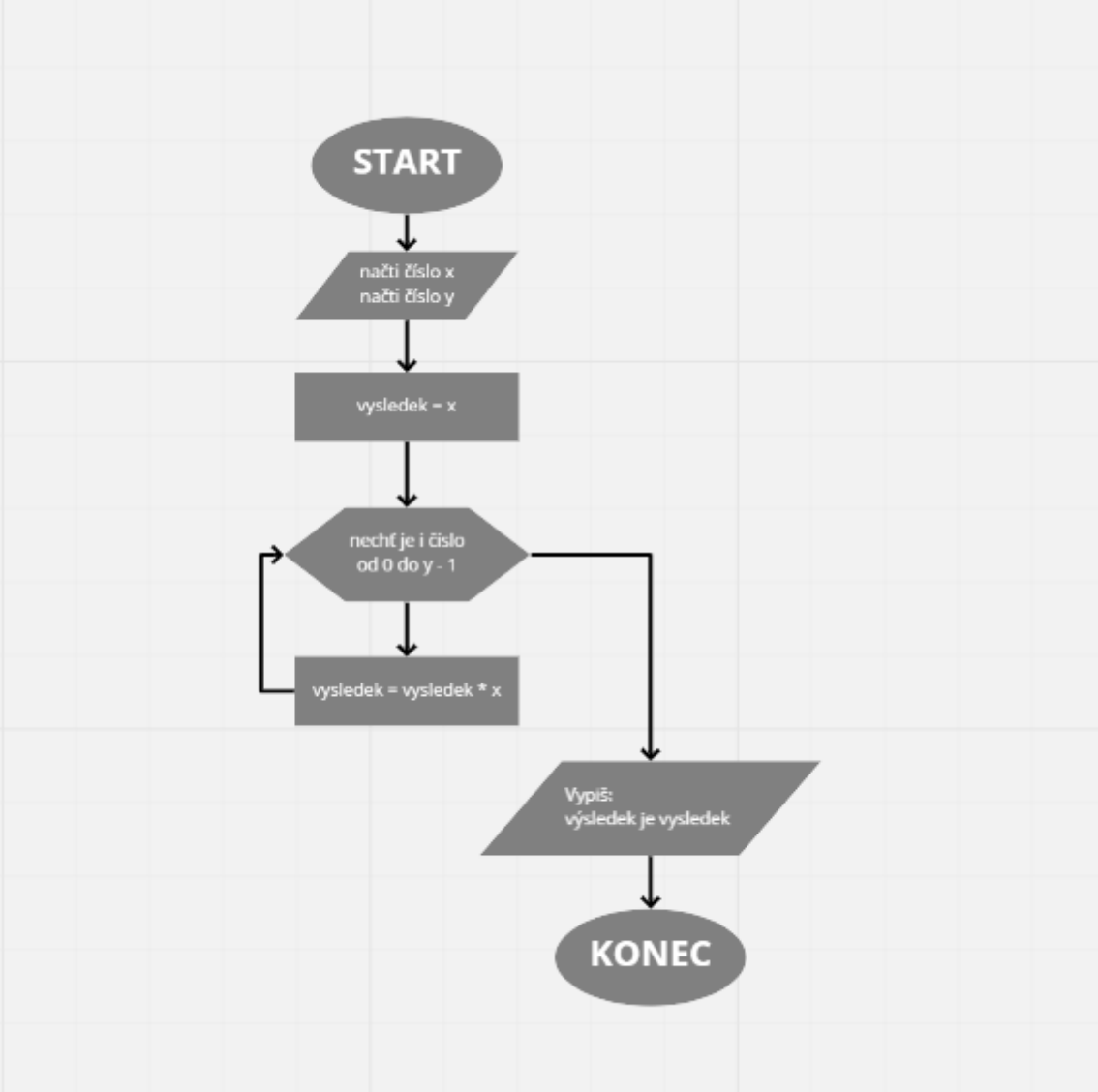
Otočení znaků v textovém řetězci naopak. Např. vstup "ahoj" se na výstup vypíše "joha".



Ukázka součinu a součtu.



Ukázka mocniny určitého čísla `x^y`.



Algoritmizace

Algoritmizace je způsob řešení určitého problému za pomocí algoritmů. Nejčastěji je rozděleno do tří fázi:

- **formulace**, kde je samotný problém definován (co požadujeme).
- **analýza**, kde prozkoumáme, jak je možné problém řešit, porovnáme rychlosti a další.
- **vytvoření**, kde vytvoříme samotný algoritmus.

Často se kroky přeskakují a nebo se rovnou programuje. Ne však vždy je to nejlepší způsob.

Programovací jazyky a C#

Programovací jazyk je specifikace s kompilátorem, který je schopen ze sekvence příkazů vytvořit program k řešení určitého problému. Pro řešení problémů používáme algoritmy. Implementace algoritmu v programovacím jazyku je konkrétní, tedy, že přesně víme, jak se program bude chovat.

Strojový kód je binární soubor s instrukcemi, které umí počítač spustit a provádět. Tyto instrukce nemusí fungovat na každém počítači - každý počítač má jiný OS i hardware a pro ně je potřeba jiný strojový kód. Nejčastěji se setkáme s `.exe` soubory na Windows. Strojový kód v podstatě nelze programovat, velmi blízko je však Assembler.

Kompilace je proces převedení zdrojového kódu (programu) v programovacím jazyce do strojového kódu. Stará se o linkování knihoven a spoustu dalších věcí, včetně například optimalizace kódu.

Kompilovaný jazyk je jazyk, který se při napsání zkompiluje a jeho strojový kód je dál distribuován. Jedná se o nejčastější typ jazyku, protože uživatel většinou nemusí nic na více stahovat.

Interpretovaný jazyk je jazyk, který se při napsání přenáší v dané formě a je kompilován až při spuštění pomocí interpretátoru. Jako ukázku můžeme uvést např. jazyk Java, který pro spuštění potřebuje stáhnout právě Javu.

My se budeme v tomto předmětu (stejně jako minulý rok) zaobírat programovacím a kompilovaným jazykem **C#** [sí šarp]. Je totiž velmi známý a velmi dobrý pro začátečníky. Sice neposkytuje hlubší poznání, ale to tento předmět nevyžaduje. My budeme používat .NET Core verzi, která je spustitelná skoro na jakémkoliv OS. Existuje taktéž .NET Framework, který cílí pouze na Windows uživatele. Prozatím budeme pracovat s konzolovými aplikacemi a později se dostaneme k formulářům na Windows. C# vytváří a specifikuje Microsoft.

Visual Studio

Visual Studio je nejrozšířenější IDE (tj. Integrated Development Environment) pro psaní C# kódu a dalších jazyků (C++ a i webové aplikace). Má velmi jednoduchou správu projektů a pro C# taktéž jednoduché spuštění a kompilaci. Veškeré věci, co umí Visual Studio lze však dělat v jakémkoliv editoru (a resp. konzolové řádce), ale VS nám velmi ulehčí práci.

Pro práci si tedy nainstalujte Visual Studio (nepleťte si s Visual Studio Code!) dle instrukcí ZDE.

Po otevření VS se vám často zobrazí úvodní menu, ve kterém vidíte naposledy otevřené projekty a taktéž tu máme tlačítka na otevření či vytvoření projektu. Při kliku na vytvoření projektu se nám otevře nabídka, ve které můžeme zvolit typ projektu. Jak jsem dříve naznačoval, aktuálně budeme používat .NET Core, a proto vybereme (vyhledáme) položku s názvem `Konzolová aplikace`, ve které máme tag Linux (tj. bez nadpisu .NET Framework).

Po výběru typu projektu se nám zobrazí možnost k výběru názvu projektu a jeho umístění. Pro projekty si doporučuji vytvořit speciální složku pro PVA a název nechávejte bez diakritiky a dalších speciálních znaků. Často taktéž budeme chtít zaškrtnout možnost `Umístit řešení a projekt do stejného adresáře`, kterým nám zmenší velikost projektu (resp. jeho filesystému). VS podporuje velké projekty a tato možnost se nám teď nebude hodit.

Po pojmenování projektu se nás VS zeptá na verzi .NET, a aktuálně budeme vybírat .NET verze `6.0`.

Po vytvoření projektu se nám zobrazí VS s kódem. V hlavním souboru `Program.cs` budeme psát náš kód a můžeme tam prozatím vidět pár řádků, které vypíší text do konzole. Samotné UI VS je rozděleno do několika částí. V hlavní liště v horní části VS nalezneme tlačítka na spuštění programu a i výběru řešení (release vs debug, kde release je připraven na distribuci). Pod kódem obvykle máme výstup a seznam chyb projektu. Napravo máme průzkumníka řešení, kde můžeme vidět všechny soubory projektu.

V složce `bin` projektu můžeme vidět zmíněné cíle řešení (release a debug) a v každém je již kompilovaný kód připravený na spuštění.

Proměnné a datové typy

Proměnná slouží v programovacích jazycích k uložení určitých dat v operační paměti (tj. v podstatě RAM). Každá proměnná musí mít přidělené jméno a datový typ. Datový typ určuje, jaké data se dovnitř mohou ukládat. Hodnoty v proměnných jsou vždy ukládané v binární hodnotě ale v programu je referujeme často pouze v desítkové soustavě. Hodnotu proměnné lze číst a měnit ji. Proces deklarace je to, když je proměnná v kódu vytvořena pomocí příkazu. Proces inicializace následuje po deklaraci a určuje proměnné její hodnotu.

Jména proměnných by neměli s ostatními věcmi kolidovat a měli by být krátké a výstižné. Názvy proměnných poté používáme k čtení a úpravě. Zejména používáme anglické názvy, ale záleží na typu projektu. Ukázka správných jmen:

```
personName
school
drinkId
age
iq
sumOfGrades
```

Tento styl pojmenování se jmenuje camelCase, existují však např. PascalCase, kebab-case, snake_case a další.

Datový typ vždy píšeme před název dané proměnné a hodnotu přiřazujeme operátorem přiřazení (=). Například tedy:

```
datovyTyp nazevPromenne = hodnota;
int age = 3;
```

Datové typy

Celé číslo

Názvy: `short/int/long`

Ukládá celé číslo v daném rozsahu. Velikosti:

- `short`: -32 768 až +32 767
- `int`: -2 147 483 648 až +2 147 483 647
- `long`: -9 223 372 036 854 778 až +9 223 372 036 854 775

```
int accountBalance = 1180040;
short sum = 3;
```

Celé kladné číslo

Názvy: `ushort/uint/ulong`

Ukládá celé kladné číslo (u = unsigned = bez znaménka) v daném rozsahu. Velikosti:

- `ushort`: 0 až +65 535
- `uint`: 0 až +4 294 967 295
- `ulong`: 0 až +18 446 744 073 709 551 615

```
ushort age = 18;
uint iq = 102;
```

Desetinné číslo

Názvy: `float/double/decimal`

Ukládá desetinné číslo v daném rozsahu. Rozsahy nejdou přesně definovat, protože neumíme některá čísla vůbec v počítači reprezentovat. Odhadované velikosti:

- `float`: přesnost cca 6 číslic
- `double`: přesnost cca 15 číslic
- `decimal`: přesnost cca 28 číslic

```
float age = 16.4f;
double average = 3.20123d;
decimal pi = 3.14159265358979323846m;
```

U čísel je nutné psát dodatky (`f/d/m`), které daný typ určují, aby se číslo správně reprezentovalo. Bez uvedení se počítá s přesností `double`.

Znak

Název: `char`

Ukládá jeden znak. Vždy používáme jednoduché uvozovky (`'`) pro určení hodnoty.

```
char sortBy = 'a';
```

Text

Název: `string`

Ukládá textový řetězec znaků. Vždy používáme jednoduché uvozovky (`"`) pro určení hodnoty.

```
string name = "Matěj Cajthaml";
string school = "Smíchovská střední průmyslová škola a gymnázium";
```

Pravdivostní hodnoty

Název: `boolean/bool`

Ukládá pravdivostní (boolean) hodnotu, která může nabývat pravdy (`true`) a nepravdy (`false`). Hodnota `1` znamená pravdu, hodnota `0` nepravdu.

```
bool isFamous = false;
bool friendly = true;
```

Přepsání hodnoty

Poté, co definujeme proměnnou, můžeme její hodnotu nadále upravovat operátorem přepsání:

```
int age = 18;
age = 19;



string school = "Základní škola Vesmírná";
school = "SSPŠaG";
```

Konstanty

Jedná se o speciální typ proměnné, který po inicializaci nelze upravovat (její hodnota zůstane stejná). Hodí se zřejmě na věci, které se v průběhu nemění. Konstanty často dovolují zrychlení programu a taky k tomu, že jejich hodnota nikde nebude omylem upravena.

```
const int age = 18;


// CHYBA:
// age = 19;
```

Komentáře a klíčové slova

Komentáře slouží k označení nějakého textu, který je pro kompilátor neviditelný. Používáme jej na různé poznámky (zpátky se vrátíme ke kódu abychom věděli, co daná část dělá) nebo obecně k dokumentaci. Máme dva typy komentářů:

```
// Komentář
/*
Víceřádkový komentář
*/
```

Klíčová slova jsou speciální sekvence písmen, které jsem rezervovány programovacím jazykem. Tyto názvy nemůžeme používat jako jména proměnných a v podstatě čehokoliv jiného. S některými názvy jsme se již setkali - jedná se např. o datové typy, nemůžeme tedy například vytvořit proměnnou int datového typu int:

```
int int a = 3; // NEJDE.
```

Seznam všech klíčových slov naleznete ZDE.

Práce s konzolí

Konzole je zobrazovací a příjímací zařízení, které umí od uživatele číst vstup a vypisovat textový výstup. Konzole jsou velmi oblíbené zejména pro svojí jednoduchost a nenáročné používání. Je to tedy komunikátor mezi programem a uživatelem. V C# k přístupu ke konzoli používáme třídu `Console`. Konzoli si můžeme představit jako pole znaků (dvojdimenzionální), kde na každou pozici můžeme vypsat znak.

Třída, je velmi zjednodušeně, seskupení vlastností a metod. Vlastnost je taková proměnná, která se váže k instanci třídy a metodu můžeme vyvolat zavoláním (`()`). Metodě můžeme předávat parametry a vždy něco vrací (datový typ a nebo tzv. void - nic). K oběma přistupujeme pomocí operátoru přístupu (`.`). Metody se často označují jako funkce, prozatím tento nedostatek promineme. O třídách si toho povíme více zejména ke konci školního roku.

Zápis

Do konzole vypisujeme pomocí příkazu `Console.Write(value);` a `Console.WriteLine(value);` kde value je textová hodnota a nebo datový typ, který lze na textovou hodnotu převést. Write i WriteLine zapisuje tam, kde je kurzor, ale WriteLine po vypsání zalomí řádek a další výpis pokračuje na další řádce.

```
Console.WriteLine("Hello world!");
```

```
const string name = "Matěj Cajthaml";

Console.WriteLine(name);

const int age = 99;

Console.WriteLine(age);

Console.Write(name);
Console.Write(age);
```

Výše uvedený kód do konzole vypíše:

```
Hello world!
Matěj Cajthaml
99
Matěj Cajthaml99
```

Čtení

Pomocí příkazu `Console.ReadLine();` můžeme z konzole od uživatele přečíst text. Při zavolání příkazu se program pozastaví (resp. další příkazy čekají) do té doby, než uživatel nezadá text a nezmáčkne enter. Metoda nám vrací textový string se zadanou hodnotou.

```
string name;
name = Console.ReadLine();

Console.WriteLine(name);
```

Výše uvedený kód do konzole vypíše to, co do něj zadáme. Tento příkaz pro čtení se používá i ke konci programu - například, když uživatel odejde, tak nechceme konzoli hned ukončit, ale ještě mu něco napsat a počkat, až něco nezmáčkne.

Nadpis

Pomocí vlastnosti Console.Title můžeme konzoli nastavit nadpis, který se zobrazí v jejím okně. Jedná se pouze o kosmetickou úpravu.

```
Console.Title = "Nadpis konzole ;)";
```

Barvy

Barvy v konzoli jsou velmi omezené (nepočítáme-li různé nástavby jako ANSI) - dovolují nám pouze 16 barev. Pro tyto barvy máme připravenou třídu `ConsoleColor`, ve které tyto barvy pod anglickými názvy nalezneme. Znovu přistupujeme pomocí operátoru přístupu (`.`).

Před vypsáním nějaké hodnoty můžeme konzoli nastavit vlastnost `BackgroundColor` nějakou hodnotu z `ConsoleColor` a daná barva se vykreslí pod daný text. Velmi podobně funguje `ForegroundColor`, která obarví daný text.

```
Console.ForegroundColor = ConsoleColor.Black;
Console.BackgroundColor = ConsoleColor.DarkBlue;
Console.Write("Kočka je kočka ale i ");
Console.BackgroundColor = ConsoleColor.DarkYellow;
Console.Write("kočka");
Console.BackgroundColor = ConsoleColor.DarkBlue;
Console.WriteLine(".");
```

Výše uvedený kód do konzole vypíše text "Kočka je kočka ale i", na modrém pozadí, text "kočka", na zlatém pozadí a text "." na modrém pozadí. Všechen text bude černé barvy. Barvy tedy můžeme skládat.

Pro resetování barev do původní hodnoty můžeme použít příkaz `Console.ResetColor();`.

Viditelnost kurzoru

Vlastnost `Console.CursorVisible` nám upravuje to, zda je kurzor (resp. blikající čára) viditelná. Často se nám hodí kurzor skrýt, když uživatel nemůže nic psát, ale čekáme na jeho vstup.

```
Console.CursorVisible = false;
```

Výmaz konzole

Metoda `Console.Clear();` vyčistí konzoli a další zápis začíná v levém horním rohu (tj. na pozici `0,0`).

```
Console.Clear();
```

Výrazy

Výrazy jsou (stejně jako v matematice) seskupení různých operací pro vypočítání nějaké hodnoty. V programování jsou zejména určeny k upravení hodnot proměnných. Výsledky výrazů však nemusíme ukládat jenom do proměnných, ale rovnou je někde jinde využít (např. je vypsat).

K dispozici máme základní matematické operace:

```
const int a = 3;
const int b = 9;

const int x1 = a + b; // sčítání
const int x2 = a - b; // odčítání
const int x3 = a * b; // násobení
const int x4 = b / a; // dělení
```

Jako operátor máme k dispozici například sčítání řetězců (nebo hodnot, které lze převést na text) -sjednotí řetězce do jednoho:

```
string text = "Mé jméno jest: ";
const string name = "Matěj";
const int age = 5;

text = text + name + ", a je mi " + age;
```

Konverze

Konverze je převádění různých hodnot datový typů do jiného datového typu. Nejčastěji potřebujeme z textového řetězce (který zadá např. uživatel) získat datový typ (např. číslo). Může to být ale jakákoliv hodnota do jakékoliv jiné.

Skoro každý datový typ můžeme převést na textový řetězec pomocí zavolání `.ToString()` na jeho hodnotě. Metoda nám vrátí nějakou reprezentaci hodnoty v textu.

```
Console.WriteLine(293.ToString());
Console.WriteLine(false.ToString());
Console.WriteLine('a'.ToString());
Console.WriteLine(213.41293d.ToString());
```

Výše uvedený kód vypíše:

```
293
False
a
213.41293
```

Převody rozdělujeme na dva typy:

- implicitní, neboli převody, které neztratí informace o datech (tedy celé data se do nového datového typu vejdou), tedy např. převedení int čísla do long (int má menší rozsah než long a vejde se do něj).
- explicitní, neboli převody, kde se nějaká část informace může ztratit, používáme operátor přetypování `(datovyTyp) hodnotaNeboPromenna`.

```
// Implicitní
int numberGoodStudents = 17;
long numberBadStudents = numberGoodStudents;

// Explicitní
Console.WriteLine((char)293);
Console.WriteLine((int)'a');
Console.WriteLine((bool)true);
Console.WriteLine((int)392.3193m);
```

Převedení mezi datovými typy můžeme pomocí metody `.Parse` na jejich datovém typu s parametrem na převod, například tedy:

```
bool  boolValue  = bool.Parse("false");
int   intValue   = int.Parse("-5");
char  charValue  = char.Parse("5");
ulong ulongValue = ulong.Parse("51231820932");

Console.WriteLine(boolValue);
Console.WriteLine(intValue);
```

```
Console.WriteLine(charValue);
Console.WriteLine(ulongValue);
```

Pro velmi podobné účely se používá předpřipravená třída `Convert`, která převádí různé datové typy. Jediný rozdíl mezí `.Parse` a těmito metodami je to, že `Convert` automaticky zpracuje hodnotu `null` (nic) na správnou základní hodnotu. Ukázky používání:

```
int a = Convert.ToInt32("312");
Convert.ToDecimal("0,12390123");
Convert.ToBoolean("true");
boolean b = Convert.ToBoolean(0);
char c = Convert.ToChar(67);
double d = Convert.ToDouble(15);
```

Když z nějakého důvodu (např. v zadaném stringu pro konverzi na číslo je znak) konverze typu `Convert` nebo `.Parse` úspěšně neproběhne, tak se vyvolá výjimka. Výjimka je varování programu, že nastala nějaká chyba a program by ji měl vyřešit. Tuto výjimku můžeme zachytit pomocí bloku try-catch, viz.:

```
try {
    Convert.ToInt32(Console.ReadLine());
} catch (Exception e) {
    Console.WriteLine("Špatné číslo.");
}
```

Více o výjimkách si povíme později.

Složené přiřazení a matematické operace

Složené přiřazení

Složené přiřazení je zrychlený zápis operací v C# a lze používat pro jakékoliv operátory, zjednodušení je definováno následovně:

```
x op= y;
// se rovná
x = x op y;
```

Například:

```
float probability = 0.5f;


probability += 0.3f;
probability /= 2;
probability -= 0.05f;
probability *= 0.95f;


Console.WriteLine(probability);
```

Vypíše:

```
0.3325
```

Matematické operace

Pro složitější matematické operace máme v C# k dispozici třídu `Math`, ve které máme připravené metody s operacemi. Jedná se např. o mocniny, odmocniny, absolutní hodnoty a nebo například zaokrouhlení.

```
double numberDouble = 10.51923d;


Console.WriteLine(Math.Ceiling(numberDouble)); // Zaokrouhlí nahoru === 11
Console.WriteLine(Math.Floor(numberDouble)); // Zaokrouhlí dolů === 10
Console.WriteLine(Math.Round(numberDouble)); // Zaokrouhlí "dle čísla" === 11
Console.WriteLine(Math.Round(numberDouble, 3)); // Zaokrouhlí "dle čísla" na 3 desetinné mí
```

```
double numberDouble = 10.51923d;


Console.WriteLine(Math.Sin(numberDouble));
Console.WriteLine(Math.Cos(numberDouble));
```

```
Console.WriteLine(Math.PI);
Console.WriteLine(Math.Tau);
Console.WriteLine(Math.E);
```

```
int number = -10;


Console.WriteLine(Math.Abs(number)); // Absolutní hodnota === 10
Console.WriteLine(Math.Pow(number, 3)); // Mocnina na třetí === (-10)*(-10)*(-10) = 100 * (
Console.WriteLine(Math.Max(number, 3)); // Největší číslo z uvedených === 3
Console.WriteLine(Math.Min(number, 3)); // Nejmenší číslo z uvedených === -10
Console.WriteLine(Math.Sign(number)); // Znaménko čísla === -1
```

Textové funkce

Textové funkce slouží k úpravě či čtení údajů z textových řetězců. Např. se tedy může jednat o délku textu, smazání bílých znaků, nahrazení části textu, změna velikosti písmen a další.

```
Console.WriteLine("Dobrý den!");
Console.WriteLine("Dobrý den!".Length); // Délka textu === 10
Console.WriteLine("Dobrý den!".Substring(0, 5)); // Text od 0 pozice do 5 pozice === "Dobrý
Console.WriteLine("Dobrý den!".PadLeft(20, '.')); // Zleva přidáme do pozice 20 tečky === "
Console.WriteLine("Dobrý den!".PadRight(30, '.')); // Zprava přidáme do pozice 30 tečky ===
Console.WriteLine("        a     Dobrý den!          ".Trim()); // Odstraníme prázdná místa
```

```
Console.WriteLine("Dobrý den!".Contains("den")); // Obsahuje "den"? === true
Console.WriteLine("Dobrý den!".Contains("ahoj")); // == false
Console.WriteLine("Dobrý den!".StartsWith("Dobrý")); // Začíná text "Dobrý"? === true
Console.WriteLine("Dobrý den!".EndsWith("den?")); // Končí na text "den?" === false
Console.WriteLine("Dobrý den!".Replace("den", "večer")); // Nahradí den na večer === "Dobrý
Console.WriteLine("Dobrý den!".ToLower()); // Převede na malá písmena === "dobrý den!"
Console.WriteLine("Dobrý den!".ToUpper()); // Převede na velká písmena === "DOBRÝ DEN!"
```

Na další stránce si řekneme, co jsou to podmínky. Pro ně se nám bude hodit metoda `.Contains`, která dle parametru zkontroluje, zda daný řetězec obsahuje hodnotu.

```
bool hasSsps = Console.ReadLine().Contains("SSPŠ");
```

Podmínky

Podmínky jsou způsob větvení programu, tedy rozhodování se dle nějakých podmínek. Podmínkami podmiňujeme volání sady příkazů, když nějaké tvrzení platí. V podmínkách využíváme pravdivostní hodnotu (boolean). Pokud je hodnota true, bloky uvnitř podmínky se

provedou. Podmínky značíme klíčovým slovem if, po kterém následuje kulatými závorkami označený výraz. Po bloku následují složené závorky s příkazy.

```
Console.WriteLine("Ahoj!");

if (true) {
    Console.WriteLine("Ahoj po druhé!");
}
```

Vypíše "Ahoj!" a "Ahoj po druhé!" na dvou řádcích.

```
Console.WriteLine("Ahoj!");

if (false) {
    Console.WriteLine("Tohle neuvidíš. :/");
}
```

Vypíše "Ahoj!".

Na porovnání často potřebujeme operátor rovnosti (`==`), který nám zkontroluje zda obě strany operátoru mají stejný datový typ a hodnotu. Porovnání je exaktní, takže se musí jednat o úplně stejné data.

```
int number = Convert.ToInt32(Console.ReadLine());

if (number == 666) {
    Console.WriteLine("satanáš!");
}

string name = Console.ReadLine();

if (name == "Yoko") {
    Console.WriteLine("kocourek ^-^");
}
```

Pro podobné účely máme operátor nerovnosti (`!=`) který porovnává opak rovnosti.

```
int number = Convert.ToInt32(Console.ReadLine());

if (number != 666) {
```

```
    Console.WriteLine(":harold:");
}
```

Operátor negace (`!`) je určen pro znegování pravdivostní hodnoty na opačnou hodnotu (tj. `true → false` a `false → true`).

```
if (!false) {
    Console.WriteLine("Tohle vidíš.");
}
```

K negaci se váže i spojování (resp. kombinování) booleanových hodnot, které se hodí nejen k podmínkám. Máme dva základní operátory spojování:

- zároveň, AND, `&&`: všechny spojené části musí být pravdivé
- nebo, OR, `||`: alespoň jedna spojená část musí být pravdivá. Pokud narazíme na pravdivou hodnotu, další se již nekontrolují.

```
const bool isAwesome = true;
const bool isBad = false;

if (isAwesome && !isBad) {
    Console.WriteLine("You're awesome!");
}



const bool isActive = false;
const int points = 180;

if (isActive || points == 200) {
    Console.WriteLine("Active student, I see!");
}
```

Pro spojování můžeme využívat kulaté závorky k určení priority a dokonce samotné výrazy spojovat.

```
string name = Console.ReadLine();
bool isCat = Console.ReadLine() == "true";

// Buď (jméno je Yoko a je kočka) a nebo (jméno je Matej a není kočka)
if ((name == "Yoko" && isCat) || (name == "Matej" && !isCat))
```

```
{
    Console.WriteLine("(^-^)__/ ");
}
```

Pro porovnání čísel máme připravené operátory je větší (`>`), je větší než (`>=`), je menší (`<`), je menší než (`<=`), které porovnávají různé číselné typy mezi sebou. `=` v operátoru znamená, že připouštíme zároveň daný operátor (tj. `>` a nebo `<`) a zároveň i jejich rovnost.

```
double number = Convert.ToDouble(Console.ReadLine());


if (number > 18) { // Dovoluje vše nad 18
    Console.WriteLine("Ameno...");
}
if (number >= 18) { // Dovoluje vše nad 18 včetně
    Console.WriteLine("Ameno^2...");
}
if (number < 27) { // TODO: is this up to date ;)? Dovoluje: vše až do hodnoty 26
    Console.WriteLine("Nemáš na hamburger z bufetu :(");
}
if (number <= 50) { // Dovoluje vše až do hodnoty 50
    Console.WriteLine("Nemáš na hamburger z bufetu :(");
}
```

Podmínky skládající se z jednoho bloku nejsou moc užitečné. V případě, že bychom chtěli udělat podmínku, když něco platí ale zároveň když neplatí (rozdělení jednotlivých možností), tak bychom museli dvakrát blok duplikovat. Což by bylo velice nepřehledné a náchylné na chyby. Proto máme k dispozici bloky `else` a `else if`, které se k podmínkám mohou vázat. `else` musí být jako poslední blok podmínky a určuje, že když podmínka nebyla pravdivá, zavolá se tento blok. `else if` musí být za `if` a nebo za dalším `else if`. `else if` je vlastně pokud předchozí podmínka neprošla, tak postavíme (klidně jiný) výraz znovu do jiné podmínky.

```
int points = 66; // Proč to tady není const? :(


if (points > 66) {
    Console.WriteLine("Jsi bohatý 😎 na bodíky.");
} else {
    Console.WriteLine("Jsi chudý na bodíky.");
```

```
}

points = Convert.ToInt32(Console.ReadLine());

// TODO: ověření vstupu

if (points == 666) {
    Console.WriteLine("😈");
} else if (points > 100) {
    Console.WriteLine("Jsi bohatý 😍 na bodíky.");
} else if (points > 50) {
    Console.WriteLine("Měl bys přidat.");
} else {
    Console.WriteLine("Jsi chudý na bodíky.");
}
```

Iterace

Iterace, často označovány jako cykly, opakování či procházení slouží k opakování určitých příkazů po nějakou dobu. Máme několik typů cyklů, ale obecně platí že každý z nich lze využít ke stejným věcem a jedná se často pouze o zjednodušení (příp. zrychlení) práce.

Cyklus `while`

Jedná se o nejjednodušší cyklus a v podstatě se jedná o opakovatelnou podmínku. Příkazu, stejně jako podmínce, určuje jednu pravdivostní hodnotu, která určuje, zda se provede další cyklus a nebo zda se blok přeskočí a program bude pokračovat.

```
while (true) {
    Console.WriteLine("Yeppers.");
}
```

Výše uvedený kód vytvoří tzv. nekonečný cyklus, který se nikdy nezastaví (max. vypnutím programu / procesu). Tento průchod cyklem se provede tisíckrát za sekundu a proto je docela náročný pro CPU. Cyklus s `while(true)` používáme např. tehdy, když chceme aby program neustále opakoval, ale třeba jej na chvíli uvnitř uspíme - viz níže.

```
int number = 10;

while (number > 0) {
    Console.WriteLine(number);
```

```
    number -= 1;
}
```

Výše uvedený kód vypíše čísla od 10 včetně do 1 včetně. Cyklus se tedy zavolá 10x a poté pokračuje na další příkazy.

```
bool continueProgram = true;

while (continueProgram) {
    // TODO: do something

    Console.Write("Chcete pokračovat [ano]: ");

    string output = Console.ReadLine().ToLower();

    if(output.Length == 0) {
        output = "ano";
    }

    if(output != "ano") {
        continueProgram = false;
    }
}

Console.WriteLine("Nashledanou! :)");
```

Výše uvedený kód je přesně určen tomu, co jsem zmínil výše, a to tedy program opakovací, který však půjde ukončit. Cyklus (kde TODO nahradíme), bude pokračovat neustále, dokud nezadáme cokoliv jiného než `ano`.

Cyklus `do while`

Cyklus je totožný s while, jen je rozdělen do dvou části. Část do se zvolá před zkontrolováním podmínky a každý další cyklus již podmínku kontroluje před spuštěním. Hodí se málo kdy.

```
int i++;

do {
    i++;
} while(i  < 30);
```

Cyklus `for`

V druhé ukázce cyklu while si můžeme všimnout, že takový kód by mohl být velmi dlouhavý. Například když chceme projíždět pole (viz později), bylo by toto procházení velmi místně nepřehledné. Proto vznikl cyklus for, který automaticky bloky slučuje (tj. inicializace proměnné, podmínku a modifikaci proměnné).

```
int number = 10;


while (number > 0) {
    Console.WriteLine(number);


    number -= 1;
}


// Pomocí cyklu for:


for (int i = 10; i > 0; i -= 1) {
    Console.WriteLine(i);
}
```

Cyklus je rozdělen do tří částí:

- inicializátor, kde inicializujeme proměnnou
- podmínka, kde definujeme podmínku, velmi často závislou na té z inicializátoru
- iterátor, kde definujeme změnu proměnné z inicializáru.

Důležité je, si uvědomit, že všechny tři části mohou být prázdné - nevytvoříme proměnnou, podmínka bude prázdná, což znamená, že je pravdivá, a iterátor taktéž necháme prázdný. V tomto případě bude cyklus nekonečný.

Jednotlivé části se volají v pořadí, kdy inicializátor je zavolán před prvním spuštěním, po zavolání cyklu je zvolán blok iterátoru, po kterém je zkontrolována podmínka.

```
for (
    int i = 0; // inicializátor
    i < 100;   // podmínka
    i += 10    // iterátor
) {
    // TODO: come on! do something
}
```

Zbytek po dělení a in/dekrementace

Operátor zbytku po dělení (%) se používá k zjištění, jaké číslo zbyde po dělení číslem druhým. Často se používá k posouzení, zda je číslo liché či sudé.

```
bool isOdd = 45 % 2 != 0;
```

Kde zbytek po dělení (`45 / 2`) je 1, tedy nerovná se 0 a tedy je číslo liché (odd).

Zbytek po dělení lze používat i s necelými čísly a i se zápornými:

```
double  value1 = 4123.123123 % 2;
double  value2 = 4123.5 % -4123.5;
int     value3 = 103 % 3;
double  value4 = -103 % 3.5;
```

Operátor zbytku po dělení nesmí mít na pravé straně 0, protože C# nedovoluje dělit nulou.

Operátor inkrementace (++) a dekrementace (--) slouží k rychlé úpravě proměnné zvětšením či zmenšením čísla o jedna. Používá se zejména v cyklech.

```
int x = 5;
x = x + 1;
x += 1;
x++;
x--;
```

Existují dva typy in/dekrementace:

- **postfixové**, (kde ++/-- je po proměnné), kde je proměnná modifikována až po přečtení v příkazu (např. při vypsání se vypíše aktuální hodnota a poté se zvětší)
- **prefixové**, (kde ++/-- je před proměnnou), kde je proměnná modifikována před přístupem k hodnotě.

```
int x = 1;
Console.WriteLine(x++); // 1
Console.WriteLine(++x); // 3
Console.WriteLine(x);   // 3

Console.WriteLine(x--); // 3
Console.WriteLine(x--); // 2
Console.WriteLine(x++); // 1
```

```
// ZDE je x == 2
```

Kurzor v konzoli

Z minulých stránek již víme, že je konzole vlastně takové pole, do kterých vyplňujeme znaky (a barvy). Nejčastěji pracujeme vertikálně, protože horizontální směr je velmi omezen.

Nejdůležitější částí konzole je kurzor. Kurzor již umím skrýt a pohybovat sním - pomocí metod `WriteLine` a `Write`. Kde oba posunují text na horizontální ose a Line zalamuje. Kurzorem můžeme pohybovat i více programově:

```
Console.CursorLeft = 3;
Console.CursorTop = 14;
```

Pokud zadáme větší číslo, než konzole dovoluje, vyvolá se výjimka.

Konzoli můžeme upravovat i její velikost:

```
Console.WindowHeight = 20;
Console.WindowWidth = 30;
```

Pokud zadáme větší číslo, než konzole dovoluje, vyvolá se výjimka. Pro určení maximální velikosti, které počítač podporuje, můžeme získat jejich hodnoty:

```
Console.WindowHeight = Console.LargestWindowHeight;
Console.WindowWidth = Console.LargestWindowWidth;
```

Práce s řetězci

Při vypisování a nebo obecně prací s řetězci potřebujeme zjednodušit zapisování. Zjednodušení přichází pomocí interpolace řetězců, kde jednodušeji můžeme vložit různé hodnoty z proměnných do stringu. Existují dva typy spojování:

```
const string name = "Vojta";
const int age = 155;
const string school = "SSPŠaG";

string text1 = $"Čau mňau, jmenuji se {name}, je mi {age} a chodím na {school}.";
Console.WriteLine(text1);
```

Výše uvedený kód vypíše do konzole "Čau mňau, jmenuji se Vojta, je mi 155 a chodím na SSPŠaG.". Je nutné si všimnout dolaru ($) na začátku stringu. Složené závorky poté otevírají možnost vložení hodnoty.

```
const string name = "Vojta";
const int age = 155;
const string school = "SSPŠaG";

Console.WriteLine("Čau mňau, jmenuji se {0}, je mi {1} a chodím na {2}.", name, age, school

string text2 = String.Format("Čau mňau, jmenuji se {0}, je mi {1} a chodím na {2}.", name,
Console.WriteLine(text2);
```

V první možnosti výše používáme výpis rovnou do konzole, který vnitřně používá druhou možnost `String.Format`. Do daného stringu v prvním parametru se dle čísel vloží hodnoty poté definované v parametrech. `String.Format` má na více různé další možnosti, ale ty jsou méně používané.

Escape znak, je speciální sekvence s určeným významem. Jeho zápisem se zruší význam následujících znaků a vytvoří se speciální příkaz uvnitř stringu. Základní je zpětné lomítko, které zruší význam dalšího znaku:

```
Console.WriteLine("Ahooooj\nMatěji :sadge:");
Console.WriteLine("A on roboticky řekl \"Coooooooo?\"");
Console.WriteLine("Znak \\ je tzv. escape znak."); // \\ znamená, že se \ do textu vypíše (
```

Existují však i různé další, všechny naleznete ZDE. Dále si můžeme ukázat např. `\n` které na daném místě vytvoří nový řádek, například:

```
Console.Write("smth\n");
// je stejné jako
Console.WriteLine("smth");
```

Pole

Pole je způsob uložení velkého množství věcí stejného datového typu. Pole musí mít vždy definovanou délku a je uloženo vždy v proměnné s jménem. Data jsou v operační paměti uložené za sebou (a proto potřebujeme dopředu vědět velikost) a můžeme je postupně číst pomocí indexů.

Indexování začíná od čísla nuly a jde do velikosti pole bez jedné, tj. `0 do n-1`.

```
short[] array1 = new short[5];
```

Výše uvedený kód do proměnné array1 uloží pole o 5 prvcích, platné indexy jsou 0, 1, 2, 3 a 4.

Pole je po vytvoření plné základní hodnot, tyto hodnoty jsou např. pro číselné typy 0, pro string prázdný řetězec a u většiny typů lze základní hodnotu odhadnout.

Přistupovat k hodnotám v poli můžeme pomocí operátoru přístup v poli (`[i]`) s indexem:

```
short[] array1 = new short[5];

short value = array1[0];

Console.WriteLine(array1[0]);
```

Záznamy můžeme upravovat i přiřazením:

```
short[] array1 = new short[5];

Console.WriteLine(array1[4]);

array1[4] = 43;

Console.WriteLine(array1[4]);
```

Přístupem na špatný index však vyvoláme výjimku `System.IndexOutOfRangeException: Index je mimo hranice pole`. Můžeme ji však zachytit (stejně jako u konverzí):

```
short[] array1 = new short[5];

try {
    short value = array1[5];
} catch (Exception) {
    Console.WriteLine("Špatné čtení!");
}
```

Obecně je však lepší index validovat pomocí podmínek 0 <= i <= array.Length - 1. Na poli máme tuto vlastnost k dispozici:

```
short[] array1 = new short[12];

Console.WriteLine(array1.Length); // === "12"
```

Důležité je si taky uvědomit, proč se vlastně pole používají - jsou skvělým nástrojem pro ulehčení práci. Představte si, že byste museli pro každé zadané jméno mít speciální proměnnou, která je ještě velmi omezena tím, že má jméno, které nemůžeme referencovat uživatelem (musíme tvořit otázky). Polem vše zjednodušíme:

```
string name1 = "";
string name2 = "";
string name3 = "";
string name4 = "";
string name5 = "";

if(wantsToEditName1) {
    name1 = edittedName;
}

if(wantsToEditName2) {
    name2 = edittedName;
}
// ...
```

```
string[] names = new string[5];

int toEdit = int.Parse(Console.ReadLine());

if(toEdit < 0) {
    toEdit = 0;
}

if(toEdit >= names.Length) {
    toEdit = names.Length - 1;
}

names[toEdit] = Console.ReadLine();
```

Pole můžeme rychle procházet již známými cykly:

```
short[] array1 = new short[5];

int i = 0;
while(i < array1.Length)
{
    Console.Write(array1[i]);

    i++;
}
```

```
short[] array1 = new short[5];

for(int i = 0; i < array1.Length; i++)
{
    Console.Write(array1[i]);
}
```

Máme však k dispozici nový cyklus typu `foreach`, který slouží právě pro procházení pole. Určujeme mu vždy jaké pole čteme a do jaké proměnné se uloží.

```
short[] array1 = new short[5];

foreach(short item in array1)
{
    Console.Write(item);
}
```

Oproti ostatním typům cyklu zde neurčujeme jak pole procházíme, ale prostě pouze jdeme jedním směrem.

Možnosti iterace

Ve všech typech cyklů máme k dispozici různé příkazy na úpravu procházení cyklu. Jedná se o příkazy `break` a `continue`.

Při zavolání příkazu `break` uvnitř cyklu se aktuální průchod a celý cyklus ukončí a pokračuje se s příkazy poté definovanými.

```
while(true) {
    string answer = Console.ReadLine();
```

```
    if(answer == "ano") { // Zjednodušení z minulé ukázky nekonečného programu z dřívějška.
        break;
    }
}
```

Při zavolání příkazu `continue` se aktuální průchod ukončí a pokud může cyklus pokračovat, tak bude pokračovat.

```
while(true) {
    Console.Write("Zadejte číslo: ");
    try {
        int number = int.Parse(Console.ReadLine());
    } catch(Exception) {
        continue; // Uživatel zadal blbost, tak ho necháme zadat znovu.
    }

    Console.Write("Chcete pokračovat? ");
    // ...
    Console.WriteLine();
}
```

Práce s poli

Pole můžeme rychle inicializovat pomocí zkráceného zápisu. Hodí se, když potřebujeme pole rychleji vyplnit (resp. nepotřebujeme je nějak upravovat na bázi uživatele).

```
string[] names = new string[] { "Yoko", "Bambi", "Matěj" };
```

Pro různé akce nad poli máme vytvořenou třídu `Array` s několika metodami.

Pole můžeme pomocí metody .Sort seřadit abecedně nebo dle číselné hodnoty. Zavoláním metody změníme původní pole.

```
string[] names = new string[] { "Yoko", "Bambi", "Matěj" };
Array.Sort(names);

foreach (string name in names) {
    Console.WriteLine(name);
```

```
}

int[] grades = new int[] { 1, 1, 3, 5, 1, 2, 5 };
Array.Sort(grades);

foreach(int grade in grades) {
    Console.WriteLine(grade);
}
```

Pole můžeme prohodit pomocí metody .Reverse, tedy se zmapují index na jejich opačné místo v poli jiném podle `iN = length - iO`.

```
int[] grades = new int[] { 1, 1, 3, 5, 1, 2, 5 };

Array.Sort(grades);
Array.Reverse(grades);

foreach(int grade in grades) {
    Console.WriteLine(grade);
}
```

Často potřebujeme vyhledat, na jakém indexu v poli najdeme nějakou hodnotu. To umí funkce `.IndexOf`, která přijímá pole a hodnotu, kterou má vyhledat. V případě, že je v poli více hodnot, vrátí se první index. V případě že se hodnota v poli nenachází, vrátí se hodnota `-1`.

```
int[] grades = new int[] { 1, 1, 3, 5, 1, 2, 5 };

Array.Sort(grades);
Array.Reverse(grades);

// Pole je: 5 5 3 2 1 1 1
// Inexy:   0 1 2 3 4 5 6

Console.WriteLine(Array.IndexOf(grades, 1)); // 4
Console.WriteLine(Array.IndexOf(grades, 3)); // 2
Console.WriteLine(Array.IndexOf(grades, 5)); // 0
Console.WriteLine(Array.IndexOf(grades, 7)); // -1
```

Rozdělení textových řetězců

Při programování je často nutné textový řetězec rozdělit do pole dle nějakého rozdělovače. Přímo na stringu existuje metoda Split, která přijímá string a nebo char hodnotu, podle které vytvoří pole, kde každá položka je část stringu.

```
string[] parts = "Yoko-mrkvička-Bambi-myška".Split('-');

foreach (string part in parts) {
    Console.WriteLine(part); // Vypíše postupně: "Yoko", "mrkvička", "Bambi" a "myška"
}

parts = "Yoko-mrkvička-Bambi-myška".Split("-Bambi-");

foreach (string part in parts) {
    Console.WriteLine(part); // Vypíše postupně: "Yoko-mrkvička" a "myška"
}
```

PS: v .NET Framework je možné rozdělovat pouze pomocí znaku, podle stringu je to komplikovanější.