**PRG otázky**

**Obsah**

[**1. Tvorba programu – algoritmizace a programování 4**](#_heading=h.1mrcu09)

[Postup při řešení problému 4](#_heading=h.46r0co2)

[Definice a vlastnosti algoritmu 5](#_heading=h.2lwamvv)

[Možnosti zápisu algoritmu 5](#_heading=h.111kx3o)

[Zpracování programu v jazyce C 7](#_heading=h.3l18frh)

[**2. Řídící struktury – větvení 8**](#_heading=h.206ipza)

[Typy větvení 8](#_heading=h.4k668n3)

[Grafický zápis větvení 9](#_heading=h.2zbgiuw)

[**3. Řídící struktury – cykly 12**](#_heading=h.1egqt2p)

[Typy cyklů 12](#_heading=h.3ygebqi)

[Break a Continue 13](#_heading=h.2dlolyb)

[Grafický zápis cyklů 14](#_heading=h.sqyw64)

[**4. Datové typy (C, Java) 16**](#_heading=h.3cqmetx)

[Rozdělení datových typů v jazyce C 16](#_heading=h.1rvwp1q)

[Rozdělení datových typů v jazyce Java 17](#_heading=h.4bvk7pj)

[Deklarace jednotlivých DT a charakteristika jejich vlastností 17](#_heading=h.2r0uhxc)

[Definování nových datových typů pomocí typedef 20](#_heading=h.1664s55)

[**5. Proměnné v jazyce C 22**](#_heading=h.3q5sasy)

[Charakteristika a deklarace lokálních a globálních proměnných 22](#_heading=h.25b2l0r)

[Typy, charakteristika a význam paměťových tříd v jazyce C 22](#_heading=h.kgcv8k)

[**6. Projekty, zdrojové a hlavičkové soubory v jazyce C 24**](#_heading=h.34g0dwd)

[Stavba vlastního hlavičkového souboru 25](#_heading=h.1jlao46)

[Projekty v programovacím jazyce C 25](#_heading=h.43ky6rz)

[Rozdíl mezi vkládáním souborů a odděleným překladem 25](#_heading=h.2iq8gzs)

[Spojování a překlad programu 26](#_heading=h.xvir7l)

[**7. Struktura, union, výčet v jazyce C 28**](#_heading=h.3hv69ve)

[Deklarace a charakteristika datového typu struktura 28](#_heading=h.1x0gk37)

[Deklarace a charakteristika datového typu struktura 29](#_heading=h.4h042r0)

[Deklarace a charakteristika datového typu enum 29](#_heading=h.2w5ecyt)

[**8. Operátory (C, Java) 30**](#_heading=h.1baon6m)

[Rozděleni operátorů 30](#_heading=h.3vac5uf)

[Ternární operátor 32](#_heading=h.2afmg28)

[Priorita a pořadí vyhodnocování 32](#_heading=h.pkwqa1)

[**9. Funkce (C) 34**](#_heading=h.39kk8xu)

[Stavba a vlastnosti funkce 34](#_heading=h.1opuj5n)

[Rekurzivní funkce 35](#_heading=h.48pi1tg)

[**10. Činnost preprocesoru (C) 36**](#_heading=h.2nusc19)

[Charakteristika činnosti preprocesoru 36](#_heading=h.1302m92)

[Makra 36](#_heading=h.3mzq4wv)

[Rozdíl mezi funkcemi a makry 37](#_heading=h.2250f4o)

[Podmíněný překlad 37](#_heading=h.haapch)

[Hlavičkové soubory 38](#_heading=h.319y80a)

[**11. Pointery (C) 40**](#_heading=h.1gf8i83)

[Deklarace, charakteristika a využití pointerů 40](#_heading=h.40ew0vw)

[Dynamické přidělování paměti v jazyce C 40](#_heading=h.2fk6b3p)

[Pointerová aritmetika 42](#_heading=h.upglbi)

[Pointery a funkce 43](#_heading=h.3ep43zb)

[**12. Jednorozměrné pole (C) 44**](#_heading=h.1tuee74)

[Deklarace a charakteristika statického pole 44](#_heading=h.4du1wux)

[Deklarace a charakteristika dynamického pole 44](#_heading=h.2szc72q)

[Pole jako parametr funkce 45](#_heading=h.184mhaj)

[Třídící algoritmy 45](#_heading=h.3s49zyc)

[**13. Vícerozměrné pole (C) 50**](#_heading=h.279ka65)

[Deklarace a charakteristika statického VP 50](#_heading=h.meukdy)

[Deklarace a charakteristika dynamického VP 50](#_heading=h.36ei31r)

[**14. Znaky a řetězce (C) 52**](#_heading=h.1ljsd9k)

[Deklarace a charakteristika znakové proměnné 52](#_heading=h.45jfvxd)

[Charakteristika řetězce v jazyce C 52](#_heading=h.2koq656)

[Základy práce s řetězci 52](#_heading=h.zu0gcz)

[Funkce pro práci s řetězci 53](#_heading=h.3jtnz0s)

[Parametry funkce main() 54](#_heading=h.1yyy98l)

[**15. Soubory (C) 56**](#_heading=h.4iylrwe)

[Textové a binární soubory 56](#_heading=h.2y3w247)

[Práce se soubory 56](#_heading=h.1d96cc0)

[Souborové funkce 58](#_heading=h.3x8tuzt)

[Testování otevření a uzavření souboru 59](#_heading=h.2ce457m)

[**16. Základy OOP, třídy a objekty (Java) 60**](#_heading=h.rjefff)

[Strukturované programování, OOP 60](#_heading=h.3bj1y38)

[Charakteristika pojmu objekt a třída 60](#_heading=h.1qoc8b1)

[Vlastnosti OOP – zapouzdření, dědičnost, polymorfismus 61](#_heading=h.4anzqyu)

[**17. Funkce (Java) 62**](#_heading=h.2pta16n)

[Stavba a vlastnosti funkce (metody) 62](#_heading=h.14ykbeg)

[Přetížení funkcí (metod) 62](#_heading=h.3oy7u29)

[Statické funkce (metody) 62](#_heading=h.243i4a2)

[Speciální metody 63](#_heading=h.j8sehv)

[**18. Pole a ArrayList (Java) 64**](#_heading=h.338fx5o)

[Charakteristika a práce s jednorozměrným polem 64](#_heading=h.1idq7dh)

[Charakteristika a práce s vícerozměrným polem 64](#_heading=h.42ddq1a)

[Třídění v poli 65](#_heading=h.2hio093)

[Kolekce (collections) – charakteristika 65](#_heading=h.wnyagw)

[Charakteristika a práce s ArrayListem 66](#_heading=h.3gnlt4p)

[**19. Znaky a řetězce (Java) 68**](#_heading=h.1vsw3ci)

[Deklarace a charakteristika znakové proměnné 68](#_heading=h.4fsjm0b)

[Charakteristika a práce s řetězci 68](#_heading=h.2uxtw84)

[Metody pro práci s řetězci 69](#_heading=h.1a346fx)

[**20. Soubory (Java) 70**](#_heading=h.3u2rp3q)

[Práce s adresáři a soubory – třída File 70](#_heading=h.2981zbj)

[Znakové proudy – třídy Reader, Writer 70](#_heading=h.odc9jc)

[Binární proudy – třídy InputStream, OutputStream 71](#_heading=h.38czs75)

[**21. Třídy a objekty – dědičnost (Java) 74**](#_heading=h.1nia2ey)

[Dědičnost a kompozice 74](#_heading=h.47hxl2r)

[Konstruktory v dědičnosti 74](#_heading=h.2mn7vak)

[Přístupová práva a dědičnost 75](#_heading=h.11si5id)

[Klíčové slovo final 76](#_heading=h.3ls5o66)

[**22. Třídy a objekty – rozhraní (Java) 78**](#_heading=h.20xfydz)

[Rozhraní a jeho konstrukce 78](#_heading=h.4kx3h1s)

[Abstraktní třída 79](#_heading=h.302dr9l)

[Využití rozhraní a abstraktní třídy 79](#_heading=h.1f7o1he)

[**23. Java a databáze – DDL, DML 80**](#_heading=h.3z7bk57)

[Nastavení projektu 80](#_heading=h.2eclud0)

[Připojení k databázi 80](#_heading=h.thw4kt)

[DDL 80](#_heading=h.3dhjn8m)

[DML 81](#_heading=h.1smtxgf)

[**24. Java a databáze – SELECT 84**](#_heading=h.4cmhg48)

[Nastavení projektu 84](#_heading=h.2rrrqc1)

[Připojení k databázi 84](#_heading=h.16x20ju)

[SELECT 84](#_heading=h.3qwpj7n)

[**25. Tvorba grafického rozhraní aplikací, rozmístění (JavaFX) 86**](#_heading=h.261ztfg)

[Soubory pro tvorbu JavaFX Application 86](#_heading=h.l7a3n9)

[Základní pojmy – Stage, … 87](#_heading=h.356xmb2)

[Rozložení 88](#_heading=h.1kc7wiv)

[Práce s komponentami 88](#_heading=h.44bvf6o)

[**26. Tvorba grafického rozhraní aplikací, Scene Builder (JavaFX) 90**](#_heading=h.2jh5peh)

[Soubory pro tvorbu JavaFXML Application 90](#_heading=h.ymfzma)

[Prostředí Scene Builder 90](#_heading=h.3im3ia3)

[Práce s komponentami 90](#_heading=h.1xrdshw)

[**27. Komponenty ze skupiny Controls – charakteristika a použití, základní vlastnosti a události (JavaFX) 92**](#_heading=h.4hr1b5p)

[Komponenty Button, CheckBox, RadioButton 92](#_heading=h.2wwbldi)

[Komponenty Label, TextField, TextArea 93](#_heading=h.1c1lvlb)

[Komponenty ListView, ComboBox 94](#_heading=h.3w19e94)

[Komponenta ColorPicker 95](#_heading=h.2b6jogx)

[**28. Komponenty ze skupiny Menu – charakteristika a použití, základní vlastnosti a události (JavaFX) 96**](#_heading=h.qbtyoq)

[Komponenta MenuBar ze skupiny Controls 96](#_heading=h.3abhhcj)

[Komponenty Menu, MenuItem 96](#_heading=h.1pgrrkc)

[Komponenty CheckMenuItem, RadioMenuItem 97](#_heading=h.49gfa85)

[Komponenta Separator 97](#_heading=h.2olpkfy)

[**29. Komponenty pro práci s databází – charakteristika a použití, základní vlastnosti a události (JavaFX) 100**](#_heading=h.13qzunr)

[Připojení k databázi 100](#_heading=h.3nqndbk)

[Komponenta TableView 100](#_heading=h.22vxnjd)

[Komponenta TableColumn 100](#_heading=h.i17xr6)

[Komponenta TableCell 101](#_heading=h.320vgez)

[**30. Třída FileChooser – charakteristika a použití, základní vlastnosti a metody (JavaFX) 102**](#_heading=h.1h65qms)

[Vlastnosti 102](#_heading=h.415t9al)

[Použití 102](#_heading=h.2gb3jie)

# 1. Tvorba programu – algoritmizace a programování

* Postup při řešení problému
* Definice a vlastnosti algoritmu
* Možnosti zápisu algoritmu
* Zpracování programu v jazyce C

Algoritmus je postup, kterým lze vyřešit daný typ úlohy. Objevuje se nejčastěji v oblasti programování, kdy se tím myslí teoretický princip řešení problému.

## Postup při řešení problému

1. Formulace problému
2. Analýza úlohy
3. Vytvoření algoritmu
4. Sestavení programu
5. Odladění programu

**Formulace problému**

Je třeba přesně formulovat požadavky, určit výchozí hodnoty, požadované výsledky, jejich formu a přesnost řešení. Tvůrce algoritmu musí dokonale rozumět řešenému problému, jinak nemůže algoritmus sestavit – v praxi programátoři spolupracují s odborníky z oblastí, pro které mají vytvořit algoritmus.

**Analýza úlohy**

Při analýze úlohy si ověříme, zda je úloha řešitelná a uděláme si první představu o jejím řešení. Dále zjistíme, zda výchozí hodnoty jsou k řešení postačující a zda má úloha více řešení. Podle charakteru úlohy vybereme nejvhodnější a pokud možno nejjednodušší řešení.

**Vytvoření algoritmu**

Sestavíme jednoznačný sled jednotlivých instrukcí (příkazů), které je třeba provést, aby byl úkol správně vyřešen. Algoritmus přesně popisuje postup zpracování daného úkolu, nedává však odpověď na daný problém, ale pouze postup, jak ji získat.

**Sestavení programu**

Na základě algoritmu řešené úlohy sestavíme program (zdrojový kód) v konkrétním programovacím jazyce. Ze zdrojového kódu se pomocí překladače přeloží do strojového kódu a vytvoří se spustitelný program.

**Odladění programu**

Odladěním chceme odstranit chyby z programu. Nejčastější chyby jsou chyby v zápise, tzv. syntaktické – ty odhalí překladač a dělají je i zkušení programátoři.

Horší jsou logické chyby, které vyplývají z nesprávně navrženého algoritmu, nebo chyby, které vzniknou špatným předpokladem v etapě formulace nebo analýzy úlohy – projeví se nesprávnou činností programu nebo špatnými výsledky – při odstraňování těchto chyb může pomoci ladící program (debugger) umožňující sledování aktuálního stavu proměnných a krokování.

## Definice a vlastnosti algoritmu

Algoritmem jsou pouze takové postupy, které splňují přesně stanovené požadavky – vlastnosti algoritmu.

**Vlastnosti algoritmu: *HERDEK***

1. **Elementárnost –** jednoduché a srozumitelné kroky
2. **Determinovanost –** vždy musí být jednoznačně určen následný krok
3. **Konečnost –** musí končit v reálném čase
4. **Rezultativnost –** vždy dostaneme nějaký výsledek
5. **Efektivnost –** co nejméně kroků za nejkratší čas
6. **Hromadnost –** řeší všechny úlohy stejného typu

## Možnosti zápisu algoritmu

1. Slovní zápis
2. Matematický zápis
3. Grafický zápis
4. V programovacím jazyce

**Slovní zápis**

např. kuchařský recept

Výhody:

* vhodné pro naučení konceptů a pro dokumentaci
* není třeba znalostí v oboru

Nevýhody:

* nejméně přehledný
* nemá nástroje na hlídání jednoznačnosti, přesnosti a srozumitelnosti nebo jestli vede algoritmus k cíli
* hodně textu
* nestrukturovatelnost

**Matematický zápis**

Je vhodný tam, kde je možné danou problematiku popsat pomocí matematických vztahů.

Např: Určení kořenů kvadratické rovnice.

Výhody:

* je jednoznačný a člověk znalý úprav matematických výrazů jednoznačně určí, za jakých podmínek je možno úlohu řešit.

Nevýhody:

* ve většině případů bývá příliš stručný a nelze jej přímo zadat počítači

**Grafický zápis**

Symbolický, algoritmický jazyk, který se používá pro názorné zobrazení algoritmu. Je to jednoznačný komunikační prostředek při týmové práci, je přehledný a má nástroje k dodržení algoritmu. Výborně se hodí k dokumentačním účelům, kde je přehlednější než výpis programu. Nevýhodou může být, že mu laik nemusí rozumět.

Výhody:

* přehlednost u jednodušších a kratších algoritmů
* jednoduché na převedení do kódu

Nevýhody:

* v případě delších algoritmů může být nepřehledný

Značky grafického algoritmu:Obsah obrázku diagram, text, skica, Technický výkres

Popis byl vytvořen automaticky

**Programovací jazyk**

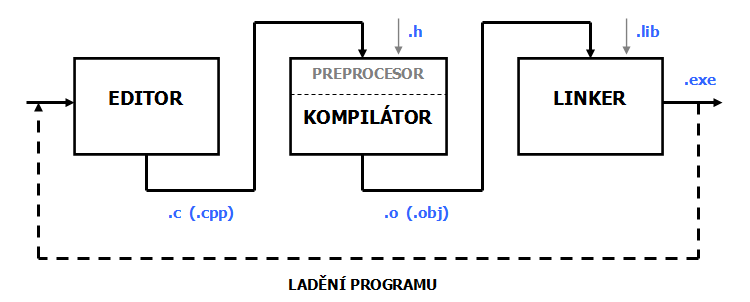
Výhody:

* je to jediná forma, které rozumí člověk (programátor) i počítač (pokud je vybaven překladačem)
* Tato forma se nedá ničím obejít ani nahradit

Nevýhody:

* Srozumitelné pouze pro programátora
* Je málo názorná a přehledná pro dokumentační účely

## Zpracování programu v jazyce C



**EDITOR**

Program v jazyce C, tzv. zdrojový kód, můžeme napsat i v jednoduchém textovém editoru (jako třeba v poznámkovém bloku). Jeho výstupem je soubor s příponou c. V názvech souborů je vhodné nepoužívat české znaky a mezery.

**PREPROCESOR**

Částečně upraví zdrojový kód. Vkládá hlavičkové soubory, odstraňuje komentáře, rozvíjí makra a provádí podmíněný překlad. Preprocesor je součástí compileru.

**COMPILER (PŘEKLADAČ)**

Překladač přeloží příkazy, vznikne tzv. relativní kód (s příponou obj nebo o). Jedná se o kód, který je srozumitelný pro počítač. Relativní znamená, že adresy proměnných a funkcí ještě nejsou vůbec známy. V případě, že se v programu vyskytnou formální chyby, je nutné je opravit a kód znovu zkompilovat.

**LINKER (SESTAVOVACÍ PROGRAM)**

Poslední fází je propojení relativního kódu a knihovních souborů (.lib). Výsledkem je spustitelný soubor (ve Windows s příponou exe). Přidělí relativním adresám absolutní adresy a provede všechny odkazy na dosud neznámé identifikátory - připojí knihovny (.a, .o).

Pokud spustitelný program nefunguje podle našich představ, musíme program přepsat.Procesu hledání logických chyb říkáme ladění programu. K tomu je možné využít ladící program – tzv. debugger.

**DEBUGGER**

Hledá chyby, které nastanou při běhu programu. Je určený pro krokování programu, umožňuje tak sledovat řádek po řádku a sledovat hodnoty proměnných, registry a místo v paměti.

# 2. Řídící struktury – větvení

* Typy a charakteristika podmíněných příkazů
* Grafický zápis a zápis v jazyce C, Java

## Větvení

* způsob jakým program za běhu může zjistit, kterou sekvenci příkazů vykonat.
* na úrovni procesoru je vykonáváno přes skokové instrukce = GOTO
* compiler optimalizuje podmínky (pokud nějaká větev 100% nenastane tak se ani nezkompiluje)
* V případě složených podmínek, jakmile se při vykonávání výrazu zjistí, že výsledek podmínky už je znám, tak vykonávání výrazu nepokračuje dál.
* např.: **false && (metoda())** nebo **1 == 1 || metoda()**
* metoda není nikdy vykonána, protože výsledek podmínky už je znám
* **typy větvení:**

1. Úplné
2. Neúplné
3. Vnořené
4. Několikanásobné

**Úplné větvení**

* obsahuje kladnou i zápornou větev
* má právě dvě větve
* vždy se provede právě jedna větev

if(podmínka) {  
  příkaz1;  
}

else {  
  příkaz2;  
}

Podmínky se vyhodnocují jedna za druhou do té doby, než se narazí na první pravdivou (TRUE). Pak se provedou příkazy v bezprostředně následujícím bloku. Další podmínky se již nevyhodnocují. Pokud se žádná podmínka nevyhodnotí jako TRUE, pak se provede tělo bloku za else.

Patří zde i **ternární operátor**:

(podmínka)? příkazAno : příkazNe;

Zde otazník za výrazem v závorkách nahrazuje klíčové slovo if, výraz před dvojtečkou reprezentuje kladnou větev a výraz za ní větev zápornou.

### Neúplné větvení

* Větev else není povinná.

if(podmínka) {  
  příkaz1;  
}

### Vnořené větvení

if(podmínka) {  
  if(podmínka) {  
   příkaz2; }

else {  
  příkaz3; }}

else {  
 příkaz4;}

**Několikanásobné větvení**

* Využívá se zde switch

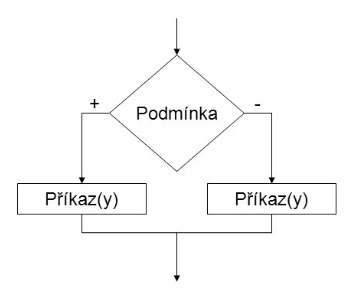
switch(*výraz*) {  
  case x:  
    *//codeBlock*  
    break;  
  case y:  
    *//codeBlock*  
    break;  
  default:  
    *//codeBlock*  
}

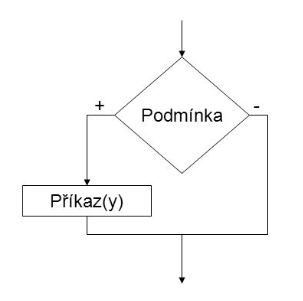
*výraz* = číslo (char)

Příkaz break provádí ukončení příkazu switch, bez něj by se provedly i ostatní příkazy, dokud by program nenarazil na další break.

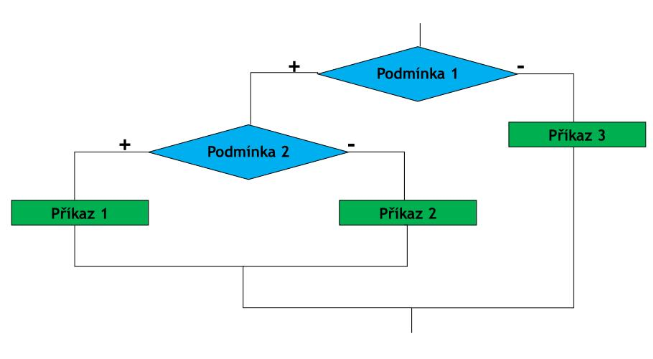
Větev default není povinná. Provede se v případě, když žádná hodnota nevyhovuje selektoru. Pokud se žádná hodnota neshoduje se selektorem a není definovaná větev default, dochází k ukončení větvení.

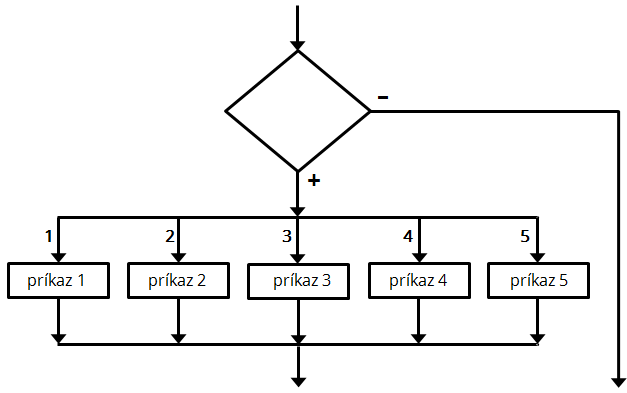
## Grafický zápis větvení

**Úplné větvení** **Neúplné větvení**



**Vnořené větvení**



**Několikanásobné větvení**



# 3. Řídící struktury – cykly

* Typy a charakteristika cyklů
* Grafický zápis a zápis v jazyce C
* Příkazy break a continue

= část programu, kterou se prochází tak dlouho, dokud je splněna zadaná podmínka. V jazyce C jsou definovány tři druhy cyklů.

V cyklech můžeme použít příkazy *break* a *continue*. Oba příkazy způsobí ukončení provádění cyklu, s tím rozdílem, že příkaz *break* ukončí právě probíhající smyčku i celý cyklus, kdežto příkaz *continue* ukončí provádění probíhající smyčky a provede odskok na podmínku cyklu (jestliže je podmínka splněna, pokračuje se další smyčkou).

## Typy cyklů

1. Cyklus s podmínkou na začátku
2. Cyklus s podmínkou na konci
3. Cyklus s řídící proměnou

### Cyklus s podmínkou na začátku

* Testování podmínky se provádí na začátku cyklu
* před každou iterací je podmínka testována, či byla splněna.
* Už před první spuštěním dojde k otestování podmínky
* Použití tohoto cyklu je vhodné v situacích, kdy je možné, že ani jednou nebude potřeba provést cyklus.

while(*podmínka*) {  
 *//*příkazy

}

### Cyklus s podmínkou na konci

* Příkaz (nebo blok příkazů) se provede vždy alespoň jednou, podmínka se kontroluje až poté.
* Lze jej využít např. u čtení vstupu z konzole. Představte si, že chceme, aby uživatel zadal číslo větší než 2. Ale co když zadá jiné? Není úplně ideální ukončit program po prvním špatném zadání. Lepší bude ptát se ho tak dlouho, až odpoví správně

do {  
*//*příkazy}  
while(*podmínka*);

### Cyklus s řídící proměnou

* Typicky se cyklus skládá z inicializátoru, podmínky, příkazu a těla cyklu.
* Nejčastěji tento cyklus použijeme k::
  + procházení pole
  + výpočtu faktoriálů
  + vykonání nějakého kódu s předem známým počtem opakování.

**for**(inicializátor; podmínka; příkaz){

// kod tady

}

* **Inicializátor** je místo, kde se nejčastěji definuje řídící proměnná cyklu. Té nastavíme počáteční hodnotu (nejčastěji 0), např. tedy i = 0.
* **Podmínka** je podmínka vykonání dalšího kroku cyklu. Jakmile nebude platit, cyklus se ukončí.
* **Příkaz** nám říká, co se má v každém kroku s řídící proměnnou stát. Tedy zda se má zvýšit nebo snížit. K tomu využijeme operátorů ++ a --. Nemusí se jen zvyšovat/snižovat o jedno, je možné použít += nebo -= pro zvýšení/snížení řídící proměnné o libovolnou hodnotu, nebo příkaz vůbec nemusí pracovat s řídící proměnnou, nebo můžeme příkaz úplně vynechat.

for(i = 0; i < 5; i++)

{

...

}

## Break a Continue

Příkaz break slouží k okamžitému ukončení aktuálního cyklu. Nezpůsobí vyskočení z vnořených cyklů – vždy se vztahuje pouze na 1 nejbližší cyklus. Pokud máme několik vnořených cyklů a chceme vyskočit ze všech najednou, můžeme použít GOTO.

for(i = 0; i < 20; i++) {

if(i\*i > 130)

break;

}

Continue ukončí jen aktuální průběh cyklem.

for(i = 2, a = 1; i < 10; i++) {

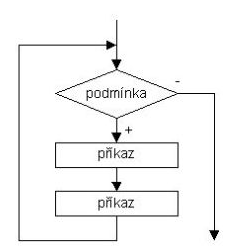
if(i%3 == 0)

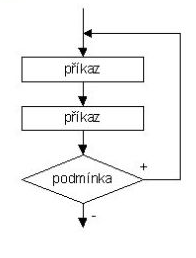
continue;

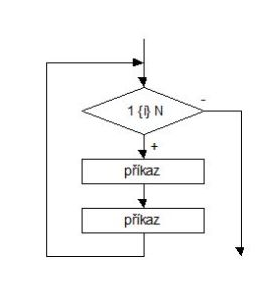
a \*= i;

}

## Grafický zápis cyklů

**Cyklus s podmínkou na začátku**

**Cyklus s podmínkou na konci**

**Cyklus s řídicí proměnou**



# 4. Datové typy (C, Java)

* Rozdělení datových typů v jazyce C a Java
* Deklarace jednotlivých datových typů a charakteristika jejich vlastností
* Definování nových datových typů pomocí typedef

Datový typ je množina hodnot, kterých může proměnná nabývat a také množina operací, které může vykonávat. Bývá deklarován v programu a zabírá místo v paměti.

## Rozdělení datových typů v jazyce C

Jazyk C je staticky\* typovaný jazyk, kde musíme vždy každé proměnné určit nejen její název, ale i typ. Pokud použijeme například celočíselný datový typ int, pak můžeme říct, že se hodnotami tohoto typu smí provádět všechny základní aritmetické operace.

*\*při kompilaci kódu jsou všechny datové typy známy*

### Jednoduché datové typy

Jsou definovány identifikátorem a datovým typem v deklarační oblasti. V paměťovém prostoru je uložena přímo hodnota této proměnné/konstanty.

Dělí se na ordinální a neordinální.

Hodnoty ordinálního typu tvoří lineárně uspořádanou množinu, kde pro každý prvek je přesně definovaný předchůdce i následovník.

Základní ordinální typy:

* int (celá čísla, 2 až 4 byty, -32768 . . . +32767)
* char (1 byte)
* enum (pouze jazyk C, v Javě je enum složený datový typ)

Neordinální datové typy:

* double
* float (reálná čísla, 4 byty)
* void

### Strukturované datové typy

Strukturované datové typy se dělí na heterogenní a homogenní

#### Homogenní

* **Pole** = kolekce prvků stejného typu

#### Heterogenní

* **Struktura** = složená z řady údajů různého typu
* **Union** = typ umožňující v jednom místě v paměti uložit jednu hodnotu více různých datových typů (vždy však výhradně pouze jeden typ zároveň)
* **FILE** = vestavěná struktura pro práci se soubory

## Rozdělení datových typů v jazyce Java

Protože jazyk Java vychází z jazyka C, mnoho datových typů je převzatých a liší se obvykle jen velikostí. Na rozdíl od jazyku C je ale Java OOP jazyk.

### Primitivní datové typy

Primitivní datové typy reprezentují elementární údaje, je jich omezené množství a nedají se uživatelsky definovat.

* Celočíselné
* Reálné
* char
* boolean
* void

| **název** | **druh** | **velikost** |
| --- | --- | --- |
| **byte** | celé číslo | 8b (28) |
| **short** | celé číslo | 16b (216) |
| **int** | celé číslo | 32b (232) |
| **long** | celé číslo | 64b (264) |
| **float** | reálné číslo | 32b (232) |
| **double** | reálné číslo | 64b (264) |
| **char** | znak | 16b (216) |
| **boolean** | logická hodnota | 8b \* |

*\* nejmenší paměťová jednotka, se kterou dokáží moderní architektury pracovat je jeden byte (8b), proto i boolean zabírá 8b, ač obsahuje pouze jeden bit informace. (V některých případech je využita tzv. boolean komprese)*

Na rozdíl od jazyka C není v Javě datový typ boolean považován za číslo. Může nabývat hodnot true nebo false, nikoliv 1 nebo 0.

Konstanta typu long jde explicitně zapsat tak, že za číslo přidáme znak ‘l‘ nebo ‘L‘, např:

**var** x = 1234L; //x je long

Pokud číslo začnete číslicí 0, znamená to, že je v osmičkové soustavě. Pokud číslo začnete znaky 0x, je číslo v šestnáctkové soustavě.

**int** x = 123; //desítková soustava

**int** y = 0123; //osmičková soustava

**int** z = 0x12A; //šestnáctková soustava

### Referenční datové typy

Jedná se o objekty (třídy) a pole. Hodnota referenční proměnné je odkaz (reference) do paměti na místo, kde je objekt (nebo pole) uložen. V porovnání s C se zde místo pointerů používají referenční proměnné.

## Deklarace jednotlivých DT a charakteristika jejich vlastností

### Celočíselné datové typy

Dělí se na dvě skupiny. Se znaménkem obsahují …, -2,-1,0,1,2, … (signed, který je výchozí) a bez znaménka obsahují 0,1,2,3,4, . . . (unsigned).

**Unsigned není v Javě.**

Deklarace:

**int** a = -1;

**unsigned int** a = 1;

Znak je v počítači reprezentován pomocí celého čísla, proto se pro kódování používá ASCII tabulka. (V dnešní době se používá spíše utf8.)

Deklarace:

**char** znak = ‘A‘;

### Výčtový datový typ enum

Jde o datový typ definovaný programátorem. Pokud nepřiřadíme hodnotu první konstantě, počítač automaticky přiřadí nulu, další prvek pak má 1 atd. . .

**typedef enum** {CERVENA, MODRA, ZELENA} **barvy**;

### Reálné datové typy

Float a double se používá pro práci s reálnými čísly s desetinnou čárkou. Reálné datové typy se nesmí napřímo porovnávat operátorem ==.

Deklarace:

**float** a = 0.1f;

**double** a = 0.1; // je přesnější

**Pole**

Jednorozměrná a vícerozměrná pole. Výhodou pole je okamžitý přístup k položce pomocí indexu prvku pole.

Deklarace pole o pěti prvcích:

**int** pole[5];

### Textový řetězec

= pole znaků (v C)

Řetězec obsahuje znaky a do indexu se udává počet znaků + 1 navíc. Posledním prvkem pole řetězce je ukončovací znak \0.

Deklarace:

**char** pozdrav[5] = {'a', 'h', 'o', 'j', '\0'};

**char** pozdrav[] = “ahoj”; // ekvivalentní zápis

### Soubory

Textové nebo binární.

Deklarace:

**FILE** \*soubor;

soubor = fopen("filename", "mode");

### Pointer

Ukazuje na místo v paměti. Při nesprávné alokaci může ukazovat na místa v paměti, kde se nachází systém a důležité soubory, nebo může být **NULL**. Pokud se snažíme dereferencovat nulový pointer, program spadne (SEGFAULT).

**int** \*p;

**void** \*p;

### Datový typ bez hodnoty

Nemá definovaný rozsah.

**void**

## Definování nových datových typů pomocí typedef

**Struktur**

**typedef** <neco> <*nejaky nazev*>  
**typedef** **long long** *ll* // priklad uziti typedef  
**typedef int** (\*cmp)(int, int) //typedef pointeru na funkci <https://www.geeksforgeeks.org/how-to-create-typedef-for-function-pointer-in-c/> okay mas pravdu

1)

**struct** ExistujiciStruktura {...}  
**typedef** **struct** ExistujiciStruktura *MojeStruktura*

2)

**typedef** **struct** ExistujiciStruktura {...} *ExistujiciStruktura*

**typedef** **struct** {...} *NepojmenovanaStruktura* // ale pojmenovany typ

ExistujiciStruktura abc1= …  
**struct** ExistujiciStruktura abc2 = …

**typedef struct** [jméno struktury] {

**typ** jméno\_položky;

**typ** jméno\_položky;

...

} [Název typu];

Jméno struktury je nepovinné (vznikne anonymní struktura).

Za definicí struktury mohou být hned definované datové typy.

**Union**

Syntaxe typu union je stejná jako syntaxe struktury, až na to, že místo klíčového slova **struct** se použije klíčové slovo **union**. Význam jednotlivých položek je stejný. Jeho použití s konstrukcí typedef také.

**typedef union** [jméno unionu] {

**typ** jméno\_položky

**typ** jméno\_položky

...

} [jmeno typu];

Zatímco struktura si vytvoří paměťové místo pro všechny položky, typ union zabírá v paměti jen tolik místa, kolik je jeho největší položka. Z toho také vyplývá, že lze používat v jeden okamžik jen jednu položku. Šetří paměť.

# 

# 5. Proměnné v jazyce C

* Charakteristika a deklarace lokálních a globálních proměnných
* Typy, charakteristika a význam paměťových tříd v jazyce C

## Charakteristika a deklarace lokálních a globálních proměnných

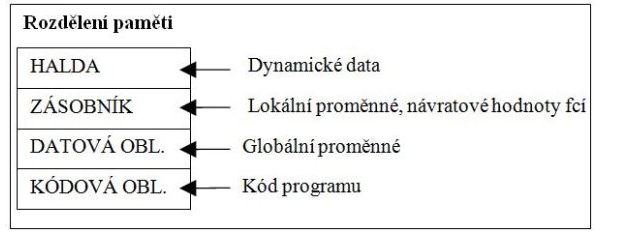
Proměnné jsou pojmenovaná místa v paměti, do kterých ukládáme nějaké hodnoty.

Každou proměnnou je třeba deklarovat, tj. oznámit překladači, jak se bude jmenovat a jaké hodnoty do ní budeme ukládat (jakého bude typu).

Deklarace nepřiděluje žádnou paměť, pouze udává její jméno a datový typ.

Inicializace při deklaraci znamená, že jsme proměnné přidělili počáteční hodnotu.

V neinicializované proměnné jsou náhodná data

Alokace paměti je vymezení místa v paměti pro proměnnou. Každé proměnné musí být během své existence přidělen paměťový prostor, který je dán datovým typem. Jméno proměnné je vlastně symbolická adresa tohoto prostoru.

**Globální proměnné**

* jsou dostupné ve všech funkcí programu od místa deklarace až do konce
* definují se mimo funkce (i mimo funkci main)
* pokud se globální proměnná jmenuje stejně jako lokální, lokální ji přebije
* paměť se alokuje na začátku programu a uvolňuje se až na konci programu
* jsou uloženy v datové oblasti
* automaticky mají hodnotu 0

**Lokální proměnné**

* definují se uvnitř bloku nebo funkce za { a platí jen v ní
* paměť se alokuje a uvolňuje v průběhu programu (v zásobníku)
* nenulují se automaticky

## Typy, charakteristika a význam paměťových tříd v jazyce C

Jak a kde jsou proměnné deklarovány, určuje jejich chování a místo v paměti, kde budou alokovány. Paměťová třída určuje, ve kterých částech bude proměnná alokována a kdy a kde bude použitelná.

**Paměťová třída auto**

* Je-li tedy proměnná definována uvnitř funkce bez určení paměťové třídy, je tato třída nastavena na auto
* Proměnné této třídy jsou uloženy ve stacku (zásobníku), a můžeme se na ně odvolávat od místa jejich definice až do konce funkce v rámci níž jsou definovány.
* Existují tedy pouze v rámci provádění kódu definiční funkce.
* Tyto proměnné při svém vzniku mají náhodnou hodnotu
* Pokud je ovšem v rámci definice explicitně neinicializujeme nějakou konkrétní hodnotou např. int i=1;

**Paměťová třída extern**

* Třída extern je implicitní paměťovou třídou pro globální proměnné.
* Tyto proměnné jsou uloženy v datové oblasti.
* Klíčové slovo extern použijeme v případě, kdy potřebujeme, aby jedna proměnná byla viditelná ve více souborech.
* V jednom souboru je třeba takovou globální proměnnou definovat (bez extern). V dalších souborech, v nichž má být viditelná ji pouze deklarujeme právě pomocí klíčového slova extern (většinou uvnitř hlavičkového souboru).

**Paměťová třída static**

* Proměnné této třídy jsou stejně jako proměnné globální uloženy v datové oblasti paměti, a tedy existují po celou dobu provádění programu.
* Tuto třídu využívají především lokální proměnné, které si ponechávají svoji hodnotu mezi jednotlivými voláními funkce. Příklad použití **strtok**
* Lokální proměnná třídy static tedy nezaniká po ukončení funkce, ale existuje dále a drží si svou aktuální hodnotu, kterou získala při posledním volání funkce. Jakožto lokální proměnná je ovšem viditelná pouze v rámci funkce. Paměťová třída static způsobí, že globální proměnná nelze znovu deklarovat.

**void** f(**void**){

**static int** pocitadlo = 0;

počitadlo++;

printf("Funkce byla volana %d krat.", pocitadlo);

}

Inicializace statické proměnné pocitadlo proběhne pouze při prvním volání funkce.

**Paměťová třída register:**

Tato třída vyvolá doporučení pro překladač, aby se pokusil proměnou uložit do registru procesoru, nikoli do paměti, neboť ji budeme používat velmi často. Překladač ovšem nemusí této žádosti vyhovět. Register je v moderních překladačích často ignorován (např. GCC ignoruje register pokud není flag -O0).

Výhoda takto umístěných proměnných tkví v mnohem rychlejším přístupu k nim ve srovnání s proměnnými uloženými v paměti což poněkud urychlí program. Specifikaci register však lze použít pouze pro lokální proměnné včetně formálních parametrů funkcí. Do registrů však lze bohužel současně uložit jen velmi malý počet proměnných.

Nelze získat adresu proměnné (&) s paměťovou třídou register.

Tuto třídu lze použít např. pro řídící proměnnou cyklu viz následující příklad:

**void** nasobilka(**register int** k){

**register int** i;

for(i = 0; i <= 10; i++) {

printf("%d krat %d = %d\n", i, k, i\*k );

}

}

# 6. Projekty, zdrojové a hlavičkové soubory v jazyce C

* Stavba vlastního hlavičkového souboru
* Projekty v programovacím jazyce C
* Spojování a překlad programu

Při nejčastějším způsobu práce odpovídá každému \*.c souboru, který poskytuje nějaké funkce a globální proměnné, jeden \*.h soubor. Název se zpravidla liší pouze příponou. V tomto \*.h souboru jsou hlavičky všech funkcí určených k použití z ostatních \*.c souborů, dále extern deklarace globálních proměnných a makra preprocesoru, která nějak souvisí s obsahem příslušného \*.c souboru. Hlavičkový \*.h soubor obsahuje i definice uživatelských typů. Tento soubor se pak pomocí příkazu #include vkládá na začátek všech \*.c nebo i jiných \*.h souborů, které jej potřebují.







## Stavba vlastního hlavičkového souboru

Vše musí být uvnitř podmíněného překladu, který zamezuje vícenásobnému vložení obsahu hlavičkového souboru.

#if konstantní\_vyraz\_l

usek\_programu\_l

#elif konstantní\_výraz\_2

úsek\_programu\_2

#else

usek\_programu\_3

#endif

Preprocesor vyhodnotí konstantní\_výraz\_1. Je-li jeho hodnota true (nenulová), přeloží úsek\_programu\_1. Úseky programu za direktivami #elif a #else ze zdrojového textu vypustí a přejde na zdrojový text za direktivu #endif.

Je-li hodnota konstantního\_výrazu\_1 rovna false, resp. 0, odstraní preprocesor úsek\_programu\_1 a vyhodnotí konstantní\_výraz\_2 za #elif. Je-li jeho hodnota nenulová, přeloží úsek\_programu\_2, odstraní úsek\_programu\_3 pak přejde za direktivu #endif.

Je-li i hodnota konstantního\_výrazu\_2 rovna false, odstraní se ze zdrojového textu také úsek\_programu\_2 přeloží úsek\_programu\_3.

## Projekty v programovacím jazyce C

Projekt je celek složený z několika modulů obsahujících zdrojový kód. Zavádí se, protože nám zjednoduší práci tím, že program rozdělíme do více modulů a tím ho zpřehledníme. Zároveň jsou také výhodné, protože provedeme-li změnu jen v jednom modulu, nemusíme kompilovat celý program, jak bychom museli udělat při změně jediného zdrojového souboru, ale stačí zkompilovat pouze daný modul.

Tento fakt se nazývá **oddělený překlad**.

## Rozdíl mezi vkládáním souborů a odděleným překladem

**Vkládání souborů**Obsah obrázku text, snímek obrazovky, Písmo, číslo

Popis byl vytvořen automaticky

* všechny soubory se vloží do hlavního souboru a překládají se společně
* musí se zbytečně překládat i soubory, které nebyly
* změněny
* #include “s\*.c”

**Oddělený překlad**Obsah obrázku text, snímek obrazovky, řada/pruh, Obdélník

Popis byl vytvořen automaticky

* #include “s1.h”
* výhodné při rozsáhlejších programech
* po překladu pak vzniklé soubory (moduly) jsou linkerem spojeny do jednoho programu
* při dalších překladech jsou již kompilovány pouze ty soubory, ve kterých skutečně došlo ke změně, což celý proces značně urychluje

## Spojování a překlad programu

Oddělený překlad znamená, že se každý soubor přeloží zvlášť, vznikne více .obj souborů a ty se spojí do jednoho programu pomocí linkeru.

Výhodou je, že stačí překládat pouze soubor, který se změnil.

# 7. Struktura, union, výčet v jazyce C

* Deklarace a charakteristika datového typu struktura
* Deklarace a charakteristika datového typu union
* Deklarace a charakteristika výčtového datového typu

## Deklarace a charakteristika datového typu struktura

Struktura je heterogenní datový typ s libovolným počtem prvků různého datového typu. Umožňuje spojit několik datových typů do jednoho a pracovat s nimi jako s celkem.

**Deklarace struktury**

**typedef struct** {

**typ** proměnná1;

**typ** proměnná2;

**typ** proměnná3;

...

...

}název;

**Deklarace struktury s pojmenováním**

**typedef struct** {

**int** vek, hmotnost;

**float** vyska;

} Clovek;

Clovek je typ struktury a pepa a franta jsou samotnými proměnnými typu Clovek.

Potom lze k jednotlivým prvkům struktury přistupovat přes tzv. tečkový operátor:

Clovek pepa, franta;

pepa.vek = 34;

pepa.vyska = 1.83;

franta.hmotnost = 87;

franta.vek = 46;

**Inicializace struktury**

**Clovek** franta1 = {"Adam", "Kiosky", 180, 60, 5};

**Pointer na strukturu**

**typedef struct** {

**char** jmeno[30];

**int** rocnik;

}Student;

Student s, \*ps;

**Přístup k dynamické struktuře**

s.rocnik = 3; //pomocí jména struktury

(\*ps).rocnik = 4; //pomocí pointeru

ps->rocnik = 4; //pomocí pointeru

## Deklarace a charakteristika datového typu union

Union se používají ve velmi specifických případech v nízko úrovňovém kódu a běžně se s nimi nesetkáme. Syntaxí se velmi podobají strukturám, jen místo klíčového slova struct se použije union. jediným rozdílem od struktur je to, že v daný okamžik existuje v paměti pouze jeden člen. Union má tak velikost odpovídající prostorově největšímu typu, mění se pouze interpretace dat.

**typedef union** {

**int** i;

**float** f;

**char** str[20];

} data;

## Deklarace a charakteristika datového typu enum

Slouží k definici seznamu symbolických konstant, které jsou většinou vzájemně závislé.

**Deklarace**

**typedef enum** {MODRA, CERVENA, ZELENA, ZLUTA} **barvy**;

barvy barvaHlavni = MODRA;

Čísluje se implicitně od nuly, ale lze i explicitně inicializovat

**typedef enum** {MODRA = 5, CERVENA = 8, ZELENA, ZLUTA = 15} **barvy**;

Zelená bude zde mít hodnotu 9.

# 8. Operátory (C, Java)

* Rozdělení operátorů
* Ternární operátor
* Priorita operátorů

**Operand** – hodnota, se kterou počítáme

**Operátor** – určuje operaci, kterou hodláme provést s operandem nebo operandy

## Rozdělení operátorů

Operátory rozdělujeme podle počtu operandů (arity) na operátory unární, binární a ternární. Každý operátor má svou prioritu a asociativitu. Priorita určuje, že například násobení se vyhodnotí dříve, než třeba sčítání a asociativita říká, vyhodnocuje-li se vyraz zleva doprava, nebo naopak.

Operátory rovněž dělíme podle pozice jejich zápisu vzhledem k operandům. Takto rozlišujeme operátory prefixové, infixové a postfixové. Operátory v jednotlivých případech zapisujeme před operandy, mezi operandy nebo za operandy.

A do třetice všeho dobrého se dělí i podle jejich funkčnosti na:

***PROPAL***

1. aritmetické
2. logické
3. relační
4. přiřazovací
5. operátory bitové
6. přístupové

**Unární operátory**

| +, - | aritmetické plus a minus |
| --- | --- |
| & | reference (získaní adresy objektu) |
| \* | dereference (získaní objektu dle adresy) |
| ! | logická negace |
| ~ | bitová negace |
| ++, -- | Inkrementace a dekrementace hodnoty (prefixový i postfixový zápis) |
| (typ) | přetypovaní na typ uvedeny v závorkách |
| sizeof | operátor pro získaní délky objektu nebo typu |

**Unární plus a minus**

Unární plus a mínus určuje znaménko čísla. Například ve výrazu 5 + (-4) je plus binární operátor sčítání (má dva operandy) a mínus je unární operátor (vztahuje se jen ke čtyřce).

**Operátor reference a dereference**

* operátor \* vrací hodnotu objektu, na který ukazatel ukazuje
* operátor & vrací adresu proměnné, před kterou stojí

**Bitová negace**

* obrací jednotlivé bity ve výrazu
* např.: ~00000101 = 11111010

**Inkrementace (a dekrementace)**

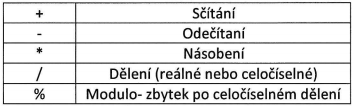
* **prefixový zápis (++a) –** hodnota je nejprve zvětšena o jedničku a pak je tato nová hodnota vrácena jako hodnota výrazu
* **postfixové** **zápis (a++)** **–** nejprve je vrácená původní hodnota proměnné a až pak je zvětšena o jedna

**Logická negace**

* výsledkem logické negace je true nebo false, což jazyk C vyhodnocuje jako 1 nebo 0
* například !3 = 0

**Binární operátory**

Pracují se dvěma operandy, operátor se píše mezi ně.

**Aritmetické**

Operátor /:

Je specifický, o tom, zda bude celočíselný nebo reálny rozhoduje

datový typ. Je-li alespoň jeden z operandů datového typu float, double

nebo long double, bude dělení reálné. Budou – li oba operandy

celočíselné, dělení bude taktéž celočíselné.

Modulo - %:

Operátor pro zjištění zbytku po celočíselném dělení.

Zbytek po celočíselném dělení je jeho návratová hodnota.

**Logické**

| **| |** | logická disjunkce OR |
| --- | --- |
| && | logická konjunkce AND |
| **!** | logická negace NOT |

**Relační**

| **==** | rovnost |
| --- | --- |
| **!=** | nerovnost |
| **<** | menší než |
| **>** | větší než |
| **>=** | větší nebo rovno |
| **<=** | menší nebo rovno |

**Přiřazovací**

| **x=y** | x=y |
| --- | --- |
| **x+=y** | x=x+y |
| **x-=y** | x=x-y |
| **x\*=y** | x=x\*y |
| **x/=y** | x=x/y |
| **x%=y** | x=x%y |

**Operátory bitového posuvu**

| **<<** | bitový posun doleva, posun všech bitů o jednu pozici doleva |
| --- | --- |
| **>>** | bitový posun doprava, posun všech bitů o jednu pozici doprava |

**Přístupové**

| **.** | přímý přístup ke členu struktury |
| --- | --- |
| **->** | nepřímý přístup ke členu struktury |

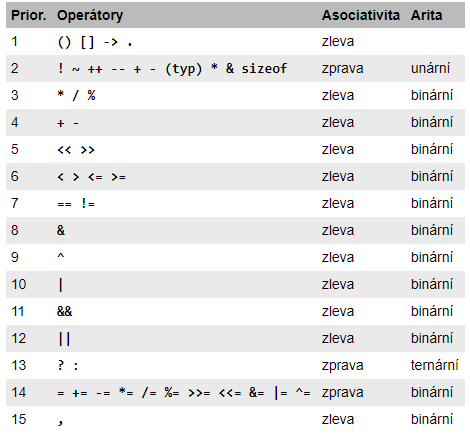
## Ternární operátor

| (cond) ? x : y | podmíněny operátor |
| --- | --- |

Podmíněný operátor ? : je ternárním operátorem (má 3 operandy). Prvním operandem je výraz, který se vyhodnotí jako logický výraz (TRUE nebo FALSE). Pokud se vyhodnotí jako TRUE, výsledkem bude druhý operand (výraz mezi ? a :), jinak třetí operand (výraz za :).

## Priorita a pořadí vyhodnocování

Řada operátorů má stejnou prioritu vyhodnocování. Pokud se na jedné úrovni výrazu vyskytuje více operátorů téže priority, jsou příslušné operace vyhodnocovány podle jejich asociativity zleva doprava nebo zprava doleva. Z důvodu vyhnutí se chybám v prioritě vyhodnocování je lepší vždy používat závorky.

Prioritu vyhodnocování udává tabulka preferencí:

**Rozdíly C k Java Operátorům**

Stejně jako u datových typů je i většina operátorů u Javy převzatých z jazyka C. K rozdílům dochází zejména u operátorů spojených s OOP, kterým jazyk C nedisponuje.

Například jedním z relačních operátorů je operátor instanceof, který zjišťuje zda-li je objekt instancí dané třídy.

if(retezec **instanceof** String)

Nebo operátor new – vytvoření nové instance třídy

**Clovek** Pepa = **new** **Clovek**();

# 9. Funkce (C)

* Stavba a vlastnosti funkcí
* Parametry funkcí, funkce bez parametru
* Rekurzivní funkce

Každý program má minimálně jednu funkci (funkce main()).

Funkce se dělí na:

* standardní funkce
* uživatelské funkce
* podpůrné a nadstavbové funkce

**Standardní funkce**

Jsou definovány normou jazyka výrobce a překladač je dodává jako součást programového balíku tvořícího překladač a jeho podpůrné programy a soubory. Tyto standardní funkce zpravidla dostáváme jako součást standardních knihoven (proto se jim někdy říká knihovní funkce) a jejich deklarace je popsaná v hlavičkových souborech. Nemáme k dispozici jejich zdrojový kód. Ostatně nás ani moc nezajímá. Tyto funkce se přeci mají chovat tak, jak definuje norma.

**Uživatelské funkce**

Jsou ty funkce, které jsme napsali a máme jejich zdrojové texty. Pokud jsme profesionály, vyplatí se nám naše funkce precizně dokumentovat a archivovat. Když už jsme je jednou napsali a odladili, můžeme je příště s důvěrou již jen používat. Může účelně sdružovat více uživatelských funkci do jednoho, připadne několika souboru, případně z nich vytvořit knihovnu, abychom je při každém použiti nemuseli znovu překládat. Hlavičkové soubory (prototypy funkci, uživatelské typy, definice maker, ...) jsou opět nezbytnosti.

**Podpůrné a nadstavbové funkce**

Nejen funkce od tzv. třetích výrobců (podpora pro spolupráci s databázemi, volaní modemu, faxu, uživatelské rozhraní, ...) ale i rozšířeni překladače o funkce nedefinované normou jazyka, funkce implementačně závislé a podobné. Nakonec sem patří i naše funkce, které používáme jako podpůrné. Jako poslední bod si můžeme představit týmovou prací, kdy používáme funkce kolegů. O jejich zdrojový text se zajímáme až v případě, kdy se funkce nechovají tak, jak mají.

## Stavba a vlastnosti funkce

**typ** jmeno([seznam parametrů]);

* **typ** představuje typ návratové hodnoty funkce, jestli datový typ neuvedeme, bude implicitně vracet int
* **jmeno** je identifikátor, který funkci dáváme
* **()** je povinna dvojice omezující deklaraci argumentu (v tomto případě prázdnou)
* **seznam parametrů** je nepovinny – funkce nemusí mít argumenty, může mít jeden nebo více argumentů. Označujeme je jako **formální parametry.** U každého parametru musí být určen datový typ a jméno.

**Příkaz return**

* způsobí okamžité ukončení funkce
* posílá návratovou hodnotu na místo, kde byla funkce volána

**Deklarace funkce**

* uvedení hlavičky funkce, ukončené středníkem
* říkáme tím překladači, že dále v programu bude s funkcí s tím názvem pracovat

**Definice funkce**

* zápis celé funkce, včetně hlavičky i těla funkce

**Volání funkce**

* parametry, které uvádíme přivolání funkce nazýváme **skutečné parametry**
* parametry v hlavičce funkce jsou **formální** **parametry**

## Rekurzivní funkce

Rozlišujeme 4 typy rekurzí:

1. **přímá rekurze** – nastává, pokud funkce volá přímo sama sebe (rekurzivní funkce)
2. **nepřímá rekurze** – je situace, kdy vzájemné volání podprogramů vytvoří kruh (např. v příkazové části A je volána funkce B, v příkazové části B je volána funkce C a v příkazové části C je volána funkce A)
3. **lineární** **rekurze** – funkce volá sama sebe pouze jednou
4. **stromová** **rekurze** – funkce se v rámci jednoho úkonu vyvolá vícekrát

Vždycky se dá zapsat algoritmus bez použití rekurze. Při rekurzivních funkcí musí být jasně definována podmínka pro zastavení. Nebývají šetřičem místa. Mohou volat dokola samy sebe, a tak alokují místa více. V případě nevhodného použití by program havaroval.

# 10. Činnost preprocesoru (C)

* Charakteristika činnosti preprocesoru
* Tvorba maker, srovnání makra a funkce
* Podmíněný překlad
* Hlavičkové soubory

## Charakteristika činnosti preprocesoru

Preprocesor odstraní ze zdrojového kódu komentáře a rozvine makra, které jsou ve zdrojovém kódu. Když zapíšeme do zdrojového kódu například #include <stdio.h>, tak preprocesor vloží na místo tohoto řádku obsah souboru stdio.h.

* Vkládá hlavičkové soubory
* Rozvíjí makra
* Provádí podmíněný překlad
* Maže komentáře

Direktiva preprocesoru musí být první na novém řádku, mohou před ní být jen „bílé znaky“ -mezera, tabulátor. Pokud chceme pokračovat na novém řádku, pak můžeme na konci řádku napsat zpětné lomítko a pokračovat na dalším řádku.

Preprocesor přijímá následující direktiva:

* #define
* #elif
* #else
* #endif
* #error
* #if
* #ifdef
* #ifndef
* #include
* #line
* #pragma
* #undef

## Makra

Dělí se na makra bez parametru a makra s parametrem.

**Makra bez parametru**

= symbolické konstanty

#define název hodnota

Preprocesor projde celý kód a nahradí všechny výskyty názvu konstanty její hodnotou. Této činnosti se říká rozvoj (expanze) makra. Název píšeme pro přehlednost velkými písmeny.

**Makra s parametrem**

Preprocesor nahradí argumenty za parametry. Výhodou i nevýhodou je, že nepracuje s typy. Výhodou proto, že pokud bychom chtěli definovat podobnou funkci, museli bychom napsat tolik jejích verzí, kolik by bylo navzájem neslučitelných variant datových typů argumentů. Nevýhodou je netypovost makra tehdy, uvedeme-li omylem třeba jako argumenty řetězce (pak by se porovnávaly adresy jejich prvních znaků) nebo dva argumenty neporovnatelných typů (např. struktura a číslo). Takové chyby pak (někdy) odhalí až překladač.

#define max(a, b) ((a > b)? a : b)

Při definici makra max nás možná překvapí zdánlivě nadbytečné závorky. Musíme jen připomenout, že makra nejsou příkazy jazyka C. Jejich rozvoj probíhá na textové úrovni. Preprocesor tedy nemůže v závislosti na kontextu jednou nadbytečné závorky vypustit, jindy chybějící přidat. Proto raději sami nadbytečné závorky nevypouštíme.

## Rozdíl mezi funkcemi a makry

* funkce potřebuje čas a paměť na alokaci parametrů
* makro zaměňuje řetězec za řetězec, nepotřebuje paměť ani čas k alokaci argumentů
* makra neumožňují typovou kontrolu ani použití rekurze

**Výhody maker**

* rychlost
* šetří paměť

**Nevýhody maker**

* vrací pouze text
* nelze ladit
* nelze provádět rekurzi

**Výhody funkcí**

* může vrátit jakýkoliv datový typ
* lze ladit
* může využívat rekurzi

**Nevýhody funkcí**

* nižší rychlost
* zabírá místo v paměti

## Podmíněný překlad

Preprocesor může během své činnosti vyhodnocovat, je-li nějaké makro definováno či nikoliv. Při použiti klíčového slova preprocesoru defined pak může spojovat taková vyhodnocení do rozsáhlejších logických výrazu. Argument defined nemusí byt uzavřen do závorek. Může se však vyskytnout jen za #if nebo #elif.

#if defined LIMIT && defined OSTRA && LIMIT == 10

V závislosti na splnění či nesplnění podmínky můžeme určit, bude-li ohraničený usek programu dále zpracován, nebo bude-li odfiltrován, a tak nebude tedy přeložen. Této možnosti použiti preprocesoru říkáme podmíněny překlad.

Vždy musí být jasno, kde podmíněná část zdrojového textu začíná a kde končí. Proto nesmíme zapomínat na #endif.

**Podmínění překlad řízený hodnotou konstantního výrazu**

V konstantním výrazu nemůžeme použít proměnné, protože v době práce preprocesoru nejsou ještě alokovány. Podmínění překlad určuje, která část bude nebo nebude přeložena podle hodnoty makra nebo výrazu. Zapisuje se pomocí #if, #elif, #else a #endif.

#if konstanta

část\_1

#else

část\_2

#endif

Jeli hodnota konstanty rovna 0, přeloží se část\_2, pokud je hodnota různá od nuly, přeloží se část\_1.

**Podmíněný překlad řízený existencí makra**

Podmíněný překlad může být také závislý na tom, zda je definováno makro nebo ne a nezáleží přitom na jeho hodnotě. Zapisuje se pomocí direktiv #ifdef, #ifndef, #else a #endif.

Direktiva ifndef, provádí opak výše direktivy ifdef.

#ifdef jméno\_makra

část\_1

#else

část\_2

#endif

Pokud je makro jméno\_makra definováno, překládá se část\_1, jinak se překládá část\_2. Opět platí, že části #else a část\_2 lze vynechat.

## Hlavičkové soubory

Vkládají se pomocí direktiv #include. Do zdrojového kódu můžeme vkládat hlavičkové soubory ze standartního adresáře nebo uživatelské hlavičkové soubory. Standartní hlavičkové soubory (např. stdio.h, time.h, math.h, . . . ) najdeme v místě instalace compileru nebo celého vývojového prostředí ve složce include.

V hlavičkovém souboru by se nikdy neměl vyskytovat kód, ani definice proměnných či konstantních proměnných. Standartní hlavičkové soubory vkládáme pomocí #include <nazev souboru> a uživatelské pomocí #include "nazev souboru".

V hlavičkových souborech se deklarují:

* Makra
* Hlavičky funkcí
* Datové typy
* Globální proměnné

Na začátku každého hlavičkového souboru by měl být použit podmíněný překlad, který zabraňuje vícenásobnému vložení hlavičkového souboru do projektu.

# 11. Pointery (C)

* Deklarace, charakteristika a využití pointerů
* Dynamické přidělování paměti v C
* Pointerová aritmetika
* Pointery a funkce

## Deklarace, charakteristika a využití pointerů

Pointery nedokáží uchovat žádnou hodnotu, ale dokáží v sobě uchovat adresu v paměti. Díky tomu mohou ukazovat na libovolný typ dat, ale pouze na ten, na který byly nadefinovány.

Jsou pouze tak velké, aby se do nich vešla nějaká adresa paměti, dá se s nimi velice rychle pracovat a zabírají málo času CPU pokud se používají ve spojení s velkými datovými strukturami. Při použití s primitivními datovými typy dojde spíše ke zpomalení z jednoduchého důvodu. Pokud přistupujeme k proměnné přímo, stačí přečíst její hodnotu. Při přístupu přes ukazatel je třeba nejdřív zjistit hodnotu ukazatele (tedy zjistit adresu kam ukazuje), přesunout se na tu adresu a tam teprve přečíst hodnotu proměnné, na kterou ukazuje. Je zde tedy jistý mezikrok.

**int** a, \*p;

p = &a;

\*p = 20;

printf("Hodnota v a je: %d\n", a);

printf("Hodnota na kterou ukazuje p je: %d", \*p);

Na prvním řádku se vytváří 2 proměnné typu int, jedna je fyzická (a) a druhá (\*p) je ukazatelová. Zatím ani v jedné není nic uloženo. Následující řádek přiřazuje do (p) adresu proměnné (a). Můžeme to přečíst asi takto "do ukazatelové proměnné (p) typu int ulož adresu proměnné (a)". Pak tady máme přiřazení hodnoty 20 do proměnné na kterou ukazuje (p). Dá se přečíst "hodnotu 20 zapiš do proměnné, na kterou ukazuje (p)". No a nakonec tady jsou dva výpisy na obrazovku pomocí funkce printf(). V prvním je použita jako argument přímo proměnná (a), ve druhém je v argumentu (\*p), což znamená "hodnota na adrese, kam ukazuje p".

* nulový pointer NULL - pro označení, že pointer nikam neukazuje
* adresu pointeru v příkazu printf vypíšeme pomocí formátovacího znaku %p
* operátor reference & slouží k získání adresy proměnné
* operátor dereference \* slouží k přístupu k obsahu, který je uložený na adrese, kde ukazuje pointer (zapsání nebo přečtení hodnoty)

## Dynamické přidělování paměti v jazyce C

K dynamické alokaci postačí hlavičkový soubor stdlib.h, poté můžeme použít funkci malloc pro alokaci paměti v haldě.

**int** \*p;

p = (**int**\*)malloc(sizeof(**int**));

Jediným parametrem funkce malloc je počet bitů, které chceme alokovat.

Defaultně je funkce malloc nastavená na void. Jestliže v haldě není dostatek místa vrací hodnotu NULL, proto je vhodné při každé alokaci kontrolovat, zda se dynamickou proměnnou podařilo alokovat.

if((p = (**int**\*)malloc(sizeof(**int**))) == NULL) {

printf("malo místa");

**return** -14;

}

**Ukázka místa s paměti**

**int** \*p;

p = (**int**\*)malloc(sizeof(**int**));

\*p = 10;

free(p);

p = NULL;

## Pointerová aritmetika

Pro pointery můžeme použít také operátory ++, --, +, -, <, == a další. Tyto operace mají význam, pokud ukazují na stejný blok paměti a mají stejný datový typ. Obsah obrázku text, diagram, snímek obrazovky, řada/pruh

Popis byl vytvořen automaticky

**Součet pointeru a celého čísla**

Přičteme-li k pointeru p ukazujícího na adresu proměnné určitého typu kladné číslo m získáme pointer ukazující na adresu v paměti jež je o m\*sizeof(typ proměnné) bytů výše a naopak, při odečtení kladného čísla se posuneme na adresu jenž leží v paměti o příslušný počet bytů níže. Rozdíl dvou pointerů má smysl tehdy, pokud oba pointery ukazují na stejný celek (např. pole). Obsah obrázku řada/pruh, Obdélník, snímek obrazovky, diagram

Popis byl vytvořen automaticky

**Subscript operátor: a[i]**

Operátor **a[i]** je ekvivalentní se zápisem v pointerové aritmetice **\*(a + i).**

**int** \*a = (**int**\*)malloc(**sizeof**(**int**) \* 10);  
a[8] == \*(a + 8) // True  
// v pameti posunes “a” o 8 \* (sizeof (int))

**Rozdíl pointerů**

*p2 - p1= d* <=> *p2 = p1+d*

Počet prvků ležících tedy mezi prvky na něž ukazují pointery p1, p2 je roven *d-1*. Rozdíl pointerů vrací tedy počet prvků mezi p1 a p2. Obsah obrázku řada/pruh, Obdélník, snímek obrazovky, diagram

Popis byl vytvořen automaticky

**Porovnání pointerů**

Porovnáním dvou pointerů daného typu lze zjistit, který z nich ukazuje na vyšší, nižší nebo stejnou adresu v paměti.

## Pointery a funkce

Využíváme tehdy, pokud chceme, aby funkce vracela více výsledků, jelikož příkaz return() může vracet jen jeden výsledek, a proto další výsledek vracíme přes pointer, který je také jako formální parametr funkce. U volání funkce použijeme jeho adresu. Tomu se říká výstupní parametr.

**void** minimum(**int** pole[][5], **int** \*radek, **int** \*sloupec);

**int** main()

{

**int** pole[5][5], i, j, radek, sloupec;

minimum(pole, &radek, &sloupec);

**return** 0;

}

**void** minimum(**int** pole[][5], **int** \*radek, **int** \*sloupec)

{

**int** i, j, pomoc = pole[0][0], r = 0,s = 0;

for(i = 0; i < 5; i++) {

for(j = 0; j < 5; j++) {

if(pomoc > pole[i][j]) {

r = i;

s = j;

pomoc = pole[i][j];

}

}

}

\*radek = r;

\*sloupec = s;

}

# 12. Jednorozměrné pole (C)

* Deklarace a charakteristika statického pole
* Deklarace a charakteristika dynamického pole
* Pole jako parametr funkce
* Třídící algoritmus

## Deklarace a charakteristika statického pole

[**char**](https://www.sallyx.org/sally/search/?q=c+char) znaky[4]; //pole pro 4 znaky  
[**int**](https://www.sallyx.org/sally/search/?q=c+int) cisla[5];  //pole pro 5 čísel

Proměnné znaky a cisla jsou proměnné typu array, které se chovají jako konstantní ukazatele. Ukazují na první prvek z pole. Konstantní ukazatel na pole obsahuje adresu paměti, kde začíná pole. Tuto hodnotu změnit nelze, můžete měnit jen hodnoty v poli. Jednotlivé položky pole se adresují od nuly pomocí hranatých závorek.

K jednotlivým prvkům pole lze přes index přistupovat.

znaky[3], cisla[0]

cisla[i] odpovídá \*(cisla + i)

## Deklarace a charakteristika dynamicky alokovaného pole

Dynamicky alokované pole vytvoříme pomocí pointeru a následné alokace paměti.

**int** i, \*pole;

if((pole = (**int**\*)malloc(sizeof(**int**)\*10)) == NULL) {

printf(“malo mista“);

}

for(i = 0; i < 10; i++) {

pole[i] = i;

}

free(pole);

pole = NULL;

Z příkladu je vidět, že není rozdíl, jestli pracujeme s dynamickým nebo statickým polem. V obou případech používáme pro přístup hranaté závorky a index nebo pointerovou aritmetiku.

Výhodou dynamicky alokovaného pole je, že můžeme určit jeho velikost, tedy počet prvků v poli, při alokaci. Na rozdíl od statického pole nelze zjistit počet prvků dynamického pole.

Pro zjištění velikosti počtu prvků statického pole stačí napsat:

**typ** pole[n];

**int** velikost = sizeof(pole);

**int** pocetPrvku = velikost/sizeof(**typ**);

U dynamického pole však výše uvedený postup nelze použít.

**typ** \*pole;

pole = (**typ**\*)malloc(sizeof(**typ**)\*pocet);

velikost = sizeof(pole);

Tento zápis zjistí pouze velikost pointeru, tedy velikost pole ne.

## Pole jako parametr funkce

Je-li pole parametrem funkce, pak se skutečný parametr předává pouze odkazem, tedy pomocí pointeru. Což má nesporné výhody. V paměti se nemusí vytvářet lokální kopie skutečných parametrů, což šetří paměť a prvky se nemusí opakovat, což šetří i čas.

**int** minimum(**int** pole[], **int** pocet)

{

**int** i, min = pole[0];

for(i = 0; i < pocet; i++) {

if(min > pole[i]) {

min = pole[i];

}

}

**return** min;

}

## Třídící algoritmy

Jedná se o uspořádání množiny dat podle klíčové položky (vzestupně nebo sestupně podle velikosti). Se seřazenými daty se pak všeobecně lépe pracuje.

**Vnitřní třídění**

* použijeme pouze tehdy, pokud lze množinu třízených dat umístit do vnitřní paměti počítače
* data jsou uložena v poli a algoritmy využívají možnost přímého přístupu k jednotlivým prvkům (třízení polí)

**Vnější třízení**

* použijeme pouze tehdy, kdy se třízená data nevejdou do vnitřní paměti počítače
* data jsou umístěna v souborech na vnějším paměťovém mediu
* je založeno na opakovaném čtení a vytváření souborů

**Algoritmy vnitřního třídění**

1. Třídění přímým výběrem (SelectSort)
2. Třídění přímým vkládáním (InsertSort)
3. Třídění přímou výměnou (BubbleSort)
4. Třídění rozdělováním (QuickSort)

**Třídění přímým výběrem (SelectSort)**

* Třízená posloupnost je rozdělena na setříděnou a nesetříděnou část
* Z nesetříděné části vybíráme minimální prvek, který je vložen na poslední místo setřízené části
* Velikost setříděné části se tak vždy zvýší o jeden prvek

**void** selectSort(**int**[] list, **int** poc) {

**int** min, i, j;

for(i = 0; i < poc i++) {

min = i;

//hledání minima

for(j = i + 1; j < poc; j++) {

if(list[min] > list[j]) {

min = j;

}

}

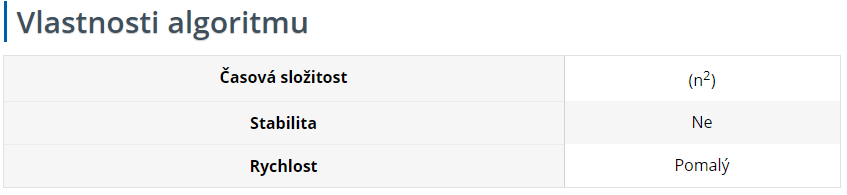
//prohození prvků

list[min] += list[i];

list[i] = list[min] – list[i];

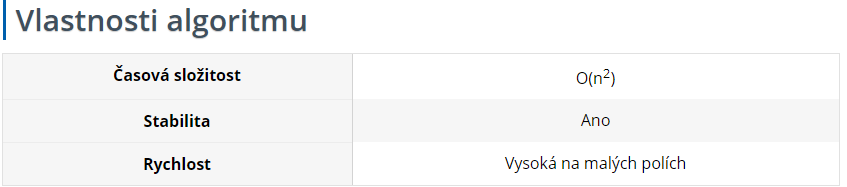
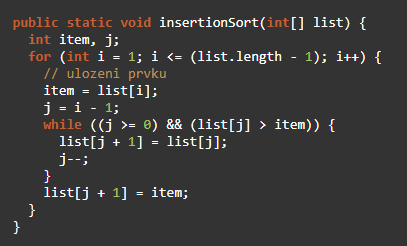
list[min] -= list[i];

}

}

**Třídění přímým vkládáním (InsertSort)**

* posloupnost tříděných složek je rozdělena na setříděnou a nesetříděnou část
* z nesetříděné části vybíráme prvky a zařazujeme je do setříděné části tak, aby tato část zůstala setříděná, tento postup se opakuje tak dlouho, dokud není nesetříděná část prázdná



**Třídění přímou výměnou (BubbleSort)**

* pole je rozděleno na setříděnou a nesetříděnou část
* opakovaně porovnáváme sousední prvky v nesetříděné části pole
* jestliže testovaná dvojice není spořádaná, pak vyměníme pozice testovaných prvků
* výsledkem je “probublání“ minimálního prvku nesetříděné části na její začátek
* velikost setříděné části se zvýší o jeden prvek

**void** BubbleSort(**int**\* pole, **int** pocetPrvku)

{

**int** i, j;

for(i = 0; i < pocetPrvku - 1; i++) {

for(j = 0; j < pocetPrvku -1; j++) {

if(pole[j + 1] < pole[j]) {

int temp = pole[j + 1];

pole[j + 1] = pole[j];

pole[j] = temp;

}

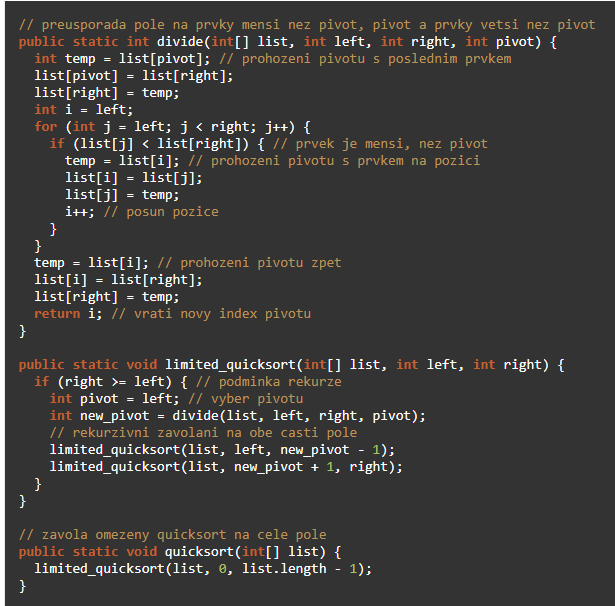
}

}

}



**Třídění rozdělováním (QuickSort)**

* metoda je založena na principu rozdělení pole na dvě části
* pro každý prvek x z jedné části pole platí x < y, kde y je libovolný prvek z druhé části
* Nejvýhodnější dělící hodnotou pole je medián
* Nalezení mediánu je však složité, a proto se volí jiný prvek – střed pole

Obsah obrázku text, snímek obrazovky, Písmo, řada/pruh

Popis byl vytvořen automaticky

# 13. Vícerozměrné pole (C)

* Deklarace a charakteristika statického vícerozměrného pole
* Deklarace a charakteristika dynamického vícerozměrného pole

## Deklarace a charakteristika statického VP

Toto pole je alokováno v paměti jako souvislý blok paměti.

**int** pole[2][4];

## Deklarace a charakteristika dynamického VP

Dynamické vícerozměrné pole můžeme deklarovat třemi způsoby:Obsah obrázku text, snímek obrazovky, diagram, řada/pruh

Popis byl vytvořen automaticky

**Pole pointerů**Obsah obrázku text, snímek obrazovky, diagram, Obdélník

Popis byl vytvořen automaticky

**Pointer na pole**Obsah obrázku text, snímek obrazovky, diagram, Písmo

Popis byl vytvořen automaticky

**Pointer na pointer**Obsah obrázku text, snímek obrazovky, diagram, Obdélník

Popis byl vytvořen automaticky

# 14. Znaky a řetězce (C)

* Deklarace a charakteristika znakové proměnné
* Charakteristika řetězce v jazyce C
* Základy práce s řetězci
* Funkce pro práci s řetězci
* Parametry funkce main()

## Deklarace a charakteristika znakové proměnné

Jedná se o proměnnou datového typu char, vnitřně se však jedná o celé číslo.

**char** a = 'a';

Při načítání znaků funkcí scanf() je důležité dát mezeru a poté až %c, jelikož by nám uživatel mohl do proměnné uložit prázdný enter.

scanf(“ %c“, &a);

Znakové proměnné využívají ASCII tabulku, ve které má každý znak danou hodnotu. Uloženou hodnotu ve znakové proměnné lze tedy přepsat třeba přičtením či odečtením čísla (např. když chceme převádět malá písmena na velká atd…)

printf(“Velké A: %c“, a – 32);

## Charakteristika řetězce v jazyce C

Řetězec je jednorozměrné pole, jehož prvky jsou proměnné datového typu char. Posledním prvkem pole vždy musí být ukončovací znak \0. Při udávání počtu prvků pole je tedy potřeba počítat s počtem znaků řetězce + 1.

**char** znak; //deklarace znakové proměnné

**char** znaky[pocetPrvku]; //deklarace řetězce

**char** a[5] = “Ahoj”;

**char** b[] = “Ahoj”;

**char** c[] = {‘A’ , ‘h’, ‘o’, ‘j’, ‘\0’};

‘A’… znak A, velikost 1 bajt

“A”… řetězec A, velikost 2 bajty (kvůli znakové nuly - to je druhý znak)

## Základy práce s řetězci

Je třeba si dávat pozor na kopírování řetězců, protože nelze napsat string = “Bum“ ale je třeba napsat string[0]= 'B';, string[1]= 'u';, string[2]= 'm';, string[3]= '\0';.

Aby k tomuto nemuselo docházet jsou v C připraveny funkce pro práci s řetězci.

## Funkce pro práci s řetězci

Načte 5 znaků, ostatní zadané znaky zůstanou v bufferu (mezipaměti)

scanf(“%5s”, retezec);

* bez &, protože pointer, který ukazuje na první prvek pole už obsahuje adresu
* znak \0 přidá compiler
* nekontroluje meze!
* nečte bílé znaky!

Načítá řetězec vč. mezer, až do zadání ENTER.

gets(retezec);

* nekontroluje meze, ale čte bílé znaky
* od určité doby už není součástí standardu

Funkce, která se defaultně používá při práci se soubory. Pokud ale jako třetí parametr použijeme místo názvu souboru “stdin”, lze ji použít při normální práci s řetězci.

fgets(str, MAX\_LIMIT, stdin);

* umožňuje nám kontrolovat meze!
* limit se počítá včetně znakové nuly

Vypsání řetězce.

printf(“%s”, retezec);

puts(retezec);

Převod řetězce na long.

atol(s);

Převod řetězce na int.

atoi(s);

Převod řetězce na double.

atof(s);

Následující funkce již potřebují knihovnu <string.h>:

Zkopíruje řetězec co do řetězce kam.

strcpy(kam, co);

Vrací počet znaků řetězce, bez znakové nuly.

pocetZnaku = strlen(retezec);

Lexikografické porovnání řetězců.

strcmp(retezec, “Ahoj”);

* vrací -1 (nebo obecně záporné číslo) pokud retezec je lexikograficky menší
* 1 (nebo obecně kladné číslo) pokud je větší než Ahoj
* porovnává řetězce na základě ASCII hodnoty prvního rozdílného znaku
* pokud žádný takový znak neexistuje, pole jsou si rovny a funkce vrací 0

Spojení dvou řetězců.

strcat(s1, s2);

Vyhledávání znaku v řetězci.

strchr(s, c);

* hledá, zda je v řetězci s obsažen znak c
* pokud ano, vrací ukazatel na začátek prvního výskytu
* pokud ne, vrací NULL

Vyhledávání řetězce v řetězci.

strstr(a, b);

* hledá, zda je v řetězci s1 obsažen podřetězec s2
* pokud ano, vrací ukazatel na začátek prvního výskytu
* pokud ne, vrací NULL

## Parametry funkce main()

**int** main(**int** argc, **char** \*argv[]) {...}

Int argc je proměnná ve které je uložen počet parametrů argv. Argv představuje pole řetězců s tím že jeho nultý prvek je vždy název souboru. Takže pokud programu předáme 2 parametry hodnota argc bude 3 protože se zde nachází nultý prvek – název programu.

argv[0] == "program.exe"

argv[1] == "pr1"

argv[2] == "pr2"

# 

# 15. Soubory (C)

* Textové a binární soubory
* Práce se soubory
* Souborové funkce
* Testování otevření a uzavření souboru

Jedná se o uspořádanou množinu dat uložených v paměťovém médiu mimo operační paměť. Pro práci s nimi stačí knihovna <stdio.h>.

## Textové a binární soubory

Základní rozdíl mezi textovými a binárními soubory je to, že se binární soubory ukládají stejně jako v paměti, takže se s nimi počítači lépe pracuje. Operační systém musí soubor při ukládání stejně převést na binární. V tomto jsou tedy binární soubory rychlejší, zapíšou se rovnou binárně. Zatímco textové se ukládají jako posloupnost znaků.

## Práce se soubory

**Textové**

**FILE** \*f = fopen("soubor.txt", "w");

Je ho možno otevřít ve 6 módech:

1. "w" (zápis)
2. "r" (čtení)
3. "a" (připisování)
4. "r+" (čtení i zápis)
5. "w+" (čtení i zápis)
6. "a+" (čtení i připisování)

Do souboru můžeme zapisovat pomocí funkce fprintf.

fprintf(f, "vysledek = %d", vysledek);

A vypisovat ze souboru můžeme pomocí fscanf. Ta se často používá v cyklu while kde je podmínka aby běžel dokud se nedostane na EOF (end of file). Postupně tedy prochází soubor znak po znaku.

while(fscanf(f, " %c", &c) != EOF) {„ “

printf("%c", c);

}

**Binární**

**FILE** \*f = fopen("soubor.txt", "wb");

Lze je otevřít ve více módech stejně jako textové soubory:

1. "wb" (zápis)
2. "rb" (čtení)
3. "ab" (připisování)
4. "rb+" (čtení i zápis)
5. "wb+" (čtení i zápis)
6. "ab+" (čtení i připisování)

**Zápis do binárního souboru**

typedef struct {

int n1, n2, n3;

}Cisla;

int main()

{

int n;

Cisla num;

FILE \*f;

if((f = fopen("C:\\program.bin","wb")) == NULL) {

printf("Soubor se nepodařilo otevřít");

return -14;

}

for(n = 1; n < 5; ++n) {

num.n1 = n;

num.n2 = 5\*n;

num.n3 = 5\*n + 1;

fwrite(&num, sizeof(Cisla), 1, f);

}

fclose(f);

return 0;

}

**Čtení z binárního souboru**

typedef struct {

int n1, n2, n3;

}Cisla;

int main()

{

int n;

Cisla num;

FILE \*f;

if((f = fopen("C:\\program.bin","wb")) == NULL) {

printf("Soubor se nepodařilo otevřít");

return -14;

}

for(n = 1; n < 5; ++n) {

fread(&num, sizeof(Cisla), 1, f);

printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);

}

fclose(f);

return 0;

}

## Souborové funkce

**Fgets()**

fgets(retezec, pocet, soubor);

* načte pocet znaků ze souboru do řetězce retezec

**Fread()**

Platí pro binární soubor.

fread(&a, sizeof(struktura), 1, soubor);

* &a – adresa struktury nebo pole struktur
* sizeof(struktura) – velikost datového typu struktury
* 1 – počet načítaných položek struktur

**Feof()**

feof(soubor);

* informuje o tom, zda jsme dosáhli konce souboru
* pokud nejsme na konci vrací 0, jinak 1

**Fseek**

fseek(soubor, oKolik, odkud);

* přesune kurzor na pozici vzdálenou „oKolik“ bytů od pozice „odkud“

Za parametr „odkud“ lze také zvolit některou z konstant:

1. SEEK\_SET – začátek souboru
2. SEEK\_CUR – aktuální pozice kurzoru
3. SEEK\_END – konec souboru

Čili kdybych chtěl přesunout kurzor o 7 míst zpět od místa kde se právě nachází bude to vypadat takto:

fseek(f, -7, SEEK\_CUR);

A takto přesune pointer souboru na pozici 0 od začátku souboru:

fseek(f, 0, SEEK\_SET);

**Ftell()**

ftell(soubor);

* vrací hodnotu aktuální pozice kurzoru v souboru

**Rewind()**

rewind(soubor);

* posune kurzor na začátek souboru

## Testování otevření a uzavření souboru

Testování otevření je ukázáno v příkladech nahoře. Při nesprávném otevření vrací funkce fopen NULL.

Uzavření souboru se provádí pomocí fclose(f).

# 16. Základy OOP, třídy a objekty (Java)

* Strukturované programování, OOP
* Charakteristika pojmu objekt a třída
* Vlastnosti OOP

## Strukturované programování, OOP

Strukturované programování znamená, že je program rozdělen na části, které jsou pak spojeny v celek.

Objektově orientované programování je zaměřeno na vytváření objektů, které obsahují jak data, tak metody. Vychází z předpokladu, že každý program je simulací reálného či virtuálního světa a každý svět je tvořen objekty. Hlavní cíl OOP je spojit data a metody které je využívají dohromady tak aby žádná další část kódu nemohla k těmto datům přistoupit.

## Charakteristika pojmu objekt a třída

**Třída**

* zobecňuje a popisuje vlastnosti množiny podobných objektů
* je v podstatě synonymum pro datový typ

**Objekt**

* entita, kterou lze jednoznačně odlišit od jiného objektu
* zjednodušený pohled na skutečný předmět, osobu či pojem
* má své vlastnosti (atributy, členská data) a své metody (funkce)
* Od ostatních objektů se liší svou jedinečností, ale sdílí s nimi vlastnosti a způsob chováníObsah obrázku text, snímek obrazovky, Písmo, číslo

  Popis byl vytvořen automaticky

Objekt, který se vytvoří podle třídy, se nazývá **instance**. Instance mají stejné **rozhraní** jako třída, podle které se vytváří.

## Vlastnosti OOP – zapouzdření, dědičnost, polymorfismus

**Zapouzdření**

Umožňuje skrýt některé metody a atributy tak, aby zůstaly použitelné jen pro třídu zevnitř. Donutí tedy programátora používat objekt jen tím správným způsobem.

Používáme klíčových slov:

1. **Private** – přistup pouze v původní třídě
2. **Protected** – přístup v původní třídě a jejích potomcích
3. **Public** – přistup vždy a všude

**Dědičnost**

Jde o možnost odvozovat nové třídy, které dědí data a metody z jedné nebo více tříd. V reálném světě bychom řekli, že člověk je typem savce, auto je dopravním prostředkem, případně že voda je druhem nápoje. Každý z těchto potomků rozvíjí rodiče o své vlastní unikátní metody a atributy.

Využívá se klíčové slovo **extends** (pro určení rodiče) a metoda **super()** (zavolá konstruktor rodiče).

**Polymorfismus (vícetvarost)**

Vlastnost, která nám umožňuje pojmenovat nějakou konkrétní metodu identickým jménem, přičemž její funkčnost se může v jednotlivých třídách hierarchie tříd lišit.

Mějme například mnoho objektů, které reprezentují nějaké geometrické útvary (kruh, čtverec, trojúhelník). Bylo by jistě přínosné a přehledné, kdybychom s nimi mohli komunikovat jednotně, ačkoli se liší. Můžeme zavést třídu GeometrickyUtvar, která by obsahovala atribut barva a metodu vykresli. Všechny geometrické tvary by potom dědily z této třídy její rozhraní. Objekty kruh a čtverec se ale jistě vykreslují jinak.

Polymorfismus nám umožňuje přepsat si metodu vykresli u každé podtřídy tak, aby dělala, co chceme. Rozhraní tak zůstane zachováno a my nebudeme muset přemýšlet, jak se to u onoho objektu volá.

Využívá se klíčové slovo **override.**

@Override

**public void** vykresli() { ... }

OOP také dokáže rozeznat metody se stejným názvem (přetížení metod) díky jejich odlišnostem (atributy či jejich datové typy).

# 17. Funkce (Java)

* Stavba a vlastnosti metody
* Přetížení metod
* Statické metody
* Speciální metody

## Stavba a vlastnosti metody

Metody jsou schopnosti, které umí objekt vykonávat. Laicky bychom mohli říct, že jsou to funkce objektu. Jsou stejně jako třídy tvořeny hlavičkou a tělem. Hlavička začíná specifikátorem přístupu (public/private/protected) za ní následuje návratový datový typ. Další položkou je název metody a v kulatých závorkách čárkou oddělené parametry volání.

Když přiřadíme v hlavičce metody parametru nějakou hodnotu, stává se parametr nepovinným.

## Přetížení metod

Přetížení metody znamená, že v jedné třídě může existovat **více metod** stejného jména jen v případě, že každá má jiný seznam parametrů.

Příklad kde je přetížená metoda nakresli:

**public void** nakresli(**String** s)

{

...

}

**public void** nakresli(**int** i)

{

...

}

**public void** nakresli(**double** f)

{

...

}

**public void** nakresli(**int** i, **double** f)

{

...

}

## Statické metody

Statické metody (metody třídy) mohou přistupovat pouze ke statickým proměnným a lze je volat i v případě, že neexistuje žádný objekt dané třídy. Pro statické metody je důležité použít klíčové slovo **static**.

Statické metody se volají na třídě. Jedná se zejména o pomocné metody, které potřebujeme často používat a nevyplatí se nám tvořit instanci. Mnoho takových metod již známe, jen jsme si to neuvědomovali. Nikdy jsme např. netvořili instanci třídy System k tomu, abychom do ní mohli zapisovat. Metoda println() na atributu out na třídě System je statická, stejně jako atribut samotný. System je jen jeden a bylo by zbytečné tvořit si z něj instanci, když jej chceme používat.

Podobně je tomu např. i u metody round() ve třídě Math. Když chceme zaokrouhlit číslo, nebudeme si k tomu přece tvořit objekt. Jedná se tedy většinou o pomocné metody, kde by instanciace zbytečně zdržovala nebo nedávala smysl.

**Person**.getName(); //statická

**Person** person1 = **new** **Person**("Anna"); person1.getName(); //nestatická

## Speciální metody

Důležité je také zmínit speciální metody jako jsou **konstruktor**, **gettery** a **settery**.

**Konstruktor**

Jedná se pouze o „speciální“ metodu, která má za úkol vytvořit objekt. Pokud ho sami nevytvoříme, použije se implicitní od Javy – vytvoří pouze místa v paměti.

Má stejný název jako třída a oproti klasické metodě nemá návratový datový typ => nic nevrací, jeho úkol je vytvořit objekt.

tři druhy:

* **Implicitní konstruktor** - Tento konstruktor je výchozí, je automaticky vytvořen, pokud však vytvoříme vlastní, tento konstruktor nebude dostupný

| public Pes(){ } |
| --- |

* **Inicializační konstruktor** - Tento konstruktor požaduje na vstupu hodnoty, kterými objekt při vytvoření naplní

| public Pes(String jmeno, int vek) {  this.jmeno = jmeno;  this.vek = vek; } |
| --- |

* **Kopírovací konstruktor** - Tento konstruktor, jak už název vypovídá, umožňuje kopírovat objekty

| public Pes(Pes pes) {  this.jmeno = pes.jmeno  this.vek = pes.vek; } |
| --- |

**Gettery a Settery**

Jedná se o metody pro buď získání, nebo nastavení hodnoty atributu mimo třídu. Zavádí se, protože atributy ve třídách jsou zapouzdřeny klíčovým slovem private. Ve většině IDE se dají vygenerovat.

//Getter

**public** **double** getVyska()

{

**return** vyska;

}

//Setter

**public** **void** setVyska(**double** vyska)

{

**if**(vyska < 0){

**throw new** IllegalArgumentException("Vyska nesmi byt zaporna");

}

**this**.vyska = vyska;

}

# 18. Pole a ArrayList (Java)

* Charakteristika a práce s jednorozměrným polem
* Charakteristika a práce s vícerozměrným polem
* Třídění v poli
* Kolekce – charakteristika
* Charakteristika a práce s ArrayListem

## Charakteristika a práce s jednorozměrným polem

Při deklaraci pole nejprve uvedeme typ hodnot, které do něj budeme ukládat, za hranaté závorky a název proměnné tohoto pole.

Při inicializaci za rovnítko vepíšeme klíčové slovo new následované datovým typem pole a v hranatých závorkách uzavřenou celočíselnou hodnotu definující jak bude pole velké.

//pole osmi integeru

**int**[] array = **new** **int**[8];

//pole čtyř řetězců

**String**[] stringArray = **new** String[4];

**String**[] stringArray2 = {"Skakal", "pes", "pres", "oves"};

## Charakteristika a práce s vícerozměrným polem

Při deklaraci vícerozměrných polí postupujeme obdobně jako u pole jednorozměrného, jediným rozdílem je počet závorek, které uvádíme. Co dvojice závorek, to jeden rozměr. Na pravé straně přiřazení vždy musíme uvést velikost prvního rozměru (samotného pole polí). Velikost ostatních rozměrů vyplnit nemusíme, ale Java pak tyto rozměry nezinicializuje (to musíme udělat ručně).

//pole 3 řádky, 4 sloupce

**int**[][] array2d = **new** **int**[3][4];

**int**[][] array2d2 = **new** **int**[3][];

**public static void** print(**int**[][] array)

{

for(**int** i = 0; i < array.length; i++) {

for(**int** j = 0; j < array[i].length; j++) {

**System**.**out**.print(array[i][j] + " ");

}

**System**.**out**.println("");

}

}

Deklaraci s inicializací lze provést rovněž zjednodušeným způsobem stejně jako u jednorozměrného pole. Jediný rozdíl je v tom, že vytváříme pole polí.

**int**[][] setting = {

{0, 0, 4, 0, 3, 6, 9, 2, 7}

,{1, 0, 0, 0, 0, 5, 0, 0, 0}

,{0, 0, 0, 2, 0, 0, 0, 0, 4}

,{0, 0, 5, 0, 0, 0, 0, 6, 0}

,{6, 4, 0, 0, 0, 0, 0, 8, 5}

,{0, 7, 0, 0, 0, 0, 2, 0, 0}

,{5, 0, 0, 0, 0, 1, 0, 0, 0}

,{0, 0, 0, 7, 0, 0, 0, 0, 2}

,{4, 3, 7, 9, 2, 0, 5, 0, 0}

};

## Třídění v poli

### Bubble sort

Bubble sort v každém svém kroku porovnává dva sousední prvky, a pokud je ten větší zařazen vlevo od menšího, tak je prohodí a pokračuje na dalším indexu, kde postup opakuje. Proto pokud byl první porovnávaný prvek největší (nejlehčí), tak se probublá až na konec pole. Pokud byl v poli nějaký jiný lehčí, tak se na konec pole probublá on. Tak či tak, po prvním průchodu je na konci pole ten správný prvek. Nyní již zbývá pouze tento postup n-1 krát (poslední prvek je seřazen triviálně) zopakovat na redukovaném poli (tj. bez již seřazené části) a celé pole bude seřazeno od nejmenší hodnoty po tu nejvyšší.

**public static void** bubbleSort(**int**[] array)

{

for(**int** i = 0; i < array.length - 1; i++) {

for(**int** j = 0; j < array.length - i - 1; j++) {

if(array[j] > array[j + 1]) {

**int** temp = array[j];

array[j] = array[j + 1];

array[j + 1] = temp;

}

}

}

}

Náhradní varianta je použití **Arrays.sort(pole).**

## Kolekce (collections) – charakteristika

Pojem kolekce označuje soubor dat, které jsou stejného typu a slouží ke specifickému účelu. Disponuje snadným přidáváním a ubíráním prvků. Existuje jich velké množství a ačkoli se zvenku mnohdy tváří podobně, uvnitř fungují odlišně a vybíráme si je podle konkrétního účelu.

Hlavním rozdílem mezi polem a kolekcí je, že velikost pole je předem pevně dána a velikost kolekce lze dynamicky měnit v průběhu programu.

Java disponuje velkým množstvím předpřipravených kolekcí, např.:

* Seznam (list) – může obsahovat prvky stejného obsahu, udržuje se pořadí vložení prvků
* Množina (set) – nesmí obsahovat prvky stejného obsahu, neudržuje se pořadí vložení prvků (kontrola jedinečnosti nějaké informace např. emailu)
* Fronta (queue) – určeno pro další zpracování, fronty typu FIFO, s prioritou, se zpožděním, s omezeným přístupem (př.: zpracování objednávek)

## Charakteristika a práce s ArrayListem

* implementace seznamu (List)
* v podstatě pole s dynamickou velikostí, které se samo zvětšuje a zmenšuje
* rychlé operace pro přístup k prvkům pomocí indexů
* pomalejší při přidávání na již obsazený index
* zpomalení při nutné realokaci (nový ArrayList je o 50 % větší)

**ArrayList**<**String**> cars = **new ArrayList**<**String**>();

Přidání prvku do listu:

cars.add("BMW");

Získání prvku z listu:

cars.get(index);

Změna hodnoty prvku listu:

cars.set(0, "Ford");

Odstranění jednoho a všech prvků listu:

cars.remove(0);

cars.clear();

Vrácení počtu prvků:

cars.size();

Úplně stejné:

for (int i = 0; i < cars.size(); i++) { cars.get(i).getName(); }

for (var auto : cars) { auto.getName(); }

private String name;

public String getName() { return name; }

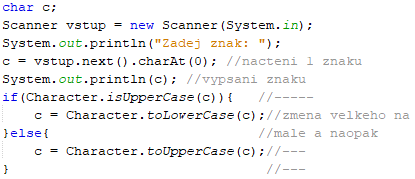
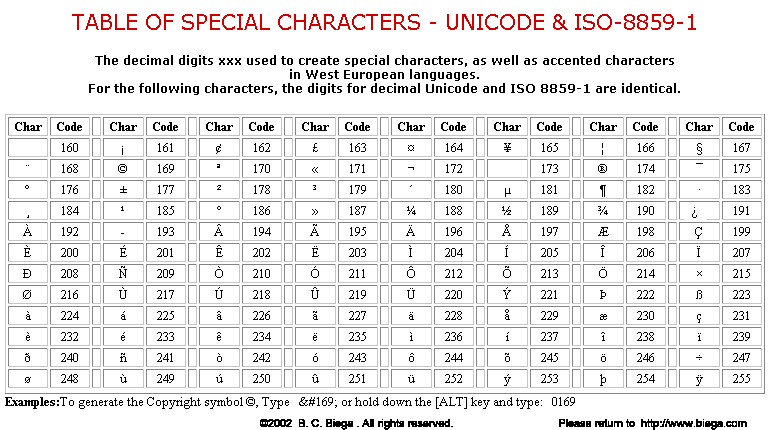
public Car(String name) { this.name = name; }

# 19. Znaky a řetězce (Java)

* Deklarace a charakteristika znakové proměnné
* Charakteristika a práce s řetězci
* Metody pro práci s řetězci

## Deklarace a charakteristika znakové proměnné

**char** a = 'a';

Znakové proměnné v Javě nevyužívají ASCII tabulku jako jazyk C, ale Unicode tabulku, která obsahuje tabulku ASCII jako podmnožinu. Velikost datového typu char v Javě je 2 byty, takže tedy 2 krát větší, než v C. 

## Charakteristika a práce s řetězci

String je v podstatě pole znaků (char) a dá se s ním také tak pracovat. Pro vybrání znaku na určitém místě v řetězci slouží třeba funkce charAt(cislo);

**String** s = "Hello world";

**String** s2 = **new String**("Hello world");

**System**.**out**.println(s);

**System**.**out**.println(s.charAt(2));

Dvě možnosti pro práci s řetězci:

1. Konstantní (read-only) – jednou vytvořený řetězec, již nelze měnit = třída **String**
   1. +Řetězec proměnné délky – je možné měnit délku řetězce = třída **StringBuffer(threadsafe)**/ **StringBuilder(rychlejší)**

## Metody pro práci s řetězci

Vrátí znak z pozice x:

s.charAt(x)

Spojení řetězců:

s1.concat(s2) nebo operátor +

Náhrada znaku v řetězci:

s.replace(stary, novy)

Vracení délky řetězce:

s.length()

Porovnání 2 řetězců:

s1.equals(s2)

Lexikografické porovnání 2 řetězců:

s1.compareTo(s2)

* vrací 0 jsou-li si rovny
* jinak > 0 když je s1 lexikograficky větší než s2 nebo < 0 když menší

Převod na malé a na velké znaky:

s = toLowerCase(s)

s = toUpperCase(s)

Vrácení částí řetězce:

ss = s.substring(od)

ss = s.substring(od, do + 1)

# 20. Soubory (Java)

* Práce s adresáři a soubory – třída File
* Znakové proudy – třídy Reader, Writer
* Binární proudy – třídy InputStream, OutputStream

Soubor je množina údajů uložená na vnější paměti počítače. Umožňují ukládat data mimo operační paměť počítače – data zůstanou zachována i po vypnutí počítače.

## Práce s adresáři a soubory – třída File

* odkazuje na cestu v adresáři
* je potřeba importovat java.io.File
* obsahuje metody pro práci se soubory a adresáři
* umožňuje definovat názvy souborů a adresářů nezávisle na platformě

**File** absolutni = **new File**("H:/program/gg/src/kopie\_ahoj.txt");

**File** relativni = **new File**("soubor.txt");

**Vytvoření a ověření existence souboru**

(boolean) **f.createNewFile()** – vytvoří nový soubor, pokud neexistuje (true)

(boolean) **f.exists()** – zjištění existence

(boolean) **f.isDirectory()** – zjištění, zda je soubor adresářem

(boolean) **f.isFile()** – zjištění, zda je zadaná cesta souborem

(String) **f.getAbsolutePath()** – zjištění absolutní cesty

**Operace se soubory**

(boolean) **renameTo(File name)** – přejmenuje soubor nebo adresář

(boolean) **delete()** – vymaže soubor nebo adresář

**Práce s adresáři**

(File[]) **d.listFiles()** – vrací pole typu File souborů a podadresářů adresáře d

(boolean) **d.mkdir(File nazev)** – Vytvoří adresář v adresáři d

V Javě se všechny operace se soubory provádí pomocí tzv. proudů.

## Znakové proudy – třídy Reader, Writer

**Reader**

**File** f = **new File**("C:/soubor.txt");

**FileReader** reader = **new FileReader**(f);

**int** i;

while((i = reader.read()) != -1) {

**char** c = (**char**)i;

**System**.**out**.println(c);

}

reader.close();

V tomto případě reader přečte text souboru f a následně se vypíšou znaky zapsané v souboru.

Reader je podobně jako writer vždy nutno zavřít po ukončení práce s ním metodou **close().**

**Writer**

**File** f1 = **new File**("H:/program/gg/src/ahoj.txt");

**File** f2 = **new File**("H:/program/gg/src/kopie\_ahoj.txt");

**FileWriter** writer = **new FileWriter**(f1);

**FileReader** reader = **new FileReader**(f2);

while((**int** i = reader.read()) != -1) {

**System**.**out**.print((**char**)i);

writer.write(i);

}

writer.close();

reader.close();

V tomto případě je ukázáno čtení z jednoho souboru a následné kopírování daného souboru do jiného souboru znak po znaku.

## Binární proudy – třídy InputStream, OutputStream

Binární proudy umožňují přenést libovolná data. Vstupní proudy dědí z třídy InputStream ,výstupní z OutputStream . Základní operací definovanou v InputStreamu je metoda read, pomocí které můžeme z proudu přečíst jeden bajt. Analogicky výstupní proud definuje metodu write. Názvy metod jsou tedy stejné jako u proudů znakových.

**File** bity = **new File**("H:\\program\\src\\ahoj.dat");

**File** bityKop = **new File**("H:\\program\\src\\ahoj\_kopie.dat");

**FileInputStream** reader = **new FileInputStream**(bity);

**FileInputStream** readerKop = **new FileInputStream**(bityKop);

**FileOutputStream** writer = **new FileOutputStream**(bity);

**FileOutputStream** writerKop = **new FileOutputStream**(bityKop);

**int** i = 0;

for(**int** j = 1; j <= 10; i++) {

writer.write(j);

}

while((i = reader.read()) != -1) {

**System**.**out**.print((**char**)i);

writerKop.write(i);

}

**System**.**out**.println("ZKOPIROVANO");

writerKop.close();

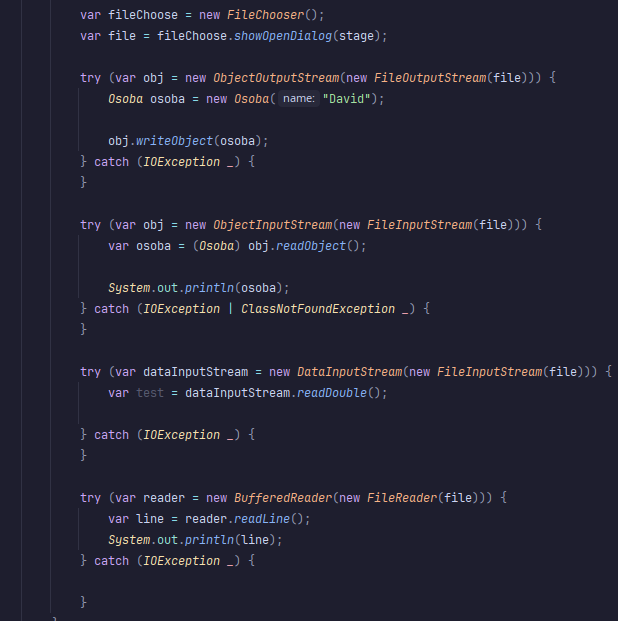
readerKop.close();

writer.close();

reader.close();

Tato funkce slouží stejně jako předchozí ukázky ke čtení a následnému zápisu do souboru.

Pouziti input steramu:



# 21. Třídy a objekty – dědičnost (Java)

* Dědičnost a kompozice
* Konstruktory v dědičnosti
* Přístupová práva a dědičnost
* Klíčové slovo final

## Dědičnost a kompozice

**Dědičnost**

Dědičnost nám umožňuje vytvářet hierarchie tříd, ve kterých můžeme o libovolném uzlu říct, že je speciálním případem libovolného ze svých předků. V reálném světě bychom řekli, že člověk je typem savce, auto je dopravním prostředkem, případně že voda je druhem nápoje.

Mezi hlavní výhody dědění patří:

* zásadním způsobem přispívá k znovupoužitelnosti programového kódu
* dědičnost je základem polymorfismu (přetížení metod)

**Kompozice(lepší)**

Způsob, jak kombinovat jednoduché objekty nebo typy dat do složitějších. Objektová kompozice je důležitý nástroj pro vytváření pokročilých datových struktur, jako je například spojový seznam nebo binární strom. Tyto složené datové struktury jsou dále používány stejným způsobem jako základní objekty.

## Konstruktory v dědičnosti

Konstruktor potomka musí vždy obsahovat parametry rodiče toho lze dosáhnout za použití metody super();

Při jednotlivých voláních metod podtypů často narazíme na to, že nechceme celou metodu překrýt, pouze k ní chceme přidat další funkcionalitu. V tento okamžik můžeme zavolat super.jmenoMetody(), čímž zavoláme funkcionalitu předka. Analogicky k řetězení konstruktorů this() můžeme volat konstruktor předka voláním super() – toto volání musí být v rámci konstruktoru potomka vždy na prvním místě.Obsah obrázku text, snímek obrazovky, Písmo, číslo

Popis byl vytvořen automaticky

## Přístupová práva a dědičnost

**Private**

Pokud nastavíme v předkovi u proměnné nebo u funkcí, tak u potomka nebudeme moci dané části použít. Funkce a proměnné může používat a měnit pouze daná třída, ve které jsou deklarovány

**Protected**

Volíme tehdy, pokud chceme, aby potomek mohl využívat a měnit proměnné a používat funkce předka. Pro normálního uživatele v main však zůstanou údaje skryté stejně, jako u private. K proměnným/funkcím se dostaneme pomocí slovíčka **super**

(ukázka nedává z logického hlediska smysl – pouze prezentuje přístupová práva)

Obsah obrázku text, snímek obrazovky, Písmo

Popis byl vytvořen automatickyObsah obrázku text, snímek obrazovky, Písmo

Popis byl vytvořen automaticky

**Public**

Zpravidla se moc nedoporučuje používat public u proměnných. Používá se především u funkcí, aby k nim měl uživatel v main přístup.

**Obsah obrázku text, snímek obrazovky, číslo, Písmo

Popis byl vytvořen automaticky**

## Klíčové slovo final

Toto klíčové slovo říká, že z dané třídy už nelze dále dědit další potomky, nebo že metodu nelze dále překrýt.

Obsah obrázku text, snímek obrazovky, Písmo, řada/pruh

Popis byl vytvořen automaticky

# 22. Třídy a objekty – rozhraní (Java)

* Rozhraní a jeho konstrukce
* Abstraktní třída
* Využití rozhraní a abstraktní třídy

## Rozhraní a jeho konstrukce

Rozhraní slouží k vícenásobnému dědění. Definují soubor metod, které musí být implementovány v jakékoliv třídě, která implementuje toto rozhraní. Obsahuje pouze konstanty a metody bez těla. Zatímco pro dědění tříd (a dědění rozhrání mezi sebou) využíváme v hlavičce klíčové slovo extends, tak pro implementaci rozhraní používáme slovo **implements**.

Vytvoření rozhraní:

**public** **interface** PtakInterface

{

**void** pipni();

**void** dychej();

}

Implementace rozhraní do třídy Ptak:

**public** **class** Ptak **implements** PtakInterface

Main :

Obsah obrázku text, snímek obrazovky, software, displej

Popis byl vytvořen automaticky

Třída může implementovat více rozhraní (na rozdíl od dědění)

**public** **class** Ptak **implements** PtakInterface, DalsiRozhrani

## Abstraktní třída

Abstraktní třídy jsou třídy, které obsahují alespoň jednu abstraktní metodu nebo jsou samy označeny jako abstraktní pomocí klíčového slova **abstract**.Pokud toto slovo vepíšeme do hlavičky třídy, tak z ní nelze vytvářet instance.

U abstraktních metod můžeme vynechat jejich tělo a napsat místo něj pouze středník, čímž dosáhneme toho, že tuto metodu musí mít implementovanou všichni neabstraktní potomci.

**public abstract** class **Employee**

{

**protected String** name;

**protected int** age;

**public Employee**(**String** name, **int** age)

{

**this**.name = name;

**this**.age = age;

}

**public abstract void** work();

**public void** sayHello()

{

**System**.**out**.println("Dobry den, jmenuji se " + name + " a je

mi " + age + " let.");

}

}

## Využití rozhraní a abstraktní třídy

Rozhraní využijeme, když chceme definovat specifikaci pro třídy, které mají poskytnout určité chování, ale nechceme specifikovat konkrétní implementaci. Tím umožňujeme, aby různé třídy poskytly vlastní implementaci rozhraní a zároveň zachovávají konzistentní rozhraní pro komunikaci s ostatními částmi programu.

Abstraktní třídu využijeme, když chceme poskytnout základní implementaci a strukturu pro skupinu souvisejících tříd, ale zároveň nechceme, aby tato třída nebyla instancovatelná. Abstraktní třída může obsahovat jak abstraktní, tak konkrétní metody, což umožňuje sdílet společnou funkcionalitu mezi odvozenými třídami a zároveň vyžaduje, aby tyto třídy implementovaly určité metody.

# 23. Java a databáze – DDL, DML

* Nastavení projektu
* Připojení k databázi
* DDL
* DML

## Nastavení projektu

Pro připojení k databázi je potřeba přidání knihovny JDBC (Java Database Connectivity). Chceme-li pracovat s MySQL databází, dosáhneme toho tak, že stáhneme mysql-connector.jar soubor, vložíme jej do lib složku projektu a přes Project Structure modul přidáme.

## Připojení k databázi

Když máme úspěšně nastavený projekt, můžeme přistoupit k připojení k databázi. Před prací s jakýmikoliv třídami a jejich metodami musíme ještě importovat balíček java.sql.

**import** java.sql.\*;

Samotné připojení v sobě uchovává instance třídy Connection, které připojení přiřadíme metodou getConnection() třídy DriverManager.

**Connection** con = **DriverManager**

.getConnection("jdbc:mysql://localhost:3306", user, password);

Takovýto url řetězec používáme v případě, že se připojujeme k databázi MySQL na lokálním počítači. Localhost v url adrese je server, na který se připojujeme a číslo 3306 značí číslo portu tohoto serveru.

Pokud máme databázi již vytvořenou, dopisujeme na konec url řetězce její název.

"jdbc:mysql://localhost:3306/nazevDatabaze"

## DDL

Data Definition Language (DDL) slouží k definici a úpravě struktury databáze. Příkazy voláme pomocí instance třídy Statement.

**Statement** stmt = con.createStatement();

stmt.execute(**String** operace);

Jako parametr funkce execute() se již zadávají konkrétní DDL operace, jako:

**Vytvoření databáze**

Pomocí příkazu CREATE DATABASE vytváříme novou databázi v databázovém systému.

Syntaxe je obvykle následující:

CREATE DATABASE nazev\_databaze;

nebo

CREATE DATABASE IF NOT EXISTS nazev\_databaze;

**Vytvoření tabulky**

Pomocí příkazu CREATE TABLE vytváříme nové tabulky v databázi a definujeme jejich sloupce a datové typy.

CREATE TABLE tabulka (

sloupec1 typ\_dat

,sloupec2 typ\_dat

);

Konkrétní příklad:

CREATE TABLE Zaci (

idZa int NOT NULL PRIMARY KEY AUTO\_INCREMENT

,jmeno varchar(16) NOT NULL

,prijmeni varchar(16) NOT NULL

);

**Vytvoření relace**

Relace mezi tabulkami definujeme pomocí cizích klíčů (Foreign Keys). Tyto klíče propojují sloupce v jedné tabulce se sloupci v jiné tabulce.

Syntaxe pro definici cizího klíče vytvořením tabulky je obvykle následující:

ALTER TABLE tabulka1

ADD CONSTRAINT fk\_tab1\_tab2 FOREIGN KEY(sloupec1)

REFERENCES tabulka2(sloupec2);

## DML

Data Manipulation Language (DML) je jazyk používaný k manipulaci s daty v databázovém systému. Zahrnuje operace jako vkládání, aktualizace, mazání a výběr dat z databázových tabulek.

Příkazy jsou tedy:

* INSERT
* UPDATE
* DELETE
* SELECT

**INSERT**

INSERT INTO Zamestnanci(ID\_za, jmeno, prijmeni, email)

VALUES(101, 'Jan', 'Novák', 'jan.novak@gmail.com');

**UPDATE**

UPDATE Zamestnanci

SET plat = 5500

WHERE ID\_za = 101;

**DELETE**

DELETE FROM Zamestnanci

WHERE ID\_za = 101;

**SELECT**

SELECT \* FROM Zamestnanci;

# 24. Java a databáze – SELECT

* Nastavení projektu
* Připojení k databázi
* SELECT

## Nastavení projektu

Popsáno v přechozí otázce.

## Připojení k databázi

Také popsáno předchozí otázce.

## SELECT

SELECT je SQL příkaz, který používáme k získání dat z databáze. V Javě můžeme provádět SELECT dotazy pomocí objektů Statement nebo PreparedStatement.

**Statement**

Statement je třída, která představuje předkompilovaný SQL příkaz. K vytvoření instance Statementu používáme metodu createStatement() objektu Connection. Metoda executeQuery() tohoto rozhraní nám umožňuje provést SELECT dotaz a vrátit výsledek jako objekt ResultSet.

**Statement** stmt = con.createStatement();

**ResultSet** resSet = stmt.executeQuery( "SELECT \* FROM Zamestnanci");

while(resSet.next()) {

**int** id = resSet.getInt("id");

**String** name = resSet.getString("name");

//atd.

}

**PreparedStatement**

PreparedStatement je třída, která představuje předkompilovaný SQL příkaz s parametry, které můžeme později nahradit konkrétními hodnotami. K vytvoření instance PreparedStatementu používáme metodu prepareStatement() objektu Connection. Poté můžeme nastavit parametry pomocí metod setX(), kde X odpovídá datovému typu parametru. Metoda executeQuery() nám poté umožní provést SELECT dotaz a vrátit výsledek jako objekt ResultSet (stejně jako u Statementu).

**PreparedStatement** prepStmt = con.prepareStatement(

"SELECT \* FROM Zamestnanci WHERE oddeleni = ?");

prepStmt.setString(1, "IT"); //indexuje se od 1

**ResultSet** resSet = prepStmt.executeQuery();

while(resSet.next()) {

**int** id = resSet.getInt("id");

**String** name = resSet.getString("name");

//atd.

}

# 25. Tvorba grafického rozhraní aplikací, rozmístění (JavaFX)

* Soubory pro tvorbu JavaFX Application
* Základní pojmy – Stage, …
* Rozložení
* Práce s komponentami

## Soubory pro tvorbu JavaFX Application

Projekt se skládá ze 3 souborů:

**HelloApplication.java**

Soubor obsahuje spustitelné metody. V novém projektu obsahuje pouze metody main() a start(). V metodě start() pomocí FXMLLoaderu načte FXML soubor a na jeho základě vytvoří tzv. scénu –scéna je v podstatě okno aplikace. Následně se objektu stage tato scéna přiřadí a zobrazí se na obrazovce.

**public class** HelloApplication **extends** Application

{   
 @Override  
 **public void** start(**Stage** stage) **throws** IOException

{  
 **FXMLLoader** fxmlLoader = **new**

**FXMLLoader**(**HelloApplication**.**class**.getResource("hello-

view.fxml"));  
 **Scene** scene = **new Scene**(fxmlLoader.load(), 320, 240);  
 stage.setTitle("Hello!");  
 stage.setScene(scene);  
 stage.show();  
 }  
  
 **public static void** main(**String**[] args)

{  
 *launch*();  
 }  
}

**HelloController.java**

Ve své podstatě funguje jako ovladač pro FXML komponenty.

**public class** HelloController **implements** Initializable

{  
 @FXML  
 **private Label** welcomeText;  
  
 @FXML  
 **protected void** onHelloButtonClick()

{  
 welcomeText.setText("Welcome to JavaFX Application!");  
 }

...

@Override  
**public void** initialize(**URL** url, **ResourceBundle** resourceBundle) {

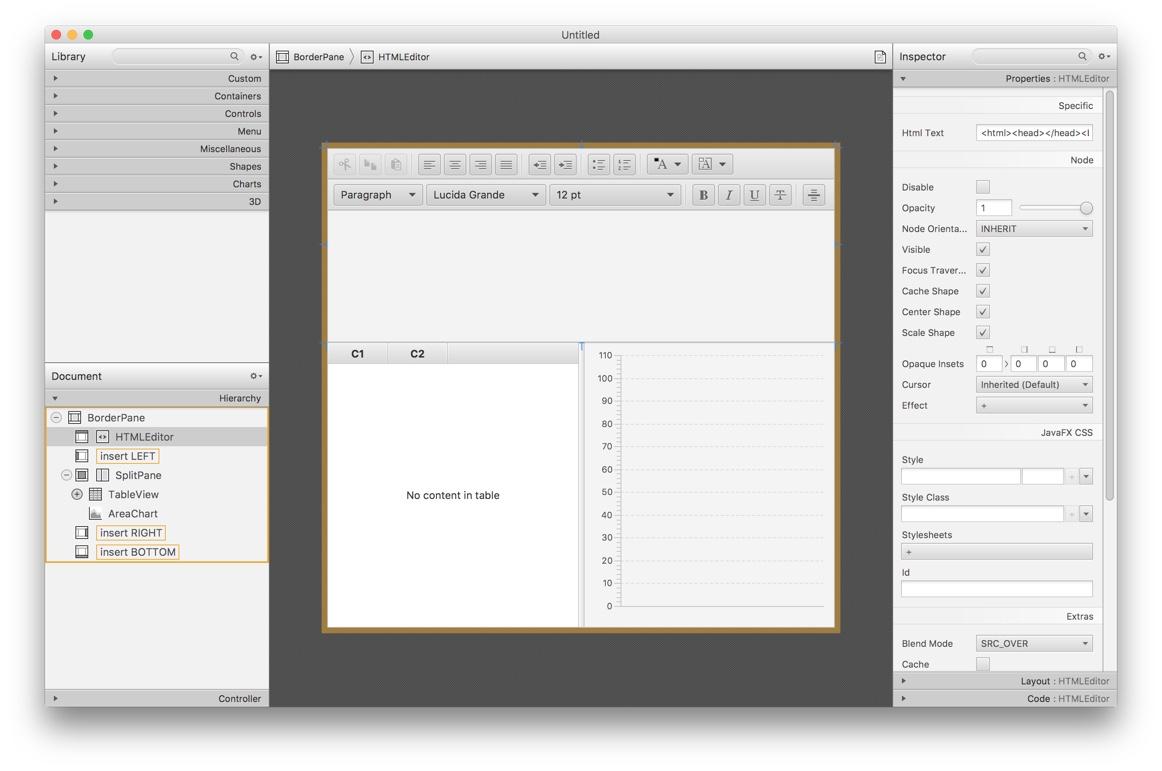
...  
}

}

V tomto příkladě implementuje třída HelloController rozhrání Initializable, která vynutí, aby kód obsahoval inicializační metodu.

**Hello-view.fxmlc**

Obsahuje FXML kód, který popisuje, jak aplikace vypadá. Možnost upravovat v JavaFX Scene Builderu.



## Základní pojmy – Stage, …

### Třída Stage

V jednoduchosti je to objekt, představující jedno okno aplikace.

The JavaFX Stage class is the top level JavaFX container. The primary Stage is constructed by the platform. Additional Stage objects may be constructed by the application. Stage objects must be constructed and modified on the JavaFX Application Thread.

### Scéna

Chceme-li v aplikaci JavaFX něco zobrazit, potřebujete scénu. Okno (Stage) může zobrazit pouze jednu scénu najednou, ale je možné ji za běhu vyměnit (vlastně stejně jako v divadle).

### WebView

Komponenta, která umožňuje kombinovat desktopovou aplikaci s webovou aplikací.

Užitečné například pokud již máme slušnou webovou aplikaci, ale potřebujete některé funkce, které lze rozumně poskytnout pouze u desktopové aplikace – například přístup na disk.

### Ovládací prvky

Jsou komponenty, které poskytují nějaký druh ovládací funkce uvnitř aplikace JavaFX. Například button nebo CheckBox.

Ovládací prvky:

Accordion, Button, CheckBox, ChoiceBox, ColorPicker, ComboBox, DatePicker, LabelList, ViewMenu, MenuBar, PasswordField, ProgressBar, RadioButton, Slider, Spinner, SplitMenuButton, SplitPane, TableView, TabPane, TextArea, TextField, TitledPane, ToggleButton, ToolBar, TreeTableView, TreeView...

## Rozložení

Pro rozložení jsou použity komponenty, které v sobě obsahují další komponenty (ovládací prvky apod.). Komponenta rozložení se stará o rozložení komponent vnořených do ní. Aby byly jakékoliv komponenty vidět v aplikaci, musí být přiřazené k alespoň jedné komponentě rozložení, která je přiřazena dané zobrazené scéně. Defaultně jsou v AnchorPane.

Komponenty rozložení:

Group, Region, Hbox, Vbox, Pane, FlowPane, BorderPane, StackPane, TilePane, GridPane, AnchorPane, TextFlow

**GridPane** – slouží k rozmístění prvků stejně jako v tabulce

**Hbox** – horizontální rozmístění prvků

**Vbox** – vodorovné rozmístění prvků

## Práce s komponentami

Práce s komponentami v JavaFX zahrnuje manipulaci s jednotlivými vizuálními prvky naší aplikace, jejich přizpůsobení a interakci s nimi.

Zde jsou některé způsoby, jak můžeme pracovat s komponentami:

### Nastavení vlastností komponent

Každá komponenta v JavaFX má různé vlastnosti, které můžeme nastavit podle našich potřeb. Například můžeme měnit barvy, velikosti, fonty a další vlastnosti, abychom dosáhli požadovaného vzhledu a chování naší aplikace.

### Přidávání a odebírání komponent

Dynamicky můžeme přidávat nové komponenty do scény nebo odebírat již existující komponenty podle potřeby. To nám umožní vytvářet flexibilní a interaktivní uživatelská rozhraní, která se mohou měnit v závislosti na akcích uživatele.

### Ovládání událostí

Jednou z klíčových funkcí JavaFX je možnost reagovat na různé události vyvolané uživatelem. Například můžeme programovat reakci na kliknutí na tlačítko, vybrání položky z rozbalovacího seznamu nebo změnu hodnoty posuvníku.

### Manipulace s daty

Pokud aplikace pracuje s daty, můžeme jednotlivé komponenty využít k získání dat od uživatele, jejich zpracování a zobrazení. Například můžeme získávat text z textového pole, vybírat hodnoty z rozbalovacího seznamu nebo zobrazovat tabulku s daty.

Práce s komponentami umožňuje vytvářet bohaté a interaktivní uživatelské rozhraní pro JavaFX aplikace a poskytuje širokou škálu možností pro tvorbu moderních a uživatelsky přívětivých aplikací.

# 26. Tvorba grafického rozhraní aplikací, Scene Builder (JavaFX)

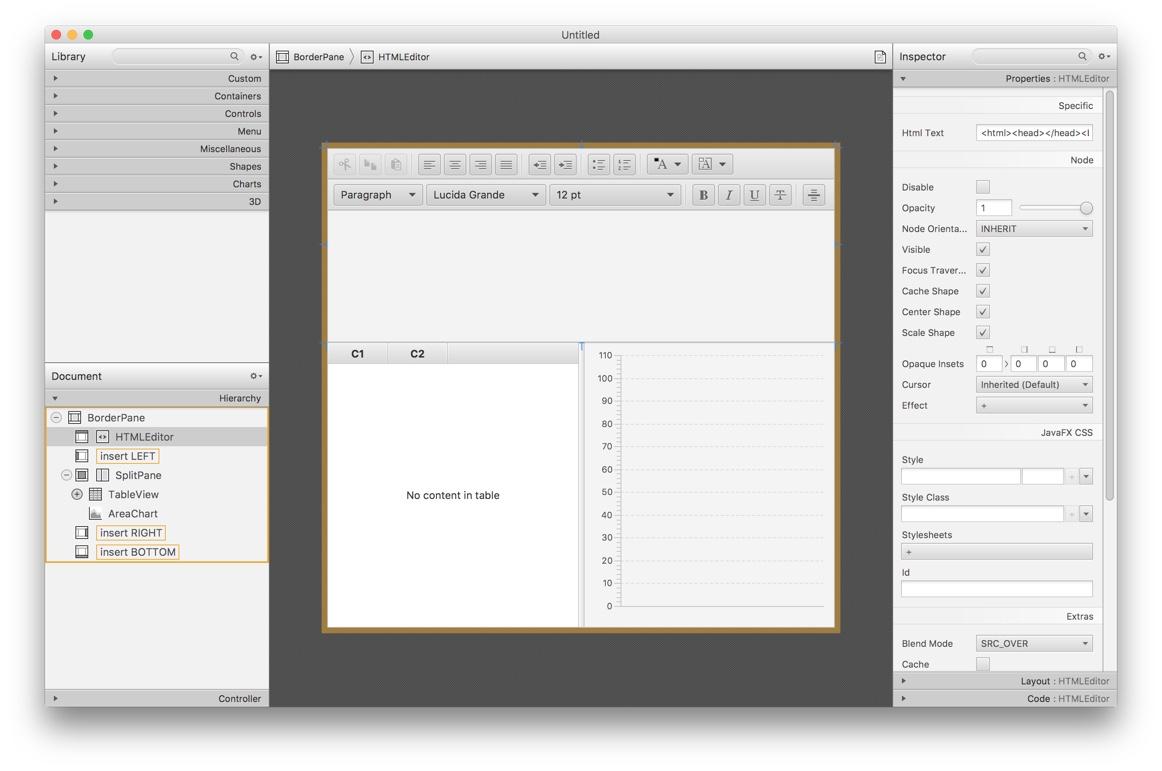
* Soubory pro tvorbu JavaFXML Application
* Prostředí Scene Builder
* Práce s komponentami

## Soubory pro tvorbu JavaFXML Application

Popsáno v předešlé otázce.

## Prostředí Scene Builder

Scene Builder je vizuální nástroj, který umožňuje snadnou tvorbu uživatelského rozhraní JavaFXML aplikací.



S jeho pomocí můžeme:

* přidávat komponenty na scénu pomocí přetahování a pouštění
* nastavovat vlastnosti komponent a jejich rozmístění
* definovat obsluhu událostí přímo v editoru

## Práce s komponentami

Práce s komponentami v Scene Builderu je intuitivní a snadná:

**Přidávání komponent**

Stačí vybrat komponentu z knihovny a přetáhnout ji na scénu. Poté můžeme nastavovat její vlastnosti a rozmisťovat ji podle potřeby.

**Nastavování vlastností**

Každá komponenta má různé vlastnosti, které můžeme nastavit přímo v editoru. To zahrnuje FX:id, barvy, velikosti, texty atd.

**Manipulace s rozvržením**

Můžeme jednoduše přesouvat a zarovnávat komponenty na scéně, abychom dosáhli požadovaného vzhledu.

**Definování obsluhy událostí**

Můžeme přidávat metody pro obsluhu událostí, jako je kliknutí na tlačítko nebo změna hodnoty v poli, přímo ve Scene Builderu.

Scene Builder je užitečným nástrojem pro rychlou a efektivní tvorbu uživatelského rozhraní JavaFXML aplikací a usnadňuje proces vytváření moderních a atraktivních uživatelských rozhraní.

# 27. Komponenty ze skupiny Controls – charakteristika a použití, základní vlastnosti a události (JavaFX)

* Komponenty Button, CheckBox, RadioButton
* Komponenty Label, TextField, TextArea
* Komponenty ListView, ComboBox
* Komponenta ColorPicker

Jsou prostředkem, jak se aplikace dorozumívá s uživatelem. Abychom mohli s komponenty jakkoliv komunikovat, musí mít nastavené FX:ID (jedinečný identifikátor).

## Komponenty Button, CheckBox, RadioButton

Slouží pro práci s myší.

**Button**Obsah obrázku text, snímek obrazovky, Písmo

Popis byl vytvořen automaticky

* prvek používají se k vyvolání nějaké akce
* reaguje na klik uživatele
* stylují se podle pozice kurzoru

Nastavení textu tlačítka:

btn.setText(“Vyhledat“)

**CheckBox**

* prvek poskytující boolean hodnotu
* reaguje na klik uživatele

Boolean funkce, zda je CheckBox zaškrtnutý:

chb.isSelected()

Zaškrtnutí CheckBoxu přes kód:

chb.setSelected(true)

**RadioButton**Obsah obrázku text, snímek obrazovky, Písmo, číslo

Popis byl vytvořen automaticky

* prvek sloužící k výběru mezi několika možnostmi
* seskupují se do skupin – ToggleGroup

@FXML

**private** **ToggleGroup** tgPohlavi;

**RadioButton** rbPohlavi = (**RadioButton**)tgPohlavi.getSelectedToggle();

if(rbPohlavi != **null**) {

**System**.**out**.println(“Pohlaví: “ + rbPohlavi.getText());

}

## Komponenty Label, TextField, TextArea

Slouží pro práci s textem.

**Label**

* prvek sloužící k zobrazování textu
* můžou na něj být aplikovány různé vizuální efekty
* minimální uživatelská interakce

Nastavení textu pro Label:

lbl.setText(“Nadpis“);

**TextField**

* prvek pro zadávání textu
* dovede zobrazovat i nápovědu a chyby

Získání textu z TextFieldu:

tf.getText()

Nastavení textu:

tf.setText(““);

**TextArea**Obsah obrázku text, Písmo, řada/pruh, Obdélník

Popis byl vytvořen automaticky

* prvek umožňující zadávat text zahrnující více řádků

Získání textu z TextArei:

ta.getText()

Nastavení textu:

ta.setText(““);

## Komponenty ListView, ComboBox

Slouží pro zobrazení seznamů hodnot.

**ListView**

* prvek zobrazující seznam hodnot
* podle nastavení je možné vybírat jeden nebo i více prvků
* je neustále rozbalený, a zabírá hodně místa
* lze scrollovat

Vrácení ObservableListu obsaženém v ListView:

ol.getItems()

Nastavení seznamu prvků do ListView:

ol.setItems(**ObservableList**<?> items)

**ComboBox (jedna moznost, není nonstop rozbalen)**

Umožňuje uživatelům vybrat si možnost z předdefinovaného seznamu možností nebo zadat jinou hodnotu, pokud žádná z předdefinovaných voleb neodpovídá tomu, co chce uživatel vybrat

Vrácení ObservableListu obsaženém v ComboBoxu:

cb.getItems()

Nastavení seznamu prvků do ComboBoxu:

cb.setItems(**ObservableList**<?> items)

Vrácení aktuálně vybrané hodnoty z ComboBoxu:

cb.getValue()

Obsah obrázku text, Písmo, číslo, řada/pruh

Popis byl vytvořen automaticky

## Komponenta ColorPicker

Umožňuje uživateli vybrat barvu v rozevíracím dialogu.

cp.setOnAction(**new EventHandler**<**ActionEvent**>()

{

@Override

**public void** handle(**ActionEvent** event)

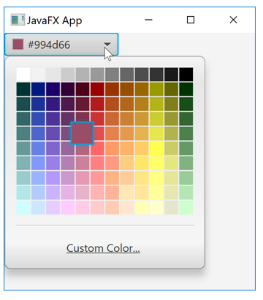
{

**Color** color = cp.getValue();

**System**.**out**.println("Vybraná barva: " + color.toString());

}

});



# 28. Komponenty ze skupiny Menu – charakteristika a použití, základní vlastnosti a události (JavaFX)

* Komponenta MenuBar ze skupiny Controls
* Komponenty Menu, MenuItem
* Komponenty CheckMenuItem, RadioMenuItem
* Komponenta Separator

Komponenty ze skupiny Menu se vytváří v souboru s main().

## Komponenta MenuBar ze skupiny Controls

* jedná se o komponentu zobrazující lištu s nabídkamiObsah obrázku text, snímek obrazovky, Písmo, číslo

  Popis byl vytvořen automaticky
* běžně se nachází na horní části aplikace a obsahuje různé nabídky (Menu)
* uživatelé mohou klikat na nabídky a zobrazovat podnabídky
* před použití MenuBar se musí vytvořit její instance v metodě start()
* instanci zviditelníme tak, že jí přidáme do Vboxu scény
* tím také MenuBar přidáme do horní části aplikace

**public void** start(**Stage** stage)

{

stage.setTitle("JavaFX App");

**MenuBar** menuBar = **new MenuBar**();

**VBox** vBox = **new VBox**(menuBar);

**Scene** scene = **new Scene**(vBox, 960, 600);

    stage.setScene(scene);

     stage.show();

}

## Komponenty Menu, MenuItem

**Komponenta Menu**Obsah obrázku text, snímek obrazovky, Písmo, řada/pruh

Popis byl vytvořen automaticky

* jedná se o svislou nabídku v MenuBar, která obsahuje další položky (MenuItem)
* po kliknutí na ní se zobrazí seznam dalších možností, které uživatel může vybrat
* před vytvořením tvoří MenuBar jen prázdnou lištu
* pokud je MenuBar vytvořený, můžeme mu vytvořit instanci

**Menu** menu1 = **new Menu**("Menu 1");

menuBar.getMenus().add(menu1);

Existuje několik Eventů, které lze volat v rámci Menu:

menu1.setOnShowing(e -> { **System**.**out**.println("Showing Menu1"); });

menu1.setOnShown (e -> { **System**.**out**.println("Shown Menu 1"); });

menu1.setOnHiding (e -> { **System**.**out**.println("Hiding Menu 1"); });

menu1.setOnHidden (e -> { **System**.**out**.println("Hidden Menu 1"); });

**Komponenta MenuItem**

* jedná se o položku v nabídce (Menu)
* reprezentuje jednu volbu, kterou může uživatel vybrat
* po kliknutí na tuto položku se provádí určitá akce

**MenuItem** menuItem = **new MenuItem**("Uložit");

menu1.getItems().add(menuItem);

## Komponenty CheckMenuItem, RadioMenuItem

**Komponenta CheckMenuItem**Obsah obrázku text, snímek obrazovky, Písmo, Značka

Popis byl vytvořen automaticky

* speciální typ MenuItem
* může být označen jako zaškrtnutý nebo nezaškrtnutý

**CheckMenuItem** checkMenuItem = **new**

**CheckMenuItem**("Check this!");

**Komponenta RadioMenuItem**

* další speciální typ MenuItem
* uživatel může zvolit pouze jednu možnost z dané skupiny (ToggleGroup)

**RadioMenuItem** radioMenuItem1 = **new RadioMenuItem**("Možnost 1");

**RadioMenuItem** radioMenuItem2 = **new RadioMenuItem**("Možnost 2");

**ToggleGroup** tg = **new** **ToggleGroup**();

radioMenuItem1.setToggleGroup(tg);

radioMenuItem2.setToggleGroup(tg);

## Komponenta SeparatorA JavaFX MenuBar with a Menu containing two menu items with a separator in between.

* Jedná se o vizuální oddělovač
* slouží k vizuálnímu oddělení různých částí uživatelského rozhraní
* Používá se ke zvýraznění oddělení mezi různými prvky, jako jsou tlačítka, položky menu apod.

**Separator** separator = **new Separator**();

Oddělovače položek nabídky jsou představovány třídou SeparatorMenuItem:

**MenuItem** item1 = **new MenuItem**( "Item 1");

**MenuItem** item2 = **new MenuItem**( "Item 2");

**SeparatorMenuItem** separator = **new SeparatorMenuItem**();

menu.getItems().add(item1);

menu.getItems().add(separator);

menu.getItems().add(item2);

# 29. Komponenty pro práci s databází – charakteristika a použití, základní vlastnosti a události (JavaFX)

* Připojení k databázi
* Komponenta TableView
* Komponenta TableColumn
* Komponenta TableCell

## Připojení k databázi

Vysvětleno v otázce 23.

## Komponenta TableView

TableView je asi nejčastěji používaná komponenta pro zobrazování tabulkových dat v JavaFX. Umí zobrazovat data ve formě tabulky.

* definují jednotlivé sloupce tabulky pomocí komponenty TableColumn
* obsah tabulky je určen datovou kolekcí, například ObservableList
* Selection Model umožňuje určit, které položky v tabulce jsou vybrány
* podporuje řazení dat v tabulce podle hodnot v jednotlivých sloupcích
* umožňuje uživateli přímo v tabulce editovat hodnoty

Při deklaraci píšeme za TableView do špičatých závorek typ dat, který bude tabulka zobrazovat.

@FXML

**public TableView**<**Knihy**> tvKnihy;

Naplnění tabulky:

tvKnihy.setItems(**ObservableList** list)

Vymazání dat z tabulky:

tvKnihy.getColumns().clear();

## Komponenta TableColumn

TableColumn reprezentuje jednotlivé sloupce v TableView. Každý TableColumn musí být propojen s odpovídajícím datovým atributem v datech, která jsou zobrazena v tabulce.

Při deklaraci píšeme za TableColumn do špičatých závorek 2 atributy – typ dat, který bude sloupec zobrazovat. Tento typ by měl odpovídat typu dat, který je součástí každého řádku tabulky (je také v deklaraci tabulky. Druhým parametrem je datový typ konkrétního sloupce. V konstruktoru udáváme, jaký název bude sloupec mít.

**TableColumn**<**Knihy**, **Integer**> col1 = **new TableColumn**<>("Id");

CellValueFactory určuje, jaký atribut třídy bude zobrazen v daném sloupci.

col1.setCellValueFactory(**new PropertyValueFactory**<>("ID\_k"));

## Komponenta TableCell

TableCell představuje jednotlivé buňky v tabulce. Každá buňka odpovídá jedné konkrétní položce v tabulce a je zodpovědná za zobrazení a případnou editaci hodnoty této položky.

* umožňuje uživateli upravit hodnotu buňky přímo v tabulce
* buňka může obsahovat libovolné prvky pro zobrazení dat, například tlačítka, ikony atd.

**Příkladné použití tabulky**

//Definování tabulky

@FXML

**public TableView**<**Knihy**> tvKnihy;

**TableColumn**<**Knihy**, **Integer**> col1 = **new TableColumn**<>("ID");

**TableColumn**<**Knihy**, **String**> col2 = **new TableColumn**<>("Název");

**TableColumn**<**Knihy**, **Integer**> col3 = **new TableColumn**<>("Rok vydání");

**TableColumn**<**Knihy**, **Integer**> col4 = **new TableColumn**<>("Cena");

col1.setCellValueFactory(**new PropertyValueFactory**<>("ID\_k"));

col2.setCellValueFactory(**new PropertyValueFactory**<>("nazev"));

col3.setCellValueFactory(**new PropertyValueFactory**<>("rok\_vydani"));

col4.setCellValueFactory(**new PropertyValueFactory**<>("cena"));

tvKnihy.getColumns().setAll(col1, col2, col3, col4);

//Naplnění tabulky

**ObservableList**<**Knihy**> list = **FXCollections**.observableArrayList();

try {

**Statement** stmt = con.createStatement();

**ResultSet** resSet = stmt.executeQuery("SELECT \* FROM knihy");

while(resSet.next()) {

**Knihy** kniha = **new Knihy**();

kniha.setID\_k(resSet.getInt("ID\_k"));

kniha.setNazev(resSet.getString("nazev"));

kniha.setRok\_vydani(resSet.getInt("rok\_vydani"));

kniha.setCena(resSet.getInt("cena"));

list.add(kniha);

}

}catch (**SQLException** e) {

e.printStackTrace();

}

//Přiřazení dat do tabulky

tvKnihy.setItems(list);

# 30. Třída FileChooser – charakteristika a použití, základní vlastnosti a metody (JavaFX)

* Vlastnosti
* Použití

## Vlastnosti

FileChooser poskytuje uživatelské rozhraní prostřednictvím dialogového okna, které umožňuje uživatelům vybírat soubory z jejich systému souborů. To poskytuje jednoduchý a přehledný způsob, jak pracovat se soubory.

**FileChooser** fileChooser = **new FileChooser**();

Umožňuje definovat filtry souborů pomocí metody getExtensionFilters(). Tyto filtry umožňují omezit typy souborů, které se zobrazí v dialogovém okně. Například můžete filtrovat pouze obrázky typu JPG nebo PNG.

fileChooser.getExtensionFilters().addAll(

**new FileChooser**.ExtensionFilter("JPG soubory", "\*.jpg")

,**new FileChooser**.ExtensionFilter("PNG soubory", "\*.png")

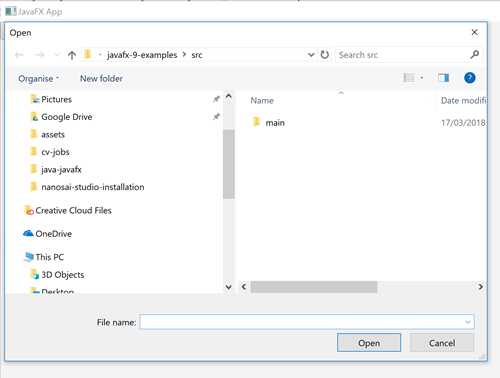
,**new FileChooser**.ExtensionFilter("Všechny soubory", "\*.\*")

);

FileChooser může obsahovat vlastní tlačítka pro rychlý přístup k běžným složkám nebo akcím. Tato tlačítka mohou uživatelům usnadnit navigaci ve svém systému souborů nebo provádění běžných úloh. Přidání je možné pomocí metody setCustomButtons();

## Použití

Nejběžnějším použitím třídy FileChooser je umožnit uživatelům vybírat soubory. To může být užitečné v aplikacích, které pracují se soubory, jako jsou editory obrázků, textové editory, apod.

**File** selectedFile = fileChooser.showOpenDialog(**new Stage**());

Vypsání vybraného souboru:

if(selectedFile != **null**) {

**System**.**out**.println( "Vybraný soubor: " +

selectedFile.getAbsolutePath());

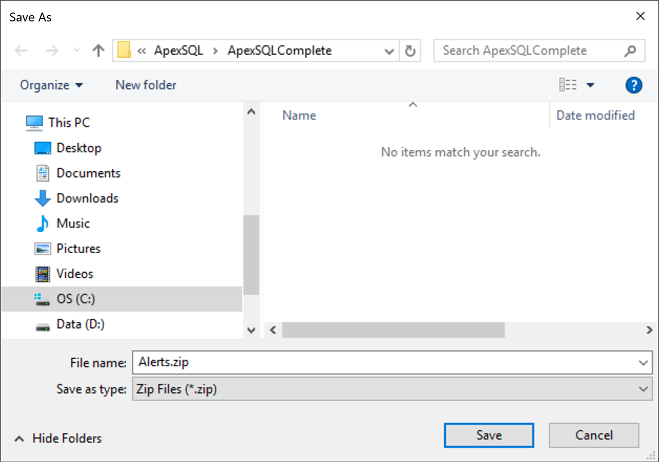
}

else {

**System**.**out**.println("Není vybrán žádný soubor.");

}

FileChooser se také používá k ukládání souborů (vytvoření nového souboru). Například, když uživatel chce uložit dokument do určité složky, může použít FileChooser k výběru umístění a zadání jména souboru.

**File** selectedFile = fileChooser.showSaveDialog(**new Stage**());

Můžeme také nastavit výchozí adresář, který se zobrazí v dialogovém okně. To může být užitečné pro předdefinování umístění, kde se uživatel očekává, že budou uloženy nebo otevřeny soubory.

fileChooser.setInitialDirectory(**File** directory)