

# Faiglovy hloupé otázky

*A hloupé odpovědi doplnil Jakub Mareda a další studenti FEL. Správnost nezaručena, jedovaté komentáře jsou zdarma jako bonus. Najdete-li chybu, neostýchejte se na ni upozornit komentářem. Necháte-li chybu být, může na ni někdo (třeba já :D) doplatit při testu!*

## Úvod:

Dokument vznikl z Word dokumentu, kde byly sepsány otázky k závěrečnému předmětu PR2. Jednou v neděli večer jsem totiž propad' dojmu, že ráno příšeme zápočťák, a tak jsem prošel všechny otázky, co mi kamarádka poslala pár dní předtím a doplnil odpovědi. Skončil jsem v šest ráno, zápočťák se psal až o týden později tak jsem to aspoň uploadnul. Trošku se nám to ale rozrostlo, takže vytvářím odstavec obsah. Budu sem průběžně doplňovat otázky a odpovědi z všech možných testů z programování.

## Používání tohoto dokumentu:

Chcete-li dokument pouze číst, doporučuji si v menu View - Mode přepnout na Viewing. Nestane se vám pak omylem, že někam připíšete nějaký znak. Stránka se pak mnohem méně seká.

Chcete-li dokument upravovat a vylepšovat, vaši spolužáci vám budou velmi vděční. Snažte se prosím držet základních pravidel:

- Editace není diskuse, pokud vidíte nesrovnalost, ale nejste si jistí, použijte funkci komentářů. Stačí označit text, poté kliknout na kolečko s obdélníčkem, který se objeví vpravo.
- Dbejte pravidel českého, popř. slovenského a anglického jazyka. Jiné jazyky v tomto dokumentu nejsou žádoucí.
- Do dokumentu jsem přidal plug-in Code Pretty na formátování kódu. Najdete ho v menu Add-ons - Code Pretty
- Text pod odpovědí se momentálně dělá pomocí Shift+Enter.

**Obsah:**

1. [Předmluva](#)
2. [Závěrečný Java test předmětu Programování 2](#)
3. [Zkouškový test C předmětu Programování 2](#)
4. [Zkouškový test z C předmětu PRP 2016Z](#)
5. [Test v semestru předmětu Procedurální programování](#)

# Předmluva

## *Desatero přikázání Faiglových*

1. *Já, Faigl, jsem tvůj Bůh, který tě vyvedl z windowské země, z domu Microsoftu. Nebudeš mít jiné bohy vedle mne.*
2. *Nevytvoříš si IDE ani jakékoliv zpodobení VIMu, toho, co je v reklamách, ani toho, co vám doporučí na RPH. Nebudeš se jím klanět, ani v nich psát svůj kód, neboť já, Faigl, tvůj Bůh, jsem Bůh žárlivý, který s trestem navštěvuje vinu cvičících na studentech i na třetí a čtvrté generaci těch, kdo mě nenávidí, ale prokazují milosrdenské tisícům těch, kdo mne milují a zachovávají příkazový řádek.*
3. *Nebudeš brát jméno Faigla, svého Boha, nadarmo, protože Faigl automagicky nenechá bez trestu toho, kdo bere jeho jméno nadarmo.*
4. *Pamatuj na sobotní den, abys ho posvětil odevzdáním úkolu. Šest dní budeš psát kód a dělat všechnu svou práci, ale sedmý den je sobota, patřící Faiglovi, tvému Bohu. Nebudeš dělat žádnou práci ty ani tvá dcera, tvůj syn ani tvá dcerka, tvé zvíře ani tvůj příchozí, který je ve tvých branách, protože šest dní Faigl tvořil knihovny a referenční řešení v upload systému a všechny své kontroly valgrindu, a sedmý den odpočinul. Proto Faigl požehnal sobotní den (v PST) a posvětil ho.*
5. *Cti bash a VIM, aby se prodloužily tvé dny na FELu, které ti dává Faigl, tvůj Bůh.*
6. *Neshodíš upload system.*
7. *Nepoužiješ jiného jazyka mimo C.*
8. *Neodevzdáš kód bližního svého.*
9. *Neposkytneš zadání svého testu bližnímu svému.*
10. *Nebudeš dychtit po programátorských schopnostech bližního svého. Nebudeš dychtit po známkách bližního svého, ani po jeho referencích ani po jeho stipendiu, vůbec po ničem, co patří tvému bližnímu.*

# Závěrečný Java test z jazyka Java

## 1. Čím se vyznačuje zapouzdření? Encapsulation

Jako zapouzdřenou vlastnost označujeme takovou vlastnost, ke které může přistupovat pouze objekt, kterému patří, zároveň však poskytuje rozhraní (getter a/nebo setter) umožňující k vlastnosti zvenčí přistupovat. Výhoda zapouzdření spočívá v tom, že rozhraní poskytované objektem může obsahovat dodatečnou logiku, která zabrání chybám – například před změnou vlastnosti původní vlastnost řádně „zlikviduje“, nebo před nastavením vlastnosti na novou hodnotu ověří, že tato hodnota je přípustná.

Příklad: chci nastavit proměnnou „poziceNaObrazovceX,“ když ní přistupuji pomocí setteru, mohu ověřit, že je v rozsahu 0-1024 a kdyby někdo zadal “-10,” mohu na to inteligentně reagovat.

## 2. Co rozumíte pod pojmem čitelnost programu?

Prvky čitelnosti jsou:

- Komentáře, vysvětlující účel funkcí, metod, proměnných, vlastností a větších bloků\*
- Správné odsazení kódu tak, že vnořený kód je více napravo. V praxi tedy obsah každého bloku odsadíme relativně vůči nadřazenému bloku.
- Smysluplné názvy proměnných, funkcí, vlastností, metod, namespaců, souborů... prostě všechno.
- Rozdělení programu na menší celky. Pětitisíci řádkový .c soubor je trestný čin.

\* Blokem se v Javě nebo C myslí de-facto část kódu umístěná mezi složenými závorkami. (Např. v případě podmínky if).

## 3. Co je to bytecode a k čemu se používá?

Bytecode, přenositelný kód, je soubor instrukcí určený softwarovému překladači (interpretu). Právě fakt, že je bytecode prováděn softwarovým interpretorem z něj dělá kód nezávislý na hardwaru, tedy kód přenositelný. Pro každý typ hardwaru je pak nutná implementace interpretu, který na daném hardwaru bude schopen bytekód provést jako program. Bytekód používá například Java, C# a Flash.

Příklad z praxe: Java může běžet na PC, Androidu, ale třeba i na LEGO NXT. Všechny tyto platformy mají svůj interpreter bytekodu Javy, nazvaný JVM, Java Virtual machine. Je jednodušší napsat interpreter bytekodu pro další platformu, než napsat komplilátor tohoto kódu do assembleru.

## 4. Co to je přetížení jména funkce/metody?

Za přetíženou metodu označujeme metodu, pro jejíž jedno jméno existuje více implementací, mezi nimiž se vybírá na základě vstupních parametrů.

## 5. Jaké znáte nelineární spojové seznamy?

Strom a binarní halda.

Trees and graphs.

## 6. Co je to hashovací funkce?

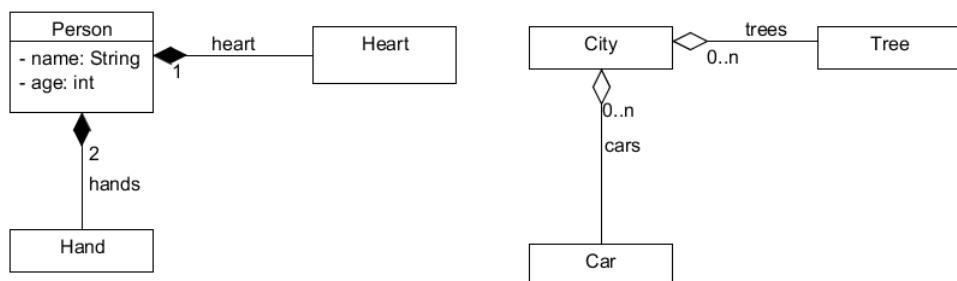
Hashovací funkce je implementace jednosměrné šifry. Zašifrovaná data již nelze rozšifrovat,

platí ale, že pro stejná data funkce vždy vrátí stejný výsledek – lze tedy zjistit, zda jsou určité bloky dat identické, aniž by bylo třeba oba celé znát. Používá se zejména při ochraně hesel, rovněž ale při indexování velkého množství dat (např. ikonek v OS, animací ve hrách).

HashMap v Javě rovněž používá hash pro indexování prvků. Hlavní nevýhodou tohoto přístupu je, že není vyloučeno, že hashovací funkce vrátí stejné výsledky pro různé bloky dat.

## 7. Jaký je rozdíl mezi relací agregace a kompozice?

V obou případech se jedná o has-a relaci. Například - šachovnice obsahuje figurky nebo viz následující diagram:



Silně záleží na kontextu aplikace, která tento vztah využívá.

V případě **kompozice** existence vnitřního objektu nemá smysl bez objektu vnějšího. Třeba pokud by došlo k zániku šachovnice, pak už je šachová figurka zbytečná.

Naproti tomu **agregace** znamená volnější vztah. Pokud by zaniklo město, tak to neznamená, že všechny automobily musí také nutně zaniknout, jelikož si na ně může držet referenci třeba firma, majitel atp.

Viz <http://www.cs.vsb.cz/benes/vyuka/upr/texty/objekty/ch01s02s03.html>

## 8. Kdy použijete vazbu is-a a has-a, příklad?

Vazba **is-a** představuje taxonomický vztah. Například drozd je pták, String je Object.

Vazbu **has-a** pak použijeme pro hierarchický vztah kupříkladu drozd má křídla, String má vlastnost length (int).

## 9. Vyjmenujte základní vlastnosti objektů a tříd objektově orientovaného programu.

### Třída:

Představuje abstraktní definici datové struktury. Popisuje, jaké vlastnosti daná struktura má a jakým způsobem na ní lze operovat (v Javě říkáme, že definuje metody).

Třída může vycházet z jiné, nadřazené třídy a upřesňovat originální definici. Třída rovněž může implementovat návrh, v Javě zvaný interface, a poskytovat API pro určitý seznam operací předem navržených v interface. Třída sama nenesе žádná data. (pomíjím teď statické vlastnosti a metody v Javě, to s obecným OOP zas tak nesouvisí)

### Objekt:

Objekt je konkrétní datová reprezentace třídy. Objekt je představován blokem dat a označen typem (názvem třídy), kterou reprezentuje. Na objektu můžeme operovat pomocí metod definovaných třídou, podle které byl vytvořen.

## **10. Jak se v Javě vytvářejí objekty.**

Je třeba použít konstruktor, metodu definovanou třídou, dle které chceme objekt vytvořit.

Syntaxe volání konstruktoru je:

```
new ClassName();
```

Konstruktor může akceptovat parametry stejně jako jakákoli jiná metoda.

## **11. Jaké základní metody poskytuje třída Object.**

```
String toString()
```

Převede objekt na string. Metodu lze přepsat v podřazených třídách. Výchozí implementace vypíše jméno třídy objektu a za ním hashCode objektu.

```
final Class<T> getClass()
```

Vrátí objekt typu Class který reprezentuje třídu, podle které byl tento objekt vytvořen.

Metodu nelze přepsat.

```
int hashCode()
```

Vrátí číslo reprezentující hash tohoto objektu. Používá se např v HashMap.

```
boolean equals(Object o)
```

Vrátí true, pokud objekt o odpovídá tomuto objektu. Co znamená, že jeden objekt odpovídá druhému, záleží na implementaci. Výchozí implementace za identitu považuje pouze identitu instancí (tzn. jedná se o ten stejný objekt).

```
Object clone() throws CloneNotSupportedException;
```

Vrátí novou instanci objektu která odpovídá této instanci. Pokud klonování nemá smysl, vyhodí chybu.

```
wait(), notify(), notifyAll()
```

Všechny tři metody slouží k práci s více vlákny. Metoda wait() pozastaví aktuální vlákno, dokud není na stejném objektu (z jiného vlákna) zavolána metoda notify(). Je-li aktuální vlákno přerušeno metodou interrupt() třídy Thread, vyhodí metoda wait() InterruptedException.

## **12. Proč třídy organizujeme do balíků?**

Balík je obdobou namespace v C++ nebo C#. Umožňuje třídy se stejným názvem (ale zcela rozdílnou implementací) roztržit do různých sekcí, a tak zabírá jinak nevyhnutelnému konfliktu názvů.

## **13. Rozdíl mezi binárním a textovým souborem a jejich zpracováním.**

Binární soubor zpravidla parsujeme byte po bytu. Obvykle se skládá z bloků dat s definovanou délkou. Textový soubor je třeba nejdříve převést na text v závislosti na použitém kódování (kterých jsou mraky) a poté jej parsujeme znak po znaku. ASCII textové soubory lze rovněž rovnou číst byte po bytu, jelikož v ASCII neexistují vícebytové znaky.

## **14. Rozdíl mezi přímým a sekvenčním přístupem k souborům.**

Přímý přístup představuje čtení z předem zadaných lokací v souboru (např. Od 100 bytu do 120 bytu). Toto praktikujeme zejména u binárních souborů. Sekvenční přístup spočívá ve čtení souboru od začátku a hledání dat, která chceme. Takhle parsujeme třeba XML, JSON či Java serializované soubory.

## **15. Hlavní synchronizační problémy u vícevláknových aplikací.**

### **1. Zdroj/konzument**

Problém dvou vláken, kdy jedno vlákno data produkuje a zařazuje do fronty, druhé data z fronty odebírá a zpracovává. Je třeba zabránit cpaní dat do plné fronty a tahání dat z prázdné fronty.

### **2. Čtení/zápis**

Problém, kdy mutexem rezervujeme proměnnou pro jedno vlákno, i když obě vlákna se z proměnné snaží pouze číst. Je ale pořád třeba zabránit kombinaci zápis+čtení.

### **3. Obědvající filozofové (deadlock)**

Analogie mě moc nebaví. Problém spočívá ve zdrojích sdílených vlákny. Ve chvíli kdy každé vlákno (ze dvou) disponuje polovinou potřebných zdrojů a čeká na tu druhou, program nikdy neskončí. Řeší se zpravidla nadřazeným algoritmem pro rozdělování zdrojů.

## **16. Jaké způsoby navazování komunikace znáte?**

Sériová komunikace – jedná se o obousměrnou komunikaci po dvou kanálech, z nichž každý je jednosměrný. Komunikace je asynchronní (kanály na sobě nejsou závislé).

TCP socketová komunikace – komunikace se navazuje mezi zařízeními pomocí síťové IP adresy a portu. Je o dost komplexnější než sériová, disponuje automatickou opravou chyb (chybně přijatá data nemusíte řešit – buď přijdou správná data, nebo se socket odpojil a nepřijde nic).

UDP komunikace – od socketové se liší tím, že nemá zpětnou vazbu ani opravu chyb. To z ní dělá komunikaci méně spolehlivou, ale nesrovnatelně rychlejší. Je rovněž méně bezpečná.

Pipe – komunikace v rámci jednoho zařízení poskytovaná operačním systémem. Umožňuje posílat data mezi běžícími programy. Je na ní založen příkazový řádek.

## **17. 3 základní prvky grafického rozhraní.**

*Neplást s MVC. Faigla jsem se ptal, ptá se opravdu na komponenty, kontejnery atd.*

Komponenty, tedy jednotlivé grafické prvky sloužící k předávání informací a to oběma směry. Může to být text, tlačítko, progress bar a podobně. Konkrétně JLabel, JButton, JMenuItem, JList...

Kontejnery, do kterých komponenty vkládáme. Kontejnery zpravidla žádné informace nepředávají ani nezobrazují zajímavé grafické prvky. Známe okno, tedy JFrame a JDialog, a JPanel sloužící obecně k seskupování prvků.

Layout managery, správce rozvržení, které určují, jak budou komponenty v kontejnerech uspořádány. Třeba BorderLayout nebo GridLayout.

Eventy slouží ke spouštění fragmentů programu v reakci na akci uživatele, jako je kliknutí na tlačítko. Eventy se spouštějí ve speciálním threadu swingu, nikdy tedy nemohou probíhat dva najednou. Eventy lze přiřadit jak komponentům tak kontejnerům. ActionListener je velmi generický typ eventu, můžeme ale také použít MouseListener pro akce myši, nebo WindowListener, který zachytává základní akce okna.

Zdroj: Faiglovy slidy, strana 27.

## **18. Co to je modální dialogové okno?**

Jedná se o okno, jemuž je přiřazeno nadřazené okno. Modální okno nepovolí uživateli interakci s nadřazeným oknem dokud uživatel neproveze jednu z akcí, které mu nabízí okno

modální. Pokud se uživatel pokusí aktivovat nadřazené okno, modální okno vztekle zabliká a vydá agresivní zvuk.

## 19. Co jsou generické typy?

Generickým typem nazýváme typ, jež nemá konkrétní velikost či (v případě OOP jazyků) třídu. Tento přístup umožňuje napsat metodu, jež obecně pracuje s různými typy objektů/datových typů aniž bychom se museli omezovat na jeden konkrétní.

Javovská syntaxe pro generické typy je:

```
class Trida<T extends JinaTrida> {...}
```

kde T může být jakýkoliv objekt který je podtypem třídy JinaTrida. Pokud klíčové slovo extends není uvedeno, předpokládá se Object. Rovněž ojedinělá metoda (jak statická, tak normální) může používat generické typy:

```
public [static] <T> T udelejNeco (T objekt);
```

Tato metoda něco provede s objektem typu T a pak vrátí nějaký objekt typu T.

V Javě jsou generické typy pouze syntaktická pomůcka, do zkompilovaného bytekódu se nepromítou. Použití generických typů lze vždy nahradit typem Object, nebo dokonce specifickým typem.

## 20. Co znamená statické a dynamické stanovení typu u proměnných?

Statické stanovení typu - při překladu (Java, C, C++, C#)

Dynamické - za běhu programu (Python, Matlab)

## 21. Jaké znáte u Javy řídící struktury?

Řídící struktury jsou příkazy jež ovládají běh programu – tedy if, else, while, do while, for, switch, try ... catch, break, continue. Přesto, že klíčové slovo goto v Javě momentálně nic nedělá, lze také definovat labely (se kterými se goto obvykle pojí). Labely lze používat k přerušení vnořených seznamů, takto:

```
String[][] array2d = ...;
cyklus:
for(String[] array1d: array2d) {
    for(String text: array1d) {
        if(...) {
            //Zrusí vnejsí i vnitřní cyklus
            break cyklus;
        }
    }
}
```

## 22. Jak je reprezentován neceločíselný typ v Javě?

**Aleš Hološka:** Aproximovaně do pevného počtu bitů. Je uložen jako trojice znaménko 1bit, mantisa 23 bitů, exponent 8bitů.

**Jakub Mareda:** Pomocí primitivních typů float a double a třídy BigDecimal.

## 23. Je efektivnější rekurze nebo iterace?

Iterace. Za každou rekurzi je znova vytvořit paměťový prostor (v Javě navíc přidat položku do

zásobníku, angl. stack) pro prováděnou funkci. Iterační algoritmus operuje pořád ve stejné funkci a tudíž maximálně alokuje paměť, ne však prostor pro stejnou funkci. Při rekurzi také hrozí (v Javě, C++) chyba zvaná StackOverflow, tedy přeplnění (přetečení) zásobníku.

Halda je seznam funkcí v pořadí, v jakém volala jedna druhou. Funkce která volá sama sebe logicky přidá do haldy mnoho položek.

#### **24. Co je to vícenásobná dědičnost?**

Možnost implementace třídy, jež vychází (`extends`) z více jiných tříd. Java nic takového nemá, je třeba si neplést interface a třídu. Vícenásobná dědičnost je opravdu komplexní a cílem Javy je, aby se jí naučila každá opice. V C++ vícenásobná dědičnost existuje, je ale záhadno si dobré rozmyslet, kdy ji použít.

#### **25. Jak popíšete algoritmus z hlediska jeho použití?**

Algoritmus je seznam rozhodovacích kroků závislých na aktuálním stavu. Algoritmu předáváme určitý stav a algoritmus nám po určitém konečném počtu operací odpoví jiným stavem (výjimkou mohou být heuristické algoritmy, které se snaží řešení pouze approximovat a k ideálnímu řešení by tak teoreticky došly po nekonečném počtu kroků).

#### **26. Jaký je rozdíl mezi objektově orientovanou analýzou a návrhem?**

OO analýza se zabývá modelováním, rozbořem a specifikací problému -> abstrakce reálného světa.

OO návrh se zabývá řešením problému -> přidává softwarovou abstrakci

#### **27. Které atributy třídy je vhodné zařadit do přístupné kategorie public a proč?**

*Nejdřív tomu říká proměnný, pak atributy...*

Jako public můžeme s klidem označit všechny final vlastnosti, pokud ovšem přístup zvenčí má smysl. Tato praxe se používá zejména pro konstanty:

`public static final int DEFAULT_SIZE = 5;`

#### **28. Co je nutné specifikovat při spuštění konkrétní třídy v javě?**

*WTF. Třídy se nespouštěj. Ani objekty se nespouštěj. A třídy nejsou konkrétní, v tom je ten vtip, o tom je celý OOP.*

Je třeba definovat statickou funkci:

`public static void main(String[] args)`

Rovněž lze použít varargs syntaxi:

`public static void main(String... args)`

Ale v tomhle případě v tom není moc rozdíl.

#### **29. Můžeme definovat více konstruktorů jedné třídy? Pokud ano, proč to děláme?**

Konstruktor je akorát speciální metoda, lze jej tedy přetěžovat dle libosti. Důvodem může být například konverze. Naše třída může být schopná se inicializovat ze `Stringu` i z `Integeru`, pokaždé trochu jiným způsobem. Rovněž tak můžeme implementovat poněkud otravným způsobem tzn. Nepovinné, či výchozí argumenty:

```
MojeTrida(int cislo, boolean stav) {...}
MojeTrida(int cislo) {
    this(cislo, true);
}
```

**30. Může mít referenční proměnná výčtového typu jinou hodnotu než validní hodnotu výčtu, pokud ano jakou?**

*Opět snaha z vás udělat blbce. Nelekejte se, výčtovým typem se myslí Enum.*

Enum je třída, proměnná „výčtového typu“ je tedy Object a může místo na validní hodnotu ukazovat na null.

**31. Jak lze vytvořit novou výjimku?**

```
public class MyException extends Exception {...}
```

Kromě z Exception lze dědit také z Throwable, Error, RuntimeException. Tyto tři zmíněné lze zachytit pomocí try ... catch, ale není nutné je deklarovat pomocí throws. Throwable - nemusí se zachytavat  
Exception - musí se zachytavat  
RuntimeException - nemusí se zachytavat

**32. Co musí úloha splňovat, aby mělo smysl uvažovat o vícevláknové architektuře aplikace?**

Je možné provádět paralelně více relativně náročných úloh na více procesorovém systému (kompilace například).

NEBO

Aplikace provádí blokující IO operace (čtení velkého souboru, síťová komunikace), ale zároveň má GUI, které musí zůstat responzivní.

NEBO

Aplikace něco kontroluje na pozadí (daemon thread) v pravidelných intervalech, zatímco na popředí s ní uživatel normálně interaguje.

**33. Lze zabránit vyvolání RuntimeException, pokud ano jak?**

*Tahle otázka je asi blbě formulovaná. Podle mně se ptá, jestli lze zabránit, aby program následkem RuntimeException spadl.*

Jakákoliv podtřída Throwable lze „chytit“ pomocí struktury try ... catch. Mnoho lidí zastává názor, že v Javě jsou kontrolované (checked, musí se deklarovat přes throw a chytit) silně nadužívané.

**34. Co je to serverový socket?**

Obšlehnuto z Oraclu:

A server socket waits for requests to come in over the network. It performs some operation based on that request, and then possibly returns a result to the requester

**35. Co je to JCF (java collection framework)?**

Jedná se o ucelený komplex tříd a interfaců reprezentujících datové struktury, například HashMap (implementuje Map), Iterator...

### **36. Popište model-view-controller.**

Jedná se o prvky anglicky nazvané Model, View a Controller.

Model představuje logickou strukturu programu, tu část která provádí výpočty.

View je grafický výstup, na který se promítají výsledky práce programu, které jsou provedené v modelové části.

Controller je skupina funkcí, které zaznamenávají uživatelsou interakci s grafickým rozhraním (view) a předává informace do modelové části. Například pokud uživatel klikne na tlačítko, controller o tom informuje model a model rozhodne, že by tlačítko mělo změnit barvu.

### **37. Jak se v javě zapisují jména tříd?**

```
class Jmeno /*implementace*/ {
```

### **38. Jak se v javě vytvoří pole polí?**

```
int[][] polepoli = new int[20][30];
```

### **39. Co jsou slabě typované jazyky.**

Představte si, že byste v Java všechny proměnné a návratové hodnoty definovali jako Object.

Jazyky jejichž proměnné nemají definovaný datový typ, který mohou obsahovat. Tyto jazyky často obsahují velkou zásobu implicitních konverzí mezi typy, takže převodu mezi textovým řetězcem a číslem si ani nevšimnete. Jsou jimi např. PHP a Javascript.

### **40. Jak se v javě deklaruje pole hodnot?**

```
int[] pole = new int[20];
```

### **41. Jak popíšete algoritmus před jeho implementací?**

Vývojový diagram, UML diagram, strukturogram, pseudokód

### **42. Hlavní důvod kolizí při hašování, jak je řešíte?**

Důvod kolizí při hashování (hlavní a myslím jediný) je, že výstup hashovací funkce má jasně definovanou velikost, vstup nikoli. Z lineární algebry víme, že se nám nezbytně musí více bodů v originálním prostoru vstupu promítnout do jednoho bodu ve výstupu, tedy že více bloků dat vygeneruje stejný hash.

Řešit lze problém buď zvětšením množiny výstupů nebo použitím více hashovacích method.

Za identitu pak považujeme pouze pokud jsou oba hashe z dvou různých metod stejné.

Pozn: Vyloženě vyřešit se to nedá, jen snížit pravděpodobnost, že k tomu dojde.

### **43. Z jakých částí se skládá odvozená třída (subclass)?**

Název odvozené třídy a název nadřazené třídy. Syntaxe javy:

```
public class Odvozena extends Nadrazena { ... }
```

### **44. K čemu slouží operátor this?**

Operátor `this` je referencí na objekt, v jehož metodě se právě nacházíme. Operátor `this` lze použít pouze v dynamických metodách nebo statických metodách dynamických tříd. V případě statických metod dynamických tříd je třeba před `this` uvést jméno třídy, ke které

chceme přistupovat:

```
public class A {  
    public void neconecko() {}  
    private class B {  
        public void Metoda() {  
            A.this.neconecko();  
        }  
    }  
}
```

#### 45. Jaké znáte typy výjimek?

Kontrolovaná (checked), vyžaduje throws deklaraci a je nutné funkci, jež ji deklaruje, obklopit strukturou try ... catch.

Nekontrolovaná (unchecked) výše uvedené nevyžaduje, pokud k ní dojde, lze ji zachytit.

#### 46. Jak lze standardní vstup a výstup použít ke komunikaci mezi procesy?

Pomocí pipes: procesSVystupem|procesCoToZpracuje

Např: ls|grep "kokodak" vypíše seznam souborů v akt. adresáři, co obsahuje v názvu kokodak

#### 47. Rozdíl mezi datovými kolekcemi list a set.

**List** - seznam - co do něj přidám, to tam je. Z listu lze také získat prvek pomocí **indexování**.

**Set** - množina - skutečně reprezentuje množinu z matematiky. Obsahuje každý prvek maximálně jednou a alespoň jeden null. Není zaručené pořadí prvků a nelze indexovat.

Můžeme se ale zeptat, jestli obsahuje daný prvek, případně ho přidat nebo odebrat...

#### 48. Co je double dispatch a k čemu se používá?

Viz [Článek o vzoru Visitor](#). Pro double dispatch srolujte níže, ale doporučuji přečíst celé.

#### 49. Relační operátory v javě.

Jedná se o:

==	identita
!=	opak identity, nerovnost
>	větší než
>=	větší nebo rovno
<	menší než
<=	menší nebo rovno

#### 50. Jak se v javě zapisují jména funkcí/metod?

Funkce v javě neexistují. Nejblíže funkcím jsou statické metody. Syntaxe:

```
public class Trida {  
    public void nazevMetody() {...}  
}
```

#### 51. Co znamená, pokud je v javě třída definována jako final?

Nelze vytvořit odvozenou třídu (subclass, pomocí extends) třídy označené jako final.

Touhle nepříjemnou vlastností se pyšní např `String`, `Integer` a většina dalších immutable typů. Důvodem je, že případná podtřída už by immutable být nemusela.

#### 52. Co je java virtual machine (JVM).

Jedná se o interpreter java bytekódu, který za chodu java bytekód kompliluje, optimalizuje a hlavně převádí na instrukce.

#### 53. Jak vypadá reprezentace reálných čísel v počítači?

Je uložen jako trojice znaménko 1bit, mantisa 23 bitů, exponent 8bitů.

#### 54. Co je polymorfismus v objektově orientovaném návrhu?

Polymorfismus, neboli dědičnost je schopnost tříd vycházet z jiných tříd a vytvářet tak taxonomický strom. V Javě je na vrcholu tohoto stromu `Object`.

#### 55. Jak se používá operátor super?

Operátor super umožňuje přistupovat k veřejným metodám a vlastnostem nadřazené třídy (superclass). Toto má smysl zejména pokud požadovaná metoda byla přepsaná (override).

Například:

```
class A {  
    public void bla() { ... }  
}  
class B extends A {  
    @Override  
    public void bla() {  
        ...  
        super.bla();  
    }  
}
```

V konstruktoru lze (a často je třeba) volat konstruktor nadřazené třídy. Dělá se to takhle:

```
class A {  
    public A(boolean stav) { ... }  
}  
class B extends A {  
    public B() {  
        super(true);  
    }  
}
```

Volání nadřazeného konstruktoru musí být první příkaz v konstruktoru. Tohle debilní pravidlo se dá občas obejít, třeba takhle:

```
super(nejakáSTATICKAmetodaCoVracíBoolean() || true)
```

#### 56. Kompilace v javě.

Pomocí programu `javac` (napsaného mimochodem taky v Javě). Vstup je `.java`, výstup je `.class`, tedy (víceméně) na platformě nezávislý bytecode pro JVM.

**57. Jak realizujeme serializaci v javě.**

```
import java.io.Serializable;  
  
public class Class implements Serializable {}
```

**58. Jaké výjimky je nutné ošetřovat a jak?**

Všechny podtypy `Exception` kromě podtypů `RuntimeException` je třeba ošetřit strukturou `try ... catch`. Chybám typu `RuntimeException` by mělo jít zabránit správnou implementací programu (kontrola na null, kontrola správného stavu nějaké třídy, tak aby nevyhodila `IllegalStateException`). Nicméně i `RuntimeException` lze odchytit pomocí `try ... catch`. Ve skutečnosti lze odchytit všechny typy výjimek a chyb jejichž nejvyšším supertypem je `Throwable`. Nicméně výjimky typu `Error` jsou výjimky, které zpravidla zanechají program v ne definovaném / nefunkčním stavu a zachytávat by se neměly. Pokud ano, pak by po jejich zachycení mělo následovat uložení do logu a ukončení programu.

**59. Co je to monitor, jak se používá při procesu synchronizace vláken?**

Monitor je objekt, který nám buďto uzavře blok příkazů (`synchronized (Object obj) {...}`) a nebo celou třídu, kde sama třída je monitorem, potom jsou v ní tzv. `synchronized` metody, které právě uzamykají tuto třídu. A jinak, monitor je k tomu aby za sebou zamknul a odemknul zase až se vše dokončí -> ochrana proti racecondition ve více vláknových aplikacích.

**60. Jaké modely vícevláknových aplikací znáte?**

Boss-worker, Pipeline, Threadpool, Peer



**61. Co to je iterátor, jak se používá?**

Iterátor je objekt, jenž představuje pozici v nějakém seznamu dat (může být spojový, pole, hashmap...). Iterátor umožňuje bezpečně procházet seznamem dat.

**62. Jak v javě zapisujeme textové řetězce?**

```
"text \"text v uvozovkach\" este nejaky text"
```

**63. Jaké znáte celočíselné typy v javě?**

`byte` (1byte, signed),  
`short` (2byty, signed),  
`int` (4byty, signed),  
`long` (8bytů, signed),  
`char` (2byty, unsigned).

**64. Jak lze předčasně opustit konání dvou vnořených for cyklů?**

Pokud je tohle potřeba, je třeba program znova opustit. Standardní smysluplný postup je znehodnotit podmínu vnějšího cyklu. Nicméně očekávaná odpověď na tuto otázku bude tato prasárna:

```

vnejsi_cyklus:
for(String[] s: stringArrays) {
    for(String string:s) {
        if(...) {
            break vnejsi_cyklus;
        }
    }
}

```

#### **65. Co je výraz v javě?**

Výraz (expression) sestává z proměnných, konstant a návratových hodnot metod (resp jejich volání) spojených operátory:

```

int cislo = Integer.MAX_VALUE-500;
System.out.println("Cislo: "+cislo);

```

#### **66. Ukazatele kvality algoritmu.**

Já bych to rozdělil na 3 základní kritéria:

- a) Časová složitost
- b) Paměťová náročnost
- c) Funkčnost - jestli to dělá to, co má

#### **67. Jak přistupujete k větám (datovým záznamům) souboru při nepřímém vyhledávání?**

Nepřímé vyhledávaní znamená sekvenčne -> prostě to musíš projet od začátku až do doby nez to najdeš. Stejná filosofie jako LinkedList.

#### **68. Co jsou virtuální metody objektově orientovaného programování?**

Metody, které může dědic přetížit. Původní definici lze volat pomocí klíčového slovíčka `super.methodName()`. Metody označené final nelze přetěžovat.

#### **69. Kategorie přístupu pro datové položky a metody třídy objektově orientovaného programování.**

public, private, protected, package private ([default](#))...

#### **70. Jak spolu souvisí efektivita a znovupoužitelnost při optimalizaci objektového návrhu?**

*Nemůžes mit všechno :D sam to říkal.*

Pro maximální efektivitu se program specifikuje pro speciální případy a tím pádem se stává méně použitelným při jiném použití. Zatímco pro maximální znovupoužitelnost maximalizujeme obecnost programu a tím pádem ztrácíme na efektivitě. - program bude buď extra efektivní a nebo extra použitelněj znova ale nikdy ne obojí.

#### **71. Jak probíhá start programu v javě?**

příkaz `java main.class` – musí obsahovat `psvm()`  
v jar je specialní soubor, který nám určuje jaká `psvm()` se má spustit

#### **72. Jak definujeme konstruktory?**

```

class A {

```

```

public A([parametry zde]) {
    [možnost volání super() nebo this() zde]
    // Pokud je volán konstruktor předka nebo této třídy, tak to musí
    být první příkaz
}
}

```

**73. Jak v javě deklarujeme datové položky třídy tzv. třídní proměnné?**

*Faigl se s terminologií opět nesere. Třídními proměnnými myslí statické vlastnosti.*

Statickou vlastnost, nebo i metodu, definujeme pomocí klíčového slova static.

**74. Jak definujeme že metoda může způsobit výjimku.**

```
void mojeMetoda() throws MojeException { ... }
```

**75. Jak zachytáváme konkrétní výjimku?**

Do hlavičky bloku catch napíšeme, o jaký typ výjimky máme zájem. Lze zachytávat i více typů odděleně:

```

try { ... }
catch(FirstException ex) { ... }
catch(SecondException ex) { ... }

```

Měli bychom nejprve zachytit konkrétnější výjimky, až poté obecné. Například nejdříve IOException a pak teprve Exception.

Lze také zachytit několik typů výjimek současně (Java 7):

```

try { ... }
catch(FirstException | SecondException ex) {
    System.err.println(ex.getMessage());
}

```

Této syntaxi říkáme multicatch.

**76. Co to je problém souběhu u více-vláknových aplikací?**

K souběhu (angl. race condition) dochází pokud dvě vlákna začnou měnit stejná data v paměti ve stejnou chvíli. Výsledná hodnota paměti je ne definovaná. Souběhu se v Javě předchází synchronized blokem a klíčovým slovem volatile.

**77. Jaké základní operace související s paralelním programováním řeší programovací jazyky s explicitní podporou paralelismu?**

Zámky používaných zdrojů, sdílení paměti, synchronizaci jednotlivých vláken. Komunikaci mezi vlákny.

**78. Popište definici vnitřní třídy, příklad situace užití.**

Vnitřní třída se definuje uvnitř těla jiné třídy, syntaxe je stejná jako u jakékoliv třídy:

```

public class A {
    public class AHelp {
    }
}

```

Vnitřní třída je dynamická (pokud ji neoznačíme klíčovým slovem `static`), v každé instanci vnější třídy se tedy vztahuje ke kontextu dané instance. Příkladem použití jsou pomocné třídy, například privátní `HashMap.Node`, nebo implementace callbacků (implementace `Runnable`, `ActionListener`, `MouseListener` etc.). Každá anonymní třída je rovněž technicky vzato vnitřní třída, kompilatorem pojmenovaná jako `VnejsiTrida$1`.

**79. Vyjmenujte tři základní typy událostí uživatelské interakce se swing aplikací.**

klávesnice (KeyListener), myš (MouseListener/MouseMotionListener), kolečko myši (MouseWheelListener).

**80. Popište událostmi řízené programování a uveďte základní rozdíly v porovnání se sekvenčním programováním.**

Událostmi řízené programování obsahuje Event Listenery, které lze různými způsoby agitovat a donutit je zpracovat náš požadavek.

Sekvenční programování jde od shora dolů a chování ovlivňují řídící prvky

**81. Co je ternární operátor?**

`variable = [logický výraz] ? [příkaz/hodnota pokud pravda] : [příkaz/hodnota pokud ne]`

Ternální operátor je zkratka pro `if( ... ) { ... } else { ... }`. Ternární operátor navíc vrací hodnotu ve vybraném bloku, takže jej lze použít k vybírání hodnot. Typické použití:

```
boolean stateBool = true;  
String stateString = stateBool ? "true" : "false"
```

**82. Jak zjistíme délku pole statické délky.**

`pole.length`

**83. Co popisuje sémantika programovacího jazyka?**

*Na tohle sem musel vzít wikipedii.*

Semantika matematicky popisuje jaké výpočty jsou výsledkem syntakticky platných fragmentů zdrojového kódu.

**84. Hlavní rysy OOP.**

Třídy a objekty. Třídy popisují, jak budou organizována data, objekty jsou praktické provedení toho, co popisuje třída.

Polymorfismus. Třídy mohou vycházet z jiných, nadřazených tříd. Vzniká tak taxonomický strom\*. Podřazené třídy dědí vlastnosti a metody nadřazených.

\* V případě vícenásobné dědičnosti spíš taxonomická pavučina.

Encapsulation  
Inheritance

**85. Rozdíl mezi strukturou zásobník/fronta.**

Fronta se také nazývá FIFO - First In First Out. Je jako tunel, nebo fronta lidí u pokladny. Zásobník je naopak jako díra, a říká se mu LIFO - Last In First Out. Co do díry dáme jako poslední budeme muset jako první vyndat.

**86. Jaká pravidla používáme při definování balíků (packaging)?**

Firmy a společnosti použijí svojí doménu pozpátku, například cz.seznam.email.

Obsahuje-li název pomlčku, nahradíme ji podtržítkem. Shoduje-li se název package s klíčovým slovem, dáme nakonec názvu podtržítko, například com.math.int\_.

**87. Nástroje na řízení překladu zdrojových souborů v javě.**

ant a maven

Když nastavím primitivní vlastnost jako final, a pak v závislosti na její hodnotě vykonávám kód, tento kód se zahodí, když podmínka není splněna již během komplikace. Toto platí pouze pokud jsou splněny podmínky popsané v příspěvku [Compile time constants and variables na Stack Overflow](#).

**88. Jak dosáhneme reentrantní funkce.**

Google říká, že nic jako "reentrantní funkce" v programování nezná. Na druhou stranu reentrance ano.

Disponuje-li funkce/program schopností reentrance, je možné ji spustit dříve, než jiná její instance skončila. Tento pojem se však týká jen funkcí používajících synchronized, zpravidla funkcí rekursivních.

- a) "Reentrantní" úsek programu nesmí uchovávat žádná trvalá (static) nebo globální (v Javě zase static),
- b) úsek programu nesmí měnit sám sebe,
- c) a nesmí spouštět kód, který předchozí dvě podmínky nesplňuje (není reentrantní)  
reentrantní => reentrantní

**89. Co je to komunikace.**

zahájení spojení,  
předání zprávy,  
reakce na zprávu,  
ukončení spojení.

**90. Základní grafické komponenty ze swing.**

---

Přehled základních grafických komponent

JLabel – Zobrazení popisku, bez generování události

JButton – Tlačítko s událostí kliknutí na tlačítko

JTextField – Zadání textu

JPasswordField – Zadání textu (hesla), vložené znaky se zobrazují jako hvězdičky

JList – Seznam položek, možnost vybrat jednu nebo více položek

JComboBox – Rozevírací seznam položek, klepnutím na položku se generuje událost

JCheckBox – Zaškrťávací políčko, prvek je/není vybrán

JRadioButton – Přepínač, výběr z možností

**91. Jaké protokoly použijete při realizaci jednoduché aplikace typu telnet?**

Telnet posílá byty pomocí protokolu TCP. Většina aplikací které umožňují připojení přes telnet podporuje také ANSI protokol.

**92. Co je proces v terminologii operačního systému?**

Proces je spuštěný program ve vyhrazeném prostoru paměti. Jedná se o entitu operačního

systému, která je plánována pro nezávislé provádění.

**93. Budete se snažit svůj program paralelizovat i když máte pouze jeden procesor? Svou odpověď zdůvodněte.**

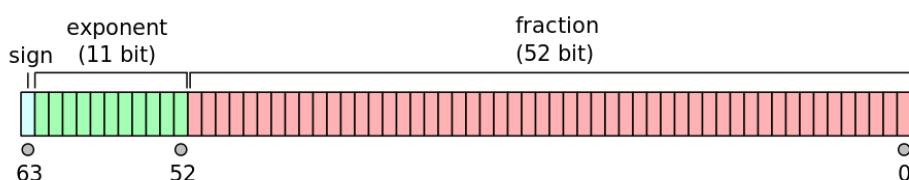
Ano, protože operační systém přepíná sám mezi vlákny a simuluje chování více-procesorového systému. Můžeme tím např. zvýšit responsivity GUI aplikace. Operační systém také sám přepne z vláken, která čekají na IO (čtení z disku například) a uvolní tak prostor pro jiné vlákno - například právě GUI vlákno.

**94. Jak realizujete předání více hodnot jako návratové hodnoty metody (funkce)?**

Vráťím Object, který obsahuje proměnné, které potřebuji

**95. Jak je reprezentován neceločíselný typ double v Javě?**

64-bit IEEE 754



**96. Co znamená, řekneme-li o algoritmu, že je univerzální?**

Algoritmus nemusí znát datový typ, nad kterým pracuje. Například sort bude fungovat s čímkoli, když mu dám komparátor, který ty dvě věci porovná.

**97. Co obsahují třídy v objektově orientovaném systému?**

Statické proměnné, objektové proměnné, statické a objektové metody.

**98. Co jsou to třídy v objektově orientovaném systému?**

Třída je konstrukce, která obsahuje:

- a) statické proměnné a metody - toto jsou parametry třídy samotné
- b) instanční proměnné a metody - každá instance třídy má své vlastní a jsou na sobě nezávislé

Pokud třída není abstraktní, lze vytvořit její instanci

**99. Jak se lze vyhnout problému uváznutí u více-vláknové aplikace.**

Deadlock - dvě nekonečné smyčky (spinlocks) na sebe vzájemně čekají - lze tomu předejít tak, že je bude někdo synchronizovat, buď semafor nebo nějaký thread

Trashing - OS je zahracen tolka procesy/thready, že jediné, co dělá je, že scheduler swapuje a přiděluje strojový čas, ale ve výsledku se nic nevykoná. - Lze tomu předejít threadpoolom, v Javě thread executorem, jemu se buď nastaví explicitně maximální počet vláken, nebo se využije výchozí hodnota, která je obvykle rovna počtu (logických)jader procesoru.

**100. Jaké vlastnosti musí splňovat proudové zpracování dat, aby bylo výhodné použít více vláken?**

Musí být asynchronní a thread safe

Dále nemá třeba cenu zpracovávat HTML document ve více vláknech, neboť si můžeme najít element, např <div>, ale již nevíme, čí je to potomek, ani kolikátý je to potomek atp.

Smysl to dává, pokud třeba máme velké množství homogenních dat, která na sobě nejsou závislá - třeba obrázky 15x15, které máme kvalifikovat na základě nějaké vlastnosti.

### 101. Jak dosáhneme thread-safe funkce?

Objekty, ke kterým může přistupovat více vláken najednou, by měly být zajištěny pomocí slova synchronized.

```
public synchronized void test(String name) {
    for(int i=0;i<10;i++) {
        SOP.print(name + " :: "+i);
        try{
            Thread.sleep(500);
        }
        catch (Exception e) {
            SOP.print(e.getMessage());
        }
    }
    //synchronized blok
    synchronized(this){
        this.count += value;
    }
}
```

### 102. Jak se liší požadavky na třídy při implementaci polymorfismu pomocí abstract class a interface

Při konkrétní implementaci tolik ani ne, jen snad tím, že lze implementovat více interface, ale extend lze pouze jednu konkrétní třídu

Liší se ale interface a abstract class, interface je jen prázdná kostra, která sice může obsahovat defaultní implementace metod, ale to je asi tak všechno

Abstract class obsahuje proměnné a metody jako každá jiná třída, nelze však vytvořit její instanci

### 103. Co rozumíte pojmem genericita a k čemu slouží?

Generická metoda:

Vrací právě takový typ, který ji pošleme v generickém parametru, něco jako template v C++.

```
//Přemění list typu T na pole typu T a vrátí jej
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o); // Correct
    }
}
```

Generické třídy/interface:

Vytvoří třídu, kde T bude nahrazeno libovolným typem, v polích i v metodách:

```
public class Box<T> {
    // T stands for "Type"
    private T t;
```

```
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Použití:

```
Box<String> stringBox = new Box<String>();
```

Na pozadí se vytvoří instance, která by měla takovouto definici:

```
public class Box {
    private String t;
    public void set(String t) { this.t = t; }
    public String get() { return t; }
}
```

**104. K čemu je rozhraní runnable?**

Použijeme ho pro takovou třídu, jejíž instance poběží v nějakém threadu, nebo bude jinak využita pro pozdější spuštění. Ve třídě, která jej implementuje, vytvoří metodu Run. Runnable se dá také předávat lambdou.

**105. Když zastíníme metodu equals(), je potřeba zastínit metodu hashCode()?**

[StackOverflow](#)

**106. Popište použití rozhraní pro přístup ke kolekcím:**

`Collection<E>, HashMap<K,V>, List<E>, Set<E>`

`Collection<E>` je společný předek pro `List` i `Set`, přináší základní metody jako `containsAll`, `addAll`, `deleteAll`.

`HashMap<K,V>` je mapa typu `K` na typ `V`. Například pokud chceme ukládat známky podle jména, použijeme `HashMap<String, Integer>`.

`List<E>` je seznam, Prvek lze přidat na libovolné místo, odebrat z libovolného místa. Typicky se používají instance `ArrayList` nebo `LinkedList`, které tento interface pochopitelně implementují.

`Set<E>` česky množina, obsahuje každý prvek pouze jednou - nezaručuje pořadí prvků, `SortedSet` už jej zaručuje

Ukázka `HashMap`:

```
HashMap<Integer, String> hmap = new HashMap<>();
// Adding elements to HashMap
hmap.put(12, "Chaitanya");
// Get values based on key
String var= hmap.get(2);
```

**107. K čemu je: isAlive(), join(), sleep(), yield()**

Vše se používá v threadech

`isAlive` vrací true pokud thread běží  
`join` počká na skončení nějakého jiného threadu, eventuálně vrátí výsledek práce generického threadu  
`sleep` uspí thread na zadanou dobu - po tuto dobu nebude thread pracovat  
`yield` pokusí se thread posunout na konec prioritní fronty, aby mohly pracovat jiné ready, [efekt je ale bez záruky.](#)

**108. Jaké znáte nečíselné primitivní datové typy?**

boolean, char

**109. Jak se zapisují konstanty?**

Java žádné konstanty nemá, můžeme ale napsat static final int CONST = 5, u kterého bychom si měli být jistí, že ho omylem nezměníme

**110. Jak se liší stack a queue (zásobník a fronta)?**

Queue - first in first out - kdo první přijde, ten první odchází

Stack - last in first out - to co umístím jako poslední vytáhnu jako první

**111. Co je to vnitřní třída a k čemu slouží?**

```
public class OuterClass {  
    private InnerClass inside;  
    public static void main(String[] args) {  
        OuterClass outerClass = new OuterClass();  
        outerClass.createInner();  
        System.out.println(outerClass.inside.gender);  
  
    }  
  
    void createInner(){  
        this.inside = new InnerClass();  
    }  
  
    private class InnerClass {  
        boolean gender = false; // false → female  
    }  
}
```

Instance vnitřní třídy může existovat pouze s instancí vnější třídy.

Pro vytvoření instance vnitřní třídy musíte nejprve vytvořit instanci vnější.

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

**112. Co to je fiktivní příkaz při předávání parametru fce?**

`~`

**113. Jak java reaguje na ztrátu reference?**

Když už na objekt neukazuje reference, garbage collector ho sežere. Viz též [null pointer exception](#).

**114. Popište trojvrstevnou strukturu proudů.**

Přenos informace se děje ve více vrstvách v proudech (streams)

1. Otevření přenosového proudu pro byty nebo znaky
2. Otevření přenosového proudu pro datové typy Javy
3. Filtrace dat podle dalších požadavků, např. bufferovaní, řádkování, atd.

**Bytové –**

FileInputStream/FileOutputStream

DataOutputStream – přenos primitivních datových typů

ObjectOutputStream – přenos objektů

BufferedOutputStream – bufferovaní

**Znakové –**

FileReader/FileWriter

BufferedReader – bufferovaní

StreamTokenizer – tokenizace

**115. Jakého typu musí být formální parametr a skutečný parametr?**

Formální parametr - void exterminatus(Race race)

Jako skutečný parametr lze ale použít cokoliv co dědí z Race - např. class Ork  
extends Race

Formální parametry nalezneme v deklaraci metody v její hlavičce. Stejně jako jakékoliv jiné proměnné musí být i formální parametry deklarovány s uvedením datového typu a názvu. Skutečné parametry používáme při volání metody, s jejich hodnotami metodu voláme.

**116. Pro jaký typ aplikací je Java nepoužitelná?**

Embedded systémy, jednak kvůli tomu, jak neefektivně pracuje s pamětí, jednak kvůli tomu, že by se na daném systému musela zprovoznit JVM. Na mnoha embedded zařízení rovněž není možné používat dynamickou alokaci paměti.

# Zkouškový test - otázky z jazyka C

Zkouškový test podle všeho může obsahovat libovolné dotazy ze zápočtového testu. Psal jsem oba testy zároveň a jedna otázka se překrývala doslovně (byla úplně stejná) a jedna se lišila jen ve formulaci. Ve zkouškovém testu jsou navíc otázky z jazyka C.

## 1. Jak v C zapisujeme identifikátory (jména funkcí a proměnných)

- 1.1. Deklarace funkce s návratovou hodnotou typu int a dvěma parametry - const int a ukazatel na int:

```
int funkce(const int cislo, int* pointer_na_cislo);
```

Definice této fce (například):

```
int funkce(const int cislo, int* pointer_na_cislo) {  
    if(pointer_na_cislo != NULL)  
        return cislo + *pointer_na_cislo;  
    else  
        return cislo;  
}
```

- 1.2. Deklarace a inicializace proměnné typu int:

```
// deklarace  
int cislo;  
// inicializace  
cislo = 5;
```

Mějme na paměti, že hodnota proměnné před inicializací **není definována** - může tam být cokoliv!

Více o identifikátorech proměnných v bodu [5. zápočtového testu](#).

## 2. Jaký je v C rozdíl v přístupu k položkám proměnné složeného typu a ukazatele na složený typ?

[Viz 12.](#)

## 3. Co v C reprezentuje type void\*?

Pointer, který ukazuje tam kde začíná nově alokovaný blok. Na rozdíl od jiných ukazatelů, u void\* kompilér nezná velikost alokovaných dat. Nemůžete na něj tedy použít konstrukt sizeof. Tento pointer se často používá pro předávání binárních dat arbitrární velikosti. Můžete například funkci která zapisuje data do souboru:

```
bool zapisDoSouboru( void* data, int delka);
```

Tu pak můžete použít s jakýmkoliv datovým typem:

```
int cislo = 666;  
// pozn. tohle je z hlediska kompatibility blbej napad  
// velikost cisla se muze lisit  
// a skoncите tak s ruznejma souborama pro stejnej vstup  
zapisDoSouboru((void*)&cislo, sizeof(cislo));
```

Druhý použití je propašování dat přes nějaký univerzální kanál. Například při

posílaní eventů můžete libovolná data poslat jako `void*` argument. Pokud je na druhé straně funkce která ví, jak `void*` převést zpátky, bude to fungovat.

**4. Jak probíhá proces spuštění programu implementovaného v jazyce C?**

Na začátku programu je napsáno, která funkce se má spustit (`main`).

**5. Jaký význam má hlavičkový soubor zdrojových souborů programu v C?**

Obsahuje deklarace funkcí které budou implementovány v souboru `.c`. Každá funkce smí být definována libovolně-krát. Každá ale smí mít pouze jednu implementaci.

Každý soubor `.c` se kompiluje zvlášť. Máte-li definici funkce v souboru `.h` hrozí, že bude zkompilována vícekrát pro více souborů `.c` a dojde ke konfliktu při vytváření výsledného programu.

**6. Jak v jazyce C dynamicky alokujete paměť za běhu programu?**

Pomocí funkcí `malloc` a `calloc`. `calloc` nastaví byty alokované paměti na 0, `malloc` ne, je proto rychlejší.

**7. Je možné v jazyce C volat funkci ze sebe sama (rekurze)?**

Ano.

**8. Co v C reprezentuje identifikátor `NULL`?**

`NULL` je "prázdný pointer" resp. pointer na adresu, která je systémem rezervovaná a garantovaně na ní nic není. Vrací ji proto např. paměť alokující funkce (`malloc`, `calloc`, `realloc`), které v případě chyby musí vrátit pointer, a proto vrací právě `NULL`. V C99 a na rozumné platformě `((void *) 0x0) == NULL`.

**9. Jak při překladu programu v C rozšíříme seznam prohledávaných adresářů s hlavičkovými soubory?**

Je potřeba použít parametr kompilátoru. Pro GCC to je `-I`. Lze použít `-isystem`, složky takto zadané budou prohledány jako poslední a nebudou generovat varování.

Zdroj: <https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Directory-Options.html>

**10. Záleží u komplikace programu v C při specifikaci adresářů s hlavičkovými soubory na jejich pořadí?**

Existují knihovny, které již očekávají, že budou určitá makra a funkce definovány.

Nicméně jak [příše uživatel na StackOverflow](#):

*What's annoying is when some headers require other headers to be included*

*first...                   That's a problem with the headers themselves, not with the order of includes.*

Lakonická odpověď tedy zní ano. Nicméně, správně navržený projekt by na pořadí neměl trvat, každý hlavičkový soubor by si měl sám includovat symboly, se kterými pracuje.

**11. Podporuje jazyk C přetěžování jmen funkcí? Pokud ano, tak od jaké verze?**

[Ne úplně](#). C11, GCC 4.9, podporuje klíčové slovo `_Generic`, které umožňuje simulovat operator overloading. C99, které používáme my, to však nepodporuje.

**12. Jak v C přistupujeme k datovým položkám složeného typu (`struct`)?**

Pomocí tečky, jako v javě:

```
struct Point {  
    uint32_t x;  
    uint32_t y;  
}
```

```

int main() {
    struct Point a;
    a.x = 5;
    a.y = 10;
}

```

Pokud se jedná o ukazatel, používáme šipku:

```

int main() {
    struct Point* a = malloc(sizeof(struct Point));
    a->x = 5;
    a->y = 10;
    free(a);
}

```

- 13. Je součástí jazyka C typ logické hodnoty “true/false”? Pokud ano, jak se používá? Pokud ne, jak jej definujete?**

Není, používá se nenulový nebo nulový `unsigned char`. Nejjednodušší metoda jak si zajistit `bool`, `true` a `false` je include ze standardní knihovny:

```
#include <stdbool.h>
```

Další možnosti viz StackOverflow: [Using boolean values in C](#)

- 14. Jak jsou v C reprezentovány textové řetězce?**

Pole charů zakončeno “\0” (hex 0x00).

- 15. Jak v C definujete složený typ (`struct`)?**

```
typedef struct {
    Type name;
} Jmeno;
```

nebo

```
struct Jmeno {
    Type name;
};
```

pak se ale při deklaraci musí používat `struct`.

- 16. Jak v C definujete ukazatel na proměnné, např. typu `int`?**

```
int* a; //ukazatel na int
```

- 17. Jsou v jazyce C definovány rozsahy neceločíselných typů?**

Nejsou, rozsah se liší podle platformy, o jejich velikosti rozhoduje **kompilér**. (pozn. na 64/32bit to [zpravidla nezáleží](#))

- 18. Jak funguje modifikátor `extern` při definici proměnné v jazyce C?**

**Martin Zázvorka:** Promnena se alokuje pri spustení a dealokuje pri ukončení programu. Je videt i z jineho souboru nez ve kterem je vytvorena. `extern` je od *external linkage*.

- 19. Jak v C zajistíte načtení textového řetězce ze souboru aniž byste překročili alokovanou paměť určenou pro uložení řetězce?**

Pomocí funkce `fgets()` která obsahuje parametr kolik toho má maximálně přečíst.

Může být použito `fscanf(FILE *fname, "%Ns", &pole)` přičemž N je číslo kterým omezíte počet načtených prvků;

- 20. Popište jak v C probíhá volání funkce `int do(int r)`? Jaká data jsou předávána do/z funkce a kam jsou hodnoty ukládány?**

**Michal Bahník:** Asi jde o chyták, `do` je ten "speciální" výraz (`z do{...} while()`). Z i do y při přejmenování měl lézt int.

**Tato otázka byla opravena podle materiálů na Course Ware** - funkce byla přejmenována na `int doit(int r)`. V tomto případě samozřejmě už je potřeba odpovědět:

Při volání `doit` nejdřív volající funkce předá argumenty. Pořadí vyhodnocení argumentů, jedná-li se o výrazy **není zaručeno**. Tedy následující výraz může mít neočekávané chování:

```
int i=0;  
some_function(i, i++);
```

Pak dojde k vytvoření místa na **stacku** pro naši fci `doit`. Na stacku jsou pak rovněž její argumenty, v tomto případě `int r`. Ty se tam z námi zadaných hodnot **překopírují!** Toto je nutno mít na paměti, chystáte-li se jako argument předat paměťově náročné struktury. Zabírá-li struktura, kterou předáváte hodně paměti, použijte raději pointer.

Poté, co funkce `doit` skončí, zkopíruje se její návratová hodnota na stack volající funkce - pokud ji tedy někam přiřadíme.

- 21. Jak rozlišíte literál typu `int` a `long`?**

Pozn.: Literál je ruční zadání nějakých dat, například když zadáváte string do uvozovek, nebo zapisujete číslo.

Literál typu `int` - normálně napište číslo:

```
const int cislo = 42;
```

U `long` je potřeba napsat za číslo (L/l na konci):

```
const long long_cislo = 9999999999L;
```

Další literál je třeba pro `float`, tedy desetinné číslo (f/F na konci):

```
const float zlomek = 0.5f;
```

Literál zadáný v šestnáctkové soustavě (0x/0X na začátku):

```
const void* don_t_do_this_use_NULL_instead = 0x0;
```

Důvodem pro toto rozlišení je, že pokud napíšete literál typu `int`, může přetéct pokud je větší než velikost typu `int`. Při dělení můžete zase použít `float` literál a zajistit, že dělení nebude celočíselné - tedy že výsledek bude desetinné číslo a ne `int`.

- 22. Jak zjistíme velikost datové reprezentace základních celočíselných typů v jazyce C?**

U libovolného datového typu - včetně struktur - lze velikost (v bytech) zjistit operátorem `sizeof`. Například:

```
size_t velikost_intu = sizeof(int);
```

V žádném případě byste neměli na základě velikosti datových typů dělat nějaká rozhodnutí. Vždy používejte přímo operátor `sizeof`, nikam velikost neukládejte. Maximálně do konstanty, chcete-li si ušetřit psaní. Nejde o to, že by to bylo nebezpečné, ale že pokud to potřebujete, patrně děláte něco špatně.

23. Jak v C dynamicky alokujete paměť pro uložení posloupnosti 20 hodnot typu int? Jak následně takové dynamické pole zvětšíte pro uložení dalších 10 položek?

```
int* numbers = NULL;
const size_t array_length = 20;
numbers = (int*) malloc( array_length * sizeof(int) );
if(numbers==NULL) {
    fprintf(stderr, "Out of memory error!");
    exit(127);
}
// Zvetseni o 10
const size_t new_array_size = array_length+10;
numbers = realloc(numbers, new_array_size*sizeof(int));
// Zde by se mela opakovat kontrola jestli numbers není NULL
```

Toto samozřejmě není nejkratší možný kód co řeší problém, v praxi se ale opravdu často vyplatí si návratové hodnoty kontrolovat a velikosti ukládat do konstant nebo definovat jako makra.

24. Garantuje uvedení `const` u deklarace proměnné, že není žádná možnost jak příslušnou hodnotou proměnné změnit?

Rozhodně ne. Zkuste například toto:

```
const int unchangeable = 42;
int* changeable_ptr = NULL;
const int* unchangeable_ptr = &unchangeable;
// Pouzijeme memcpy, kompilér nepozna ze presouvame
// adresu z const pointeru na normalni pointer
memcpy(&unchangeable_ptr, &changeable_ptr, sizeof(int*));
// Zaremuje toto pokud chcete aby program probehl normalne
*changeable_ptr = 666;
printf("Unchangeable number: %d", unchangeable);
return 0;
```

Na [ideone](#) můžeme vidět, že tento program spadne za chodu. Ani toto však není globálně zaručeno. Pracujte s ukazateli na konstanty opatrně.

Jste bych dodal, že prepisování `const` je proti standardu specifikace.

25. Jaké znaky používané v C pro řízení výstupu?

Tohle je poněkud matoucí otázka. Patrně jde o metaznaky:

- 25.1. \n - nový řádek
- 25.2. \t - tabulátor, v pekle je speciální koutek pro lidi, co tento znak používají v jakémkoliv kontextu
- 25.3. \r - carriage return. Pokud pošlete pouze \r tak na některých terminálech můžete začít přepisovat aktuální řádku

25.4. \a - tón varování. Na nových počítačích cinknutí - pokud je vůbec podporován - na starých pípnutí

Nicméně žádný z těchto znaků není charakteristický pro C a jejich podpora je nekonzistentní.

Alternativně by mohlo jít o formátovací sekvence printf - %d a podobně.

## 26. Jak se v C liší ukazatel a pole[]?

Záleží na kontextu. Postupně:

**Pole jako deklarovaná proměnná:**

```
int pole[] = {1,2,3,4}; // kompiler velikost doplní sám
int neinicializovane_pole[4];
const char pole_charu[] = "Hello world.;"
```

V tomto případě se na stacku připraví souvislá paměť pro tato pole. Obě proměnné jsou technicky nejblíže typu int\* const - tedy ukazatel na číslo (nebo spíš čísla), jehož adresu nesmíme měnit.

Je tu ale zásadní rozdíl: sizeof(pole) vrací celkovou velikost pole v bytech. Chceme-li procházet pole, správný postup je:

```
int pole[] = {1,2,3,4}; // kompiler velikost doplní sám
for(int i=0; i<sizeof(pole)/sizeof(int); ++i) {
    ... kod ...
}
```

Paměť pro tato pole se uvolní jako u jakékoliv jiné proměnné. Pořád si ale musíme dávat pozor, abychom nelezli mimo pole.

**Pole variabilní délky:**

Běžně se chovají jako obyčejná pole, liší se tím, že jejich délka není určena konstantou ale třeba proměnnou:

```
size_t delka = 0;
printf("Zadej delku pole vole:");
scanf("%zu", &delka);
if(delka == 0)
    return 1;
int nove_pole[delka];
```

Doporučuju nastudovat více, chcete-li tato pole používat jako argumenty a podobně.

**Pole jako argument:**

```
void funkce(const int pole[10]);
```

Toto se nijak neliší od const int\* a je jedno, jestli v závorkách je nějaké číslo nebo ne. jediná výjimka, pouze od C99:

```
// predane pole musi mit alespon 10 prvku
void funkce(const int pole[static 10]);
```

Více na [StackOverflow](#).

## 27. Stručně popište typ union používaný v jazyce C.

Stručně to můžete napsat pak do testu. Typ union lze použít, chcete-li na stejné místo v paměti ukládat různé datové typy. Typické použití:

```
typedef enum { INTEGER, STRING, REAL, POINTER } Type;

typedef struct
{
    Type type;
    union {
        int integer;
        char *string;
        void *pointer;
    } x;
} Value;
```

Jak je vidět, potřebujete druhou proměnnou, abyste si pamatovali datový typ, který v dané paměti ve skutečnosti je. Použití takovéto struktury:

```
Value value_new_integer(int v)
{
    Value v;
    // INTEGER pochází z enum definovaného výše
    v.type = INTEGER;
    v.x.integer = v;
    return v;
}
```

Přirozeně v praxi potom na ukládání/načítání hodnot budete mít velký switch/case. Pokud se pokusíte přečíst jiný typ, než máte uložený, dostanete nesmyslná data. Systém toto nijak nekontroluje, ani nemůže.

## 28. Jaké znáte překladače jazyka C?

Nechápu smysl otázky, každej si přece může vygooglit "C compilers"... Což je přesně to co jsem udělal poté co jsem vyčerpal první tři:

- 28.1. **gcc** (GNU Compiler Collection)- asi "nejlepší". Drží se standardů, je opensource a chodí na velkym množství systémů.
- 28.2. **mingw** - pouze port GCC pro dělání Windows aplikací, je ale natolik významnej, že se zaslouží druhý místo. Jeho alternativou je Cygwin
- 28.3. **clang** (C language family frontend for LLVM) - no comment
- 28.4. **keil** - pojmenovanej podle firmy která se zabývá embedded systémama. Ukázalo se, že je docela rozšířenej, jejich IDE ale stojí za pícu
- 28.5. **MSVC** - microsoft kompilér. On není zase tak špatnej, ale pokud nemáte k dispozici nejnovější verzi, budete trpět.

29.

30. prázdnou položku vždy nechte na konci seznamu, dík

# Zkouška 2016Z

Jsou-li některé otázky již zodpovězeny výše, vytvořte na ně odkaz. Odkazy jdou vytvořit i na záložky v dokumentu.

## 1. Jak v C vytisknete řetězec na standardní výstup ?

```
char *s = "řetězec";  
printf("%s\n", s);
```

## 2. Jaké bloky pro řízení cyklu definuje for cyklus?

Pravděpodobně to bude toto:

```
for(inicializace; podmínka; iterace){ ... }
```

Šance je, že se chtěl zeptat na continue a break.

## 3. Popište k čemu v C slouží příkaz dlouhého skoku.

Převzato ze 7. přednášky

Příkaz goto je možné použít pouze v rámci jedné funkce

Knihovna <setjmp.h> definuje funkce setjmp() a longjmp() pro skoky mezi funkcemi  
setjmp() uloží aktuální stav registrů procesoru a pokud funkce vrátí hodnotu různou  
od 0, došlo k volání longjmp()

Při volání longjmp() jsou hodnoty registrů procesoru obnoveny a program pokračuje  
od místa volání setjmp()

"Kombinaci setjmp() a longjmp() lze využít pro implementace ošetření výjimečných  
stavů podobně jako try–catch " - tedy zavedení a ošetření výjimek

## 4. Ve kterém hlavičkovém souboru standardní knihovny jsou definovány matematické funkce?

Záchranná otázka pro plebs...

```
#include <math.h>
```

## 5. Jaké znáte funkce standardní knihovny pro náhodný přístup k souborům?

**int fseek, void rewind - Slidy 7. přednášky, je tam práce se soubory**

## 6. Jak zjistíte, že jste při čtení souboru dosáhly [sic!] konce souborů? Jakou funkci standardní knihovny použijete?

Pokud dojdete na konec souboru, většina “čtecích” funkcí, které normálně vrací počet načtených prvků, vrátí nulu. Samotný konec souboru pak rozlište funkcí `feof(FILE * file)`.

Příklad viz na stránce [cppreference](#).

7. **proměnná vs. ukazatel (už je popsáno výše, najděte to někdo)**
8. **Jaké znáte pořadí navštívení uzlů binárního stromu?**

[Binary tree traversal](#) - pre-order, in-order, post-order (viz [2. odkaz](#))

9. **Kdy budete reprezentovat binární strom polem?**

Pokud si budeme jistí, že se bude jednat o plný strom (nebo když chceme implementovat binary heap (je to taky binarni strom)).

10. **Charakterizujte rozdíly v implementaci zásobníku polem a spojovým seznamem.**

Polem: musíme znát předem počet prvků (nebo realokovat), jen prvky stejného typu  
Seznamem: libovolný počet prvků, libovolný typ; k reprezentaci prvků se typicky používá složený datový typ (např. struct), který kromě požadovaných hodnot obsahuje též ukazatel(e) na předchozí/následující prvek v seznamu. pro základní zásobník postačí jednosměrný seznam s ukazateli na další prvek

Přidání prvku:

V obou případech je potřeba udržovat ukazatel na poslední prvek. V seznamu však každý prvek alokujeme zvlášť. Následně je třeba nastavit ukazatel nového prvku na hlavičku a až poté hlavičku posunout na nový prvek. V poli stačí posunout ukazatel a prvek přidat (pokud není plné).

Odebrání prvku: Obdobné. Pro pole je záhodno jednou za čas měnit velikost (když je z většiny prázdné)

11. **Charakterizujte prioritní frontu. V čem se implementace líší od běžné fronty?**

[Implementace priority queue \(tam najdete odkaz na heap\)](#)

PQ je bezne implementava pres nejaky heap (napr. binary), Q je bezne implementovana pres dynamic array nebo doubly linked list. Hlavni odlišnost PQ od Q je to, ze Q funguje za principem FIFO (First In First Out - první přidaný prvek také jako první odebíráme), ale PQ funguje za principem priazení priorit (jako první odebereme prvek s aktuálně nejvyšší prioritou (reprezentovanou např. číslem)).

12. **Definujte co je to halda a jak se používá pro implementaci prioritní fronty.**

[Co je heap](#)

Jak se pouziva je napsano v odkazu otazky 11.

[Slidy 11. přednášky](#) ukazují přesně použití haldy pro prioritní frontu.

**13. Jakou strukturu použijete na implementaci spojového seznamu, který má složitost operace  $O(1)$  pro přidání prvního nebo posledního prvku.**

# Test v semestru - Procedurální programování

## 1. Co v C reprezentuje `NULL`?

Doporučená četba: <http://stackoverflow.com/q/1296843/607407>

`NULL` reprezentuje adresu nulové paměti. Jedná se o konstantu definovanou ve standardní knihovně. Výraz `if( pointer==NULL )` je korektní způsob jak zjistit, že je `pointer` neplatný. Metody `malloc`, `calloc` a `realloc` vrací `NULL` dojde-li k chybě.

## 2. Jak jsou v C reprezentovány textové řetězce?

Textové řetězce jsou reprezentovány polem charů, tj. `char string[]` nebo `char*` nebo `const char*`. Řetězec zadáný literálem je typicky `const`. Pole je zakončeno nulovým znakem '`\0`'. Délka paměti, která je pro řetězec alokována není obecně nikde k dispozici. Při používání funkcí jako `strcpy` je nutné s velikostí paměti pracovat zvlášť.

## 3. Co reprezentuje `void`?

`void` v C reprezentuje příznačně "nic":). Ne vážně, `void` indikuje absenci datového typu. To má dvě využití:

- `void jmeno_fce() {}` - signatura funkce bez návratové hodnoty
- `void* raw_data;` - pointer na paměť který nemá žádný datový typ. Pokud chcete číst z lokace, na kterou ukazuje, musíte ho nejdříve přetypovat na nějaký typ. Každý pointer jde přetypovat na `void` pointer a naopak.

## 4. Co způsobí definování `NDEBUG` v souvislosti s `assert`?

`NDEBUG` pochází z "*no debug*" a vypíná tedy debugovací nástroje a symboly. Jedná se o konstantu, kterou lze definovat parametrem kompiléra. Makro `assert` je jedna z věcí, které `NDEBUG` odstraní. Veškeré asserty se vůbec nepromítají do binárky. Toto je potřeba si uvědomovat a nepoužívat `assert` na kontrolu chyb které se reálně mohou vyskytnout!

**Pozn:** Jde definovat i normálně v kódu `#define NDEBUG`. To bych ale v tomto případě **nedoporučoval**, protože pak se konstanta vztáhne jen na kód po použití defnice. Nejsem si jistej, jestli by se to vůbec spustilo.

## 5. Vyjmenujte základní paměťové třídy, ve kterých mohou být uloženy hodnoty proměnných.

Paměťové třídy modifikují typ paměti a přístupu k paměti u proměnné. Existují tyto typy:

- `auto` – toto je implicitní chování lokálních proměnných. Hlavním charakterem je, že je paměť vyhrazena při vstupu do bloku (scope; to v složených závorkách) a opět uvolněna po opuštění bloku dané proměnné.
- `static` – tato proměnná se liší tím, že je její životnost vztažená na celý běh programu. Všechny vstupy do bloku této proměnné budou pracovat se stejnými daty. Vhodné např. pokud chcete inkrementovat číslo po každém zavolání funkce.

5.3. **register** – tenhle modifikátor “zajistí”, že bude proměnná na registru CPU (pokud nevíte co to je, budete rádi. Pokud nejste, použijte google ale varoval jsem vás). V praxi dnes už nemá cenu používat, kompilér moc dobře ví, co dát na registr a co ne a tenhle modifikátor **není závazný**. Nicméně tento modifikátor má jeden vedlejší efekt - **již nemůžete pomocí & získat adresu proměnné**. A to ani v případě, kdy se kompilér rozhodl ji na registr nedat.

5.4. **const** – asi jediná užitečná věc v tomto seznamu. Modifikátor const ohlídá, že neměníte hodnotu proměnné. Tady bych si dovolil malý tutoriál na míchání const a pointerů:

```
const int* cislo; // toto je ukazatel na const int, tedy *cislo je  
const a nesmíte jej menit  
int* const cislo; //konstantní ukazatel na nekonstantní int - lze tedy  
měnit *cislo, ale nelze třeba cislo=NULL  
const int* const cislo; //kombinace obojího, nelze menit ani cislo ani  
*cislo
```

Ani const samozřejmě neznamená 100% záruku, že se proměnná nezmění. Vygooglete si “cast away const”.

5.5. **extern** – rozšiřuje viditelnost statických proměnných z modulu na celý program. Globální proměnné s extern jsou definované v datové oblasti.

Toto níže je špatně (ponecháno protože je to dobrá odpověď najinou otázku, snad bude časem přesunuto):

Potřeboval bych se podívat do přednášek, ale předpokládám, že Faigl myslí "heap" (halda) a "stack". Heap se používá pro dynamickou alokaci, na stacku jsou statické proměnné jednotlivých funkcí, které právě probíhají. Ke slavné chybě "Stack overflow" dojde, když se probíhající funkce už nevejdou na stack – např. pokud funkce volá donekonečna sama sebe. Více: <http://stackoverflow.com/q/26158/607407>

## 6. Vysvětlete rozdíl mezi proměnnou a ukazatelem.

Technicky vzato, ukazatel je taky proměnná. Je to proměnná typu ukazatel, jejíž hodnota je číselná adresa paměti jiné proměnné. Toto je potřeba si v C uvědomovat. Jinak samozřejmě platí:

```
int cislo; //promenna  
int* pointer_na_cislo = &cislo; // pointer - & pred promennou nam vrati její  
adresu  
int cislo_2 = *pointer_na_cislo; // hvězdičkou získame hodnotu paměti, na  
kterou pointer ukazuje
```

Prosím mějte na paměti, že dokud operujete s pointerem, chcete-li změnit hodnotu na kterou ukazuje, musíte před něj dát tu hvězdičku. Zároveň věnujte taky pozornost faktu, že cislo\_2 obsahuje **KOPII** toho, co je v \*pointer\_na\_cislo, tyto proměnné pak dále již nejsou nijak svázané!

## 7. Napište program, který bude mít následující výstup:

\*\*\*\*\*

\*\*123\*\*

\*\*456\*\*

\*\*\*\*\*

Toto nestojí za řeč. Pokud nevíte:

printf ("\*\*\*\*\*\n\*\*\*123\*\*\n\*\*456\*\*\n\*\*\*\*\*\n");

**Poznámka:** Pan Fišer nás varoval, ať se mu tam toto ani neopovažujeme napsat. Je doufám jasné, že řešení výše nepředvádí programátorské schopnosti v hodnotě dvou bodů.