1. Vývoj programovacích jazyků, styly a vlastnosti

1.1. Charakterizujte **paradigmata** programování
procedurální – program je tvořen sekvenčně prováděnými procedurami, charakteristickým rysem je přiřazovací příkaz (Pascal, C)
OOP – principem je modelování reálného světa pomocí objektů, objekt je černá skříňka, která provádí určitou činnost a komunikuje s okolím – zapouzdření, dědičnost, polymorfismus (Java, C#)
funkcionální – algoritmus je popsán pomocí funkcí, často se pracuje se seznamy, tento styl je blízký matematice, je však dosti nepřehledný (Lisp), lambda kalkul, základní konstrukce - výraz
deklarativní – pomocí deklarací popisuje co se má řešit, ale ne jakým způsobem (tedy nikoliv algoritmizace), patří sem logické a funkcionální jazyky (neboli neprocedurální jazyky)
aplikativní – výpočet je popsán pomocí výrazů
logické – založeno na deklaraci faktů (predikátů) a relací (vztahů), často rekurzivně definovaných, využívající logických formulací k vyvozování požadovaných informací, používají se v UI (Prolog)
souběžné – může být paralelní nebo distribuované, principem je souběžné zpracování více úloh (multitasking nebo skutečný paralelismus)
vizuální – základem vizuální návrh a událostní řízení. Slouží pro komunikaci s uživatelem (GUI) – ovládací prvky, manipulace s objekty a jejich parametry (Delphi, .NET WinForms)

1.2. Jaká jsou **globální kritéria** na programovací jazyk?
spolehlivost – typová kontrola, zpracování výjimečných situací
efektivita – překladu a výpočtu
strojová nezávislost
čitelnost a vyjadřovací schopnosti – jednoduchost, ortogonalita, humánní čitelnost/zapisovatelnost
řádně definovaná syntax a sémantika – strojová a humánní čitelnost
úplnost v Turingově smyslu – schopnost popsat libovolný algoritmus

1.3. Co ovlivňuje **spolehlivost** programovacího jazyka?
typová kontrola, zpracování výjimečných situací apod...

1.4. Co ovlivňuje **efektivitu** programovacího jazyka?
efektivita procesu překladu a vlastního výpočtu

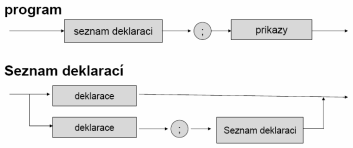
1.5. Co ovlivňuje **čitelnost a vyjadřovací schopnosti** programovacího jazyka?
jednoduchost (kontra př. V C: a++, a+=1, ++a, a=a+1)
ortogonalita – malá množina primitivních konstrukcí, ze kterých lze kombinovat další
strukturované příkazy, všechny kombinace jsou legální (kontra př. C: struktury fční. hodnotou, pole ne)
strukturované příkazy a datové typy
podpora abstrakčních prostředků
strojová čitelnost – existence algoritmu překladu s lineární časovou složitostí
humánní čitelnost – závisí na způsobu abstrakce dat a řízení

1.6. Definujte pojmy **syntax a sémantika** programovacího jazyka.
syntax – forma či struktura výrazů, příkazů a programových jednotek
sémantika – význam výrazů, příkazů a programových jednotek

1.7. Definujte pojem **úplnosti v Turingově smyslu**.
programovací jazyk je úplný v TS, pokud je schopný popsat libovolný algoritmus (stačí celočíselná aritmetika a proměnné, sekvenčně prováděné příkazy a cyklus (while))

1.8. Zapište v **BNF** (Backus Naurova forma) tvar příkazu.
<program> → <seznam deklaraci> ; <prikazy>
<seznam deklaraci> →
  <deklarace> |
  <deklarace>;<seznam deklaraci>
<deklarace> → <spec.typu> <sez.promennych>
if: <if> → <podmínka>; <tělo> | <podmínka>; <tělo>; <else-větev>

1.9. Zapište **syntaktickým diagramem** tvar příkazu.



1.10. Objasněte pojmy **statická a dynamická sémantika**.
statická – neměnný význam výrazů, příkazů a programových jednotek, ověřitelná při překladu (nedef. proměnná, chyba v typech)
dynamická – překladač ji nemůže ověřit při překladu (dělení nulou)

1.11. Jaké **druhy chyb** v programu rozlišujeme?
lexikální – nedovolený znak
syntaktické – chyba ve struktuře
statické sémantiky – nedef. proměnná, chyba v typech
dynamické sémantiky – dělení nulou, nastávají při výpočtu, neodhalitelné při překladu
logické – chyba v algoritmu

1.12. Jaké **druhy chyb a ve které fázi** je schopen nalézt překladač?
lexikální a syntaktické chyby při překladu
chyby statické sémantiky až před výpočtem

2. Logické programování - Prolog

2.1. Co jsou to **termy** jazyka Prolog?
termy jsou elementy programu jako konstanty, proměnné a struktury pro označování objektů

2.2. Co jsou **atomy** Prologu?
atomy jsou nečíselné konstanty, mohou to být:
- posloupnosti písmen, čísel a dalších znaků začínající malým písmenem (a06, franta)
- posloupnosti libovolných znaků uzavřených v uvozovkách (“NECO“)

2.3. Co je **anonymní proměnná** Prologu a jaké má vlastnosti?
jedná se o speciální proměnnou označenou jako “_“, není nikdy vázána na žádnou hodnotu (na jejím obsahu „nezáleží“). Může jich být v klauzuli i více, navzájem spolu nesouvisí, zpřehledňuje programy.

2.4. Princip **rezoluce**.
metoda hledání sporu v koneč. množ. klauzulí, rozvinutí podcílů. a:-a1,a2. b:-b1,a,b2 → b:-b1,a1,a2,b2

2.5. Princip **unifikace**.
porovná-li se volná proměnná s konstantou, naváže se na tuto konstantu
porovnají-li se dvě volné (neinstalované) proměnné, stanou se synonymy
porovná-li se volná proměnná s termem, naváže se na tento term
porovnají-li se termy, které nejsou volnými proměnnými, musí být pro úspěšné porovnání stejné

2.6. Způsob **plnění cílů** v Prologu.
Dotaz může být složen z několika cílů. Při konjunkci cílů jsou cíle plněny postupně zleva. Pro každý cíl je při jeho plnění prohledávána databáze od začátku. Při úspěšném porovnání klauzule s cílem je její místo v databázi označeno ukazatelem. Každý z cílů má vlastní ukazatel. Při úspěšném porovnání cíle s hlavou pravidla pokračuje výpočet plněním cílů zadaných tělem pravidla. Cíl je splněn, je-li úspěšně porovnán s faktem databáze nebo s hlavou pravidla databáze a jsou splněny podcíle těla pravidla. Není-li během exekuce některý cíl splněn ani po prohlédnutí celé databáze, je aktivován mechanismus návratu. Splněním jednotlivých cílů dotazu je splněn globální cíl a systém vypíše hodnoty proměnných zadaných v dotazu. Zjistí-li se při výpočtu, že globální cíl nelze splnit, je výsledkem „no“.

2.7. Jak probíhá **návrat při nesplnění cíle** v Prologu?
exekuce se vrací k předchozímu splněnému cíli, zruší se instalace proměnných a pokouší se opětovně splnit tento cíl prohledáváním databáze dále od ukazatele na tento cíl. Splní-li se opětovně tento cíl, pokračuje se plněním dalšího (předtím nesplněného) vpravo stojícího cíle. Nesplní-li se předchozí cíl, vrací se výpočet opětovně zpět.

2.8. Vysvětlete mechanismus působení **predikátu řezu**.
používá se pokud chceme zabránit hledání jiné alternativy. Odřízne další provádění cílů z hlavy pravidla. Je bezprostředně splnitelným cílem, který nelze opětovně při návratu splnit. Projeví se pouze, když má přes něj dojít k návratu. Změní se mechanismus návratu tím, že znepřístupní ukazatele vlevo od něj ležících cílů (přesune je na konec DB).

2.9. Čím se odlišuje **červený a zelený řez** Prologu?
červený řez – znemožní návrat, který by našel jiné řešení (mění deklarativní smysl)
zelený řez – znemožní návrat, který by stejně skončil neúspěchem (nemění deklarativní smysl)

2.10. Jaký účinek má **predikát**
repeat – nekonečněkrát splnitelný cíl
fail – vždy nesplněný cíl
asserta(X) – přidání klauzule instalované na X na začátek databáze

3. Funkcionální programování - Lisp

3.1. Co jsou to „**čisté výrazy**"?
nemění stavový prostor (globální proměnné) programu, nemají vedlejší efekt. Lze je bez problémů použít při paralelním programování.

3.2. Popište **Church-Roseovu vlastnost** výrazů.
hodnota výsledku nezávisí na pořadí vyhodnocování výrazu, výraz lze vyhodnocovat paralelně

3.3. Definujte **S-výrazy** (symbolické výrazy) Lispu.
- atomy – čísla (celá i reálná), znaky, řetězce, symboly k označování proměnných a funkcí, (T, NIL)
- seznamy (el1 el2 ... elN), NIL

3.4. Popište základní **cyklus vyhodnocování** lispovského programu.
- výpis promptu
- uživatel zadá lispovský výraz (zápis funkce)
- provede se vyhodnocení argumentů
- aplikuje funkci na vyhodnocené argumenty
- vypíše se výsledek (funkční hodnota) a pokračuje se opět výpisem promptu
při chybě se přejde do nové úrovně interpretu

3.5. Jaké jsou **elementární funkce** Lispu a jejich sémantika?
CAR (FIRST) – selektor, který vrací první prvek seznamu
CDR (REST) – selektor, který vrací zbytek seznamu
CONS – konstruktor CONS('a 'b) → (a . b)
ATOM – test zda se jedná o atom
NULL – test, zda je seznam prázdný nebo má výraz hodnotu false
EQUAL – test zda jsou hodnoty argumentů stejné s-výrazy (hodnota)

3.6. Popište sémantiku funkce **COND**.
funkce s proměnným počtem parametrů – umožňuje vykonání těla v závisl. na testovací části, umožňuje výběr alternativy.
syntax: (COND (podm1 forma11 forma12 ... forma1n)
(podm2 forma21 forma22 ... forma2n) ...
(podmk formak1 formak2 ... formakn)
sémantika: COND if podm1 then {forma11 forma12 ... forma1n}
else if podm2 then {forma21 forma22 ... forma2n} ...
else if podmk then {formak1 formak2 ... formakn}
postupně vyhodnocuje podmínky, dokud nenarazí na první, která je pravdivá. Pak vyhodnotí formy patřící pravdivé podmínce. Hodnotou COND je hodnota poslední z vyhodnocených forem. Při nesplnění žádné podmínky není hodnota COND definovaná.

3.7. Popište tvar a účinek funkce **DEFUN**.
(DEFUN jméno-fce (argumenty) tělo-fce)
přiřadí jménu funkce lambda výraz definovaný tělem funkce, tj. (LAMBDA(argumenty) tělo-fce). Vytvoří funkční vazbu symbolu jméno-fce. Argumenty jsou ve funkci lokální. DEFUN nevyhodnocuje své argumenty. Hodnotou formy DEFUN je nevyhodnocené jméno-fce. Tělo-fce je posloupnost forem. Při vyvolání fce se všechny formy vyhodnotí a funkční hodnotou je hodnota poslední z nich.

3.8. Definujte tvar a **využití lambda výrazů**.
lambda výraz specifikuje nepojmenovanou funkci
obecný tvar: (LAMBDA seznam_proměnných forma)
př: ((LAMBDA (X Y) (CONS X (LIST Y))) 1 2)

3.9. Co jsou to **funkcionály**, popište některý z nich.
Funkce, jejichž argumentem je funkce nebo vrací funkci jako svoji hodnotu. Při použití nahradíme název funkce i jméno uvnitř použité funkce skutečnými jmény.
př.: (FUNCALL #'fce argumenty) aplikuje funkci na argumenty
(MAPCAR (FUNCTION +) '(1 2 3 4) '(1 2 3)) aplikuje fci na prvky seznamů, které jsou dalšími argumenty, z výsledků vytvoří seznam

4. Datové abstrakce

4.1. Jaké znáte **konstrukce pro implementaci ADT**?
podprogramy – původní forma abstrakce výpočtů
moduly – kontejnery vzájemně souvisejících podprogramů a dat
kompilační jednotky – kolekce podpgm a dat přeložitelných bez nutnosti souč. překládat zbytek pgm
package ADA, třídy C++, třídy Java, modifikátory přístupu

4.2. Základní **vlastnosti ADT**.
ADT je zapouzdření datového typu a podprogramů poskytujících operace pro tento typ do jedné syntaktické jednotky, je použitelný k deklaraci proměnných
skutečná reprezentace je uživateli skryta, instanci ADT nazýváme objektem

4.3. **Výhody použití ADT**.
lepší organizace programu, lepší modifikovatelnost programu, separátní překlad
větší spolehlivost – důsledkem ukrytí dat
nezávislost uživatele na konkrétní implementaci ADT

4.4. Popište odlišnost **package v jazyce Ada a Java**.
Ada – zapouzdřující konstrukce, dvě části – specification (interface) a body (implementace), obdoba Jav. třídy
Java – balík je seskupením tříd, rozhraní, adaptérů atd., obdoba namespace v C++

4.5. Porovnejte **vlastnosti konstruktorů** jazyků C++ a Java.
C++ – konstruktor inicializuje členská data, ale nevytváří žádné objekty. Může alokovat paměť, pokud je část objektu dynamická na haldě
Java – konstruktor vytváří instanci třídy neboli objekt

4.6. Popište **mechanismus namespace** v C++.
umožňuje explicitní určení rozsahu platnosti jmen pro vyloučení kolizí jmen z různých separátně překl. knihoven. V hlavičk. souboru: namespace JmenoProstoru, klient: using namespace JmenoP…

4.7. Přístup k metodám a proměnným Javy v závislosti na **modifikátoru přístupu**.
public – jsou viditelné všude včetně různých tříd v různých balících
private – přístupné pouze v rámci třídy
protected – přístupné i v potomcích své třídy (i pokud jsou potomci v jiných balících) a ve třídách stejného balíku.
implicitní (žádný modifikátor) – přístupné všem třídám v rámci balíku

4.8. Vysvětlete pojem **parametrický polymorfismus**.
umožňuje násobné využití operace nad různými typy
Ada: generické moduly, klíč. slovo generic a následně gener. Parametry
C++: gener. třídy popis. obecný algoritmus. Při vytváření objektu bude určen konkrétní typ jako parametr. formát: templáře <class Ttyp1, Ttyp2,…> class jmeno-tridy {…}

4.9. Popište základní rysy **generických tříd** C++ a **generických modulů** Ady. – viz. předch.
popisují obecný algoritmus, konkrétní typ dat je určen pomocí parametru až při vytváření objektu
ADA – určení generic. ve specifikační části package, C++ výše

4.10. Jaké typy mohou (nemohou) být generickým parametrem Javy
mohou pouze třídy (např. kontejnery jako ArrayList a LinkedList), rozhraní nemůže

4.11. Uveďte rozdíl mezi parametrizovanými metodami, přetíženými metodami a metodami s param. - Java
parametrizované m. – generické m. Mají typové parametry informující překladač o skutečných parametrech a návratové hodnotě při volání metody

přetížené m. – metody se stejným jménem, ale různými parametry (i různé typy parametrů)
metody s parametry – obecně metody se vstupem realizovaným pomocí parametrů – mohou být přetížené i parametrizované

5. Objektově orientované programování

5.1. Základní **vlastnosti OOP**.
zapouzdření – realizuje se formou ADT, data a metody jsou společně umístěny ve struktuře zvané objekt = instance třídy
dědičnost – třídy mohou dědit data a operace od nadřaz. tříd, jednoduchá vs. vícenásobná dědičnost
polymorfismus – jedna zpráva může mít různé významy, třída může rodiče nejen zdědit, ale i modifikovat, dynamická kontrola typové konzistence.

5.2. Jmenujte **kategorie OOP jazyků** a uveďte jejich příklady.
- OOP prostředky přidány k existujícímu jazyku – C++, Ada95, ObjectPascal, Scheme
- navržen jako OOP jazyk, jeho základní struktura vychází z existujících (imperativních) zvyklostí – Eiffel (nemá přímého předchůdce), Java (vychází z C++)
- čistě OOP jazyk - Smalltalk

5.3. Jaké jsou **možnosti modifikace sw komponent**?
rozšiřování (dědičnost), omezování, předefinování (polymorfismus), abstrakce, polymorfizace

5.4. **Rozdíl mezi ADT a třídami**.
ADT spolu navzájem nesouvisí, netvoří hierarchie (obtížná znovupoužitelnost, jejich přizpůsobování je zásah do celého ADT), kdežto pomocí tříd lze tvořit hierarchie.

5.5. Rozdíl mezi **proměnnými třídy a proměnnými instance třídy**.
proměnné třídy jsou statické, sdílí je všechny instance třídy
proměnné instance (objektu) náleží pouze objektu samotnému

5.6. Jaké **vlastnosti** by měla mít **třída pro využitelnost dědičnosti**?
měla by obsahovat pouze jednu logickou entitu (kontra př. PesAKocka)

5.7. Objasněte pojem **polymorfismus v OOP**, jeho výhody a nevýhody.
jedna zpráva může mít různé významy, třída může překrýt metody rodiče
výhody: snadná modifikovatelnost a rozšiřování programu
nevýhody: dynamická kontrola typové konzistence → časově náročné

5.8. Možnosti **vazby mezi metodou a objektem, který ji volá**.
vazba = určení adresové části příkazu skoku do podprogramu
dynamická (pozdní) – určuje se až při výpočtu → čas. náročné, je nutná pro realizaci polymorfismu
statická – určuje se okamžitě

5.9. Které metody Javy používají **dynamickou** a které **statickou vazbu** s objektem který je volá?
dynamická – metody použité při polymorfismu
statická – typicky u statických metod

5.10. Jakou funkci má **konstruktor** v Javě?
používá se k inicializaci proměnných instance, vytváří vlastní objekt, instance jsou na haldě

5.11. Za jakých okolností systém Java spouští metodu **finalize()**?
jedná se o metodu, kterou má každý objekt. Je spuštěna automaticky, když chce GC zrušit objekt.

5.12. Proč Java nepotřebuje **destruktory**?
má systém GC pro automatické čistění paměti. Každý objekt má metodu finalize(), kterou GC zavolá v případě, že chce objekt zrušit.

5.13. Význam **this** v Javě.
implicitní ukazatel na objekt, ze kterého je zavolán

5.14. Způsob dovolující v Javě pojmenovat stejně proměnnou instance a parametr metody.
pomocí implicitního ukazatele this

5.15. **Modifikátory přístupu** v Javě.
public – jsou viditelné všude včetně různých tříd v různých balících
private – přístupné pouze v rámci třídy
protected – přístupné i v potomcích své třídy (i pokud jsou potomci v jiných balících) a ve třídách stejného balíku.
implicitní (žádný modifikátor) – přístupné všem třídám v rámci balíku

5.16. Jaké podmínky musí splňovat **přetěžované** metody v Javě?
musí se lišit typem nebo počtem argumentů, návratový typ k rozlišení nestačí

5.17. Co je to **ad hoc polymorfismus**?
přetěžování metod (v jedné třídě lze definovat více metod stejného jména, musí se však lišit typem nebo počtem argumentů) a konstruktorů (dovoluje konstruovat objekty různými způsoby)

5.18. Vlastnosti **static metod a proměnných** Javy?
lze je použít nezávisle na jakémkoli objektu, patří třídě a nikoli objektu
volání statické **metody** má tvar: JménoTřídy.jménoMetody(), příkaz volání nahradí překladač skokem na začátek metody
statické **proměnné** jsou v podstatě globální. Existují v jedné kopii, kterou instance sdílejí, ke statickým proměnným se přistupuje: JménoTřídy.jménoProměnné

5.19. **Omezení pro statické metody**.
mohou volat pouze jiné statické metody, nemají definovaný odkaz this, mohou pracovat pouze se statickými daty

5.20. Co jsou **vnořené třídy** Javy a kde jsou použitelné?
jsou to třídy definované uvnitř jiné (vnější) třídy. Jsou použitelné pouze v jejich uzavírací třídě. Mají přístup k metodám a proměnným uzavírací třídy (opačně to neplatí).

5.21. Jak lze ve **vnější třídě** použít proměnnou či metodu **vnitřní třídy**?
je nutno vytvořit instanci vnitřní třídy

5.22. Jaký **způsob dědění** a jakou konstrukci pro definici podtřídy používá Java?
jednoduché (jednopohlavní) dědění
obecná konstrukce: JménoPodtřídy extends JménoNadtřídy {tělo podtřídy}

5.23. Jak dovoluje Java **zpřístupnit v podtřídě privátní elementy z nadtřídy**?
nelze to přímo, je nutno použít autorizovaný přístup - v nadtřídě implementovat metody pro přístup k elementům nadtřídy.

5.24. **Konstruktory** Javy při vytváření instance **podtřídy**.
konstruktor nadtřídy vytváří část objektu patřící nadtřídě
konstruktor podtřídy vytváří část objektu patřící podtřídě
pokud není konstruktor uveden, uplatní se implicitní

5.25. Způsoby použití příkazu Javy **super()**.
definuje-li konstruktor nadtřída i podtřída, musí se při provádění konstruktoru podtřídy vyvolat konstruktor nadtřídy pomocí super(). Musí to být první příkaz konstruktoru podtřídy.

5.26. Možnosti nastavení **přístupových práv u děděných metod a proměnných** Javy.
přístupová práva k předefinovaným metodám a proměnným nelze zeslabit

potomek / rodič	private	neuvedeno	protected	public
private	+	+	+	+
neuvedeno	-	+	+	+
protected	-	-	+	+
public	-	-	-	+

5.27. Uveďte **příklad vytvoření třídy kompozicí**.
class Auto {public Motor motor; public Karoserie karoserie; ...}

5.28. Jaká je v Javě **výjimka ze silného typování** při přiřazení referenční proměnné?
referenční proměnné nadtřídy může být přiřazena referenční proměnná kterékoliv její podtřídy, zpřístupní se pouze ty části objektu, které patří nadtřídě.

5.29. Jaké **části objektu** zpřístupní ref. proměnná nadtřídy, je-li jí přiřazena ref. prom. podtřídy?
pouze ty části objektu, které patří nadtřídě

5.30. Rozdíl mezi **přetížením a překrytím** metod Javy.
překrytí – metoda v podtř. předefinuje metodu v nadtř., má stejné jméno a stejný počet a typ parametrů
přetížení – metoda v podtřídě předefinuje metodu v nadtřídě, má stejné jméno, ale liší se v parametrech

5.31. Proč v Javě **nepostačuje při přetížení metody silný návratový typ**?
při volání metody by nebylo možné rozhodnout která metoda bude vykonána

5.32. Popište **princip objektového polymorfismu**.
předefinování metod (překrytí nebo přetížení) – dynamická identifikace metody = schopnost rozpoznat verzi volané (předefinované) metody až při výpočtu. Obsahuje-li nadtřída metodu předefinovanou v podtřídě, pak se při odkazech na různé typy objektů budou provádět různé verze metod. Rozhodne se na základě typu objektu, jehož referenční proměnná je při volání metody použita. Samotný typ ref. proměnné není pro identifikaci metody rozhodující.

5.33. Co je to **dynamická identifikace metody** a jaký je její princip v Javě?
viz. 5.32

5.34. Jaký je účel a vlastnost **abstract** tříd a metod Javy?
definují zobecněné vlastnosti ve formě abstraktních metod, které budou moci podtřídy sdílet. Abstraktní metody nemají tělo, tzn. nejsou v abstraktní třídě implementovány. Podtřídy je musejí implementovat. Ostatní metody nadtřídy mohou zdědit, předefinovat nebo přetížit. Abstr. třídy se používají tehdy, když nadtřída nemůže vytvořit smysluplnou implementaci a určí tedy jenom šablonu. Třída obsahující alespoň jednu abstr. metodu musí být také abstraktní (opačně neplatí). Od abstr. třídy nelze vytvořit objekt.

5.35. Jaký je účel a vlastnost **final** tříd a metod Javy?
používá se v případech, kdy chceme vzhledem k důležitosti metody nebo třídy zabránit její modifikaci. Metodu označenou jako final nelze v podtřídách předefinovat. Od tříd označených jako final nelze oddědit žádné potomky. Označení metod jako final nevynucuje označení celé třídy jako final. Final u proměnné znemožňuje její modifikaci (lze jí pouze přiřadit počáteční hodnotu = de facto konstanta).

5.36. Popište vlastnost třídy **Object** jazyka Java.
je implicitní nadtřídou všech ostatních tříd. Proměnná typu Object může odkazovat na objekt kterékoliv třídy, na libovolné pole apod. Třída Object obsahuje základní metody, které mají všechny třídy (clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait).

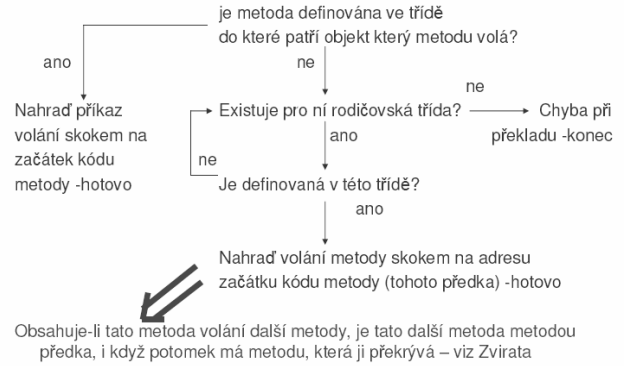
5.37. Jakými konstrukcemi lze způsobit v Javě **brzkou vazbu** metody s objektem který ji volá?
metoda musí být označena jako static, poté se jedná o statickou (brzkou) vazbu

5.38. Jaké jsou základní vlastnosti **konstruktoru v C++**?
jmenují se stejně jako třída, inicializují datové segmenty, implicitně volatelné, může jich být více, nevytváří objekt (narozdíl od Javy), pouze inicializují proměnné

5.39. K čemu slouží v **C++ destruktory**?
slouží k uvolnění paměťového místa dynamicky vytvořeného objektu (C++ nemá GC)

5.40. **Porovnejte objektové vlastnosti jazyků** Java, C++, Pascal
Java: jednoduchá dědičnost, GC, rozhraní, stat. proměnné, podpora paralelních výpočtů,
C++: vícenásobná dědičnost, nemá GC, generické typy, stat. proměnné, přetěžování operátorů
Pascal: jednoduchá dědičnost, nemá GC, rozhraní, podpora paralelních výpočtů, variantní typy

5.41. Objasněte **způsob zjišťování adresy začátku staticky** vázané metody v OOP.



5.42. Objasněte **způsob zpracování virtuálních metod** (při překladu a výpočtu).
při překladu se vytváří pro každou třídu tzv. datový segment obsahující *údaj o velikosti instance a datových složkách, *o předku třídy a *ukazatele na kód metod s pozdní vazbou (TVM).
Před prvním voláním metody musí být v dané instanci zavolána (případně implicitně) speciální inicializační metoda – constructor, který vytvoří za běhu programu spojení mezi instancí volající konstruktor a TVM. Součástí instance je místo pro ukazatel na TVM třídy, ke které patří instance. Volání metody je realizováno nepřímým skokem přes TVM. Pokud není znám typ instance při překladu, umožní TVM polymorfní chování.

5.43. Jaké funkce plní **konstruktor** v Javě, v C++ a v Pascalu?
Java: inicializace členských dat a vytvoření objektu
C++: inicializuje členská data, ale nevytváří žádné objekty. Může alokovat paměť, pokud je část objektu dynamická na haldě
Pascal:

5.44. Co je obsaženo v **Class Instant Record**?
údaj o velikosti instance a datových složek, údaj o předku třídy, ukazatele na kód metod s pozdní vazbou

5.45. Popište možnosti a důsledky použití **konstrukce friend** v C++.
specifikuje název funkce, která může pracovat se soukromými prvky objektu.

5.46. Popište jaké **problémy** vznikají **při násobné dědičnosti** a jak je řeší C++.
složitost a nepřehlednost
konflikt jmen – řešení: k položkám se přistupuje pomocí plně kvalifikovaných jmen
opakovaná dědičnost – řešení: použití virtuálních nadtříd
pořadí volání konstruktorů a destruktorů

5.47. Základní vlastnosti Java **rozhraní**.
částečně nahrazuje násobnou dědičnost, je obdobou abstraktní třídy
- definuje jen hlavičky metod, všechny bez implementace
- nemůže deklarovat žádné proměnné
- třída může implementovat (zdědit) více než jedno rozhraní
- nepříbuzné třídy mohou implementovat stejné rozhraní, tj. rozhraní nevynucuje příbuzenské vztahy
- rozhraní může dědit jiné rozhraní pomocí extends

5.48. Charakterizujte **situace**, kdy je **vhodné využít** Java **rozhraní**.
pokud třídy nemohou mít společného předka a mají vykonávat podobné funkce
pokud je nutno „obejít" vícenásobnou dědičnost

5.49. Zapište o**becný tvar deklarace rozhraní** a způsob jeho **implementace**.
Deklarace:

```
interface Jméno {
  hlavicka metody1;                    //pouze jmeno, typ a pocet parametru
  typ jmenopromenne_1 = hodnota;  //implicitne je public,final,static
  hlavicka metody2;
  . . .
  hlavicka metodyn;
  typ jmenopromenne_m = hodnota;  //neni to promenna instance
}
```

Implementace:

```
class Trida [extends Nadtrida] implements Jméno {
tělo Třídy          //včetně implementace všech metod rozhraní
                    //metody musí být deklarovány jako veřejné
}
```

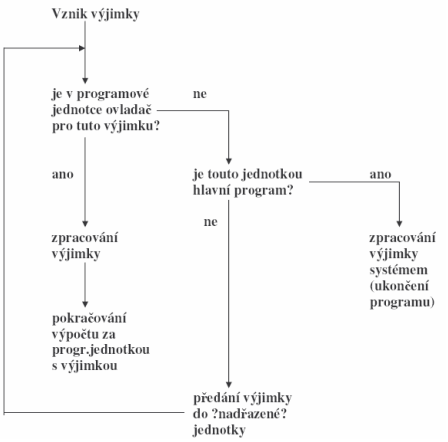
Při implementaci z více rozhraní se uvede Jméno1, Jméno2, Jméno_n

5.50. Vlastnosti **referenční proměnné typu rozhraní**.
může odkazovat na jakýkoliv objekt implementující její rozhraní, lze pomocí ní přistupovat pouze k těm metodám instance, které deklarace rozhraní definuje. Volání metody pomocí proměnné referující na rozhraní způsobí realizaci verze metody spojené s objektem.

5.51. Využitelnost operátoru **instanceof** v Javě.
syntaxe: ObjektReferencnihoTypu instanceof NejakyReferencniTyp
vrací booleovskou hodnotu, zda je první operand přetypovatelný na druhý operand

6. Zpracování výjimečných situací

6.1. Definujte základní **druhy výjimečných situací**.
Výjimečná situace je neobvyklou událostí (chybovou či nechybovou), která je detekovatelná hardwarem či softwarem a vyžaduje speciální zpracování. Druhy výjimek: vestavěné a uživatelské.



6.2. Vysvětlete **mechanismus propagace výjimky**.

6.3. Popište **způsob**, kterým jsou **zpracovávány výjimečné** situace v jazyce **C++**.
tvar: try {programový segment, ve kterém vzniká výjimka}
catch (formální parametr) {příkazy ovladače}
catch (formální parametr) {příkazy ovladače}
catch je jméno všech ovladačů, rozlišují se form. parametrem. Nemusí jím být proměnná, může jím být jméno typu. Form. par. lze použít k přenosu informace do ovladače, catch(...) chytá všechny výjimky. Neošetřené výjimky se propagují do místa volání funkce, ve které výjimka vznikla. Propagace může pokračovat až do funkce main. Pokud ani tam není ovladač nalezen, program je ukončen. Po provedení příkazů ovladače je řízení přeneseno na příkaz za posledním z ovladačů (z nichž jeden výjimku zpracoval). Všechny výjimky jsou uživatelské, vyvolávají se pouze explicitně: throw[výraz];

6.4. Zapište tvar **ovladače**, který v **C++** zachytává všechny vyhození výjimky.
try {programový segment}
catch(...) {příkazy ovladače}

6.5. Kde lze v **C++** použít příkazu **throw bez operandu**?
pouze v ovladači, kde způsobí znovuvyvolání výjimky a její zpracování ve vyšší úrovni.

6.6. Popište **hierarchii tříd výjimek v Javě** a jejich základní vlastnosti.
všechny výjimky jsou objekty tříd, které jsou potomky třídy Throwable. Knihovna Javy obsahuje dvě podtřídy Throwable: *__error__ – výjimky této třídy jsou vyvolávány Java interpretem, jejich zpracování uživateli nepřísluší (např. přetečení haldy). *__exception__ – uživatelské výjimky, má dvě podtřídy: IOException a RuntimeException (vznikají když program obsahuje chyby – např. ArrayIndexOutOfBoundException).

6.7. Zapište **v Javě obecný tvar vytvoření objektu typu výjimka**, jeho volání a definice jeho třídy.
definice třídy:
```
class MyException extends Exception {
    public MyException() {}
    public MyException(String message) {super(message)}
}
```
vytvoření objektu: MyException myExcept = new MyException(„hlaska“);
vyhození výjimky: throw myExcept;

6.8. Charakterizujte **kontrolované a nekontrolované výjimky Javy** a možnosti jejich zpracování.
nekontrolované (unchecked) – patří sem výjimky třídy Error a RunTimeException
kontrolované (checked) – všechny ostatní výjimky. Metoda, která tyto výjimky vyvolává, je musí mít v seznamu throws nebo musí mít v sobě ovladač.

6.9. Popište vlastnosti a způsoby použití klauzule **finally** v Javě.
slouží pro „úklid“ bez ohledu na to, zda výjimka nastane nebo nikoliv. Uvádí se za klauzulemi catch.

6.10. Jaké zásady v Javě platí při použití **supertříd a podtříd výjimek** v ovladačích?
klauzule catch pro nadtřídu se vztahuje také na všechny její podtřídy. Chceme-li tedy zachytit výjimku podtřídy, musíme její catch uvést dříve než catch její nadtřídy. Opačný zápis by způsobil nedosaž. kód.

6.11. Jaké zásady platí v Javě při **vnořování příkazů try**?
do try bloku lze vnořovat další try bloky. Výjimky z vnitřního try nezachycené v tomto úseku budou propagovány do dynamicky nadřazené jednotky.

6.12. Vysvětlete důvod použití a zásady vytvoření seznamu **throws** v Javě.
některé výjimky, pokud je metoda nemůže zpracovat, musí vyjmenovat v throws seznamu a tím jasně deklarovat, že je bude vyhazovat (propagovat) dále. Do seznamu throws není třeba uvádět výjimky odvozené z RuntimeError a Error (tzv. nekontrolované neboli unchecked exception). Tímto se vynucuje např. ošetření I/O operací.

6.13. Popište způsob jak lze v Javě **zavést vlastní programátorovy výjimky**.
programátor může definovat vlastní výjimky jako podtřídy Exception. Třída Exception nemá žádné vlastní metody, dědí ale metody svého rodiče Throwable. viz 6.7.

6.14. Jaký je **obecný tvar pro zavedení ovladače výjimky v jazyce Ada**?
```
begin
  příkazy
exception
  when výjimka1 => příkazy ...
  when výjimkaN => příkazy
end;
```

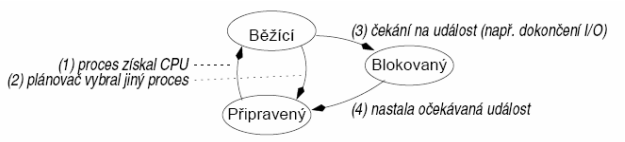
6.15. Porovnejte **možnosti zpracování vestavěných a uživatelských výjimek v C++, Javě a Adě**.
C++: není blok finally, všechny výjimky jsou uživatelské
Java: try-catch-finally, hierarchie výjimek – viz 6.6.
Ada: není blok finally

7. Paralelní programování

7.1. Popište rozdíl mezi **fyzickým, logickým** paralelismem a **kvaziparalelismem**.
fyzický – více procesorů pro více procesů
logický – time-sharing jednoho procesoru, v programu je více procesů
kvaziparalelismus – korutiny – speciální druh podprogramů, kdy volající a volaný jsou si rovni (symetrie), mají více vstupních bodů a zachovávají svůj stav mezi aktivacemi.

7.2. Jaké **problémy** vznikají **v paralelně prováděných výpočtech** + uveďte příklad.
uvíznutí (deadlock) – nastává např. když dva procesy soupeří o zdroj (přístup k DB)
vyhladovění (starvation) – „zacyklení“!
časová závislost

7.3. Principy možných **způsobů komunikace procesů**.
sdílené nelokální proměnné, parametry, zasílání zpráv



7.4. V jakých **stavech** se může nacházet **proces** a jaké jsou důvody **přechodů** mezi stavy?

7.5. Princip **semaforu**.
datová struktura obsahující čítač a frontu pro ukládání deskriptorů úkolů. Má dvě atomické operace – zaber a uvolni (P a V). Nebezpečí semaforu: deadlock.

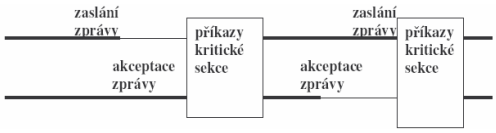
```
P(semafor)      /* binární*/
if semafor = 1 then semafor := 0
               else pozastav volající proces a dej ho do fronty na semafor

V(semafor)      /* binární*/
if fronta na semafor je prázdná then semafor := 1
               else vyber prvého z fronty a aktivuj ho
```

7.6.

7.7. Princip **monitoru**.
pgmový modul zapouzdřující data spolu

s procedurami, které s daty pracují. Procedury mají vlastnost, že vždy jen jeden úkol (vlákno) může provádět monitorovou proceduru, ostatní čekají ve frontě.

7.8. Princip **synchronizace pomocí předávání zpráv**.



7.9.

7.10. **Kritické sekce** programu + uveďte příklad.
jedná se o řešení problému sdílení zdrojů formou vzájemného vyloučení současného přístupu
metoda s označením synchronized uzamkne objekt pro který je volána. Jiná vlákna pokoušející se použít synchr. metodu uzamčeného objektu musí čekat ve frontě.

Když proces opustí synchr. metodu, objekt se odemkne. Příklad:

```
class BankovniUcet {
  public synchronized int vyber() {}
  public synchronized int vloz(int castka) {}
}
```

7.11. Definujte v Javě **třídu**, jejíž objekty mohou obsahovat **paralelně prováděné metody**.

```
class MojeVlakno extends Thread {                    // definice tridy
   public void run() {...}
}
MojeVlakno v = new MojeVlakno();                     // vytvoreni objektu
v.start();                                                    // spusteni vlakna
```

7.12. Jmenujte **základní metody třídy Thread a rozhraní Runnable**.
Thread: final String getName(), final int getPriority(), final void setPriority(), final boolean isAlive(), void run(), void start(), static void sleep(long milisec)
Runnable: void run()

7.13. Proč Java zavádí možnost odvozovat objekty s vlákny **implementací rozhraní Runnable**?

7.14. Jakým způsobem lze v Javě **ovlivnit prioritu provádění vlákna** + uveďte příklad.
metodou void setPriority(int priorita) , kde priorita má hodnotu 1 – 10
příklad: vlakno.setPriority(8);

7.15. V jakých **stavech** se může **nacházet vlákno Javy**?
nové vlákno (new thread) – stav, kdy je vlákno vytvořeno, ale ještě nebylo spuštěno
běhuschopné (runnable) – stav po spuštění metodou start(). V tomto stavu se může nacházet více spuštěných vláken, z nichž jen jedno je právě běžící.
neběhuschopné (not runnable) – do tohoto stavu se dostane, pokud je uspáno metodou sleep(), je odstaveno metodou suspend() nebo čeká na I/O zařízení
mrtvé (dead) – stav po ukončení metody run() nebo po zavolání stop()

7.16. Zapište příkaz, který zjistí, zda **vlákno** v1 **běží**.
v1.isAllive()

7.17. Zapište příkaz, který způsobí **pokračování výpočtu** vlákna v2 **po skončení činnosti** vlákna v1.
v1.join()

7.18. Jakým příkazem a jakým mechanismem **dává vlákno najevo**, že **čeká na skončení vlákna** v1?
v1.interrupt()

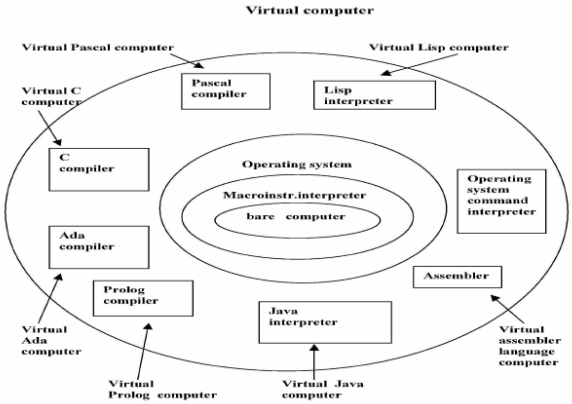
7.19. Vysvětlete způsob chování **synchronized** metod Javy.
viz 7.10

7.20. Jaký je rozdíl v efektu příkazu **yield**(), **sleep**(200) a **wait**(200)?
yield() – vzdá se zbytku přiřazeného času a zařadí se na konec fronty
sleep(200) – zablokuje vlákno na dobu 200 ms
wait(200) – zablokuje vlákno na dobu 200ms, vlákno lze předčasně probudit pomocí notify()

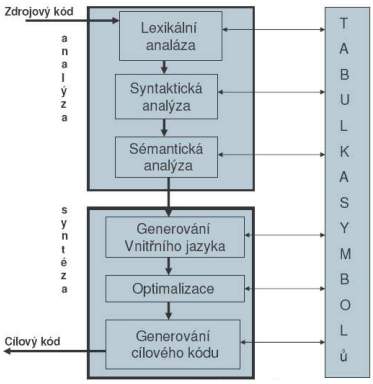
7.21. Popište účel a způsob použití příkazu **notify() a notifyAll()**.
notify() – oživí vlákno z čela fronty na objekt
notifyAll() – oživí všechna vlákna nárokující si přístup k objektu

7.22. Charakterizujte **SIMD** a **MIMD** architekturu.
SIMD – stejná instrukce současně zpracovávána na více procesorech, na každém s jinými daty – vektorové procesory
MIMD – nezávisle pracující procesory, které mohou být synchronizovány

8. Úvod do překladačů



8.1. Popište jednotlivé **vrstvy virtuálního počítače**.



8.2.

8.3. Hlavní části **kompilačního** a **interpretačního překladače**.

8.4. Co je výsledkem **lexikálního analyzátoru**?
vnitřní jazyk lexikálního analyzátoru (mezijazyk), což je číselný kód lexikálních elementů (komentáře jsou vynechány). Výhoda: pevná délka symbolů pro další fáze zpracování

8.5. Co je výsledkem činnosti **syntaktického analyzátoru**? Uveďte možnou formu výstupu SA.
posloupnost použitých gramatických pravidel při odvození tvaru programu z počátečního symbolu gramatiky. Příklad: 1,2,4,5,7,10,12,10,13, . . .

8.6. Jmenujte nejužívanější **vnitřní jazyky překladače** a jejich charakteristiky.
postfixová notace – operátor následuje za operandy (LD a, LD b, PLUS)
prefixová notace – operátor je před operandy (PLUS LD a, LD b)

víceadresová instrukce (PLUS a b vysledek)

- Objasněte rozdíl mezi „klíčovými slovy „ a „předdefinovanými slovy“
  Klíčová slova = v určitém kontextu mají speciální význam
  Předdefinovaná slova = identifikátory speciálního významu, které lze
  předefinovat (např vše z balíku java.lang – String, Object, Systém…)
  –Rezervovaná slova = nemohou být použita jako uživatelem definovaná jména
  (např.abstract, boolean, break, …, if, …, while)
- Objasněte rozdíl mezi dobou existence a rozsahem platnosti proměnné
  Rozsah platnosti (scope) proměnné je částí programového textu, ve kterém je proměnná viditelná. Pravidla viditelnosti určují, jak jsou jména asociována s proměnnými.
  Rozsah existence (lifetime) je čas, po který je proměnná vázána na určité paměťové místo.
- Uveďte příčiny vzniku synonym (alias) v programech
  Alias – dvě proměnné sdílí ve stejné době stejné místo (špatnost)
  Pointery, Referenční proměnné, Variantní záznamy (Pascal), Uniony (C, C++), Fortran (EQUIVALENCE), Parametry podprogramů
- Popište princip statické a dynamické vazby jména proměnné s typem
  Statická vazba (jména s typem/s adresou) - navázání se provede před dobou výpočtu a po celou exekuci se nemění. Vazba s typem určena buď explicitní deklarací nebo implicitní deklarací.
  Dynamická vazba (jména s typem / s adresou) nastane během výpočtu nebo se může při exekuci měnit
  – Dynamická vazba s typem - specifikována přiřazováním (např. Lisp), výhoda – flexibilita (např. generické jednotky), nevýhoda - vysoké náklady + obtížná detekce chyb při překladu.
  – Vazba s pamětí (nastane alokací z volné paměti, končí dealokací), doba existence proměnné (lifetime) je čas, po který je vázána na určité, paměťové místo.
- Popište princip výhody a nevýhody statické a dynamické vazby proměnné s adresou – výše
  Statické = navázání na paměť před exekucí a nemění se po celou exekuci (Fortran 77, C static)
  výhody: efektivní – přímé adresování, podpr. senzitivní na historii, nevýhody: bez rekurze
  Dynamické:
  V zásobníku = přidělení paměti při zpracování deklarací. Pro skalární proměnnou jsou kromě adresy přiděleny atributy staticky (lokální prom. C, Pascalu). výhody: rekurze, nevýhody: režie s alokací/dealokací, ztrácí historickou informaci, neefektivní přístup na proměnné (nepřímé adresy)
  Explicitní na haldě = přidělení / uvolnění direktivou v programu během výpočtu. Zpřístupnění pointery nebo odkazy (objekty ovládané new/delete v C++, objekty Javy), výhody: umožňují plně dynamické přidělování paměti, nevýhody: neefektivní a nespolehlivé (zejm. při slabším typovém systému)
  Implicitní přidělování na haldě = alokace/dealokace způsobena přiřazením (proměnné APL,
  Výhody: flexibilita, nevýhody: neefektivní – všechny atributy jsou dynamické špatná detekce chyb
- Popište rozdíl mezi statickým a dynamickým rozsahem platnosti proměnné
  V jazycích se statickým rozsahem platnosti jsou referenčním prostředím jména lokálních proměnných a nezakrytých proměnných obklopujících jednotek
  V jazycích s dynamickým rozsahem platnosti jsou referenčním prostředím jména lokálních proměnných a nezakrytých proměnných aktivních jednotek
- Definujte pojem silný typový systém programovacího jazyka
  Programovací jazyk má silný typový systém, pokud typová kontrola odhalí veškeré typové chyby
- Definujte pojmy strukturální a jmenná kompatibilita typů
  Jmenná kompatibilita – dvě proměnné jsou kompatibilních typů, pokud jsou uvedeny v téže deklaraci, nebo v deklaracích používajících stejného jména typu. dobře implementovatelná, silně restriktivní
  Strukturální kompatibilita – dvě proměnné jsou kompatibilní mají-li jejich typy identickou strukturu
  flexibilnější, hůře implementovatelné. (Java, Ada)
- Jaké rozlišujeme konstanty podle doby jejich určení
  Určené v době překladu– př.Javy: static final int zero = 0;
  Určené v době zavádění programu: static final Date now = new Date();
- Definujte pojmy „literál“ a „manifestová konstanta“
  Literály = konstanty, které nemají jméno; Manifestová konstanta = jméno pro literál
- Jaké typy označujeme jako ordinální
  (zobrazitelné/přečíslitelné do integer), primitivní mimo float, definované uživatelem (pro čitelnost a spolehlivost programu)
  -**vyjmenované typy** =uživatel vyjmenuje posloupnost hodnot typu, Implementují se jako seznam pojmenovaných integer konstant,
  -**typ interval** =souvislá část ordinálního typu. Implementují se jako typ rodiče.
- Popište jaké typy označujeme jako uniony
  typy, jejichž proměnné mohou obsahovat v různých okamžicích výpočtu hodnoty různých typů.
- Uveďte, jakým způsobem vzniká dangling pointer
  Špatnost ukazatelů = dangling (neurčená hodnota) pointers a ztracené proměnné, new(P1); P2 := P1; dispose(P1**);**
- Uveďte, jakým způsobem vzniká ztracená proměnná z haldy
- Popište pojmy precedence, asociativita a arita operátorů ve výrazech
  arita – počet operandů
- Jaká jsou pozitiva a negativa příkazů skoku
  Nevýhody - znepřehledňuje program, je nebezpečný, znemožňuje formální verifikaci programu
  Výhody - snadno a efektivně implementovatelný
- Které vlastnosti jsou důležité pro příkazy cyklů v programovacích jazycích
- Jmenujte kriteria, dle kterých lze hodnotit vlastnosti podprogramů programovacích jazyků
  Způsob předávání parametrů
  Možnost typové kontroly parametrů
  Jsou lokální proměnné umisťovány staticky nebo dynamicky?
  Jaké je platné prostředí pro předávané parametry, které jsou typu podprogram
  Je povoleno vnořování podprogramů
  Mohou být podprogramy přetíženy (různé pp mají stejné jméno)
  Mohou být podprogramy generické
  Je dovolena separátní kompilace podprogramů
- K čemu slouží aktivační záznamy podprogramů
  Místo pro lokální proměnné, předávané parametry, funkční hodnotu u funkcí. Návratová adresa, informace o uspořádání aktivačních záznamů, místo pro dočasné proměnné při vyhodnocování výrazů.
  Umožňuje vnořování rozsahových jednotek
  Umožňuje rekurzivní vyvolání