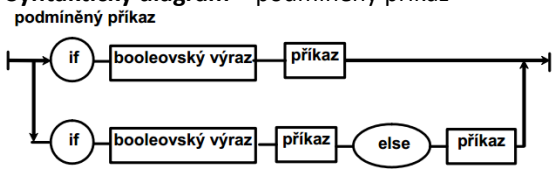
1. **Charakterizujte:**
    a. **procedurální programován** (jako kuchařka) popisuje výpočet pomocí posloupnosti příkazů a určuje přímý postup (algoritmus), jak úlohu řešit
    b. **OOP** – program je množina objektů. Objekty mají stav, jméno, chování. Předávají si zprávy. Zapouzdřenost, dědičnost, polymorfismus.
    c. **generické programování** – rozdělení programu na algoritmus a datové typy tak, aby zápis algoritmu byl obecný bez ohledu na datový typ. Konkrétní algoritmus se z něj stává dosazením datového typu (C++, šablony)
    d. **komponentově orientované programování** – souvisí s komponentově orientovaným softwarovým inženýrstvím. Využívá prefabrikované komponenty (násobná použitelnost, nezávislost na kontextu). Soft. komponenta – jednotka systému, která nabízí předem definovanou službu a je schopna komunikovat s ostatními komponentami.
    e. **deklarativní programování** – opak procedurálního. Založeno na popisu cíle (přesný alg. udělá až interpret). K předání hodnot slouží návratové hodnoty fcí. Nemožné moc předělávat. Využívá rekurze -> větší nároky na programátora.
    f. **programování ohraničením** – vyjadřuje počet pomocí relací mezi proměnnými. Př.: celsius = (far-32)*5/9 definuje relaci, ne přiřazení. Forma deklarativního prog. Modeluje svět, ve kterém je množství ohraničení splněno v jednu chvíli, svět obsahuje neznámé proměnné. Úkol: najít jejich hodnoty.
    g. **logické programování** – použití mat. logiky v programování. Rozdělení řešení problému mezi: 1) programátora – zodpovědný za pravdivost programu vyjádřeného log. formulemi, 2) generátor modelu – zodpovědný za efektivní vyřešení problému. Využití v UI, zpracování řeči. Zástupce – PROLOG.
    h. **funkcionální programování** – výpočet řízen vyhodnocováním matematických funkcí. Funkcionální (aplikace funkcí) vs. imperativní (změna stavu). Zástupce – LISP. Využití dříve v UI.
    i. **událostní programování** – tok programu je určen akcemi uživatele nebo zprávami od jiných programů.
    j. **vizuální programování** – specifikuje program interaktivně pomocí grafických prvků. Microsoft VS (VB, C#) nejsou viz. jazyky!! „Boxes and arrows“ – boxy reprezentují entity a šipky relace.
    k. **souběžné programování** – program je navržen jako kolekce interagujících procesů. Paralelní x distribuované. Jeden procesor x více procesorů. Komunikace: sdílené paměť x předávání zpráv. Použité konstrukce: multithreading, podpora diatrib. zpracování, předávání zpráv, sdílení zdrojů.
2. **Jaké jsou globální kritéria na programovací jazyk** – spolehlivost, efektivita – překladu, výpočtu, strojová nezávislost, čitelnost a vyjadřovací schopnosti, řádně definovaná syntax a sémantika, úplnost v Turingově smyslu
3. **Jaká hlediska ovlivňují:**
    a. **Spolehlivost** – typová kontrola : typované (specifikace operace definuje typy dat – chyba při dělení čísla řetězcem) x netypované (všechny op. na jakýmikoliv daty), statické (výrazy mají určený typ, před spuštěním prog.) x dynamické (určuje typovou bezpečnost při běhu programu), slabé (dovoluje nakládat s jedním dat. typem jako s jiným) x silné (nedovoluje)
    b. **Čitelnost a vyjadřovací schopnosti** – jednoduchost (C=C+1, C++), ortogonalita (malá množina primitiv. konstrukcí, lze z nich kombinovat další, všechny jsou legální), strukturované příkazy a datové typy, strojová čitelnost (= exist. alg. překladu s lineární časovou složitostí, = bezkontextová syntax)
    c. **Řádně definovaná syntax a sémantika** – syntax (= forma či struktura výrazů, příkazů a programových jednotek), sémantika (= význam)
4. **Definovat syntax a sémantika**
    a. **Syntax** – formálně je jazyk množinou vět. Věta je řetězcem lexémů (terminálních symbolů). Syntax lze popsat: rozpoznávacím mechanismem – automatem, generačním mechanismem – gramatikou. Backus-Neurova forma (BNF) – metajazyk používaný ve vyjádření bezkontextové gramatiky. Derivační strom – vyjádření resoluční metody.
    b. **Sémantika** – studuje a popisuje význam výrazů/programů. Aplikace mat. logiky. Statická (v době překladu, nedefinovaná proměnná, chyba v typech) x dynamická (v době běhu, dělení nulou)
5. **Úplnost v Turingově smyslu** – Turingův stroj (= jednoduchý, ale neefektivní počítač použitelný jako formální prostředek k popisu algoritmu). Prog. jazyk je úplný v TS, jestliže je schopný popsat libovolný výpočet. Potřeby: celočíselná aritmetika a proměnné sekvenčně prováděnými příkazy zahrnujícími přiřazení a cyklus (while)
6. **Tvar příkazu BFN** - podmíněný příkaz

    <podmíněný příkaz> = if <booleovský výraz>
    <příkaz>|
    if <booleovský výraz> <příkaz> else <příkaz>

7. **Syntaktický diagram** – podmíněný příkaz



8. **Objasněte pojmy statická a dynamická sémantika** – sémantika studuje a popisuje význam výrazu/programu. Statická (v době překladu, nedefinovaná proměnná, chyba v typech) x dynamická (v době běhu, dělení nulou)
9. **Jaké druhy chyb rozlišujeme?**
   Lexikální (např. nedovolený znak), syntaktické (chyba ve struktuře), sémantické (statické – nedef. proměnná, chyba v typech, dynamické – nastávají při běhu, dělení nulou), logické (chyba v algoritmu)
10. **Druhy chyb a fáze nalezení překladačem**
    a. kompilátor je schopen nalézt lexikální a syntaktické chyby při překladu
    b. statická sémantika – před výpočtem
    c. dynamická sémantika – při výpočtu
    d. logické chyby – nejsou nahlásit
11. **Amdahlův zákon** – určuje urychlení výpočtu při užití více procesorů Urychlení je limitováno sekvenčními částmi výpočtu.

    Obecně: $\frac{1}{\sum_{k=0..n} \frac{P_k}{S_k}}$

    P_k % instrukcí, které lze urychlit
    S_k multiplikátor urychlení
    n počet různých úseků programu
    k je index úseku
12. **Zjistěte možné urychlení výpočtu při zadaných údajích o úsecích a jejich urychlení**
    Př. úseky
    P1 = 11 %, P2 = 48 % P3 = 23 % P4 = 18 %
    S1 = 1 S2 = 1,6 S3 = 20 S4 = 5

    Urychlení je 1 / (0,11/1 + 0,48/1,6 + 0,23/20 + 0,18/5) ≈ **2,19**
13. **Popište rozdíl mezi fyzickým paralelismem, logickým paralelismem a kvaziparalelismem**
    a. **Fyzický** – má více procesorů pro více procesů
    b. **Logický** – sdílení času jednoho procesoru, v programu více procesů
    c. **Kvaziparalelismus** – zdánlivý paralelismus, např. korutiny – speciální druh podprogramů, kdy volající a volaný jsou si rovni (symetrie), mají více vstupních bodů a zachovávají svůj stav mezi aktivacemi.
14. Problémy při paralelním zpracování - rychlostní závislost, uvíznutí (vzájemné neuvolnění prostředků pro jiného), vyhladovění (obdržení příliš krátkého času k zajištění progresu), livelock (obdoba uvíznutí, ale nejsou blokovány čekáním, zaměstnávají se navzájem)
15. **Způsoby komunikace**
    a. přes sdílenou paměť (Java, C#) – musí se zamykat přístup k paměti
    b. předáváním zpráv (Occam, Ada) – vyžaduje potvrzení o přijetí zprávy
16. **V jakých stavech se může nacházet proces a jaké jsou důvody přechodů mezi stavy**
    a. Stav probihajici (running) - procesu je přidělen procesor a právě se provádí příslušné programy
    b. Stav čekající (waiting) – proces čeká na určitou událost, např. ukončení I/O operace
    c. Stav připraven (ready) proces je připraven k vykonání a čeká na přidělení procesoru

    Tyto 3 hlavni procesy nestačí pro úplný popis pohybu úlohy v OS. Další jsou:

    a. Stav předaná (submit)
    b. Stav přijatá (hold)
    c. Stav ukončená (komplete)
17. **Semafor** = datová struktura obsahující čítač a frontu pro ukládání deskriptorů/úkolů/procesů/vláken. Má dvě atomické operace Zaber a Uvolni (P a V). Je použitelný jak pro soutěžící, tak pro spolupracující úkoly. Nebezpečnost – může nastat dreadlock, rychlostní závislost.
18. **Monitor** = modul zapouzdřující data spolu s procedurami, které s daty pracují. Procedury mají vlastnost, že vždy jen jeden úkol/vlákno může provádět monitorovanou činnost, ostatní čekají ve frontě
19. **Popište princip synchronizace pomocí předávání zpráv**
    Proces předá zprávu druhému a druhý ji ale musí očekávat, Randes-vous v ADě, příklad hospoda číšník-host, číšník čeká na objednávku
20. **Kritická sekce programu** se vyskytuje tam, kde dvě a více vláken pracuje nad stejnými daty.

Příklad: Máme dvě metody *set()* a *get()* a dvě vlákna. Jedno vlákno nastavuje nějaké hodnoty v metodě *set()*, vlákno druhé získává hodnoty z metody *get()*. Pokud by druhé vlákno získávalo hodnoty právě ve chvíli, kdy vlákno první hodnoty zapisuje, získáme data, kde budou nějaké hodnoty nové (již přepsané) a nějaké staré (ještě nepřepsané). A to je špatně.
Řešení: Pomocí monitoru (v Javě to znamená použití *synchronized* metod nebo bloků).

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

```
public class Vlakno extends Thread {
    public void run() {
       //příkazy
    }
}
```

22. **Jmenujte základní metody a) třídy Thread, b) rozhraní Runnable**
    a. **Thread** – run (udává činnost vláken), start (spustí vlákno), yield (odevzdá zbytek přiděleného času), sleep(milisec) (zablokování vlákna), isAlive (životnost), join (počká na skončení vlákna), getPriority, setPriority
    b. **Runneable** – run

Když uživatelova vlákna nepřepisují ostatní metody (musí přepsat jen run), upřednostňuje se Runnable.

23. **Proč Java zavádí možnost odvozovat objekty s vlákny implementací rozhranní Runnable**
    Hodně tříd potřebuje dědit jinou třídu. Není umožněna vícenásobná dědičnost -> Runneable. Je vhodné třídy rozšiřovat jen za účelem modifikace.
24. **Akce při vytváření a spuštění vlákna**
    a. **Thread** – vytvoříme třídu MyThread, která podědí od Thread. Překrejeme metodu run() , vytvoříme instanci a spustíme metodou start()
    b. **Runneable** – vytvoříme třídu MyThred která implementuje a překryjeme run(). Vytvoříme instanci třídy a z ní vlákno. Spustíme start().
25. **Jakým způsobem lze v Javě ovlivnit prioritu provádění vlákna. Uveďte příklad**
    Priorita = pravděpodobnost častosti získání času procesoru
    Vysoká priorita – hodně času, nízká priorita – málo času
    potomek má stejnou prioritu jako nadřízený proces
    lze měnit metodou setPriority(int cislo), kde číslo je v intervalu <1, 10>
26. **V jakých stavech se může nacházet vlákno Javy**
    nové (ještě nezačalo běžet), připravené (nemá přidělený procesor), běžící (má přidělený procesor), blokované (čeká ve frontě na monitor), mrtvé
27. **Zapište příkaz, který zjistí zda vlákno v1 běží**

```
if(v1.isAlive())  //vrací true, pokud vlákno není mrtvé, ale vlákno nemusí být právě prováděné
```

28. **Zapište příkaz, pozastavující vlákno v2, dokud neskončí činnost vlákno v1**

```
v2.yield(); // vlákno v2 předá řízení vláknu se stejnou prioritou (takže v1)
v1.join(); // dokud vlákno v1 nedoběhne, jiná vlákna nemohou běžet
```

29. **Jakým příkazem a jakým mechanismem dává vlákno najevo, že čeká na skončení vlákna v1**
    v1.interupt(); //oznamuje v1, že na něj čekáme, předčasně ho probudí
30. **Vysvětlete způsob chování „synchronized“ metod Javy**
    Uzamkne objekt, pro který je volána, jiná vlákna pokoušející se o synchr. metodu musí čekat ve frontě. Slouží k ošetření kritických sekcí v programu.
31. **Jaký je rozdíl v efektu příkazu Javy yield( ), sleep(200) a wait(200)**
    a. *yield()* - vlákno se vzdává své činnosti a dobrovolně přenechává procesor vláknu se **stejnou** prioritou
    b. *sleep(200)* - vlákno se uspí (převede do neběhuschopného stavu) na 200 ms, poté bude vlákno opět ve stavu běžícím
    c. *wait(200)* - vlákno čeká (převede se do neběhuschopného stavu) 200 ms, může být probuzeno (převedeno do běžícího nebo běhuschopného stavu) metodami *notify()* nebo *notifyAll()*
32. **Popište účel a způsob použití příkazu notify() a notifyAll()**
    a. **notify**( ) - oživí vlákno z čela fronty čekajících na přístup k objektu
    b. **notifyAll**( ) - oživí všechna vlákna nárokující si přístup k objektu, ta pak o přístup normálně soutěží (na základě priority nebo plánovacího algoritmu JVM)

    Mohou být volány jen z vláken, které vlastní zámek (synchronized metod a příkazů), jsou děděny z třídy Object.
33. **Popište efekt metody Javy interrupt()**
    Metada *interrupt()* - metoda ze třídy **Thread**, přeruší běh vlákna.
    Pokud je vlákno v neběhuschopném stavu (např. uspané) a je na něj zavolána metoda *interrupt()* je "násilně probuzeno" a bude vyhozena výjimka **InterruptedException**

34. **Jaký je tvar a účinek synchronized příkazu Javy**
    a. **synchronized metody**

```
synchronized public void setHodnota(Object o) - hlavička synchronized metody
synchronized public void getHodnota() - hlavička synchronized metody
```

    Účinek: ošetření tzv. kritických sekcí, pokud je metoda synchronized pak má monitor a vlákno nikdy nemůže předat řízení uvnitř této metody.

b. **synchronized bloky**

```
synchronized(object) {
 //příkazy
}
```

Účinek: monitor je nad určitým objektem (zde object).

35. **Popište metody suspend, resume a stop Javy. Proč nejsou doporučované?**
    a. suspend( ) - pozastavení vlákna, kterému zašleme suspend
    b. resume ( ) - obnovení vlákna, kterému zašleme resume
    c. stop( ) - ukončení vlákna, kterému zašleme stop

    Důvod zavržení - nebezpečné konstrukce, které snadno způsobí deadlock, když se aplikují na objekt, který je právě v monitoru. Lze je nahradit bezpečnějšími konstrukcemi s wait a notify.

36. **Jaký je rozdíl mezi uživatelskými a démon vlákny?**
    a. **Uživatelská vlákna** - pokud program používá jen tato vlákna, nemůže skončit dříve, než jsou všechna vlákna ukončena
    b. **Démon vlákna** - pokud je v programu démon vlákno, program skončí bez ohledu na to, jestli toto vlákno doběhlo nebo ne

37. **Zapište konstrukci a) zjišťující, je-li vlákno démonem, b) určující, že má být démonem**

```
if (vlakno.isDaemon()) //zjištění, jestli je vlákno démon
vlakno.setDaemon(true);  // vlákno vlakno je nyní démon
```

38. **Zapište konstrukci, která začlení vlákno do skupiny vláken**

```
ThreadGroup mojeSkupina = new ThreadGroup(„jmeno“); // vytvoří skupinu
Thread mojeVlakno = new Thread(mojeSkupina, „jmeno“); // přiřadí ke skupině
```

39. **Jaké jsou důvody, pro využívání synchronizačních prostředků z balíku conccurent?**
    Vestavěná primitiva Javy nestačí k pohodlné synchronizaci protože:
    a. Neumožňují couvnout po pokusu o získání zámku, který je zabrán, a po vypršení času, po který je vlákno ochotno čekat na uvolnění zámku
    b. Nelze měnit sémantiku uzamčení
    c. Neřízený přístup k synchronizaci, každá metoda může použít blok synchronized na libovolný objekt
    d. Nelze získat zámek v jedné metodě a uvolnit ho v jiné.

40. **Popište základní prostředky třídy ReentrantLock z z balíku conccurent.**
    a. **Konstruktory -** instance s férovým chováním při true, nepředbíhá
        a. `ReentrantLock()`
        b. `ReentrantLock(boolean fair)`
    b. **Metody**
        a. `int getHoldCount() kolikrát aktuální vlákno žádalo o`
        b. `int getQueueLength() kolik vláken chce tento zámek`
        c. `protected Thread getOwner() vrátí vlákno, které vlastní zámek, nebo null`
        d. `boolean hasQueuedThread(Thread thread) čeká zadané vlákno na tento lock`
        e. `void lock() zabrání zámku, není-li volný musí čekat`
        f. `void unlock() uvolnění zámku`
        g. `boolean tryLock() zabrání je-li volný, jinak může dělat něco jiného`

41. **Popište základní prostředky třídy Semaphore z balíku conccurent**
    a. Konstruktory
        a. `Semaphore(int povoleni)`
        b. `Semaphore(int povoleni, boolean f)`
    b. Metody
        a. `acquire( ) získání jednoho povolení`
        b. `acquire(int povoleni) pro více povolení`
        c. `release() uvolní 1 povolení`
        d. `release(int povoleni) uvolní zadaný počet povolení`

42. **Popište základní prostředky třídy CyclicBarrier z balíku conccurent**
    Dovoluje čekání množiny vláken na sebe navzájem před pokračováním výpočtu. Nazývá se cyklická, protože může být znovu použita po uvolnění čekajících vláken.

a. Konstruktory
    a. `CyclicBarrier(int účastníci) účastníci určují počet podúloh = vláken`
    b. `CyclicBarrier(int účastníci, Runnable barierováAkce) akce se provede po spojení všech vláken, ale před jejich další exekucí`
b. Metody:
    a. `await( ) čeká, až všechni účastníci vyvolají await na této bariéře`
    b. `await(long timeout, TimeUnit unit) čeká, dokud buď všechny vyvolají await nebo nastane specifikovaný timeout`
    c. `getNumberWaiting( ) vrací počet čekajících na bariéru`
    d. `getParties( ) vrací počet účastníků procházejících touto bariérou isBroken( ) vrací`

43. **Popište základní prostředky třídy CountDouwnLatch z balíku conccurent**
Synchronizační prostředek, který dovoluje vláknu/vláknům čekat až se dokončí operace v jiných vláknech. Inicializuje se zadaným čítačem, který je součástí konstruktoru.
    a. Metody:
        a. `await( ) způsobí čekání volajícího vlákna až do vynulování čítače`
        b. `await(long timeout, TimeUnit unit) čekání se ukončí i vyčerpáním času.`
        c. `countDown( ) dekrementuje čítač a při 0 uvolní všechna čekající vlákna.`
        d. `getCount( ) vrací hodnotu čítače`
        e. `toString( ) vrací řetězec identifikující závoru a její stav (čítač)`
44. **Charakterizujte SIMD a MIMD architekturu**
    a. SIMD - stejná instrukce běží současně na více procesorech, na každém s jinými daty
    b. MIMD - nezávislé pracující procesory, mohou být synchronizovány
45. **Uveďte charakteristické vlastnosti skriptovacích jazyků**
Integrovaný překlad a výpočet, nízká režie a snadné použití, zduřelá funkčnost ve specif. oblastech (řetězce, reg. výrazy), není důležitá efektivita provedení (často se spustí jen jednou), nepřítomnost sekvence překlad-sestavení-zavedení
46. **Jmenujte oblasti použití skriptovacích jazyků**
    a. K vytváření aplikací z předpřipravených komponent
    b. K řízení aplikací, které mají programovatelný interface
    c. Ke psaní programů, je-li rychlost vývoje důležitější než efektivita výpočtu
    d. Jsou nástrojem i pro neprofesionální programátory
    e. Administrace systémů (sekvence shell příkazů ze souboru)
    f. Vzdálené řízení aplikací (dávkové jazyky)
47. **Jaké datové typy označuje Python jako sekvence, uveďte příklady zaváděných operací**
Sekvence jsou uspořádané množiny prvků. Zahrnují 8bitové řetězce, unikódové řetězce, seznamy a n-tice.
Všechny sekvence S sdílí společnou množinu funkcí: len(S) – počet prvků, max(S) - největší prvek, min(S) – nejmenší prvek, x in S – test přítomnosti x v S, tuple(S) – konverze členu S na n-tici, list(S) – konverze S na seznam, S*n – nová sekvence tvořená n kopiemi S
    a. `t = tuple('100') t = ('1', '0', '0')`
    b. `l = list('100') l = ['1', '0', '0']`
48. **Jednoduché příklady na příkazy Pythonu se seznamy**
    a. **append** - připojuje svůj argument na konec seznamu
    b. **extend(L)** - přidá na konec prvky seznamu L
    c. **insert(i, x)** - vloží prvek x na pozici i
    d. **remove(x)** - odstraní první výskyt prvku x v seznamu
    e. **pop(i)** - odstraní prvek na pozici i a vrátí jeho hodnotu
    f. **count(x)** - vrátí počet prvků s hodnotou x
    g. **sort()** seřadí prvky dle velikosti (modifikuje seznam), výsledek nevrací
    h. **reverse()** obrátí pořadí prvků
49. **Jednoduché příklady na příkazy Pythonu s množinami**

Množiny nepatří mezi sekvence, protože nejsou uspořádané. Jsou ale založeny na seznamech, prázdná množina je prázdný seznam. Musíme importovat „sets“.

a. `M=sets.Set() – provede převedení seznamu na množinu (N=sets.Set['a',2,3])`
b. `N.union(O) – sloučení množin`
c. `U.intersection(N) – průnik množin`
d. `U.issubset(N) – je podmnožinou`
e. `set('abracadabra') – vytvoří množinu z řetězce`

50. **Jednoduché příklady na příkazy Pythonu s n-ticemi**
N-tice Pythonu je posloupnost hodnot uzavřená do ( ), lze s ní nakládat jako s celkem. Po vytvoření nelze n-tici měnit.
```
ntice = (1, 2, 3, 4, 5)
len(ntice)
max(ntice)
min(ntice)
```
51. **Jednoduché příklady na příkazy Pythonu se slovníky**
Jsou to asociativní pole. Položky jsou zpřístupněny klíči, hodnotou může být libov. typ. Realizují se hash tabulkami.
```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
A.update(B) – spojí dva slovníky
a = b.copy() – nakopíruje obsah jednoho slovníku do druhého
a.clear() – vymaže slovník
a.items() – navrací seznam(key, value) všech položek slovníku
a.keys() – vrací seznam klíčů
a.values() – vrací seznam hodnot
a.popitem() – vrátí a vymaže libovolnou položku
```

52. **Popište hlavní odlišnosti OOP v Javě a Pythonu**
a. všechny metody jsou public
b. lze provést vícenásobné dědění
c. není definováno rozhraní
d. jsou definovány destruktory
53. **Jak řeší Python problém násobné dědičnosti**
Řešení: Není-li atribut v potomkovi, hledá python atribut v rodiči1, následně v rodiči rodiče 1. Až prohledá všechny rodiče rodiče 1, začne teprve hledat atribut v rodiči2. Tzn. hledá atributy do hloubky a pak následně z leva doprava.
```
class Potomek(Matka, Otec):
   def __init__(self):
       Otec.__init__(self)        # 1. konstruktor otce
       Matka.__init__(self)       # 2. konstruktor matky
   def popis(self):
       print self.oci, self.usi, self.ruce, self.nos, self.nohy
```

54. **Charakterizujte perzistentní objekty a možnost jejich vytváření**
Objekty, které mohou být uloženy a znovu použity jiným programem, "žijí" nezávisle na programu. Persistentní objekty se ukládají do souboru.
55. **K čemu slouží a jak funguje destruktor Pythonu**
Příkazy se provedou, když je objekt odstraňován z paměti. To nastane samovolně, když je čítač odkazů na objekt 0. Slouží k provedení příkazů těsně před odstraněním z paměti.
```
class Point:
   def __init__(self, x=0, y=0):
      self.x = x
      self.y = y
   def __del__(self): # destruktor
      class_name = self.__class__.__name__
      print class_name, "destroyed"
```

56. **Charakterizujte událostmi řízené programování**
a. Program po spuštění čeká v nekonečné smyčce na výskyty událostí.
b. Při výskytu události provede odpovídající akci a nadále čeká ve smyčce.
c. Skončí, až nastane konec indikující události
Události může generovat OS nebo vnější čidla.
57. **Jak lze s využitím Tkinter implementovat čekací smyčku**
Tkinter je modul pro Python, který umožňuje vytvářet okna.
```
top = Tk() #vytvoří widget (ovládací prvek) na nejvyšší úrovni
top.mainloop() #odstartuje provádění smyčky
```
58. **Jmenujte alespoň 6 grafických prvků Tkinter**
Tkinter je modul pro Python, který umožňuje vytvářet okna.
Frame, Label, Button, ScrollBar, Canvas, RadioButton, CheckButton

59. **Jaké vlastnosti má well-formed XML dokument**
    a. Jeden kořenový element
    b. Neprázdné elementy musí být ohraničeny startovací a ukončovací značkou (<ovoce>Jablko</ovoce>)
    c. Prázdné elementy mohou být označeny tagem „prázdný element“(<ovoce/>)
    d. Všechny hodnoty atributů musí být uzavřeny v uvozovkách – jednoduchých (') nebo dvojitých ("), ale jednoduchá uvozovka musí být uzavřena jednoduchou a dvojitá dvojitou. Opačný pár uvozovek může být použit uvnitř hodnot
    e. Elementy mohou být vnořeny, ale nemohou se překrývat; to znamená, že každý (ne kořenový) element musí být celý obsažen v jiném elementu
60. **Charakterizujte DTD** (Document Type Definition)
    a. jazyk pro popis struktury XML dokumentu
    b. Omezuje množinu přípustných dokumentů spadajících do daného typu nebo třídy
    c. Struktura třídy nebo typu dokumentu je v DTD popsána pomocí popisu jednotlivých elementů a atributů. Popisuje, jak mohou být značky navzájem uspořádány a vnořeny. Vymezuje atributy pro každou značku a typ těchto atributů.
    d. Připojení ke XML: `<!DOCTYPE kořen SYSTEM "soubor.dtd">`
    e. DTD je poměrně starý a málo expresivní jazyk. Jeho další nevýhoda je, že DTD samotný není XML soubor.
61. **Jaké vlastnosti definuje XSD** (XML Schema Definition) – popisuje strukturu XML dokumentu
    Definuje:
    a. místa v dokumentu, na kterých se mohou vyskytovat různé elementy
    b. atributy
    c. které elementy jsou potomky jiných elementů
    d. pořadí a počty elementů
    e. zda element může být prázdný, nebo zda musí obsahovat text
    f. datové typy elementů a jejich atributů

62. **Zdůvodněte potřebu propojování programovacích jazyků s databázemi.**
    Data potřebná k běhu složitějších programů mohou být externě uložena například v databázi. Databáze má výhodu v tom, že dokáže uchovat velké množství dat, které je nějakým způsobem roztříděno do tabulek. Jednotlivé tabulky jsou k sobě připojeny relacemi. Data nemusejí být po začátku běhu programu načítaná celá. Přistupujeme pouze k těm, které potřebujeme.
63. **Charakterizujte relační databázi**
    Relační databáze je založena na tabulkách, jejichž řádky obvykle chápeme jako záznamy a eventuelně některé sloupce v nich (tzv. cizí klíče) chápeme tak, že uchovávají informace o relacích mezi jednotlivými záznamy v matematickém slova smyslu.
64. **Co to je a k čemu slouží primární klíč**
    Primární klíč je jednoznačný identifikátor záznamu, řádku tabulky. Primárním klíčem může být jediný sloupec či kombinace více sloupců tak, aby byla zaručena jeho jednoznačnost. Má dvě základní vlastnosti: jedinečnost v rámci tabulky a ne-NULL-ovou hodnotu
65. **Jaké jsou přednosti a nedostatky jazyka SQL?**
    SQL jazyk – dotazovací jazyk používaný pro práci s daty v relačních databázích, umožňuje úplnou kontrolu nad systémem řízení báze dat
    Práce s daty je sofistikovanější. Dotazy mají strukturu jako věty. Je logický a jednoduchý na naučení. Nezávislost na databázi.
    Nízká bezpečnost. Malé možnosti pro ladění a refaktoring.
66. **Uveďte tvar příkazu SELECT** – výběr hodnot specifikovaných atributů
    tvar: SELECT atributy FROM tabulka WHERE podmínka

```
SELECT Jmeno, Prijmeni FROM Studenti WHERE Vek > 23
```

67. **Uveďte tvar příkazu INSERT** – vkládá hodnoty do tabulky
    Tvar: INSERT INTO tabulka (atributy) VALUES (hodnoty)

```
INSERT INTO Studenti (Jmeno, Prijmeni, Vek, Skola, Prumer) VALUES (‘Jan’, ‘Novy’, 25, ‘FAV ZCU’, 3,1)
```

68. **Uveďte tvar příkazu DELETE** – vymaže data z tabulky
    Tvar: DELETE FROM tabulka WHERE podminka

```
DELETE FROM Studenti WHERE Vek>39 and Prumer>3
```

69. Uveďte tvar příkazu UPDATE – aktualizuje data
Tvar: UPDATE tabulka SET atribut = hodnota, atd WHERE podminka

```
UPDATE Studenti SET Skola='VUT', Vek= 12 WHERE Jmeno='Gustav' and Prijmeni='Klaus'
```

70. **Jakou konstrukcí umožní modul pyodbc provádět SQL příkazy**

```
import pyodbc
c = pyodbc.connect("DSN=Lide") #vytvori spojeni c se zdrojem dat (byl pojmenovan Lide)
cursor = c.cursor() #vytvori kurzor cursor
```

71. **Co je to databázový kurzor, k čemu slouží?**
Objekt kurzor dovoluje provádět operace na databázi (selekty, inserty, updaty, delety). V objektu kurzor jsou interně uloženy výsledky dotazu.
72. **Popište sémantiku metod fetchone, fetchmany, fetchall** – používají se k výběru řádků výsledku dotazu v podobě objektu
    a. **fetchone()** - vrací n-tici = další řádek výsledku uloženého v kurzoru
    b. **fetchmany(n)** - vrací n řádků, které jsou na řadě ve výsledku uloženém v kurzoru
    c. **fetchall()** - vrací všechny řádky výsledku
73. **Popište sémantiku metody commit**
Příkaz k zakončení transakce (zapsání změn do databáze)

```
c = pyodbc.connect("databaze")
c.commit() # musí se udelat, není zde autocommit
c.close() # uzavření databáze
```

74. **Definujte využití INNER JOIN fráze v příkazu SELECT**
Spojení záznamů z více tabulek k získání jediné výsledné relace.

```
"select Prijmeni, Vek from Studenti inner join Skoly on Skoly.Skola=Studenti.Skola where
Skoly.Mesto='Plzeň'"
```

75. **Definujte slovně termy jazyka Prolog**
Term – jednotná datová struktura, se kterou pracuje prog. jazyk Prolog.

```
•struktura
   •jednoduchý term
     -proměnná
     -konstanta (číslo, atom)
```

76. **Co jsou atomy Prologu?**
    a. řetězce znaků začínající malým písmenem obsahující pouze písmena, číslice a podtržítko
    b. znaků uzavřená v apostrofech (některé implementace používají uvozovky)
    c. atomy skládající se pouze ze speciálních znaků
77. **Objasněte pojem anonymní proměnná Prologu a její vlastnosti**
Speciální typ proměnné. Značí se jako podtržítko a používá se v pravidlech. Její hodnota není podstatná a Prolog ji ve výsledcích nezobrazuje.

Příklad: predikát, zda X je dítě

```
je_dite(X) :- dite(X,_).
```

78. **Popište princip rezoluce Prologu** – jak Prolog hledá řešení
    a. Předpokládáme pravidla a, b tvaru:
    $a :- a_1, a_2, \ldots a_n.$
    $b :- b_1, b_2, \ldots b_n.$
    b. Nechť b_i je a.
    c. Pak rezolucí je:
    $b :- b_1, b_2, \ldots b_{i-1}, a_1, a_2, \ldots a_n, b_{i+1}, .. b_m.$

Když se při plnění cílů z těla pravidla b narazí na zpracování cíle b_i alias a, začnou se zpracovávat cíle těla a.

79. **Popište princip unifikace v Prologu**
    a. porovná-li se volná proměnná s konstantou, naváže se na tuto konstantu
    b. porovnají-li se dvě volné (neinstalované) proměnné, stanou se synonymy
    c. porovná-li se volná proměnná s termem, naváže se na tento term
    d. porovnají-li se termy, které nejsou volnými proměnnými, musí být pro úspěšné porovnání stejné
80. **Popište způsob plnění cílů v Prologu** - dotaz může být složen z několika cílů
    a. Při konjunkci cílů jsou cíle plněny postupně zleva
    b. Pro každý cíl je při jeho plnění prohledávána databáze od začátku

c. Při úspěšném porovnání klauzule s cílem je její místo v databázi označeno ukazatelem. Každý z cílů má vlastní ukazatel
d. Při úspěšném porovnání cíle s hlavou pravidla, pokračuje výpočet plněním cílů zadaných tělem pravidla
e. Cíl je splněn, je-li úspěšně porovnán s faktem databáze, nebo s hlavou pravidla databáze a jsou splněny podcíle těla pravidla
f. Není-li během exekuce některý cíl splněn ani po prohlédnutí celé databáze, je aktivován mechanismus návratu.
g. Splněním jednotlivých cílů dotazu je splněn globální cíl a systém vypíše hodnoty proměnných zadaných v dotazu.
h. Zjistí-li se při výpočtu, že globální cíl nelze splnit, je výsledkem no.

81. **Jak probíhá návrat při nesplnění cíle v Prologu**
    a. exekuce se vrací k předchozímu splněnému cíli, zruší se instalace proměnných a pokouší se opětovně splnit tento cíl prohledáváním databáze dále od ukazatele pro tento cíl
    b. splní-li se opětovně tento cíl, pokračuje se plněním dalšího, (předtím nesplněného) vpravo stojícího cíle
    c. nesplní-li se předchozí cíl, vrací se výpočet opětovně zpět
82. **Vysvětlete mechanismus působení predikátu řezu**
    a. Použijeme, když chceme zabránit hledání jiné alternativy
    b. Odřízne další provádění cílů z hlavy pravidla
    c. Je bezprostředně splnitelným cílem, který nelze opětovně při návratu splnit
    d. Projeví se pouze, když má přes něj dojít k návratu
    e. Změní mechanismus návratu tím, že znepřístupní ukazatele vlevo od něj ležících cílů (přesune je na konec DB)

==========

83. **Charakterizujte „čisté výrazy“** – nemění stavový prostor programu
    a. Hodnota výsledku nezávisí na pořadí vyhodnocování (tzv. Church-Roserova vlastnost)
    b. Výraz lze vyhodnocovat paralelně, např ve výrazu (x*2+3)/(fce(y)*x) lze pak současně vyhodnocovat dělence i dělitele. Pokud ale fce(y) bude mít vedlejší efekt a změní hodnotu x, nebude to čistý výraz a závorky paralelně vyhodnocovat nelze.

84. **Popište Church-Rosserovu vlastnost výrazů** (čistých výrazů)
    Hodnota výsledku nezávisí na pořadí vyhodnocování
85. **Definujte S-výrazy Lispu**
    Souhrnný název pro data v Lispu (čísla, znaky, řetězce, symboly, seznamy). Reprezentují data tak, aby byla člověku čitelná.
86. **Popište základní cyklus vyhodnocování Lispovského programu**
    1. Výpis promptu
    2. Uživatel zadá lispovský výraz (zápis funkce)
    3. Provede se vyhodnocení argumentů
    4. Aplikuje funkci na vyhodnocené argumenty
    5. Vypíše se výsledek (funkční hodnota)

87. **Jaké jsou elementární funkce Lispu a jejich sémantika**
    a. **CAR** alias **FIRST** selektor - vyběr prvého prvku
    b. **CDR** alias **REST** selektor - výběr zbytku seznamu (čti kúdr)
    c. **CONS** konstruktor-vytvoří dvojici z agumentů
    d. **ATOM** test zda argument je atomický
    e. **EQUAL** test rovnosti argumentů

88. **Popište sémantiku lispovské funkce COND**
    Větvení programu v LISPU
    například výpočet absolutní hodnoty:

```
(cond((> x 0) x)
      ((= x 0) 0)
      (t(-x)))
```

    pokud x > 0 je vráceno x
    pokud x == 0 je vrácena 0
    V ostatních případech je vráceno –x

89. **Popište tvar a účinek Lipovské funkce DEFUN** – pro definice funkcí
Zápis: (DEFUN jméno-fce (argumenty) tělo-fce)
Přiřadí jménu-fce lambda výraz definovaný tělem-fce. Vytvoří funkční vazbu symbolu jméno-fce.
Argumenty jsou ve funkci lokální. DEFUN nevyhodnocuje své argumenty. Hodnotou formy DEFUN je nevyhodnocené jméno-fce
```
(defun fce(x) (+xx)(*xx))
```
volání: `(fce 3)` vrátí 25

90. **Definujte tvar a využití lambda výrazů**
    a. popisují bezejmenné funkce
    b. jsou aplikovány na parametry

    př.
```
((lambda (x) ( * x x)) 5)
((lambda (y) ((lambda (x) (+ (* x x) y )) 2 )) 3)
```

91. **Co jsou to funkcionály, popište některý**
Funkce, jejichž argumentem je funkce nebo vrací funkci jako svoji hodnotu. Vytváří programová schémata použitelná pro různé aplikace (Highet order functions).
Př.: pro každý prvek s seznamu S proveď f(s)
```
(DEFUN zobrazeni (S)
      (COND  ((NULL S) NIL)
             (T (CONS     (transformuj(FIRST S))
                          (zobrazeni (REST S)) ))
))
```

92. **Objasněte rozdíl mezi „klíčovými slovy „ a „předdefinovanými slovy“**
    a. **Klíčová slova** - identifikátory, které tvoří slovní zásobu programovacího jazyka. Pro jiné účely, jako například název proměnné, je použít nelze.
    b. **Předdefinovaná slova** - identifikátory speciálního významu, které lze předdefinovat (např. vše z balíku java.lang–String, Object, System…)
93. **Objasněte rozdíl mezi dobou existence a rozsahem platnosti proměnné**
    a. **doba existence (lifetime)** - čas, po který je vázána na určité paměťové místo
    b. **rozsah platnosti (scope)** - interval instrukcí, po který je možné získat hodnotu proměnné z paměti (proměnna je viditelná)
94. **Uveďte příčiny vzniku synonym (alias) v programech**
Aliasy – dvě proměnné sdílejí ve stejné době stejné místo
    - pointery, referenční proměnné, parametry podprogramů, variantní záznamy (Pascal), uniony (C, C++)
95. **Popište princip statické a dynamické vazby jména proměnné s typem**
    a. **statická vazba** (jména s typem / s adresou)
        a. navázání se provede před dobou výpočtu a po celou exekuci se nemění
        b. Vazba s typem určena buď explicitní deklarací nebo implicitní deklarací
    b. **dynamická vazba**
        a. nastane během výpočtu nebo se může při exekuci měnit
        b. vazba s typem – specifikována přiřazením, výhoda – flexibilita, nevýhoda – vysoké náklady
        c. vazba s pamětí, doba existence proměnné
96. **Popište princip, výhody a nevýhody statické a dynamické vazby proměnné s adresou**
    a. **Dynamická** vazba proměnná s adresou - (nastane alokací z volné paměti, končí dealokací) doba existence proměnné (lifetime) je čas, po který je vázána na určité paměťové místo.
        a. v zásobníku = přidělení paměti při exekuci zpracování deklarací, výhody – rekurze
        b. explicitní na haldě = přidělení / uvolnění direktivou v programu během výpočtu.
        c. implicitní přidělování na haldě = alokace / dealokace způsobena přiřazením
    b. **Statická** - Navázání na paměť před exekucí a nemění se po celou dobu exekuce. Výhody – efektivní, přímé adresování, Nevýhody – bez rekurze
97. **Popište rozdíl mezi statickým a dynamickým rozsahem platnosti proměnné**
    a. **statický** (lexikální) rozsah platnosti:
        a. určen programovým textem
        b. k určení asociace jméno – proměnná je třeba nalézt deklaraci
        c. vyhledávání: nejprve lokální deklarace, pak globálnější rozsahování jednotka, pak ještě globálnější
        d. proměnné mohou být zakryty
        e. prostředkem k vytváření rozsahových jednotek jsou bloky

b. **dynamický** rozsah platnosti
    a. založen na posloupnosti volání programových jednotek (namísto hlediska statického tvaru programového textu, řídí se průchodem výpočtu programu)
    b. proměnné jsou propojeny s deklaracemi řetězcem vyvolaných podprogramů

98. **Definujte pojem silný typový systém programovacího jazyka**
Programovací jazyk má silný typový systém, pokud typová kontrola odhalí veškeré typové chyby

99. **Definujte pojmy strukturální a jmenná kompatibilita typů**
    a. **Jmenné kompatibility** – dvě proměnné jsou kompatibilních typů, pokud jsou uvedeny v téže deklaraci, nebo v deklaracích používajících stejného jména typu (ADA, Java)
    b. **Strukturální kompatibility** – dvě proměnné jsou kompatibilní, mají-li jejich typy identickou strukturu (Pascal a C)

100. **Definujte pojmy „literál" a „manifestová konstanta"**
    a. **Literál** - konstanta, která nemá jméno
```
String s="cat"; // cat je literal
```
    b. **Manifestová konstanta** - jméno pro literál

101. **Jaké rozlišujeme druhy konstant podle doby jejich určení. Popište je**
Konstanty - mají fixní hodnotu po dobu trvání jejich existence v programu, nemají atribut adresa = na jejich umístění nelze v programu odkazovat
    a. statické
        a. určené v době překladu (př.: static final int ZERO = 0;)
        b. určené v době zavádění programu
    b. dynamické
        a. v C# konstanty definované readonly
        b. v Javě: každé non-static final přiřazení v konstruktoru
        c. v C : určena při překladu

102. **Jaké typy označujeme jako ordinální**
    a. primitivní mimo float
    b. definované uživatele (pro čitelnost programu)
        a. vyjmenované typy = uživatel vyjmenuje posloupnost hodnot typu, implementují se jako seznam
        b. typ interval = souvislá část ordinárního typu, implementují se jako typ jejich rodiče

103. **Popište, jaké typy označujeme jako uniony**
Uniony – typy, jejichž proměnné mohou obsahovat v různých okamžicích výpočtu hodnoty různých typů.

104. **Uveďte, jakým způsobem vzniká dangling pointer**
např. v jazyce C si vyžádáme paměť funkcí malloc. Zavoláme free(), ale pointer ukazuje na část paměti, která již byla uvolněna.
```
#include <stdlib.h>
{
  char *cp = malloc ( A_CONST );
  /* ... */
     free ( cp );      /* cp je nyní dangling pointer, ukazuje na část paměti,která již
     byla uvolněna */
 cp = NULL;        /* nyní je problém vyřešen */
  /* ... */
}
```

105. **Uveďte, jakým způsobem vzniká ztracená proměnná z haldy**
Vzniká, když je ukazateli přiřazena nová adresa dříve, než je uvolněna paměť, na kterou ukazuje.

106. **Popište pojmy precedence, asociativita a arita operátorů ve výrazech**
    a. **precedence – pořadí vyhodnocování**
        a. V aritmetice a algebře jsou používána různá pravidla, která určují pořadí, v jakém se vyhodnocují operace ve výrazu. Priorita, s jakou se vyhodnocuje daná operace, se nazývá precedence. Precedence každé operace je otázkou vzájemné domluvy - konvence. Nelze ji nijak matematicky odvodit.
    b. **asociativita – směr vyhodnocování**
        a. Asociativita je v matematice, zejména v algebře, vlastnost binární operace, říkající, že nezáleží na tom, v jakém pořadí operace provádíme, pokud se jich vedle sebe vyskytne více (například násobíme nebo sčítáme tři (čtyři …) čísla).
    c. **arita – počet operandů procující nad operátorem**
        a. Arita operace je rovna aritě kartézského součinu vstupu, tzn. obsahuje-li vstup n množin, pak říkáme, že operace je n-ární.
        b. př.: n=0 -> nulární, n=1 ->unární, n=2 -> binární

107. **Jaká jsou pozitiva a negativa příkazů skoku**
    a. nevýhody:
        a. znepřehledňuje program
        b. je nebezpečný
        c. znemožňuje formální verifikaci programu
    b. výhody
        a. snadno implementovatelný
        b. efektivně implementovatelný
108. **Které vlastnosti jsou důležité pro příkazy cyklů v programovacích jazycích**
    a. Jakého typu mohou být parametry a meze cyklu?
    b. Kolikrát se vyhodnocují meze a krok?
    c. Kdy je prováděna kontrola ukončení cyklu?
    d. Lze uvnitř cyklu přiřadit hodnotu parametru cyklu?
    e. Jaká je hodnota parametru po skončení cyklu?
    f. Je přípustné skočit do cyklu?
    g. Je přípustné vyskočit z cyklu?
109. **Jmenujte kriteria, dle kterých lze hodnotit vlastnosti podprogramů programovacích jazyků**
    a. Způsob předávání parametrů?
    b. Možnost typové kontroly parametrů?
    c. Jsou lokální proměnné umisťovány staticky nebo dynamicky?
    d. Jaké je platné prostředí pro předávané parametry, které jsou typu podprogram?
    e. Je povoleno vnořování podprogramů ?
    f. Mohou být podprogramy přetíženy (různé podprogramy mají stejné jméno)?
    g. Mohou být podprogramy generické ?
    h. Je dovolena separátní kompilace podprogramů ?
110. **K čemu slouží a co obsahují aktivační záznamy podprogramů a funkcí**

    Jsou v nich uloženy lokální data podprogramů.
    a. Místo pro lokální proměnné
    b. Místo pro předávané parametry
    c. Místo pro funkční hodnotu u funkcí
    d. Návratová adresa
    e. Informace o uspořádání aktivačních záznamů
    f. Místo pro dočasné proměnné při vyhodnocování výrazů

111. **Objasněte rozdíl statickou a dynamickou vazbou metod s místem jejich volání**
    a. brzká vazba – statická - Volání metod objektů je zařízeno pevnou adresou vzniklou při překladu.
    b. pozdní vazba – dynamická - V místě volání metody se nesmí použít pevná adresa objektu, musí tam být pouze symbolický odkaz. Ten se naplní v okamžiku přiřazení konkrétního objektu, tedy až v době běhu
112. **Uveďte jakou vazbu**
    a. statická - java, c++
    b. dynamická - python, obj. pascal