6. Rysy imperativně orientovaných jazyků, jazyků funkcionálního programování a logického programování.Rysy objektově orientovaných jazyků. Znalost na úrovni porozumění základním paradigmatům.

Paradigma= souhrn způsobů formulace problémů, metodologických prostředků řešení, metodik zpracování a pod.

Rysy imperativně orientovaných jazyků

V tomto případě je program tvořen posloupností příkazů (odtud imperativní).
Programátor tak musí analyzovat všechny situace, do kterých se může program dostat a podle toho tvořit program.
Imperativní programování popisuje jednotlivé úkony pomocí algoritmů. Zjednodušeně to lze popsat tak, že imperativní programy obsahují algoritmy, kterými se dosáhne chtěný cíl, zatímco deklarativní jazyky specifikují cíl a algoritmizace je ponechána programu (interpretu) daného jazyka.

Z toho vyplývá, že program napsaný v imperativním programovacím jazyce předepisuje **jak něco spočítat a jak dospět k výsledku**.

Imperativní jazyky mají svou podstatou blízko k samotným počítačům, které provádějí strojový kód, který je sekvence nějakých příkazů. Např. imperativní programovací jazyky běžně používají explicitní přiřazení do proměnné a její uložení na nějaké paměťové místo. Vyšší programovací jazyky pak mohou používat cykly, podmínky cyklů, funkce a procedury.

Algoritmus je subor pravidiel, ktory na zaklade zadanych vstupnych udajov dospeje po konecnom pocte krokov k pozadovanemu vysledku.

Algoritmizácia: Je proces vytvarania algoritmu.Pri tvorbe algoritmu pre vytváraný budúci softvérový systém vychádzame z analýzy formulovaného algoritmického problému. Postupne rozkladáme abstraktné príkazy, ktoré špecifikujú "ČO" sa má robiť a nahrádzame ich konkrétnymi príkazmi, ktoré určujú "AKO" sa to má robiť.

Vlastnosti algoritmu:

- elementárnosť- postup je zložený z činností, kt. sú pre realizátora elementárne a zrozumiteľné
- determinovanosť- pre každý krok algoritmu je jednoznačne určený nasledujúci krok
- hromadnosť- algor. musí vždy vychádzať z meniteľných vstupných údajov,t.j.mal by byť použiteľný pre riešenie každej úlohy daného typu
- rezultatívnosť- algor. vedie po konečnom počte krokov k výsledku
- konečnosť- postup skončí vždy v určitom čase a po vykonaní konečného počtu činností
- efektívnosť- postup sa uskutočňuje v čo najkratšom čase a s využitím čo najmenšieho počtu prostriedkov

imperativní programování se dělí na tři skupiny:

- Naivní paradigmabývá někdy chápáno jako samostatné paradigma a ještě častěji se mezi paradigmaty programování ani neuvádí. Typickým zástupcem je například jazyk BASIC
- Nestrukturované paradigma je velmi blízké assemblerům. Programy jsou lineárními sekvencemi příkazů a skoky jsou v nich realizovány příkazem typu „go to“ – tedy „jdi na (řádek)“. Typickými zástupci byly například rané verze jazyků FORTRAN a COBOL.
- Strukturované paradigma. Kvůli nepraktičnosti příkazu skoku „go to“ (ta vězí zejména v tom, že struktura programu nedává prakticky žádnou informaci o jeho vykonávání, což velmi komplikuje jeho ladění) vzniklo strukturované paradigma. Jeho hlavním přínosem je fakt, že nahrazuje příkazy skoku pomocí podmíněných cyklů („opakuj, dokud platí podmínka“) a jiných strukturovaných instrukcí, které se do sebe vnořují. Typickými zástupci jsou jazyky C, Pascal.

Strukturované programování

Označuje programovací techniku, kdy se implementovaný algoritmus rozděluje na dílčí úlohy, které se spojují v jeden celek. Každý celek se může skládat z menších bloků. Na nejnižší úrovni jsou bloky složeny z příkazů programovacího jazyka nebo volání funkcí. K implementaci v programu se používá řídících struktur.

Neodmyslitelnou součástí strukturovaných metodik je hierarchický přístup:

Přiřazení obecně provádí operaci s informací uloženou v paměti a ukládá výsledek do paměti pro pozdější použití. Vyšší programovací jazyky navíc dovolují provádění komplexnějších výrazů, jež mohou sestávat z kombinace aritmetických operací, programových operací, programových funkcí a přiřazování výsledných hodnot do paměti.

Řídící struktury

Všechny příkazy se provedou postupně

Větvení - Příkaz se provede v závislosti na splnění/nesplnění podmínky

Cyklus - V závislosti na splnění podmínky se část programu vykoná vícekrát

- nekonečný cyklus - za normálních okolností není vůbec ukončen „Elegantní“ způsob, jak naprogramovat nekonečný cyklus v jazyku C/C++/C# je použít for cyklus bez inicializace podmínky i inkrementu:
- while-do - cyklus s podmínkou na začátku, tělo cyklu se nemusí provést ani jednou:
- do-while(repeat-until) - cyklus s podmínkou na konci, tělo cyklu se provede aspoň jednou:
- for cyklus - cyklus s podmínkou na začátku, obvykle užívaný pro výčet prvků z množiny prvků

Datové struktury

Hlavním cílem je zjednodušit a zpřehlednit program, který provádí operace s datovým typem. Jsou-li všechny komponenty dané struktury téhož typu, označujeme strukturu homogenní, v opačném případě heterogenní.
Rozdělení:

- Statické – nemůžou v průběhu výpočtu měnit počet svých komponent ani způsob jejich uspořádání (pole, záznam)
- Dynamické – jejich rozsah se může v průběhu vykonávání programu měnit (ukazatel, zásobník, fronta, seznam, strom, atd.)

Datové typy

Datový typ definuje druh proměnných. Je určen oborem hodnot a zároveň typickými výpočetními operacemi, které lze s daty provádět.
Tři základní skupiny:

1. *Jednoduché*
    - Standardní – jsou definované jazykem
        - celočíselné (integer)
        - reálné (real)
        - znak (char)
        - logická hodnota (boolean)
    - Programátorem definované
        - dynamické – deklarace obsahují proměnné
        - statické – mohou obsahovat pouze konstanty
2. *Strukturované* – obsahují jeden nebo více datových prvků. Strukturované datové typy se skládají z jednoduchých typů. Dělí se na homogenní a heterogenní. Podle toho jestli mohou obsahovat více různých datových typů.
    - Homogenní
        - pole (array)
        - textový řetězec (string)
        - výčtový typ (enum)
    - Heterogenní – datový typ je složen z prvků různých datových typů (opak homogenního) a navenek se chová jako kompaktní celek.
        - Pascal typ record
        - seznam (list) – obdoba pole, ['a', 'b', 9, „cokoliv“]
        - C typ struct

```
typedef struct {
  int vek;
  char *jmeno;
  enum { muz, zena } pohlavi;
} Osoba;
```

3. *Zvláštní*
    - ukazatel (pointer) – ukazuje na místo v paměti
    - soubor (file) – reprezentuje ukazatel na soubor

Abstraktní datový typ

Abstraktní datový typ je implementačně závislá specifikace struktury dat s operacemi povolenými na této struktuře. Základní ADT jsou například:

- Zásobník
- Seznam
- Fronta
- Množina
- Textový řetězec

Zabudované ADT
Protože některé abstraktní datové typy jsou velmi užitečné a běžně používané, některé programovací jazyky používají tyto ADT jako primitivní datové typy, které jsou přidány do jejich knihoven. Například v Perlu je možné pole považovat za implementaci seznamu, standardní knihovny C++ a Javy zase nabízejí implementaci seznamu, zásobníku, fronty a řetězců

Procedury a funkce

Tvoří posloupnost instrukcí, které potřebujeme v programu provádět na různých souborech dat, nebo na různých místech v programu. Procedura nebo funkce může být po deklaraci použita kdekoliv v následujícím bloku programu.

- Funkce – vrací hodnotu (může být použita přímo ve výrazech) Funkce je operace, u které podle daného vstupu požadujeme určitý výstup. Vstupem rozumíme parametry a výstupem návratovou hodnotu.
- Procedura – nevrací hodnotu(vyvolá se příkazem volaní procedury ke splnění jedné nebo více operací) Procedura má stejnou konstrukci jako funkce, s tím rozdílem, že nevrací žádnou hodnotu (nemá return). Místo návratového typu má „void“.

Bloková struktura programu

Blok je řídící struktura, která kromě příkazů obsahuje též deklarace (ty platí pouze v daném bloku). Deklarace provedené uvnitř bloku ztrácí mimo blok platnost. Blok má 2 části: *deklarační a příkazovou*. Bloky mohou být do sebe vnořovány, přičemž věc deklarovaná v bloku je viditelná ve vnořeném bloku také.

Modulární struktura programu

Modul je programová jednotka, která na rozdíl od bloku umožňuje, aby v ní deklarované proměnné a metody byly použitelné i v jiných modulech.
Skládá se ze dvou částí:

- specifikační – zde jsou veřejné deklarace (obsahuje hlavičky procedur a funkcí)
- implementační – může obsahovat i neveřejné deklarace

Příkladem modulu může být například knihovna v C.

Rysy jazyků funkcionálního programování

Funkcionální programování je paradigma postavené na evaluaci funkcí. Nepoužívá stavy ani neprovádí přiřazení do proměnných, výpočet je jednostavový. Program je výraz a výpočet je jeho úprava.

Funkcionální programování patří mezi deklarativní programovací principy. Alonzo Church vytvořil formální výpočtový model nazvaný λ-kalkul. Tento model slouží jako základ pro funkcionální jazyky.

Funkcionální jazyky dělíme z hladiska používania lambda kalkulu na:

- typované – Haskell
- netypované – Lisp, Scheme

Výpočtem funkcionálního programu je posloupnost vzájemně ekvivalentních výrazů, které se postupně zjednodušují. Výsledkem výpočtu je výraz v *normální formě*, tedy dále nezjednodušitelný. Program je chápán jako jedna funkce obsahující vstupní parametry mající jediný výstup. Tato funkce pak může být dále rozložitelná na podfunkce.

Vlastnosti

Funkcionální programování je paradigmatické programování, které zachází s výpočtem jako s vyhodnocením matematických funkcí. Zaměřuje se na aplikaci složenou z funkcí, na rozdíl od imperativního programování, které se zaměřuje na změny stavu. Jinými slovy imperativní jazyky se orientují na popis toho, jak se má provádět výpočet, a jazyky deklarativní jsou naopak zaměřené na to, co se má vypočítat.

Referenční transparentnost (Čistě funkcionální programy)
znamená, že v daném kontextu představuje proměnná vždy jednu a tutéž hodnotu nebo také, že volání funkce na stejných parametrech vrátí vždy stejnou hodnotu (chybí totiž globální proměnné, vedlejší efekty a přiřazení do proměnné). Čistě funkcionální je například Haskell a Miranda
Příklad:
y = f(x) * f(x);
může kompilátor transformovat na

```
z = f(x);
y = z * z;
```

a ušetří jednu evaluaci funkce.
Nicméně ne všechny funkcionální jazyky jsou čisté. Jazyky z rodiny Lispu nejsou čisté, protože způsobují vedlejší efekty:

```
y  = random() * random();
```

Druhé volání random nemůže být eliminováno, protože návratová hodnota se může lišit od předchozího volání.

Striktní a líné vyhodnocování
funkcionální jazyky mohou býk kategorizovány podle toho, používají-li striktní nebo nestriktní vyhodnocení, což jsou koncepty, které říkají jak budou zpracovány argumenty funkce při vyhodnocování výrazu.. Striktní vyhodnocování vyžaduje předávání funkce hodnotou, kdežto líné vyhodnocování může za parametry funkce předávat i nevyhodnocený výraz (vyhodnocuje podle potřeby).

```
f:=x^2+x+1
g:=x+y

Následující výraz bude vyhodnocen jednou z těchto cest.
f(g(1, 4))

Výpočet vnitřnější funkce g jako první. Striktní výpočet -
argumenty funkce jsou vyhodnoceny před voláním funkce
f(g(1, 4))   →   f(1+4)   →   f(5)   →   5^2+5+1   →   31

Výpočet vnější funkce f jako první. Nestriktní výpočet -
argumenty jsou přenechány ve funkci nevyhodnocené a volání
funkce určuje kdy budou argumenty vyhodnoceny.
f(g(1, 4))   →   g(1,4)^2+g(1,4)+1   →   (1+4)^2+(1+4)+1   →   5^2+5+1   →
31
```

Striktní vyhodnocení je efektnější. Ve striktním výpočtu je argument počítám jednou, zatímco v nestriktním může být počítán vícekrát, jak můžete vidět v příkladu nahoře, kde je funkce g vypočítána vícekrát. Striktní výpočet je také jednodušší implementovat, pokud argumenty předané datové funkci jsou datové hodnoty, v nestriktním výpočtu mohou být argumenty výrazy. Jazyky jako LISP, ISWIM a ML spolu s hodně novými funkcionálními jazyky používají striktní výpočet.
Nicméně jsou tu důvody preferovat nestriktní výpočet. Lambda kalkul poskytuje silnější teoretické základy pro jazyky, které používají nestriktní výpočet. Nestriktní výpočet používají nejvíce definiční jazyky. Například podporuje nekonečné datové struktury jako seznam všech kladných proměnných typu integer nebo všech prvočísel. S nestriktním výpočtem jsou tyto struktury vypočítány pouze v kontextu, kde je vyžadována konečná délka. To vedlo k vývoji líného výpočtu, což je typ nestriktního výpočtu, kde výsledek počátečního výpočtu kteréhokoliv argumentu může být sdílen přes výpočtovou sekvenci. Ve výsledku není argument spočítán nikdy více než jednou. Líný výpočet je používán hlavně línými moderními čistě funkcionálními jazyky jako je Miranda, Clean a Haskell.

Funkce vracející funkce (Higher-order funkce)
Funkce jsou higher-order ve chvíli, kdy můžou převzít druhou funkci jako argument a navrátit ji jako výsledek. (Derivace a antiderivace jsou toho příkladem). Higher-order funkce jsou blízce příbuzné s funkcemi první třídy v tom, že obě dovolují funkci jako argument a výsledek jiné funkce. Rozdíl mezi těmito funkcemi je tento: higher-order popisuje matematický koncept funkce aplikovanou na jinou funkci, přičemž First class je počítačově-vědní termín, který popisuje programové jazykové entity, které nemají žádné omezení v jejich použití (tudíž first class funkce se může objevit kdekoliv v programu, stejně jako jiné first class entity, jako například čísla, včetně použití jako argument funkce nebo návratová hodnota funkce). Higher-order funkce aktivují currying, což je technika, v které je funkce používána na vlastní argument najednou, přičemž při každém použití navrátí další higher-order funkci, která přijme další argument.

Rekurze
Opakování je ve funkcionálních jazycích obvykle provedeno pomocí rekurze. Rekurzivní funkce vyvolávají samy sebe, čímž dovolují opakování programu. Konec rekurze může být rozpoznán a optimalizován kompilerem do stejného kódu, který se používá na implementaci

opakování u imperativních jazyků. Programovací jazyk Scheme standardně vyžaduje další implementaci k rozpoznávání těchto funkcí.
Obecné vzory rekurzí mohou být změněny za použití higher order funkcí, catamorphismus a anamorphismus jsou toho nejzřejmější příklady. Takové higher order funkce hrají obvykle roli podobnou vestavěným kontrolním strukturám, jako jsou smyčky v imperativních programovacích jazycích.

Použití

Funkcionální programovací jazyky, hlavně čistě funkcionální, se používají spíše v akademických kruzích, než v komerčním softwarovém vývoji. Nicméně funkční programovací jazyky používané v průmyslu a komerčních aplikacích zahrnují Erlang, R (statistický), Matematika (symbolická matematika), Haskell, ML, J a K (finanční analýza) a domain-specific programovací jazyky jako XSLT. Dále jsou funkcionální programovací jazyky důležité pro některá odvětví informatiky, například zabývající se umělou inteligencí, formální specifikací, modelováním nebo rychlým prototypováním.

Rysy jazyků logického programování

Logické programování vychází z použití matematické logiky v programování. Patří mezi deklarativní paradigma – obdobně jako funkcionální programování je jeho úkolem zadat co je třeba spočítat nikoliv jak to spočítat. Program je teorie a výpočet je odvození z ní. Logické programy popisují vztahy mezi hodnotami pomocí tzv. Hornových klauzulí. Program se skládá ze seznamu klausulí (teorie) a z formule (cílu, dotazu). Výpočet je hledání splňujících substitucí, tj. takových ohodnocení proměnných z cíle, při nichž cíl vyplývá (lze odvodit) z teorie. Ve většině logických jazyků záleží na pořadí klauzulí a podcílů v nich. Když tedy řešení existuje, nemusí se při nevhodném uspořádání najít; když neexistuje, nemusí se to v konečném čase zjistit. Operátor ! (řez) zabraňuje opakovanému vyhodnocování podcíle, pokud tento cíl již jednou uspěl.

Logické programování je založeno na ohraničené části logiky prvního řádu. Pro logiku prvního řádu platí, že dovoluje užívání predikátů "Pro každé něco" nebo "Existuje něco", ale něco musí být individuum, ne predikát. Dále je založeno na logice Hornových klauzulí, což jsou konečné množiny literálů spojených disjunkcí (nejvýše jeden literál je pozitivní), a které se pro potřeby logického programování přepisují do tvaru:

$(P1 \,\&\, P2 \,\&\, P3 \ldots) \rightarrow Q$

Literály jsou výrokové proměnné nebo jejich negace.

Programovanie v Prologu

Základními využívanými přístupy v LP (Prolog) jsou unifikace, rekurze a backtracking.

Programovanie v Prologu pozostáva z 2 krokov:

1. Vytváranie databázy obsahujúcej informácie o objektoch sveta a vzťahoch medzi nimi. Databáza je vlastne prologovský „program“.

2. Formulácia otázok o objektoch a vzťahoch medzi nimi. Prologovský výpočet je vlastne dialóg, v ktorom používateľ kladie systému otázky a systém na ne odpovedá na základe informácií obsiahnutých v databáze.

Jednotlivé informácie sú v prologovskej databáze vyjadrené tzv. klauzulami.

Klauzuly sú trojakého typu:

- Fakty – vyjadrujú základné informácie o vlastnostiach objektov a vzťahoch medzi nimi - napr. muz(jozef). otec(jozef,marek).
- Pravidlá – všeobecné konštatovania o objektoch a o vzťahoch medzi nimi – napr. surodenci(X,Y) :- rodic(X,R), rodic(Y,R), X \= Y.
- Príkazy – sa vykonajú hneď pri načítavaní(nakonzultovaní) databázy – napr. :-consult(prve).

Fakt je tvorený štruktúrou v tvare: funktor(arg1, arg2, ..., argN)

- Funktor reprezentuje vlastnosť (napr. muz), resp. vzťah (napr. otec)
- argumenty sú jednotlivé objekty, ktorých sa daná vlastnosť, resp. vzťah týka – napr. jozef a marek vo fakte otec(jozef,marek)
- argumentami štruktúry môžu byť ľubovoľné objekty jazyka, teda aj štruktúry

Atómy považujeme za štruktúry s nulovým počtom argumentov. Začínajú sa vždy malým písmenom – napr. otec alebo marek

Činnosť systému Prolog

- Cieľom tejto časti je dôkladnejšie pochopiť jednotlivé kroky, ktoré Prolog pri plnení cieľov vykonáva
- Výber tých informácií z databázy, ktoré Prolog potrebuje pri plnení zadaného cieľa, sa deje vyhľadaním prvej klauzuly, ktorá je s týmto cieľom unifikovateľná
  - Keď je touto klauzulou fakt, cieľ je okamžite splnený
  - Keď je ňou pravidlo, potom podmienkou splnenia cieľa je splnenie podcieľov v tele tohoto pravidla.
- Unifikácia predstavuje najvšeobecnejšiu substitúciu, ztotožňujúcu dva objekty jazyka.

Unifikácia

- V konvenčných programovacích jazykoch je základným výpočtovým krokom priradenie hodnôt premenným. Prolog využíva oveľa všeobecnejší typ priradenia, ktorý nazývame unifikácia.
- Na začiatku sú všetky premenné voľné (resp. neviazané), t.j. nezastupujú žiadny konkrétny objekt
- K viazaniu premenných môže dôjsť práve v procese unifikácie
- Prolog má zabudovaný predikát (infixný operátor) = , ktorý vyvolá pokus o unifikáciu jeho dvoch argumentov

Pravidlá unifikácie

1. Dve konštanty sú unifikovateľné iba keď sú zhodné
2. Voľná premenná je unifikovateľná s každým objektom, pričom dochádza k viazaniu premennej na tento objekt
3. Dve voľné premenné sa po unifikácii stávajú združenými, čo znamená, že keď sa niektorá z nich v ďalšom naviaže na nejaký konkrétny objekt, aj druhá bude súčasne viazaná na ten istý objekt
4. Dve štruktúry sú unifikovateľné, keď majú zhodný funktor a rovnaký počet argumentov, pričom všetky odpovedajúce si dvojice argumentov musia byť konzistentne unifikovateľné
5. Cieľ (resp. podcieľ) je unifikovateľný s faktom v databáze, keď sú reprezentované unifikovateľnými štruktúrami (atóm sa chápe jako štruktúra bez argumentov)
6. Cieľ (resp. podcieľ) je unifikovateľný s pravidlom v databáze, keď cieľ a hlava pravidla sú tvorené unifikovateľnými štruktúrami

Príklady unifikácie

```
1 ?- alfa = alfa.
Yes

2 ?- 25 = 37.
No

3 ?- muz(X) = muz(karol).
X = karol
Yes

4 ?- lubi(peter,Y) = lubi(peter,pivo(gemer,12)).
Y = pivo(gemer, 12)
Yes

5 ?- lubi(X,pivo(Y,12)) = lubi(peter,pivo(plzenske,12)).
X = peter
Y = plzenske
Yes

6 ?- lubi(X,pivo(plzenske,X)) = lubi(peter,pivo(plzenske,12)).
No

7 ?- lubi(peter,X) = lubi(Y,Z).
X = _G316
Y = peter
Z = _G316
Yes
```

Negace jako selhání

Micro-planner obsahoval konstrukci nazvanou "thot", která po aplikaci na výraz vracela "pravda", pouze když vyhodnocování výrazu selhalo. Ekvivalentní operátor je součástí všech moderních implementací prologu a byl nazýván **negace jako selhání**. Běžně se značí *not(p)*, kde p je atom, jehož proměnně jsou vytvořeny v době, kdy je not(p) zavolán. Literály negace jako selhání se můžou vyskytnout jako podmínka not(B_i) v těle programu klauzulí.

Backtracking

(česky zpětné vyhledávání, metoda pokusů a oprav, metoda zpětného sledování, metoda prohledávání do hloubky) je způsob řešení algoritmických problémů založený na prohledávání stavového stromu problému.[1] Jedná se o vylepšení hledání řešení hrubou silou v tom, že velké množství potenciálních řešení může být vyloučeno bez přímého vyzkoušení. Algoritmus je založen na prohledávání do hloubky možných řešení.

Aplikace LP

- zpracování přirozeného jazyka
- expertní systémy
- sybolická manipulace s výrazy, řešení rovnic
- simulace

Výhody

- vyšší úroveň, tj. program je bližší popisu problému než implementaci
- logika programu oddělena od řízení výpočtu
- snazší důkazy korektnosti

Nevýhody

- efektivita
- nevýhodné pro aplikace s intenzivním I/O

Zástupcem logického programování je programovací jazyk Prolog (další Gödel, Mercury), jehož vznik se datuje do roku 1972.

Rysy objektově orientovaných jazyků

Objektově orientované programování je metoda vytváření software. OOP vzniklo v 70. letech 20. století, oproti strukturovanému programování nedekomponuje problém shora dolů (rozdělení na menší celky, viz ap3), ale vytváří stavební objekty, pomocí nichž se program modeluje (styl zdola nahoru). Lépe odpovídá uspořádání skutečného světa. Mezi OO programovací jazyky patří Smalltalk, C++, Java, Python apod.

Objekty

Základní princip objektového programování je jednoduchý – vše jsou *objekty*. Objekty jsou nějaké entity, které mají nějaké vlastnosti – atributy a mají nějaké metody – procedury, tj. mohou něco provádět. Dále se pak uplatňují následující vlastnosti.
Když je objekt vytvářen, volá se také automaticky speciální funkce třídy nazývaná **konstruktor** (užívaná k inicializaci členů třídy). V některých jazycích (Java, platforma .NET) se automaticky stará o destrukci nepotřebných objektů **garbage collector**. Objekty jsou rušeny, jakmile na něj neexistuje žádný odkaz (např. při konci bloku, kde byl objekt

definován, nebo přiřazením hodnoty odpovídající *null*). Při zániku objektu je volána automaticky speciální funkce **destruktor**.

Třídy

Jsou vyjádřením abstrakce skupiny stejných objektů (stejné atributy a vlastnosti). Např. třída člověk. Každý objekt vytvořený ze třídy je pak *instance* třídy a dědí z ní všechny atributy a metody.
Speciálním případem třídy je **abstraktní třída**. Je to třída která obsahuje nejméně jednu *abstraktní metodu* a z této abstraktní třídy není dovoleno vytvářet instance. Teprve třída, která je jejím potomkem a implementuje všechny abstraktní metody, může vytvářet instance. Neabstraktní metody abstraktní třídy jsou společné všem potomkům.
Abstraktná metóda je metóda, ktorá je deklarovaná bez implementácie. V rodičovských triedach je nadeklarovaná len pre dedenie na podtriedy a musí byť implementovaná prv, než sa vytvorí objekt takejto triedy.
Interface (rozhraní) je množina hlaviček metod, které mohou být implementovány třídou. Pokud třída implementuje rozhraní, musí doimplementovat všechny jeho metody. Pokud tedy existuje rozhraní

Dědičnost

objekty jsou organizovány stromovým způsobem, kdy objekty nějakého druhu mohou dědit z jiného druhu objektů, čímž přebírají jejich schopnosti, ke kterým pouze přidávají svoje vlastní rozšíření. Tato myšlenka se obvykle implementuje pomocí rozdělení objektů do tříd, přičemž každý objekt je instancí nějaké třídy. Každá třída pak může dědit od jiné třídy (v některých programovacích jazycích i z několika jiných tříd).

Zapouzdření

Je stav, kdy objekt z nějaké třídy skrývá provádění některé své metody před okolím a navenek místo toho vystupuje jinou metodou např. pevně daným interface. Např. objekt třídy pes má metodu štěkat, ale uvnitř má další metody nadechnout a vydechnout.

Abstrakce

Je princip při modelování reálného světa, kdy si u reálných tříd objektů všímáme jen vlastností pro nás podstatných a ty pak modelujeme.

Polymorfismus

Je vlastnost metod, kdy metody mohou mít stejné jméno, ale jejich chování závisí na tom, ke které třídě patří.
Obecně rozlišujeme 3 druhy polymorfismu

1. objektový polymorfismus - v objektovém programovacím jazyku pod pojmem polymorfismus rozumíme to, že dva objekty rozšiřující třídu různým způsobem implementují stejnou abstraktní metodu **(1)**
2. překrytí(override) - objekt přetěžuje(znovu definuje) poděděnou metodu **(2)**
3. pretížení (overload) - stejně nazvaná metoda má stejnou obecnou funkcionalitu, ale může pracovat s různými parametry (např. zásobník.push(X) s parametry int X, bool X, string X bude fungovat vždy jako zásobník) **(3)**

Skládání

Objekt může obsahovat jiné objekty.

Delegování

Objekt může využívat služeb jiných objektů tak, že je požádá o provedení operace.

Předávání zpráv

Je komunikace mezi objekty jednoduše realizovaná tím, že objekt vyvolává metodu u jiného objektu.

Spolupráce objektů

Pokud objekty implementují rozhraní, tak z vnějšku s tímto objektem komunikují ostatní objekty pomocí rozhraní.

Spolupráce objektů je úzce svázaná s Objektově orientovaným návrhem. V návrhu se definují vazby mezi jednotlivými třídami, tj. volání metod mezi třídami, přistupování k atributům tříd atd. Pokud je dobře provedený návrh, tak je možné z tohoto návrhu rovnou vygenerovat balíčky, rozhraní a třídy včetně metod a atributů a programátor pak jen naimplementuje jednotlivé metody.

Funguje to podobně jako volání metod v rámci jedné třídy - vytvoříme si instanci třídy (nebo použijeme vhodnou existující), zavoláme metodu a dostaneme návratovou hodnotu(pokud je void, tak jen provede kód a nevrátí nic). Samozřejmě je třeba myslet i na výjimky.

Většina jazyků již má hotové nejrůznější balíčky objektů, které můžeme použít při běžné práci - například při komunikaci s databází, tvorbu GUI atd.

Událostmi řízené programování

Základní princip reakce objektového programu na vnější akci. Používá se např. u programů s GUI – běh programu je řízen událostmi.
Objekty GUI mají nadefinovány události (eventy; např. onClick, onMouseOut, onLoad, onClose…) a programátor k nim programuje metody – eventListenery – obsahující příkazy (např. volání privátních metod) které na událost reagují.
Události však nejsou pouze u objektů GUI, používají se u veškerých objektů, kde může vzniknout událost (object.onUnload, timer.onTimeout, http.onResponse, thread.onDispose…).
Událostmi řízené programování bývá vícevláknové, ale většinou spouštění vláken řídí běhové prostředí a my se o vlákna starat nemusíme. Existují ale případy, kdy je vhodné si separátní vlákna spustit samostatně (např. z důvodu využití výkonu vícejádrových procesorů).