Programování

Netriviální, kreativní proces, kdy nějakou myšlenku transformujeme na popis pro počítač. Tento proces je často zdlouhavý, náročný, vychází z nějaké reálné potřeby.

Spotřebovává **čas** a **zdroje**. V praxi jsou požadavky často vágní, nekompletní, během vývoje se mění. Typicky programátoři podceňují složitost zadání a přeceňují své schopnosti. Pravidlo 80:20 – po spotřebování 80 % zdrojů/času hotovo jen 20 % práce.

Programovací jazyk je „prostředník mezi přirozenou řečí a posloupností binárních číslic“. Poskytuje *jistou míru* abstrakce od toho hardwaru.

Programovací jazyk je konečná množina příkazů konkrétní syntaktické formy s přesně definovanou sémantikou.

Proč programovací jazyky? Přirozené jazyky nejsou vhodné – příliš složitá analýza založená na kontextu, nejednoznačnosti. Přesné popisy jsou příliš dlouhé, obsáhlé. Binární kód je velmi jednoduchý, ale extrémně obsáhlý, nedá se jednoduše zapamatovat, není vhodný pro rychlý vývoj.

Proč chceme pochopit programovací jazyk? Chceme lepší programy.

50. léta: symbolické názvy, subrutiny – jedna implementace, *n* volání – základní možnost opakování výpočtů.  
60. léta: zvyšování úrovně abstrakce  
70. léta: počítače se začínají dostávat mezi lidi – jazyky s vyšší abstrakcí, jednodušší konstrukce, které ale dovolí udělat prakticky všechno  
80. léta: počítače se dostávají na stoly – různé nové pohledy, modularita, objektová orientace, logické jazyky  
90. léta: Internet, propojování – přicházejí interpretované jazyky, jazyky s virtuálním strojem, funkcionální jazyky

Syntaxe: definuje programovou strukturu; říká, jak mohou za sebou následovat konstrukce programovacího jazyka. *Syntakticky správně zapsaný program nemusí dávat smysl, nemusí být správně sémanticky*.

Sémantika: popisuje a definuje význam jednotlivých konstrukcí jazyka, způsob jejich zpracování. Statická sémantika: popisuje vlastnosti, které můžeme studovat při překladu. Dynamická: popisuje vlastnosti, které mohou být detekovány a studovány až při chodu programu.

Formální modely sémantiky:

* Axiomatická sémantika: pro každý konstrukt definujeme množinu axiomů, typicky platí něco před, při a po provedení.
* Operační sémantika: vychází z modelu dnešních počítačů, definuje sekvenci přechodu stavů.
* Denotační sémantika: definuje program jako matematickou funkci, která mapuje vstupní hodnoty na výstupní.

Deklarace: vymezuje *atributy* dané entity. Proměnné – jméno, typ, kde se vezme; funkce – jméno, parametry.  
Definice: vymezuje způsob alokace (na zásobníků, v dynamické paměti…), u funkcí vymezuje tělo.

Vazby: vazby mezi tím, co já zapisuji, mezi jazykem a jeho prováděním. Spojuje entitu s jejími atributy, schopnostmi. Dějí se v různých časech: při návrhu jazyka, při implementaci programu, při překladu, při linkování, při načítání a spouštění, při běhu.

int count; ... ; count = count + 5;

* Množina typů, které mohu proměnné přiřadit (statická – vychází z programovacího jazyka; dynamická – mohu definovat vlastní typy)
* Typ konkrétně této proměnné
* Množina možných hodnot
* Konkrétní hodnota této proměnné
* Množina významů symbolu + (nejen sčítání, ale třeba i inkrementace nebo přetypování)
* Význam symbolu + na tomto místě

Vlastnosti proměnné:

* Jméno
  + Délka: maximální (kolik mi kompilátor dovolí), efektivní (kolik stačí k rozlišení dvou)
  + Znaková sada
  + Citlivost na velikost písmen
  + Klíčová slova, rezervovaná slova – nesmíme použít
* Adresa, umístění v paměti
  + Jméno může být spojeno s různými umístěními a adresami: lokální proměnné stejného jména v různých funkcích; lokální proměnné ve stejné funkci volané rekurzivně; globální proměnné; konstanty
  + Adresa může být přístupná pod různámi jmény
* Hodnota
* Typ
  + Vyjadřuje možné hodnoty a operace.
  + Statická vazba typu: typické u klasických jazyků, typ uveden v deklaraci.
  + Dynamická vazba typu: typická u skriptovacích, imperativních jazyků. Dynamická typová kontrola – časově náročná.
  + Jazyky bývají:
    - Beztypové (type-less): obvykle v nějakých formalismech, lambda kalkul… žádný typ tam prostě není
    - Netypované/dynamicky typované (non-typed): typ nepíšeme, odvozuje se, často se může měnit za běhu, ale je tam!
    - Typované/staticky typované: vazba mezi proměnnou a typem je statická. Mohou být explicitně typované nebo využívat odvozování – type inference.
* Doba života (*není to životnost!*)
  + Časový interval, ve kterém je pro danou proměnnou přidělena paměť
  + Statická vazba – statická alokace – před začátkem provádění programu
  + Dynamická vazba – dynamická alokace
    - Automatická
    - Ruční (příkazem alokovaná, dealokovaná)
* Rozsah platnosti (scope)
  + Část programu, ve které je možné s danou entitou pracovat.
  + Souvisí s viditelností proměnné a skrýváním proměnné (variable hiding).
  + Obvykle statická vazba – daná programovou strukturou.

L = R

* L-hodnota: výraz vyhodnocen na *umístění + adresa*
* R-hodnota: výraz vyhodnocen na hodnotu, kterou je možné někam umístit

Statická vazba umístění a adresy: statické proměnné. Statická vazba hodnoty: konstanty.  
Dynamická vazba: automatická (proměnná na stacku), manuální (alokace, dealokace); hovoříme o tom, kam to ukazuje.

# Principy OOP, OO jazyků

Modelování = tvorba abstrakce reálného systému

Abstrakce – můžeme si pod tím představit spoustu věcí: funkci, modul, třídu, objekt, celý systém…; do jisté míry se překrývá s pojmem *blackbox*.

Různé vlastnosti abstrakcí: míra (volba variantní a invariantní části problému), skrývání vnitřní struktury, kvalita (uživatelská přívětivost, přesnost chování, kvalita implementace – rychlost…).

Záleží, na co je abstrakce zaměřená: algoritmy (=> abstrakce zaměřená na algoritmy) -> ADT -> objekty (=> abstrakce zaměřené na data)

Vývoj: instrukce (ISU) -> *rutiny* (IZP) -> moduly (IAL) -> objekty (IPP, ICP).

U ADT byl problém v tom, že nebyly pevně spjaté *operace* a *data* – zvlášť rozhraní, k datům se dalo šahat i jinak než pomocí příslušných operací. U objektů jsou operace a data zapouzdřeny dohromady.

Data = atributy = datové položky  
Operace = metody = členské funkce  
Objektový systém = kolekce komunikujících objektů

Statické modely – překládané, soustředěné na zdrojový kód (typicky C++)  
Dynamické modely – větší expresivita, **reflektivita**, často dynamické typované, semitypované (s virtuálním strojem)  
ale často se **překrývají**

Pro minimální objektový model **ne**potřebujeme: třídy, dědičnost, atributy (ale je třeba možnost modifikovat kód metod „za běhu“), zapouzdření (přístupové modifikátory), parametry metod.

* Minimální model: *sigma kalkul* – proměnná (identifikuje objekt) + konstrukce objektu + zaslání zprávy – je výpočetně úplný
  + Dva typy zpráv: **invokuj metodu** nebo **modifikuj metodu** (což není typická/typicky využívaná vlastnost běžných jazyků)
  + + nějaká startovací zpráva

Další formalismy: UML, OCaml

*Syntaxe* se popisuje zase nějakými gramatikami + texty, příklad  
Popsat *sémantiku* není vůbec jednoduché: kombinace textu, příkladů, diagramů; víceúrovňové popisy.

Základní koncepty:

* Objekty (abstrakce)
* Zapouzdření – encapsulation
* Mnohotvárnost – polymorphism
* Dědičnost – inheritance

**Objekt:** autonomní („sám si rozhoduje“, jak na zprávy reagovat), výpočetně úplná entita („sám o sobě dokáže cokoliv“ => dekompozicí systému na objekty nesnižujeme jeho sílu vyjádřit nějaký program);  
*Identita* nezávislá na atributech (dva objekty se stejnými metodami a hodnotami atributů nejsou shodné);  
Jediný způsob, jak s ním komunikovat, je zprávou (ad zapouzdření)

**Zapouzdření:** nevidíme do objektu – vůči okolním objektům je objekt uzavřený (obvykle možné přesněji vyjádřit pomocí *modifikátorů viditelnosti*);  
Přístup pouze přes *veřejný* protokol (*message lookup*);  
Příjemce implicitně neví, odkud zpráva přišla;  
Zpráva má příjemce (objekt), selektor (název metody), argumenty;  
Protokol příjemce rozhodne reakci: invokace odpovídající metody a navrácení hodnoty odesílateli, nebo chyba.  
Objekt může poslat zprávu sám sobě, pak používá *interní* protokol – oproti veřejnému má navíc možnost přístupu k atributu (čtení nebo modifikace) – praxi obvykle syntakticky rozlišeno.

**Polymorfismus:** mnohotvárnost, stejnou zprávu lze zaslat různým objektům. Implementaci volané metody neznám – v různých objektech se může lišit, ale její signatura, způsob volání, může být stejná.  
Statické určení reakce = *brzká vazba* – v době překladu, rychlejší  
Dynamické určení reakce = *pozdní vazba* – až při běhu

**Dědičnost:** sdílení společných položek (metod, atributů, struktury) – cílem je znovupoužití metod a možnost specializace obecnějších objektů (přidávání nových položek, modifikace metod).

Způsoby tvorby nových objektů (in theory…):

* Vytvoření *prázdného objektu* a úprava jeho položek.
* Klonování existujícího objektu (*prototypu*) a úprava jeho položek.
* Vytvoření podle *šablony* a naplnění atributů – šablona = **třída**. Třída může být taky objektem „první kategorie“ (třeba Python, Smalltalk).

Třída: šablona pro vytváření nových objektů – instancí třídy. Je to jakási *entita* (může a nemusí být sama o sobě objektem) obsahující seznam atributů (+metadata – typ, výchozí hodnota) v instancích (které potom budou k dispozici v instancích) a implementace metod (sdílené instancemi).

Třídy se značně využívají v (praktické podobě třídní) dědičnosti – *relace dědičnosti je definovaná nad třídami, ne přímo nad objekty*.

**Instanciace:** proces vytvoření objektu (někde v paměti) + volání konstruktoru (speciální metoda zařizující naplnění atributů).  
**Instance** = identita (spíš tam nějak figuruje, je to speciální vlastnost, nemusí být nutně přímo obsažena v objektu) + reference na třídu (v podstatě metadata, může mít různé podoby) + data

Čistý třídní model OOJ Smalltalk: *všechno* je objekt, včetně tříd + každý objekt má svou třídu.

Dle přístupu k vytváření objektů: beztřídní (prototype-based, class-less) / třídní (class-based)  
Dle čistoty modelu: čisté (vše je objekt) / hybridní (míchání s jinými paradigmaty)  
Dle běhového prostředí: překládané / interpretované / částečně interpretované (mezikód, virtuální stroj)  
Dle uložení objektů: zdrojový kód + knihovny (při spuštění se znovu v paměti „vytvoří a pospojuje celý objektový model“) / obraz objektové paměti (objekty jsou uloženy v nějakém obrazu, i po vypnutí „žijí“ – po zapnutí jsou načteny ve stejné identitě)  
Dle způsobu iniciace výpočtu: výrazem / speciálním voláním  
Dle dědičnosti (počtu ***přímých*** předků): jednoduchá (C#) / vícenásobná (C++)  
Dle předmětu dědění: dědičnost implementace (třídní dědičnost / delegovaná dědičnost) / dědičnost rozhraní  
Dle způsobu určení typů: beztypové / netypované / typované  
Dle důslednosti kontroly typů: silně typované (type-safe) / slabě typované (implicitní konverze)  
Dle doby kontroly typů: staticky typované (při překladu) / dynamicky typované (za běhu)

# UML

Vizuální jazyk vhodný pro objektově orientované modelování. Společný jazyk pro zákazníka, návrháře a programátora.  
Umožňuje pohled z různých stran: strukturální (zkoumám statickou část, to, co pak bude „kód“), behaviorální (jak komunikují objekty, jak se mění jejich vnitřní stav…).  
Koncipován spíš jako prototypovací než jako úplný – spousta nepovinných prvků, modelujeme tím, čím se nám to zrovna hodí. Není ani výpočetně úplný – slouží pro komunikaci, ne „programování“ (ale existují nějaká rozšíření, která to umožňují).

### Diagram tříd a objektů

Viz prezentace.

Modifikátory přístupu: **-** (privátní), **#** (chráněný), **+** (veřejný).

Druhy vztahů: agregace a kompozice (*nějaké kolekce*), závislost, realizace, generalizace…

**Agregace**: instance jedné třídy může *obsahovat* několik různých instancí jiné třídy (resp. odkazy na ně). Nejsou na sobě „tvrdě závislé“: když vyšší objekt zemře, nemusí s sebou vzít ty odkazované. (SchoolClass – Pupil)

**Kompozice**: instance jedné třídy vzniká a zaniká spolu s instancí jiné třídy. (Person – SocialSecurityData).

# Výhody OOP

* „Jednodušší na pochopení“
* Praktické
* Znovupoužitelnost
* Zlepšení produktivity (ale čekalo se víc)
* Stabilita
* Delší křivka učení
* Režie objektů – bývá pomalejší

# Třídní OO jazyky

**Třída**:

* Seznam instančních atributů
* Implementace instančních metod
* U čistých třídních jazyků reference na její třídu (Smalltalk, Python)
* Třídní položky a data třídních atributů – statické členy (mezi *statickými* a *třídními* členy můžeme vnímat lehký rozdíl – statické členy jsou obvykle inicializovány při prvním načtení třídy do paměti, o třídních členech se můžeme bavit v těch velmi dynamických jazycích, kde mohou třídy vznikat za běhu).

**Extent** třídy A: kolekce všech *instancí* třídy A a jejích podtříd.

**Dědičnost**: umožňuje sdílení, znovupoužití položek; také umožňuje definovat typový systém.  
Superclass (předek, bázová třída) -> subclass (potomek).

Umožňuje redefinici zděděných metod – **overriding – překrývání** („redefinice“)!

Relace dědičnosti vytváří na množině tříd částečné uspořádání (reflexivní – leč obvykle v praxi nemůže dědit sama ze sebe; antisymetrické – nevznikne cyklus; transitivita).

## Typový systém

**Čistý**: vše je objekt.  
**Hybridní**: odvozený od strukturovaných jazyků – existují základní, *primitivní* datové typy. Mívají odlišnou sémantiku, někdy i syntaxi; někdy je třeba je zabalovat do objektů (*boxing*).

Hierarchie třídní dědičnosti: s kořenovou třídou / bez ní.

## Implementace třídního OOJ

Rozkol mezi požadavky a skutečností:

* Objektová paměť (asociativní, heterogenní) vs. počítačová paměť (lineární, homogenní).
* Model výpočtu: zasílání *zpráv,* invokace vs. skoky, zásobník volání

Jak uložit instance entity? Potřebujeme rozumně přistupovat ke zděděným položkám.  
Jak přistupovat ke členům (atributům, metodám)? Dynamic method binding – pozdní vazba, reflektování třídní hierarchie.  
Jak řešit vícenásobnou dědičnost?

Ve staticky typovaných OOJ je to složitější.

**Uložení instance v paměti**: viz slidy.

**Zahrnutí typu B do typu A** (***subsumption***): pokud je B podtřída A, lze instanci B využít kdekoliv tam, kde je očekávána instance A.

**Vytváření + rušení** instancí: viz slidy.

## Běhová prostředí

* Kompilované
* Částečně interpretované – s virtuálním strojem
  + Často se optimalizuje pomocí JIT
  + Lepší možnosti multiplatformnosti
* Interpretované
  + V dnešní době často přebírají principy částečně interpretovaných – proběhne nějaká celková analýza kódu („frontendu“), vznikne nějaká vnitřní reprezentace a ta se interpretuje
  + Typicky neobsahuje třeba JIT techniky
  + Může být flexibilnější

## Reflektivita

Způsob, který umožňuje programu zkoumat sama sebe – jaké datové struktury existují a podobně.  
Typicky dvě úrovně: **introspection** (zkoumání vnitřní reprezentace) a **intercession** (kauzální změna vnitřní reprezentace).

Má různé aspekty – **strukturální** reflektivita (pracuje se statickými strukturami), **behaviorální** (pracuje s prováděním programu).

Smalltalk – možnost za běhu měnit nadtřídy jiné třídy nebo měnit typ objektu beze změny identity.

# Typový systém

### Typ daný jménem

Typy proměnných jsou přesně dané jménem třídy (a relací dědičnosti – zahrnutí)  
… ?

### Skutečné podtypy

Typy se zjišťují pouze prozkoumáním položek instance. Je výrazně náročnější to implementovat (zejména ve staticky typovaných jazycích).

Testování implementace *potřebného podtypu* za běhu: **duck typing** (vypadá to jako kachna, kváká to jako kachna = je to kachna). Polymorfismus je nezávislý na třídní dědičnosti.

… ?

## Vícenásobná třídní dědičnost

Zděděná třída může mít více než jednu **přímou** nadtřídu (C++, Python).

### Problémy

Metody stejného jména a typu v definici nové podtřídy: zákaz / skrytí (nedostanu se k tomu, nebo se k tomu dostanu jen s plnou kvalifikací nadtypu, ze kterého to beru) / použití prvního nalezeného výskytu / povinná redefinice s plnou kvalifikací.

Metody stejného jména a různých typů v definici nové podtřídy: zákaz / implementace pomocí přetěžování.

Atributy stejného jména a typu: zákaz / skrytí / sloučení (jediný výskyt – Python) / replikace v paměti (problémové, komplikované)

### Inicializace

Může se provádět implicitně, nebo musí být definovaná explicitně. Různé možnosti – například DFS.

### Uložení instance v paměti

Viz slidy.

## Rozhraní

Cesta k částečnému překlenutí nevýhod jednoduché dědičnosti, aniž bychom zaváděli vícenásobnou dědičnost, ale zachovali nějakou *typovou vícejakost* instancí. Jednoduchá třídní dědičnost je využita pro dědění implementace, rozhraní se využívají pro dědění kontraktu.

Je to schéma deklarující sadu metod. Rozhraní mohou mezi sebou dědit i vícenásobně, ale v rozhraních jsou jen deklarace – nevznikají tak problémy vícenásobné dědičnosti.

Stále má některé problémy: vazby přes jména, výběr správné tabulky metod – chceme to dělat chytře a efektivně.

## Systémy s rolemi

Spíše filozofie návrhu, koncept. Objekt (vícetypový objekt, objekt s rolemi) má najednou více typů ve formě **rolí** a **dynamicky** je nabývá/pozbývá bez nutnosti změny své identity. Obvykle se nějak vyskytuje v interpretovaných jazycích s dlouhou dobou života objektů (Python, JS) nebo v databázových systémech.

Svou identitu spravuje objekt sám, ne podle nějaké třídy. Na objekt se dá nahlížet skrze jednu z rolí (typ).

**Perzistentní (instance) objekt(u)**: instance (některých) objektů přežívají dobu běhu aplikace, při dalším spuštění jsou k dispozici – se stejným stavem i **identitou**.

# Beztřídní OO jazyky

Vytváří objekty **klonováním prototypů**. Obvykle funguje zapouzdření objektu, obvykle to jsou dynamicky typované, interpretované jazyky.

Velmi dynamický pohled na dědičnost. Realizuje se jako **delegace** – zpráva přeposlaná rodiči nese i odkaz na původního příjemce.

Klasifikace a výběr jazyka

Podle míry abstrakce: strojové – vyšší – univerzální – strukturované – strukturované s ADT – objektově orientované

Strojově orientované jazyky: manipulují s bity, bajty. Vše si řídí programátor, nejsou k dispozici abstrakce.

Ranné jazyky vyšší úrovně: jednoduché datové typy, které abstrahují hardware, ale velmi target platform specific.

PL/I: *univerzální jazyk* – navržen tak, aby se dal použít pro všechno (minimálně v těch 80. letech). Měl předdefinováno obrovské množství různých abstrakcí, ale nedaly se definovat vlastní!

Strukturované jazyky: malý počet klíčových slov, které umožňují zadefinovat prakticky jakoukoliv abstrakci.

Jazyky s ADT: neobjektové, v podstatě strukturované jazyky s předdefinovanou sadou různých abstraktních datových typů.

OO jazyky: spojují data a operace.

### Jiná paradigmata

Logické jazyky

Funkcionální jazyk – blízké matematickým funkcím; nemají stavový model

Jazyky pro zpracování textu (typesetting)

DML, DDL – SQL

## Abstrakce řízení

**Imperativní jazyky** (procedurální)**:** programátor říká, **jaké operace** se mají provést a **v jakém pořadí**. Výpočet je stavový, probíhá po krocích, v podstatě dostává program z jednoho stavu do druhého. Do značné míry odpovídá té klasické von Neumannovské koncepci.

Viz slidy – abstrakce na různých úrovních.

**Deklarativní jazyky**: neobsahují stav! Řízení výpočtu sestává jen z informací o tom, **jaké operace** se mají provést. *Kompilátor* má jakousi strategii, jak podle toho sestavit program. Jsou tedy na výrazně vyšší úrovni abstrakce (od strojového kódu). Byly pokusy implementovat procesory, které by takovéto jazyky žraly přímo, ale nebyly moc úspěšné.

Najdeme v nich větvení, ale ne cyklení (nemáme posloupnost kroků) – místo něj silně využívá rekurzi. Různé strategie předávání toku řízení (call-by-need = lazy, call-by-name = non-strict, call-by-value = strict) – neplést volání a předávání parametrů (pass-by-…)!

## Formální báze

Formální prostředek (kalkul, algebra), kterým se dají popsat všechny konstrukty jazyka. Můžeme pomocí něj například provádět důkazy o vlastnostech jazyka.

SQL – formální bází je relační algebra – dotazy můžeme přeložit na jejich relační ekvivalent a ten pak snadněji (podle znalostí o komutativitě apod.) optimalizovat.

# Nestrukturované jazyky

**FORTRAN**: formula translation – vědecké a technické výpočty.

**BASIC**: oldschool.

Co nám z nich zbylo: nestrukturované řízení toku pomocí ***goto*** (cykly s if-goto…).

Formální báze pro ně typicky neexistují – jejich popisy jsou obvykle ve formě příručky listující možné příkazy a příklady jejich použití.

Jejich syntaxe bývá často jednoduchá, „řádkově orientovaná“ = jeden řádek, jeden příkaz, jedna sémantika.

Pojem *knihovní funkce* = předdefinovaná!

Jednoduché nebo žádné datové abstrakce – v basicu jen jeden číselný typ, změna mezi int/float se prováděla tak nějak sama.

Otevřený podprogram: část programu, kterou obcházíme, dokud ji explicitně nepotřebujeme – řízení jej obchází, dokud není explicitně vyžadováno její použití.

Dost těžko se daly aplikovat nějaké metodologie návrhu, tvorby kódu… náročná práce v týmu, vysoká chybovost.

Obvykle netypované/dynamicky typované. Někdy se typ vynucoval názvem proměnné.

# Strukturované jazyky

Zejména Pascal, Algol. Jejich vlastnosti se promítly ve většině pozdějších jazyků, naopak úplně čistě *strukturovaný jazyk* není ani to C.

Přinesly klíčový zlom v řízení toku i datových abstrakcích.

**Formální báze:** obvykle neexistuje, resp. neexistovala při vzniku jazyka, ale obvykle může být nebo byla později doplněna. Návrh jazyka prochází vývojem na základě zkušeností, potřeb (různé revize, standardizace).

**Syntaxe**: obvykle definovaná formálně (EBNF, syntaktické grafy, gramatiky…).

**Sémantika**: obvykle definovaná neformálně, ale *velice důkladně* – už ne na příkladech, ale obecně. Občas se vyskytne i nějaké semiformální/formální vysvětlení.

**Datové abstrakce**: obvykle nějaká celá čísla (odvozená přímo od HW + i jiná), desetinná čísla, textové řetězce a *nepředpřipravené* typy odvozené, které si mohu vytvářet **sám** kompozicí těch základních! Významný je obvykle taky typ *ukazatel*.

Vzniká rozdíl mezi deklaracemi a definicemi! Nově možnost vytvářet „rekurzivně“ datové struktury a vzájemně rekurzivní funkce. Otevírá dveře k modularitě. Bylo to tak, aby mohly být kompilátory jednoprůchodové.

## Návrh programu

Umožňuje částečné využití „dobrých programátorských technik“ (metodik SWE).

**Uzavřené podprogramy**! Jednoznačně dané rozhraní (jméno, typ výsledku, parametry). Možnost týmové spolupráce – i když ještě *nemáme moduly*, je možné nezávisle psát jednotlivé uzavřené podprogramy a pak to dát dohromady.

Přestávají mít vliv globální proměnné (v moderních jazycích už často vůbec nejsou).

## Výhody a nevýhody

Je možné dělat jakési „knihovny“ (nakopíruji si předpřipravený kód odněkud).  
Křivka učení je strmější (ale opravdu o tolik?)  
Mnohem efektivnější tvorba programu, možnost spolupráce.

## Typovost

Často jsou staticky a explicitně typované. K tomu ale často existují implicitní typové konverze, přetížené operátory.

Uživateli definované typy jsou náročné pro překladač.

## Ukládání v paměti

Obvykle se položky struktur ukládají v paměti jedna za druhou, za sebe. Liší se **zarovnáním v paměti** – někdy bez mezer, někdy s mezerami – liší se podle toho, co bereme za důležité (místo v paměti, efektivita vykonávání na procesoru).

???

# Modulární jazyky

*Modula-2* wtf, Ada, C, Pascal – a většina moderních jazyků.