7. Procedurální a objektově orientované programování (OOP)

**Imperativní programování** (též **procedurální programování**) je jedno z programovacích způsobů, jak jsou v programovacím jazyku formulována řešení problémů. Imperativní programování popisuje výpočet pomocí posloupnosti příkazů a určuje přesný postup ([algoritmus](https://cs.wikipedia.org/wiki/Algoritmus)), jak danou úlohu řešit. Program je sado [proměnných](https://cs.wikipedia.org/wiki/Prom%C4%9Bnn%C3%A1_(programov%C3%A1n%C3%AD)), jež v závislosti na vyhodnocení podmínek mění pomocí příkazů svůj stav. Základní metodou imperativního programování je procedurální programování, tyto termíny bývají proto často zaměňovány.

OOP na rozdíl od procedurálních jazyků, kde data a pokyny pro jejich zpracování existují odděleně, v objektově orientovaném programování se tato informace kombinuje do jediné entity.

**Objektově orientované programování**  je programovací způsob, ve kterém [Kód](https://cs.wikipedia.org/wiki/Zdrojov%C3%BD_k%C3%B3d) je v přidružen k [datům](https://cs.wikipedia.org/wiki/Data) ([metody](https://cs.wikipedia.org/wiki/Metoda_(objektov%C4%9B_orientovan%C3%A9_programov%C3%A1n%C3%AD)) jsou *zapouzdřeny* v [objektech](https://cs.wikipedia.org/wiki/Instance_t%C5%99%C3%ADdy)).

**Programovací jazyky** sobjektově orientovaným programováním:  
*-objektové jazyky*: výpočet probíhá výhradně interakcí objektů, a to vzájemným zasíláním zpráv, např. Ruby  
*-objektově orientované jazyky*: jsou založeny na imperativním programování a obvykle pouze částečně implementují vlastnosti objektového programování, např. C++ , Java  
*-objektová rozšíření původně neobjektových jazyků*, např. Object Pascal: vznikl rozšířením jazyka Pascal

**Základní koncept OOP** = snaha modelovat principy reálného světa, pokud možno jedna ku jedné.

**Objektově orientovaný program** je v nějakém programovacím jazyce zapsaný popis tříd, jejich instancí a zpráv, které si objekty mezi sebou posílají.

**Tři základní principy OOP:**

1. **Polymorfizmus.** Každý objekt má svou identitu. Důsledkem identity je polymorfismus, který říká, že **různé objekty mohou reagovat na zprávu různě.** Voláme pořád stejnou metodu, ale ta provádí pokaždé něco jiného

**Polymorfizmus neboli mnohotvarost = Jeden objekt může mít mnoho tvarů.** Například jedna instance třídy *grafický\_objekt*může obsahovat dle okolností bod, kružnici, ... Metoda nakresligrafických objektů: všechny grafické objekty chceme nakreslit, takže všechny ji budou mít. Zdědí ji od společného předka, třídy g*rafický\_objekt*, ale v každém z potomků bude dělat něco jiného - v jednom bude kreslit úsečku, v jiném kružnici, atd. To znamená, že tuto metodu musíme v každém z potomků **překrýt**(= **přebít** = naprogramovat, aby opravdu kreslila odpovídající objekt).

U **polymorfismu podmíněného dědičností** to znamená, že **na místo, kde je očekávána instance nějaké třídy, můžeme dosadit i instanci libovolné její podtřídy**. U **polymorfismu nepodmíněného dědičností** je dostačující, jestliže **se rozhraní (nebo jejich požadované části) u různých tříd shodují,** pak jsou vzájemně polymorfní.

**Polymorfismus**  umožňuje:

* jednomu objektu volat jednu [metodu](https://cs.wikipedia.org/wiki/Metoda_(objektov%C4%9B_orientovan%C3%A9_programov%C3%A1n%C3%AD)) s různými [parametry](https://cs.wikipedia.org/wiki/Parametr_funkce);
* objektům odvozeným z různých tříd volat tutéž metodu se stejným významem v kontextu jejich třídy, často pomocí rozhraní;
* [přetěžování](https://cs.wikipedia.org/wiki/P%C5%99et%C4%9B%C5%BEov%C3%A1n%C3%AD) operátorů = provedení rozdílné operace v závislosti na typu operandů;
* jedné funkci dovolit pracovat s argumenty různých typů (není ve všech programovacích jazycích).

**Základní trik polymorfizmu = potomek může nahradit předka.** To prakticky znamená, že do proměnné, původně deklarované s typem předka, můžeme přiřadit odkaz na instanci potomka. Pak lze přes proměnnou předka využívat i metody, které jsou v potomkovi.

**Abstraktní třída** je přímo předurčena k tomu, aby polymorfizmus využila. Jedná se o případ, kdy už při konstrukci hlavní třídy víme, jaké budeme potřebovat metody, ale jejich konkrétní implementace bude záviset na povaze zděděných tříd. Z tohoto důvodu není možné metody naprogramovat. Výhodným řešením je vytvořit abstraktní metodu s definovanými parametry a návratovým typem. Tím donutíme programátory zděděných tříd, aby tuto metodu překryli.

1. **Skládání=kompozice:** **objekt** může **obsahovat jiné objekty**.  
   Při definici nového nového typu nám nic nebrání použít jako datovou složku typ, který už existuje.

Příklad: Třída může mít členskou proměnnou objektového typu. Třída Zamestnanec využívá dvě proměnné na třídu Datum: class Zamestnanec { public String jmeno; public Datum narozeni, public Datum nastup; … }

**Dědičnost:** je **schopnost jedné třídy využívat metody druhé bez opakování jejich skutečné implementace.** Dědičnost představuje možnost přidat k základní třídě (tj. rodiči, předkovi, bázové třídě) další vlastnosti a vytvořit tak odvozenou třídu (tj. potomka, dceřinou třídu). Platí důležité pravidlo:**Potomek může vždy zastoupit předka.**To znamená, že na místě, kde v programu očekáváme instanci předka, můžeme vždy použít instanci potomka.

Vždy, když se rozhodujeme, zda použít dědění nebo kompozici, stačí se zeptat, zda třída eventuálně vkládaná kompozice do jiné třídy je i touto třídou. **Je** Datum nástupu Zamestnanacem? Není, proto použijeme kompozici. **Má** Zaměstnanec Datum nastupu? **Má, proto použijeme kompozici.**

Odvozená třída bude obsahovat všechny datové složky a všechny metody předka, k nim můžeme přidat nové datové složky a nové metody. Můžeme tedy **překrýt** (změnit, předefinovat) některou z metod předka. Při dědění nemůžeme žádnou datovou složku ani žádnou metodu odstranit.

Třída, která je sama potomkem, může sloužit jako předek jiné třídy. Tak mohou vzniknout dědické hierarchie, posloupnosti tříd, navzájem odvozené děděním. Pro vytvoření podtypu stačí v hlavičce třídy po jejím názvu uvést klíčové slovo *extends* a název rozšiřované třídy. Takto vytvořená třída zdědí všechny **nesoukromé** metody a třídní proměnné předka, které může znovu deklarovat a překrýt: **extens***jméno\_rodičovské\_třídy*

**Jestliže specifikaci předka vynecháme, doplní překladač jméno předka třídu *Object.***

**Java zná pouze jednoduchou dědičnost  = každá třída může mít pouze jednoho přímého předka.** Jako částečnou náhradu za ni má Java možnost používat **interface (= rozhraní).**

Při jednotlivých voláních metod podtypů často narazíme na to, že **nechceme celou metodu překrýt**, pouze k ní chceme přidat další funkcionalitu. V tento okamžik můžeme **zavolat *super.jmenoMetody()***, čímž zavoláme funkcionalitu předka. Analogicky k řetězení konstruktorů *this()* **můžeme volat konstruktor předka voláním *super()* – toto volání musí být v rámci konstruktoru potomka vždy na prvním místě**.

Jak jsme si již několikrát řekli, tak můžeme libovolnou metodu překrýt v potomkovi vlastní implementací. Pokud pak nad daným objektem tuto metodu zavoláme, dojde vždy k vykonání překrývajícího kódu. Toto ovšem platí pouze pro překrývání metod, nikoliv pro jejich **přetěžování**. Pokud budeme mít na daném objektu **dvě metody, které se budou lišit pouze parametrem** (jedna pro předka, druhá pro potomka), tak bude volána vždy ta metoda, která má v parametru stejný typ, jako je aktuální reference na objekt.

**Rozhraní objektů**

**definuje soubor metod, které ale v něm nejsou implemetovány**, tj. v deklaraci rozhraní je pouze hlavička metody, **stejně jako je to u abstraktní metody.** Třída, která rozhraní implementuje (tj. jakoby zdědí), musí překrýt (v tomto případě lépe řečeno implementovat) všechny její metody.

Rozhraní ale není totéž jako abstraktní třída, protože **nemůže deklarovat proměnné, jen konstanty**. **Třída** **může implementovat více než jedno rozhraní.** Naprosto **odlišné třídy z hlediska dědičnosti mohou implementovat stejné rozhraní.** Rozhraní je výhodné v případech, kdy chceme třídě vnutit zcela konkrétní metody nebo vidíme podobnosti v různých třídách, ale začlenit tyto podobnosti do vlastností tříd pomocí dědění by vyžadovalo zkonstruovat jejich předka.

**Konstrukce rozhraní příklad:** public **interface** **I**nfo {public void kdoJsem(); }

Specifikátor public není nutný, všechny metody rozhraní jsou implicitně public.

**Implementace rozhraní**: **implements***seznam\_rozhaní*

Rozhraní je syntaktická konstrukce, představující seznam hlaviček metod. **Třída, která rozhraní implementuje, se zavazuje implementovat všechny metody** z tohoto seznamu.

**Instance rozhraní může využívat jen metody rozhraní:** Proměnná typu rozhraní umožňuje přístup jen k  metodám deklarovaným v rozhraní. **Implementované rozhraní se dědí beze změny:** Zdědíme-li třídu, která implementovala rozhraní, bude metoda z rozhraní přístupná v obou třídách, ovšem bude se jednat o tutéž metodu. **Použití rozhraní:** V případě, že se společný předek těžko hledá, je možné použít rozhraní.

3) **Zapouzdření** v objektech znamená, že k obsahu objektu se nedostane nikdo jiný, než sám vlastník. Navenek se objekt projeví jen svým rozhraním (operacemi, metodami) a komunikačním protokolem.

**Zapouzdření** je spojení dat a metod třídy, tzn. data se vždy vyskytují společně s metodami, které s nimi manipulují.  Autorizovaný přístup k datům je důsledkem zapouzdření, kdy **zajistíme, aby s daty nebylo možno z vnějšku třídy manipulovat jinak, než pomocí metod této třídy**. Autorizovaný přístup v praxi znamená, že **proměnné deklarujeme s přístupovým právem private** místo public. V OOP bychom měli s datovými položkami manipulovat pomocí metod. Tím před zbytkem programu **skrýváme, jaká je skutečná datová reprezentace** tohoto typu = **ukrýváme jeho implementaci.**

8. OOP - třídy, instance tříd, dědičnost a polymorfismus

**TŘÍDA = základní stavební blok OOP (**=základní konstrukční prvek OOP **=základní stavební kámen OOP). Slouží jako předpis k definici** objektu jako **instance třídy. Třída definuje vlastnosti (atributy) a metody ( funkce = chování) objektů,** které při běhu programu vzniknou (též instancí). Hodnoty vlastností (atributů) se mohou u jednotlivých instancí odlišovat, na každém objektu (vzniklém z jedné třídy) můžeme zavolat shodnou sadu metod.

**Objektový typ = TŘÍDA**Klasické programování uvažuje o programování převážně z hlediska počítače, hardwaru. Objektové programování se na programování dívá z hlediska programátora. Jeho program musí vždy odpovídat zadané úloze, musí být jejím počítačovým modelem. Je vhodné, aby si programátor mohl definovat své vlastní **datové typy**. V OOP se jim říká objektové typy neboli TŘÍDY,neboť popisují třídy objektů. V objektovém programu představuje i třída samotná zvláštní objekt, kterému lze poslat zprávu - například žádost o vytvoření instance.

**Třída je forma, jež definuje** jasně ohraničený **soubor typů dat a operací nad nimi.** **Pomocí této formy pak vytváříme instance**, jednotlivé objekty, které obsahují samotná data (nad kterými jsou volány příslušné operace).

Pro deklaraci třídy používáme klíčové slovo***class***, před kterým je **specifikátor přístupu**, a za kterým následuje název třídy. Samotné tělo třídy je uzavřeno v bloku (složené závorky). **Pokud chceme třídu použít, tak musíme nejprve vytvořit její instanci**. Toto se dělá **pomocí** klíčového slova ***new***, **které vrací ukazatel** (referenci) **na paměťové místo, ve kterém se instance nachází. Dokud ukazatel nepřiřadíme instanci, tak ukazuje do prázdna (ukazatel *null*).**

**abstract**= abstraktní třída, tj. třída, ve které operace mají smysl jen pro odvozené třídy. Abstraktní třída shnuje společné vlastnosti svých potomků. **Java nedovoluje vytvářet instance abstraktních tříd.** **Třída, která obsahuje alespoň jednu abstraktní metodu, musí být definována jako*abstract.***

**final**= v jistém smyslu opak slova abstract, znamená *konečnou* třídu, tedy **třídu, od níž nelze odvozovat potomky**. Abstraktní třída nemůže být konečná.

**Přístupová práva složek třídy:** **public**= veřejné, tj. může je použít kdokoliv.

**protected**= chráněné. Můžeme je používat v metodách třídy, do které patří, v metodách jejích **potomků** **a** ve všech třídách téhož **balíku**.

**neovedeno**= složky smíme používat jen v témže **balíku** **private** = soukromé, smí se používat jen v dané třídě

**Dvě skupiny datových složek třídy = statické a nestatické:**

**Nestatické** datové složky: Každá **instance** obsahuje své vlastní nestatické složky.

**Deklarace** nestatické datové složky: *přístup*final datový\_*typ identifikátor inicializace*

**final** = nepovinný, představuje konstantu *inicializace* = předepisuje počáteční hodnotu dané složky

Jestliže v deklaraci uvedeme inicializaci, bude tato složka mít uvedenou hodnotu bezprostředně po vytvoření instance. Když ji neuvedeme, bude ihned po vytvoření instance obsahovat tzv. *implicitní hodnotu daného typu* (čísla budou obsahovat  *0*, znaky *´\u0000*´, logické složky *false*, pole a objekty *null*). Inicializace nestatických datových složek proběhne ještě před tělem konstruktoru. Konstantní datové složky musí být inicializovány v deklaraci nebo v konstruktoru.

**Statické datové složky:** hovoříme o nich jako o datových složkách třídy, protože jde o data společná všem instancím. Deklarace statické datové složky: *přístup***static** final *typ identifikátor inicializace;*

V metodách třídy, v níž je statická složka deklarována, ji používáme bez kvalifikace. Budeme-li ji chtít použít mimo třídu, musíme ji kvalifikovat jménem třídy. **+ viz otázka číslo 9 metody třídy**

**INSTANCE** (v některých [programovacích jazycích](https://cs.wikipedia.org/wiki/Programovac%C3%AD_jazyk) také objekt) **je konkrétní datový**[**objekt**](https://cs.wikipedia.org/wiki/Objekt_(informatika))**v paměti odvozený z nějakého vzoru (**[**třídy**](https://cs.wikipedia.org/wiki/T%C5%99%C3%ADda_(programov%C3%A1n%C3%AD))**)** používaný v [objektově orientovaných](https://cs.wikipedia.org/wiki/Objektov%C4%9B_orientovan%C3%A9_programov%C3%A1n%C3%AD) programovacích jazycích ([Java](https://cs.wikipedia.org/wiki/Java_(programovac%C3%AD_jazyk)), [C++](https://cs.wikipedia.org/wiki/C%2B%2B), ..). **Objekt představuje základní stavební prvek OOP.** Instance bývá obvykle vytvořena pomocí [konstruktoru](https://cs.wikipedia.org/wiki/Konstruktor) a klíčového slova *new*. Každý takový objekt má své vlastní [atributy](https://cs.wikipedia.org/wiki/Atribut) a [metody](https://cs.wikipedia.org/wiki/Metoda_(objektov%C4%9B_orientovan%C3%A9_programov%C3%A1n%C3%AD)) podle vzoru (třídy). Často se říká, že objekt je instance třídy, často se termíny objekt a instance vzájemně zaměňují. Podle jednoho vzoru třídy lze vytvořit libovolné množství instancí. V teorii OOP tedy znamená termín objekt totéž co instance.  V souvislosti s jazykem C++ a Java se termín objekt používá v širším smyslu - jako označení manipulovatelné oblasti paměti. **Vytvoření instance:** JmenoTridy jmenoNoveInstance = new JmenoTridy();

**Dědičnost** – viz otázka č. 7 **Polymorfizmus** – viz otázka č. 7

9. OOP - objekty, jejich tvorba, vlastnosti, metody

Všechny programy jsou simulací světa (skutečného nebo vymyšleného). Svět obsahuje objekty, které mají vlastnosti a komunikují spolu.

**OBJEKT** **= základní stavební prvek OOP**

OBJEKT = **vše, co můžeme nazvat podstatným jménem**: osoba, zvíře, věc, vlastnost (barva, vůně, ..), událost (připojení, přerušení, ...), stav (klid, pohyb, ...), abstraktní vlastnosti (krása, pohoda, ...), ...

Objekt nabízí abstraktní, zjednodušený popis skutečných věcí. Objekt můžeme používat, aniž bychom měli podrobnou představu o jeho vnitřním uspořádání.

**Objekt** je [entita](https://cs.wikipedia.org/wiki/Entita_(informatika)), která **v sobě slučuje**[**datovou část**](https://cs.wikipedia.org/wiki/Data_(informatika)) (uchovávající stav) **a**[**operace**](https://cs.wikipedia.org/wiki/Metoda_(objektov%C4%9B_orientovan%C3%A9_programov%C3%A1n%C3%AD)) ([procedury a funkce](https://cs.wikipedia.org/wiki/Podprogram), které jsou součástí objektu se zpravidla nazývají **metody**), které s daty objektu pracují. Datové složky objektu se obvykle nazývají **atributy**. Mnoho [objektově orientovaných jazyků](https://cs.wikipedia.org/wiki/Objektov%C4%9B_orientovan%C3%A9_programov%C3%A1n%C3%AD#Programovac%C3%AD_jazyky) je založeno na konceptu [třídy](https://cs.wikipedia.org/wiki/T%C5%99%C3%ADda_(programov%C3%A1n%C3%AD)), a objekt v nich je [instanci třídy](https://cs.wikipedia.org/wiki/Instance_t%C5%99%C3%ADdy), proto se pojem objektu často zužuje na toto pojetí, existují však objektově orientované jazyky, které koncept třídy nemají.

Důležitou vlastností objektů jsou jejich **vztahy s jinými objekty**. Tyto vztahy se dají obecně rozdělit do dvou skupin, na **vztahy typu je a má**. Příklad vztah mezi objektem učitel a člověk: Každý učitel je člověk. Třída učitelů dědí vlastnosti třídy lidé.

Příklad vztahu má:  Každé auto má motor. Tento vztah se označuje výrazem skládáni (=agregece). Při psaní třídy auto můžeme využít hotovou třídu motor.

**Tvorba objektů:** S objekty pracujeme pomocí odkazů. Je-li *A*třída, vytvoří deklarace

**A a;**

odkaz na instanci třídy *A*.

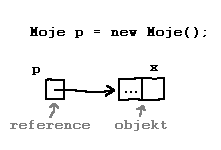
Instanci vytvoříme poté pomocí **konstruktoru** **a** operátoru ***new*:**

**a = new A();** // Konstruktor může mít i parametry

**Konstruktor**: Objekty odkazových typů v Javě vytváříme pomocí konstruktoru **new.** Při vytváření instancí tříd se vždy volá konstruktor. **Konstruktor je metoda**. Uplatní se v životě objektu **jen 1x** a to při vzniku objektu. Konstruktor se vždy **jmenuje stejně jako jeho třída**. Java dovoluje **přetěžování** metod, takže třída může mít několik konstruktorů, které se liší počtem nebo typem parametrů.

**Destruktor**: Narozdíl od C++ nebo Pascalu se o zrušení objektu v Javě destruktorem nemusíme starat, součástí JVM je automatická správa paměti = nástroj, který se automaticky postará o zrušení objektů, na které neexistuje žádný platný odkaz. V C++ se destruktor volá automaticky před smazáním instance.

Třídu deklarujeme pomocí klíčového slova class. Za ním následuje jméno třídy. Jméno třídy obvykle začíná velkým písmenem. Pokud se skládá z více slov, oddělujeme slova tak, že první písmeno každého slova je velké (např. BarevnyObrazek). Třída může obsahovat deklarace proměnných a metod. Proměnné i metody mohou být buď třídní (statické) nebo instanční. Instančním proměnným říkáme atributy. Třídní (statické) metody a proměnné se deklarují pomocí klíčového slova static. Instanční metody se deklarují podobně jako statické. Na rozdíl od statických metod však jejich hlavičky neobsahují klíčové slovo static. Instanční metody často pracují s instančními atributy.



Od dané třídy můžeme vytvářet instance (říkáme jim též objekty). Třída je šablona, která říká, jak budou objekty vypadat, tj. jaké budou mít atributy a metody. **Deklarací** proměnné typu Moje zavedeme proměnnou, do níž můžeme uložit referenci (odkaz) na instanci třídy Moje:

**Moje p;**

Protože proměnné typu třída odkazují na objekty, nazývají se referenční proměnné. Každá referenční proměnná zabírá v paměti stejný prostor: 32 bitů v 32-bitové JVM a obvykle 64 bitů v 64-bitové JVM. Velikost objektu je dána jeho atributy. Objekty vytváříme pomocí klíčového slova new:

p = **new Moje()**;

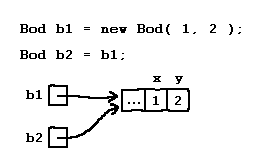
Po provedení tohoto příkazu bude proměnná p obsahovat odkaz na instanci třídy Moje. K atributům a metodám přistupujeme pomocí tečky:

p.x = 1; // atribut

p.vypisX(); // metoda

Konstruktor se volá při vytváření instance a obvykle provádí inicializaci objektu. Volání konstruktoru zapisujeme za klíčové slovo new. Konstruktor může mít parametry.

V jedné třídě je možné deklarovat více konstruktorů, pokud se liší seznamem parametrů. Má-li třída alespoň dva konstruktory, říkáme o konstruktoru, že je **přetížený**. Při provádění konstruktoru tedy atributy mají definovanou hodnotu. Buď mají nulové hodnoty nebo hodnoty, které jsme jim přiřadili pomocí inicializátorů.



**Vlastnosti** Proměnným v OOP říkáme vlastnosti.

viz otázka číslo 8 datové složky třídy (= vlastnosti)

**Metody** Operace v OOP se nazývají METODY.

Metody definují odezvu objektu nebo třídy na zprávy, které jiné objekty nebo třídy poslaly. Volání metody způsobí, že s instancí nebo s třídou proběhnou nějaké operace.

Metody v Javě lze **přetěžovat**. To znamená, že jedna třída může obsahovat několik metod se stejným identifikátorem, jestliže se liší počtem nebo typem parametrů.

Děděním lze nejen rozšiřovat repertoár metod o další, ale i „měnit“ dosavadní zděděné: metody můžeme **překrývat**.

**Deklarace***hlavička\_metody****:*** *modifikátory datový\_typ identifikátor (parametry)*

*modifikátory =*specifikace přístupových práv, význam je podobný jako u datových složek, určuje, kdo smí datovou metodu volat. Pak může následovat jeden z identifikátorů: **static, final, abstract.**

Deklarace metody:  *hlavička \_metody  { tělo metody }*

**Deklarace abstraktní metody** neobsahuje tělo a končí středníkem**:** *přístup****abstract****hlavička\_metody;*

**Parametry metod:**Formální parametry se objevují v deklaraci metody, skutečné parametry jsou výrazy, které zapíšeme na místě formálních parametrů při volání metody. Formální parametry se označují jako **parametry** a skutečné parametry jako **argumenty**.

Specifikace formálních parametrů se podobá deklaraci proměnných: *typ identifikátor*

*typ*= datový typ parametru, *identifikátor =* jméno parametru

**Nestatické metody**: Na volání nestatické metody se můžeme dívat jako na reakci instance na zaslanou zprávu. Nestatickou metodu voláme vždy pro konkrétní instancí a tato metoda může pracovat s jejími daty.  Při volání kvalifikujeme jméno nestatické metody jménem instance, pro kterou ji voláme:

x.getInt()

**this**: Při volání nestatických metod potřebujeme, aby v těle těchto metod program znal instanci, pro kterou je metoda volána, jinak by nedokázal určit, kde vzít složky, se kterými má pracovat. K tomu slouží klíčové slovo *this*, jenž v těle nestatické metody představuje odkaz na instanci, pro niž je metoda volána. Hodnota *this*se nestatickým metodám předává jako skrytý parametr = jako parametr, který nedeklarujeme, protože tam vždy bude.Parametr *this*mají i konstruktory.

**Statické** **metody**: Na volání statické metody se můžeme dívat jako na odezvu na zprávu poslanou třídě (nikoli jednotlivé instanci).

Když chceme vytvořit nový objekt, musíme poslat zprávu třídě. Když chceme vědět, kolik instancí dané třídy existuje, tak musíme poslat zprávu třídě.

Statické metody nevoláme pro žádnou konkrétní instanci. Při volání této metody z jiné třídy ji musíme kvalifikovat jménem třídy. Ve statické metodě můžeme vytvořit instanci a volat pro ni nestatické metody.

K překryvným i nepřekryvným metodám třídy z rodičovské třídy přistupujeme pomocí slova **super**.

**Pokud** není v rodiči konstruktor bez parametrů nebo implicitní, tj. **je v rodiči konstruktor alespoň s jedním parametrem, potom konstruktor potomka musí existovat a jako svůj první příkaz musí volat pomocí *super* konstruktor rodiče.**

**Finální metody - nechceme, aby bylo možné metodu překrýt.** Tomu lze zabránit slovem **final**.

**Abstraktní metody a třídy - chceme, aby bylo nutné metodu překrýt.**Vynutíme ve zděděné metodě použitím slova abstract u metody v rodičovské třídě.

Jakmile je minimálně jedna metoda označení abstract, pak nelze vytvořit instanci této třídy a z toho důvodu musí být celá třída označená jako abstract.

**Finální třídy - nechceme, aby bylo možné třídu zdědit.**Překrytímu určité metody zabráníme použitím slova final. Pokud použijeme final na třídu, pak tato třída se nedá zdědit a bere se za koncovou (=finální) třídu. Označení metody jako final nevynucuje označení celé třídy jako final.

10. Java - charakteristika, syntaxe, použití, vlastnosti, VM (=Virtual Machine)

**Charakteristika, použití Java** je [objektově orientovaný](https://cs.wikipedia.org/wiki/Objektov%C4%9B_orientovan%C3%A9_programov%C3%A1n%C3%AD) [programovací jazyk](https://cs.wikipedia.org/wiki/Programovac%C3%AD_jazyk), který vyvinula firma [Sun Microsystems](https://cs.wikipedia.org/wiki/Sun_Microsystems).Díky své [přenositelnosti](https://cs.wikipedia.org/wiki/Portace_softwaru) je používán pro [programy](https://cs.wikipedia.org/wiki/Po%C4%8D%C3%ADta%C4%8Dov%C3%BD_program), které mají pracovat na různých systémech počínaje čipovými kartami, přes [mobilní telefony](https://cs.wikipedia.org/wiki/Mobiln%C3%AD_telefon) a různá zabudovaná zařízení, aplikace pro desktopové počítače (platforma [Java SE](https://cs.wikipedia.org/wiki/Java_SE)) až po rozsáhlé distribuované systémy pracující na řadě spolupracujících počítačů rozprostřené po celém světě. Později Sun uvolnil zdrojové kódy Javy. Java je dále vyvíjena jako [**open source**](https://cs.wikipedia.org/wiki/Otev%C5%99en%C3%BD_software).V pozdějších verzích Javy nebyl mezikód přímo interpretován, ale před prvním svým provedením dynamicky zkompilován do [strojového kódu](https://cs.wikipedia.org/wiki/Strojov%C3%BD_k%C3%B3d) daného [počítače](https://cs.wikipedia.org/wiki/Po%C4%8D%C3%ADta%C4%8D)  – [JIT](https://cs.wikipedia.org/wiki/Just-in-time_kompilace).Abychom mohli vyvíjet a spouštět programy v Javě, potřebujeme vývojové nástroje označované zkratkou **JDK (Java Development Kit).** JDK lze zdarma získat na webových stránkách firmy Oracle.

**Vlastnosti Javy**

* jeho syntaxe je zjednodušenou a upravenou verzí syntaxe jazyka [C](https://cs.wikipedia.org/wiki/C_(programovac%C3%AD_jazyk)) a [C++](https://cs.wikipedia.org/wiki/C%2B%2B).
* *objektově orientovaný* – s výjimkou osmi primitivních datových typů jsou všechny ostatní datové typy [objektové](https://cs.wikipedia.org/wiki/Objektov%C4%9B_orientovan%C3%A9_programov%C3%A1n%C3%AD).
* *distribuovaný* – je navržen pro podporu aplikací v síti, práce se vzdálenými soubory, umožňuje vytvářet distribuované klientské aplikace a servery
* ***interpretovaný* – místo skutečného**[**strojového kódu**](https://cs.wikipedia.org/wiki/Strojov%C3%BD_k%C3%B3d)**se vytváří pouze tzv. [bajtkód](https://cs.wikipedia.org/wiki/Bajtk%C3%B3d" \o "Bajtkód)**. Tento formát je nezávislý na [architektuře počítače](https://cs.wikipedia.org/wiki/Architektura_po%C4%8D%C3%ADta%C4%8De) nebo zařízení. Program pak může pracovat na libovolném [počítači](https://cs.wikipedia.org/wiki/Po%C4%8D%C3%ADta%C4%8D) nebo zařízení**,** který **má k dispozici**[**interpret**](https://cs.wikipedia.org/wiki/Interpret_(software))**Javy, tzv.**[**virtuální stroj Javy**](https://cs.wikipedia.org/wiki/Java_Virtual_Machine)**–** **Java Virtual Machine (JVM).** **Existují též**[**mikroprocesory**](https://cs.wikipedia.org/wiki/Mikroprocesor)**, které dokáží spustit Javu hardwarově** namísto softwarové emulace [Java Virtual Machine](https://cs.wikipedia.org/wiki/Java_Virtual_Machine).

• je určen pro psaní vysoce spolehlivého [softwaru](https://cs.wikipedia.org/wiki/Software) – z tohoto důvodu neumožňuje některé programátorské konstrukce, které bývají častou příčinou chyb (např. [správa paměti](https://cs.wikipedia.org/wiki/Spr%C3%A1va_pam%C4%9Bti), příkaz **goto**, používání ukazatelů).

* *generační správa paměti* – správa paměti je realizována pomocí automatiky, která vyhledává již nepoužívané části paměti a uvolňuje je pro další použití.
* má vlastnosti, které chrání počítač v [síťovém prostředí](https://cs.wikipedia.org/wiki/Po%C4%8D%C3%ADta%C4%8Dov%C3%A1_s%C3%AD%C5%A5), na kterém je program zpracováván, před nebezpečnými operacemi nebo napadením vlastního operačního systému nepřátelským kódem.
* vytvořená aplikace běží na libovolném operačním systému nebo libovolné architektuře. **Ke spuštění programu je potřeba pouze to, aby byl na dané platformě instalován správný**[**virtuální stroj**](https://cs.wikipedia.org/wiki/Java_Virtual_Machine) **(VM).** Podle konkrétní platformy se může přizpůsobit vzhled a chování aplikace.
* vedle zmíněné nezávislosti na architektuře je jazyk nezávislý i co se týká vlastností základních [datových typů](https://cs.wikipedia.org/wiki/Datov%C3%BD_typ) (je například explicitně určena vlastnost a velikost každého z primitivních datových typů).
* do strojového kódu se překládá jen ten kód, který je opravdu zapotřebí.
* *víceúlohový* – podporuje zpracování vícevláknových aplikací
* knihovna může být dynamicky za chodu rozšiřována o nové třídy a funkce, a to jak z externích zdrojů, tak vlastním programem.

**VM** (Virtual Machine)Jazyk Java je navržen tak, aby mohl běžet na jakékoliv kombinaci hardware a operačního systému. Proto je zdrojový kód překládán do tzv. bajtového kódu jazyka, který lze pokládat za strojový kód fiktivního procesoru. V prostředí, kde chceme tento program spustiti, musí být instalován **javský** **virtuální stroj (JVM)**, který interpretuje přeložený program.

**Syntaxe** jazyka Java je odvozena od jazyka C++.  Program začíná prvním příkazem metody se jménem **main()**a končí, když metoda skončí. **Metoda je statická** - to znamená, že ji lze volat, i když neexistuje žádná isntance třídy, která ji obsahuje.

V Javě rozlišujeme dvě základní skupiny typů: primitivní typy (čísla, znaky, logické hodnoty) a odkazové typy (třídy, pole, rozhraní). **Deklarace proměnné odkazového typu vždy znamená pouze vytvoření odkazu na proměnnou, nikoli proměnné - tu musíme vytvořit samostatně.**

Všechny třídy v Javě jsou odvozeny od společného předka = třídy **Object**. To znamená, že všechny obsahují určité základní metody.

viz otázka číslo 8 + 9 syntaxe třídy, vlastností, metod, …