EDICE
myslíme v ...

knihovna programátora

- Podrobný výklad vlastností jazyka od naprostých základů až po pokročilé, běžně neprobírané konstrukce
- Probírá i konstrukce, které budou zařazeny až do příštích verzí jazyka, a předvádí, jak tyto konstrukce vyzkoušet
- Vysvětluje nejenom jak probírané konstrukce používat, ale také proč jsou právě takové
- Využívá zabudované REPL prostředí pro demonstraci vykládaných konstrukcí bez zbytečného pomocného kódu
- Může sloužit současně jako učebnice i referenční příručka

RUDOLF PECINOVSKÝ

# Java 14

## Kompletní příručka jazyka

GRADA

**Ing. Rudolf Pecinovský, CSc.** je absolventem *Fakulty Elektrotechnické ČVUT* z roku 1979. Titul CSc. získal v *Ústavu teorie informace a automatizace ČSAV* v roce 1983. Od počátku 80. let učí a publikuje, přičemž svůj výzkum soustředí především na oblast vstupních kurzů moderního programování pro naprosté začátečníky. V současné době učí na *Vysoké škole ekonomické v Praze*, na *Fakultě jaderné a fyzikálně inženýrské ČVUT* a na *Vysoké škole podnikání a práva*. Doposud mu vyšlo přes 60 knih, které byly přeloženy do pěti jazyků. Většina jeho knih je zaměřena na výuku moderního programování a na umění návrhu objektově orientované architektury.

knihovna programátora

RUDOLF PECINOVSKÝ

# Java 14

## Kompletní příručka jazyka

GRADA
Publishing

**Rudolf Pecinovský**

# Java 14
## Kompletní příručka jazyka

Vydala Grada Publishing, a.s.
U Průhonu 22, Praha 7
obchod@grada.cz, www.grada.cz
tel.: +420 234 264 401
jako svou 7620. publikaci

Odpovědný redaktor: Ivana Palasová
Návrh vnitřního layoutu Rudolf Pecinovský
Zlom Rudolf Pecinovský
Počet stran 578
První vydání, Praha 2020
Vytiskly Tiskárny Havlíčkův Brod, a.s.



ISBN 978-80-271-1605-8 (ePub)
ISBN 978-80-271-1604-1 (pdf)
ISBN 978-80-271-1369-9 (print)

*Všem, kteří se chtějí něco naučit*

# Stručný obsah

# Podrobný obsah

## Část III Pokročilé objektové konstrukce 387

# Úvod

Tato kniha seznamuje se čtrnáctou verzí jazyka *Java*. Soustředí se především na výklad vlastností jazyka a minimalizuje výklad probírající obsah standardní knihovny – tomu se budou věnovat další příručky z této série.

## Komu je kniha určena

Kniha je určena všem, kteří se chtějí naučit jazyk *Java*. Nevyžaduje od čtenáře žádné předchozí znalosti programování a snaží se, byť stručně, vysvětlit vše, co je potřeba.

Naprostým začátečníkům bych sice doporučil, aby začali s některou z mých příruček úvodu do programování, v nichž nevysvětluji, jak se program zapisuje, ale soustředím se především na to, jak se vymýšlí. V těchto příručkách spolu navrhujeme architekturu programu. Zakódování navrženého programu (tj. jeho zápis v programovacím jazyce) přenecháváme generátoru kódu, který je integrální součástí použitého vývojového prostředí. K výkladu zápisu programu v kódu, na nějž se soustředí tato učebnice, pak přejdeme až v okamžiku, kdy složitost navrhovaných programů překročí možnosti onoho generátoru. V té době ale již mají účastníci kurzu základy objektově funkcionálního paradigmatu zažité a nedělají proto chyby, s nimiž zápasí studenti, kteří začali studiem syntaxe použitého programovacího jazyka.

Na druhou stranu nechci nikoho zrazovat, a proto jsem se v této knize pokusil důkladně vysvětlit i základní programové konstrukce a doporučené způsoby jejich využití, takže i naprostí začátečníci zde najdou všechny potřebné informace.

Jak už jsem řekl, kniha se soustředí na výklad jazyka. Tím, že výklad knihoven (např. práce s kolekcemi, datovody, soubory a datovými proudy, regulárními výrazy či vlákny) odložím do jiných příruček, ušetřím prostor, který mohu věnovat podrobnému výkladu těch vlastností jazyka, na které v jiných příručkách často nezbývá místo. Jejich neznalost ale vede k hodinám marného pátrání po skutečném původci vzniklé chyby a následným experimentálním změnám programu s myšlenkou: „Co kdyby to náhodou vyšlo?“

## Koncepce výkladu

Značná část učebnic začíná definicí programu typu *Hello world*, což je koncepce převzatá z historické učebnice jazyka C vydané v roce 1978. Základní nevýhodou této koncepce je, že na počátku používá několik programových konstrukcí, které čtenáři vysvětlí až někde v průběhu dalšího výkladu.

Tehdy to dost dobře jinak ani nešlo. Od té doby se objevila řada nástrojů, které umožňují koncipovat výklad tak, aby se v něm používaly pouze konstrukce, které již byly vysvětleny. O to se pokouším i v této knize a používám doposud nevysvětlené konstrukce opravdu výjimečně pouze v situacích, kdy to výrazně zpřehlední výklad. Aby se mi to podařilo, používám k výkladu program *JShell*, který je standardní součástí vývojářské sady.

Knihu jsem se navíc pokusil napsat tak, aby mohla sloužit stejně dobře jako učebnice jazyka i jako referenční příručka, ve které by bylo možno v případě potřeby najít klíčové informace rychleji a snáze než na internetu. Pasáže vysvětlující jednotlivé programové konstrukce jsem se proto snažil jasně oddělit od pasáží probírajících teorii, jak řešit tu kterou třída problémů, a sloužících především začátečníkům, kteří ještě nemají s programováním žádné zkušenosti.

## Rozdělení textu

Text je rozdělen do čtyř částí. První část začíná úvodem do prostředí *JShell*, které bude v celé první části využíváno ke spouštění demonstračních programů. Zbytek první části pak probírá algoritmické konstrukce a prostředky, které *Java* nabízí k jejich realizaci.

Druhá část se soustředí na základy objektově orientovaného programování. Začíná teoretickou kapitolou vysvětlující hlavní zásady objektově orientovaného paradigmatu. Po ní následují kapitoly postupně probírající základní objektové konstrukce a jejich účel.

Třetí část je věnována výkladu nadstavbových objektových konstrukcí včetně rozboru některých základních pravidel, jejichž nedostatečné pochopení dělá studentům často problémy. Závěr třetí části je věnován koncepci modularity. Modularita sice není součástí jazyka, ale platformy; nicméně její osvojení přináší výrazné zvýšení spolehlivosti vyvíjených programů a do jisté míry i efektivity jejich vývoje i následného provozu.

Čtvrtá část obsahuje přílohy. První z nich vás seznámí s omezeními jazyka *Java*, druhá pak ukáže, jak se v *Javě* navrhuje GUI. Nejprve představí nástroje použitelné pro superjednoduché GUI, pak principiálně naznačí, jak se navrhuje komplexní GUI.

# Terminologie

Při zavádění české terminologie se bohužel potýkáme často s tím, že v anglické literatuře zavádějí mnozí autoři terminologii vlastní. V oblasti, kterou se má zabývat tato publikace, je terminologický guláš obzvlášť vydatný. Budu-li proto uvádět ve svém výkladu anglické ekvivalenty zaváděných termínů, budu používat terminologii zavedenou v oficiální referenční publikaci *Java® Language Specification – Java SE 14 Edition* ([5]), na níž se budu občas odvolávat zkratkou JLS.

# Použité nástroje

Pro úspěšné studium knihy budete potřebovat několik programů, které musíte nejprve stáhnout a instalovat.

## Vývojová sada JDK

Především budete potřebovat mít instalovanou vývojovou sadu pro *Javu 14* označovanou JDK, což je zkratka z anglického *Java Development Kit*. Stáhnete ji ze stránky http://www.oracle.com/technetwork/java/javase/downloads/. Jenom musíte před vlastním stažením nastavit přepínač u seznamu stáhnutelných souborů do polohy **Accept License Agreement**, protože jinak vás stránka daný program stáhnout nenechá.

Spolu s vývojovou sadou vřele doporučuji stáhnout i dokumentaci. Tu najdete na téže stránce, jenom musíte popojet trochu níže do sekce **Additional Resources**, ve které najdete podsekci **Java SE 14 Documentation**.

## Vývojové prostředí *JShell*

Pro většinu výkladu budu využívat prostředí *JShell*, které je součástí JDK a se kterým vás seznámí úvodní kapitola. (Podrobnější seznámení s tímto prostředím najdete v příručce *Java 9 – JShell*.) Výhodou tohoto přístupu je, že součástí programů nemusí být nejrůznější vata, bez které byste standardní program v *Javě* nespustili. Navíc již nemusíte instalovat žádné další vývojové prostředí, takže nebudete muset zápasit s důsledky známé skutečnosti, že zvládnutí současných profesionálních vývojových prostředí je náročnější než zvládnutí základů jazyka.

V závěru knihy budu potřebovat vysvětlit některé konstrukce, které se v prostředí *JShell* dost dobře vysvětlit nedají. Až na to dojde, tak vás upozorním.

## Samostatné vývojové prostředí

Od kapitoly *12 Vývojová prostředí a vytvoření aplikace* na straně 243 začnu vedle prostředí *JShell* používat i prostředí *NetBeans*. Budu v něm demonstrovat vlastnosti, které se projeví až u samostatně vyvíjených programů. Nelpím přitom na tom, abyste používali právě toto prostředí, ale doporučuji je těm, kteří ještě nemají s profesionálními vývojovými prostředími žádné zkušenosti.

Adresy, z nichž si můžete stáhnout některé z doporučovaných prostředí, najdete v pasáži *Nejpoužívanější IDE* na straně 244. Každopádně budete-li chtít experimentovat s vybraným profesionálním prostředím hned od počátku, nic vám v tom nebrání.

## Doprovodné programy

Všechny doprovodné programy zmiňované a používané v textu najdete na stránce knihy na adrese http://knihy.pecinovsky.cz/62_j14lang. Zde jsou umístěny ZIP-soubory obsahující doprovodné programu nebo jejich odvozeniny.

Doprovodné programy jsou zpočátku skripty prostředí *JShell*, v druhé části to pak jsou projekty pro vývojové prostředí *NetBeans*, které lze snadno transformovat na projekty pro vaše oblíbené vývojové prostředí. Podrobnosti o struktuře doprovodných programů najdete stránce s programy.

Knihu jsem se snažil zalomit tak, aby všechny výpisy programů a komunikace s počítačem, které se vejdou na jednu stránku, byly vždy na jedné stránce a nelámaly se přes hranu stránky. Čtenářům papírové verze by se tak měly minimalizovat problémy při studiu doprovodných textů diskutujících obsah těchto výpisů.

Čtenáři elektronické verze jsou na tom s listováním hůře. Těm bych doporučil, aby si text knihy otevřeli dvakrát a umístili oba dokumenty vedle sebe. (Na většině současných počítačů s širokými monitory by to neměl být problém.) Pak lze mít v jednom dokumentu otevřenou stránku s výpisem a ve druhém dokumentu stránku s rozborem obsahu daného výpisu.

Druhou možností je, že si studovaný výpis vytisknete (v doprovodných programech najdete příslušné soubory i s čísly řádků výpisu v knize), budete se dívat do výpisu a na čtečce číst doprovodný text.

## Předběžné definice nových konstrukcí

Od verze 10 bylo zavedeno vydávání nových verzí dvakrát do roka: na jaře (typicky v březnu) vycházejí sudé verze, a na podzim (typicky v říjnu) verze liché. Protože dotažení návrhu některých konstrukcí trvá delší dobu, bylo v zájmu maximalizace zpětné vazby od uživatelů zavedeno, že vydané verze podporují i konstrukce, které jsou teprve ve vývoji, a nedoporučuje se je proto používat ve finálních verzích aplikací.

Ve standardní instalaci JDK se můžete setkat se třemi typy rozpracovaných konstrukcí, funkcionalit a aplikací:

#### Předběžná funkce/konstrukce/vlastnost (Preview Feature[1])

Jako předběžná je označena konstrukce jazyka nebo funkce VM, která je plně specifikována, plně implementována, ale nemusí být ve své definitivní verzi. Má iniciovat zpětnou vazbu vývojářů, kteří se ji pokusí použít ve svých aplikacích.

Při standardním nastavení tyto budoucí novinky překladač ani běhové prostředí nepodporují. Jejich podporu překladačem, resp. virtuálním strojem, resp. programem `javadoc`, nastavíte zadáním parametru

```
--enable-preview
```

příslušného programu (překladače, virtuálního stroje, programu javadoc).

#### Experimentální funkce/konstrukce/vlastnost (Experimental Feature)

Experimentální funkce představují rané verze funkcí, většinou na úrovni VM. Tyto funkce mohou být neúplné nebo dokonce nestabilní. Ve většině případů potřebují povolit pomocí vyhrazených příznaků.

Experimentální prvek je považován za zhruba z 25 % „hotový“, zatímco předběžný prvek by měl být „hotový“ alespoň z 95 %.

#### Inkubační funkce/konstrukce/vlastnost (Incubating Feature, Incubator[2])

Inkubační funkce jsou experimentální API distribuované ve formě samostatných modulů se jmény s předponou `jdk.incubator`. Abyste s nimi mohli pracovat, musíte tyto moduly explicitně přidat.

## Syntaktické definice a diagramy

V testu je uváděna řada nejrůznějších programových konstrukcí, jejichž zápis se řídí přesně danými syntaktickými pravidly. Většina učebnic možné způsoby zápisu zpravidla formálně nedefinuje, anebo používá syntaktické definice odvozené z Backusovy normální formy. Syntaktické definice jsou výhodné pro strojové zpracování, lidé se v nich často špatně orientují. Rozhodl jsem se proto zobrazovat syntaktická pravidla zápisu jednotlivých konstrukcí pomocí syntaktických diagramů.

Syntaktický diagram ukazuje, jak je možno zobrazovanou konstrukci zapsat. Pojedete-li po čarách, tak jakýkoliv průjezd generuje syntakticky správnou konstrukci. Toho, kdo syntaktické diagramy ještě nezná a chtěl by rychle některý vidět, bych odkázal např. na diagram na obrázku 2.1 na straně 56.

---

[1] Podrobněji je koncepce předběžných vlastností popsána v JEP 12, které najdete na adrese https://openjdk.java.net/jeps/12.

[2] Podrobněji se o koncepci inkubačních modulů dozvíte v JEP 11, které najdete na adrese https://openjdk.java.net/jeps/11.

# Použité typografické konvence

K tomu, abyste se v textu lépe vyznali a také abyste si vykládanou látku lépe zapamatovali, používám několik prostředků pro odlišení a zvýraznění textu.

**Termíny** První výskyt nějakého termínu a další texty, které chci zvýraznit, vysazuji **tučně**.

*Název* Názvy firem a jejích produktů vysazuji *kurzivou*. Kurzivou vysazuji také názvy kapitol, podkapitol a oddílů, na které se v textu odkazuji.

Odkaz Celá kniha je prošpikovaná křížovými odkazy na související pasáže. Není-li odkazovaný objekt (kapitola, obrázek, výpis programu, …) na některé ze sousedních stránek, je pro čtenáře tištěné verze doplněn o číslo stránky, na níž se nachází. Čtenářům elektronické verze stačí, když na něj klepnou, a použitý prohlížeč by je měl na odkazovaný objekt ihned přenést.

**Citace** Texty, které si můžete přečíst na displeji, např. názvy polí v dialogových oknech či názvy příkazů v nabídkách, vysazuji **tučným bezpatkovým písmem**.

Adresy Názvy souborů a internetové adresy vysazuji obyčejným bezpatkovým písmem.

`Program` Identifikátory a další části programů zmíněné v běžném textu vysazuji `neproporcionálním písmem`, které je v elektronických verzích pro zvýraznění tmavě červené.

`metoda(?)` Při odkazech na metody budu v závorkách za názvem metody vždy uvádět seznam typů jejich parametrů – např. `equals(Object)`. Nebude-li v danou chvíli jasné, jaké má zmiňovaná metoda parametry, budu do závorek psát otazník – např. `metoda(?)`.

**`zadání`** Ve výpisech komunikace s prostředím *JShell* bude zadání uživatele vysazeno tučně (v elektronických verzích pro zvýraznění tmavě červeně), a odpovědi prostředí netučně (a černě).

*`Komentář`* Pokud se v programech objeví komentáře, budou podbarveny zeleně.

Kromě výše zmíněných částí textu, které považuji za důležité zvýraznit nebo alespoň odlišit od okolního textu, najdete v textu ještě řadu doplňujících poznámek a vysvětlivek. Všechny budou v jednotném rámečku, který bude označen ikonou charakterizující druh informace, kterou vám chce poznámka či vysvětlivka předat.

Symbol jing-jang bude uvozovat poznámky, s nimiž se setkáte na počátku každé kapitoly. Zde vám vždy prozradím, co se v dané kapitole naučíte.

Píšící ruka označuje obyčejnou poznámku, která pouze doplňuje informace z hlavního proudu výkladu o nějakou zajímavost.

Symbol znamení raka označuje upozornění, že následný výklad se týká funkcionality, která je prozatím definována jako předběžná (preliminary). Lze očekávat, že bude v některé z příštích verzí zprovozněna, ale výsledné vlastnosti se mohou od stávajících lišit.

Otevřená schránka s dopisy označuje informace o projektu, s nímž budeme v dalším textu pracovat, nebo v něm najdete vzorové řešení aplikující probranou látku. Příslušný projekt získáte pomocí generátoru projektů popsaného výše.

Obrázek knihy označuje poznámku týkající se používané terminologie. Tato poznámka většinou upozorňuje na další používané termíny označující stejnou skutečnost nebo na konvence, které se k probírané problematice vztahují. Seznam všech terminologických poznámek najdete v rejstříku pod heslem „terminologie“.

Ruka s hrozícím prstem upozorňuje na věci, které byste měli určitě vědět a na které byste si měli dát pozor, protože jejich zanedbání vás většinou dostane do problémů.

Usměváček vás bude upozorňovat na různé tipy, kterými můžete vylepšit svůj program nebo zefektivnit svoji práci.

Mračoun vás naopak bude upozorňovat na různá úskalí programovacího jazyka nebo programů, s nimiž budeme pracovat, a bude vám radit, jak se těmto nástrahám vyhnout či jak to zařídit, aby vám alespoň pokud možno nevadily.

Brýle označují tzv. „poznámky pro šťouraly“, ve kterých se vás snažím seznámit s některými zajímavými vlastnostmi probírané konstrukce nebo upozorňuji na některé souvislosti, avšak které nejsou k pochopení látky nezbytné.

### Odbočka – podšeděný blok

Občas je potřeba vysvětlit něco, co nezapadá přímo do okolního textu. V takových případech používám podšeděný blok se silnou čarou po straně. Tento podšeděný blok je takovou drobnou odbočkou od ostatního výkladu. Nadpis podšeděného bloku pak najdete i v podrobném obsahu mezi nečíslovanými nadpisy.

## Zpětná vazba

Přes veškeré úsilí, které jsme knize já i moji spolupracovníci věnovali, nemohu vyloučit, že v textu či doprovodných příkladech zůstaly skryté nějaké chyby. Předem se za ně omlouvám. Objevíte-li proto v knize nějakou chybu nebo budete-li mít návrh na nějaké její vylepšení, pošlete prosím e-mail s předmětem 62_Java14_DOTAZ na adresu rudolf@pecinovsky.cz. Na stránku knihy http://knihy.pecinovsky.cz/62_j14lang se pak pokusím co nejdříve zanést příslušná errata s opravou, kterou zapracujeme do případného dalšího vydání.

Tento mail pošlete i v případě, když vám bude někde připadat text nepříliš srozumitelný nebo budete-li mít nějaký dotaz, ať už k vykládané látce či použitému vývojovému prostředí. Bude-li se dotaz týkat něčeho obecnějšího, zveřejním na stránce knihy odpověď i pro ostatní, které by mohl obdobný dotaz napadnout za pár dní, anebo jsou natolik ostýchaví, že si netroufnou sami se zeptat.

Dopředu se omlouvám, že vzhledem k velkému pracovnímu zatížení občas odpovídám na maily až se značným zpožděním.

# Část I: Neobjektové konstrukce

Tato část vás nejprve seznámí s prostředím *JShell*, které budu ve svém příkladu využívat k tomu, abychom získali okamžitou odpověď na zadané programové obraty. Budeme tak moci od začátku zkoušet probírané konstrukce, aniž bychom museli používat něco, co jsme ještě neprobrali. Poté postupně probereme algoritmické konstrukce, které vyšší programovací jazyky nabízely ještě před příchodem objektového paradigmatu, kterému bude věnována druhá část.

# Kapitola 1
# Prostředí JShell

**Co se v kapitole naučíte**

Tato kapitola vás seznámí s prostředím *JShell*, které bude v následujících kapitolách používáno pro demonstraci základních programových konstrukcí jazyka a jeho pravidel. V závěru pak ještě vysvětlí některá nejzákladnější syntaktická pravidla jazyka a seznámí vás s komentáři, kterými budou doprovázeny uložené výpisy programů.

Tato kapitola bude obsahovat jen velmi stručný úvod, jenž vás seznámí s klíčovými vlastnostmi prostředí *JShell*, které budu v následujícím výkladu používat. Jak už bylo řečeno v úvodu, podrobné informace o tomto prostředí včetně návodu na jeho začlenění do vlastního programu najdete v publikaci *Java 9 – JShell*.

## 1.1 Nejprve trocha terminologie

Než se pustíme do vlastního výkladu, bylo by vhodné se seznámit s několika základními termíny, které musíme používat daleko dříve, než vstřebáte dostatek vědomostí, abyste je pochopili v celé jejich šíři.

#### Objekt

Programovací jazyk *Java* je postaven nad základní myšlenkou objektově orientovaného programování (v dalším textu budu často používat zkratku OOP), že *všechno je objekt*. A když říkám všechno, tak myslím opravdu všechno. Kdykoliv se proto tento termín v textu objeví, dosaďte si za něj „*cokoliv, s čím můžeme v programu pracovat*“, anebo „*cokoliv, co můžete nazvat podstatným jménem*“.

#### Třída, datový typ

Objekty řadíme podle jejich vlastností a schopností do různých skupin, které označujeme jako třídy, resp. datové typy. Termíny *třída* a *datový typ* můžeme do jisté míry považovat za synonyma.

Každý objekt, s nímž se potkáte, je objektem nějakého datového typu. Datový typ objektu říká, co je objekt zač, jaké má vlastnosti a schopnosti – jinými slovy: co po něm můžeme chtít.

#### Proměnná

Proměnná je místo v paměti, kam si můžeme uložit nějakou hodnotu pro budoucí použití.

#### Atributy × metody

Objekty mají svoje atributy a svoje metody. Termínem *atribut* označujeme proměnnou, která je ve vlastnictví oslovovaného objektu a uchovává nějakou jeho charakteristiku – např. pozici, velikost či barvu.

Termínem *metoda* označujeme část kódu patřícího danému objektu. Metodu můžeme spustit a v případě potřeby od ní získat nějakou hodnotu a/nebo ji nechat provést nějakou akci.

## 1.2 Charakteristika programu a prostředí *JShell*

Program *JShell* zařazujeme do kategorie nástrojů typu REPL, což je zkratka z anglického *read-evaluate-print-loop* (česky: *cyklus přečti-vyhodnoť-vytiskni*). Tyto nástroje se používají jako poskytovatelé prostředí (platformy) pro rychlou interakci s podkladovým prostředím, kterým může být jak operační systém, tak nějaký program. Uživatel platformy typu REPL může komunikovat s podkladovým prostředím v programovacím jazyce dané platformy bez potřeby překladu a samostatného spuštění.

## 1.3 Problémy s klávesnicí

Už od svého prvního uvedení ve verzi 9 mělo prostředí *JShell* problém s některými jinými rozloženími kláves než US. V českém prostředí bylo možno až do verze 12 tyto problému řešit používáním rozložení označovaného jako *České (QWERTY)*. To používá řada programátorů, protože znaky, které se nevyskytují na klávesnici psacího stroje, má toto rozložení umístěny na stejných klávesách jako rozložení US, jenom se aktivují s jiným nastavením přeřaďovačů.

Bohužel se pak někdo rozhodl prostředí *JShell* „vylepšit“. Od té chvíle sice funguje rozložení označované jako *České*, ale pro změnu zase přestalo chodit rozložení *České*

*(QWERTY)*, které nám nyní neumožňuje zadat některé důležité znaky – např. složené závorky ({}) a „svislítko“ (|), a to ani tak, že je do textu vložíte ze schránky.

Vložit je můžete buďto přímým zadáním jejich kódu ({=123, |=124, }=125), anebo použitím jiného rozložení kláves.

## 1.4 Příprava programu *JShell* a první spuštění

Zdánlivě nejjednodušší je otevřít okno konzoly (terminálu) a přesunout se v něm do podsložky bin složky, do které jste instalovali JDK. Tam najdete soubor jshell.exe, který můžete spustit příkazem

```
jshell -v
```

kde parametrem -v spouštěný program žádáme, aby nám o všech provedených akcích referoval co nejpodrobněji. Budete-li chtít spustit *JShell* v režimu podporujícím předběžnou funkcionalitu příštích verzí (preview features), spustíte jej příkazem

```
jshell --enable-preview -v
```

Zabíhání do složky s programem *JShell* však většinu uživatelů obtěžuje. Chcete-li proto program používat opravdu jednoduše, pak se nabízejí dvě cesty:

- Upravit systémovou proměnnou PATH definující seznam složek, které systém při hledání zadaného souboru prohledá.
- Definovat jednoduchý skript, který program *JShell* spustí v aktuální složce a případně mu i zadá potřebné parametry.

Pro tento kurz dám přednost druhé cestě. Jak znám uživatele systému *Linux* a jeho verzí (sem patří i MacOS), tak ti dávkové soubory hojně používají a žádný výklad nepotřebují. Proberu proto tvorbu dávkových souborů pouze pro *Windows*, jejichž uživatelé o této možnosti často ani nevědí.

### Dávkové soubory pro *Windows*

*Windows* používají dva druhy dávkových souborů: z dob systému DOS převzaly soubory s příponou BAT (zkratka z anglického *batch* – dávka, skupina, …) a od verze *Windows* XP přidaly ještě soubory s příponou PS1 (PS je zkratka z názvu programu *PowerShell*, ale co tam dělá ta jednička, netuším).

Obě verze se spouští v konzolovém okně. Jeho velikost, umístění, použité písmo a barvy si můžete nastavit ve vlastnostech dosažitelných ze systémové nabídky okna[3]. My budeme používat první z nich, tj. soubor s příponou BAT. Postupujte následovně:

[3] Systémovou nabídku aktivujete klepnutím na ikonu na levém okraji titulkové lišty okna.

1. Otevřete složku, do které jste prostřednictvím generátoru doprovodných programů uložili soubor `Start.jsh`.
2. Ve svém oblíbeném textovém editoru[4] vytvořte prázdný textový soubor.
3. Do souboru napište příkaz

   ```
   chcp 1250
   ```

   který nastaví správnou kódovou stránku[5]. Česká a slovenská *Windows* totiž implicitně používají stránku 852, ale spuštěnému programu tvrdí, že používají stránku 1250. Proto je třeba na tuto kódovou stránku nastavit i příkazový panel.
4. Na další řádek napište příkaz

   ```
   <JDK>\bin\jshell -v
   ```

   resp.

   ```
   <JDK>\bin\jshell --enable-preview -v
   ```

   kde `<JDK>` zastupuje cestu ke složce, do které jste instalovali JDK *Javy*. Tím spustíte program *JShell* a jím definovanou platformu, kterou budeme v učebnici používat.
5. Na další řádek napište příkaz

   ```
   pause
   ```

   Ten zabezpečí, že po opuštění platformy *JShell* zůstane příkazový panel otevřen, abyste si mohli před jeho ukončením prohlédnout případné závěrečné informace.
6. Soubor uložte pod názvem `!_JShell.bat`. Vlastní jméno souboru není důležité (počáteční vykřičník má zabezpečit, aby se při seřazení podle názvu soubor zobrazoval jako první) – doporučuji jej pouze proto, abych se na něj mohl v dalším textu odvolávat. Důležitá je přípona `bat`.
7. Spusťte *JShell* poklepáním na právě vytvořený dávkový soubor.

Řada začínajících programátorů ponechává nastavení *Windows* pro laické uživatele, které matou přípony souborů. Proto jsou přípony známých typů implicitně skryty. Necháváte-li ale operační systém, aby před vámi skrýval přípony, tak se stává, že vám sice průzkumník ukazuje, že se ve složce nachází soubor `!_JShell.bat`, ale ve skutečnosti je tam uložen soubor `!_JShell.bat.txt`. Přípona `txt` je pro operační systém známá, a proto ji nezobrazuje. **Zabezpečte proto, aby se vám při práci s dávkovými soubory zobrazovaly jejich přípony.**

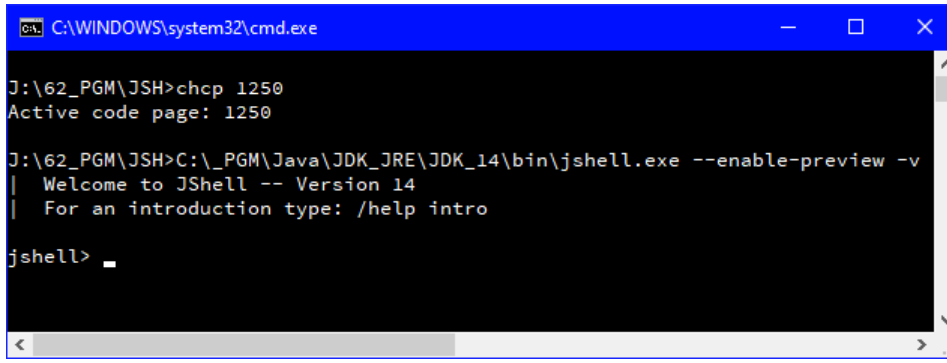
[4] Já používám program PAPad, který můžete stáhnout z adresy http://pspad.com v některé z osmi lokalizací. Dáváte-li ale přesnost jinému, nebudu vám bránit.

[5] Název `chcp` je zkratkou z `change code page`.

## Po spuštění

Po spuštění programu se otevře konzolové okno podobné oknu na obrázku 1.1. Lišit se budou pouze v cestách k aktuální složce vypisovaných na počátku prvního a čtvrtého řádku a v cestě k programu `jshell.exe` ve čtvrtém řádku. Uživatelům systémů *Linux* a *MacOS* budou chybět první tři řádky (nastavení kódové stránky, oznámení nastavené stránky a oddělující prázdný řádek).

Pravda, já mám na svém počítači doplněnou systémovou proměnnou `PATH` o cestu k programu *JShell*[6], takže jsem ve čtvrtém řádku celou cestu k programu `jshell.exe` uvádět nemusel. Chtěl jsem ale hlavně pro ty, kteří s konzolovým oknem ještě nemají velké zkušenosti, ukázat výpis kompletní.




***Obrázek 1.1:***
*Konzolové okno otevřené po spuštění dávky `!_JShell.bat` na mém počítači*

V dalším textu již nebudu ukazovat výpisy z okna ve formě obrázků, ale ve formě klasických výpisů programů s číslovanými řádky, abych se mohl na čísla řádků snáze odvolávat. Obsah okna na obrázku 1.1 je ve výpisu 1.1.

***Výpis 1.1:*** *Obsah konzolového okna otevřeného po spuštění dávky `!_JShell.bat` na mém počítači*

```
1 J:\62_PGM\JSH>chcp 1250
2 Active code page: 1250
3
4 J:\62_PGM\JSH>C:\_PGM\Java\JDK_JRE\JDK_14\bin\jshell.exe --enable-preview -v
5 |  Welcome to JShell -- Version 14
6 |  For an introduction type: /help intro
7
8 jshell>
```

Na posledním řádku výpisu je již výzva (anglicky prompt) programu/prostředí *JShell*. Za ní můžete začít psát své úryvky a příkazy.

[6] Návod, jak se to dělá, určený pro ty, kteří nemají s nastavováním této proměnné žádné zkušenosti, je v příloze výše zmíněné příručky *Java 9 – JShell*.

Nelekněte se, když se program *JShell* nespustí hned, přesněji když hned nevypíše své přivítání. Je to proto, že se na počátku načítá startovní skript a podle něj se prostředí konfiguruje. Přivítání se vypisuje až poté, co se startovní skript načte a provede.

## 1.5 Úryvky (snippets)

Výrazy (expressions), příkazy (statements), deklarace (declarations) a definice (definitions) jazyka *Java*, které za tuto výzvu napíšete, se hromadně označují jako **úryvky** (snippets). Kdykoliv napíšete nějaký úryvek, prostředí *JShell* jej ihned vyhodnotí a uloží výsledek. Ve výpisu 1.2 najdete zadání dále popsaných úryvků spolu s odpověďmi prostředí.

Zkuste napsat jednoduchý aritmetický výraz, např. `6+5` (řádek `1` výpisu) a potvrdit jej stiskem `<ENTER>`. Jak už jsem řekl, prostředí *JShell* výraz spočítá a na dalším řádku oznámí jak výsledek, tak jeho uložení do pomocné proměnné nazvané `$1`.

Úplným začátečníkům prozradím, že **proměnná** je místo v paměti, kam si můžeme uložit nějakou hodnotu pro budoucí použití. Proměnných může být v programu více, a proto má každá přidělené svoje jméno. Když pak později chceme uloženou hodnotu využít, odvoláme se na ni jménem příslušné proměnné.

Prostředí *JShell* nám navíc na řádku `3` přidalo informaci, že vytvořená pomocná proměnná je typu `int`, což je zkratka z anglického *integer* a prozrazuje to, že obsah oné proměnné bude v programu vždy interpretován jako celé číslo.

***Výpis 1.2:*** *První tři pokusné úryvky*

```
1  jshell> 6+5
2  $1 ==> 11
3  |  created scratch variable $1 : int
4
5  jshell> 6 +
6     ...> 7+
7     ...> 9
8  $2 ==> 22
9  |  created scratch variable $2 : int
10
11 jshell> $1+$2
12 $3 ==> 33
13 |  created scratch variable $3 : int
14
15 jshell>
```

Úryvky se nemusejí nutně vejít na jeden řádek. Když prostředí vyhodnotí, že zadávání úryvku ještě neskončilo (ve výpisu 1.2 řádky 5 a 6), zahájí další řádek pokračovací výzvou. Vyhodnocení zadaného výrazu zahájí až v okamžiku, kdy je zadání zkompletováno, k čemuž dojde na řádku 7, takže se na řádcích 8 a 9 dozvíme výsledek.

## Použití proměnných

Jak jste si jistě všimli, na řádku 11 jsem místo čísel použil názvy dříve vytvořených proměnných. V poznámce jsem uvedl, že když se objeví ve výrazu název proměnné, překladač danou proměnnou najde a na dané místo ve výrazu vloží hodnotu, která je v proměnné uložena.

## Identifikace úryvků

*JShell* vytvořené úryvky postupně čísluje. Tato čísla se pak používají jako identifikátory úryvků. V dalším textu je budu označovat jako `ID` úryvků.

Obsahuje-li zadaný úryvek výraz, *JShell* tento výraz vyhodnotí a pro výsledek vytvoří novou pomocnou proměnnou, kterou pojmenuje znakem `$` (dolar) následovaným `ID` zadaného úryvku. O tom jste se mohli přesvědčit např. ve výpisu 1.2 na řádcích 3, 9 a 13.

Definujeme-li úryvek, který obsahuje nějaký pojmenovaný objekt (proměnnou, metodu, datový typ), tak se v příkazech pro prostředí *JShell* můžeme na tento objekt odvolávat jak jeho jménem, tak prostřednictvím `ID` jeho úryvku (až budeme probírat příkazy pro prostředí *JShell*, tak si to předvedeme).

## Středník

*Java* převzala od jazyka C pravidlo, že každý příkaz musíme ukončit středníkem. Platí, že tento ukončovací středník dělá z výrazu příkaz, který je třeba provést.

Zapomenutý středník patří mezi „oblíbené“ chyby. Zadáme-li proto výraz bez ukončujícího středníku (byly tak zadány všechny doposud zadané výrazy), překladač ohlásí chybu. *JShell* se v takovém případě podívá, jestli by nepomohlo přidání středníku na konec řádku. Pokud je po přidání středníku překladač spokojen, *JShell* nás žádným chybovým hlášením neobtěžuje, prostě si daný úryvek zapamatuje i s přidaným středníkem.

## Více objektů na řádku, zavlečené chyby

Definujeme-li více pojmenovaných objektů na jednom řádku, *JShell* vytvoří pro každý z těchto objektů samostatný úryvek. Aby však poznal, kde jedna definice končí a jiná začíná, musíme všechny definice s výjimkou té poslední ukončit středníkem. Poslední

definici středníkem ukončovat nemusíme (ale samozřejmě můžeme), protože, jak jsme si řekli před chvílí, závěrečný středník je *JShell* ochoten vložit na konec řádku za nás.

Ve výpisu 1.3 je na řádku 1 pokus vložit více úryvků, aniž bych první ukončil středníkem. Jak vidíte, *JShell* ohlásil na řádku 2 chybu, na řádku 3 vysvětlil, že mu chybí očekávaný středník a na řádku 5 dokonce ukázal, kde si myslí, že by se měl onen středník vložit do úryvku vypsaného na řádku 4.

Pak se sice pokusil vyhodnocovat dál, ale první chyba jej poněkud rozhodila, takže další chyby, na něž mne na následujících řádcích upozorňuje, jsou tzv. **zavlečené chyby**, což jsou chyby vzniklé v důsledku špatného pochopení programu zapříčiněného předchozí chybou.

Výpis chyb končí na řádku 20 výzvou k zadání dalšího úryvku. Když jsem poslechl radu z řádků 4 a 5, doplnil středník a zadal za výzvu upravený příkaz, *JShell* upravené zadání akceptoval a vytvořil proměnné $4 a $5, do nichž uložil hodnoty výrazů v zadaných úryvcích.

Jak víme, proměnná $4 vznikla při vyhodnocení prvního úryvku na daném řádku. Důležité však je, že v okamžiku, kdy se začal vyhodnocovat druhý úryvek na řádku, již byla proměnná vytvořena, takže ji druhý úryvek mohl použít.

***Výpis 1.3:*** *Více úryvků na jednom řádku*

```
1  jshell> $1+$3  $1+$4
2  |  Error:
3  |  ';' expected
4  |  $1+$3  $1+$4
5  |       ^
6  |  Error:
7  |  not a statement
8  |  $1+$3  $1+$4
9  |         ^---^
10 |  Error:
11 |  cannot find symbol
12 |    symbol:   variable $4
13 |  $1+$3  $1+$4
14 |            ^^
15 |  Error:
16 |  unreachable statement
17 |  $1+$3  $1+$4
18 |         ^---^
19
20 jshell> $1+$3; $1+$4
21 $4 ==> 44
22 |  created scratch variable $4 : int
23 $5 ==> 55
24 |  created scratch variable $5 : int
25
26 jshell>
```

# 1.6 Příkazy (commands)

Jednou za čas potřebujeme po prostředí i něco jiného než vyhodnocení zadaného úryvku. Potřebujete si připomenout, co už jsme zadali, uložit to do souboru nebo naopak ze souboru nějakou zapamatovanou skupinu úryvků načíst. Pak využijeme příkazy.

Žádný výraz ani příkaz programu v jazyce *Java* nemůže začínat lomítkem. Všechny příkazy prostředí *JShell* proto lomítkem začínají . Prostředí tak snadno pozná, jestli se chystáme zadat další úryvek nebo příkaz. V této podkapitole se seznámíme pouze s několika základními příkazy, které se nám budou hodit v dalším výkladu.

## Vyloučení úryvku: /drop

Občas se dostaneme do situace, kdy by bylo nejlepší nějaký úryvek odstranit, přesněji **vyloučit ze seznamu aktivních** (úplně odstranit nejde, *JShell* si pamatuje i ty vyloučené). K tomu slouží příkaz `/drop`, jemuž v parametru předáme identifikační číslo úryvku. Vytváří-li daný úryvek objekt s nějakým názvem, název, můžeme místo identifikačního čísla úryvku zadat název vytvořeného objektu.

## Přehled aktivních úryvků: /list

Zadáním příkazu `/list` požádáte prostředí o vypsání všech aktivních úryvků. Úryvky, při jejichž zadávání jste udělali chybu, ani úryvky, které jste vyloučili nebo nahradili novější verzí, se nevypisují.

***Výpis 1.4:*** *Výpisy úryvků příkazem `/list`*

```
jshell> /list

   1 : 6+5
   2 : 6 +
       7+
       9
   3 : $1+$2
   4 : $1+$3;
   5 :  $1+$4

jshell> /drop 2
|  dropped variable $2

jshell> /list

   1 : 6+5
   3 : $1+$2
   4 : $1+$3;
   5 :  $1+$4

jshell>
```

Při výpisu úryvků se dodržuje formátování, v jakém jste úryvky zadali. Když jsem proto druhý úryvek zadal rozprostřený do tří řádků s různým oddělením čísel a operátoru `+` mezerami, tak se tak také vypíše – viz výpis 1.4, řádky 4–6. Obdobně vidíte u úryvku `5` zobrazeného na řádku 9, že do něj *JShell* zahrnul všechny znaky následující za středníkem ukončujícím předchozí úryvek včetně úvodní mezery (zadání viz řádek 24 ve výpisu 1.3).

Když jsem však druhý úryvek vyloučil z aktivních (řádky 11 a 12), tak ho příkaz `/list` již nevypsal (řádky 16–19).

## Přehled všech úryvků: `/list -all`

O vypsání všech úryvků včetně těch chybných a těch vyloučených požádáte zadáním příkazu `/list -all` (stačí `/list -a`). Ten vypíše všechny zadané úryvky včetně úryvků startovního skriptu (jejich `ID` začíná písmenem `s`) spouštěného při spuštění programu *JShell*.

Za ním následuje seznam všech zadaných úryvků (viz výpis 1.5) včetně těch neaktivních, tj. těch, které jste smazali (náš úryvek `2` zobrazený na řádcích 14–16), při jejichž zadávání jste udělali chybu (úryvek `e1` na řádku 18), nebo které jste nahradili novější verzí (takový tam zatím není).

***Výpis 1.5**: Výpis všech úryvků příkazem `/list -all`*

```
1  jshell> /list -all
2
3    s1 : import java.io.*;
4    s2 : import java.math.*;
5    s3 : import java.net.*;
6    s4 : import java.nio.file.*;
7    s5 : import java.util.*;
8    s6 : import java.util.concurrent.*;
9    s7 : import java.util.function.*;
10   s8 : import java.util.prefs.*;
11   s9 : import java.util.regex.*;
12  s10 : import java.util.stream.*;
13    1 : 6+5
14    2 : 6 +
15        7+
16        9
17    3 : $1+$2
18   e1 : $1+$3  $1+$4
19    4 : $1+$3;
20    5 :  $1+$4
21
22 jshell>
```

Jak jste jistě odhadli, chybně zadané úryvky poznáte ve výpisu podle toho, že jejich `ID` začíná písmenem `e` (první písmeno slova *error* = chyba). Já jsem prozatím zadal jediný

chybný úryvek, když jsem zadával dva výrazy na jednom řádku a neukončil první středníkem. Tento úryvek dostal `ID e1` a ve výpisu 1.5 je na řádku 18.

Bohužel, úryvky vyloučené z aktivních (např. úryvek s `ID=2`) nejsou ve výpisu nijak označeny, takže je musíte odhalit porovnáním se seznamem aktivních úryvků.

## Přehled objektů daného druhu

Někdy nás nezajímají ani tak vlastní úryvky, ale spíše objekty, které jsme zadáním úryvku vytvořili, resp. stav, který je právě nastaven.

Už jsme se setkali s tím, že jsme vytvořili proměnnou. V dalším textu se naučíte vytvářet (= definovat) i metody a datové typy a zadávat různé importy. Pro každý z těchto druhů objektů existuje příkaz pro zobrazení objektů daného typu.

- Příkaz `/vars`, resp. `/vars -all` zobrazí aktivní, resp. všechny proměnné. S proměnnými se seznámíte v kapitole *3 Proměnné* na straně 72.
- Příkaz `/methods`, resp. `/methods -all` zobrazí aktivní, resp. všechny definované metody. Definovat metody se naučíte v kapitole *5 Definice metod* na straně 104.
- Příkaz `/types`, resp. `/types -all` zobrazí aktivní, resp. všechny datové typy definované uživatelem. Definovat vlastní datové typy se začnete učit v kapitole *11 Třídy a jejich členy* na straně 221.
- Příkaz `/imports` zobrazí zadané importy. O importování různých částí programu si povíme v kapitole *13 Balíčky a knihovny* na straně 262.

Definice metod, datových typů ani importů jsme doposud neprobírali, takže si na použití odpovídajících příkazů musíte počkat[7]. Můžete si ale vyzkoušet výpis všech aktivních proměnných. Výsledek zobrazuje výpis 1.6. Všimněte si, že zadáte-li argument `-all`, bude příkaz `/vars -all` vypisovat i ty neaktivní.

***Výpis 1.6:*** *Výpis aktivních proměnných a všech proměnných*

```
jshell> /vars
|    int $1 = 11
|    int $3 = 33
|    int $4 = 44
|    int $5 = 55

jshell> /vars -all
|    int $1 = 11
|    int $2 = (not-active)
|    int $3 = 33
|    int $4 = 44
|    int $5 = 55

jshell>
```

[7] Myslel jsem tím smysluplné použití. Použít je můžete hned, ale *JShell* vám v odpovědi zobrazí pouze prázdný seznam.

### Uložení aktivních úryvků: /save <file>

I při práci s prostředím *JShell* potřebujeme občas odběhnout a práci na tuto dobu uložit. K uložení práce slouží příkaz `/save`, který je schopen uložit všechny aktivní úryvky. Jako parametr příkazu se zadává cesta k cílovému souboru, přičemž relativní cesta se odvozuje od složky, v níž jste program spustili. Chcete-li soubor uložit do aktuálního adresáře, stačí napsat pouze jeho název.

Uložené soubory jsou vnímány jako skripty a používá se pro ně přípona `jsh` jako zkratka z názvu *JShell*. Je to sice jenom konvence, ale doporučuji vám ji používat.

Zadávat můžete jak absolutní, tak relativní cestu. Zadáte-li pouze název souboru, uloží se do složky, z níž jste program *JShell* spustili. Zadáte-li název souboru i s celou cestou, uloží se tam, kam jste si objednali.

### Uložení všech zadaných úryvků: /save -all <file>

Zadáním příkazu `/save -all` uložíte všechny zadané úryvky včetně těch neaktivních (chybových nebo přepsaných). To se může hodit např. tehdy, chcete-li se o svých chybách s někým poradit.

### Uložení dosavadního průběhu seance: /save -history <file>

Zadáním příkazu `/save -history` uložíte vše, co jste v průběhu seance zadali; nejenom úryvky, ale i příkazy. Zadáte-li příkaz `/reset` nebo `/reload` (budou vysvětleny za chvíli), uloží se i ten. Načtením skriptu uloženého tímto příkazem můžete zopakovat kompletní historii seance od spuštění programu *JShell* až do chvíle uložení historie. Tímto příkazem proto budu ukládat průběh jednotlivých kapitol.

### Načtení skriptu: /open <file>

Uložený skript můžete znovu načíst. K tomu slouží příkaz `/open`, jehož jediným parametrem je cesta k načítanému souboru. Opět můžete zadávat absolutní i relativní cestu k souboru. Nejvýhodnější proto je mít načítané soubory uloženy tamtéž, co dávkový soubor, s jehož pomocí jste *JShell* spustili.

V některých kapitolách vám v poznámce o použitém projektu občas doporučím, abyste načetli nějaký soubor, v němž je připraven užitečný kód, který v dané kapitole použiju. Budete-li chtít zkoušet vše přesně tak, jak to ve výpisech zadávám, budete tento kód potřebovat.

### Ukončení seance: /exit

Příkaz ukončí běh programu *JShell*. Nicméně *JShell* si mezi seancemi pamatuje dříve zadané příkazy, takže při následující seanci můžete pomocí šipek aktivovat příkazy z minulé seance, aniž byste je museli znovu celé vypisovat.

### Restart: /reset

Občas se dostaneme do situace, v níž bychom nejraději vše zapomněli a začali zcela znovu. Po příkazu `/reset` *JShell* všechny zapamatované úryvky smaže, restartuje virtuální stroj a znovu načte startovní skript.

### Znovuzavedení: /reload -restore

Příkaz `/reload -restore` použijete ve chvíli, kdy potřebujete počítač vypnout a po čase se k rozdělané práci vrátit. Zadáte-li po opětovném spuštění programu jako první příkaz `/reload -restore`, načtou se znovu všechny úryvky, které byly při ukončení předchozí seance aktivní.

### Natavení startovního skriptu: /set -start <file>

Po spuštění programu *JShell* a po každém resetu se nejprve načte startovní skript. Jeho úryvky se ale příkazem `/list` nezobrazí. Zobrazit byste je mohli pouze příkazem `/list -all` nebo `/list -start`. V tomto skriptu si můžete připravit sadu definic úryvků, které pak budete v průběhu seance používat.

Startovní skript můžete kdykoliv změnit. Nově nastavený se začne načítat od příštího příkazu `/reset` nebo `/reload` a bude platit až do konce seance. Startovní skript nastavíte zadáním příkazu

```
/set -start <file>
```

kde `<file>` zastupuje soubor či skupinu souborů, které se při startu načtou.

V některých kapitolách vám bude na počátku doporučeno, abyste nastavili zadaný startovní skript, resp. startovní skripty. Jsou v nich připraveny úryvky, které budu v průběhu dané kapitoly využívat.

### Nápověda: /?

Nápovědu můžeme vyvolat dvěma příkazy: výše uvedeným příkazem `/?`, anebo příkazem `/help`. Za příkaz `/help` můžeme dát parametr s názvem objektu, o kterém se chceme dozvědět něco podrobněji.

Nápovědu můžete získat i tak, že rozepíšete úryvek nebo příkaz a stisknete klávesu `<TAB>`. Prostředí se „zamyslí“, jestli vám může radit. Pokud ano, tak doplní zadávaný příkaz a/nebo vypíše pod něj seznam možných pokračování.

Napovídací schopnosti prostředí umožní nezadávat příkazy v plném znění. Stačí napsat dostatečný počet znaků, aby prostředí zadávaný příkaz poznalo, a zadat jej.

# 1.7 Základní syntaktická pravidla

Na závěr představování programu *JShell* bych vás rád seznámil se dvěma základními součástmi kódu, které pomohou zpřehlednit skripty se záznamem průběhu jednotlivých kapitol.

Jak jistě víte, kód je správně zapsaná posloupnost znaků, přičemž ono „správně" znamená, že je zapsaná podle syntaktických pravidel daného jazyka. V této pasáži vám představím ta hlavní pravidla, v dalším textu se pak budeme postupně seznamovat s dalšími.

## Bílé znaky

Kód programu je tvořen posloupností symbolů, mezi něž patří identifikátory (= názvy), operátory (např. `+ - =`) a oddělovače (např. `, ; :` – čárka, středník, dvojtečka).

Mezi tyto symboly je možno vkládat libovolný počet bílých znaků, přičemž definice jazyka specifikuje bílý znak jako jeden ze znaků:

- mezera (`'\u0020'`),
- vodorovný tabulátor (`'\u000C'`),
- konec stránky – zde jsou tři možnosti:
  - znak LF = `'\u000A'`,
  - znak CR = `'\u000D'`,
  - dvojice znaků CRLF = `"\u000D\u000A"`).

Libovolný počet je i nula. Pokud by se však v případě, kdy mezi dva symboly nevložíte žádný bílý znak, tyto dva symboly slily a definovaly tak symbol jiný, je třeba mezi ně bílý znak vložit.

Když např. napíšete `a+b`, nemusíte znak `+` od okolních identifikátorů oddělovat, protože tento znak nemůže být součástí identifikátoru, takže je zcela zřejmé, kde jeden symbol končí a druhý začíná.

Když ale budete potřebovat napsat `int i` (za chvíli to použijeme), tak mezi tyto dva identifikátory bílý znak vložit musíte, protože `inti` by překladač považoval za identifikátor jediný.

## Komentáře

Komentář je část programu, kterou překladač ignoruje a která slouží pouze k tomu, aby čtenář získal o programu nějaké informace, které by mu při čtení kódu nemusely dojít. Komentář můžeme v programu napsat všude tam, kam můžeme napsat mezeru. Překladači je pak jedno, jestli na dané místo napíšeme komentář nebo mezeru – můžeme se na jeho práci dívat tak, že před vlastním překladem nahradí všechny komentáře mezerami.

*Java* definuje dva druhy komentářů: blokové, které mohou zabírat více řádků, a řádkové, které jsou na jediném řádku.

**Řádkový komentář** začíná dvojicí lomítek (//) a končí s koncem řádku. Potřebujeme-li pokračovat s informací i na dalším řádku, musíme před pokračovací text znovu vložit dvojici lomítek.

**Blokový komentář** začíná „*otevírací komentářovou závorkou*“ tvořenou znaky /* (lomítko následované hvězdičkou) a končí „*zavírací komentářovou závorkou*“ tvořenou znaky */ (hvězdička následovaná lomítkem). Uvnitř komentáře mohou být libovolné znaky (včetně přechodu na nový řádek) s výjimkou posloupnosti znaků tvořících zavírací komentářovou závorku. Z toho vyplývá, že blokové komentáře nemůžeme vnořovat. Vložíme-li do blokového komentáře posloupnost /*, bude to mít stejný vliv, jako kdybychom tam vložili cokoliv jiného s výjimkou zavírací komentářové závorky.

Speciálním případem blokového komentáře je **dokumentační komentář**, což je blokový komentář začínající trojicí znaků /**. Dokumentační komentáře slouží (jak název napovídá) k dokumentaci označeného kódu. Podrobněji vás s nimi seznámím, až bude jejich použití smysluplné, konkrétně v podkapitole *14.1 Dokumentační komentáře a API* na straně 283.

Komentáře můžete začlenit jako součást úryvku. Hodí se to např. tehdy, když si budete chtít seanci uložit a uložený skript později prohlížet. Komentář vám může připomenout, co jste zadáním daného úryvku sledovali či předat jinou informaci, kterou byste mohli zapomenout.

Ukázky řádkového i blokového komentáře si můžete prohlédnout ve výpisu 1.7. Dokumentační komentář jsem nepředváděl – později se mu budeme věnovat podrobněji. Na řádku 7 je předvedeno, jak lze komentář použít jako součást úryvku.

***Výpis 1.7:*** *Ukázky použití komentářů*

```
1  jshell> //Řádkový komentář
2
3  jshell> /* Několikařádkový blokový komentář
4     ...>    /* Vložení nové otevírací závorky neodstartuje vnořený komentář
5     ...>    vše se ukončí zapsáním zavírací závorky. */
6
7  jshell> $1 + $5      //Komentář jako součást úryvku
8  $6 ==> 66
9  |  created scratch variable $6 : int
10
11 jshell> /list
12
13    1 : 6+5
14    3 : $1+$2
15    4 : $1+$3;
16    5 :  $1+$4
17    6 : $1 + $5     //Komentář jako součást úryvku
18
19 jshell>
```

Ve výpisu si všimněte, že příkaz `/list` nevypisuje řádky, které obsahují pouze komentáře. Když je ale komentář součástí nějakého úryvku, příkaz `/list` jej vypíše jako součást daného úryvku – viz řádek 17.

K úryvkům obsahujícím pouze komentáře se *JShell* chová obdobně jako k příkazům (ty také začínají lomítkem). Nezobrazuje je ani příkaz `/list -all`, ale pouze příkaz `/list -history`, který vypíše celou konverzaci včetně případných řádků obsahujících pouze komentáře.

Pro úsporu místa reakce systému neuvádím ve výpisu 1.7 reakce na příkazy `/list -all` a `/list -history`. Přepokládám, že si je zájemci vyzkouší sami.

# 1.8 Ovládání

Při editaci úryvků a příkazů používáme nejčastěji editační šipky: šipkami doprava a doleva se přesouváme po aktuálně zadávaném textu, šipkou nahoru a dolů procházíme seznam doposud zadaných úryvků a příkazů.

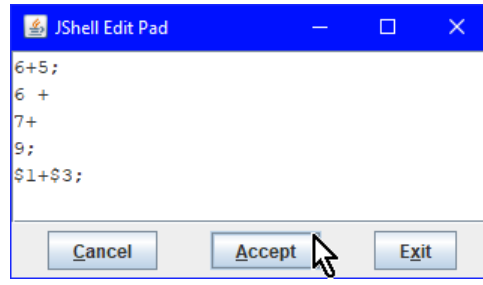
Prostředí *JShell* sice nabízí řadu dalších klávesových zkratek, ale domnívám se, že byste je stejně nepoužívali. Koho zajímají, ten je najde v již několikrát zmíněné příručce *Java 9 – JShell*.

## Použití editoru

Začnete-li používat *JShell* intenzivněji, budete chtít definovat složitější úryvky než ty prostoduché, které jsme definovali doposud. Pro takovéto případy nabízí *JShell* jednoduchý zabudovaný editor, v němž můžete vytvářet a upravovat složitější definice. Editor aktivujete zadáním příkazu

```
/edit <ID>
```

kde parametr `<ID>` může zastupovat jako identifikační číslo úryvku, tak název definovaného objektu.




***Obrázek 1.2:***
*Okno zabudovaného editoru otevřené po zadání příkazu `/edit 1 $2 4`*

Můžete dokonce uvést několik názvů či ID za sebou oddělených mezerami, přičemž můžete uvést jak aktivní úryvky, tak ty vyloučené. Editor pak otevře všechny jmenované. Na obrázku 1.2 je okno editoru otevřené po zadání příkazu

```
/edit 1 $2 4
```

Příkaz můžete zadat i bez parametru. Pak se v editoru otevřou všechny aktivní úryvky, ale to by byl v současné situaci pouze úryvek 1.

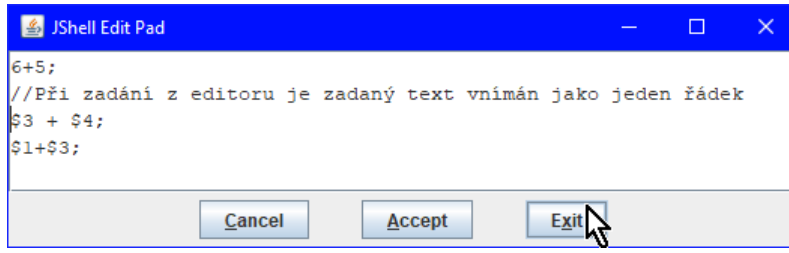
Okno se ovládá tak, že si v něm prohlédnete zobrazený kód a v případě potřeby jej upravíte. Význam tlačítek je následující:

- Tlačítko **Cancel** použijete tehdy, když si svoji úpravu rozmyslíte nebo když jste si chtěli definice jen prohlédnout. Okno editoru se zavře a vy můžete pokračovat v práci v konzolovém okně, aniž by se zaměnil původní stav.
- Tlačítko **Exit** stisknete tehdy, budete-li chtít své úpravy potvrdit. I po jeho stisku se okno editoru zavře, ale před zavřením se zadaný text předá k vyhodnocení.

  Je ale dobré vědět, že pokud v editačním okně nic nezměníte (anebo něco sice změníte, ale pak vrátíte text do původního stavu), nic se po stisku tohoto tlačítka nezadá.
- Tlačítko **Accept**, které na obrázku 1.2 stiskává myš, použijete ve chvíli, kdy si nejste jisti tím, že vaše zadání je bezchybné a chcete ho nejprve prověřit. Vaše zadání se pak předá k vyhodnocení, vy si v konzolovém okně můžete prohlédnout výsledek a pokračovat v editaci.

  I zde platí, že shoduje-li se výsledný text s výchozím, tak se platformě nic nezadává.

Když jsem ale upravil podobu druhého (tj. vyloučeného) úryvku podle obrázku 1.3, *JShell* jej akceptoval a uložil upravenou verzi jako úryvek s ID=7 – viz výpis 1.8.




***Obrázek 1.3:***
*Okno zabudovaného editoru otevřené po zadání příkazu* `/edit 1 $2 4`

Ve výpisu 1.8 bych vás chtěl upozornit na několik drobností. Jak naznačuje komentář v okně editoru, *JShell* akceptuje text obdržený z editoru obdobně, jako bychom ho celý zadali v jediném řádku. Z toho vyplývají následující vlastnosti:

- *JShell* si pamatuje podobu zobrazovaného textu a zpracovává pouze tu část, která se změnila – v našem případě pouze prostřední dva řádky definující nový úryvek.

- Středník si můžete odpustit pouze na posledním řádku. Pokud by nebyl některý z úryvků zadaných na předchozích řádcích ukončen středníkem, *JShell* by ohlásil chybu (nebo celou sérii chyb). Stejně jako tomu bylo ve výpisu 1.3 na straně 39.
- Chcete-li, aby se komentář stal součástí úryvku, musíte jej zadat před daným úryvkem. Napíšete-li jej za úryvek (přesněji za ukončující středník), bude jej *JShell* zpracovávat jako součást následujícího úryvku.

***Výpis 1.8:*** *Příkaz editace a výpis úryvků po editaci v okně na obrázku 1.3*

```
jshell> /edit 1 $2 4
$7 ==> 77
|  created scratch variable $7 : int

jshell> /list

   1 : 6+5
   3 : $1+$2
   4 : $1+$3;
   5 :  $1+$4
   6 : $1 + $5     //Komentář jako součást úryvku
   7 : //Při zadání z editoru je zadaný text vnímán jako jeden řádek
       $3 + $4;

jshell>
```

Vyzkoušejte si, jaká bude reakce prostředí, pokud byste v editoru smazali středník ukončující první úryvek a jaká bude reakce prostředí, přidáte-li vysvětlující komentář na konec druhého úryvku za jeho ukončující středník.

## Nastavení vlastního editoru

Připadá-li vám zabudovaný editor příliš jednoduchý a prostý, můžete použít svůj vlastní. Ten zadáte použitím příkazu

```
/set editor -retain -wait <command>
```

V tomto příkazu parametr `-retain` zadává, že si vaše nastavení bude *JShell* pamatovat a při příštím spuštění bude tento editor již přednastaven. Nechcete-li zadávat editor jako trvalý, ale budete-li jej zadávat pouze pro danou seanci, můžete parametr `-retain` vynechat.

Zadáním parametru `-wait` upravíte reakci prostředí *JShell* tak, že nebude čekat na zavření zavolaného editoru, ale na to, až opět aktivujete okno, v němž běží *JShell* (pravděpodobně konzolové okno). Poté stisknete dvakrát klávesu ENTER. Pomocný soubor určený k editaci přitom může zůstat v osloveném editoru nadále otevřený.

Parametr `<command>` reprezentuje příkaz operačního systému, kterým se daný editor spouští. *JShell* za tento příkaz doplní název souboru s editovaným textem.

Kdybyste měli s nastavením svého oblíbeného editoru nějaké problémy, zkuste si sehnat knihu *Java 9 – JShell*, v níž je tato problematika poměrně podrobně rozebrána.

## 1.9 Doprovodné programy

V knize je probíraná látka demonstrována z velké části na úryvcích programů spouštěných v prostředí *JShell*. V doprovodných programech je několik typů souborů:

- Soubory s příponou `.jsh`, u nichž za počátečním číslem kapitoly následuje znak `C`, obsahují zadávané úryvky doplněné o komentáře označující číslo výpisu, v němž je daná sada úryvků zobrazena i s odpověďmi systému.
- Soubory s příponou `.jsh` začínající slovem `Start` obsahují startovní úryvky spouštěné na počátku seance s úryvky kapitoly. Číslo za slovem `Start` označuje první z kapitol, v nichž se daný soubor používá.
- Soubory s příponou `.rec`, u nichž za počátečním číslem kapitoly následuje znak `R`, obsahují záznam seance k dané kapitole a vedle zadávaných úryvků v nich najdete i odpovědi systému.
- Soubory s příponou `.wrd`, u nichž za počátečním číslem kapitoly následuje znak `W`, obsahují výpisy programů v knize včetně čísel řádků.

## 1.10 Shrnutí

Skript uchovávající naše akce z této kapitoly spolu s výše uvedenými informačními komentáři je uložen v souboru **01C_JShell_Intro.jsh**.

# Kapitola 2
# Základní datové typy a jejich literály

**Co se v kapitole naučíte**
Tato kapitola vás seznámí se základními datovými typy, konkrétně s primitivními datovými typy a s typy `Object` a `String`. Současně vás naučí, jak v programu zadávat konstanty těchto typů – tzv. literály.

V této kapitole odstartujeme novou seanci. Nicméně pokud máte program *JShell* spuštěný, nemusíte jej ukončovat – stačí, když zadáte příkaz `/reset`.

## 2.1 Datové typy

Než se rozhovoříme o tom, jak pracovat se zprávami, které vracejí hodnoty, musíme si nejprve povědět něco o datových typech. Datový typ (nebo zkráceně jen typ) je označení pro trojici charakteristik specifikujících vlastnosti hodnot, které budeme označovat za data daného typu. Svůj typ mají veškerá data, se kterými program pracuje. Datový typ specifikuje:

- množinu přípustných hodnot, resp. stavů,
- způsob uložení těchto hodnot (stavů) v paměti (o ten se zatím nebudeme zajímat a zpracování této informace přenecháme virtuálnímu stroji),
- operace, které lze s instancemi daného typu provádět.

Jinými slovy: datový typ prozrazuje, co můžeme od hodnot daného typu očekávat a co s nimi můžeme dělat. Tím se na jednu stranu zvyšuje efektivita práce programu, ale především se tím snižuje počet chyb. K této otázce se v budoucnu ještě několikrát vrátím.

*Java* (a spolu s ní řada dalších moderních jazyků) trvá na tom, aby byl u každého údaje předem znám jeho typ. Za to se nám odmění tím, že bude pracovat rychleji (nemusí v průběhu výpočtu bádat nad tím, co je které „dato“ zač), a navíc odhalí řadu našich chyb již v zárodku.

## Dělení datových typů

Jak jistě víte, *Java* se řadí mezi jazyky podporující objektově orientované programování – OOP. V čistém OOP jsou všechny datové typy objektové. Tvůrci *Javy* ale chtěli, aby výsledné programy pracovaly co nejrychleji i na jednoduchých procesorech zabudovaných v nejrůznějších zařízeních, a proto rozdělili datové typy do tří skupin:

- Speciální jednoprvkovou skupinu představuje degenerovaný typ-netyp `void`, který se používá pouze k tomu, aby programátor mohl veřejně vyhlásit, že nějaká metoda nic nevrací (co to znamená, si vysvětlíme v kapitole *5 Definice metod* na straně 104). Nikde jinde se použít nedá.
- Osm datových typů, které mají přímou podporu v instrukčních souborech většiny mikroprocesorů, označili jako *primitivní datové typy*. (Hodnoty těchto typů budeme občas označovat jako *primitivní hodnoty*.) Práci s nimi převádí virtuální stroj přímo na příslušné instrukce daného procesoru. Hodnoty primitivních typů se často označují souhrnným názvem *primitiva*.

  Práce s hodnotami primitivních datových typů je sice mnohem jednodušší a rychlejší než práce s objekty, ale tato jednoduchost je vykoupena nemožností dále je rozšiřovat a upravovat jejich vlastnosti a schopnosti.
- Všechny ostatní datové typy zařadili mezi *objektové datové typy*. S nimi se pracuje jinak. Objektových typů bývá v programu definováno většinou mnohem víc. Jenom ve standardní knihovně *Javy 14* je definováno 5938 veřejně použitelných datových typů a několikanásobně více těch soukromých určených pro interní použití v rámci knihovny.

  Objektové datové typy si může každý programátor definovat sám. S jistou rezervou bychom mohli říci, že objektové programování spočívá v návrhu a implementaci těch správných objektových datových typů.

Knihovnou označujeme sadu datových typů, které tvoří nějaký ucelený soubor a které můžeme ve svém programu používat. Standardní knihovna je součástí instalace *Javy*. Ostatní je třeba nějak získat a začlenit do programu.

Objektových datových typů je několik druhů. Nejpoužívanější z nich jsou **třídy**. Setkáte-li se v dalším výkladu s termínem třída, takž vězte, že to je jeden z možných druhů objektových datových typů.

## Primitivní datové typy

Primitivní datové typy bychom mohli rozdělit do dvou skupin: na ty poměrně často využívané a na ty, s nimiž se setkáte spíše výjimečně. Mezi ty často používané patří datové typy:

`boolean` Definuje typ logických hodnot, které mohou nabývat pouze hodnot `true` (pravda, ano, …) a `false` (nepravda, ne, …). Název tohoto typu nám připomíná matematika George Boola, který se v 19. století zabýval logikou. Na jeho počest se práce s logickými výrazy označuje jako Booleova algebra.

`char` Znaky (zkratka z anglického *character* = znak) respektující kódování podle normy *Unicode*. Vedle číslic a písmen všech abeced včetně čínských, japonských a korejských znaků, egyptských hieroglyfů apod. sem patří i další znaky jako noty, kartografické značky, emotikony a další.

`double` Označuje datový typ pro reprezentaci reálných čísel. Čísla typu `double` se v Javě ukládají s přesností na 15 platných číslic a v rozsahu přibližně $10^{\pm 308}$ (číslo s 308 nulami před, resp. za desetinnou čárkou). Své jméno dostal od toho, že v době svého zavedení (šedesátá léta minulého století) definoval čísla s dvojitou přesností oproti číslům tehdy většinou používaným.

`int` Označuje typ celých čísel, jejichž hodnoty se mohou pohybovat v rozsahu ±2 miliardy (přesně od `-2 147 483 648` do `+2 147 483 647`, tj. od $-2^{31}$ do $+2^{31}-1$). Název typu je zkratkou ze slova integer (= celé číslo). Je to nejpoužívanější datový typ.

`long` Velká celá čísla v rozsahu přibližně $\pm 9 \times 10^{18}$ (chcete-li to přesně, tak je to od `-9 223 372 036 854 775 808` do `+9 223 372 036 854 775 807`, tj. od $-2^{63}$ do $+2^{63}-1$.

Zbylé primitivní datové typy se příliš nepoužívají a uvádím je pouze pro úplnost:

`byte` Celá čísla od `-128` do `+127`, tj. od $-2^7$ do $+2^7-1$.

`short` Celá čísla v rozsahu od `-32 768` do `+32 767`, tj. od $-2^{15}$ do $+2^{15}-1$.

`float` Reálná čísla v rozsahu asi $10^{\pm 38}$ uchovávaná s přesností přibližně 6 platných čísel.

Nepleťte si platné číslice a desetinná místa. Např. číslo `0,00123` má pět desetinných míst, ale pouze tři platné číslice. Na druhou stranu číslo `12300` nemá žádné desetinné místo a může mít tři až pět platných číslic podle toho, jsou-li závěrečné nuly přesné, anebo vznikly zaokrouhlením.

## Objektové datové typy

Jak jsem již řekl, objektových datových typů je nepřeberné množství[8] a sami můžete vytvářet další. Podrobněji se jim budu věnovat v druhé části učebnice. Prozatím vás seznámím jenom se třemi nejdůležitějšími, přičemž všechny patří mezi třídy:

`Object` Společný rodič všech datových typů. Mohli bychom jej (s jistou nadsázkou) považovat za programovou realizaci základní teze, že **všechno je objekt** (ještě se k ní vrátíme).

`String` Datový typ reprezentující textové řetězce, tj. posloupnosti znaků. Kdykoliv budeme chtít pracovat s nějakým textem (budeme jej chtít někam vložit, zapsat, poslat, …), budeme pracovat s objektem typu `String`.

Hodnoty typu `String` se sice oficiálně označují jako textové řetězce, ale v praxi programátoři tento termín vůbec nepoužívají, ti mladší jej často ani neznají. Prakticky vždy označují tyto hodnoty jako stringy. Je to sice slangový výraz, ale je mezi programátory tak hluboce zakořeněný, že jsem se rozhodl jej ve svých knihách používat.

`Class` Datový typ, jehož instance (hodnoty) reprezentují jiné datové typy. Podrobněji se s ním seznámíte až v druhé části knihy.

*null-type* Speciální typ, který je potomkem všech ostatních datových typů (o tom, co je to předek a potomek, se rozpovídám v podkapitole *10.13 Dědění* na straně 214). Tento datový typ je oficiálně bezejmenný, ale v dokumentaci je označovaný jako *null type*. Jeho jedinou hodnotu je totiž konstanta `null` reprezentující tzv. *prázdný odkaz*.

V objektovém programování se objekty daného datového typu často označují jako ***instance*** daného datového typu. Vedle objektů, které jsou instancemi nějakého typu, totiž existují i objekty, které nejsou instancemi žádného typu. To vše si ale podrobně probereme až ve druhé části.

### Odkazy na objekty

V *Javě* jsou data všech objektů uložena ve speciální oblasti paměti nazývané *halda* (anglicky *heap*). Program nikdy nepracuje přímo s těmito daty, ale vždy pouze s odkazem na ně. Hovoříme-li proto o tom, že v proměnné je uložena hodnota objektového typu, znamená to, že v proměnné je uložen odkaz na místo na haldě, kde jsou uložena data reprezentující daný objekt.

[8] Jen ve standardní knihovně je 5938 veřejně dostupných objektových datových typů a nepočítaně typů interních.

K tomuto tématu se ještě vrátím v druhé části, až budeme probírat konstrukce podporující objektově orientované programování.

## 2.2 Literály

Literály jsou vlastně konstanty nazvané svojí hodnotou. Když někam napíšete např. číslo `3.14` nebo string `"Dobrý den"`, použili jste literál. Pojďme se nyní podívat, jak se správně zapisují literály jednotlivých datových typů.

### Literály typu `boolean`

Potřebujete-li někam uložit hodnotu informující o tom, zda něco je či není pravda, zadáváte hodnotu `true`, resp. `false` – viz výpis 2.1.

*Výpis 2.1:* *Literály typu `boolean`*

```
jshell> false   //NE, neplatí, nepravda, ...
$1 ==> false
|  created scratch variable $1 : boolean

jshell> true    //ANO, platí, pravda, ...
$2 ==> true
|  created scratch variable $2 : boolean

jshell>
```

### Literály typu `int`

U celých čísel je to složitější, mimo jiné proto, že jejich hodnoty je možno zapisovat ve čtyřech číselných soustavách: ve dvojkové, osmičkové, desítkové nebo šestnáctkové soustavě. Zápisy v jednotlivých soustavách se liší předponou (prefixem) a použitelnými číslicemi. V následujícím přehledu je vždy uveden základ číselné soustavy následovaný pravidly pro zápis v této soustavě.

2 Číslo zapisované ve dvojkové soustavě má jako předponu dvojici znaků `0b` nebo `0B`. Povolené jsou pouze číslice `0` a `1`.

8 Číslo zapisované v osmičkové soustavě musí začínat číslicí `0` (to je jeho prefix). Povolené jsou pouze číslice `0` až `7`.

10 Číslo zapisované v desítkové soustavě nemá žádný prefix a NESMÍ začínat *číslicí 0 (pak by bylo považováno za zapsané v osmičkové soustavě). Uvnitř čísla* jsou povolené číslice `0` až `9`.

16 Číslo zapisované v šestnáctkové (hexadecimální) soustavě má jako prefix dvojicí znaků `0x` nebo `0X`. Povolené jsou číslice `0` až `9` a znaky `A` až `F`, resp. `a` až `f` reprezentující číslice s hodnotami `10` až `15`.

## Historická vsuvka – číselné soustavy

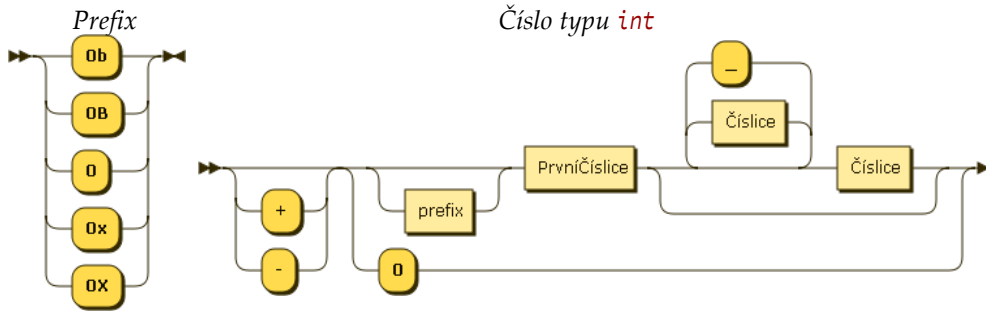
Osmičková (můžete se také setkat s termínem *oktalová*) číselná soustava se v současné době již příliš nepoužívá. Její podpora je převzata z jazyka C, na nějž *Java* nepřímo navazuje. Jazyk C byl vyvinut v sedmdesátých letech pro operační systém UNIX, jenž byl původně vyvinut pro počítače PDP. Na nich se často používala osmičková soustava. Ostatně délka slova těchto počítačů byla zpočátku vždy dělitelná třemi – jejich slova měla 12, 18, 24 nebo 36 bitů, takže se jejich obsah dal výhodně zapsat několika osmičkovými číslicemi (osmičková číslice je ekvivalentem tří dvojkových číslic – tří bitů).

Délka slov, která je mocninou čísla 2 (4, 8, 16, 32, 64 či 128 bitů) se ustálila později. Spolu s ní se prosadilo i používání šestnáctkové soustavy, protože jedna šestnáctková číslice je ekvivalentem čtyř číslic dvojkových, takže obsah slova je výhodně definován příslušným počtem šestnáctkových číslic.

Šestnáctková (můžete se také setkat s termínem hexadecimální[9]) číselná soustava používá k zápisu čísla šestnáct číslic představující hodnoty nula až patnáct. Pro šestnáctkové číslice větší než `9` se používají písmena z počátku abecedy: `A=10`, `B=11`, `C=12`, `D=13`, `E=14`, `F=15`. Přitom nezáleží na tom, používají-li se písmena malá či velká.

Aby se zpřehlednil zápis dlouhých čísel, je povoleno vkládat do prostoru mezi první a poslední číslicí znaky podtržení. Syntaktický diagram pravidel pro zápis čísla je vidět na obrázku 2.1. V levé části jsou vyjmenovány povolené prefixy a v pravé je pak vlastní definice zápisu čísla.

V desítkové soustavě nesmí být první číslicí nula, v ostatních soustavách může být první číslicí libovolná číslice platná v dané soustavě.




***Obrázek 2.1:***
*Syntaktický diagram zápisu čísla v různých číselných soustavách*

[9] Nepleťte si termín *hexadecimální*, tj. šestnáctková, s termínem *hexagesimální* (někdy se používá i termín *sexagesimální*), tj. šedesátková. Šedesátkovou číselnou soustavu používali staří Sumerové. Dnes se používá při měření času (minuty, sekundy) a při měření úhlů.

### Názvy skupin bitů

V souvislosti s ukládáním do paměti a bitovými operacemi bych připomenul základní termíny:

**Bit** Dvojková číslice 0/1

**Nibble** Čtveřice dvojkových číslic umožňující původně reprezentovat jednu desítkovou číslici. Později se termín ustálil jako označení pro paměťový prostor pro právě jednu šestnáctkovou číslici.

**Bajt** Anglicky byte, původně překládáno do češtiny jako slabika, ale nyní už výhradně jako bajt. Bajt měl na různých počítačích různý počet bitů, ale posléze se prosadila velikost 8 bitů, která je také standardizována v ISO/IEC 2382-1 a následně v IEC 80000-13.

**Slovo** Anglicky *word* představuje základní paměťovou jednotku počítače. Současné počítače používají většinou 32 nebo 64bitová slova.

Slovo bývá nejmenší jednotka, kterou umí procesor načíst. Má-li pracovat s menšími jednotkami, načte slovo a požadovanou menší jednotku (např. bajt) z něj extrahuje.

Ve výpisu 2.2 jsou příklady různých zadání čísel ve všech vyjmenovaných číselných soustavách.

Na řádku 1 je ve dvojkové soustavě zadáno `23` (`= 1×16 + 0×8 + 1×4 + 1×2 + 1×1`) Znak podtržení je vložen tak, aby se usnadnil převod do šestnáctkové soustavy.

Na řádku 5 je zapsáno totéž číslo v téže soustavě, ale tentokrát je znak podtržení vložen tak, aby se usnadnil převod do osmičkové soustavy

Na řádku 9 je toto číslo v osmičkové soustavě (`2×8 + 7×1`) a na řádku 13 v šestnáctkové soustavě (`1×16 + 7×1`).

Na řádku 17 ukazuji, že napíšete-li v čísle začínajícím nulou číslici větší než `7`, překladač se vzbouří a ohlásí syntaktickou chybu. Takové číslice totiž v osmičkové soustavě nejsou.

Na řádku 23 je další „oblíbená“ chyba: při pokusu o zápis čísla `0x17` je místo počáteční nuly zapsáno písmeno O. Posloupnost znaků `0x17` pak překladač logicky chápe jako identifikátor (název). Protože takto pojmenovaný objekt nezná, ohlásí chybu.

Toto je častá chyba, kterou začátečníci špatně odhalují, protože ten rozdíl mezi znaky nevidí. Proto byste měli při programování používat písma, která jasně odlišují nulu od O a jedničku od l (malé L) a I. Správně bylo číslo zapsáno na řádku 13.

## Literály typu `long`

Literály typu `long` se od literálů typu `int` liší pouze příponou (sufixem) `l` nebo `L`, přičemž se vřele nedoporučuje používat malé L, které se dá velmi snadno splést s jedničkou. U zadávání hodnot typu `long` se musí přípona uvádět vždy.

***Výpis 2.2:*** *Literály typu `int` v různých číselných soustavách*

```
1  jshell> 0b1_0111    //23 = 1*16 + 0*8 + 1*4 + 1*2 + 1*1 = 0x17
2  $3 ==> 23
3  |  created scratch variable $3 : int
4
5  jshell> 0B10_111    //027 - Číslo upravené pro převod na osmičkové
6  $4 ==> 23
7  |  created scratch variable $4 : int
8
9  jshell> 027         //Stejné číslo zapsané v osmičkové soustavě
10 $5 ==> 23
11 |  created scratch variable $5 : int
12
13 jshell> 0x17        //Stejné číslo zapsané v šestnáctkové soustavě
14 $6 ==> 23
15 |  created scratch variable $6 : int
16
17 jshell> 08          //Začíná nulou, ale obsahuje příliš velkou číslici
18 |  Error:
19 |  integer number too large: 08
20 |  08          //Začíná nulou, ale obsahuje příliš velkou číslici
21 |  ^
22
23 jshell> 0x17   //Začíná písmenem O => není to číslo, ale identifikátor
24 |  Error:
25 |  cannot find symbol
26 |    symbol:   variable 0x17
27 |  0x17        //Začíná písmenem O => není to číslo, ale identifikátor
28 |  ^--^
29
30 jshell>
```

Výpis 2.3 ukazuje, že u malých čísel v rozsahu čísel typu `int` ovlivníte příponou typ proměnné, kterou pro vaši hodnotu překladač vyhradí. Pro nulu zadanou na řádku 1 vyhradil proměnnou typu `long`, kdežto pro mnohem větší číslo na řádku 5 zadané bez přípony vyhradil pouze proměnnou typu `int`, protože číslo nemělo příponu `L`.

Na řádku 9 jsem zadal číslo přesahující rozsah čísel typu `int`. Protože jsem však nezadal příponu, překladač ohlásil syntaktickou chybu. Stačilo ale zadat příponu (viz řádek 15) a vše prošlo naprosto hladce.

## Literály typu `byte` a `short`

Datové typy `byte` a `short` nemají žádné své literály. Jak se s nimi pracuje, si proto ukážeme, až probereme operátor přetypování.

## Literály typu `double`

Reálná čísla jde zapsat v desítkové nebo šestnáctkové soustavě. Druhý způsob se ale používá pouze ve výjimečných situacích, takže jej tu nebudu rozebírat. I tak je to