EDICE
myslíme v ...

knihovna programátora

- Podrobný výklad vlastností jazyka od naprostých základů až po pokročilé, běžně neprobírané konstrukce
- **Vedle aktuálních konstrukcí vysvětluje i ty, které jsou zabudovány jen předběžně a stanou se řádnými až v některé z příštích verzí**
- Pro demonstraci vykládaných konstrukcí bez zbytečného pomocného kódu využívá zabudované REPL prostředí JShell
- Ukazuje, jak efektivně experimentovat a využitím prostředí JShell získat okamžité odpovědi
- Ideální jako učebnice i referenční příručka

RUDOLF
PECINOVSKÝ

Java 21

Kompletní příručka jazyka

GRADA

GRADA

EDICE
myslíme v ...

knihovna programátora

RUDOLF
PECINOVSKÝ

Java 21

Kompletní příručka jazyka

GRADA
Publishing

Rudolf Pecinovský

Java 21
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 8677. publikaci

Odpovědný redaktor: Petr Somogyi
Grafická úprava a sazba Rudolf Pecinovský
Počet stran 640
První vydání, Praha 2023
Vytiskly Tiskárny Havlíčkův Brod, a.s.

ISBN 978-80-271-7041-8 (pdf)
ISBN 978-80-247-0599-6 (print)

Všem, kteří se chtějí něco naučit

Stručný obsah

Stručný obsah 6
Podrobný obsah 8
Úvod 25

Část I Neobjektové konstrukce 35

1 Prostředí JShell 36
2 Základní datové typy a jejich literály 58
3 Proměnné 82
4 Základní operátory 96
5 Definice metod 115
6 Ostatní operátory 135
7 Pole 165
8 Rozhodování 181
9 Opakování části kódu 200

Část II Základní objektové konstrukce 221

10 Základy objektově orientovaného paradigmatu 222
11 Třídy a jejich členy 246
12 Vytvoření aplikace a vývojová prostředí 270
13 Balíčky a knihovny 291
14 Dokumentace API 310
15 Konstrukce interface 322
16 Podrobnosti o konstruktorech 340
17 Úvod do dědění implementace 357
18 Viditelnost členů tříd 380
19 Virtuální metody a jejich přebíjení 396
20 Abstraktní třídy 407

Část III Pokročilejší objektové konstrukce 419

21 Výjimky a aserce 420
22 Generické datové typy a metody 447
23 Typové parametry a argumenty 466
24 Interní datové typy 487
25 Výčtové typy – třídy typu enum 503
26 Záznamové třídy – třídy typu record 521
27 Další použití rozpoznávání vzorů 537
28 Lambda-výrazy 552
29 Anotace a šablonové procesory 569
30 Vlákna a paralelní procesy 584
31 Moduly 593
32 Kategorie datových typů, hodnotové typy 607

Část IV Přílohy 615

A Tvorba jednoduchého GUI 616
Literatura 630
Rejstřík 632

Podrobný obsah

Stručný obsah **6**
Podrobný obsah **8**
Úvod **25**
Komu je kniha určena **25**
Koncepce výkladu **25**
Rozdělení textu **26**
Terminologie **26**
Použité nástroje **27**
Vývojová sada JDK **27**
Vývojové prostředí *JShell* **27**
Samostatné vývojové prostředí **27**
Doprovodné programy **28**
Zlom 28
Předběžné definice nových konstrukcí **29**
Předběžná funkce/konstrukce/vlastnost (Preview Feature) 29
Globální nastavení podpory předběžných konstrukcí 29
Experimentální funkce/konstrukce/vlastnost (Experimental Feature) 30
Inkubační funkce/konstrukce/vlastnost (Incubating Feature, Incubator) 30
Syntaktické definice a diagramy **30**
Použité typografické konvence **30**
Odbočka – podšeděný blok **32**
Zpětná vazba **32**

Část I Neobjektové konstrukce **35**

1 Prostředí JShell **36**
1.1 Nejprve trocha terminologie **36**
Objektově orientované paradigma – OO paradigma, OOP 37
Objekt v programu 37
Třída, datový typ 37
Proměnná 38
Atributy 38
Metody 38
Vlastnosti 38
Interní typy 38
Členy 38
1.2 Charakteristika programu a prostředí JShell **38**
1.3 Problémy s klávesnicí **39**
1.4 Příprava programu JShell a první spuštění **39**
Dávkové soubory pro *Windows* **40**
Po spuštění **41**
1.5 Úryvky (snippets) **42**
Použití proměnných **43**

Identifikace úryvků ... 44
Středník ... 44
Více objektů na řádku, zavlečené chyby ... 44
1.6 Příkazy (commands) ... 45
Vyloučení úryvku: /drop ... 46
Přehled aktivních úryvků: /list ... 46
Přehled všech úryvků: /list -all ... 47
Přehled objektů daného druhu ... 47
Uložení aktivních úryvků: /save <file> ... 48
Uložení všech zadaných úryvků: /save -all <file> ... 49
Uložení dosavadního průběhu seance: /save -history <file> ... 49
Načtení skriptu: /open <file> ... 49
Ukončení seance: /exit ... 49
Restart: /reset ... 50
Znovuzavedení: /reload -restore ... 50
Nastavení startovního skriptu: /set -start <file> ... 50
Nastavení zpětnovazebního režimu: /set feedback ... 50
Aktivace dalšího zdroje: /env ... 51
Nápověda: /? ... 52
1.7 Základní syntaktická pravidla ... 52
Bílé znaky ... 52
Komentáře ... 53
1.8 Ovládání ... 54
Použití editoru ... 54
Nastavení vlastního editoru ... 57
1.9 Doprovodné programy ... 57
1.10 Soubory pro opakování ... 57
2 Základní datové typy a jejich literály ... 58
2.1 Datové typy ... 58
Dělení datových typů ... 59
Primitivní datové typy ... 60
Implicitní kódování znaků – UTF-8 ... 61
Objektové datové typy ... 61
Odkazy na objekty ... 62
2.2 Literály ... 62
Literály typu boolean ... 62
Literály typu int ... 63
Historická vsuvka – číselné soustavy ... 63
Názvy skupin bitů ... 64
Literály typu long ... 66
Literály typu byte a short ... 66
Literály typu double ... 66
Celé číslo s příponou ... 67
Obyčejné desetinné číslo ... 67
Číslo v exponentovém tvaru ... 67
Literály typu float ... 69
Literály typu char ... 69
Prázdný odkaz null ... 72
Literály typu String ... 73
Odbočka: volání metod ... 73
Příklady ... 74
Textové bloky ... 76
Literály typu Class ... 79
2.3 Ještě trocha terminologie ... 79
2.4 Nestandardní hodnoty reálných typů ... 79

2.5 Soubory pro opakování ... 81
3 Proměnné ... 82
3.1 Pravidla pro tvorbu identifikátorů ... 82
Používání znaku $ ... 84
Konvence pro velikost písmen ... 84
3.2 Druhy typování ... 85
Statické × dynamické typování ... 85
Definice × odvození datového typu ... 85
Silné (přísné) × slabé typování ... 86
Shrnutí ... 86
3.3 Definice × deklarace ... 86
3.4 Deklarace a definice proměnných ... 87
3.5 Středníky ... 89
3.6 Současná deklarace více proměnných ... 90
Reakce prostředí *JShell* ... 90
3.7 Redeklarace proměnných v JShell ... 90
3.8 Deklarace s přiřazením počáteční hodnoty ... 92
Pozor na velikost znaků ... 92
Zpět k deklaraci s přiřazením počáteční hodnoty ... 93
3.9 Syntaktický diagram ... 94
3.10 Definice proměnných s využitím `var` ... 94
3.11 Soubory pro opakování ... 95
4 Základní operátory ... 96
4.0 Inicializace prostředí JShell ... 97
4.1 Nejprve trocha teorie ... 97
4.2 Operátor přiřazení = ... 98
Přiřazení je výraz ... 98
4.3 Unární `+` a `-` ... 99
4.4 Aritmetické operátory `+ - * / %` ... 99
Operátor sčítání ... 99
Sčítání stringů ... 99
Operátor odčítání ... 100
Operátor násobení ... 101
Operátor dělení ... 101
Operátor zbytku po dělení ... 101
4.5 Kulaté závorky `()` ... 102
Alternativní řešení ... 103
4.6 Operátor přetypování (*typ*) ... 103
Implicitní přetypování ... 103
Příklady implicitního přetypování ... 104
Explicitní přetypování ... 106
Priorita ... 106
Kontrola ... 106
Explicitní přetypování hodnot primitivních typů ... 107
Příklady ... 107
Přetypování instancí objektových datových typů ... 109
Univerzální „přetypování“ na `String` ... 110
Textový podpis ... 111
4.7 Specifika číselných typů ... 111
Malé celočíselné typy ... 111
Ztráta přesnosti ... 113
Pořadí vyhodnocování ... 113
První příklad ... 114
Druhý příklad ... 114

4.8 Soubory pro opakování ... 114
5 Definice metod ... 115
5.1 Historické ohlédnutí ... 115
5.2 Definice a volání metody ... 116
5.3 Volání metody ... 118
5.4 Metody s parametry ... 119
Parametry versus argumenty ... 120
Více parametrů ... 121
Předávání hodnot parametrům ... 121
5.5 Metody vracející hodnotu ... 122
5.6 Přetěžování metod ... 122
5.7 Lokální proměnné metod ... 124
Postup volání metody ... 125
Parametry × lokální proměnné ... 125
Zásobník návratových adres – ZNA ... 126
Životnost lokálních proměnných ... 126
5.8 Příklady ... 127
Jídelna ... 128
Návratová hodnota ... 128
Definice metod v editoru ... 128
5.9 Metody s proměnným počtem argumentů ... 128
5.10 Přehled definovaných metod ... 129
5.11 Syntaktický diagram ... 130
5.12 Přímé spuštění souboru se sadou metod (preview) ... 131
Definice spustitelného souboru ... 131
Překlad a spuštění vytvořeného programu ... 132
Omezení oproti JShell ... 134
5.13 Soubory pro opakování ... 134
6 Ostatní operátory ... 135
6.1 Inkrementační a dekrementační operátory ++ -- ... 135
6.2 Porovnávací operátory < <= == != >= > ... 137
Testování shody reálných čísel ... 138
Zvláštnosti porovnávání stringů ... 139
p12 == false ... 140
p13 == true ... 140
p23 == false ... 141
Porovnávání objektů reprezentujících hodnotu ... 141
6.3 Logické operátory ! & && | || ... 141
6.4 Bitové operátory ~ & | ^ << >> >>> ... 143
6.5 Složené přiřazovací operátory Op= ... 147
Příklady využití přetypování ... 147
6.6 Ternární operátor :? – podmíněný výraz ... 148
Ještě jednou porovnávání reálných čísel ... 150
6.7 Přepínač – výraz switch ... 151
Pravidla ... 152
Příklad ... 152
6.8 Operátor instanceof ... 153
Rozšíření funkcionality operátoru instanceof ... 156
6.9 Zbylé operátory: new [] () 157
Operátor new ... 157
Operátor . (tečka) ... 158
Operátor [] ... 159
Operátor volání metody () ... 159

6.10 Priorita, asociativita a komutativita operátorů ... 159
Priorita ... 159
Asociativita ... 161
Komutativita ... 161
6.11 Šablonový výraz (předběžné – preview) ... 161
6.12 Soubory pro opakování ... 164
7 Pole ... 165
7.1 Strukturovaný datový typ – kontejner – pole ... 165
7.2 Deklarace a inicializace polí ... 167
Syntaxe zděděná od jazyků C/C++ ... 168
7.3 Přiřazení hodnoty poli a přetypování polí ... 169
7.4 Počet prvků pole ... 170
7.5 Práce s prvky pole ... 171
7.6 Vícerozměrná pole – pole polí ... 173
Obdélníková pole ... 174
Zubatá pole ... 174
Inicializace dvourozměrného pole ... 176
Inicializace vícerozměrného pole ... 177
7.7 Proměnný počet argumentů metod ... 177
7.8 Arrays – knihovna metod pro práci s poli ... 178
7.9 Emulace předání argumentu odkazem ... 178
7.10 Pole a moderní programování ... 179
7.11 Soubory pro opakování ... 180
8 Rozhodování ... 181
8.1 Jednoduchý podmíněný příkaz ... 181
8.2 Blok příkazů (složený příkaz) ... 182
Vnořování bloků ... 183
Proměnné lokální v bloku ... 184
8.3 Rozpoznávání vzorů operátorem instanceof ... 187
Lokální proměnná vytvořená v rámci vyhodnocování složitějšího výrazu ... 189
8.4 Úplný podmíněný příkaz ... 190
8.5 Složený podmíněný příkaz ... 191
8.6 Přepínač – příkaz switch ... 193
Pravidla ... 194
Příklad ... 196
Rozšíření funkcionality ... 199
8.7 Soubory pro opakování ... 199
9 Opakování části kódu ... 200
9.1 Obecný cyklus ... 200
9.2 Cyklus s ukončovací podmínkou – cyklus do…while ... 201
9.3 Cyklus s počáteční podmínkou – cyklus while ... 202
9.4 Cyklus s parametrem – cyklus for ... 204
Metody s proměnným počtem argumentů ... 206
9.5 „Dvojtečkový“ cyklus for (cyklus „for each“) ... 208
9.6 Vnořování cyklů ... 211
9.7 Cyklus s prázdným tělem ... 211
9.8 Nekonečný cyklus ... 212
9.9 Cyklus s podmínkou uprostřed ... 213
9.10 Příkaz break s návěštím ... 215
9.11 Příkaz continue ... 216
9.12 Rekurze ... 217
Princip ... 218

Přímá a nepřímá rekurze 219
Přeplnění zásobníku návratových adres 219
9.13 Soubory pro opakování 220

Část II Základní objektové konstrukce 221

10 Základy objektově orientovaného paradigmatu 222
10.1 Předmluva 222
Terminologická vsuvka 223
Syntaxe 223
Sémantika 223
Paradigma 223
Přehled paradigmat podporovaných *Javou* 224
Procedurální paradigma 224
Funkcionální paradigma 224
Objektově orientované paradigma 225
10.2 Trocha historie 225
10.3 Motivace OOP 226
10.4 Objekty 226
Členy objektů 227
10.5 Třídy a jejich instance 227
10.6 Třída jako objekt 228
10.7 Členy třídy a jejích instancí 229
Přežívající lokální proměnné 230
10.8 Zprávy 230
10.9 Metody 231
10.10 Entity 231
10.11 Polymorfismus, rozhraní, interfejs 232
Rozhraní × implementace 232
Atributy × vlastnosti 233
Vlastnosti v knihovně/platformě/frameworku *JavaFX* 234
Signatura × kontrakt 234
Rozhraní × `interface` 234
Interfejs a jeho instance 235
10.12 Objektové datové typy 236
10.13 Dědění 236
Jak potomek „rozšiřuje“ předka 237
Tři druhy dědění 237
Přirozené (nativní) dědění 238
Dědění typu (rozhraní) 238
Dědění implementace 239
Dva způsoby dědění, kachní typování 239
Statické a dynamické typování 239
Strukturální dědění 240
Kachní typování 240
Hierarchie jmenovitého dědění 240
Problémy s děděním – substituční princip Liskové (LSP) 240
10.14 Vlastní instance třídy a mateřská třída objektu 241
10.15 Tři základní principy OOP 242
10.16 Jazyk UML 243
10.17 Správa paměti 244
10.18 Další informace 245
10.19 Soubory pro opakování 245
11 Třídy a jejich členy 246
11.1 Nejjednodušší definice třídy 246

11.2 Konstruktory ... 247
Implicitní konstruktor ... 247
Vlastní konstruktor a skrytý parametr this ... 247
Proč se liší podpisy ... 248
Definice tříd jako úryvky ... 249
11.3 Třída se všemi členy ... 249
Statické (třídní) členy ... 250
Instanční členy ... 251
Konstrukce objektů ... 252
11.4 Kvalifikace posílaných zpráv ... 252
Implicitní kvalifikace ... 253
11.5 Přetěžování konstruktorů ... 254
Kvalifikace klíčovým slovem this ... 257
11.6 Modifikátory přístupu a skrývání implementace ... 258
Veřejné a „neveřejné“ datové typy ... 259
11.7 Přístupové metody ... 259
11.8 Modifikátor final ... 261
Konstantní atributy ... 261
Konstanty vyhodnotitelné v době překladu ... 261
Konstantní lokální proměnné ... 262
Efektivní konstanty ... 262
Zveřejňování konstantních atributů ... 262
Modifikátor final v procesu dědění ... 262
Neměnnost objektů ... 263
11.9 Primitivní a obalové datové typy – autoboxing ... 263
Převody stringů na hodnoty primitivních typů ... 265
11.10 Důležité metody klíčových tříd ... 266
Třída Object ... 266
Object clone() ... 266
Mělké a hluboké kopie objektů ... 266
boolean equals(Object) ... 266
Class<?> getClass() ... 267
int hashCode() ... 267
String toString() ... 267
void finalize() ... 267
Třída String ... 268
char charAt(int index) ... 268
boolean contains(CharSequence s) ... 268
boolean equals(Object) ... 268
boolean equalsIgnoreCase(String anotherString) ... 268
int indexOf(int ch) ... 268
int indexOf(String ch) ... 268
boolean isEmpty() ... 268
static String join(CharSequence delimiter, CharSequence... elements) static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements) ... 268
int length() ... 268
String replace(X oldX, X newX) ... 268
String[] split(String regex) ... 268
String strip() String trim() ... 269
String toLowerCase() String toUpperCase() ... 269
static String valueOf(X x) ... 269
Třída Class ... 269
boolean equals(Object) ... 269
String getName() ... 269

`String getSimpleName()` ... 269
`String toString()` ... 269
11.11 Soubory pro opakování ... **269**
12 Vytvoření aplikace a vývojová prostředí ... **270**
12.1 Doprovodné programy ... **271**
12.2 Malé ohlédnutí ... **271**
12.3 Zdrojové soubory třídy ... **272**
12.4 Představení balíčků ... **274**
Spuštění třídy z balíčku ... **275**
12.5 Překlad ... **276**
Překlad třídy ... **276**
Překlad rozsáhlejšího programu ... **278**
12.6 Používání knihoven ... **279**
12.7 JAR-soubory ... **279**
Vytvoření JAR-souboru ... **280**
Vytvoření knihovny ... **280**
Vytvoření spustitelného JAR-souboru ... **282**
12.8 Spuštění JAR-souborů ... **283**
Syntaktický diagram spouštění aplikace ... **284**
Java ... 284
ArgumentVM ... 284
Spouštěná třída ... 284
ArgumentProgramu ... 285
12.9 Spustitelný program používající knihovnu ... **285**
12.10 Vývojová prostředí – IDE ... **287**
JShell ... **287**
BlueJ* a *BlueJ++ ... **288**
Nejpoužívanější profesionální IDE ... **288**
IntelliJ IDEA ... 288
Eclipse ... 289
NetBeans ... 289
Visual Studio Code ... 289
12.11 Spuštění hlavní třídy a její metody `main(String[])` ... **290**
12.12 Soubory pro opakování ... **290**
13 Balíčky a knihovny ... **291**
13.1 Velké programy a jejich problémy ... **291**
13.2 Balíčky ... **292**
Umístění zdrojových souborů ... **293**
Kořenový (implicitní, defaultní, nepojmenovaný) balíček ... **293**
Podbalíčky ... **294**
Konvence pro názvy balíčků ... **294**
Balíčky doprovodných programů a knihoven ... **294**
Zakázaný balíček `java` ... **295**
Názvy datových typů ... **296**
13.3 Explicitní ukončení aplikace ... **297**
13.4 Příkaz `import` ... **298**
Import zadaného datového typu ... **298**
Import všech veřejných typů ze zadaného balíčku ... **298**
Podpora zadávání příkazu `import` ve vývojových prostředích ... **299**
Příkaz `import` v prostředí JShell ... **300**
Výjimečnost balíčku `java.lang` ... **300**
13.5 Příkaz `import static` ... **300**
13.6 Syntaktický diagram příkazu `import` ... **301**
13.7 Entity soukromé v rámci balíčku ... **302**

13.8 Typy se stejným názvem v různých balíčcích ... 304
Shrnutí ... 306
13.9 Použití knihovny v JShell ... 306
Nastavení proměnné `classpath` ... 306
Nastavení importů ... 306
Ukázky ... 307
Násilné ukončení aplikace ... 307
13.10 Zakázaný balíček `java` ... 309
13.11 Soubory pro opakování ... 309
14 Dokumentace API ... 310
14.1 Dokumentační komentáře a API ... 310
14.2 Proč psát srozumitelné a komentované programy ... 311
POBLIOCHA ... 312
Jak dokumentační komentáře zobrazovat ... 313
14.3 Jak psát dokumentační komentáře pro `javadoc` ... 313
14.4 Pomocné značky pro tvorbu dokumentace ... 314
14.5 Dokumentace balíčku a modulu ... 315
14.6 Vytvoření a zobrazení dokumentace ... 317
14.7 Struktura dokumentace API ... 317
Struktura dokumentace datového typu ... 318
Rychlé vyhledání ... 319
14.8 Zpřehlednění programu ... 319
14.9 Zakomentování a odkomentování části programu ... 321
14.10 Soubory pro opakování ... 321
15 Konstrukce interface ... 322
15.1 Definice typického interfejsu ... 322
Deklarace abstraktních metod ... 323
Příklad ... 323
15.2 Implementace interfejsu třídou ... 324
15.3 Interfejs se všemi přípustnými typy členů ... 326
Motivace k rozšíření – implicitní metody ... 326
Statické členy ... 328
Instanční členy ... 328
15.4 Dědění interfejsů ... 329
15.5 Příklad ... 329
Plynulé posuny ... 330
Plynulé změny velikosti ... 331
Sloučení knihoven ... 331
15.6 Výhody implicitních metod při návrhu architektury ... 332
15.7 Řešení kolizí ... 333
15.8 Specifikace zdroje použité metody ... 335
Možné problémy ... 335
15.9 Speciální interfejsy ... 337
Značkovací interfejsy ... 337
`java.lang.Cloneable` ... 337
`java.io.Serializable` ... 337
Současné trendy a doporučení ... 337
Funkční interfejsy ... 337
Interfejs `Iterable` ... 338
15.10 Soubory pro opakování ... 339
16 Podrobnosti o konstruktorech ... 340
16.1 Opakování: co víme o konstruktorech instancí ... 340
16.2 Zavádění třídy – `java.lang.ClassLoader` ... 341

16.3 Statický konstruktor – konstruktor třídy ... 342
Konstruktor interfejsu ... 342
16.4 Instanční inicializační blok ... 343
16.5 Dvě části těla konstruktoru instancí ... 343
16.6 Příklad ... 344
Konstruktor třídy ... 349
3–9: Úvodní statický inicializační blok ... 349
25: Předčasné použití atributu ... 350
8: Nekorektní použití metod ... 350
42: Předčasné použití konstanty ... 350
62: Nekorektní volání konstruktoru ... 351
Inicializační část konstruktoru instancí ... 351
12–15: Úvodní instanční inicializační blok ... 351
149: Deklarace konstanty loaded ... 351
151–155: Inicializační výpočet ... 352
163: Použití this v inicializaci ... 352
260–264: Závěrečný inicializační blok ... 352
Těla konstruktorů instancí ... 352
175–180: Bezparametrický konstruktor ... 352
188–194: Jednoparametrický konstruktor ... 353
203–208: Dvouparametrický konstruktor ... 353
218–231: Tříparametrický konstruktor ... 353
16.7 Experimenty ... 353
16.8 Doporučení ... 354
Jediný statický inicializační blok ... 354
Bez instančních inicializačních bloků ... 354
Inicializovat všechny atributy jednotně ... 355
16.9 Skutečný název metody konstruktoru ... 355
16.10 Soubory pro opakování ... 356
17 Úvod do dědění implementace ... 357
17.1 Úvodní poznámky ... 357
17.2 Definice dceřiné třídy ... 358
17.3 Rodičovský podobjekt ... 360
Dědění implementace od více rodičů ... 361
17.4 Konstruktor ... 361
Konstrukce rodičovského podobjektu ... 362
17.5 Přetížené verze konstruktorů, použití super × this ... 364
17.6 Konstruktory rodiče a potomka ... 365
17.7 Demonstrace chování konstruktorů ... 366
Definice třídy Graddaughter17 ... 366
Provedení akce před příkazem this() nebo super() ... 368
Definice metody constructorReport(Object,Class) ... 369
Spuštění testu ... 370
Zavedení třídy ... 370
Tisk nehotových objektů ... 370
Preference vlastních metod ... 372
Dokončení testu ... 372
Rodičovský podobjekt je abstrakce ... 372
17.8 Dědění přístupových práv ... 372
17.9 Zákaz vytváření potomků třídy ... 373
17.10 Zalepené třídy a interfejsy ... 373
Definice potomků uvnitř definice zalepeného typu ... 374
Definice potomků uvnitř definice zalepeného typu ... 374
Pravidla pro potomky ... 376
17.11 Soubory pro opakování ... 379

18 Viditelnost členů tříd 380
18.1 Úpravy použitého kódu 380
18.2 Trocha terminologie 381
Posílání zpráv a volání metod 381
Přetěžování×přebíjení×zakrývání×přepisování×předefinování metod 381
Přetěžování metod 381
Přebíjení metod 381
Zakrývání metod 382
Přepsání či předefinování metod 382
18.3 Chráněné členy – modifikátor přístupu `protected` 382
Shrnutí 385
18.4 Dědění metod 385
Zděděné, dále neupravované metody 386
Zděděné metody, pro něž potomek definuje „lepší“ implementaci 386
Kompatibilita signatur 386
18.5 Zakrývání metod předka (method hiding) 387
18.6 Metody, které není možno v potomku zakrýt či přebít – modifikátor `final` 390
18.7 Zakrývání atributů předka 391
18.8 Metody nově definované v potomku 393
Proč je situace jednoduchá jen zdánlivě 393
Anotace `@Override` 393
Staticky × dynamicky typované jazyky 394
18.9 Závěr 395
18.10 Soubory pro opakování 395
19 Virtuální metody a jejich přebíjení 396
19.1 Princip 396
Časná a pozdní vazba 397
Virtuální metody 397
19.2 Které metody jsou v Javě virtuální 398
19.3 Chování virtuálních metod 398
19.4 Zdokonalení třídy `Square` – třída `Square19` 401
Přebití metody `copy()` 401
Problémy s nastavováním velikosti 401
První návrh definice metody `setSize(int,int)` 402
Test prvního návrhu 403
Oprava 404
19.5 Co se nám na dědění nelíbí 406
19.6 Soubory pro opakování 406
20 Abstraktní třídy 407
20.1 Abstraktní třídy a jejich role v dědické hierarchii 407
Vytváříme hybrida 408
Abstraktní třída bez abstraktních metod 409
20.2 Konstruktor abstraktní třídy 409
20.3 Deklarace a implementace abstraktních metod 410
20.4 Účel abstraktních tříd 412
20.5 Proč společný rodič 413
20.6 Účel abstraktních metod 413
20.7 Návrhový vzor Šablonová metoda (Template method) 414
Princip 414
Implicitní metody interfejsů 414
Architektura balíčku `ruplib.geom` 415
Metoda `toString()` 416
20.8 Soubory pro opakování 418

Část III Pokročilejší objektové konstrukce 419

21 Výjimky a aserce 420
21.1 Co to jsou výjimky 421
21.2 Analýza chybové zprávy 421
Oznámení o chybě 421
Jak chyba vznikla – výpis zásobníku návratových adres 422
21.3 Nejdůležitější výjimky 423
21.4 Vyhození výjimky 424
Oddělené vytvoření výjimky 426
21.5 Výjimky a nedosažitelný kód 427
21.6 Co výjimky umí 427
21.7 Hierarchie dědění výjimek 428
21.8 Zachycení vyhozené výjimky 429
Chování metody `exceptionCatching(int)` 431
21.9 Syntaktický diagram bloku `try … catch` 431
Několik současně odchytávaných výjimek 431
Společná reakce na několik výjimek 432
Společný úklid – blok `finally` 433
Příklad 433
21.10 Definice vlastních výjimek 435
21.11 Kontrolované výjimky 436
21.12 Převedení kontrolované výjimky na nekontrolovanou 438
21.13 Informace o skutečném původci výjimky 439
21.14 Ověřování podmínek – příkaz `assert` 441
Design by Contract 442
Výběrová aktivace provádění příkazu `assert` 444
21.15 Kdopak mne to volal 446
21.16 Soubory pro opakování 446

22 Generické datové typy a metody 447
22.1 Motivace 447
22.2 Generické a parametrizované datové typy 450
22.3 Definice generických typů 451
22.4 Použití generických typů 453
22.5 Překlad generických datových typů a očišťování 455
22.6 Rizika nepoužití typových argumentů 456
22.7 Varování překladače a jejich potlačení 459
Zobrazená varování 460
22.8 Generické metody 461
22.9 Soubory pro opakování 465

23 Typové parametry a argumenty 466
23.1 Omezení typových argumentů 466
Typové argumenty s více předky 467
Vzájemné závislosti typových parametrů 467
23.2 Překlad a očišťování podrobněji 468
Doporučené pořadí omezujících interfejsů 468
Ztráta informace při běhu 470
Přemosťovací metody 470
23.3 Zakázané operace 472
Za typové parametry nelze dosazovat primitivní typy 472
Typové parametry třídy není možno použít u statických členů 472
Nelze vytvořit instanci typového parametru 472
Reflexe 473

Nelze vytvořit pole instancí typového parametru ani parametrizovaného typu ... 474
Výjimky ... 475
23.4 Proměnný počet argumentů – @SafeVarargs ... 475
Omezení ... 476
Vytvoření pole hodnot ... 477
23.5 Nejednoznačnosti a kolize ... 477
Falešně přetížená metoda ... 477
Nová metoda koliduje se zděděnou ... 478
Kolize požadovaných interfejsů ... 479
Kolize implementovaných interfejsů ... 480
Potomci a předci generických typů – špatné pochopení dědičnosti ... 480
23.6 Žolíky ... 481
23.7 Příklad: datový typ Interval<T extends Comparable<? super T>> ... 483
23.8 Soubory pro opakování ... 486
24 Interní datové typy ... 487
24.1 Motivace ... 487
Pomocný soukromý typ ... 487
Objekt znající útroby a implementující veřejné rozhraní ... 488
Sdružení souvisejících typů ... 488
24.2 Terminologie ... 488
24.3 Společné charakteristiky interních typů ... 490
24.4 Globální interní (členské) datové typy ... 490
24.5 Vnořené datové typy ... 491
24.6 Vnitřní třídy ... 492
Interní interfejsy a výčtové typy bez modifikátoru static ... 492
24.7 Příklad na vnořené a vnitřní třídy ... 493
Vnořená Elements × vnitřní SAIterator ... 494
Veřejná Elements × soukromá SAIterator ... 494
Definice třídy SparseArray.Element ... 495
Definice třídy SparseArray.SAIterator ... 495
Definice třídy SparseArrayTest ... 496
24.8 Lokální třídy a interfejsy ... 498
Pojmenované lokální třídy ... 499
Anonymní třídy ... 499
Použití anonymních tříd ... 501
24.9 Ještě jednou názvy tříd ... 501
24.10 Soubory pro opakování ... 502
25 Výčtové typy – třídy typu enum ... 503
25.1 Nejjednodušší definice ... 503
Překladačem přidané atributy a metody ... 505
Atribut $VALUES ... 505
public static final NázevTypu[] values() ... 506
public static NázevTypu valueOf(String name) ... 506
25.2 Třída Enum ... 506
Zděděné metody ... 510
public final String name() ... 510
public final int ordinal() ... 510
public final int compareTo(E o) ... 510
public final Class<E> getDeclaringClass() ... 510
public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name) ... 511
Přebité verze metod zděděných od třídy Object ... 511
protected final Object clone() throws CloneNotSupportedException ... 511
equals(Object) ... 511

`hashCode()` ... 511
`public String toString()` ... 511
Serializace ... 511
25.3 Použití výčtových typů v programu ... 512
Přepínač ... 512
Přepínač jako výraz ... 513
Přidaná anonymní třída ... 513
Další možnosti ... 514
Cyklus ... 514
25.4 Složitější definice výčtových typů ... 514
25.5 Akční výčtové typy ... 518
Class-objekty instancí funkčních výčtových typů ... 519
25.6 Soubory pro opakování ... 520
26 Záznamové třídy – třídy typu record ... 521
26.1 Objekty reprezentující hodnotu (ORV) a objekty reprezentující entitu (ORE) ... 521
Proměnné a neměnné hodnotové typy ... 522
26.2 Přepravky ... 523
Problémy a motivace ... 523
Koncepční vlastnosti ... 523
26.3 Základní syntaxe záznamových tříd ... 524
26.4 Automaticky definované členy ... 524
Názvy přístupových metod ... 525
26.5 Některé možnosti ... 525
26.6 Kanonický konstruktor ... 526
26.7 Možnosti a omezení ... 528
26.8 Komplexnější definice ... 530
Interfejs `ITriangleSSS` ... 530
Třída `RectangularTriangleSSS` ... 531
Třída `GeneralTriangleSSS` ... 534
Prověrka řešení ... 534
Možné alternativy ... 534
26.9 Soubory pro opakování ... 536
27 Další použití rozpoznávání vzorů ... 537
27.1 Rozpoznání vzoru záznamu ... 537
Proměnné s hodnotami atributů ... 537
Proměnné s vnořenými záznamy nebo jejich atributy ... 538
Generické datové typy ... 540
27.2 Rozšíření funkcionality výrazu/příkazu `switch` ... 542
Rozpoznávání vzorů ve výrazu/příkazu `switch` ... 543
Nemožnost použití násobných vzorů ... 544
Použití hodnoty `null` a omezení `when` ... 544
Detekce konstant výčtových typů ... 545
Nevyčerpání všech možností ... 547
Rozpoznání typových parametrů ... 548
27.3 Nepojmenované vzory a proměnné (preview) ... 550
27.4 Soubory pro opakování ... 551
28 Lambda-výrazy ... 552
28.1 Motivace ... 552
28.2 Koncepce lambda-výrazů ... 553
28.3 Funkční interfejsy ... 553
28.4 Syntaxe lambda-výrazů ... 556
Jednoduchý příklad ... 557
Deklarace parametrů ... 557
Lambda-výrazy zastupující metody ... 559

Lambda-výraz vedoucí na volání konstruktoru ... 560
28.5 Použití lokálních proměnných z okolního bloku ... 560
28.6 Trochu složitější příklad ... 562
Atributy ... 564
Konstruktory ... 564
Metoda `copy()` ... 565
Přístupové metody ... 565
Metoda `paint(Painter)` ... 566
Metody `switchOff()` a `switchOn()` ... 566
Metoda `toString()` ... 566
Testovací třída ... 566
28.7 Soubory pro opakování ... 568
29 Anotace a šablonové procesory ... 569
29.1 Co jsou anotace ... 569
29.2 Označování deklarací anotacemi ... 570
29.3 Kde všude můžeme anotace použít ... 572
Anotování balíčků ... 573
Anotování parametru `this` ... 574
29.4 Anotace ve standardní knihovně ... 574
Standardní anotace v balíčku `java.lang` ... 574
`@Deprecated` ... 575
`@Override` ... 575
`@SuppressWarnings` ... 575
`@SafeVarargs` ... 576
`@FunctionalInterface` ... 576
Standardní anotace v balíčku `javax.annotation` ... 576
Metaanotace ... 576
`@Documented` ... 576
`@Inherited` ... 576
`@Repeatable` ... 577
`@Retention` ... 577
`@Target` ... 578
29.5 Syntaxe definice anotací ... 578
Jednoduchá značkovací anotace ... 580
29.6 Získávání informací o anotacích za běhu programu ... 581
29.7 Šablonový výraz a jeho procesory ... 581
Interní zpracování ... 582
29.8 Soubory pro opakování ... 583
30 Vlákna a paralelní procesy ... 584
30.1 Paralelní provádění více činností ... 584
Kooperativní plánování ... 585
Preemptivní plánování ... 585
Použité plánování ... 585
30.2 Vlákna a jejich stavy ... 585
30.3 Sdílení zdrojů ... 587
30.4 Kritické sekce a monitory ... 588
30.5 Synchronizace ... 588
30.6 Uvolnění kritické sekce ... 589
30.7 Jemnější způsoby synchronizace ... 589
Modifikátor `volatile` ... 589
Atomické objekty ... 590
30.8 Novější metody práce s vlákny ... 590
30.9 Virtuální vlákna ... 590

Dřívější stav a jeho nevýhody ... 590
Výhody virtuálních vláken ... 591
Koncept virtuálních vláken ... 591
30.10 Soubory pro opakování ... 592
31 Moduly ... 593
31.1 Motivace ... 593
Problémy předchozích verzí *Javy* ... 594
Cíle projektu Jigsaw ... 595
Dosažené výhody ... 595
31.2 Srovnání instalace Javy 8 a Javy 21 ... 596
31.3 Modul × balíček ... 597
31.4 Soubor `module-info.java` ... 597
Syntaktický diagram deklarace modulu ... 598
Název modulu ... 598
Direktiva `requires` ... 600
Direktiva `exports` ... 600
Direktiva `opens` a modifikátor `open` ... 600
Direktiva `uses` ... 601
Direktiva `provides` ... 601
31.5 Modulární JAR-soubor ... 601
31.6 Proměnná `modulepath` ... 601
31.7 Klasifikace modulů ... 602
Běžný modul (normal module) ... 602
Otevřený modul (open module) ... 603
Automatický modul (automatic module) ... 603
Vlastnosti automatických modulů ... 604
Nepojmenované moduly (unnamed modules) ... 605
31.8 Moduly a platforma JShell ... 605
31.9 Soubory pro opakování ... 606
32 Kategorie datových typů, hodnotové typy ... 607
32.1 Trocha historie ... 608
Proměnné ... 608
Dynamická alokace ... 608
Automatická správa paměti ... 608
32.2 Kategorizace typů z hlediska umístění dat ... 609
Primitivní versus objektové datové typy v Javě ... 609
Odkazované hodnoty primitivních typů ... 610
Výhody a nevýhody použití odkazů ... 610
Neodkazované instance objektových typů – „hodnotové“ datové typy ... 610
32.3 Přístup Javy – projekt Valhalla ... 611
32.4 Obalové datové typy ... 613
32.5 Předběžné zavedení hodnotových typů ... 614
32.6 Soubory pro opakování ... 614

Část IV Přílohy 615

A Tvorba jednoduchého GUI ... 616
A.1 Trocha historie – přehled rozšířených platforem ... 616
Platforma AWT ... 616
Platforma Swing ... 617
Platforma SWT ... 617
Platforma JavaFX ... 617
A.2 Základní koncepce platforem pro tvorbu GUI ... 618
GUI a využívání vláken ... 618

Modalita dialogových oken ... 619
A.3 Prostředky pro triviální okenní vstup a výstup ... 619
Třída eu.pedu.lib20s.util.IO ... 620
void setDialogsPosition(int x, int y) ... 620
boolean confirm(Object question) ... 620
int choose(Object question, String... buttons) ... 620
double enter(Object prompt, double defaultDouble) int enter(Object prompt, int defaultInt) String enter(Object prompt, String defaultString) ... 620
String select(Object prompt, String... options) ... 620
void inform(Object text) ... 620
Třída javax.swing.JOptionPane ... 620
showMessageDialog ... 621
showConfirmDialog ... 621
showOptionDialog ... 621
showInputDialog ... 621
A.4 Základy koncepce platforem AWT a Swing ... 621
Komponenty a kontejnery ... 621
Správce rozvržení – LayoutManager ... 622
BorderLayout ... 622
BoxLayout ... 622
FlowLayout ... 622
Kontejnery – okna a panely ... 622
JFrame ... 623
Box ... 623
JPanel ... 623
Události a jejich posluchači ... 623
A.5 Příklad ... 623
Atributy a rodiče ... 624
Konstruktor ... 624
Metoda axisPanel(Box box, int axis) ... 627
Posluchač událostí ... 627
Opakování ... 628
Metoda addButtonPanel() ... 628
Pokračujeme definicí konstruktoru ... 628
Spuštění aplikace ... 629
A.6 Závěr ... 629
Literatura ... 630
Rejstřík ... 632

Úvod

Tato kniha seznamuje s dvacátou verzí jazyka *Java*. Soustředí se především na výklad vlastností jazyka a minimalizuje výklad probírající obsah standardní knihovny – tomu se do jisté míry věnuje příručka [24].

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 výše zmíněnou učebnicí [24], případně s některou z mých starších příruček věnovaných úvodu do programování, v nichž nevysvětluji jen to, jak se program zapisuje, ale soustředím se především na to, jak se vymýšlí. V těchto příručkách spolu navrhujeme architekturu programu.

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, jenom poněkud hutněji.

Jak už jsem řekl, kniha se soustředí na výklad jazyka. Tím, že jsem výklad knihoven (např. práce s kolekcemi, datovody, soubory a datovými proudy, regulárními výrazy či vlákny) odložil do jiných příruček, jsem ušetřil 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 primitivní GUI. Nejprve představí nástroje použitelné pro superjednoduché GUI, pak principiálně naznačí, jak se navrhuje komplexní GUI. Tato část je organickou součástí příručky pouze u elektronických verzí. Čtenáři tištěné verze ji najdou jako PDF soubor na webové stránce knihy.

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 21 Edition* ([6]), 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 21* označovanou JDK, což je zkratka z anglického *Java Development Kit*. Stáhnete ji ze stránky https://www.oracle.com/java/technologies/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 21 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 Vytvoření aplikace a vývojová prostředí* na straně 270 začnu vedle prostředí *JShell* používat i prostředí *IntelliJ IDEA*. Demonstruji v něm vlastnosti, které se projeví až u samostatně vyvíjených programů. Nelpím přitom na tom, abyste používali právě toto prostředí, ale zvolil jsem je proto, že je v současné době s odstupem nejpoužívanější.

Adresy, z nichž si můžete stáhnout některé z doporučovaných prostředí, najdete v pasáži *Nejpoužívanější profesionální IDE* na straně 288. 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/72_j20lang. Zde jsou umístěny některé doprovodné texty a především pak ZIP-archiv s doprovodnými programy. Po rozbalení tohoto archivu se vytvoří čtyři složky:

- Složka `72_INP` bude obsahovat skripty pro prostředí *JShell*, které stačí zkopírovat do schránky a zadat v prostředí *JShell* jako vstupní data.
- Složka `72_IWD` bude obsahovat úryvky a příkazy z výše uvedených skriptů spolu s reakcemi systému tak, jak jsou uvedeny ve výpisech v textu knihy, a to včetně čísel řádků. Mohlo by vám to pomoci se při pročítání výkladu snáze orientovat v textu analyzovaných výpisů.
- Složka `72_Java` bude obsahovat zdrojové soubory, na něž začneme přecházet ve druhé části knihy, kde už budou vysvětleny všechny konstrukce potřebné pro zadání standardního zdrojového kódu, a my budeme moci začít vyvíjet standardní aplikace. Tyto zdrojové kódy budete moci otevřít a dále zpracovávat ve vašem oblíbeném vývojovém prostředí.
- Složka `72_LibSrc` bude obsahovat zdrojové kódy používaných knihoven.
- Složka `72_Lib` bude obsahovat archiv s grafickou knihovnou, kterou budete moci ve svých programech používat.

Všechny textové soubory budou používat kódování UTF-8, na které konečně *Java* přešla jako na standardní způsob kódování bez ohledu na aktuálně používaný operační systém.

Já pracuji pod operačním systémem *Windows*, protože jej používá 85–90 % mých studentů a účastníků firemních kurzů. Všechny uvedené složky mám na svém počítači umístěné v kořenové složce substituovaného disku `M:`. Pokud by mne chtěl někdo následovat a nevěděl, jak se vytvářejí substituované disky, najde stručný popis v příloze.

Zlom

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 argumentu

```
--enable-preview
```

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

Globální nastavení podpory předběžných konstrukcí

Pracujete-li na finálním projektu, bude vám vyhovovat, že jsou předběžné konstrukce blokovány a musíte o jejich zprovoznění explicitně požádat.

Pokud naopak studujete nebo se prostě zajímáte o novinky jazyka, mohlo by vás obtěžovat, že při každém spuštění musíte zadávat argument `--enable-preview`. Mohlo by vás pak potěšit, že pro překladač i virtuální stroj je možné tento argument zadat globálně, takže bude při každém spuštění zadán automaticky.

Globální automatické zadávání libovolného argumentu překladače, resp. virtuálního stroje se nastavuje prostřednictvím systémových proměnných prostředí (anglicky environment variables), a to:

- `JDK_JAVAC_OPTIONS` nastavuje argumenty pro překladač,
- `JDK_JAVA_OPTIONS` nastavuje argumenty pro virtuální stroj.

Předpokládám, že uživatelé *Linuxu* potřebný postup znají. Uživatelé *Windows*, kteří s nastavením systémových proměnných nemají zkušenosti, najdou návod v příloze na webové stránce knihy.

---

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

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 Backus-Naurovy 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 64.

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

[2] Podrobněji se o koncepci inkubačních modulů dozvíte v *1JEP 11: Incubator Modules*, které najdete na adrese https://openjdk.java.net/jeps/11.

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(?)`.

`JShell>` Výzvy programu *JShell* k zadání dalšího příkazového řádku budou podbarveny a barevně odlišeny.

`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).

`klíč` Klíčová slova jsou v programech pro zvýraznění vysazena tučně tmavě červenou barvou. Aby byla zvýrazněna i v tištěných knihách, jsou navíc podtržena.

`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 jin-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.

Otevřená schránka s dopisy označuje informace o kódu, s nímž budeme v dalším textu pracovat, nebo v něm najdete vzorové řešení aplikující probranou látku.

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 72_JAVA_21_ZPRAVA na adresu

rudolf@pecinovsky.cz. Na stránku knihy http://knihy.pecinovsky.cz/72_j21lang se pak pokusím co nejdříve zanést příslušná errata s opravou, kterou zapracujeme do případného dalšího vydání.

Tento mail můžete poslat i v případě, bude-li vám někde připadat text nepříliš srozumitelný nebo budete-li mít nějaký dotaz, ať už k vykládané látce či použitému vývojovému prostředí. Vzhledem ke svému velkému vytížení se však k probrání pošty dostanu jen jednou za čas, takže odpovídám s velkým zpožděním. Dostanete-li se proto do úzkých a něco se vám nebude dařit rozchodit, navštivte stránku http://vyuka.pecinovsky.cz/konzultace.html, zjistěte si, kdy neučím, a neostýchejte se zavolat. Kdybych to nezvedal, pošlete SMS s informací, že voláte s dotazem ke knize, a já se vám co nejdříve ozvu.

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* ([23]).

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.

Java byla uvedena jako jazyk podporující objektově orientované paradigma. Měli bychom si proto pro jistotu nejprve vysvětit, co to znamená.

Objektově orientované paradigma[3] – OO paradigma, OOP

Objektově orientované paradigma (nebo chcete-li objektově orientované programování) je postavené na myšlence, že každý program je simulací reálného či virtuálního světa. Tento simulovaný svět je tvořen objekty, které na sebe vzájemně působí. Programátoři říkají, že „*objekty si posílají zprávy*".

Tezi, že simulovaný svět je tvořen objekty, bychom mohli interpretovat tak, že v něm *je všechno 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*".

Jako objekt jsou pak chápány nejenom věci, které běžně označujeme jako objekty, ale i události (např. přerušení, výpočet, …), vlastnosti (směr, barva, …), anebo abstraktní pojmy (krása, spravedlnost, …).

S objektově orientovaným programováním a příslušným paradigmatem se začneme podrobněji seznamovat v kapitole *10 Základy objektově orientovaného paradigmatu* na straně 222. V dalším textu pak budu často místo zdlouhavého *objektově orientovaný* používat pouze zkratku OO.

Objekt v programu

Abychom mohli v programu s nějakým objektem pracovat, musíme jej nějak popsat. Jinými slovy, potřebujeme dát dohromady data, která jej budou charakterizovat. *V programu budeme termínem objekt označovat množinu dat charakterizujících reprezentovaný objekt simulovaného světa*. Je přitom jedno, zda tato data charakterizují nějaký konkrétní objekt v simulovaném světě (např. osobu nebo bankovní účet), anebo některý z výše uvedených abstraktních „objektů" (např. přerušení, směr či krásu).

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. V praxi říkáme, že objekt je **instancí** nějaké třídy.

Datový typ říká, co jsou jeho instance zač, jaké mají vlastnosti a schopnosti – jinými slovy: co po nich můžeme chtít.

Třídy začneme podrobněji probírat v kapitole *11 Třídy a jejich členy* na straně 246.

Podle zásad OOP je i třída objekt. Protože však *Java* nedefinuje třídu, jejímiž instancemi by byly třídy (tedy metatřídu), tak se v *Javě* nedá s třídami pracovat jako s běžnými objekty, ale můžeme s nimi provádět pouze podmnožinu operací s objekty. V *Javě* například nemůžeme uložit třídu do proměnné.

[3] Termínem paradigma označujeme celkový přístup k dané problematice – v našem případě k programovému řešení zadaného problému.

Proměnná

Proměnná je pojmenované místo v paměti, kam si můžeme uložit nějakou hodnotu pro budoucí použití. Na toto místo (přesněji na v něm uloženou hodnotu) pak odkazujeme jeho názvem.

S proměnnými a jejich vlastnostmi se začneme podrobněji seznamovat v kapitole *3 Proměnné* na straně 82.

Atributy

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

Metody

Termínem *metoda* označujeme pojmenovanou část kódu patřícího danému objektu. Metodu můžeme spustit a získat od ní nějakou hodnotu a/nebo ji nechat provést nějakou akci. Metody můžeme vnímat jako kód definující reakci svého majitele na zaslanou zprávu.

Možnostem definice vlastních metod a jejich vlastnostem se začneme věnovat v kapitole *5 Definice metod* na straně 115.

Vlastnosti

Jako vlastnost označujeme charakteristiku objektu, kterou mohou okolní objekty zjistit, nebo nastavit. Zjišťování a nastavování vlastnosti se v *Javě* realizuje prostřednictvím přístupových metod.

Vlastnostem a jejich přístupovým metodám se budeme věnovat v podkapitole *11.7 Přístupové metody* na straně 259.

Interní typy

Za interní datový typ považujeme datový typ definovaný uvnitř jiného datového typu.

Interními datovými typy se začneme zabývat v kapitole *24 Interní datové typy* na straně 487.

Členy

Objekty mají svoje atributy, metody a interní datové typy. Všechny tyto součásti dohromady označujeme jako **členy** daného objektu.

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 samostatného překladu a následného spuštění přeloženého kódu.

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

Máte-li správě instalované JDK, měl by jít *JShell* spustit z okna konzoly (terminálu, panelu příkazového řádku) zadáním příkazu

```
jshell
```

případně, chcete-li podrobnější zprávy o tom, jak *JShell* zpracoval vámi zadaný příkaz, tak příkazem

```
jshell -v
```

kde argumentem `-v` žádáme *JShell*, aby nám o všech provedených akcích referoval co nejpodrobněji. Budete-li chtít spustit *JShell* v režimu podporujícím předběžně definovanou funkcionalitu plánovanou do příštích verzí (preview features), spustíte jej příkazem

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

Budete-li chtít spouštět nějaké předem připravené skripty (např. doprovodné programy této příručky), je vhodné jej spustit ve složce, v níž se dané soubory nacházejí.

Není-li tato složka aktuální složkou, tak se do ní v panelu příkazového řádku přesunete vhodně zadaným příkazem `cd` (change directory). Uživatelé *Windows* mají možnost otevřít danou složku v *Průzkumníku* a zadat nahoru do adresního pole příkaz `cmd`, čímž otevřou panel příkazového řádku právě v této složce.

Nepodaří-li se program spustit ze zadané složky, zkuste se nejprve přesunout do složky, kam jste instalovali *Javu*. Program *JShell* by se měl nacházet v její podsložce `bin`.

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 argumenty.

Pro tento kurz dám přednost druhé cestě právě kvůli možnosti zadávat dodatečné argumenty. 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.[4] My budeme používat první z nich, tj. soubor s příponou `BAT`. Postupujte následovně:

1. Otevřete složku, do které jste z archivu doprovodných programů uložili soubor `Start.jsh`.
2. Ve svém oblíbeném textovém editoru[5] vytvořte prázdný textový soubor.
3. Do souboru napište příkaz

   ```
   chcp 1250
   ```

   který nastaví správnou kódovou stránku.[6] 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
   ```

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

[5] 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.

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

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 této příručce 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.

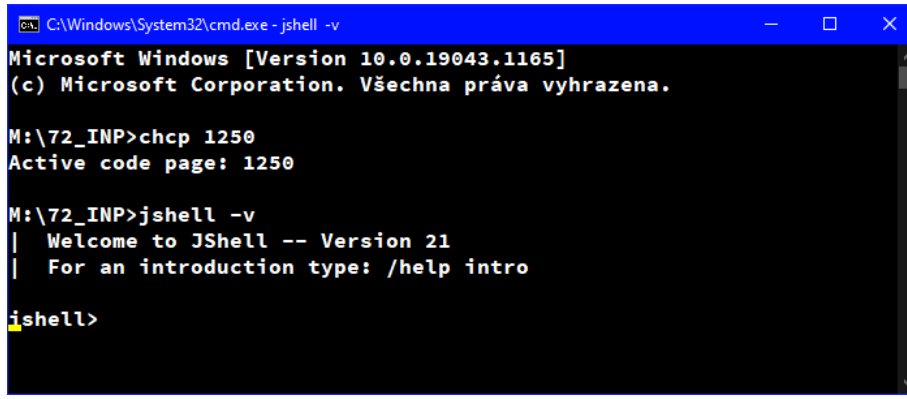
Řada začínajících programátorů ponechává nastavení *Windows* pro laické uživatele, které matou přípony souborů. Proto jsou přípony známých typů implicitně skryty. Necháváte-li ale operační systém, aby před vámi skrýval přípony, tak se stává, že vám sice průzkumník ukazuje, že se ve složce nachází soubor `!_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.**

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*,[7] 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.

[7] 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*.



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 M:\72_INP>chcp 1250
2 Active code page: 1250
3
4 M:\72_INP>jshell -v
5 |  Welcome to JShell -- Version 21
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.

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`.

V úvodu jsem sice psal, že tato kniha není pro začátečníky, ale v obdržených reakcích na předchozí vydání se ukázalo, že po ní občas bystří začátečníci sáhnou. Případným úplným začátečníkům tedy prozradím, že **proměnná** je pojmenované 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. Podrobněji si o nich povíme v kapitole 3 *Proměnné* na straně 82.

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.

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.

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>
```

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 úryvku**.

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*

```
jshell> $1+$3  $1+$4
|  Error:
|  ';' expected
|  $1+$3  $1+$4
|       ^
|  Error:
|  not a statement
|  $1+$3  $1+$4
|         ^---^
|  Error:
|  cannot find symbol
|    symbol:   variable $4
|  $1+$3  $1+$4
|            ^^
|  Error:
|  unreachable statement
|  $1+$3  $1+$4
|         ^---^

jshell> $1+$3; $1+$4
$4 ==> 44
|  created scratch variable $4 : int
$5 ==> 55
|  created scratch variable $5 : int

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 `ID` 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.

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).

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

```
1  jshell> /list
2
3     1 : 6+5
4     2 : 6 +
5         7+
6         9
7     3 : $1+$2
8     4 : $1+$3;
9     5 :  $1+$4
10
11 jshell> /drop 2
12 |  dropped variable $2
13
14 jshell> /list
15
16    1 : 6+5
17    3 : $1+$2
18    4 : $1+$3;
19    5 :  $1+$4
20
21 jshell>
```

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` je doplněno předponou `s` a počítá se samostatně) 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 82.
- Příkaz `/methods`, resp. `/methods -all` zobrazí aktivní, resp. všechny definované metody. Definovat metody se naučíte v kapitole *5 Definice metod* na straně 115.
- 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 246.
- Příkaz `/imports` zobrazí zadané importy. O importování různých částí programu si povíme v kapitole *13 Balíčky a knihovny* na straně 291.

Definice metod, datových typů ani importů jsme doposud neprobírali, takže si na použití odpovídajících příkazů musíte počkat.[8] 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>
```

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.

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

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 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.

Nastavení 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.

Nastavení zpětnovazebního režimu: /set feedback

JShell umožňuje nastavit míru podrobností informace o tom, co provedl při zpracovávání zadaného úryvku. Můžeme si vybrat jeden ze čtyř režimů:

`verbose` Podrobný režim (verbose mode) zobrazuje zpracovaný výsledek, občas také přidá vysvětlující komentář. Před další výzvou odřádkuje. V tomto režimu bude dále zobrazována většina doprovodných programů spouštěných v prostředí *JShell*.

`normal` Standardní režim (normal mode) se od podrobného režimu liší tím, že nepřidává vysvětlující komentáře.

`concise` Stručný režim (concise mode) se již neobtěžuje s oznamováním toho, že udělal to, o co jste ho požádali. Oznamuje však přiřazení hodnoty do proměnné. Kromě toho neodřádkuje před vypsáním nové výzvy.

`silent` Tichý režim (silent mode) neposkytuje žádnou dodatečnou zpětnou vazbu. Vypisuje pouze chybové zprávy a texty, které úryvek záměrně tiskne. Navíc používá místo čtyřznakové výzvy pouze dvouznakovou.

Zpětnovazební režim se nastavuje příkazem

```
/set feedback <mode>
```

kde `<mode>` zastupuje některý z výše uvedených režimů. Místo argumentu `feedback` stačí napsat jeho počáteční písmena (nejméně dvě), režim stačí uvést pouze písmenem jediným – např. `/set fe s`.

Mezi argument `feedback` a následující režim je možné vložit argument `-retain`, jehož zadáním *JShell* žádáme, aby si zadané nastavení zapamatoval a použil je i při příštím spuštění. I tento argument stačí uvést pouze počátkem – např.

```
/set fe -r c
```

Aktivace dalšího zdroje: /env

Příkaz `/env` slouží k zadání některých úprav parametrů běhového prostředí – např. kde všude se mohou nacházet používané části kódu.

Většina programů neřeší svůj problém osamoceně, ale využívá nejrůznější knihovny a frameworky. Chce-li je program využívat, musí nejprve překladači prozradit, kde je najde, aby mohl zkontrolovat, že je vše použito korektně.

Překladač má pro tento účel definovanou speciální proměnnou nazvanou `classpath`. V ní je uložen seznam cest ke kořenovým složkám stromů se zdrojovými a/nebo přeloženými soubory. Chceme-li používat nějakou knihovnu, musíme do této proměnné přidat cestu k jejím přeloženým souborům, které mohou být uloženy v nějaké složce, anebo ve zvláštním JAR-souboru. O JAR-souborech si budeme povídat v podkapitole *12.7 JAR-soubory* na straně 279.

V prostředí *JShell* zadáte použité cesty prostřednictvím příkazu `/env` s argumentem `-class-path`, za nímž uvedete seznam cest ke knihovnám a frameworkům, které chcete používat. Knihoven můžete zadávat i více, ale musíte zadat všechny současně, protože *JShell* v reakci na tento příkaz začíná tím, že všechny doposud zadané rozšiřující knihovny zapomene. Zadávané cesty k jednotlivým knihovnám přitom oddělujte oddělovačem cest daného operačního systému: ve *Windows* středníkem, v *Linuxu* dvojtečkou.

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 310.

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*.

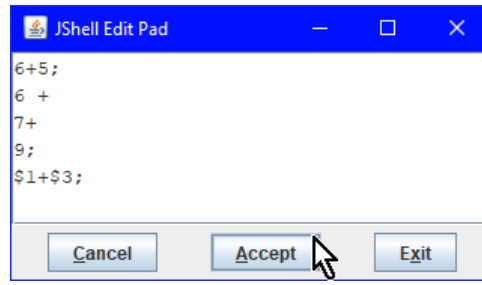
Použití editoru

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.

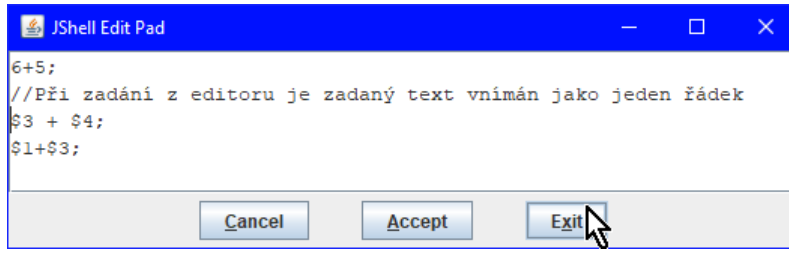
Okno se ovládá tak, že si v něm prohlédnete zobrazený kód a v případě potřeby jej upravíte. Význam tlačítek je následující:

- 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 45.
- 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*

```
1  jshell> /edit 1 $2 4
2  $7 ==> 77
3  |  created scratch variable $7 : int
4
5  jshell> /list
6
7     1 : 6+5
8     3 : $1+$2
9     4 : $1+$3;
10    5 :  $1+$4
11    6 : $1 + $5     //Komentář jako součást úryvku
12    7 : //Při zadání z editoru je zadaný text vnímán jako jeden řádek
13        $3 + $4;
14
15 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

Jak už bylo řečeno v pasáži *Doprovodné programy* na straně 28, v první části knihy je probíraná látka demonstrována prakticky výhradně na úryvcích programů spouštěných v prostředí *JShell*, které po rozbalení archivu s doprovodnými programy najdete ve složce `72_INP`. Najdete zde tři typy souborů s příponou `jsh`:

- Soubory, u nichž za počátečním dvojčíslím označujícím číslo kapitoly následují dva znaky podtržení, obsahují příkazy a úryvky zadávané v odpovídající kapitole doplněné o komentáře označující číslo výpisu, v němž je daná sada úryvků zobrazena i s odpověďmi systému.
- Soubory, u nichž za počátečním dvojčíslím následuje písmeno a jeden znak podtržení, obsahují samostatné skripty používané v dané kapitole.
- Soubory 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 ve složce `72_IWD` mají příponu `jdoc` a obsahují kopie výpisů z knihy včetně čísel řádků, abyste si je mohli zobrazit samostatně při pročítání jejich analýzy v knize.

1.10 Soubory pro opakování

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

Kapitola 2
Základní datové typy a jejich literály

Co se v kapitole naučíte
Tato kapitola vás seznámí se základními datovými typy: 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 rozhovořím o tom, jak pracovat se zprávami, které vracejí hodnoty, musím nejprve těm méně zkušeným 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 jsme si již řekli, *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 115). 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 patří mezi *objektové datové typy*. S nimi se pracuje jinak. Objektových typů bývá v programu definováno většinou mnohem víc. Jenom ve standardní knihovně *Javy 21* 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 cifer.

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.

Implicitní kódování znaků – UTF-8

Standardní API jazyka Java pro čtení a zápis souborů a pro zpracování textu umožňují předávat některým metodám pracujícím s texty v argumentu znakovou sadu, která definuje převod mezi zpracovávanými bajty a 16bitovými znakovými hodnotami programovacího jazyka Java.

Není-li argument `charset` předán, standardní API Javy obvykle používá implicitní znakovou sadu (anglicky *default charset*). V *Javě 17* a dřívějších verzích byla implicitní znaková sada určena při spuštění běhového prostředí Javy na základě informací získaných od operačního systému. Protože výchozí znaková sada nebyla všude stejná, představovalo její používání mnohá skrytá nebezpečí, a to i pro zkušené vývojáře.

Při přípravě verze 18 bylo proto rozhodnuto implicitní kódování sjednotit. *Java 18* tedy změnila specifikaci statické metody `java.nio.charset.Charset.defaultCharset()`, která nyní pro všechny operační systémy vrací implicitní kódování UTF-8. Pokud by někdo chtěl používat jiné implicitní kódování, musí je zadat už při zavádění *Javy*.

Objektové datové typy

Jak jsem již řekl, objektových datových typů je nepřeberné množství[9] a sami můžete vytvářet další. Podrobněji se jim budu věnovat v druhé části příručky. 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

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

straně 236). 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 odkazy 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.

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).

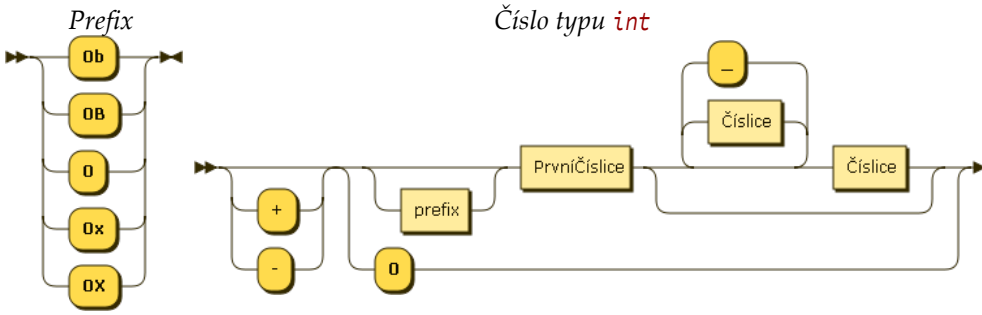
Délka slov, která je mocninou čísla 2 (4, 8, 16, 32, 64 či 128 bitů) se ustálila později. Spolu s ní se prosadilo i používání šestnáctkové soustavy, protože jedna šestnáctková číslice je ekvivalentem čtyř číslic dvojkových, takže obsah slova je výhodně definován příslušným počtem šestnáctkových číslic.

Šestnáctková (můžete se také setkat s termínem hexadecimální[10]) 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.

[10] 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.

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

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.