

edice

začínáme s ...

PROGRAMOVÁNÍ V JAZYKU JAVA

Učebnice pro budoucí profesionální programátory

- » Přehled základních jazykových konstrukcí a principů OOP
- » Návrh objektové architektury pro škálovatelné aplikace
- » Tvorba základního rámce projektu a jeho API pro efektivní práci týmu
- » Zefektivnění vývoje pomocí testovacích scénářů

📄 soubory ke stažení na www.grada.cz



edice
začínáme s ...

Programování v jazyku JAVA

**Učebnice pro budoucí
profesionální programátory**

RUDOLF PECINOVSKÝ

Rudolf Pecinovský

Programování v jazyku Java

Učebnice pro budoucí profesionální programátory

Vydala Grada Publishing, a.s.
U Průhonu 22, Praha 7
obchod@grada.cz, www.grada.cz
tel.: +420 234 264 401
jako svou 10132. publikaci

Odpovědný redaktor Petr Somogyi
Fotografie na obálce Depositphotos/KostyaKlimenko
Grafická úprava a sazba Rudolf Pecinovský
Počet stran 576
První vydání, Praha 2025
Vytiskly Tiskárny Havlíčkův Brod, a. s.

© Grada Publishing, a.s., 2025
Cover Design © Grada Publishing, a. s., 2025
Cover Photo © Depositphotos/KostyaKlimenko

Upozornění pro čtenáře a uživatele této knihy
Všechna práva vyhrazena. Žádná část této tištěné či elektronické knihy
nesmí být reprodukována a šířena v papírové, elektronické či jiné podobě
bez předchozího písemného souhlasu nakladatele.
Neoprávněné užití této knihy bude trestně stíháno.
Názvy produktů, firem apod. použité v knize mohou být ochrannými známkami
nebo registrovanými ochrannými známkami příslušných vlastníků.
Automatizovaná analýza textů nebo dat ve smyslu čl. 4 směrnice 2019/790/EU
a použití této knihy k trénování AI jsou bez souhlasu nositele práv zakázány.

ISBN 978-80-271-8054-7 (ePub)
ISBN 978-80-271-8053-0 (pdf)
ISBN 978-80-271-5751-8 (print)

Věnováno mé ženě Jarušce

Stručný obsah

Poděkování	22
Úvod	23
O autorovi	30
Část A Superzáklady	31
Kapitola 1 Předehra.....	32
Kapitola 2 Prostředí JShell.....	45
Kapitola 3 Zadávání hodnot.....	60
Kapitola 4 Proměnné, výrazy a příkazy	74
Část B Začínáme programovat	91
Kapitola 5 Práce s objekty	92
Kapitola 6 Balíčky, knihovny, robot Karel	103
Kapitola 7 Definice metod.....	119
Kapitola 8 Opakování kódu – cykly	140
Kapitola 9 Rozhodování.....	152
Část C Objektově orientované programování	171
Kapitola 10 Základy definice třídy.....	172
Kapitola 11 Uspořádání kódu	188
Kapitola 12 Programátorská dokumentace.....	202
Kapitola 13 Rozhraní a interfejs	211
Kapitola 14 Dědění interfejsů	222
Kapitola 15 Návrhové vzory.....	236
Kapitola 16 Prohlubujeme znalosti	252
Kapitola 17 Lambda-výrazy, funkční interfejsy a generické typy a metody	274
Kapitola 18 Dědění implementace	292
Kapitola 19 Abstraktní třídy	312

Část D Standardní knihovna	329
Kapitola 20 Speciální datové typy.....	330
Kapitola 21 Výjimky.....	348
Kapitola 22 Kontejnery	362
Kapitola 23 Pole	377
Kapitola 24 Interní datové typy	391
Kapitola 25 Datovody.....	405
Kapitola 26 Čtení a ukládání dat	416
Kapitola 27 Vytváření aplikací.....	434
Část E Vývoj aplikace	445
Kapitola 28 Zásady objektové architektury.....	446
Kapitola 29 Návrh základní architektury	455
Kapitola 30 Testovací scénáře – dokončení API	471
Kapitola 31 První a druhá etapa: scénáře HAPPY a BASIC	482
Kapitola 32 Třetí etapa: Implementace navržené architektury	494
Kapitola 33 Čtvrtá etapa: spuštění hry	505
Kapitola 34 Pátá etapa: tvorba světa	514
Kapitola 35 Šestá etapa: definice standardních akcí	523
Kapitola 36 Sedmá a osmá etapa: děláme aplikaci robustní	530
Kapitola 37 Etapy 9–11: přidáváme nestandardní akce	542
Kapitola 38 Závěr: doplnění uživatelského rozhraní	555
Literatura.....	568
Rejstřík.....	571
Část F Přílohy	577
Příloha A Příprava spuštění JShell pod Windows.....	578
Příloha B Význam cizích slov	583
Příloha C Česko-americký slovník.....	585
Část G Seznamy	588
Seznam S1 Výpisy programů	589
Seznam S2 Seznam obrázků	595
Seznam S3 Seznam tabulek	597
Seznam S4 Seznam odboček – podšeděných bloků.....	598

Podrobný obsah

Poděkování	22
Úvod	23
Komu je kniha určena	23
Koncepce výkladu	24
Potřebné vybavení	25
Doprovodné programy	26
Použité typografické konvence	27
Odbočka – podšeděný blok	29
Zpětná vazba	29
O autorovi	30
Část A Superzáklady	31
Kapitola 1 Předehra	32
1.1 Trocha historie	32
1.1.1 Co je to program	33
1.1.2 Postupná změna priorit	33
1.1.3 Vývoj metodik programování	35
1.1.4 OOP a OFP	36
1.2 Překladače, interprety, platformy	36
1.2.1 Operační systém a platforma	37
1.2.2 Programovací jazyky	37
1.2.3 Způsoby zpracování programu	38
1.3 Java a její zvláštnosti	39
1.3.1 Java je jazyk i platforma	40
1.4 Vývojové nástroje	41
1.4.1 Základní vývojářská sada	41
1.4.2 IDE	41
1.5 Shrnutí a soubory pro opakování	44
Kapitola 2 Prostředí JShell	45
2.1 Prostředí JShell	45
2.1.1 Spuštění programu JShell	45
2.1.2 JShell v IDE	47
2.2 Úryvky (snippets)	48
2.2.1 Použití proměnných	49
2.2.2 Identifikace úryvků	49
2.2.3 Terminologie: výrazy, příkazy, deklarace, definice	50
2.2.4 Středník	50
2.2.5 Více objektů na řádku, zavlečené chyby	50
2.3 Příkazy prostředí JShell	52
2.3.1 Vyloučení úryvku: /drop	52
2.3.2 Přehled aktivních úryvků: /list	52

2.3.3 Přehled všech úryvků: /list -all	53
2.3.4 Uložení aktivních úryvků: /save <file>	54
2.3.5 Uložení všech zadaných úryvků: /save -all <file>	55
2.3.6 Uložení dosavadního průběhu seance: /save -history <file>	55
2.3.7 Načtení skriptu: /open <file>	55
2.3.8 Ukončení seance: /exit	56
2.3.9 Restart: /reset	56
2.3.10 Znovuzavedení: /reload -restore	57
2.3.11 Natavení startovního skriptu: /set -start <file>	57
2.3.12 Nápověda: /?	58
2.3.13 Spuštění editoru: /edit	58
2.3.14 Nastavení běhového prostředí: /env	59
2.4 Shrnutí a soubory pro opakování	59
Kapitola 3 Zadávání hodnot	60
3.1 Datové typy	60
3.1.1 Primitivní datové typy	61
3.1.2 Objektové datové typy	62
3.2 Komentáře	63
3.3 Zadávání celočíselných hodnot	64
3.4 Zadávání reálných hodnot	64
3.5 Zadávání znaků	66
3.6 Prázdný odkaz null	68
3.7 Hodnoty typu String	68
3.7.1 Textové bloky	71
3.8 Zadávání logických hodnot	72
3.9 Literály	73
3.10 Shrnutí a soubory pro opakování	73
Kapitola 4 Proměnné, výrazy a příkazy	74
4.1 Pravidla pro tvorbu identifikátorů	74
4.1.1 Používání znaku \$	75
4.2 Deklarace proměnných	76
4.3 Proměnné versus konstanty	77
4.4 Výrazy a jejich terminologie	78
4.5 Operace přiřazení =	79
4.5.1 Asociativita	80
4.6 Aritmetické operace	80
4.6.1 Sčítání	80
4.6.2 Odčítání	80
4.6.3 Násobení	81
4.6.4 Dělení	81
4.7 Přetypování	81
4.8 Volání metod	84
4.9 Závorky	84
4.10 Složené přiřazení	85
4.11 Inkrementační a dekrementační operátory	86
4.12 Porovnávací operátory	88
4.13 Příkazy versus výrazy	89
4.14 Shrnutí a soubory pro opakování	89

Část B Začínáme programovat 91

Kapitola 5 Práce s objekty 92

5.1 Nejprve trocha teorie.....	92
5.1.1 Principy OOP	92
5.1.2 Objekty	93
5.1.3 Atributy	93
5.1.4 Zprávy	94
5.1.5 Metody	94
5.1.6 Třídy a jejich instance	95
5.1.7 Interní datové typy	96
5.1.8 Třída jako datový typ	96
5.1.9 Třída jako objekt	96
5.1.10 Kontejner	97
5.1.11 Terminologie objektových datových typů	97
5.2 Práce s atributy	98
5.3 Volání metody	98
5.4 Konstruktory a tovární metody	100
5.4.1 Tovární metoda	100
5.4.2 Konstruktory	101
5.4.3 Správa paměti	101
5.5 Shrnutí a soubory pro opakování	102
Kapitola 6 Balíčky, knihovny, robot Karel	103
6.1 Velké programy a jejich problémy	103
6.2 Balíčky	104
6.2.1 Kořenový balíček a stromy balíčků	104
Datová struktura strom	105
6.3 Import objektových typů	105
6.3.1 Import všech typů z daného balíčku – hvězdičkový import	107
6.4 Knihovny a JAR-soubory	108
6.4.1 Začlenění knihovny robota Karla	108
6.4.2 Začlenění knihovny při startu	110
6.5 Robot Karel a jeho svět	110
Historie robota Karla	111
6.5.1 Vytvoření světa	111
6.5.2 Vytvoření robota	113
6.5.3 Akce	113
6.5.4 Testy	114
6.5.5 Zrychlování a skrývání	114
6.5.6 Reakce na zavření okna	116
6.6 Zakázaný balíček java	118
6.7 Shrnutí a soubory pro opakování	118
Kapitola 7 Definice metod	119
7.1 Start JShell na počátcích dalších kapitol	119
7.2 Zásada DRY	120
7.3 Definice a volání metody	120
7.3.1 Povinný středník	121
7.3.2 Příklad jednoduché metody pro robota	121
7.4 Blok příkazů	122
7.5 Metody s parametry	123
7.5.1 Parametry	123
7.5.2 Argumenty	123
7.5.3 Metody robota versus metody interaktivního prostředí	124
7.6 Metody s více parametry	125
Tisk na standardní výstup	126
7.7 Lokální proměnné a konstanty	126
7.7.1 Proměnné lokální v bloku	127
7.7.2 Demonstrace použití konstant	127

7.7.3 Dočasná existence proměnných v blocích	128
Zásobník návratových adres – ZNA	128
7.8 Přetěžování metod	129
7.9 Ještě jednou robot Karel a jeho svět	130
7.9.1 Třídy RobotWorld a RobotWindow	130
7.9.2 Přetížené tovární metody tříd RobotWorld a RobotWindow	131
7.9.3 Přetížené konstruktory třídy Karel	132
7.10 Metody vracející hodnotu	133
7.11 Přehled aktuálně definovaných metod v JShell	134
7.12 Logické výrazy	135
7.12.1 Příklad	136
7.13 Předávání hodnotou a odkazem	137
7.14 Shrnutí a soubory pro opakování	139
Kapitola 8 Opakování kódu – cykly	140
8.1 Cykly	140
8.2 Cyklus s počáteční podmínkou – cyklus while	141
8.3 Cyklus s koncovou podmínkou – cyklus do...while	143
8.4 Cyklus s parametrem – cyklus for	145
8.4.1 Příklad	146
8.5 Nekonečný cyklus	149
8.6 Cyklus s podmínkou uprostřed	149
8.7 Vnořování cyklů	150
8.8 Shrnutí a soubory pro opakování	151
Kapitola 9 Rozhodování	152
9.1 Jednoduchý podmíněný příkaz	153
9.2 Úplný podmíněný příkaz	154
9.3 Podmíněný výraz	155
9.4 Složený podmíněný příkaz	156
9.5 Přepínač – příkaz/výraz switch	158
9.5.1 Pravidla	160
9.5.2 Přepínací výraz – výraz switch	162
9.5.3 Využití klasické verze příkazu switch	163
9.6 Cyklus s podmínkou uprostřed – příkaz break	164
9.7 Příkaz continue	166
9.8 Rekurze	167
9.9 Shrnutí a soubory pro opakování	169

Část C Objektově orientované programování 171

Kapitola 10 Základy definice třídy	172
10.1 Vytváříme vlastní třídu	172
Bílé znaky a uspořádání programu	173
10.2 Viditelnost tříd a jejich členů: public, private	174
10.3 Výměna knihoven – knihovna tvarů	174
10.4 Diagramy tříd	176
10.5 Konstruktory	177
10.5.1 Podrobnosti o konstruktorech	177
10.6 Atributy	179
10.6.1 Definice atributů	179
10.6.2 Modifikátor final	179
10.7 Konstruktory a parametr this	180
10.7.1 Kvalifikace parametrem this	181

10.8	Magické hodnoty	182
10.9	Modifikátor <code>static</code>	183
10.9.1	Definice třídy počítající své instance	183
10.10	Zděděné metody, metoda <code>toString()</code>	184
10.11	Výsledná definice a test.....	184
10.11.1	Upravená definice	185
10.11.2	Test upravené definice	185
10.12	Shrnutí a soubory pro opakování	187
Kapitola 11	Uspořádání kódu	188
11.1	Uspořádání programů ve složce <code>77_JAVA</code>	188
11.1.1	Vývojové prostředí	189
11.1.2	Problémy s překladem	189
11.1.3	Knihovna <code>Java77_NZ2</code>	189
11.1.4	Úprava prostředí pro <code>JShell</code>	190
11.2	Konvence pro názvy balíčků v Javě	190
11.3	Samostatně definované třídy.....	190
11.4	Příkazy <code>package</code> a <code>import</code>	192
11.5	Zapouzdření a skrývání implementace.....	193
11.6	Entity programu	194
11.7	Rozhraní versus implementace.....	194
11.7.1	Signatura versus kontrakt	195
11.7.2	API.....	195
11.8	Atributy versus vlastnosti; přístupové metody	196
11.8.1	Možné kombinace a výhody.....	196
11.8.2	Pravidla pro názvy.....	197
11.9	Uspořádání jednotlivých prvků v těle třídy	197
11.9.1	Motivace pro zavedení některých zásad.....	198
11.10	Prázdná standardní třída.....	199
11.11	Shrnutí a soubory pro opakování	201
Kapitola 12	Programátorská dokumentace.....	202
12.1	Komentáře a dokumentace.....	202
12.1.1	Proč psát srozumitelné programy	202
12.1.2	Odsazování	204
12.1.3	Dokumentační komentáře	204
12.2	Pomocné značky pro tvorbu dokumentace	205
12.3	Dokumentace balíčku.....	206
12.4	Ukázka dokumentované třídy	206
12.5	Zobrazení dokumentace.....	208
12.5.1	Obsah a uspořádání dokumentace.....	208
12.6	Zakomentování a odkomentování části programu.....	210
12.7	Shrnutí a soubory pro opakování	210
Kapitola 13	Rozhraní a interfejs	211
13.1	Motivace	211
13.2	Rozhraní versus interfejs versus <code>interface</code>	212
13.2.1	Interfejs a jeho instance	212
13.3	Použití v programu	213
13.3.1	Knihovní balíček <code>shapes77.canvas_2</code>	214
13.4	Použití interfejsu na příkladu.....	215
13.4.1	Počáteční úvahy.....	215
13.5	Definice interfejsu.....	217
13.6	Implementace interfejsu třídou	218
13.6.1	Anotace <code>@Override</code>	218
13.6.2	Ověření funkcionality	220

13.7 Shrnutí a soubory pro opakování	221
Kapitola 14 Dědění interfejsů	222
14.1 Problém více rolí	222
14.2 Současná implementace více interfejsů	223
14.2.1 Realizace v kódu	224
14.3 Dědění interfejsů	225
14.3.1 Trocha teorie o dědění	225
14.3.2 Dědění a přetypování	226
14.3.3 Aplikace dědění interfejsů – balíček canvas_4	227
14.4 Deklarace rodičů interfejsu v kódu	229
14.5 Vlastnosti interfejsů na příkladu Multishape	229
14.5.1 Podrobnosti o třídě Multishape	230
14.5.2 Mnohotvar se skládá z kopií	230
14.6 Příklad: definice vozidla	230
14.6.1 Kontrakt	231
14.6.2 Test vytvořeného vozidla	234
14.6.3 Problém plátna	235
14.7 Shrnutí a soubory pro opakování	235
Kapitola 15 Návrhové vzory	236
15.1 Úvod do návrhových vzorů	236
15.2 Přehled vzorů, s nimiž jsme se již setkali	238
15.2.1 Knihovni třída (Utility class)	238
15.2.2 Statická tovární metoda (Static factory method)	238
15.2.3 Jedináček (Singleton)	239
15.2.4 Náhradník (Substitute)	239
15.2.5 Výčtový typ (Enumerated type)	239
15.2.6 Multiton, Originál	240
15.2.7 Služebník (Servant)	240
15.2.8 Prázdný objekt (Null Object)	241
15.2.9 Prototyp (Prototype)	241
15.3 Teoretické pozadí nové koncepce kreslení	242
15.3.1 Návrhový vzor Prostředník (Mediator)	242
15.3.2 Inverze závislostí	243
15.3.3 Návrhový vzor Pozorovatel (Observer), hollywoodský princip	245
15.4 Správce plátna – CanvasManager	246
15.4.1 Balíček canvasmanager	247
15.4.2 Import klíčových tříd knihovny	247
15.5 Nová verze třídy Robot	249
15.6 Shrnutí a soubory pro opakování	251
Kapitola 16 Prohlubujeme znalosti	252
16.1 Statický import	252
16.2 Další členy interfejsů	253
16.2.1 Statické konstanty	253
16.2.2 Statické metody	255
16.2.3 Implicitní metody	255
16.2.4 Soukromé metody	255
16.3 Návrhový vzor Adaptér (Adapter)	256
16.4 Návrhový vzor Přepřavka – třídy typu record	257
16.4.1 Třídy typu záznam (record)	257
16.4.2 Záznamy v používané knihovně	258
16.5 Návrhový vzor Šablonová metoda	260
16.5.1 Příklad	261
16.6 Návrhový vzor Abstraktní továrna	262
16.6.1 Motivace	262

16.6.2 Návrhový vzor Tovární metoda.....	263
16.6.3 Co je abstraktní továrna	263
16.6.4 Použití.....	263
16.6.5 Tovární třída převádí konstruktory a statické členy na instanční.....	263
16.7 Podrobnosti o konstruktorech	265
16.7.1 Dvě části těla konstrukturu instancí – inicializační bloky.....	266
16.7.2 Inicializace konstant	268
16.7.3 Konstruktor třídy	268
16.7.4 Konstanty vyhodnotitelné v době překladu	271
16.8 Shrnutí a soubory pro opakování	273
Kapitola 17 Lambda-výrazy, funkční interfejsy a generické typy a metody	274
17.1 Nezávislé opakování zadané akce	274
17.1.1 Možná řešení.....	275
17.2 Syntaxe lambda-výrazů.....	275
17.2.1 Aplikace na světlo	276
17.3 Lambda-výrazy a lokální proměnné.....	278
17.4 Lambda-výrazy zastupující metody	279
17.5 Funkční interfejsy	280
17.6 Generické datové typy a metody.....	281
17.6.1 Motivace	281
17.6.2 Syntaxe zadávání a používání generických typů	281
17.6.3 Specifika generických typů Javy	282
17.6.4 Omezení typových parametrů	282
17.6.5 Příklad: Interval	282
17.6.6 Typové parametry s více předky.....	284
17.6.7 Potomci a předci generických typů	284
17.6.8 Generické metody	285
17.7 Funkční interfejsy – pokračování.....	285
17.7.1 Demonstrace použití	287
17.8 Lambda-výrazy nelze přetypovat na Object přímo	289
17.9 Shrnutí a soubory pro opakování	291
Kapitola 18 Dědění implementace	292
18.1 Tři druhy dědění.....	292
18.1.1 Přirozené (nativní) dědění	292
18.1.2 Dědění rozhraní	293
18.1.3 Dědění implementace	293
18.1.4 LSP – Liskov Substitution Principle.....	294
18.1.5 Shrnutí	294
18.2 Základy dědění tříd	294
18.2.1 Univerzální (pra)rodič Object.....	295
18.3 Co dědíme od třídy Object.....	295
18.3.1 Class-objekt	296
18.3.2 Úplný název tříd v JShell	297
18.3.3 Přehled veřejných členů zděděných od třídy Object	297
18.4 Definice třídy s předkem – Square	298
18.4.1 Rodičovský podobjekt.....	298
18.4.2 Volání rodičovského konstrukturu.....	299
18.5 Přebíjení metod.....	301
18.5.1 Implementace přebíjení	303
18.6 Skrytá nebezpečí: úprava třídy Square	305
18.7 Modifikátor final v procesu dědění	307
18.7.1 Virtuální versus konečné metody	307
18.7.2 Obyčejné versus konečné třídy	308
18.8 Zakrývání statických metod.....	308
18.8.1 Metody interfejsů se nezakrývají	309

18.9 Jediný implementační předeek	309
18.10 Přístupová práva.....	310
18.11 Akceptovatelné modifikace signatur potomků	310
18.12 Shrnutí a soubory pro opakování	311
Kapitola 19 Abstraktní třídy.....	312
19.1 Abstraktní třídy a jejich role v dědické hierarchii	312
19.1.1 Definice abstraktních tříd a metod	314
19.1.2 Experimenty s abstraktní třídou	314
19.2 Účel abstraktních tříd	316
19.3 Zavedení abstraktních tříd do projektu	316
19.4 Návrhový vzor Stav	317
19.5 Aplikace na příklad s vozidlem.....	318
19.5.1 Hlavní třída vozidla – třída Robot4_4	319
19.5.2 Stavově nezávislé metody	322
19.5.3 Stavově závislé metody	322
19.5.4 Společný rodič jednosměrných tříd – třída ARobot1_4.....	322
19.5.5 Jednostavové (jednosměrné) třídy	324
19.5.6 Test.....	326
19.6 Shrnutí a soubory pro opakování	327

Část D Standardní knihovna 329

Kapitola 20 Speciální datové typy.....	330
20.1 Třída Object	330
20.2 Objekty reprezentující hodnotu (ORV) a objekty reprezentující entitu (ORE).....	331
20.2.1 Metody equals (Object)	332
20.2.2 Metoda hashCode ()	333
20.2.3 Proměnné a neměnné ORV	333
20.3 Primitivní a obalové typy	335
20.4 Třída String	336
20.4.1 Možné problémy při práci s některými znaky	336
20.4.2 Interfejs java.lang.CharSequence	336
Problémy s kódováním znaků	337
20.4.3 Třídy StringBuilder a StringBuffer	338
20.5 Výčtové typy – třídy typu enum.....	339
20.5.1 Složitější příklad: Direction	341
20.6 Třída Throwable	342
20.7 Třída Thread	342
20.8 Třída java.util.Optional<T> & spol.....	343
20.8.1 Motivace.....	343
20.8.2 Alternativní řešení.....	343
20.9 Třída java.util.Random	344
20.10 Interfejs java.lang.Comparable<T>	345
20.11 Interfejs java.util.Comparator<T>	346
20.12 Shrnutí a soubory pro opakování	347
Kapitola 21 Výjimky.....	348
21.1 Co to jsou výjimky	348
21.2 Nejdůležitější výjimky.....	349
21.3 Vyhození výjimky.....	350
21.3.1 Reakce systému na vyhození výjimky	351
21.4 Výjimky a nedosažitelný kód	353
21.5 Hierarchie dědění výjimek	353

21.6 Zachycení vyhozené výjimky.....	355
21.7 Společný úklid – blok finally	356
21.8 Definice vlastních výjimek	358
21.9 Kontrolované výjimky	359
21.10 Převedení kontrolované výjimky na nekontrolovanou	360
21.11 Shrnutí a soubory pro opakování	361
Kapitola 22 Kontejnery	362
22.1 Co je to kontejner v Javě	362
22.2 Kategorizace kontejnerů	363
22.2.2 Kontejnery objektů	363
22.3 Knihovna kolekcí	364
22.4 Deklarujte typy co nejobecněji	366
22.5 Collection<E> – kolekce	367
22.6 Dvojtečkový cyklus for (:)	368
22.7 Množiny – Set<E>	369
22.8 Seznamy – List<E>	371
22.8.1 Pořadí prvků.....	371
22.8.2 Příklad: seřazení prvků v seznamu.....	373
22.9 Slovníky a mapy – Map<K,V>	374
22.9.1 Pohledy.....	376
22.10 Shrnutí a soubory pro opakování	376
Kapitola 23 Pole.....	377
23.1 Představení	377
23.1.1 Deklarace polí	378
23.2 Atributy a metody polí.....	378
23.3 Pole jako kontejner	378
23.3.1 Pole odkazů na objekty.....	379
23.3.2 Pole hodnot primitivních typů.....	379
23.3.3 Hlídaní mezi polí	381
23.3.4 Inicializace polí v deklaraci	382
23.3.5 Inicializace vytvářeného pole.....	383
23.3.6 Neinicializovaná pole objektových typů	384
23.4 Vícerozměrná pole.....	385
23.4.1 Obdélníková pole	385
23.4.2 Neobdélníková pole	386
23.4.3 Inicializace vícerozměrného pole	387
23.5 Metody s proměnným počtem argumentů	387
23.6 Pole, kolekce a moderní programování.....	388
23.7 Závěrečný příklad	388
23.8 Shrnutí a soubory pro opakování	390
Kapitola 24 Interní datové typy	391
24.1 Přehled.....	391
24.1.1 Terminologie.....	391
24.1.2 Společné charakteristiky	392
24.1.3 Použití.....	392
24.1.4 Jedna hladina soukromí	393
24.2 Globální interní (členské) datové typy.....	393
24.3 Vnořené datové typy	394
24.4 Vnitřní třídy.....	395
24.5 Lokální třídy	395
24.5.1 Pojmenované lokální třídy	396
24.5.2 Anonymní třídy	396
24.6 Návrhový vzor Iterátor.....	397

24.6.1	Interfejsy <code>java.util.Iterator<E></code> a <code>java.lang.Iterable<E></code>	398
24.6.2	Vnější (sekvencní) a vnitřní (dávkové) iterátory	399
24.6.3	Iterátory a dvojtečkový cyklus <code>for(:)</code>	400
24.7	Definice vlastní iterovatelné třídy	401
24.7.1	Generátor náhodných stringů	402
24.7.2	Použití v cyklu <code>for(:)</code>	403
24.7.3	Použití interního iterátoru <code>forEach()</code>	404
24.8	Shrnutí a soubory pro opakování	404
Kapitola 25	Datovody	405
25.1	Analogie	405
25.2	Druhy operací	406
25.3	Princip práce datovodu	406
25.4	Specifické vlastnosti	407
25.5	Datové typy související s datovody	408
25.6	Vytváření datovodů	408
25.6.1	Kolekce	408
25.6.2	Pole	409
25.6.3	Interfejs <code>java.util.stream.Stream</code>	409
25.7	Konceptní příklad	409
25.8	Operace s daty v datovodu	410
25.8.1	Koncové operace	410
25.8.2	Průběžné operace	411
25.8.3	Částečně průběžné operace	412
25.9	Kolektory	412
25.9.1	Příklad	413
25.10	Shrnutí a soubory pro opakování	415
Kapitola 26	Čtení a ukládání dat	416
26.1	Koncepce čtení a ukládání dat	416
26.2	Soubory: bleskové opakování	417
26.2.1	Soubor, souborový systém, cesta	417
26.2.2	Relativní, absolutní a kanonická cesta	418
26.2.3	Substituované disky ve Windows	418
26.3	Třída <code>java.io.File</code>	419
26.3.1	Instanční metody	419
26.4	Návrhový vzor Dekorátor	422
26.4.1	Motivace	422
26.4.2	Princip funkce	423
26.5	Rozdělení datových proudů	424
26.6	Zápis textů	425
26.6.1	Splachování a zavírání proudů	426
26.6.2	Přidávání dat na konec existujícího souboru	427
26.6.3	Příklad	427
26.7	Čtení textů	429
26.7.1	Příklad	430
26.8	Třída <code>java.util.Scanner</code>	431
26.9	Shrnutí a soubory pro opakování	433
Kapitola 27	Vytváření aplikací	434
27.1	Superjednoduchá demonstrační aplikace	434
27.2	Základní pravidla pro větší aplikace	437
27.3	Hlavní třída aplikace	437
27.3.1	Argumenty příkazového řádku	438
27.3.2	Definice třídy <code>Main</code>	438
27.3.3	Definice třídy <code>GuessIO</code>	440

27.4 JAR-soubor	441
Prohlížení obsahu JAR-souborů.....	441
27.4.1 Soubor MANIFEST.MF	442
27.4.2 Vytvoření JAR-souboru s aplikací	442
27.5 Spuštění aplikace	443
27.6 Shrnutí a soubory pro opakování	444

Část E Vývoj aplikace 445

Kapitola 28 Zásady objektové architektury.....	446
28.1 Předmluva	446
28.2 Architektura.....	447
28.3 Hlavní zásady návrhu	447
28.3.1 Přípravenost na změny	448
28.3.2 CRIDP – maximální přehlednost.....	448
28.3.3 KISS – maximální jednoduchost.....	448
28.3.4 YAGNI – žádné zbytečnosti.....	448
28.3.5 SoC – jediný zodpovědný.....	449
28.3.6 SRP – jediná zodpovědnost	449
28.4 Antivzory	449
28.5 Návrh programu	450
28.6 Druhy vytvářených sólo-objektů	451
28.7 Dva způsoby návrhu.....	452
28.7.1 Návrh shora dolů.....	452
28.7.2 Návrh zdola nahoru	452
28.7.3 Porovnání.....	453
28.8 UML – diagram tříd	454
28.9 Důležitost čitelnosti programu	454
28.10 Shrnutí a soubory pro opakování	454
Kapitola 29 Návrh základní architektury.....	455
29.1 Proč právě textová konverzační hra	455
29.1.1 Proč hra vyvíjená pod frameworkem.....	456
29.2 Organizace balíčků a záznamy průběhu vývoje	456
29.2.1 Umístění studentských úloh	457
29.2.2 Umístění tříd frameworku	457
29.3. Koncepce vyvíjené aplikace	457
Co to je h-objekt	458
29.3.1 Nestandardní akce	459
29.4 Zadání	459
29.4.1 Vyjmenované požadavky	460
29.5 Účastníci – objekty vystupující ve hře.....	461
29.6 Správci skupin objektů	465
29.6.1 Správci v naší aplikaci	465
29.7 Specifikace jednotlivých účastníků	466
29.8 Společné API.....	466
29.9 Schopnosti jednotlivých účastníků	467
29.9.1 IGame	467
29.9.2 IWorld	468
29.9.3 INamed.....	468
29.9.4 IItemContainer	469
29.9.5 IPlace.....	469
29.9.6 IItem	469
29.9.7 IBag	469
29.9.8 IAction.....	470

29.10 Shrnutí a soubory pro opakování	470
Kapitola 30 Testovací scénáře – dokončení API	471
30.1 Jak testovat	471
30.1.1 Programování řízené testy	471
30.1.2 Jednotkové, integrační a regresní testy	472
30.1.3 Možnosti testování naší hry	473
30.2 Scénáře	473
30.3 Kroky scénáře – třída ScenarioStep	474
30.4 Výčtový typ TypeOfStep	476
30.5 Výčtový typ TypeOfScenario	476
30.6 Třída Scenario	477
30.7 Postup vývoje a požadavky na scénáře	477
30.7.1 Jednotlivé etapy vývoje	478
30.8 Doplnění API	479
30.8.1 Interfejs IAuthor	479
30.8.2 Interfejs IPortal	479
30.8.3 Třída BasicActions	480
30.8.4 Balíček game77.testers	481
30.8.5 Interfejs IGame	481
30.9 Shrnutí a soubory pro opakování	481
Kapitola 31 První a druhá etapa: scénáře HAPPY a BASIC	482
31.1 Balíček etapy	482
31.2 Návrh portálu	482
31.3 Správce scénářů – ScenarioManager	485
31.4 Test na hladině Level.HAPPY	488
31.5 Druhá etapa – Scénář BASIC	490
31.5.1 Balíček etapy	490
31.5.2 Návrh portálu	490
31.5.3 Úpravy třídy game77.ck1b_duplet.ScenarioManager	490
31.5.4 Test na hladině Level.DUPLET	492
31.6 Shrnutí a soubory pro opakování	493
Kapitola 32 Třetí etapa: Implementace navržené architektury	494
32.1 Tvorba kostry vyhovující API	494
32.2 Návrh tříd v balíčku ck1c_architecture	495
32.2.1 Třída Portal	495
32.2.2 Třída ANamed	495
32.2.3 Třída Action	496
32.2.4 Třída Item	497
32.2.5 Třída AItemContainer	497
32.2.6 Třída Bag	498
32.2.7 Třída Place	499
32.2.8 Třída World	499
32.2.9 Třída Game	501
32.3 Test na hladině Level.ARCHITECTURE	503
32.4 Shrnutí a soubory pro opakování	504
Kapitola 33 Čtvrtá etapa: spuštění hry	505
33.1 Návrh tříd v balíčku ck1d_start	505
33.2 Tři druhy objektů	505
33.3 Delegování zodpovědnosti	506
33.4 Statické metody třídy Action	507
33.4.1 Metody isActive() a stop()	507

33.4.2	Statická metoda <code>executeCommand(String)</code>	507
33.4.3	Definice má být krátká	508
33.4.4	Pomocné statické metody	509
33.5	Úpravy třídy <code>SceneManager</code>	509
33.5.1	Problematika statických konstant	510
33.5.2	Vlastní úprava	510
33.6	Spuštění testu na hladině <code>Level.START</code>	512
33.7	Shrnutí a soubory pro opakování	513
Kapitola 34	Pátá etapa: tvorba světa	514
34.1	Balíček <code>ck1e_world</code> a třída <code>Portal</code>	514
34.2	Vytváříme prostory	514
34.2.1	Inicializace prostoru	515
34.3	Modifikace třídy <code>ItemContainer</code>	516
34.4	Další úpravy třídy <code>SceneManager</code>	517
34.4.1	Simulace chodu hry podle scénáře	519
34.5	Modifikace třídy <code>World</code>	519
34.5.1	Hledání chyby	520
34.6	Test na hladině <code>Level.WORLD</code>	521
34.7	Shrnutí a soubory pro opakování	522
Kapitola 35	Šestá etapa: definice standardních akcí	523
35.1	Balíček <code>ck1f_basic</code> a třída <code>Portal</code>	523
35.2	Definice akcí	523
35.3	Funkce <code>executeStandardCommand(String)</code>	524
35.4	Definice výkonného kódu jednotlivých akcí	525
35.4.1	Akce <code>Jdi</code> a metoda <code>move(String[])</code>	526
35.4.2	Akce <code>Vezmi</code> a metoda <code>take(String[])</code>	527
35.4.3	Akce <code>Polož</code> a metoda <code>put(String[])</code>	528
35.4.4	Akce <code>?</code> a metoda <code>help(String[])</code>	528
35.4.5	Kontrolní test	529
35.5	Shrnutí a soubory pro opakování	529
Kapitola 36	Sedmá a osmá etapa: děláme aplikaci robustní	530
36.1	Nesplněné body zadání	530
36.2	Balíček pro hladinu <code>Level.TRIPLET</code>	531
36.2.1	Proč samostatná etapa	531
36.3	Chybový scénář	531
36.3.1	Co vše se má zkontrolovat	532
36.3.2	Reakce na pokus o nekorektní spuštění	533
36.3.3	Vlastní definice chybového scénáře	533
36.3.4	Test chybového scénáře	533
36.4	Přechod na hladinu <code>Level.MISTAKES</code>	534
36.4.1	Chybné spuštění hry	534
36.5	Problémy s argumenty standardních akcí	535
36.5.1	Nezadané argumenty	535
36.5.2	Finalizace změny prostoru	536
36.5.3	Problémy se zvedáním h-objektu	536
36.6	Změny v definici h-objektů a práce s nimi	537
36.6.1	Nová definice třídy <code>Item</code>	537
36.6.2	Úprava definice prostorů	538
36.6.3	Úprava <code>batohu</code>	539
36.6.4	Dokončení metody <code>take(String[])</code>	539
36.6.5	Dotažení metody <code>put(String[])</code>	541
36.7	Shrnutí a soubory pro opakování	541

Kapitola 37 Etapy 9–11: přidáváme nestandardní akce	542
37.1 Předehra	542
37.1.1 Plánovaný postup –etapy dalšího vývoje.....	542
37.2 RUNNING: Rozchození scénáře HAPPY	543
37.2.1 Úprava scénáře HAPPY.....	543
37.2.2 Úprava konstrukturu akcí	543
37.2.3 Výchozí podoba definic nestandardních akcí.....	545
37.3 QUADRUPLLET: Příprava testu robustnosti	545
37.3.1 Nutné podmínky pro korektní zadání nestandardních akcí.....	546
37.3.2 Zanesení nutných podmínek	546
37.3.3 Zanesení nutných podmínek do scénářů	547
37.3.4 Průběžný test	550
37.3.5 Jaká chybná zadání se budou testovat	550
37.3.6 Přidání scénáře MISTAKE_NS	550
37.4 WHOLE: Rozchození výsledného kódu hry	550
37.4.1 Definice metod conditions() a tests() ve třídě Game.....	552
37.4.2 Definice atributů CONDITIONS a TESTS ve třídě Action	552
37.4.3 Definice kódu nestandardních akcí	553
37.5 Shrnutí a soubory pro opakování	554
Kapitola 38 Závěr: doplnění uživatelského rozhraní	555
38.1 Hlavní třída aplikace	555
38.2 Jednoduché textové uživatelské rozhraní.....	555
38.2.1 Možnost opakovaného spouštění	557
38.2.2 Přidání informací o aktuálním stavu	558
38.3 Opět trocha teorie	558
38.3.1 Vzor Model-View-Controller (MVC)	558
38.3.2 Kompaktnější varianta: M(V+C).....	559
38.4 Volitelné uživatelské rozhraní	559
38.4.1 Interfejs IUI.....	559
38.4.2 Definice třídy Main2.....	560
38.4.3 Úprava metod run(...) a multirun(...)	561
38.4.4 Definice třídy Console	563
38.5 Vytvoření primitivního GUI	563
38.5.1 Modalita dialogových oken.....	563
38.5.2 Třída javax.swing.JOptionPane.....	564
Návrhový vzor Fasáda.....	564
38.5.3 Třída PrimitiveGUI.....	565
38.6 Několik dalších námětů.....	566
38.7 Shrnutí a soubory pro opakování	567
Literatura.....	568
Rejstřík.....	571

Poděkování

Vím, že se v českých knížkách většinou neděkuje, ale tvorba knih je spojena s takovými obětmi řady lidí z mého blízkého i vzdálenějšího okolí, že bych měl velkou újmu na duši, kdybych tak neučinil.

Chtěl bych především nesmírně poděkovat své ženě Jarušce, která byla po celou dobu mojí největší oporou a jejíž nekonečná trpělivost a vstřícnost mi pomohla dokončit knihu v termínu, který se příliš nelišil od toho, jež jsme původně s nakladatelem dohodli, a ne až někdy za rok po něm. Stále marně přemýšlím, kde má schovanou svatozář.

Původně jsem se domníval, že s dalším vydáním nebude moc práce. Šeredně jsem se zmýlil, protože veškeré úpravy, jež jsem do knihy zanesl, jsou vykoupeny hodinami studia a experimentování, které rodina s neuvěřitelnou trpělivostí snášela.

Na vylepšování textu nového vydání se podílela řada dalších lidí. Mezi nimi musím poděkovat především těm, kteří si dali práci s odhalováním případných chyb ve vznikajícím rukopisu. Mezi nimi pak především Ludkovi Šťastnému, který po celou přípravu rukopis pročetl a odhaloval v něm pasáže, jež by si zasloužily vylepšit.

Neméně velkou zásluhu na současné podobě má i Marek Chadim a Jakub Kolář, kteří se na text dívali očima vedoucích programátorských týmů a doporučili řadu užitečných vylepšení. Řadou podnětných myšlenek přispěl Michal Palas, jenž pracuje v oblasti analýzy a zpracování dat a přispěl tak řadou poznatků z praxe.

V neposlední řadě patří můj dík redakci, především redaktoru Petru Somogyimu, který trpělivě snášel mé neustálé modifikace již zkorigovaného textu, a Radku Matulíkovi, který mne k napsání jednotlivých knih z posledních let vyhecoval a byl pak ochoten pár týdnů počkat, když se mi nepodařilo přesně dodržet původně dohodnutý termín.

Úvod

Otevíráte nové, výrazně přepracované a aktualizované vydání knížky, která vás chce naučit programovat moderním, objektově orientovaným stylem, jímž se v dnešní době vyvíjí drtivá většina klíčových aplikací. Tento styl dnes dominuje vývoji naprosté většiny klíčových aplikací, ať už se jedná o mobilní aplikace, podnikové informační systémy nebo vývoj webových služeb.

Kniha je výsledkem mnohaletého experimentování s tím, jak co nejlépe učit *Javu* a *soudobé* programování. Vyzkoušel jsem výuku programování a *Javy* snad na všech typech zařízení. Učil jsem v zájmových kroužcích na základní škole, na střední škole, na univerzitách i v rozšiřujících kurzech pro profesionální programátory.

Při rozhovorech s vedoucími programátorských týmů zjišťuji, že střední i vysoké školy opouští řada programátorů, kteří sice programují v nějakém „moderním“ jazyce, ale neumějí programovat moderně. Absolvované kurzy je sice naučily navržený program zakódovat, ale nenaučily je netriviální program samostatně navrhnout. Dokážou používat nejrůznější frameworky, ale používají je většinou mechanicky, bez pochopení základních principů, na jejichž základě jsou navrženy.

Ve svých učebnicích se to pokouším napravit. Reakce čtenářů prozatím naznačují, že se mi to snad alespoň částečně daří.

Komu je kniha určena

Kniha je určena čtenářům, kteří se chtějí programování věnovat seriózně a dlouhodobě. Není nutné mít předchozí programátorské zkušenosti. Kniha začíná od naprostých základů, ale současně předpokládá, že čtenář se chce opravdu naučit programovat a pochopit souvislosti, nikoli jen „opisovat kód“. Pokud je vaším cílem stát se profesionálním programátorem, kniha vás na tuto cestu připraví.

Když jsem dával rukopis několika vedoucími programátorských týmů k lektorování, abych se ujistil, že kniha dobře odráží potřeby a zvyky praxe, tak mne potěšili zprávou, že ve svých týmech použijí tuto učebnici jako povinnou literaturu pro nováčky. Ti se totiž ve školách a programátorských kurzech naučí kódovat v několika jazycích a používat řadu frameworků, ale neosvojí si dostatečně klíčové dovednosti potřebné pro práci v týmu na velkém projektu. A v tom by jim mohla tato učebnice pomoci.

Koncepce výkladu

Musím vás upozornit na to, že kniha, kterou držíte v ruce, se od běžných učebnic poněkud liší. Ostatní učebnice jsou většinou především učebnicemi nějakého programovacího jazyka. Jejich autoři se proto ve svém výkladu soustředí hlavně na výklad vlastností popisovaného jazyka a jeho knihoven. Bohužel se v nich ale nedozvíte skoro nic o tom, jak při návrhu programů přemýšlet, aby vás nezaskočily náhlé změny zadání, kterými je současné programování pověstné. Takovéto učebnice proto nevyhovávají návrháře, kteří by uměli program navrhnout, ale pouze kodéry, kteří umějí zanalyzované zadání zakódovat.

Vypadá to, jako kdyby autoři předpokládali, že se při čtení jejich knihy naučíte programovat nějak sami od sebe – obdobně, jako se to museli naučit oni. Zkušenosti s programátory, kteří navštěvují mé kurzy ve firmách či na univerzitě, však ukazují, že tohoto výsledku bývá dosaženo jen zřídka a platí to, co jsem říkal před chvílí: většina z nich zná poměrně dobře konstrukce nějakého objektově orientovaného programovacího jazyka a základy často používaných frameworků, ale bohužel, skoro nikdo z nich v něm neumí objektově programovat,¹ neumí přepnout na objektový způsob uvažování. Nerozlišují objektové programování a objektové kódování, což je způsob práce, kdy sice používáte objektové konstrukce, ale ve skutečnosti vyvíjíte program podle klasického paradigmatu.



Tady si neodpustím vzpomínku na jednoho studenta, který na konci semestru vysvětloval, že už profesionálně programuje, takže si myslel, že kurz hravě zvládne. Když jsme na začátku semestru probírali naprosté základy, řekl si, že to je pouhé hraní a že si počká, až začneme doopravdy programovat, a pak chybějící body rychle dožene. Když se k nám však v polovině semestru připojil, zjistil, že řadu konstrukcí, které běžně používáme, vůbec nechápe. Nezbylo mi, než mu zopakovat, že mezi prostým používáním objektových konstrukcí a návrhem objektových programů je velký rozdíl a že řada účastníků přichází do mých kurzů právě proto, aby se tento jiný způsob programátorského myšlení naučila.

Pro jistotu proto varuji: toto není učebnice jazyka *Java*, toto je učebnice objektového programování v *Javě*. Zpočátku vás sice budu učit, jak program zapsat, ale poté přejdeme k výkladu toho, jak jej navrhnout. Cílem není vychovat z čtenářů kodéry v *Javě*, ale připravit je na to, aby z nich mohli vyrůst schopní architekti, jejichž schopnosti současná umělá inteligence zatím nenahradí.

¹ Výzkum z přelomu století ukázal, že pouze 10 % programů psaných v objektově orientovaných jazycích je navrženo opravdu objektově. (Goddard, D. 1994. Is it really object oriented? *Data Based Advis.* 12, 12 (Dec. 1994), 120-123.) Od té doby se situace trochu zlepšila, nicméně na svých školeních stále pozoruji, že v mnoha softwarových firmách převažují strukturovaně navrhované programy napsané v objektových jazycích.

Seznamování s knihovnami a frameworky proto omezují na nezbytné minimum, protože jejich používání je poměrně podrobně a přehledně popsáno v dokumentaci. V ušetřeném čase (a stránkách) se vás pokusím naučit efektivně navrhovat a vytvářet spolehlivé a snadno udržitelné programy. Jinými slovy: chci vás naučit dovednostem, které budete používat, ať už budete programovat v jakémkoliv objektově orientovaném jazyku. Jazyky přicházejí a odcházejí. Základní programátorské techniky a způsob myšlení však žijí daleko déle než jazyky, se kterými byly zavedeny.

Díky tomu, že se místo na knihovny soustředím spíše na vlastní programování, vznikl v knize prostor pro výklad řady programátorských zásad a technik, o nichž se klasické učebnice vůbec nezmiňují, a to často ani učebnice pro zkušené programátory. Tyto zásady a techniky se většinou přednášejí až v nadstavbových kurzech, které učí vytvářet programy tak, aby s vámi mohly růst a aby vás nezaskočily rychle se měnící požadavky zákazníků (a připravte se na to, že tyto měnící se požadavky vás potkají i v případě, kdy jste zákazníkem sami sobě). Přitom vůbec nejde o techniky složité, které by začátečník nemohl pochopit. Ostatní učebnice je pomíjejí pouze proto, že nesusouvisí přímo se syntaxí programovacího jazyka, ale týkají se obecného programování.

Tato učebnice je naopak vysvětluje od samého počátku výkladu, protože vím, že byste si je měli osvojit co nejdříve. Ti, kteří se nejprve učí kódovat, a teprve následně se dozvídají, jak lze návrh programu zefektivnit, bývají příliš v zajetí svých zkušeností s kódováním a při návrhu programů se pak zbytečně silně soustředí na některé nepodstatné detaily.

Jazyk identifikátorů

Podle všeobecných zvyklostí budou v programech používány názvy vycházející z angličtiny. Pouze u AHA-příkladů, tj. u příkladů sloužících především k tomu, aby si čtenář řekl: „AHA, takto to funguje“, budou pro větší názornost použity názvy (identifikátory) české. České budou ve všech programech i doprovodné komentáře a tištěné texty.

Potřebné vybavení

K vývoji programů budete potřebovat vývojovou sadu JDK, kterou můžete stáhnout na adrese <https://www.oracle.com/java/technologies/javase-downloads.html>. Potřebujete ale sadu JDK 21 nebo mladší. Na starších verzích *Javy* by naše programy nemusely pracovat. Počítejte s tím, že instalační soubory zabírají zhruba 170 MB a po instalaci bude sada zabírat přes 300 MB.

K vývojové sadě je vhodné si na téže stránce stáhnout a nainstalovat i dokumentaci. Pokud ale neopouštíte web, tak můžete využívat i její trvalou instalaci na webu. Musíte

se však smířit s tím, že ji seženete pouze anglicky. ZIP soubor s dokumentací zabírá asi 50 MB a po rozbalení bude zabírat zhruba 460 MB.

Pro úspěšný vývoj programů potřebujete ještě vhodný vývojový nástroj. S těmi nejpoužívanějšími vás stručně seznámím v podkapitole [1.4 Vývojové nástroje](#) na straně [41](#).

Doprovodné programy

Text knihy je prostoupen řadou doprovodných programů. Budete-li si je chtít spustit a ověřit jejich funkci, potřebujete je nejprve stáhnout. Najdete je na stránce knihy na adrese² http://knihy.pecinovsky.cz/77_java_nz2. Měli byste zde najít příslušné soubory i jejich stručný popis.

Na stránkách knihy najdete archivní ZIP-soubor, jehož název začíná **Java77_NZ2_v** a za tímto konstantním úvodem je uvedeno číslo verze a datum jejího uložení. To pro případ, že by se v programech objevila chyba a já jsem na stránku uložil opravenou verzi.

Když soubor rozbalíte na disk, vytvoří složku **Java77_NZ2**, ve které budou následující podsložky:

77_JAVA Zdrojové soubory *Javy* vytvářené od kapitoly 11 dále. O jejich struktuře si povíme podrobněji, až je začneme používat.

77_JSH Kód zadávaný operačnímu systému anebo programu *JShell*, s jehož pomocí demonstruji vykládanou látku. Soubory mají příponu **jsh**.

Názvy souborů začínají vždy písmenem **s** následovaným dvojmístným číslem kapitoly, k níž se soubor vztahuje. Následují-li dvě podtržítka a text (např. **s02__Prostředí_JShell**), jedná se o soubor příkazů zadaných v průběhu kapitoly. Je-li za číslem malé písmeno následované jedním podtržítkem (např. **s01a_script**), jedná se o samostatný skript.

Text ze souborů příkazů můžete zkopírovat do schránky a zadat jej přímo virtuálnímu stroji, abyste při zkoušení příkladů nemuseli vše psát ručně.

77_JWD Kopie výpisů v knize, a to včetně čísel řádků. Jsou určeny k tomu, abyste si je otevřeli v samostatném okně, anebo si je vytiskli a nemuseli při čtení delšího doprovodného komentáře výpisů neustále listovat mezi komentářem a komentovaným výpisem.

Soubory v této složce mají stejné názvy jako odpovídající soubory ve složce **77_JSH**, ale mají příponu **jdoc**.

77_LIB Knihovny, které budeme v demonstračních doprovodných programech využívat. V průběhu textu vás s nimi seznámím.

² Číslicí 67 v adrese se nevzdrušujte, jedná se pouze o interní označení pořadí vytvářené knihy, protože bychom v nich jinak bloudili.

Složku s doprovodnými programy si umístíte, kam uznáte za vhodné. Já je mám rozbalené do kořenové složky substituovaného disku **J:**. Pracujete-li na *Windows* a budete-li si chtít také umístit programy na samostatný disk, najdete na webové stránce knihy přílohu popisující, jak se takový disk vytváří.

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, jsou vysazeny tučně .
<i>Název</i>	Názvy firem a jejích produktů jsou vysazeny <i>kurzivou</i> . Kurzivou jsou také názvy kapitol, podkapitol a oddílů, na které v textu odkazují.
Citace	Texty, které si můžete přečíst na displeji, např. názvy polí v dialogových oknech či názvy příkazů v nabídkách, jsou vysazeny tučným bezpatkovým písmem .
<u>Odkaz</u>	Celá kniha je prošpikovaná křížovými odkazy na související pasáže. Ne-ní-li odkazovaný objekt (kapitola, obrázek, výpis programu, ...) na stejné stránce nebo na některé ze sousedních stránek, je pro čtenáře tištěné verze doplněn o číslo stránky, na níž se nachází. Čtenářům elektronické verze stačí, když na něj klepnou, a použitý prohlížeč by je měl na odkazovaný objekt ihned přenést.
Program	Identifikátory a další části programů zmíněné v běžném textu jsou uvedeny 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 výpustku – např. metoda(...) .
keyword	Klíčová slova, což jsou názvy se specifickým významem, jsou vysazena tučně a slabě podtržená. V elektronických verzích budou navíc tmavočervená.
"String"	Texty, které jsou součástí kódu, se slangově označují jako stringy. Tyto texty tvoří často důležitou součást kódu a v elektronických verzích jsou proto pro zvýraznění vysazeny fialově a jemně podbarvené.
výzva	Výzvy operačního systému a programu <i>JShell</i> k zadání dalšího příkazu jsou vysazeny tmavočerveně a jsou podbarvené.
zadání	Ve výpisech programů a odpovědí počítače jsou texty zadávané uživatelem vysazeny tučně (v elektronických verzích pro zvýraznění modře).
Chyba	Chybová hlášení jsou vysazena červeně na bílém podkladu.

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, jenž 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.



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“.



Obrázek počítače označuje zadání úkolu, který máte samostatně vypracovat. Seznam všech úloh najdete v rejstříku pod heslem „úloha“.



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



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, jimiž 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 zařídit, aby vám alespoň pokud možno nevadily.



Brýle označují tzv. „poznámky pro šfouraly“, v nichž se vás snažím seznámit s některými zajímavými vlastnostmi probírané konstrukce nebo upozorňuji na některé souvislosti, které však nejsou k pochopení látky nezbytné.

Odbočka – podšeděný blok

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

Zpětná vazba

Přes veškeré úsilí, které jsem knize věnoval, nelze 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 vylepšení, pošlete prosím e-mail s předmětem [77 Java NZ2 DOTAZ](mailto:77_Java_NZ2_DOTAZ) na adresu rudolf@pecinovsky.cz. Na stránku knihy http://knihy.pecinovsky.cz/77_Java_NZ2 se pak pokusím co nejdříve zanešt příslušná errata s opravou, kterou pak zapracuji do případného dalšího vydání.

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

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

O autorovi

Ing. Rudolf Pecinovský, CSc. studoval teoretickou kybernetiku na *Fakultě jaderné a fyzikálně inženýrské ČVUT* a poté technickou kybernetiku na *Fakultě Elektrotechnické ČVUT*, kterou ukončil v roce 1979. Titul CSc. získal v Ústavu teorie informace a automatizace ČSAV v roce 1983. Od počátku osmdesátých let učí a publikuje, přičemž svůj výzkum soustředí především na oblast vstupních kurzů moderního programování pro naprosté začátečníky. V současné době učí na *Fakultě jaderné a fyzikálně inženýrské ČVUT* a na *Fakultě informatiky a statistiky Vysoké školy ekonomické v Praze*. Vedle toho učí ještě v řadě kurzů pro začátečníky i profesionální programátory. Doposud mu vyšlo přes 70 knih, které byly přeloženy do pěti jazyků. Většina jeho knih je zaměřena na výuku moderního programování a návrh objektově orientované architektury.



Část A

Superzáklady

Tato část je určena pro čtenáře, kteří se s programováním doposud nikdy nesetkali. Přináší proto řadu základních informací, jež byste měli znát před tím, než začnete doopravdy programovat. Nejprve vás uvede do světa programování a prozradí jeho základní vlastnosti a současné trendy. Poté vám velmi stručně představí nejpoužívanější nástroje pro vývoj programů. Pak poněkud podrobněji rozebere prostředí *JShell*, které bude po většinu výkladu použito k tomu, abychom získali okamžitou odpověď na zadané programové obraty. Budete tak moci od začátku zkoušet probírané konstrukce, aniž byste museli používat něco, co jsme ještě neprobrali. Na to naváže výklad o datových typech a zadávání jednoduchých hodnot. Část končí představením proměnných a výkladem nejpoužívanějších výrazů a operací.

Kapitola 1

Přehra



Co se v kapitole naučíte

V této kapitole se seznámíte s nástroji, které budete potřebovat při studiu dalších částí knihy. Nejprve vám prozradí něco o historii a současných trendech v programování a seznámí vás s programovacím jazykem a platformou *Java*. Na závěr velice stručně představí nejpoužívanější vývojové nástroje.

1.1 Trocha historie

Historie počítačů (tj. strojů, u nichž je postup výpočtu řízen programem) a programování se začala psát již v devatenáctém století. Charles Babbage tehdy dostal zakázku od anglického námořnictva na vylepšení svého počítacího stroje pro výpočet navigačních tabulek, které se hemžily chybami. V roce 1837 dokončil návrh počítacího stroje, který byl řízený programem zadávaným na děrných štítcích. Zbytek života se pak věnoval jeho konstrukci. Stroj byl mechanický a měl být poháněn parním strojem, ale z programátorského hlediska již umožňoval značnou část operací, kterými se honosí současné počítače.

První plně funkční počítač postavil až v roce 1936 v Německu Konrád Zuse. Nabídl jej armádě, ale tehdejší německá generalita prohlásila, že takový nesmysl nebude armáda nikdy potřebovat. Bůh ví, kam by se vývoj ubíral, kdyby generálové nebyli tak krátkozrací a počítač pomáhal německým inženýrům za druhé světové války při výpočtech.

Další počítače se objevily na konci druhé světové války v USA a ve Velké Británii. Nejslavnějším z nich byl ENIAC, který byl vyroben v roce 1944 a byl prvním čistě elektronickým počítačem (ostatní ještě používaly relé³ či dokonce mechanické prvky). Programoval se však propojováním jednotlivých částí kabely, takže výměna programu byla velmi náročná a zdlouhavá.

³ Relé je elektromechanický prvek, kde průchod proudu cívkou zapříčiní sepnutí nebo rozpojení kontaktů. Rychlá relé dokázala sepnout i 100krát za sekundu – to byla také maximální rychlost tehdejších počítačů.

Skutečné počítačové (a tím i programátorské) orgie však vypukly až v padesátých letech, kdy se začaly počítače vyrábět sériově a počítačová věda (computer science) se zabydlela na všech univerzitách.

Problémem prvních počítačů byla jejich vysoká poruchovost. Antonín Svoboda tehdy přišel s řadou konstrukčních vylepšení umožňujících vytvořit z nespolehlivých součástek spolehlivý počítač.⁴ Konstrukteři se postupně naučili vytvářet vysoce spolehlivé počítače. Nejslabším článkem celého komplexu se tak stával program.

1.1.1 Co je to program

Program bychom mohli charakterizovat jako v nějakém programovacím jazyku zapsaný předpis popisující, jak má procesor, pro nějž je program určen (v našem případě počítač), splnit zadanou úlohu.

Cílovým procesorem nemusí být vždy počítač. Oblíbeným příkladem programů jsou např. kuchařské předpisy. V předpočítačových dobách zaměstnávaly některé instituce (např. armáda) velké skupiny počtářů,⁵ jež na mechanických kalkulačkách řešily podle zadaných programů výpočty, které dnes předhazujeme počítači.

Na programu je důležité, že musí být napsán v nějakém jazyku, kterému rozumí programátor. Napsaný program je pak převeden do „jazyka“, kterému rozumí počítač – do takzvaného strojového kódu. Tento převod mívá na starosti jiný program, který označujeme jako překladač nebo kompilátor.

Vybrat jazyk pro kuchařský předpis je jednoduché, vybrat jazyk pro počítač je mnohem složitější. Vlastnosti použitého jazyka totiž naprosto zásadně ovlivňují jak rychlost vývoje programu, tak i rychlost a kvalitu výsledných programů. Proto také prošly programovací jazyky i metodiky jejich používání celou řadou revolučních změn.

1.1.2 Postupná změna priorit

Počítače řešily zpočátku vědecko-technické výpočty, ale postupně byly nasazovány v dalších a dalších oblastech a programátoři pro ně vytvářeli dokonalejší a dokonalejší programy. Programy byly čím dál rafinovanější a složitější, a to začalo vyvolávat velké problémy. Programátoři totiž přestávali být schopni své programy rozchodit, a když je vítězně rozchodili, nedokázali z nich v rozumném čase odstranit chyby, které se v programu objevily.

⁴ V tehdejší době (tj. v padesátých letech) byla naše republika na špičce světového vývoje počítačů. Svobodovy myšlenky byly později uplatněny v počítačích řídících americké kosmické lodě. Jenže to už bylo v době, kdy byl z naší republiky vypuzen, emigroval do USA a tam působil jako špičkový počítačový odborník.

⁵ Přiznejme si, že to tehdy byly většinou počtářky, protože muži dělají při práci podobného druhu příliš mnoho chyb. Takovéto „lidské počítače“ pomáhaly např. s vývojem atomové pumy či prvními lety do vesmíru.

Ještě větším problém nastal v okamžiku, kdy si uživatelé program oblíbili a chtěli po programátorech, aby do něj doplnili další funkce nebo aby upravili vlastnosti, které jim nevyhovovaly. Začalo se hovořit o softwarové krizi.

Tato krize vedla ke vzniku nových programovacích jazyků a metodik, které měly jediný cíl: pomoci programátorům psát spolehlivé a snadno upravovatelné programy. V padesátých letech minulého století se tak prosadily *vyšší programovací jazyky*, v šedesátých letech *modulární programování*, v sedmdesátých letech na ně navázalo *strukturované programování* a v průběhu osmdesátých a zejména pak devadesátých let ovládlo programátorský svět *objektově orientované programování*, jehož vláda pokračuje dodnes. (Jednotlivé termíny si za chvíli probereme podrobněji.)

V současné době se začíná v některých oblastech prosazovat i *funkcionální programování*, které se soustřeďuje na efektivní vyřešení některých částí projektů, ale základní paradigma, na němž stojí opravdu rozsáhlé projekty, je nadále objektové.



Termínem *paradigma* označujeme celkový přístup k řešení problému a základní programovací styl. Nejpopulárnější programovací jazyky současné doby (*Java* patří mezi ně) jsou navrženy tak, aby umožňovaly vývoj programů podle několika paradigmat.

Hlavním cílem programátorů v počátcích programování bylo, aby jejich programy spotřebovaly co nejméně paměti a byly co nejrychlejší. Tehdejší počítače totiž měly paměti málo, byly z dnešního hlediska velice pomalé a jejich strojový čas byl drahý. Se stoupající složitostí programů však byly takto psané programy stále méně stabilní a stále hůře udržovatelné. Současně s tím, jak klesala cena počítačů, jejich strojového času i paměti, začínal být nejdražším článkem v celém vývoji člověk.

Cena strojového času a dalších prostředků spotřebovaných za dobu života programu začínala být pouze zlomkem ceny, kterou bylo nutné zaplatit za jeho návrh, zakódování, odladění a následnou údržbu. Začal být proto kladen stále větší důraz na produktivitu programátorů i za cenu snížení efektivity výsledného programu.

Prakticky každý program zaznamená během svého života řadu změn. Požadavky zákazníka na to, co má program umět, se většinou průběžně mění, a program je proto nutné průběžně upravovat, rozšiřovat a vylepšovat. Celé současné programování je proto vedeno snahou psát programy nejenom tak, aby pracovaly efektivně, tj. rychle a s minimální spotřebou různých zdrojů (operační paměť, prostor na disku, kapacita sítě atd.), ale aby i jejich vývoj byl co nejefektivnější a aby je také bylo možné kdykoliv jednoduše upravit a vylepšit.

Předchozí zásady krásně shrnul Martin Fowler ve své knize *Refactoring* ([8], [9]):

„Napsat program, kterému porozumí počítač, umí i hlupák.

Dobry programátor píše programy, kterým porozumí i člověk.“

Mějte při tvorbě svých programů tuto zásadu neustále na paměti.

1.1.3 Vývoj metodik programování

Jak jsme řekli, v průběhu doby se prosadilo několik metodik, které doporučovaly, jak programovat, abychom byli s programem co nejdříve hotovi a výsledný program byl co nej kvalitnější. Tyto metodiky na sebe postupně navazovaly. Každá z nich těžila ze zkušeností získaných při aplikaci předchozí metodiky a stavěla na nich.

První revolucí bylo zavedení vyšších programovacích jazyků, jež programátory osvobodily od zapisování programů v podobě, která přesně odpovídala použitým instrukcím použitého počítače, a zavedly vyjadřování, jež bylo daleko bližší zvyklostem lidí. To pak umožnilo vědcům a technikům nečekat s každou prkotinou na programátora a naprogramovat si spoustu jednodušších výpočtů sami.

Zavedení vyšších programovacích jazyků umožnilo tvorbu složitějších programů. Vyvíjené programy byly zanedlouho tak složité, že se v nich programátoři přestali orientovat. Modulární programování ukazovalo, že rychlost vývoje i kvalitu výsledného programu zvýšíme, když podle starého římského hesla *divide et impera* (rozděl a panuj) vhodně rozdělíme velký projekt do sady menších, rozumně samostatných modulů, které na závěr sloučíme.

Produktivita programátorské práce se zvýšila, ale záhy se začalo ukazovat, že programátoři mají problémy nejenom s rozchozením obřích programů, ale i jednotlivých modulů. V roce 1968 publikoval E. W. Dijkstra článek,⁶ který kritizoval v té době převládající způsob psaní programů a prosazoval tzv. *strukturované programování*. Jeho článek rozpoutal obrovské diskuse, jež krásně charakterizoval jiný článek nazvaný *Real Programmers Don't Use Pascal*⁷ a publikovaný v červnu 1983 v časopise *Datamation* (v češtině vyšel pod názvem *Opravdoví programátoři nepoužívají Pascal*⁸ a vřele vám doporučuji, abyste si jej přečetli).

Strukturované programování se soustředilo na kód a ukazovalo, že dalšího zvýšení produktivity vývoje i kvality výsledných programů dosáhneme dodržením několika jednoduchých zásad při vlastním psaní kódu. Jeho hlasatelé předváděli, že opuštěním nejrůznějších fint, kterými se programátoři snažili o optimalizaci kódu, a maximálním zpřehledněním vytvářeného programu dosáhneme nejenom vyšší produktivity programátora, ale v mnoha případech i vyšší efektivity výsledného programu.

Se stále rostoucí složitostí vyvíjených programů se začalo ukazovat, že jednou z největších překážek v efektivní tvorbě kvalitních programů je tzv. *sémantická mezera* mezi tím, co chceme vytvořit, a tím, co máme k dispozici. Naše programy mají řešit široké spektrum úkolů od řízení mikrovlnné trouby přes nejrůznější kancelářské a grafické programy a hry až po složité vědecké úlohy, kosmické lety či protivzdušnou obranu kontinentu. Ve svém rozletu jsme ale odkázáni na stroje, které si umějí

⁶ Dijkstra, E. W. "Letters to the editor: go to statement considered harmful". Communications of the ACM 11 (3): 147–148. DOI:10.1145/362929.362947. ISSN 0001-0782.

⁷ Najdete jej např. na adrese <http://www.webcitation.org/659yh1oSh>, ve Wikipedii pak má vlastní heslo [Real Programmers Don't Use Pascal](#).

⁸ Na české wiki zatím heslo nemá, ale po zadání hesla *opravdoví programátoři* vrátí Google řadu odkazů – např. <http://www.logix.cz/michal/humornik/Pojidaci.Kolacu.xp>.

pouze hrát s nulami a jedničkami. Čím budou naše vyjadřovací možnosti blíže zpracováváné skutečnosti, tím rychleji a lépe dokážeme naše programy navrhnout, provoznit a udržovat.

Jinými slovy: Kvalita programu a rychlost jeho tvorby je velice úzce svázána s hladinou abstrakce, kterou při jejich tvorbě používáme. Budeme-li např. programovat ovládání robota, bude pro nás výhodnější programovací jazyk, v němž můžeme zadat příkazy typu „zvedni pravou ruku“, než jazyk, v němž musíme vše vyjadřovat pomocí strojových instrukcí typu „dej do registru A dvojku a obsah registru A pak pošli na port 27“.

Objektově orientované programování (v dalším textu budeme používat zkratku OOP) se proto obrátilo do vyšších hladin abstrakce a ukázalo, že vhodným zvýšením abstrakce dokážeme rychle a spolehlivě navrhovat a vyvíjet ještě větší a komplikovanější projekty. Studie z konce osmdesátých let dokonce ukázaly, že programy obsahující více než cca 100 000 příkazů již není v lidských silách naprogramovat s akceptovatelnou trojicí (*spolehlivost, doba vývoje, cena*) bez použití OOP.

Jak už jsme řekli, poslední dobou se vedle OOP prosazuje i funkcionální programování. To sice neaspiruje na pozici základního paradigmatu pro návrh rozsáhlých systémů, ale umožňuje efektivně navrhnout některé jejich části.

1.1.4 OOP a OFP

OOP přichází s výrazovými prostředky, jež nám umožňují maximálně zvýšit hladinu abstrakce, na níž se „bavíme“ s našimi programy, a tím maximálně zmenšit onu sémantickou mezeru mezi tím, co máme k dispozici a co bychom potřebovali. Umožňuje programovat efektivněji a vytvářet spolehlivější programy. Chceme-li však jeho výhod plně využít, nesmíme zapomínat na nic z toho, s čím přišly předchozí metodiky.

OOP je považováno za hlavní proud současného programování. Všechny rozšířené moderní programovací jazyky se honosí tím, že umožňují navrhovat programy objektově. Podpora OOP byla postupně doplněna i do těch jazyků, které vznikly dávno před tím, než se začalo o nějakém OOP hovořit. Prakticky žádný z nově vznikajících jazyků aspirujících na široké použití si již nedovolí nepodporovat OOP, i když často nepodporuje jeho původní podobu, ale nějakou jeho modifikaci.

Funkcionální programování pak přichází s některými zásadami, jejichž dodržení usnadňuje návrh programů, u nichž je vhodné jejich řešení kvůli zvýšení rychlosti rozdělit mezi více procesorů. Jeho použití má své výhody i v některých dalších oblastech. I s jeho základy vás v této učebnici seznámíme.

1.2 Překladače, interprety, platformy

Tato podkapitola je určena těm, kteří se nespokojí jen s tím, že věci fungují, ale chtějí také vědět, jak fungují. Naznačíme si v ní, jak je v počítači zařízeno, že programy pracují.

1.2.1 Operační systém a platforma

Operační systém je sada programů, jejímž úkolem je zařídit, aby počítač co nejlépe sloužil zadanému účelu. Operační systémy osobních počítačů se snaží poskytnout co největší komfort a funkčnost jak lidským uživatelům, tak programům, které operační systém nebo tito uživatelé spouští. (Teď nehodnotíme, jak se jim to daří.)

Operační systém se snaží uživatele odstínit od hardwaru použitého počítače. Uživatel může střídat počítače, avšak dokud bude na všech stejný operační systém, bude si se všemi rozumět.

Při obsluze lidského uživatele to má operační systém jednoduché: člověk komunikuje s počítačem pomocí klávesnice, obrazovky, myši a případně několika dalších zařízení. Ty všechny může operační systém převzít do své správy a zabezpečit, aby se nejrozumnější počítače chovaly vůči uživateli stejně.

U programů to má ale složitější. Programy totiž potřebují komunikovat nejenom s operačním systémem (např. když chtějí něco přečíst z disku nebo na něj něco zapsat), ale také přímo s procesorem, kterému potřebují předat své instrukce k vykonání. Problémem ale je, že různé procesory rozumí různým sadám instrukcí.

Abychom věděli, že náš program na počítači správně poběží, musíme vědět, že počítač bude rozumět té správné sadě instrukcí a že na něm poběží ten správný operační systém. Kombinaci *operační systém + použitý hardware* budu v dalším textu označovat jako **platforma HWOS**.⁹

Nejrozšířenější platformou HWOS současných osobních počítačů je operační systém *Windows* na počítačích s procesory postavenými na architektuře x86-64, které však mají i verze běžící na počítačích s jinými procesory. Další vám známé platformy budou nejspíš operační systémy z rodiny *Linux* běžící na různých hardwarových platformách, mezi něž svým způsobem patří i operační systém *MacOS* běžící na počítačích *Apple*. V našem přehledu bychom neměli zapomínat ani na platformu *Android* určenou pro mobilní telefony a tablety, které však svým výkonem občas překonají leckterý osobní počítač.

1.2.2 Programovací jazyky

Jak asi všichni víte, pro zápis programů používáme nejrozumnější programovací jazyky. Ty jsou vymyšleny tak, aby v nich mohl člověk co nejlépe popsat svoji představu o tom, jak má počítač splnit požadovanou úlohu.

Program zapsaný v programovacím jazyku pak musíme nějakým způsobem převést do podoby, které porozumí počítač. Podle způsobu, jakým postupujeme, dělíme programy na překládané a interpretované.

⁹ Přesnější definice by byla: *HWOS je souhrn všech softwarových a hardwarových komponent, které umožňují spouštění programů.*

1.2.3 Způsoby zpracování programu

Abychom měli z vytvořených programů nějaký užitek, musíme je spustit. K tomu se používá několik přístupů.

Překládaný program

U překládaných programů se musí napsaný program nejprve předat zvláštnímu programu nazývanému překladač (někdo dává přednost termínu kompilátor), který jej přeloží (zkompiluje): převede jej do podoby, s níž si již daná platforma ví rady, tj. musí jej přeložit do kódu příslušného procesoru a používat instrukce, kterým rozumí použitý operační systém. Přeložený program pak můžeme kdykoliv na požádání spustit.

Interpretovaný program

Naproti tomu interpretovaný program předáváme v podobě, v jaké jej programátor vytvořil, programu označovanému jako interpret. Ten obdržený program prochází a ihned jej také provádí.

Porovnání

Výhodou překládaných programů je, že většinou běží výrazně rychleji, protože u interpretovaných programů musí interpret vždy nejprve přečíst kus programu, zjistit, co má udělat, a teprve pak může tento požadavek vykonat.

Výhodou interpretovaných programů bývá na druhou stranu to, že jim většinou tak moc nezáleží na tom, na jaké platformě z rodiny HWOS běží. Stačí, když na ní běží potřebný interpret. Mohli bychom říci, že platformou těchto programů je právě onen interpret. Vytvoříte-li pro nový počítač interpret, můžete na něj vzápětí přenést i všechny vytvořené programy. Kdykoliv tyto programy po systému něco chtějí, požádají o to svoji platformu (interpret) a ta jim příslušné služby zprostředkuje. Takovým programům pak může být jedno, na jakém procesoru a pod jakým operačním systémem běží, protože se beztak „baví“ pouze se svou platformou.

Naproti tomu překládané programy se většinou musí pro každou platformu trochu (nebo také hodně) upravit a znovu přeložit. Při implementaci programu pro více platform je bývá pracnost přizpůsobení programu jednotlivým platformám srovnatelná s pracností vývoje jeho první verze.

Hybridní programy

Vedle těchto základních druhů programů existují ještě hybridní programy, které jsou současně překládané i interpretované a které se snaží sloučit výhody obou skupin. Hybridní program se nejprve přeloží do jakéhosi mezijazyka, který je označován jako **bajtkód**. Ten je vymyšlen tak, aby jej bylo možné co nejrychleji interpretovat.

Program přeložený do bajtkódu je potom interpretován speciálním interpretem často označovaným jako **virtuální stroj**.¹⁰

¹⁰ Virtuální stroj se mu říká proto, že se vůči programu v mezijazyku chová obdobně, jako se chová procesor vůči programu v čistém strojovém kódu.

Hybridní programy spojují výhody obou kategorií. K tomu, aby v nich napsané programy mohly běžet na různých platformách, stačí pro každou platformu vyvinout potřebný virtuální stroj. Ten pak vytváří vyšší, mnohem univerzálnější platformu. Je-li tento virtuální stroj dostatečně „chytřý“ (a to jsou v současné době prakticky všechny), dokáže odhalit často se opakující části kódu a někde stranou je přeložit, aby je nemusel pořád kolem dokola interpretovat. Hovoříme pak a JIT (*Just-In-Time*) překlada.

Ty nejchytřejší virtuální stroje dokonce sledují, co program dělá, a zjistí-li, že je to výhodné, přeloží rychle část programu znovu tak, aby využila některých právě platících speciálních podmínek a mohla běžet ještě rychleji. Na těchto strojích pak může běžet hybridní program skoro stejně rychle jako překládaný a v některých speciálních případech dokonce i rychleji. Přitom si stále udržuje nezávislost na hardwarové platformě a použitém operačním systému.

Takovýto rychlý virtuální stroj používá i platforma *Java*. Její virtuální stroj bývá označován JVM, což je zkratka z anglického *Java Virtual Machine*.



Na překládané, interpretované a hybridní bychom měli dělit programy, avšak často se takto dělí i programovací jazyky. Je sice pravda, že to, zda bude program překládaný, interpretovaný nebo hybridní, není závislé na použitém jazyku, ale je to především záležitostí implementace daného jazyka, nicméně každý z jazyků má svoji typickou implementaci, podle níž je pak zařazován.

Prakticky všechny jazyky sice mohou být implementovány všemi třemi způsoby a řada jich opravdu ve všech třech podobách existuje (jako příklad bychom mohli uvést jazyky *Basic*, *C*, *Java* a *Python*), ale u většiny převažuje typická implementace natolik výrazně, že se o těch ostatních prakticky nemluví. Z vyjmenované trojice je např. původní *Basic* považován za interpretovaný jazyk, *Java* za hybridní, jazyk *C* za překládaný a jazyk *Python* za interpretovaný, přestože je zpracováván hybridně obdobně jako *Java*.

Hybridní implementace jazyků se v posledních letech výrazně prosadily a jsou dnes králi programátorského světa. Vyvíjí v nich převážná většina programátorů a procento implementací v těchto jazycích neustále vzrůstá.

1.3 Java a její zvláštnosti

O splnění zásad objektově orientovaného programování se snaží tzv. objektově orientované jazyky. Na počátku tohoto století mezi nimi získal největší popularitu programovací jazyk *Java*, který se narodil v roce 1995 a hned po svém vzniku zaznamenal velký ohlas. Brzy se stal nejpoužívanějším programovacím jazykem, a přestože už není králem, tak si oblibu stále udržuje. V oblasti návrhu rozsáhlých systémů dnes nemá konkurenci. V současné době v něm vyvíjí (alespoň podle firmy Oracle) přes 9 miliónů programátorů.

1.3.1 Java je jazyk i platforma

Jedním z klíčových záměrů autorů Javy bylo vytvořit nejenom jazyk, ale celou platformu. To znamená, že programy napsané v Javě mohou běžet na různých HWOS bez nutnosti úprav.

Platformu *Javy* tvoří výše zmíněný virtuální stroj (JVM) spolu se základní knihovnou nejpoužívanějších podprogramů, která bývá označována jako *Java API (Application Programming Interface)* – programové rozhraní *Javy*.

Programům napsaným v *Javě* je pak jedno, na jaké platformě z rodiny HWOS běží, protože nad touto platformou bude platforma *Javy*, a ta bude z hlediska dané aplikace na všech počítačích stejná, nezávisle na podkladové platformě HWOS.

Marketingové oddělení firmy Sun se při uvádění Javy rozhodlo pojmenovat jazyk i platformu stejně (obáváme se, že ani netušili, že jsou to dvě různé věci). Z toho ale vycházela řada nedorozumění. Musíme proto vždy rozlišovat, kdy hovoříme o platformě a kdy o jazyku.

Překladače do bajtkódu platformy *Java* existují pro nepřehledné množství nejrůznějších programovacích jazyků,¹¹ které bývají hromadně označovány jako JVM jazyky, protože se překládají do bajtkódu pro virtuální stroj *Javy (Java Virtual Machine)*. Jazyk *Java* je pouze jedním z nich.

Zdrojové a přeložené soubory

Programy vytváříme jako textové soubory, které označujeme jako **zdrojové soubory**. Zdrojové soubory programů v jazyku *Java* mají podle konvence příponu **.java**.

Vytvořenou sadu zdrojových souborů předáme překladači, který obdržený text převede na bajtkód – každý zdrojový soubor převede na jeden nebo více **class-souborů**, což jsou soubory s příponou **class** obsahující výsledný bajtkód. Tyto class-soubory pak zpracovává virtuální stroj.

Programy napsané v různých jazycích jsou všechny převedeny do bajtkódu, takže jsou z hlediska virtuálního stroje více-méně ekvivalentní a jsou proto většinou schopny vzájemně komunikovat a spolupracovat. Můžete proto napsat část svého programu např. v *Javě* a druhou část v nějakém jazyku, který je pro daný úkol výhodnější (populární jsou např. jazyky *Clojure*, *Kotlin*, *Groovy* nebo *Scala*).

Více platform

Asi bych se zde měl zmínit o tom, že *Java* není jediná platforma, ale jsou to hned čtyři platformy:

- *Java SE (Standard Edition)* označuje základní platformu určenou pro vývoj desktopových aplikací a jednodušších verzí serverových aplikací. Tu budeme v této knize používat i my.
- *Java EE (Enterprise Edition)* označuje nadstavbu nad *Java SE* obsahující další knihovny specializované pro tvorbu rozsáhlých distribuovaných aplikací.

¹¹ Ty nejznámější najdete např. ve Wikipedii pod heslem [List of JVM languages](#), ale skutečný počet je mnohem větší a obnáší několik set jazyků a jejich verzí.

- *Java ME (Micro Edition)* označuje platformu určenou pro vývoj aplikací pro malá zařízení a pracuje s mírně zjednodušenou verzí *Javy*.
- *Java Card* je ještě osekanější verze používaná při vývoji aplikací určených pro čipové karty. Tato platforma bývá někdy zařazována pod *Java ME*.

V této učebnici budeme vytvářet programy a skripty pro základní platformu *Java SE*. To, kterou platformu pak budete používat ve své praxi, záleží na oblasti, pro niž budete vyvíjet aplikace.

1.4 Vývojové nástroje

Při vývoji programů se používají nejrůznější nástroje, které mají programátorům (a nejen jim) maximálně ulehčit práci.

1.4.1 Základní vývojářská sada

Jako vývojáři musíte rozlišovat dvě verze programových balíčků, které je možné stáhnout:

- Jednodušší **JRE** (Java Runtime Environment – běhové prostředí *Javy*) poskytuje vše potřebné pro běh programů.
- Komplexnější **JDK** (Java Development Kit – sada pro vývoj v *Javě*) označovaná někdy také **SDK** (Software Development Kit – sada pro vývoj softwaru) je vlastně sada JRE doplněná o základní vývojové nástroje (překladač, generátor dokumentace, ladící program a další) a poskytuje tak vše potřebné pro vývoj programů.

Vy se chcete naučit pomocí této učebnice programovat, budete tedy potřebovat **JDK** (doporučuji vám pořídit si verzi 21 či vyšší). Verze 21 patří mezi verze s dlouhodobou podporou (*Long Term Support*). Další verzi s dlouhodobou podporou by měla být verze 25. Verze mezi nimi mají podporu vždy jen půl roku do příchodu další verze.

Uživatelům, kteří budou spouštět vaše programy, stačí **JRE**. V dalším textu budu předpokládat, že máte **JDK** nainstalováno podle pokynů v pasáži [Potřebné vybavení](#) na straně [25](#).

1.4.2 IDE

S **JDK** při vývoji programů teoreticky vystačíte (některé učebnice ani nic jiného nepoužívají), nutí vás však starat se o řadu konfiguračních detailů, které odvádějí vaši pozornost od vlastního programování. Převážná většina programátorů proto používá programy označované jako **IDE**, což je zkratka z anglického *Integrated Development Environment* – integrované vývojové prostředí. Ty za ně vyřeší konfigurační detaily a navíc jim pomohou i v řadě dalších oblastí.

Problémem těch „dokonalejších“ bývá, že zvládnutí daného vývojového prostředí je často náročnější než zvládnutí použitého programovacího jazyka. Při následné práci se ale vynaložená námaha bohatě vrátí, protože dobré IDE udělá řadu věcí za vás a v řadě dalších vás povede „za ručičku“ nebo vám alespoň bude dobře napovídat.

Vývojových prostředí pro *Java* existuje celá řada.¹² Některá jsou určena spíše začátečníkům, jiná jsou optimalizovaná pro profesionální programátory. Práce s těmi profesionálními je sice poněkud náročnější, ale pokud překonáte počáteční stadium, kdy vám bude připadat, že zvládnutí daného IDE je těžší než zvládnutí programovacího jazyka, tak se vám vyvinuté úsilí při pozdějším vývoji složitějších programů vrátí.

Pojďme si nyní alespoň stručně představit ta nejrozšířenější IDE.

JSHELL

JSHELL není plnohodnotné vývojové prostředí. Je to prostředí určené spíše pro experimenty a rychlé ověřování některých nápadů. Jeho hlavní výhodou je, že je součástí standardní instalace, takže je ihned k dispozici.

Druhou výhodou tohoto prostředí je, že netrvá na tom, abyste zadali celý program, ale je ochotné zpracovat i pouhé úryvky. Proto jej budeme používat u značné části AHA-příkladů demonstrujících funkci aktuálně probíraných konstrukcí. Podrobněji vám toto prostředí představíme v následující kapitole.

BlueJ¹³

BlueJ je vývojové prostředí navržené speciálně pro podporu výuky ve vstupních kurzech programování. Je maximálně jednoduché a názorné, takže si jej v pohodě osvojíte za půl hodiny. Jeho velkou výhodou je, že zobrazuje program primárně prostřednictvím diagramu tříd, což vám pomůže pochopit celou řadu zásad moderního programování a jejich aplikaci ve vlastních programech.

Pro mnohé studenty je výhodné i to, že vývojové prostředí *BlueJ* je lokalizované do češtiny a slovenštiny.

Nevýhodou *BlueJ* je, že málo napovídá (uživatel proto musí řadu operací realizovat „ručně“) a obecně je pro praktické použití příliš prostoduché, takže se pro reálný vývoj nepoužívá.

NetBeans¹⁴

Zárodek tohoto prostředí vznikl v roce 1996 jako závěrečná práce na MFF UK a jeho vývojové centrum sídlí stále v Praze. Po členité historii, kdy se několikrát měnili vlastníci, je v současné době vyvíjeno v rámci nadace *Apache Software Foundation*.

IDE *NetBeans* je nejstarší z „velké trojky“ profesionálních vývojových prostředí. Recenze o něm říkají, že z těch opravdu profesionálních je pro začínající programátory nejsnáze zvládnutelné, aniž by to nějak omezovalo jeho schopnosti. Kromě toho mu

¹² Jejich stručný (i když zdaleka ne kompletní) přehled najdete např. ve Wikipedii na adrese https://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments#Java.

¹³ Prostor *BlueJ* stáhnete na <https://bluej.org/>.

¹⁴ Prostor *NetBeans* stáhnete na <https://netbeans.apache.org/download/index.html>.

dávají přednost ti, kteří potřebují budovat velké systémy a nejsou ochotni za vývojové nástroje platit.

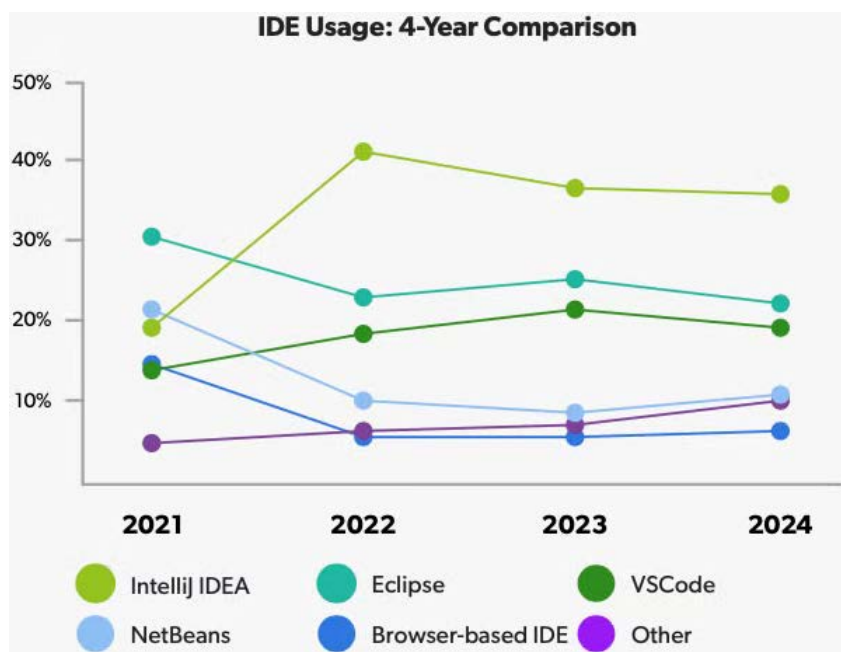
IntelliJ IDEA¹⁵

Toto IDE vydala v lednu 2001 pražská firma *JetBrains*. Přestože zpočátku nabízela pouze placenou verzi, získalo si mezi profesionálními programátory velkou oblibu. Když ale v srpnu 2013 vyšla *Community edition*, kterou bylo možné legálně stáhnout a používat zdarma, začala popularita tohoto prostředí raketově stoupat, takže je dnes s odstupem nejpoužívanější IDE pro *Java*.

IntelliJ IDEA nabízí mnoho pokročilých funkcí včetně zabudované umělé inteligence. Tyto funkce se sice mohou zpočátku zdát složité, ale s postupným učením se stávají jeho velkou výhodou.

Eclipse¹⁶

Poslední IDE z „velké trojky“. Narodilo se v srpnu 2001 pod patronací firmy IBM. Protože bylo zdarma a IBM se podařilo vytvořit velkou koalici firem, jež je pomáhaly zdokonalovat a vybavily je rozsáhlou sadou pluginů, které ho rozšiřují o podporu mnoha technologií.



Obrázek 1.1:

Průběh popularity nejpopulárnějších IDE
(zdroj: <https://www.jrebel.com/blog/best-java-ide>)

¹⁵ Prostředí IntelliJ IDEA stáhnete na <https://www.jetbrains.com/idea/download/>.

¹⁶ Prostředí Eclipse stáhnete na <https://www.eclipse.org/downloads/>.

Visual Studio Code¹⁷

V roce 2015 vydal *Microsoft* vývojové prostředí *Visual Studio Code*, které bylo (mohli bychom říci, že k všeobecnému překvapení) pod otevřenou licencí MIT, takže se zveřejněným a všeobecně dostupným zdrojovým kódem. Toto prostředí bylo primárně určeno pro vývoj programů pro .NET, ale rychle se začaly objevovat pluginy pro další jazyky a platformy, mezi jinými i pro *Javu* (např. *Extension Pack for Java*). Rychle získávalo oblibu mezi studenty a dnes patří k velmi rozšířeným obecně použitelným vývojovým prostředím.

Shrnutí

Jak už jsme naznačili, pro demonstraci látky budeme většinou používat prostředí *JShell*. Když pak v části [C](#) začneme definovat celé třídy, tak bude jedno, které vývojové prostředí použijete, protože zdrojové kódy jsou stejné.

Pro první kroky bych naprostým začátečníkům doporučoval používat prostředí *BlueJ*, protože je nejjednodušší a jeho zvládnutí nebude odvádět pozornost od vlastního návrhu programu. Navíc je to zdaleka nejefektivnější nástroj pro rychlý návrh a zobrazení základní architektury prostřednictvím diagramu tříd, a kvůli tomu jej budu v této učebnici občas využívat.

Ve chvíli, kdy se vám bude zdát, že už jste si základy návrhu objektově orientovaných programů osvojili, a budete rozhodnutí se programování dále věnovat, můžete přejít na některé z prostředí „velké trojky“ nebo na *Visual Studio Code*.

Kterému z nich dáte přednost, necháme na vás. Obecně radíme studentům, že budou-li mít kamaráda, kterým jim některé prostředí doporučí s tím, že jim pomůže s jeho zvládnutím a řešením případných problémů, tak ať jej poslechnou. Pokud takového kamaráda nemají, budou pro samostatný start asi nejvhodnější *NetBeans*, ale na školách, které znám, se většinou používá *IntelliJ IDEA*, které firma nabízí studentům zdarma v plné profesionální verzi.

1.5 Shrnutí a soubory pro opakování

V této kapitole jsme nic neprogramovali, takže po ní neexistují ani doprovodné soubory s programy pomáhajícími opakovat probranou látku.

¹⁷ Prostředí *Visual Studio Code* lze stáhnout na <https://github.com/Microsoft/vscode>.

Kapitola 2

Prostředí JShell



Co se v kapitole naučíte

Tato kapitola vás stručně seznámí s prostředím *JShell*, které budeme používat pro demonstraci základních konstrukcí jazyka a které je vynikajícím nástrojem pro nejrůznější experimenty. Dopředu se omlouvám, že při výkladu použiju pár termínů, které vysvětlím až v následujících kapitolách.

2.1 Prostředí JShell

Program *JShell* zařazujeme do kategorie nástrojů typu REPL, což je zkratka z anglického *read-evaluate-print-loop* (česky: *cyklus přečti-vyhodnoť-vytiskni*). Používá se pro rychlé testování kódu, experimentování s jazykem a učení se novým funkcím. Umožňuje psát a spouštět Java kód bez nutnosti vytváření kompletních programů. Jeho hlavní výhodou je interaktivita, která uživateli umožní rychle najít odpověď na nejrůznější otázky týkající se jeho kódu nebo možností nabízených jazykem.



Zde vám představím jen nejzákladnější vlastnosti prostředí *JShell*. Těm, které vlastnosti tohoto prostředí zaujmou a chtěli by je pro své experimenty používat častěji, doporučuji publikaci [\[22\]](#) nazvanou [Java 9 – JShell](#).

2.1.1 Spuštění programu JShell

Prostředí *JShell* spusťte v konzolovém okně, v němž je jako aktuální nastavena složka, v níž máte zdrojové soubory, s nimiž chcete pracovat. V případě doprovodných programů k této knize je touto složkou podsložka **77_JSH** složky, kam jste rozbalili archiv s doprovodnými programy.

Windows

Ve *Windows* toho dosáhnete nejnázem tak, že v *Průzkumníku* přejděte do požadované složky, v níž chcete pracovat, a zadáte nahoru do adresního řádku příkaz `cmd` (na velikosti písmen nezáleží).

Linux, macOS

Používáte-li grafické rozhraní, otevřete správce souborů (např. Nautilus, Files, Finder) a přejděte do požadované složky.

V mnoha správcích souborů existuje v kontextovém menu (po kliknutí pravým tlačítkem myši na složku nebo prázdné místo v okně složky) možnost **Otevřít v terminálu**. Tato možnost otevře terminál s již nastavenou aktuální složkou.

Alternativně, někteří správci souborů mají v horní liště tlačítko **Terminál** nebo tlačítko s podobným významem.

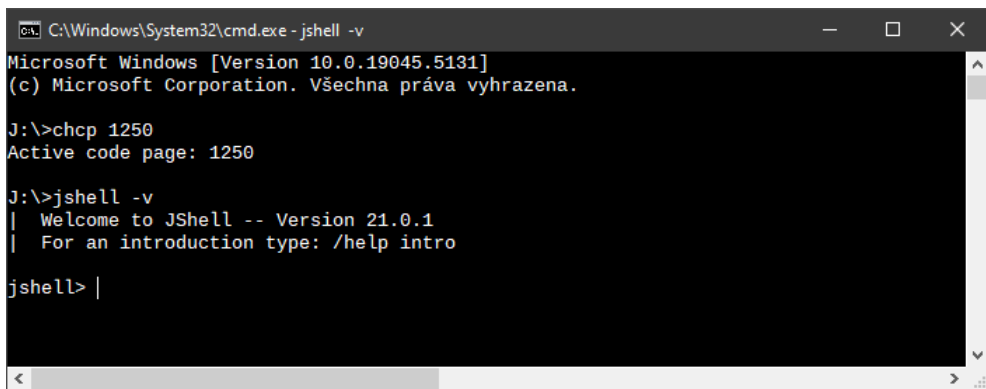
Vlastní spuštění

Máte-li správně instalovanou *Java*, tak už stačí zadat příkaz

```
jshell -v
```

kde argumentem `-v` žádáte *JShell*, aby vám o všech provedených akcích referoval co nejpodrobněji. Až budete zkušenější, můžete tento argument vynechat, případně jej nahradit jiným, kterým naopak žádáte o stručnější odpovědi. Podrobnosti najdete v publikaci [22].

Po spuštění programu se otevře konzolové okno podobné oknu na obrázku 2.1. Lišit se bude pouze v počátečních řádcích, v nichž se představuje operační systém a nastává kódová stránka, a v cestách k aktuální složce vypisovaných na počátku řádku.



Obrázek 2.1:

Konzolové okno otevřené po spuštění dávky `!_JShell.bat` na mém počítači

V dalším textu již nebudu ukazovat výpisy z okna jako obrázek, ale ve formě klasických výpisů programů s číslováními řádky, abych se mohl na čísla řádků snáze odvolávat. Obsah okna na obrázku 2.1 je ve výpisu 2.1.

Výpis 2.1: *Přepsaný obsah konzolového okna otevřeného po spuštění dávky `!_JShell.bat` na mém počítači*

```
1 Microsoft Windows [Version 10.0.19045.5131]
2 (c) Microsoft Corporation. Všechna práva vyhrazena.
3
4 J:\77_JSH>chcp 1250
5 Active code page: 1250
6
7 J:\77_JSH>jshell -v
8 | Welcome to JShell -- Version 21.0.1
9 | For an introduction type: /help intro
10
11 jshell>
```

Problémy s kódovou stránkou ve Windows

Uživatelé operačního systému *Windows* si musí ještě před spuštěním *JShell* nastavit kódovou stránku 1250 určenou pro střední a východní Evropu píšící latinkou. Při práci v konzolovém okně ve *Windows* totiž musíme obcházet drobný problém: když se českých *Windows* zeptáte v okně konzoly příkazem `chcp` na používanou kódovou stránku, tak vám odpoví, že 852, což je stránka převzatá z historického DOSu. Když se ale ptá program v *Javě*, tak tomu operační systém řekne, že 1250, což je kódová stránka *Windows* zavedená v roce 1992 pro systém *Windows* 3.1 a určená pro jazyky střední a východní Evropy píšící latinkou.

Když si proto kódovou stránku nepřepnete, tak se vám pak při tisku některých textů nebude zobrazovat text, ale nějaký rozsypaný čaj.

Nejhorší přitom je, že *Java* nativně používá kódování UTF-8 (kódová stránka 65001), ale když ji v konzolovém okně *Windows* nastavíte příkazem `chcp 65001`, tak operační systém bude virtuálnímu stroji *Javy* nadále tvrdit, že pracuje v kódové stránce 1250, takže zobrazovaný text bude opět chybný.

Proto je na řádku 4 příkazem `chcp 1250` nastavena kódová stránka 1250, čímž se většina problémů eliminuje.

Spuštění JShell

Na řádku 7 je spuštěn *JShell*, jenž vás na následujících řádcích vítá.

Na řádku 11 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.

2.1.2 JShell v IDE

Každé lepší IDE určené pro vývoj programů v *Javě* nabízí možnost spuštění *JShell* ve vyhrazeném panelu či okně. Pokud ale pracujete na nějakém složitějším projektu,

v němž používáte různé cizí knihovny a zdrojové soubory z různých zdrojů, může se vaše představa toho, co má být vašemu kódu dostupné, lišit od představy používaného IDE.

Abych těmto nesrovnalostem zamezil a abych navíc nezanášel potenciální problémy odvozené z toho, že každý čtenář používá jiné, anebo alespoň jinak konfigurované IDE, budu důsledně používat *JShell* přímo spouštěný v konzolovém okně. Budeme-li někdy používat nějaké externí zdroje, ukážeme si, jak se k tomu přihlásit. Tím by se mělo sjednotit chování programu pro různé čtenáře.

2.2 Úryvky (snippets)

Výrazy (anglicky *expressions*), **příkazy** (anglicky *statements*), **deklarace** (anglicky *declarations*) a **definice** (anglicky *definitions*) jazyka *Java*, které za výzvu napíšete, se hromadně označují jako **úryvky** (anglicky *snippets*). Kdykoliv napíšete nějaký úryvek, prostředí *JShell* jej ihned vyhodnotí a uloží výsledek.

Základní výzva má standardně podobu "**jshell**> ", tj. text `jshell` následovaný většítkem a mezerou. Nevejde-li se zadávaný příkaz na řádek, začínají další řádky příkazu pokračovací výzvou, která má standardně podobu " ...> ", tj. tři mezery, za nimi tři tečky, většítko a mezera.

Ve výpisu [2.2](#) najdete zadání dále popsaných úryvků spolu s odpověďmi prostředí.

Výpis 2.2: *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>
```

Zkuste napsat jednoduchý aritmetický výraz, např. `6 + 5` (řádek [1](#) výpisu), a potvrdit jej stiskem ENTER. Jak už bylo řečeno, prostředí *JShell* výraz spočítá a na dalším řádku oznámí jak výsledek, tak jeho uložení do pomocné proměnné nazvané `$1`. *JShell* totiž pro každý vyhodnocený výraz automaticky vytvoří proměnnou, jejíž jméno začíná znakem `$` a pokračuje číslem úryvku.



Úplným začátečníkům prozradím, že **proměnná** (anglicky *variable*) je místo v paměti, kam si můžeme uložit nějakou hodnotu pro budoucí použití. Proměnných může být v programu více, a proto má každá přidělené svoje jméno. Když pak později chceme uloženou hodnotu využít, odvoláme se na ni jménem příslušné proměnné a virtuální stroj dosadí na dané místo hodnotu, jež je v dané proměnné uložena. O proměnných si budeme podrobněji povídat v kapitole [4 Proměnné, výrazy a příkazy](#) na straně [74](#).

Prostředí *JShell* nám navíc na řádku [3](#) přidalo informaci, že vytvořená pomocná proměnná je typu **int** (datové typy probereme v následující kapitole), což je zkratka z anglického *integer* a prozrazuje, 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 [2.2](#) 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.



Abyste se ve výpisech lépe orientovali, jsou výzvy programu *JShell* podbarvené tmavší barvou (v elektronických verzích jsou vysazeny tmavě červenou), uživatelské příkazy jsou tučně (v elektronických verzích modře), standardní odpovědi programu obvykle a chybová hlášení nepodbarvená (v elektronických verzích červeně na bílém podkladu).

2.2.1 Použití proměnných

Jak jste si jistě všimli, na řádku [11](#) jsou místo čísel použity názvy dříve vytvořených proměnných. V poznámce, v níž jsem začátečníkům naznačoval, co je to proměnná, 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. To udělal i při vyhodnocování výrazu na řádku [11](#).

2.2.2 Identifikace úryvků

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

Obsahuje-li zadaný úryvek výraz, *JShell* tento výraz vyhodnotí a pro výsledek vytvoří novou pomocnou proměnnou, kterou pojmenuje znakem **\$** (dolar) následovaným **ID** zadaného úryvku. O tom jste se mohli přesvědčit např. ve výpisu [2.2](#) 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).

2.2.3 Terminologie: výrazy, příkazy, deklarace, definice

Tato pasáž bude trochu teoretická, takže ji ti netrpěliví mohou přeskóčit a vrátit se k ní, až si budou potřebovat ujasnit, že dobře chápou některé části výkladu.

Jako **výraz** (anglicky *expression*) označujeme část kódu, kterou můžeme vyhodnotit a získat výsledek. Výrazem jsou třeba součty v posledním výpisu. Výrazem je i název proměnné.

Jako **příkaz** (anglicky *statement*) pak označujeme část kódu, ve které počítači nařizujeme, co má v danou chvíli (tj. až bude příslušnou část kódu provádět) vykonat – např. vyhodnotit výraz a uložit výsledek do proměnné.

Deklarace (anglicky *declaration*) je část kódu, kde popisujeme, co budeme v okolním kódu používat – např. že budeme používat proměnnou pro ukládání celých čísel.

Definice (anglicky *definition*) je deklarace, která danou entitu nejenom popíše, ale současně iniciuje její vytvoření. Když např. proměnnou jenom deklarujeme, nemusí to znamenat, že bude vytvořena. Bude-li však dané proměnné přiřazena počáteční hodnota, tak vytvořena být musí, aby bylo hodnotu kam uložit. Z dané deklarace se tak automaticky stane definice.

Dokumentace *Javy* zavádí deklarace (a tím pádem i definice) jako zvláštní druh příkazů. Výkonný kód v *Javě* pak zapisujeme jako posloupnost příkazů.

2.2.4 Středník

Java požaduje, abychom každý příkaz ukončili středníkem. Zapomenutý středník patří mezi „oblíbené“ začátečnické chyby. Zadáme-li proto v programu výraz bez ukončujícího středníku (tak jsme doposud zadávali všechny výrazy), byla by to chyba. *JShell* se v takovém případě podívá, jestli by nepomohlo přidání středníku na konec řádku. Pokud je po přidání středníku vše v pořádku, *JShell* nás žádným chybovým hlášením neobtěžuje, prostě si daný úryvek zapamatuje i s přidáním středníkem.

2.2.5 Více objektů na řádku, zavlečené chyby

Definujeme-li více výrazů na jednom řádku, *JShell* vytvoří pro každý z těchto výrazů 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.

Výpis 2.3: Více úryvků na jednom řádku

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

Ve výpisu [2.3](#) je na řádku [1](#) pokus vložit více výrazů, aniž by byl první ukončen středníkem. Jak vidíte, *JShell* postupně ohlásil celou sérii chyb. Na řádku [3](#) vysvětlil, že mu chybí očekávaný středník, a na řádku [5](#) dokonce ukázal, kam si myslí, že by se měl onen středník do úryvku vypsaneho na řádku [4](#) vložit.

Pak se sice pokusil vyhodnocovat dál, ale první chyba jej poněkud rozhodila, takže další chyby, na něž 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. Po doplnění středníku na základě rad na řádcích [4-5](#) (středník a zadání upraveného příkazu) *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.

2.3 Příkazy prostředí JShell



‘Těm, kteří už chtějí co nejdříve začít s tvorbou kódu v *Java*, poradím, že mohou zbytek této kapitoly přeskočit a začít rovnou kapitolou [3 Zadávání hodnot](#) na straně [60](#). Příkazy, které budu ve zbytku této kapitoly vysvětlovat, sice budeme používat, ale k jejich výkladu se sem můžete vrátit, až je při práci s *JShell* použijeme, protože pak budete hned vědět, k čemu vám budou dobré.

Jednou za čas potřebujeme po prostředí i něco jiného než zpracování zadaného úryvku. Potřebujete si připomenout, co už jsme zadali, uložit to do souboru nebo naopak ze souboru nějakou zapamatovanou skupinu úryvků načíst. Pak využijete příkazy (anglicky *commands*) prostředí *JShell*.

Žádný výraz ani příkaz programu v jazyce *Java* nemůže začínat lomítkem. Příkazy prostředí *JShell* proto lomítkem začínají a podle toho je také poznáte. 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.



Začátečníky bych chtěl upozornit, že anglické termíny *command* a *statement* se sice oba překládají jako *příkaz*, ale každý znamená něco trochu jiného. Termín *statement* označuje syntaktickou jednotku jazyka, kdežto *command* označuje příkaz, který se má vykonat. Většinou je z kontextu jasné, který význam máme v danou chvíli na mysli. Pokud by mohlo dojít ke zmatení, budeme doplňovat termín přívlastkem, např. *příkaz jazyka Java* (*statement*) nebo *příkaz programu JShell* (*command*).

2.3.1 Vyloučení úryvku: **/drop**

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

2.3.2 Přehled aktivních úryvků: **/list**

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

Výpis 2.4: 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>
```

Při výpisu úryvků se dodržuje formátování, v jakém jste úryvky zadali. Když byl druhý úryvek ve výpisu 2.2 na straně 48 zadán rozprostřený do tří řádků s různým oddělením čísel a operátoru `+` mezerami, tak se tak také vypíše – viz výpis 2.4, 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 2.3).

Jestliže byl druhý úryvek vyloučen z aktivních (výpis 2.4, řádky 11 a 12), tak ho příkaz `/list` již nevypsal (řádky 16–19).

2.3.3 Přehled všech úryvků: `/list -all`

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

Za ním následuje seznam všech zadaných úryvků (viz výpis 2.5) včetně těch neaktivních, tj. těch, které jste smazali (naš úryvek 2 zobrazený na řádcích 14–16), při jejichž zadávání jste udělali chybu (úryvek `e1` na řádku 18 zadaný na řádku 1 ve výpisu 2.3), nebo které jste nahradili novější verzí (takový tam zatím není).

Výpis 2.5: 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). Zatím jsme zadali jediný chybný úryvek, když jsme zadávali dva výrazy na jednom řádku a neukončili první středníkem. Tento úryvek dostal **ID e1** a ve výpisu 2.5 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.

2.3.4 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 argument příkazu se zadává cesta k cílovému souboru, přičemž relativní cesta se odvozuje od složky, v níž jste program spustili. Chcete-li soubor uložit do aktuálního adresáře, stačí napsat pouze jeho název.

Uložené soubory jsou vnímány jako skripty a používá se pro ně přípona `jsh` (jako zkratka z názvu *JShell*). Je to sice jenom konvence, ale doporučujeme 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 zadali.

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

Zadáním příkazu `/save -all` (místo argumentu `-all` stačí zadat pouze `-a`) 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.

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

Zadáním příkazu `/save -history` (místo argumentu `-history` stačí zadat pouze `-h`) uložíte vše, co jste v průběhu seance zadali; nejenom úryvky, ale i příkazy, a to včetně posledního příkazu ukládajícího historii.

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.

Ve výpisu 2.6 jsou ukázky zadání všech tří způsobů záznamu dosavadní činnosti. Zkuste tyto příkazy zadat a pak si v nějakém editoru prohlédněte obsah vytvořených souborů `s02a_save.jsh`, `s02b_save_all.jsh` a `s02c_save_history.jsh` a zjistěte, jaké jsou mezi nimi rozdíly.

Výpis 2.6: Různé způsoby uložení dosavadní práce příkazem `/save`

```
1 jshell> /save s02a_save.jsh
2
3 jshell> /save -a s02b_save_all.jsh
4
5 jshell> /save -h s02c_save_history.jsh
6
7 jshell>
```

2.3.7 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 na stejném místě jako dávkový soubor, s jehož pomocí jste *JShell* spustili.

V některých kapitolách vám v poznámce o použitém projektu občas doporučí, abyste načtli nějaký soubor, v němž je připraven užitečný kód, který v dané kapitole použijeme. Budete-li chtít zkusit vše přesně tak, jak to ve výpisech zadáváme, budete tento kód potřebovat.

Prostředí *JShell* má připraveno několik standardních skriptů, které nám ušetří trochu psaní. My budeme používat skript `PRINTING`, který zjednoduší zápis žádostí o tisk. Budeme-li chtít v dalším textu něco vytisknout, tak v kapitole vložíme do prvního výpisu, který bude tyto tisky používat, příkaz

```
/open PRINTING
```

Ostatní výpisy v dané kapitole budou předpokládat, že již je zaveden.

Ve výpisu [2.7](#) jsem maličko předběhl a použil volání funkce a zadání textu, jejichž výklad nás teprve čeká. Ukazuje však, jaký má načtení skriptu **PRINTING** vliv.

Výpis 2.7: Vliv otevření skriptu **PRINTING**

```
1 jshell> println("Nazdar")
2 | Error:
3 | cannot find symbol
4 | symbol:   method println(java.lang.String)
5 | println("Nazdar")
6 | ^-----^
7
8 jshell> /open PRINTING
9
10 jshell> println("Nazdar")
11 Nazdar
12
13 jshell>
```

Na řádku **1** jsem se pokusil zavolat funkci `println()`, která měla vytisknout zadaný text (jak se zanedlouho dozvíte, texty se v *Javě* zadávají v uvozovkách). *JShell* však ohlásil chybu a tvrdil, že takovou funkci nezná.

Na řádku **8** jsem proto načtl skriptu **PRINTING** a zadal na řádku **10** příkaz znovu. Tentokrát se již text na následujícím řádku vytiskl.

2.3.8 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.

Zkuste po zadání příkazu **/exit** načíst úryvky z minulé seance příkazem **/open s02a_save.jsh** a potom ještě jednou zadat **/exit** a načíst kompletní konverzaci příkazem **/open s02c_save_history.jsh**.

2.3.9 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.

2.3.10 Znovuzavedení: `/reload -restore`

Příkaz `/reload -restore` (místo argumentu `-restore` stačí zadat pouze `-r`) 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.

Tento příkaz oceníte ve chvíli, kdy musíte od počítače odejít a počítač vypnout. Když se k němu druhý den vrátíte a spustíte *JShell*, vrátí vás tento příkaz do stavu, v němž jste počítač opouštěli.

Ve výpisu [2.8](#) jsem ukázal reakci počítače poté, co jsem po načtení „historického“ skriptu `s02c_save_history.js`, o němž jsem hovořil v předminulé pasáži, zadal příkaz `/reset` a poté příkazem `/reload -restore` znovu obnovil původní stav. Příkaz ale bude fungovat stejně i poté, co příkazem `/exit` *JShell* opustíte a zadáte jej jako první po příštím spuštění.

Výpis 2.8: Opětovné načtení předchozího stavu příkazem `/reload -restore`

```
1 jshell> /reset
2 | Resetting state.
3
4 jshell> /list
5
6 jshell> /reload -restore
7 | Restarting and restoring from previous state.
8 -: 6 + 5
9 -: 6 +
10 7+
11 9
12 -: $1 + $1 + $1
13 -: $1+$3;
14 -: $1+$4
15 -: /drop 2
16
17 jshell>
```

Na řádku **1** jsem se příkazem `/reset` vrátil do počátečního stavu. Jak už jsem ale řekl, mohl jsem stejně dobře *JShell* vypnout a opět spustit.

Příkaz `/list` na řádku **4** pouze dokazuje, že aktuálně není definován žádný úryvek.

Po zadání příkazu `/reload -restore` na řádku **6** se postupně načetly všechny úryvky včetně druhého, který byl po chvíli opět smazán. Kdybyste nyní znovu zadali příkaz `/list`, už by je vypsál.

2.3.11 Natavení startovního skriptu: `/set -start <file>`

Po spuštění programu *JShell* a po každém resetu se nejprve načte startovní skript. Jeho úryvky se ale příkazem `/list` nezobrazí. Zobrazit byste je mohli pouze příkazem

`/list -all` nebo `/list -start`. V tomto skriptu si můžete připravit sadu definic úryvků, které pak budete v průběhu seance používat.

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

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

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

Startovní skript můžeme nastavit i při spouštění programu *JShell* zadáním argumentu `--startup`, např.

```
JShell -v --startup Startovni_skript.jsh
```

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

2.3.12 Náповěda: `/?`

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

Náповědu 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 pod něj vypíše seznam možných pokračování.

Náповídací schopnosti prostředí umožní nezadávat příkazy v plném znění. Stačí napsat dostatečný počet znaků, aby prostředí zadávaný příkaz poznalo, a zadat jej. Toho už jsem v předchozím textu několikrát využil.

2.3.13 Spuštění editoru: `/edit`

Chcete-li provést nějakou větší úpravu na zadaných úryvcích, oceníte možnost aktivace editoru. K tomu slouží příkaz `/edit`. Zadáte-li jako argument seznam identifikátorů a/nebo ID, editor otevře označené úryvky. Nezádáte-li žádný argument, otevře všechny aktivní úryvky.

Zabudovaný editor je velice jednoduchá grafická aplikace. Budete-li chtít používat *JShell* pro své experimenty častěji, jistě oceníte možnost zadat vlastní editor prostřednictvím příkazu `/set editor`.

Abyste u editorů doporučujících současnou editaci několika souborů nemuseli po každém ukončení editace zavírat editor a mohli jste pouze zavřít editovaný soubor, zadejte před název editačního programu argument `-wait`. Já např. spouštím u sebe editor *PSPad* příkazem

```
/set editor -wait C:/_PGM/Office/PSPad/PSPad.exe
```

Po skončené editaci zadaného úryvku (případně zadaných úryvků, je-li jich více) a uložení daného souboru se stačí vrátit do okna *JShell* a stisknout dvakrát ENTER.

2.3.14 Nastavení běhového prostředí: */env*

Příkaz */env* slouží k zadání některých úprav parametrů běhového prostředí – např. kde všude se mohou nacházet používané části kódu. Příklad jeho použití předvedu v podkapitole [6.4.1 Začlenění knihovny robota Karla](#) na straně [108](#).

Podrobněji zde ale možnosti prostředí *JShell* rozvádět nebudu a odkážu vás na příručku [\[22\]](#), nápovědu a oficiální dokumentaci.

2.4 Shrnutí a soubory pro opakování

Zadávané příkazy i obsah výpisů se záznamem komunikace s interpretem probíhající v této kapitole najdete v souborech nazvaných *s02__Prostředí_JShell*. Připomínám, že čistý zdrojový text je ve složce *77_JSH* v souboru s příponou *jsh* a text doplněný o čísla řádků z výpisů je ve složce *77_JWD* v souboru s příponou *jdoc*.

V této kapitole byly navíc vytvořeny samostatné skripty *s02a_save.jsh*, *s02b_save_all.jsh* a *s02c_save_history.jsh*.

Kapitola 3

Zadávání hodnot



Co se v kapitole naučíte

Tato kapitola vás seznámí se základními datovými typy, vysvětlí vám, jaký je rozdíl mezi primitivním a objektovým datovým typem. Naučí vás zadávat hodnoty primitivních datových typů a textových řetězců. Současně vám představí některé všeobecně používané konstanty a vysvětlí, co to jsou literály.

3.1 Datové typy

Programy jsou určeny k tomu, aby zadaným způsobem zpracovávaly data. Aby program mohl data zpracovávat, musíme mu je nejprve zadat. Každý údaj má svůj datový typ, který ovlivňuje, co s ním program může dělat. Než proto začneme hovořit o zadávání údajů, musíme si povědět něco o datových typech.

Datový typ (nebo zkráceně jen typ) je označení pro trojici charakteristik specifikujících vlastnosti hodnot, které budu označovat jako *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),
- množinu operací, 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 programem, protože překladač má pro svá rozhodnutí tyto informace k dispozici, ale především se tím snižuje počet chyb. K tomuto problému se v budoucnu ještě několikrát vrátím.

Tvůrci Javy rozdělili všechny 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 zadaná část programu (přesněji zadaná metoda) žádnou hodnotu nedodá. Uvádím jej zde pro úplnost. Podrobněji vás s ním seznámím, až budeme probírat metody.