EDICE
myslíme v ...

PROGRAMOVACÍ JAZYK Swift

Příručka pro zájemce se znalostí jiných jazyků

- Z Javy, C#, C++ či Pythonu ke Swiftu bez zbytečné omáčky
- Protokoly a hodnotové typy místo hlubokého dědění tříd
- Přímé a nezkreslené srovnání architektonických konceptů Swiftu s nejpopulárnějšími objektovými jazyky
- Skrytá úskalí a pasti, které při přechodu na Swift zaskočí i zkušené programátory

RUDOLF PECINOVSKÝ

GRADA

GRADA

EDICE
myslíme v ...

RUDOLF PECINOVSKÝ

PROGRAMOVACÍ JAZYK Swift

Příručka pro zájemce se znalostí jiných jazyků

GRADA
Publishing

Rudolf Pecinovský

Programovací jazyk Swift
Příručka pro zájemce se znalostí jiných jazyků

Vydala Grada Publishing, a.s.
U Průhonu 22, Praha 7
obchod@grada.cz, www.grada.cz
tel.: +420 234 264 401
jako svou 10 636. publikaci

Odpovědný redaktor Petr Somogyi
Grafická úprava a sazba Rudolf Pecinovský
Počet stran 144
První vydání, Praha 2026
Vytiskly Tiskárny Havlíčkův Brod a.s.

© Grada Publishing, a.s., 2026
Cover Design © Grada Publishing, a. s., 2026
Cover Photo © Depositphotos/Bushko

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. 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.

Názvy produktů, firem apod. použité v knize mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

ISBN 978-80-271-8543-6 (pdf)
ISBN 978-80-271-8493-4 (print)

Stručný obsah

Kapitola 1 **Představení jazyka a pracovní prostředí** 13
Kapitola 2 Základy *Swiftu* 16
Kapitola 3 Algoritmické konstrukce 46
Kapitola 4 **Koncepce datových typů jazyka Swift** 64
Kapitola 5 **Vestavěné datové typy a** správa dat 94
Kapitola 6 **Pokročilé progr**amové konstrukce 109
Kapitola 7 **Vybrané pokročilé koncepty** 127
Kapitola 8 Co ve Swiftu nenajdete 132

Podrobný obsah

Kapitola 1 **Představení jazyka a pracovní prostředí** 13
1.1 Historie a filozofie jazyka Swift 13
1.1.1 **Daň za bezpečnost a „uke**canost“ 14
1.1.2 Zlatá klec ekosystému Apple 14
1.2 **Vývojové prostředí a nástroje** 14
1.2.1 Vlastníte-li Mac (Xcode a Playground) 15
1.2.2 Nevlastníte-li Mac (Online nástroje a VS Code) 15
Kapitola 2 Základy *Swiftu* 16
2.1 Syntaxe 16
Absence středníků 16
Závorky a bloky kódu 16
Všudypřítomná typová inference 17
Deklarace versus definice 17
2.2 Identifikátory 17
2.2.1 Konvence úvodních podtržítek 18
Jedno podtržítko (gentlemanská dohoda) 18
Dvě podtržítka (rezervováno systémem) 19
2.3 Základní prvky programu 19
2.3.1 **Proměnné** 19
2.3.2 **Příkazy a výrazy** 20
2.3.3 **Komentáře** 20
Tvorba dokumentace 21
Navigace v kódu pomocí značek MARK 21
2.4 Datové typy 21
2.4.1 Trocha teorie 22
2.4.2 Deklarace typu a typová inference 22
Základní datové typy 23
Specializované číselné datové typy 23
2.4.3 Literály a jejich specifika 24
Číselné soustavy a oddělovače 24
Absence typových přípon 24
Znaky versus stringy 25
Absence komplexních čísel 25
2.4.4 **Interpolace stringů** 25
2.4.5 N-tice 26
2.5 Operátory 26
2.5.1 Základní terminologie 27
2.5.2 **Upozornění pro programátory se zkušenostmi z jiných jazyků** 27
Význam termínů operátor a operace 27
2.5.3 **Pravidla vyhodnocování výrazů** 28
Priorita operátorů (operator priority) 28
Asociativita operátorů (operator associativity) 28
Přehled operátorů 29
2.5.4 **Přiřazovací operátor** 30
2.5.5 **Složené přiřazení (+=, -=, *=, /=, ...)** 30

2.5.6 Aritmetické operátory ... 31
Sčítání a násobení stringů ... 31
Odlišné chování operátoru `%` ... 31
Absence operátoru mocnění ... 31
Operátory pro práci s přetečením ... 31
2.5.7 Logické operátory ... 32
2.5.8 Bitové operátory ... 33
Specifikum pro vývojáře v jazyce Python ... 33
2.5.9 Porovnávací operátory ... 34
Zákaz zřetězení operátorů ... 34
Rovnost (`==`) versus Identita (`===`) ... 34
Porovnávání n-tic ... 35
2.5.10 **Operátory rozsahů (intervalů)** ... 35
Uzavřený rozsah (`...`) ... 35
Polootevřený rozsah (`..<`) ... 35
Jednostranné rozsahy ... 35
2.5.11 **Ternární podmíněný operátor** ... 36
2.6 Funkce ... 37
2.6.1 Konzole versus panel náhledu ... 37
2.6.2 Parametry a argumenty ... 37
Název argumentu a název parametru ... 38
Vynechání názvů argumentů ... 38
Funkce s více parametry ... 39
Výchozí hodnoty parametrů ... 39
2.6.3 Podrobnosti o funkci `print()` ... 40
Oddělovač – `separator` ... 40
Ukončovač – `terminator` ... 40
2.6.4 Návratové hodnoty z funkcí ... 41
Vracení jedné hodnoty ... 41
Funkce bez návratové hodnoty (typ `Void`) ... 41
Vracení n-tic z funkcí ... 42
2.6.5 **Lokální a globální proměnné** ... 42
Lokální proměnné ... 42
Globální proměnné ... 42
2.6.6 Odložený kód (blok `defer`) ... 43
Vyhodnocování v opačném pořadí (LIFO) ... 43
2.7 Syntaktické zvláštnosti Swiftu ... 44
2.7.1 **Symetrie mezer u binárních operátorů** ... 44
Operátor na konci řádku ... 44
2.7.2 **Zákaz mezer u operátoru tečka (`.`)** ... 45
Kapitola 3 Algoritmické konstrukce ... 46
3.1 **Samostatný blok kódu (příkaz `do`)** ... 46
3.2 **Podmíněný příkaz** ... 47
3.2.1 **Jednoduchý podmíněný příkaz – příkaz `if`** ... 47
3.2.2 **Úplný podmíněný příkaz – příkaz `if…else`** ... 48
3.2.3 **Rozšířený podmíněný příkaz – příkaz `if…else if`** ... 48
3.2.4 Konstrukce `if` použitá jako výraz ... 49
3.3 **Ochranný příkaz – příkaz `guard`** ... 50
3.4 **Přepínač – příkaz `switch`** ... 51
3.4.1 **Klasické použití přepínače** ... 52
3.4.2 **Porovnávání vzorů** ... 53
Vázání hodnot (value binding) ... 53
Dodatečné podmínky pomocí `where` ... 54
3.4.3 Explicitní propadnutí – **klíčové slovo `fallthrough`** ... 54
3.4.4 **Přepínací výraz** – konstrukce **`switch`** použitá jako výraz ... 55

3.5 Cykly ... 56
3.5.1 **Cyklus s počáteční podmínkou** – cyklus **`while`** ... 56
3.5.2 Cyklus s koncovou podmínkou – cyklus **`repeat-while`** ... 57
3.5.3 Cyklus **s podmínkou uprostřed** – **příkaz `break`** ... 57
3.5.4 **Předčasné ukončení těla cyklu** – **příkaz `continue`** ... 58
3.5.5 Cyklus s parametrem – cyklus **`for…in`** ... 59
Procházení číselných rozsahů ... 59
Ignorování hodnoty (zástupný znak **`_`**) ... 60
Procházení polí a slovníků ... 60
Funkce **`stride`** pro iteraci s krokem ... 61
3.6 **Příkazy `break`** a **`continue`** s **návěštím** ... 62
Kapitola 4 **Koncepce datových typů jazyka Swift** ... 64
4.1 **Teoretická předehra** ... 64
Datové typy ... 64
4.1.1 Instance datového typu ... 65
4.1.2 Rozhraní versus implementace ... 65
Rozhraní (interface) ... 65
Signatura ... 65
Kontrakt ... 65
Implementace ... 65
Zapouzdření ... 66
4.1.3 **Druhy datových typů Swiftu** ... 66
Protokoly ... 66
Hodnotové typy (value types) ... 66
Odkazové typy (reference types) ... 67
Funkční typy ... 67
Metatypy ... 67
Zobecněný typ **`Any`** ... 67
4.1.4 **Tři druhy dědění** ... 68
Nativní dědění ... 68
Dědění rozhraní ... 68
Dědění implementace ... 68
4.1.5 **Dva způsoby dědění** ... 69
Jmenovité dědění ... 69
Strukturální dědění ... 69
4.2 Funkce ... 69
4.2.1 Funkce jako datový typ ... 70
4.2.2 Signatura funkcí ... 70
Jednotlivé části signatury funkcí ... 71
Podrobnosti o položce parametry ... 71
4.2.3 Modifikátor **`inout`** ... 72
4.3 Protokoly ... 73
4.3.1 **Deklarace protokolu a značkovací protokoly** ... 73
4.3.2 **Dědičnost protokolů** ... 74
4.3.3 Požadavky na metody ... 74
4.3.4 Požadavky na vlastnosti ... 75
4.4 Struktury ... 76
4.4.1 Definice struktur a jejich vlastností ... 76
4.4.2 **Vytváření instancí a jejich inicializace** ... 77
Inicializace a implicitní inicializátor (memberwise initializer) ... 77
4.4.3 **Vlastní inicializátor a klíčové slovo `self`** ... 78
Více inicializátorů ... 78
4.4.4 **Vypočítávané vla**stnosti ... 79
Vlastnosti určené pouze pro čtení (read-only) ... 80
4.4.5 Pozorovatelé vlastností ... 80
4.4.6 Metody ... 81

Modifikátor **mutating** ... 82
4.4.7 Interní datové typy ... 82
4.5 **Výčtové typy** ... 83
4.5.1 Srovnání s jinými jazyky ... 83
4.5.2 **Zkrácená tečková notace** ... 84
4.5.3 **Použití výčtového typu** ... 84
4.6 **Třídy** ... 85
4.6.1 **Definice třídy a inicializace** ... 85
4.6.2 **Odkazová povaha instancí tříd** ... 86
4.6.3 **Dědění implementace** ... 87
Zápis dědičnosti a přebíjení metod ... 87
Dvoufázová inicializace (Two-Phase Initialization) ... 87
4.6.4 Kontrola typu (operátor **is**) ... 88
4.6.5 Deinicializace ... 89
4.7 Metatypy ... 90
4.8 **Zobecněné datové typy** ... 91
AnyObject ... 91
AnyClass ... 91
Any ... 91
4.9 **Opakní typy a klíčové slovo some** ... 92
Kapitola 5 **Vestavěné datové typy a** správa dat ... 94
5.1 Volitelné typy, typ **Optional** ... 94
5.1.1 Absence hodnoty a symbol **nil** ... 95
5.1.2 **Bezpečné rozbalení** (optional binding) ... 95
Konstrukce **guard let** ... 96
5.1.3 **Slučování s nil** (operátor **??**) ... 97
5.1.4 Vynucené rozbalení (operátor **!**) ... 97
5.2 Stringy ... 98
5.2.1 Struktura **String** a typ **Character** ... 98
5.2.2 **Vnitřní implementace stringů** ... 99
Optimalizace pro krátké stringy (SSO) ... 99
Copy-on-Write (CoW) ... 99
Rychlé porovnávání ... 100
5.2.3 **Zrádné indexování stringů** ... 100
5.2.4 **Vykrajování stringů** (slicing) a typ **Substring** ... 101
5.3 Kolekce (kontejnery) ... 101
5.3.1 **Společné vlastnosti kolekcí v jazyku Swift** ... 102
5.3.2 Pole (**Array**) ... 102
5.3.3 Množiny (**Set**) ... 103
5.3.4 Slovníky (**Dictionary**) ... 104
5.4 **Správa paměti (Model ARC)** ... 104
5.4.1 Princip fungování ARC ... 105
5.4.2 **Zacyklení silných odkazů** (strong reference cycles) ... 105
5.4.3 Slabé odkazy (**weak**) ... 106
5.4.4 **Nevlastněné odkazy (unowned)** ... 107
Kapitola 6 **Pokročilé programové konstrukce** ... 109
6.1 **Uzávěry** (closures) ... 109
6.1.1 Syntaxe a zkracování zápisu ... 110
6.1.2 **Ukládané uzávěry (@escaping)** ... 111
6.1.3 Atribut **@autoclosure** ... 111
6.2 **Rozšíření (extension)** ... 112
6.2.1 **Rozšiřování vlastností a metod** ... 112
6.2.2 **Modifikační metody v rozšířeních** ... 113
6.2.3 **Retroaktivní adoptování protokolů** ... 113

6.3 Generické datové typy a podprogramy ... 114
6.3.1 Generické datové typy a funkce v jiných jazycích ... 115
Java ... 115
C# ... 115
C++ ... 115
Python ... 115
Swift ... 116
6.3.2 Generické funkce ... 116
6.3.3 **Generické struktury a třídy** ... 116
6.3.4 Typová omezení (type constraints) ... 117
6.3.5 **Přidružené typy v protokolech** (associated types) ... 117
6.4 **Ošetření chyb** (error handling) ... 118
6.4.1 **Ošetření chyb v jiných jazycích versus** *Swift* ... 118
Řešení jazyka Swift ... 118
6.4.2 Protokol `Error` a vyhazování chyb ... 119
6.4.3 Zachycení chyb (blok `do…catch`) ... 120
6.4.4 Volitelné ošetření chyb (`try?` a `try!`) ... 120
6.5 **Řízení přístupu** (access control) ... 121
6.5.1 **Řízení přístupu v jiných jazycích versus S**wift ... 121
6.5.2 Struktura programu: jazyky V4 versus Swift ... 122
6.5.3 **Přístupové úrovně ve Swiftu** ... 122
6.5.4 Praktické použití v kódu ... 123
6.6 Obálky vlastností (property wrappers) ... 123
6.6.1 Definice a použití obálky vlastností ... 124
6.6.2 Projektovaná hodnota (projected value) ... 124
6.7 **Tvůrci výsledků** (result builders) ... 125
Kapitola 7 **Vybrané pokročilé koncepty** ... 127
7.1 Moderní asynchronní programování ... 127
7.2 Swift makra (metaprogramování) ... 128
7.2.1 Princip fungování ... 129
7.2.2 Ukázka použití ... 129
7.3 **Nízkoúrovňové programování a práce s pamětí** ... 130
7.3.1 **Rodina typů `Unsafe`** ... 130
Kapitola 8 Co ve Swiftu nenajdete ... 132
8.1 **Vícenásobné dědění implementace** ... 132
8.1.1 **Důvody odmítnutí vícenásobného dědění tříd** ... 133
8.1.2 **Řešení ve Swiftu: protokoly s výchozí implementací** ... 133
8.1.3 **Nevýhody a skrytá úskalí swiftového řešení** ... 133
Nemožnost dědit stav ... 133
Záludnosti statického vs. dynamického odesílání zpráv ... 134
Konflikty při kolizi protokolů ... 135
8.2 **Abstraktní třídy** ... 135
8.2.1 **Proč Swift abstraktní třídy neobsa**huje ... 135
8.2.2 **Jak se absence abstraktních tříd řeší v praxi** ... 136
Přístup A: Náhrada protokolem (bezstavový model) ... 136
Přístup B: Standardní třída s běhovými chybami (stavový model) ... 136
8.2.3 **Důsledky pro návrh aplikací** ... 137
8.3 **Přístupový modifikátor protected** ... 137
8.3.1 **Důvody vynechání a s tím spojená diskuse** ... 138
8.3.2 **Jak se absence protected řeší ve Swiftu** ... 138
Náhrada pomocí private / fileprivate ... 138
Náhrada pomocí internal ... 138
8.3.3 **Důsledky pro návrh aplikací** ... 139
8.4 Implicitní typové konverze ... 139
8.4.1 **Důvody odmítnutí implicitních konverzí** ... 139
8.4.2 Jak se absence implicitních konverzí projevuje v praxi ... 140

8.4.3 **Důs**ledky pro návrh aplikací ... 140
8.5 Absence bloku **`finally`** ... 141
8.5.1 **Proč Swift opustil koncept `finally`** ... 141
8.5.2 **Rozdíly v myšlení pro vývojáře z velké čtyřky** ... 141
Zápis na začátku místo konce ... 141
Past s lokálním rozsahem platnosti (Scope) ... 142
Přirozená hierarchie úklidu (LIFO) ... 142
8.6 **Větve `else` u cyklů a ošetřování c**hyb ... 142
8.7 Klasický garbage collector ... 143
8.7.1 **Závěr: Přesun odpovědnosti** ... 143

Úvod

Otevíráte knížku, jež vás chce seznámit s moderním, objektově orientovaným jazykem *Swift* z dílny společnosti *Apple*. Je určena pro programátory, kteří již mají základní znalosti a zkušenosti s nějakým jiným objektovým programovacím jazykem.

V době přípravy této knihy patřily podle serveru TIOBE mezi nejpopulárnější objektově orientované jazyky, které měly v žebříčku oblíbenosti zastoupení alespoň 5 %, jazyky *Python* (22,5 %), *Java* (8,7 %), *C++* (8,7 %) a *C#* (7,4 %). Zastoupení dalších jazyků už bylo pod 3 %.

Uvedl jsem stav z ledna 2026. Přesné procentuální zastoupení jednotlivých jazyků se v průběhu času mění, ale globálně lze říci, že tato čtveřice si dlouhodobě udržuje dominanci. Tuto skupinu jazyků proto budu v dalším textu označovat jako „*velká čtyřka*“, nebo na ně budu odkazovat pouze prostřednictvím zkratky V4.

Protože očekávám, že čtenáři této příručky budou mít nejpravděpodobněji zkušenosti právě s některým z těchto jazyků, budu při výkladu průběžně upozorňovat na rozdíly mezi jazykem *Swift* a jazyky z V4. Naštěstí je všechny znám.

Dopředu se musím omluvit, že *Swift* není jazyk, v němž bych napsal tisíce řádků kódu. O napsání této příručky jsem byl požádán poté, co se nakladatelství *Grada* rozhodlo vydat příručku, která u čtenářů předpokládala znalost jazyka *Swift*, a ve svém portfoliu doposud žádnou takovou neměla.

Ponořil jsem se do dokumentace a pokusil se vytvořit příručku, kterou držíte v ruce. Mé znalosti OOP a jazyků z V4 mi umožnily se na *Swift* podívat jako programátor zkoumající nový jazyk a porovnat jej s nejpoužívanějšími jazyky.

Možná si mnozí z vás řeknou, že jsem ji mohl nechat za sebe napsat umělou inteligencí. To ale mohou tvrdit jenom ti, kdo něco podobného nikdy nezkusili. Použití AI zrychlí získávání informací, ale má-li být rukopis ve tvaru, který je seriózní nakladatelství ochotno vydat, nemůžete to nechat na AI. Všechny informace musíte průběžně kontrolovat a ověřovat.

Nevěřili byste, kolik se do těch informací od AI dostane nejrůznějšího marketingového balastu firmy *Apple*, v němž vás budou přesvědčovat, že právě jejich řešení je to nejlepší na světě. Boj s marketingovými prohlášeními příruček *Applu* a seriózní porovnání jazyka *Swift* s jazyky z V4 bylo opravdu vyčerpávající.

Je to moje první kniha o *Swiftu*. Takže se na závěr ještě jednou omlouvám za všechny nedokonalosti předkládané příručky a přeju vám, aby vám pomohla v rozšiřování obzorů.

Kapitola 1
Představení jazyka a pracovní prostředí

Začínáte-li se učit nový programovací jazyk, je nesmírně užitečné znát nejen jeho syntaxi, ale především filozofii, se kterou byl navržen. Syntaxe je totiž vždy jen důsledkem architektonických rozhodnutí jeho tvůrců. V této kapitole se společně stručně ohlédneme za historií jazyka *Swift*, vysvětlíme si důvody jeho specifické přísnosti a ukážeme si nástroje, ve kterých budeme psát a zkoušet první programy.

1.1 Historie a filozofie jazyka Swift

Když společnost *Apple* v roce 2014 jazyk *Swift* představila, ambice jeho tvůrců nebyly vůbec malé. Nechtěli pouze nahradit letitý *Objective-C* při vývoji mobilních aplikací. Chtěli vytvořit univerzální, moderní a extrémně bezpečný systémový jazyk, který by ovládl i svět obrovských serverových aplikací a stal se přímou konkurencí pro *C++* či *Javu*. V roce 2015 byl *Swift* uvolněn jako open-source a ve spolupráci s *IBM* do něj byly investovány obrovské prostředky právě pro expanzi na servery.

Tento velkolepý plán však poměrně rychle ztroskotal na tvrdé vývojářské realitě. Jazyk totiž v prvních letech trpěl několika inženýrskými neduhy.

Prvním z nich byla absence binární a zdrojové stability (takzvaná ABI stabilita). Překladač jazyka *Swift* se mezi verzemi 1 až 4 neustále a drasticky měnil. S každou novou verzí přestal starý kód fungovat a programátoři byli nuceni jej pracně přepisovat. Pro korporace spravující backendy o milionech řádků to představovalo neakceptovatelné riziko.

Druhým důvodem byla poněkud falešná multiplatformnost. Ačkoliv se jádro jazyka dalo na Linuxu přeložit, klíčové systémové knihovny (jako `Foundation`) byly uvnitř operačního systému *macOS* historicky pevně svázány s běhovým prostředím *Objective-C*. Vývoj pro jiné platformy než *Apple* tak dlouhé roky znamenal boj s chybějícími nebo nedokončenými knihovnami.

1.1.1 Daň za bezpečnost a „ukecanost“

Největším úskalím, které odradilo mnoho vývojářů z jiných ekosystémů, se však paradoxně stala největší chlouba jazyka – jeho absolutní důraz na typovou a paměťovou bezpečnost.

Aby *Swift* dosáhl své pověstné spolehlivosti bez nutnosti používat pomalý systém automatické správy paměti (*garbage collector*), stal se jazykem extrémně pedantským. Úkony, které dynamické jazyky vyřeší na jediném řádku nebo ke kterým benevolentní jazyky z rodiny *C* přistupují prostřednictvím tichých implicitních konverzí, vyžadují ve *Swiftu* explicitní přístup.

Programátor je neustále nucen řešit rozbalování volitelných typů, ručně konvertovat základní číselné typy a obalovat sebemenší nejistotu bloky na ošetření chyb. Pro backendové vývojáře, zvyklé na rychlý pragmatismus jazyků jako *Go* nebo *Python*, byla tato byrokratická „ukecanost“ jazyka příliš velkou překážkou. *IBM* se tak ze serverového vývoje stáhlo a *Swift* si místo v cloudu nevybojoval.

1.1.2 Zlatá klec ekosystému Apple

Tento turbulentní vývoj vedl dokonce k tomu, že se *Swift* na čas propadl ve statistikách oblíbenosti (např. v indexu TIOBE) a předběhl jej samotný *Objective-C*, jelikož se technologické firmy zdráhaly stabilní kód do nového jazyka přepisovat.

Kde je tedy *Swift* dnes? Nalezl své definitivní místo. Serverový a multiplatformní svět sice přenechal jiným, ale na domovských platformách společnosti *Apple* se z něj stal absolutní a neohrožený vládce.

Od verze 5 (kdy jazyk konečně dosáhl ABI stability) a s příchodem moderních deklarativních frameworků dozrál v mimořádně robustní nástroj. Výsledkem je takzvaná **zlatá klec**.

Chcete-li programovat pro platformy *iPhone*, *Mac*, *Apple Watch* či *Vision Pro*, je *Swift* nástrojem, který vám společnost *Apple* v podstatě vnutí. Na svých platformách totiž vývoj v jiných jazycích buď aktivně ztěžuje (pomocí omezeného přístupu k systémovým rozhraním pro multiplatformní frameworky), nebo jej u některých technologií přímo blokuje.

K psaní čehokoliv mimo tento ekosystém jej však pravděpodobně nevyužijete. A právě s tímto realistickým pohledem na jeho pedantskou povahu k němu musíme přistupovat.

1.2 Vývojové prostředí a nástroje

Chceme-li se učit nový jazyk, potřebujeme prostor pro experimenty. Samotný *Swift* pro vývoj aplikací vyžaduje komplexní integrované vývojové prostředí, avšak pro zkoušení základní syntaxe si vystačíme s mnohem lehčími nástroji. Postup pro přípravu prostředí se liší podle toho, jaký hardware máte k dispozici.

1.2.1 Vlastníte-li Mac (Xcode a Playground)

Pro plnohodnotný vývoj v ekosystému *Apple* je počítač *Mac* nezbytností. Standardním nástrojem je vývojové prostředí **Xcode**, které lze zdarma stáhnout z obchodu *Mac App Store*.

Prostředí *Xcode* pracuje ve dvou základních režimech. První režim slouží k vývoji reálných aplikací, což vyžaduje zakládání složitých projektů a sestavování celého programu.

Pro výuku syntaxe, kterou budeme procházet v kapitolách 2 až 6, je mnohem vhodnější druhý režim – takzvaný **Playground** (hřiště). Jedná se o interaktivní laboratoř, kde kód nepíšete do složitého projektu, ale do jednoduchého skriptu. Překladač tento skript průběžně vyhodnocuje a výsledky výpočtů i případné chyby zobrazuje okamžitě v postranním panelu, bez nutnosti ručního překladu.

Nové hřiště vytvoříte po spuštění *Xcode* přes nabídku `File → New → Playground`, kde stačí zvolit prázdnou šablonu (`Blank`) a soubor si uložit na disk.

1.2.2 Nevlastníte-li Mac (Online nástroje a VS Code)

Nemáte-li počítač *Mac* a chcete si jazyk zatím pouze osahat, nejste zcela odstřiženi. Samotné jádro jazyka *Swift* je open-source a existuje v nativních verzích pro systémy *Linux* i *Windows*.

Pro zkoušení všech příkladů v této knize vám dokonale poslouží dostupné online překladače, které běží přímo ve webovém prohlížeči. Nejznámějším a na míru ušitým prostředím je web *SwiftFiddle* (swiftfiddle.com).

Pro serióznější psaní kódu mimo platformu *Apple* lze rovněž využít prostředí *Visual Studio Code* od Microsoftu, do něhož existuje oficiální rozšiřující balíček pro jazyk *Swift*.

Je však nutné si uvědomit jedno zásadní omezení. V těchto ne-jablečných prostředích máte k dispozici čistý jazyk (např. třídy, protokoly, cykly a operátory), ale chybí v nich všechny knihovny určené pro tvorbu uživatelského rozhraní.

Tato kniha se zabývá výhradně samotným jazykem *Swift*, takže s online nástroji nebo systémy *Windows*/*Linux* si bez problémů vystačíte až do jejího konce. Jakmile se však po jejím dočtení rozhodnete posunout dál a začít vyvíjet skutečné grafické aplikace pro zařízení *Apple*, bude pro vás pořízení počítače *Mac* nevyhnutelností.

Kapitola 2
Základy Swiftu

V této kapitole se seznámíme se základními stavebními kameny jazyka *Swift*, od deklarace proměnných přes operátory až po funkce. Přestože vám jeho syntaxe bude na první pohled připadat pravděpodobně velmi povědomá (alespoň přecházíte-li z jazyků se syntaxí odvozenou z jazyka *C*), velmi rychle narazíte na zásadní odlišnosti.

Ve výkladu se proto zaměříme především na to, v čem je *Swift* unikátní, jaká nekompromisní bezpečnostní pravidla zavádí a na co si vývojáři zvyklí na benevolentnější jazyky musí dát od samého začátku největší pozor.

2.1 Syntaxe

Jak jistě víte, syntaxe představuje soubor pravidel určujících, jak zapisovat nejrůznější konstrukce jazyka, aby tomu překladač správně porozuměl. Pojďme se na úvod zaměřit na nejvýraznější vizuální a strukturální odlišnosti, kterých si při psaní kódu ve *Swiftu* okamžitě všimnete. Trošku při tom předběhnu výklad, ale chtěl jsem, abyste měli globální principiální odchylky uvedené pohromadě.

Absence středníků

Jazyk *Swift* sice syntakticky patří do rodiny jazyků *C*, ale na rozdíl od těch starších už netrvá na ukončování příkazů středníkem. Konec příkazu překladač zpravidla bezpečně pozná podle konce řádku. Středníky musíme použít výhradně v situaci, kdy potřebujeme na jeden řádek zapsat více příkazů za sebou. V opačném případě se jejich použití považuje za prohřešek proti idiomatickému zápisu.

Závorky a bloky kódu

Píšeme-li podmínku pro příkazy `if`, `while` nebo `switch`, ve *Swiftu* se kolem logického výrazu nepíší kulaté závorky. Překladač sice jejich použití stráví, ale vývojové prostředí vás na ně vzápětí upozorní jako na zbytečný balast.

Zcela nekompromisní je však *Swift* u složených závorek `{}`. Tělo podmínky či cyklu jimi musí být striktně ohraničeno, a to i v případě, obsahuje-li pouze jediný příkaz.

Swift tímto přístupem elegantně eliminuje celou třídu chyb spojených se zavádějícím odsazením kódu.

Všudypřítomná typová inference

Přestože je *Swift* přísně staticky typovaný jazyk (podobně jako *Java* nebo *C++*), idiomatický kód působí velmi odlehčeně. Jazyk totiž masivně spoléhá na automatické odvozování typů (*type inference*). Při definici proměnné tak většinou neuvádíme její typ, přiřadíme-li jí rovnou výchozí hodnotu, z níž si překladač vše potřebné odvodí sám.

Deklarace versus definice

V některých jazycích je zásadní rozdíl mezi deklarací a definicí. Deklarace pouze oznamuje existenci a typ proměnné bez alokace paměti (což umožňuje její sdílení napříč moduly), zatímco definice jí reálně vyhradí místo v paměti. Ve *Swiftu* se tento historický rozdíl téměř stírá. Zápis `var x: Int` je ve *Swiftu* vnímán jako deklarace i definice současně, neboť okamžitě rezervuje potřebnou paměť, i když do ní konkrétní hodnotu přiřadíme až později.

2.2 Identifikátory

Každá entita, s níž chceme v programu pracovat, musí mít svůj název, anebo musíme mít možnost se k ní přes pojmenované entity dostat. Názvy entit v programu označujeme jako **identifikátor**y, protože jejich prostřednictvím tyto entity identifikujeme.

Ovšem oficiální pravidla jazyka *Swift* pro tvorbu identifikátorů psal zřejmě nějaký patlal ráno po flámu. Protože v první větě oznamuje, že na první pozici mohou být alfanumerické znaky ze sady *Unicode*, aby v další větě oznámil, že od druhého znaku tam mohou být i číslice. Jenomže číslice jako alfanumerický znak smí teoreticky být i na první pozici.

> ***Poznámka:***
> *Unicode je mezinárodní standard, který umožňuje jednotně reprezentovat znaky všech písem, včetně speciálních symbolů a emoji.*

Tolik definice. Touha autorů vytvořit jazyk s většími možnostmi než konkurence, jazyk, který v identifikátorech povolí i emoji, vedla autory k naprosto nekonzistentní definici identifikátorů.

Pojďme se proto podívat, jak se na to dívá překladač. Ten (na rozdíl například od *Javy*) nedělí znaky podle kategorií *Unicode* na písmena, číslice a další kategorie, ale za znaky, které se nesmějí nacházet na první pozici, považuje pouze číslice ze sady ASCII.

Jenže v sadě *Unicode* je asi 60 sad znaků, které patří do kategorie *Desítkové číslice*, takže např. identifikátor `0123456789` obsahující znaky s kódem `U+1D7F6` až `U+1D7FF` (znaky ze sady *Monospace digits*) je platný identifikátor *Swiftu*.

Pojďme se ale dohodnout, že tyto možnosti využívat nebudeme a v identifikátorech se omezíme, jak bývá mezi programátory zvykem, na znaky ze sady ASCII. Pak platí pravidlo, že identifikátor může být jakákoliv posloupnost alfanumerických znaků a znaků podtržení nezačínající číslicí a nepatřící mezi klíčová slova.

V otázce klíčových slov se *Swift* přidal k současnému trendu definovat pevnou sadu a sadu, kterou může s příchodem nových funkcionalit rozšiřovat. Rozlišuje proto:

- **Vyhrazená klíčová slova** (*reserved keywords* nebo prostě jen *keywords*) jsou uvedena v tabulce 2.1
- **Kontextová klíčová slova** (*contextual keywords*) jsou uvedena v tabulce 2.2. Jejich počet se může s příchodem dalších verzí jazyka dále zvětšovat.

Tabulka 2.1: *Vyhrazená klíčová jazyka Swift*

`Any`	`as`	`associatedtype`	`break`	`case`	`catch`
`class`	`continue`	`default`	`defer`	`deinit`	`do`
`else`	`enum`	`extension`	`fallthrough`	`false`	`fileprivate`
`for`	`func`	`guard`	`if`	`import`	`in`
`init`	`inout`	`internal`	`is`	`let`	`nil`
`operator`	`private`	`protocol`	`public`	`repeat`	`rethrows`
`return`	`self`	`Self`	`static`	`struct`	`subscript`
`super`	`switch`	`throw`	`throws`	`true`	`try`
`typealias`	`var`	`where`	`while`		

Tabulka 2.2: *Kontextová klíčová jazyka Swift*

`actor`	`any`	`async`	`await`	`convenience`	`didSet`
`dynamic`	`final`	`get`	`indirect`	`infix`	`isolated`
`lazy`	`left`	`macro`	`mutating`	`nonisolated`	`nonmutating`
`open`	`optional`	`override`	`postfix`	`prefix`	`Protocol`
`required`	`right`	`set`	`some`	`Type`	`unowned`
`weak`	`willSet`				

2.2.1 Konvence úvodních podtržítek

Při čtení cizího kódu nebo při procházení rozhraní systémových knihoven můžeme často narazit na identifikátory začínající jedním nebo dvěma podtržítky. Ačkoliv to základní syntaktická pravidla jazyka technicky umožňují, s používáním těchto znaků na počátku názvu je spojena silná programátorská konvence.

Jedno podtržítko (gentlemanská dohoda)

Začíná-li název jedním podtržítkem (například `_internalState`), jedná se o vizuální signál, že daná entita představuje vnitřní detail implementace. Tento přístup důvěrně znají vývojáři z jazyka *Python*.

Ve *Swiftu* se s ním setkáme nejčastěji v situacích, kdy musí být vlastnost nebo metoda z technických důvodů překladače označena jako veřejná (například z důvodu optimalizace nebo splnění podmínek protokolu), ale autoři knihovny si nepřejí, aby ji ostatní programátoři zvenčí přímo používali.

Přistoupí-li náš kód k takové entitě, vystavujeme se riziku, že při další aktualizaci dané knihovny její autoři tento detail změní a naše aplikace se bez varování rozbije.

Dvě podtržítka (rezervováno systémem)

Identifikátory začínající dvěma podtržítky (například `__builtin`) si jazyk *Swift* striktně rezervuje pro vnitřní potřeby svého překladače a pro vývoj své standardní knihovny. Tento ochranný princip přesně odpovídá pravidlům z jazyků *C* a C++.

Vytvoříme-li si ve vlastním programu takto pojmenovanou proměnnou, může kód v danou chvíli fungovat, ale vystavujeme se velkému riziku. Některá z budoucích verzí překladače totiž může zavést stejně pojmenovaný systémový prvek a náš program následkem kolize jmen zcela přestane fungovat.

Něco podobného znají i programátoři v *Pythonu* v případě identifikátorů typu dunder (*double underscore*), které si systém zamlouvá pro sebe.

2.3 Základní prvky programu

Než se v následující podkapitole ponoříme do podrobného zkoumání datových typů, musíme si představit elementární stavební kameny, ze kterých se kód ve *Swiftu* skládá. V této podkapitole si ujasníme, jakým způsobem se v paměti bezpečně uchovávají data, vymezíme si rozdíl mezi výrazem a příkazem a ukážeme si, jak do zdrojového textu vpisovat poznámky a jak z nich následně generovat plnohodnotnou dokumentaci.

2.3.1 Proměnné

Proměnná je pojmenované místo v paměti počítače, do kterého může aplikace uložit nějakou hodnotu. Tuto hodnotu lze kdykoliv číst nebo měnit, aniž bychom museli znát přesnou adresu v paměti.

Proměnnou deklarujeme tak, že zapíšeme klíčové slovo `var`, kterým sdělíme překladači, že se jedná o deklaraci proměnné, a za ním její název – například `age`.

Chceme-li vytvořenou proměnnou hned inicializovat, můžeme za název zapsat rovnítko následované přiřazovanou počáteční hodnotou. Následující příkaz vytvoří proměnnou `age` s počáteční hodnotou `8`.

```
var age = 8
```

Proměnné deklarované klíčovým slovem `var` je možné měnit na jiné hodnoty. Po přiřazení hodnoty `20` v tomto příkladu má proměnná novou hodnotu `20`.

```
var age = 8
age = 20
```

Chceme-li mít jistotu, že daná hodnota nebude nikdy změněna, použijeme tzv. konstantu. **Konstanta** je neměnná hodnota v našem programu. Deklarujeme ji klíčovým slovem `let`.

```
let pi = 3.141592
pi = 3   // Chyba, konstantu nelze změnit!
```

2.3.2 Příkazy a výrazy

V programování je důležité rozumět dvěma základním pojmům.

Výraz (expression) je část kódu, která po svém vyhodnocení produkuje nějakou hodnotu. Například samotné číslo `8` je výraz, stejně jako matematický výpočet `5 + 3`.

Příkaz (statement) je kompletní instrukce, která provádí nějakou akci. Jedním z nejběžnějších příkazů je definice, kterou vytváříme novou proměnnou nebo konstantu.

```
var age = 8
```

Jak jsme si již řekli, jazyk *Swift* netrvá na tom, že se za příkazem musí napsat středník. Příkaz lze stejně dobře ukončit koncem řádku.

Je-li pro nás příkaz neakceptovatelně dlouhý a rozepíšeme-li jej na více řádků, překladač to ve většině případů pozná. Nedaří-li se mu to, můžeme si pomoci například tím, že vhodnou část kódu zalomeného na druhý řádek uzavřeme do závorek, což je pro překladač dostatečná nápověda.

Středníky se ve *Swiftu* musí používat pouze v případě, kdy chceme na řádek umístit několik příkazů současně. Pak použijeme středníky jako oddělovače jednotlivých příkazů na řádku.

2.3.3 Komentáře

Přicházíte-li z jazyků *Java*, *C++* nebo *C#*, budete zde jako doma. *Swift* používá pro běžné komentáře naprosto stejnou syntaxi: dvě lomítka (`//`) pro jednořádkový komentář a dvojici `/*` a `*/` pro komentář blokový. Vývojářům přicházejícím z jazyka *Python* tak jen stručně připomínám, že známý křížek (`#`) zde nefunguje. Z hlediska překladače se jakýkoliv komentář chová jednoduše jako bílý znak.

Zásadním a velmi užitečným rozdílem oproti jazykům *Java*, *C++* i *C#* je skutečnost, že ve *Swiftu* lze blokové komentáře bezpečně vnořovat. Potřebujete-li při ladění dočasně zakomentovat rozsáhlý blok kódu, který již uvnitř nějaké blokové komentáře obsahuje, *Swift* si správně pohlídá úrovně zanoření a nevyhodí syntaktickou chybu při nalezení prvního vnitřního ukončení `*/`.

Nicméně při zadání příkazu k zakomentování části kódu se editory *Swiftu* zachovají obdobně jako editory jiných jazyků a zakomentují každý z vybraných řádků řádkovým komentářem, protože je to přehlednější.

Tvorba dokumentace

Pro začleňování dokumentace přímo do zdrojového kódu má *Swift* vyhrazené speciální dokumentační komentáře. Standardním a idiomatickým přístupem je uvození každého řádku třemi lomítky (///), a to i v případě velmi rozsáhlých víceřádkových textů. Syntaxe pomocí /** a */ v jazyce sice rovněž existuje, ale v běžném kódu se používá zcela výjimečně. Komentáře se umísťují vždy bezprostředně nad deklaraci dokumentované entity.

Klíčovou vlastností dokumentačních komentářů ve *Swiftu* je plná podpora formátovací syntaxe Markdown. Při popisování rozhraní tak můžete zcela přirozeně používat tučné písmo, odrážkové seznamy nebo formátované bloky s ukázkovým kódem.

V editoru *Xcode* se v nápovědě *Quick Help* zapsaný Markdown automaticky zobrazuje v čitelné vizuální podobě. Jde také o standardní formát, ze kterého systém *DocC* generuje finální HTML dokumentaci pro celý projekt.

```
1  /// Sečte dvě celá čísla a vrátí výsledek.
2  ///
3  /// Funkce využívá standardní operátor +.
4  /// - Parameters:
5  ///   - a: První sčítanec.
6  ///   - b: Druhý sčítanec.
7  /// - Returns: Celkový součet obou čísel.
8  func add(a: Int, b: Int) -> Int { a + b }
9
10 // MARK: - Pomocné metody
```

Navigace v kódu pomocí značek MARK

Na posledním řádku předchozího výpisu se nachází komentář začínající MARK:. Nejedná se o součást definice jazyka *Swift* ani o instrukci pro překladač, ale o vizuální a navigační pomůcku vyvinutou pro vývojové prostředí *Xcode*.

V horní navigační liště *Xcode* nad editorem kódu se nachází takzvaný *Jump Bar*. Vložíte-li do kódu komentář ve formátu // MARK:, vytvoří se v této liště navigační záložka obsahující text uvedený na zbytku řádku. Kliknutím na ni editor okamžitě odroluje na dané místo v souboru. Záložka má dvě možné podoby:

- // MARK: Nadpis – Do navigační nabídky se přidá pouze položka **Nadpis**.
- // MARK: - Nadpis – Do navigační nabídky se přidá vizuální oddělovač (čára) a těsně pod něj položka Nadpis.

2.4 Datové typy

Každý program pracuje s daty a jazyk *Swift* patří mezi přísně staticky typované jazyky. Znamená to, že typ každé proměnné či konstanty musí být znám již v době překladu, což překladači umožňuje odhalit obrovské množství chyb ještě před samotným spuštěním programu.

V této podkapitole se podíváme na to, co to vlastně datový typ je, ukážeme si, jak si za nás překladač dokáže typy inteligentně domýšlet, a představíme si základní stavební kameny pro práci s čísly a textem.

2.4.1 Trocha teorie

Datový typ objektu definuje tři věci:

- **Množinu hodnot**, kterých může objekt daného typu nabývat (jiných hodnot nabývají čísla a jiných např. stringy).
- **Množinu operací**, které můžeme s objekty daného typu provádět (s čísly můžeme provádět např. aritmetické operace, kdežto stringy můžeme např. skládat nebo převádět na malá či velká písmena).
- **Vnitřní reprezentaci** hodnot daného typu, která současně určuje, kolik paměti je pro objekty daného typu třeba vyhradit (alokovat).

2.4.2 Deklarace typu a typová inference

Při definici proměnné musíme vždy určit typ hodnot, které lze do dané proměnné ukládat. Explicitní určení datového typu můžeme vynechat v případě, kdy v rámci definice přiřazujeme definované proměnné počáteční hodnotu, protože překladač si z této hodnoty dokáže příslušný typ odvodit. V předchozím příkladu jsme definovali:

```
var value = 8
```

Díky automatickému odvození datového typu z přiřazované hodnoty (*type inference*), jsme datový typ nemuseli uvádět.

V uvedeném příkladu překladač zjistí, že hodnota `8` představuje celé číslo, jež ve *Swiftu* představuje datový typ `Int`. Stejný kód jde zapsat i tak, že typ uvedeme explicitně:

```
var value: Int = 8
```

V dalším příkladu definujeme konstantu `pi` s hodnotou rovnou zaokrouhlené hodnotě čísla π. V tomto případě použijeme datový typ `Double` pro tvorbu reálných čísel. K definici konstanty (neměnné hodnoty), použijeme klíčové slovo `let`.

```
let pi: Double = 3.141592
```

Zvláštní pozornost musíme věnovat znakům a textům. Zápis `"A"` do uvozovek překladač automaticky vyhodnotí jako string, tedy jako typ `String`. Chceme-li do proměnné uložit pouze jediný znak reprezentovaný typem `Character`, musíme to výslovně uvést:

```
var name = "Filip"              // Odvodí se jako String
var letter: Character = "A"     // Explicitní vynucení typu Character
```

Vážení čtenáři, právě jste dočetli ukázku z knihy ***Programovací jazyk Swift*.**
Pokud se Vám ukázka líbila, na našem webu si můžete zakoupit celou knihu.