O'REILLY®

Zgodne z iOS 12, Xcode 10 i Swift 4.2

iOS 12
Wprowadzenie do programowania w Swifcie

PODSTAWY SWIFTA, XCODE I COCOA

Helion

Matt Neuburg

Tytuł oryginału: iOS 12 Programming Fundamentals with Swift, Fifth Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-5257-5

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/ios12s
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/ios12s.zip

Printed in Poland.

Spis treści

Część II. Środowisko IDE

Dodatki

ROZDZIAŁ 1.

Architektura Swifta

Na początku dobrze jest ogólnie poznać konstrukcję Swifta i sposób działania utworzonego w tym języku programu na platformie iOS. W tym rozdziale przedstawię ogólną architekturę i naturę języka Swift. Natomiast w kolejnych rozdziałach znacznie dokładniej poznasz ten język.

Zaczynamy

Pełna instrukcja w Swifcie nosi nazwę *polecenia*. Plik tekstowy wraz z kodem Swifta zawiera wiele *wierszy* tekstu. Znaki końca wiersza nie mają znaczenia. Typowy układ programu to jedno polecenie w każdym wierszu.

```
print("Witaj,")
print("świecie!")
```

(Polecenie `print` powoduje natychmiastowe wyświetlenie danych w konsoli Xcode).

Istnieje możliwość umieszczenia w wierszu więcej niż tylko jednego polecenia, choć wtedy należy je rozdzielić średnikiem.

```
print("Witaj,"); print("świecie!")
```

Na końcu polecenia lub ostatniego polecenia w wierszu możesz umieścić średnik, ale tak naprawdę nikt tego nie robi — chyba że z przyzwyczajenia, ponieważ języki C i Objective-C *wymagają* średnika na końcu polecenia.

```
print("Witaj,");
print("świecie!");
```

Pojedyncze polecenie może zostać podzielone na wiele wierszy, aby uniknąć sytuacji, w której długie polecenie spowoduje powstanie długiego wiersza. Jednak podział należy starać się przeprowadzić w sensowny sposób. Na przykład dobre miejsce na umieszczenie znaku nowego wiersza znajduje się po nawiasie otwierającym.

```
print(
    "świecie!")
```

Komentarzem jest wszystko to, co w danym wierszu znajduje się po dwóch ukośnikach (to tzw. komentarz w stylu C++).

```
print("świecie!") // To jest komentarz, który zostanie zignorowany przez Swifta.
```

Istnieje również możliwość umieszczania komentarzy w /*...*/, podobnie jak w języku C. Jednak w przeciwieństwie do C, komentarze w stylu języka C mogą być zagnieżdżane.

W wielu konstrukcjach Swifta nawias klamrowy działa w charakterze ogranicznika.

```
class Dog {
    func bark() {
        print("hau")
    }
}
```

Zgodnie z konwencją przed nawiasem klamrowym i po nim znajduje się znak nowego wiersza, a sam nawias jest wcięty, aby zapewnić większą przejrzystość kodu źródłowego, jak pokazałem we wcześniejszym fragmencie kodu. Wprawdzie Xcode pomaga w stosowaniu tej konwencji, ale tak naprawdę to nie ma żadnego znaczenia dla Swifta. Układ zastosowany w kolejnym przykładowym poleceniu jest prawidłowy i czasami okazuje się znacznie wygodniejszy.

```
class Dog { func bark() { print("hau") }}
```

Swift to język *kompilowany*. Oznacza to, że utworzony kod źródłowy musi być *skompilowany* — przekazany kompilatorowi i zmieniony z tekstu na niskiego poziomu postać zrozumiałą dla komputera — zanim będzie mógł zostać *uruchomiony* i użyty do wykonania pewnego zadania. Kompilator Swifta jest bardzo restrykcyjny, w trakcie tworzenia programu będziesz go wielokrotnie kompilował i uruchamiał tylko po to, aby się przekonać o braku możliwości przeprowadzenia kompilacji ze względu na istnienie *błędu*. Uruchomienie kodu będzie wymagało wcześniejszego usunięcia tego błędu. W niektórych sytuacjach kompilator będzie generował *ostrzeżenie*. W takim przypadku kod może zostać uruchomiony, choć należy poważnie potraktować ostrzeżenie i usunąć wskazywany w nim problem. Ta rygorystyczność kompilatora jest jedną z największych zalet Swifta i oznacza dokładne sprawdzenie poprawności kodu jeszcze przed jego uruchomieniem.

Generowane przez kompilator Swifta komunikaty błędów i ostrzeżeń są różne, od niezwykle pomocnych po myląće. Bardzo często będziesz wiedział o istnieniu *problemu* związanego z wierszem kodu, natomiast kompilator Swifta nie będzie w stanie *dokładnie* wskazać nieprawidłowości ani nawet *miejsca*, na które należy zwrócić uwagę. W takim przypadku najlepiej jest podzielić polecenie na wiele prostszych, co powinno ułatwić znalezienie źródła problemu. Spróbuj pokochać kompilator, mimo że czasami generowane przez niego komunikaty okazują się niewystarczające. Pamiętaj, że kompilator wie więcej od Ciebie, nawet jeśli czasami nie potrafi jasno tego przekazać.

Czy wszystko jest obiektem?

We Swifcie „wszystko jest obiektem”. To stwierdzenie dotyczy wielu nowoczesnych zorientowanych obiektowo języków programowania, ale co tak naprawdę oznacza? Odpowiedź zależy od tego, co rozumiesz przez określenia „obiekt” i „wszystko”.

Zacznę od określenia, że obiekt to ogólnie rzecz biorąc, coś, do czego można przekazać komunikat. Z kolei komunikat to pewne polecenie. Na przykład psu można wydać polecenia: „Daj głos!” i „Siad!”. W przedstawionej analogii te wyrażenia są komunikatami, natomiast pies to obiekt, do którego zostały skierowane te komunikaty.

W Swifcie składnia wysyłania komunikatu opiera się na *kropce*. Najpierw wskazuje się obiekt, następnie umieszcza kropkę, a później dodaje komunikat. (Część komunikatów zawiera również nawias okrągły, ale w tym momencie można to zignorować. Dokładne omówienie składni przekazywania komunikatów znajdziesz w dalszej części książki). Spójrz na przykłady poprawnej składni.

```
fido.bark()
rover.sit()
```

Idea polegająca na tym, że *wszystko* jest obiektem, sugeruje możliwość przekazywania komunikatów nawet do najprostszych elementów. Rozważ np. 1. Wydaje się, że jest to po prostu literał cyfry i nic poza tym. Jeżeli masz jakiekolwiek doświadczenie w pracy z dowolnym językiem programowania, wówczas nie będzie dla Ciebie zaskoczeniem możliwość użycia następującego polecenia:

```
let sum = 1 + 2
```

Natomiast zaskoczeniem *jest* możliwość użycia składni kropki i przekazania komunikatu. Kolejne polecenie jest prawidłowe w Swifcie (w tym momencie nie zastanawiaj się nad jego znaczeniem).

```
let s = 1.description
```

Zanim przejdziesz dalej, warto jeszcze na chwilę powrócić do niewinnego zapisu `1 + 2` we wcześniejszym poleceniu. Okazuje się, że jest to rodzaj lukru składniowego, wygodny sposób na ukrycie i wyrażenie tego, co tak naprawdę się dzieje. Element 1 jest obiektem, a + to komunikat stosujący składnię specjalną (składnię *operatora*). W Swifcie każdy rzeczownik jest obiektem, natomiast każdy czasownik jest komunikatem.

W Swifcie prawdopodobnie najlepszy sposób na ustalenie, czy dany element jest obiektem, to próba jego modyfikacji. Obiekt typu może być w Swifcie *rozszerzony*, co oznacza możliwość zdefiniowania własnych komunikatów tego typu. Na przykład liczbie nie można standardowo przekazać komunikatu `sayHello()`. Jednak można zmodyfikować typ liczby, aby wspomniana możliwość się pojawiła.

```
extension Int {
    func sayWitaj() {
        print("Witaj, to jest liczba \(self).")
    }
}
1.sayWitaj() // Dane wyjściowe: "Witaj, to jest liczba 1.".
```

Dlatego też w Swifcie element 1 jest obiektem. W niektórych językach programowania, np. Objective-C, element 1 zdecydowanie nie jest obiektem, a wbudowanym „prostym”, czyli skalarnym typem danych. Mamy więc wyraźne rozróżnienie między typami obiektu i typami skalaru. Natomiast w Swifcie nie ma skalarów, *wszystkie* typy są obiektami. I na tym polega znaczenie stwierdzenia „wszystko jest obiektem”.

Trzy odmiany obiektu typu

Jeżeli znasz Objective-C lub inny język zorientowany obiektowo, być może będziesz zdziwiony *rodzajem* obiektu, jakim w Swifcie jest element 1. W wielu językach programowania, np. Objective-C, obiekt jest *klasą* lub jej egzemplarzem. (W dalszej części rozdziału dowiesz się, czym jest egzemplarz). Wprawdzie Swift ma klasy, ale element 1 nie jest ani klasą, ani jej egzemplarzem.

Typem elementu `1` jest `Int`, a dokładnie *struktura*. Dlatego też element `1` to egzemplarz struktury. W Swifcie wiadomości można przekazywać do jeszcze jednego rodzaju obiektu: *wyliczenia*.

Swift ma trzy rodzaje obiektu typu: klasę, strukturę i wyliczenie. Ja posługuję się określeniem *odmiany* obiektu typu. Dokładne różnice między nimi staną się wyraźnie widoczne podczas lektury książki. Wszystkie trzy odmiany to zdecydowanie obiekty typu, a podobieństwa między nimi są znacznie większe niż dzielące je różnice. W tym momencie wystarczy wiedzieć o istnieniu trzech odmian obiektu typu.

(Uznanie struktury i wyliczenia za obiekty typu w Swifcie może być zaskoczeniem dla kogoś, kto ma doświadczenie w programowaniu za pomocą Objective-C. Język Objective-C zawiera struktury i wyliczenia, ale nie są one obiektami. Szczególnie struktury w Swifcie mają znacznie większe znaczenie niż struktury w Objective-C. Różnica między traktowaniem struktur i wyliczeń przez Swifta i Objective-C ma znaczenie podczas pracy z frameworkiem Cocoa).

Zmienna

Zmienna to *nazwa* dla obiektu. Pod względem technicznym *odwołuje się* do obiektu. Można więc ją określić mianem *odwołania* do obiektu. W sposób nietechniczny zmienną można potraktować jak pudełko, w którym został umieszczony obiekt. Obiekt w pudełku może być wielokrotnie modyfikowany lub nawet zastąpiony innym, ale nazwa pozostaje bez zmian. Obiekt, do którego odwołuje się zmienna, to *wartość* zmiennej.

W Swifcie żadna zmienna nie pojawia się samoistnie, wszystkie muszą być wyraźnie *zadeklarowane*. Jeżeli potrzebna jest nazwa dla czegokolwiek, wówczas trzeba jawnie ją utworzyć. Do tego celu służą dwa słowa kluczowe: `let` i `var`. Deklaracji w Swifcie zwykle towarzyszy *inicjalizacja* — za pomocą znaku równości zmiennej można natychmiast przypisać wartość. Oba kolejne polecenia przedstawiają deklarację (i inicjalizację) zmiennych.

```
let one = 1
var two = 2
```

Gdy pewna nazwa istnieje, możesz z niej korzystać. Spójrz na przykład pokazujący, jak zmiennej `two` przypisać wartość zmiennej `one`.

```
let one = 1
var two = 2
two = one
```

W ostatnim wierszu tego kodu zostały użyte nazwy `one` i `two` zadeklarowane w dwóch wcześniejszych wierszach. Nazwa `one` znajdująca się po prawej stronie znaku równości jest użyta, aby odwołać się do wartości w pudełku `one` (tutaj jest to `1`). Z kolei nazwa `two` po lewej stronie znaku równości jest użyta do *zastąpienia* wartości w pudełku `two`. Takie polecenie, zawierające nazwę zmiennej po lewej stronie znaku równości, nosi nazwę *przypisania*. Z kolei znak równości jest określany mianem *operatora przypisania*. Znak równości nie oznacza równości, jak np. we wzorze algebraicznym. Jest to polecenie, które można przedstawić następująco: „Weź wartość po prawej stronie znaku równości i użyj jej do zastąpienia wartości znajdującej się po lewej stronie znaku równości”.

Te dwa rodzaje deklaracji zmiennej różnią się: nazwa zadeklarowana za pomocą słowa kluczowego `let` *nie pozwala na zmianę jej wartości*. Dlatego też zmienna zadeklarowana z użyciem `let` jest *stałą*, tzn. przypisana wartość pozostaje niezmienna. Przedstawiony tutaj fragment kodu nawet się nie skompiluje.

```
let one = 1
var two = 2
one = two // Błąd w trakcie kompilacji.
```

Nazwę zawsze można zadeklarować za pomocą słowa kluczowego `var`, aby zachować największą elastyczność. Jeżeli jednak wiesz, że wartość początkowa zmiennej nigdy się nie zmieni, wtedy lepiej użyć słowa kluczowego `let`, ponieważ pozwala ono Swiftowi na znacznie efektywniejsze działanie. Różnica w efektywności jest na tyle duża, że kompilator Swifta zwraca Twoją uwagę na każde użycie polecenia `var`, gdy zamiast niego można zastosować `let`, i oferuje możliwość przeprowadzenia takiej zmiany.

Zmienna ma również *typ*. Konkretny typ jest ustalany podczas deklarowania zmiennej i *nigdy się nie zmienia*. Dlatego też przedstawiony tutaj fragment kodu nawet się nie skompiluje.

```
var two = 2
two = "witaj" // Błąd w trakcie kompilacji.
```

Po zadeklarowaniu zmiennej `two` i zainicjowaniu jej z wartością `2` przedstawia ona liczbę całkowitą i zawsze musi przechowywać wartość takiego typu. Wartość zmiennej `two` można zastąpić przez `1`, ponieważ to również jest liczba całkowita. Jednak wartością zmiennej `two` nie może być `"witaj"` — jest to ciąg tekstowy, a nie liczba całkowita.

Można stwierdzić, że zmienna żyje własnym życiem — a dokładnie ma swój *cykl życiowy*. Dopóki zmienna istnieje, dopóty zachowuje przypisaną jej wartość. Tak więc zmienna to nie tylko wygodny sposób na *nadanie nazwy* czemuś, ale również na *zachowanie* tego czegoś. Więcej informacji na ten temat przedstawię w dalszej części rozdziału.

Zgodnie z konwencją nazwy typów, np. `Int` i `String` (bądź `Dog` lub `Cat`), zaczynają się od dużej litery. Z kolei nazwa zmiennej rozpoczyna się od małej litery. *Nie łam tej konwencji.* Wprawdzie jeżeli tak zrobisz, kod źródłowy wciąż może się skompilować i uruchomić, ale ostrzegam: wyślę do Twojego domu agentów, którzy w nocy przestrzelą Ci kolana.

Funkcja

Możliwy do wykonania kod, np. `fido.bark()` lub `one = two`, nie może zostać umieszczony gdziekolwiek w programie. (Nieumiejętność zaakceptowania tego to często popełniany przez początkujących błąd, który może prowadzić do wygenerowania tajemniczych komunikatów błędów podczas kompilacji, takich jak „`Expected declaration.`"). Ogólnie rzecz biorąc, taki kod powinien znajdować się w *funkcji*. Funkcja to zbiór poleceń, które można uruchomić. Funkcja zwykle ma nazwę otrzymywaną podczas jej deklarowania. Składnia deklaracji funkcji to jeden ze szczegółów, którymi zajmę się w dalszej części książki. Spójrz na poniższy przykład funkcji.

```
func go() {
    let one = 1
    var two = 2
    two = one
}
```

Mamy tutaj przedstawioną sekwencję zadań do wykonania — zadeklarowanie elementów `one` i `two`, zmiana wartości elementu `two` na wartość elementu `one` — i nadanie jej *nazwy* `go`. To jednak nie wystarczy do *wykonania* tej sekwencji. Wykonanie sekwencji następuje dopiero po wywołaniu *funkcji*. Dlatego też w innym miejscu programu może się znaleźć następujące polecenie:

```
go()
```

To polecenie powoduje wywołanie funkcji `go()` i jej faktyczne uruchomienie. To również jest przykład kodu przeznaczonego do wykonania, który jednak nie może wisieć w próżni. Przedstawione polecenie można więc umieścić w innej funkcji.

```
func doGo() {
    go()
}
```

Zaczekaj, to staje się nieco szalone. Mamy deklarację kolejnej funkcji przeznaczonej do wykonania wcześniej zdefiniowanej `go()`. Nowa funkcja również jest kodem możliwym do wykonania. To rodzaj nigdy niekończącej się regresji — czy utworzony kod będzie można *kiedykolwiek* uruchomić? Jeżeli każdy kod wykonywalny musi być zdefiniowany w funkcji, to co może spowodować uruchomienie *jakiejkolwiek* funkcji? Przecież początkowy impuls musi skądś pochodzić.

Na szczęście problem regresji nie istnieje. Czy pamiętasz, jak wspomniałem, że ostatecznym celem programisty jest utworzenie aplikacji dla systemu iOS? Taka aplikacja zostanie uruchomiona w urządzeniu iOS (lub symulatorze) przez środowisko uruchomieniowe, które chce wywołania pewnych funkcji. Pracę zaczynasz więc od zdefiniowania funkcji specjalnych, o których wiadomo, że będą wywołane przez środowisko uruchomieniowe. W ten sposób masz możliwość uruchomienia aplikacji i otrzymujesz miejsce do definiowania własnych funkcji wywoływanych przez środowisko uruchomieniowe w kluczowych momentach, np. podczas uruchamiania aplikacji lub po naciśnięciu przez użytkownika przycisku w interfejsie użytkownika aplikacji.

Swift ma również regułę specjalną, zgodnie z którą plik o nazwie *swift.main* wyjątkowo może mieć kod zdefiniowany na najwyższym poziomie struktury pliku, poza jakąkolwiek funkcją. Ten kod będzie wykonany po uruchomieniu programu. Możesz opracować aplikację, wykorzystując plik *main.swift*, choć tak naprawdę to nie jest konieczne.

Struktura pliku Swifta

Program utworzony w języku Swift może się składać z jednego lub więcej plików. W Swifcie plik jest uznawany za pewną znaczącą jednostkę i istnieją reguły dotyczące struktury kodu Swifta, który może się w nim znajdować. (Przyjmuję założenie, że *nie umieszczasz* kodu w pliku *main.swift*). Tylko wybrane elementy mogą się znajdować na najwyższym poziomie struktury pliku Swifta.

Polecenia importowania modułów
: Moduł to jednostka znajdująca się na wyższym poziomie niż plik. Moduł może zawierać wiele plików, które automatycznie mogą mieć do siebie dostęp. Jednak moduł nie zobaczy innego modułu, jeżeli nie zostanie użyte polecenie `import`. Dlatego też jeśli w programie iOS chcesz się odwołać do frameworka Cocoa, pierwszy wiersz pliku musi zawierać polecenie `import UIKit`.

Deklaracje zmiennych
: Zmienna zadeklarowana na początku pliku jest zmienną *globalną*. Cały kod widzi tę zmienną i może uzyskać do niej dostęp bez konieczności jawnego wysyłania komunikatu do jakiegokolwiek obiektu. Taka zmienna pozostaje dostępna przez cały czas działania programu.

Deklaracje funkcji
: Funkcja zadeklarowana na najwyższym poziomie pliku staje się funkcją *globalną*. Cały kod widzi tę funkcję i może ją wywoływać bez konieczności jawnego wysyłania komunikatu do jakiegokolwiek obiektu.

Deklaracje typu obiektu
: Deklaracje klasy, struktury lub wyliczenia.

W kolejnym fragmencie kodu przedstawiłem (w celach demonstracyjnych) prawidłowy plik Swifta z poleceniem `import` oraz deklaracjami zmiennej globalnej, funkcji, klasy, struktury i wyliczenia.

```
import UIKit
var one = 1
func changeOne() {
}
class Manny {
}
struct Moe {
}
enum Jack {
}
```

To niezwykle prosty i pusty przykład. Moim celem jest tutaj pokazanie elementów języka i struktury pliku Swifta.

Wewnątrz nawiasów klamrowych dla poszczególnych elementów w tym przykładzie mogą się znajdować kolejne deklaracje zmiennych, funkcji i typów obiektu. W rzeczywistości *każdy* strukturalny nawias klamrowy może zawierać takie deklaracje.

Prawdopodobnie zauważyłeś, że *nie wspomniałem* o możliwości umieszczenia kodu wykonywalnego na najwyższym poziomie struktury pliku. Powód jest prosty: nie można tego zrobić. *Tylko funkcja może zawierać kod wykonywalny*. Polecenia typu `one = two` i `print(name)` są przykładami kodu wykonywalnego i nie mogą się znajdować na najwyższym poziomie struktury pliku Swifta. W przedstawionym wcześniej przykładzie mamy deklarację funkcji, `func changeOne()`, i dlatego można w jej nawiasie klamrowym umieścić kod wykonywalny, jak pokazałem w kolejnym fragmencie kodu.

```
var one = 1
// W tym miejscu nie można umieścić kodu wykonywalnego.
func changeOne() {
    let two = 2 // Kod wykonywalny.
    one = two   // Kod wykonywalny.
}
```

Kod wykonywalny nie może zostać również umieszczony bezpośrednio w nawiasie klamrowym deklaracji klasy `Manny` w omawianym przykładzie. Oznacza to brak możliwości umieszczenia go na najwyższym poziomie klasy. Natomiast deklaracja klasy *może* zawierać deklarację funkcji, która z kolei *może* zawierać kod wykonywalny.

```
class Manny {
    let name = "Mariusz"
    // W tym miejscu nie można umieścić kodu wykonywalnego.
    func sayName() {
        print(name) // Kod wykonywalny.
    }
}
```

Podsumowując: na listingu 1.1 przedstawiłem prawidłowy plik Swifta i w sposób schematyczny zilustrowałem dostępne możliwości strukturalne. (Zignoruj znajdującą się tutaj deklarację zmiennej `name` w deklaracji wyliczenia `Jack`. Zmienne wyliczenia najwyższego poziomu stosują się do pewnych reguł specjalnych, które omówię w dalszej części książki).

Listing 1.1. Schematyczna struktura prawidłowego pliku Swifta

```
import UIKit
var one = 1
func changeOne() {
    let two = 2
    func sayTwo() {
        print(two)
    }
    class Klass {}
    struct Struct {}
    enum Enum {}
    one = two
}
class Manny {
    let name = "Mariusz"
    func sayName() {
        print(name)
    }
    class Klass {}
    struct Struct {}
    enum Enum {}
}
struct Moe {
    let name = "Mieczysław"
    func sayName() {
        print(name)
    }
    class Klass {}
    struct Struct {}
    enum Enum {}
}
enum Jack {
    var name : String {
        return "Jacek"
    }
    func sayName() {
```

```
            print(name)
        }
        class Klass {}
        struct Struct {}
        enum Enum {}
    }
```

Oczywiście można zastosować rekurencję w dół do dowolnego poziomu: deklaracja klasy może zawierać kolejną deklarację klasy, która z kolei będzie miała następną deklarację klasy itd. Uważam, że nie ma potrzeby ilustrowania tego.

Zasięg i cykl życiowy

W programie Swifta tworzące go elementy mają *zasięg*. Odwołuje się on do możliwości uzyskania dostępu do danego elementu z poziomu innych elementów. Elementy są zagnieżdżane w innych i tym samym tworzą hierarchię zagnieżdżoną. Zgodnie z regułą elementy mają dostęp do innych elementów *znajdujących się na tym samym lub wyższym poziomie*. Do dyspozycji mamy następujące poziomy zasięgu:

- moduł,
- plik,
- nawias klamrowy.

Gdy coś zostało zadeklarowane, znajduje się na pewnym poziomie tej hierarchii. Miejsce w hierarchii, czyli zasięg, określa dostępność danego elementu w innych elementach.

Spójrz raz jeszcze na przykład przedstawiony na listingu 1.1. W deklaracji klasy `Manny` znajdują się deklaracje zmiennej `name` i funkcji `sayName()`. Kod znajdujący się *wewnątrz* nawiasu klamrowego funkcji `sayName()` ma dostęp do tych elementów *spoza* nawiasu klamrowego, które *znajdują się na wyższym poziomie*. Dlatego też ten kod ma dostęp do np. zmiennej `name`. Podobnie kod zdefiniowany w funkcji `changeOne()` ma dostęp do zmiennej `one` zadeklarowanej na najwyższym poziomie pliku. Tak naprawdę *każdy* komponent w pliku ma dostęp do zmiennej `one` zadeklarowanej na najwyższym poziomie pliku, czyli jako zmiennej globalnej.

Zasięg ma bardzo ważne znaczenie podczas *współdzielenia informacji*. Dwie różne funkcje zadeklarowane w klasie `Manny` będą miały dostęp do elementu `name` zadeklarowanego na najwyższym poziomie tej klasy. Kod znajdujący się w strukturze `Moe` i wyliczeniu `Jack` również ma dostęp do elementu `name`.

Elementy mają także *cykl życiowy*, który w praktyce odpowiada ich zasięgowi. Element istnieje, dopóki jest dostępny obejmujący go zasięg. W przypadku kodu przedstawionego na listingu 1.1 zmienna `one` istnieje tak długo jak plik, czyli w trakcie całego działania programu. Jest to zmienna globalna *i trwała*. Natomiast zmienna `name` zadeklarowana na najwyższym poziomie klasy `Manny` istnieje, dopóki jest dostępny egzemplarz klasy `Manny`. (Do tego tematu wkrótce powrócę).

Elementy deklarowane na bardziej zagnieżdżonych poziomach mają jeszcze krótsze cykle życiowe. Spójrz na przedstawiony tutaj fragment kodu.

```
func silly() {
    if true {
        class Cat {}
        var one = 1
        one = one + 1
    }
}
```

To tylko prosty, choć w pełni prawidłowy przykład. Wcześniej wspomniałem, że deklaracje zmiennych, funkcji i typów obiektu mogą się znajdować w *dowolnym* strukturalnym nawiasie klamrowym. W tym przykładzie klasa `Cat` i zmienna `one` będą istniały po wywołaniu funkcji `silly()`, ale tylko wtedy, gdy wynikiem wykonania konstrukcji `if` będzie `true`. Przyjmijmy założenie o wywołaniu funkcji `silly()` i przejściu do konstrukcji `if`. Najpierw następuje utworzenie egzemplarza klasy `Cat`, następnie zadeklarowanie zmiennej `one` i wreszcie wykonanie polecenia `one = one + 1`. Później zasięg zostaje usunięty, co oznacza pozbycie się również egzemplarza klasy `Cat` i zmiennej `one`. W trakcie ich krótkiego cyklu życiowego egzemplarz klasy `Cat` i zmienna `one` pozostały całkowicie niewidoczne dla pozostałej części programu. (Czy potrafisz powiedzieć, dlaczego tak się stało?).

Elementy składowe obiektu

Wewnątrz trzech typów obiektu (klasa, struktura i wyliczenie) elementy zadeklarowane na najwyższym poziomie mają nazwy specjalne, przede wszystkim z powodów historycznych. Zademonstruję to na przykładzie klasy `Manny`.

```
class Manny {
    let name = "Mariusz"
    func sayName() {
        print(name)
    }
}
```

W tym fragmencie kodu:

- `name` to zmienna zadeklarowana na najwyższym poziomie deklaracji obiektu i dlatego nazywana *właściwością* tego obiektu;
- `sayName()` to funkcja zadeklarowana na najwyższym poziomie deklaracji obiektu i dlatego nazywana *metodą* tego obiektu.

Elementy deklarowane na najwyższym poziomie obiektu — czyli właściwości, metody i wszelkie inne obiekty na tym poziomie — są razem określane mianem *elementów składowych* danego obiektu. Te elementy składowe mają znaczenie specjalne, ponieważ definiują *komunikaty*, które mogą być przekazywane temu obiektowi.

Przestrzeń nazw

Przestrzeń nazw to nazwany obszar programu. Elementy znajdujące się w przestrzeni nazw są niedostępne dla elementów zewnętrznych, jeżeli nie zostanie najpierw *podana* nazwa obszaru. To dobre rozwiązanie, ponieważ pozwala na wielokrotne użycie tej samej nazwy w różnych miejscach bez powodowania konfliktu. Nie ulega wątpliwości, że przestrzeń nazw i zasięg są ze sobą ściśle powiązane.

Przestrzeń nazw pomaga w wyjaśnieniu znaczenia deklaracji obiektu na najwyższym poziomie innego obiektu, np.:

```
class Manny {
    class Klass {}
}
```

Sposób zadeklarowania klasy `Klass` powoduje, że to jest *typ zagnieżdżony*. Obiekt `Klass` jest tak naprawdę ukryty w obiekcie `Manny`, który w takim przypadku jest przestrzenią nazw. Kod zdefiniowany *wewnątrz* klasy `Manny` ma bezpośredni dostęp do obiektu `Klass`. Natomiast kod spoza klasy `Manny` nie ma takiej możliwości, zatem konieczne jest *wyraźne* podanie przestrzeni nazw, aby można było pokonać barierę nałożoną przez tę przestrzeń nazw. Dlatego też najpierw trzeba podać nazwę egzemplarza klasy `Manny`, następnie kropkę i dalej element egzemplarza klasy `Klass`. Innymi słowy: konieczne jest wywołanie w postaci `Manny.Klass`.

Przestrzeń nazw sama w sobie nie zapewnia prywatności, to tylko udogodnienie dla programisty. W przykładzie przedstawionym na listingu 1.1 klasy `Klass` zostały zdefiniowane w klasie `Manny` i strukturze `Moe`. Nie powoduje to żadnego konfliktu, ponieważ te klasy `Klass` znajdują się w różnych przestrzeniach nazw, co pozwala na ich rozróżnianie, gdy zachodzi potrzeba, np. `Manny.Klass` i `Moe.Klass`.

Prawdopodobnie nie umknęło Twojej uwadze to, że składnia używania przestrzeni nazw odpowiada składni kropki przeznaczonej do przesyłania komunikatów. Tak naprawdę obie te składnie to jedno i to samo.

W efekcie podczas przekazywania komunikatu można uzyskać dostęp do zasięgu niewidocznego w innych okolicznościach. Kod struktury `Moe` *automatycznie* ma dostęp do klasy `Klass` zadeklarowanej w `Manny`. Ten sam dostęp *można* sobie zapewnić za pomocą dodatkowego kroku w postaci jawnego podania przestrzeni nazw `Manny.Klass`. To *możliwe*, ponieważ kod ma dostęp do egzemplarza `Manny` (klasa `Manny` została zadeklarowana na poziomie dostępnym dla kodu struktury `Moe`).

Moduł

Przestrzeń nazw najwyższego poziomu to *moduł*. Domyślnie aplikacja jest modułem, dlatego istnieje w niej przestrzeń nazw. Jej nazwa odpowiada nazwie aplikacji. Na przykład jeśli nazwa aplikacji to `MyApp`, wówczas po zadeklarowaniu klasy `Manny` na najwyższym poziomie pliku *rzeczywistą* nazwą tej klasy będzie `MyApp.Manny`. Zwykle nie używa się tej nazwy rzeczywistej, ponieważ kod już znajduje się wewnątrz tej samej przestrzeni nazw i bez żadnych problemów ma dostęp do klasy `Manny`.

Framework również jest modułem, dlatego istnieje w nim przestrzeń nazw. Gdy importujesz moduł, wszystkie jego deklaracje znajdujące się na najwyższym poziomie są dostępne dla kodu. Aby się do nich odwoływać, nie ma konieczności wyraźnego podawania przestrzeni nazw modułu.

Na przykład jeden z frameworków Cocoa, Foundation, zawiera moduł `NSString`. Podczas tworzenia programu dla systemu iOS używasz polecenia `import Foundation` (lub częściej `import UIKit`, które wykonuje polecenie `import Foundation`) i dlatego możesz używać egzemplarza `NSString` bez konieczności podawania `Foundation.NSString`. Oczywiście *można* stosować wywołanie `Foundation.NSString`

i jeśli we własnym module zdecydujesz się na zadeklarowanie innej klasy `NSString`, będziesz musiał używać `Foundation.NSString`, aby rozróżniać te klasy. Istnieje możliwość tworzenia własnych frameworków, które także będą modułami.

Sam Swift również został zdefiniowany w module o nazwie `Swift`. Kod *zawsze niejawnie importuje moduł Swifta*. Możesz to zrobić jawnie przez umieszczenie na początku pliku wiersza `import Swift`. Nie ma takiej konieczności, choć dodanie tego polecenia również nie zaszkodzi.

Ten fakt jest ważny, ponieważ dostarcza rozwiązanie pewnej tajemnicy: skąd biorą się polecenia takie jak `print()` i dlaczego można ich używać na zewnątrz dowolnego komunikatu każdego obiektu? Tak naprawdę `print()` to funkcja zadeklarowana na najwyższym poziomie modułu Swifta. Twój kod ma dostęp do deklaracji najwyższego poziomu w module Swifta, ponieważ kod niejawnie importuje ten moduł. Dlatego też funkcja `print()` jest z perspektywy utworzonego przez Ciebie kodu zwykłą funkcją najwyższego poziomu, jak wiele innych. Ta funkcja jest globalna i do działania nie wymaga podania przestrzeni nazw. Wprawdzie *można* podać przestrzeń nazw — dozwolone jest stosowanie wywołania w stylu `Swift.print("Witaj")` — ale prawdopodobnie nigdy tak nie będziesz robił z powodu braku konfliktów do rozwiązania.

Istnieje możliwość *zobaczenia* najwyższego poziomu deklaracji Swifta, ich odczytywanie i analizowanie — to będzie pouczające zadanie. Na przykład jeśli chcesz sprawdzić deklarację wywołania `print()`, naciśnij klawisz *Cmd* i kliknij wywołanie prawym przyciskiem myszy. Ewentualnie umieść w kodzie polecenie `import Swift`, a następnie naciśnij klawisz *Cmd* i kliknij słowo *Swift* prawym przyciskiem myszy. Na ekranie powinieneś zobaczyć deklaracje najwyższego poziomu w Swifcie. To nie będzie żaden *kod* wykonywalny Swifta, ale deklaracje dla wszystkich dostępnych pojęć Swifta, w tym także dla funkcji najwyższego poziomu, tj. `print()`, operatorów, tj. `+`, typów wbudowanych, tj. `Int` i `String` (poszukaj `struct Int` i `struct String`), itd.

Egzemplarz

Obiekty typu — klasa, struktura i wyliczenie — mają jedną ważną wspólną cechę: możliwość *utworzenia egzemplarza* na ich podstawie. Dlatego też podczas deklarowania typu obiektu tak naprawdę jedynie definiujesz *typ*. Faktyczne powstanie typu wymaga utworzenia jego *egzemplarza*.

Spójrz na przedstawiony tutaj przykład deklaracji klasy o nazwie `Dog` wraz z metodą klasy.

```
class Dog {
    func bark() {
        print("hau")
    }
}
```

To jednak nie powoduje utworzenia w programie żadnych obiektów typu `Dog`. Ten kod jedynie opisuje *typ* czegoś, co może powstać po utworzeniu *egzemplarza*. Aby otrzymać rzeczywisty obiekt, trzeba go *utworzyć*. Proces tworzenia rzeczywistego obiektu będącego typem klasy `Dog` to właśnie utworzenie jego egzemplarza. W wyniku tej operacji powstaje nowy obiekt — *egzemplarz* typu `Dog`.

Utworzenie egzemplarza w Swifcie odbywa się przez użycie nazwy tego typu jako nazwy funkcji i jej wywołanie. Oznacza to konieczność zastosowania nawiasu okrągłego. Umieszczenie nawiasu okrągłego po nazwie typu obiektu powoduje wysłanie do tego typu obiektu komunikatu specjalnego: utwórz egzemplarz.

Oto polecenie, które powoduje utworzenie egzemplarza typu `Dog`:

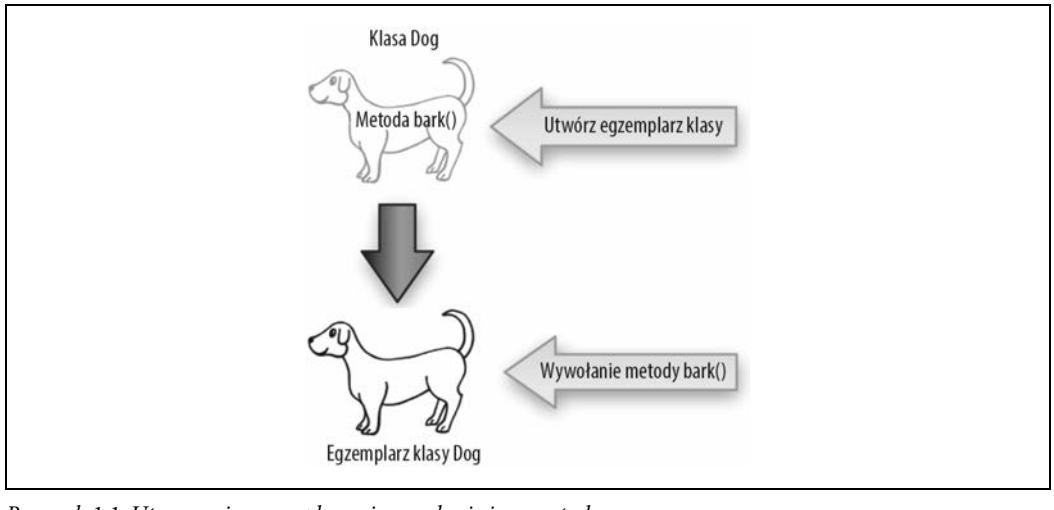
```
let fido = Dog()
```

Całkiem sporo się w tym kodzie dzieje. Wykonane zostały dwa zadania. Przede wszystkim nastąpiło utworzenie egzemplarza typu `Dog`. Ponadto ten egzemplarz został umieszczony w pudełku o nazwie `fido` — zadeklarowałem i zainicjalizowałem zmienną przez przypisanie jej nowo utworzonego egzemplarza typu `Dog`. W tym momencie `fido` to *egzemplarz typu Dog*. (Ponieważ użyłem słowa kluczowego `let`, `fido` zawsze będzie tym samym egzemplarzem typu `Dog`. Wprawdzie można było użyć słowa kluczowego `var`, ale wówczas inicjalizacja `fido` jako egzemplarza typu `Dog` oznaczałaby, że później `fido` mógłby być dowolnym egzemplarzem typu `Dog`).

W tym momencie masz egzemplarz typu `Dog`, do którego można *wysyłać komunikaty*. Jak sądzisz, czym są te komunikaty? To właściwości i metody klasy `Dog`, np.:

```
let fido = Dog()
fido.bark()
```

Przedstawiony tutaj kod jest prawidłowy i powoduje wyświetlenie w konsoli ciągu tekstowego `hau`. Utworzyłeś egzemplarz klasy `Dog` przedstawiający psa, który zaszczekał (rysunek 1.1)



Rysunek 1.1. Utworzenie egzemplarza i wywołanie jego metody

Mamy tutaj ważną kwestię, nad którą warto się przez chwilę zastanowić. Domyślnie właściwości i metody są właściwościami i metodami *egzemplarza*. Dlatego nie można ich używać jako komunikatów przekazywanych obiektowi samego typu, lecz wcześniej trzeba utworzyć *egzemplarz* danego typu, do którego będzie można wysyłać komunikaty. Oznacza to, że kolejne polecenie jest nieprawidłowe i spowoduje wygenerowanie błędu podczas kompilacji.

```
Dog.bark() // Błąd w trakcie kompilacji.
```

Istnieje możliwość zadeklarowania metody `bark()` w sposób, dzięki któremu będzie *można* użyć wywołania `Dog.bark()`. Jednak wówczas otrzymasz zupełnie inny rodzaj metody — metodę *klasy*, czyli *statyczną*. Rodzaj takiej metody trzeba podać w trakcie jej deklarowania.

To samo dotyczy również właściwości. Aby to zilustrować, w klasie `Dog` zdefiniuję właściwość o nazwie `name`.

```
class Dog {
    var name = ""
}
```

W ten sposób będzie można przypisać wartość właściwości `name`, choć musi to być właściwość *egzemplarza* typu `Dog`.

```
let fido = Dog()
fido.name = "Fido"
```

Istnieje możliwość zadeklarowania właściwości `name` w sposób, dzięki któremu będzie *można* użyć wywołania `Dog.name`. Jednak wówczas otrzymasz zupełnie inny rodzaj właściwości — właściwość *klasy*, czyli *statyczną*. Rodzaj takiej właściwości trzeba podać w trakcie jej deklarowania.

Dlaczego egzemplarz?

Nawet jeśli nie ma czegoś takiego jak egzemplarz, typ obiektu jest sam w sobie obiektem. Wiesz o tym, ponieważ masz możliwość wysyłania komunikatu do typu obiektu (przykładem może być przedstawione wcześniej polecenie `Manny.Klass`). Mógłbyś więc zapytać, jaki jest powód istnienia egzemplarzy.

Odpowiedź wiąże się z naturą właściwości egzemplarza. Wartość właściwości egzemplarza jest definiowana w związku z *określonym egzemplarzem*. W tym miejscu egzemplarz pokazuje swoją rzeczywistą użyteczność i potęgę.

Powróć na chwilę do klasy `Dog`. Masz w niej właściwość `name` i metodę `bark()`. Pamiętaj, że jest to odpowiednio właściwość egzemplarza i metoda egzemplarza.

```
class Dog {
    var name = ""
    func bark() {
        print("hau")
    }
}
```

Egzemplarz `Dog` powstaje wraz z pustym ciągiem tekstowym jako wartością właściwości `name`. Właściwość `name` została zdefiniowana za pomocą słowa kluczowego `var`, więc mając egzemplarz klasy `Dog`, można tej właściwości przypisać nową wartość w postaci ciągu tekstowego.

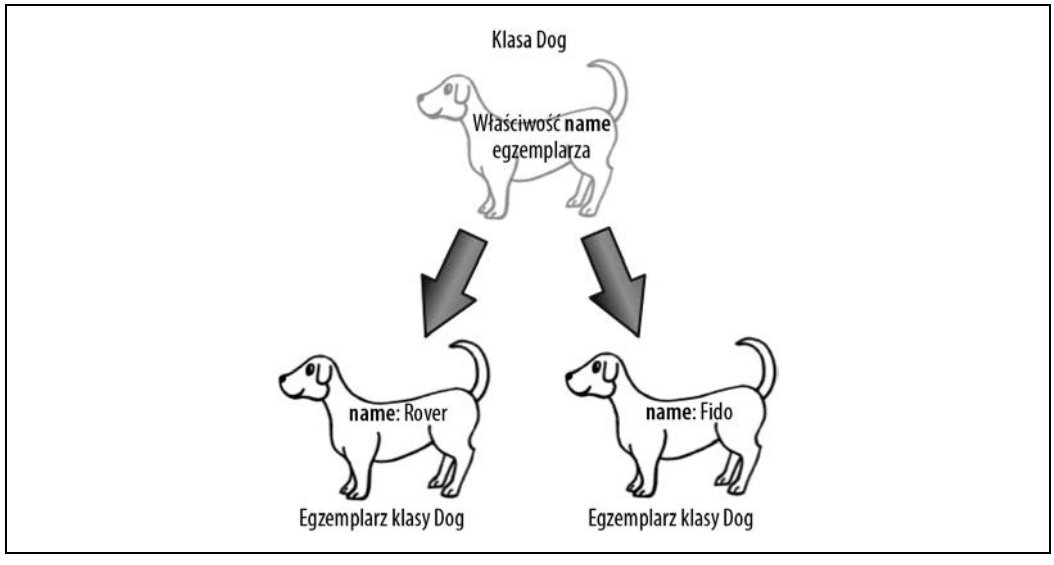
```
let dog1 = Dog()
dog1.name = "Fido"
```

W każdej chwili można sprawdzić wartość właściwości `name` egzemplarza klasy `Dog`.

```
let dog1 = Dog()
dog1.name = "Fido"
print(dog1.name) // Dane wyjściowe: "Fido".
```

Największą zaletą jest możliwość utworzenia więcej niż tylko jednego egzemplarza klasy `Dog`. Na przykład dwa różne egzemplarze klasy `Dog` mogą mieć zupełnie odmienne wartości właściwości `name`, jak pokazałem na rysunku 1.2.

```
let dog1 = Dog()
dog1.name = "Fido"
let dog2 = Dog()
dog2.name = "Rover"
print(dog1.name) // Dane wyjściowe: "Fido".
print(dog2.name) // Dane wyjściowe: "Rover".
```



Rysunek 1.2. Dwa egzemplarze klasy Dog z różnymi wartościami właściwości name

Zwróć uwagę na to, że właściwość `name` egzemplarza klasy `Dog` nie ma nic wspólnego z nazwą zmiennej, której został przypisany dany egzemplarz. Ta zmienna to jedynie pudełko. Wprawdzie egzemplarz można przekazywać z pudełka do pudełka, ale mimo to zachowuje on swoją wewnętrzną spójność.

```
let dog1 = Dog()
dog1.name = "Fido"
var dog2 = Dog()
dog2.name = "Rover"
print(dog1.name) // Dane wyjściowe: "Fido".
print(dog2.name) // Dane wyjściowe: "Rover".
dog2 = dog1
print(dog2.name) // Dane wyjściowe: "Fido".
```

Ten fragment kodu nie powoduje zmiany właściwości `name` egzemplarza `dog2`. Zamiast tego zawartość pudełka `dog2` zastępuje zawartością pudełka `dog1`.

W ten sposób możesz dostrzec potężne możliwości programowania zorientowanego obiektowo. Mamy obiekt typu `Dog` definiujący to, *czym jest egzemplarz klasy* `Dog`. Zgodnie z przedstawioną tutaj deklaracją klasy `Dog`, jej egzemplarz — *dowolny* egzemplarz typu `Dog`, *każdy* egzemplarz typu

`Dog` — ma właściwość `name` i metodę `bark()`. Jednak *każdy* egzemplarz klasy `Dog` może mieć własną *wartość* właściwości `name`. Są to *odmienne* egzemplarze i zachowują swój wewnętrzny *stan*. Dlatego też wiele egzemplarzy tego samego obiektu może *zachowywać się* w podobny sposób — w omawianym przykładzie `Fido` i `Rover` mogą szczekać po wysłaniu im komunikatu `bark()` — ale to zupełnie różne egzemplarze, które mogą mieć odmienne wartości przypisane właściwościom. Wartość właściwości `name` egzemplarza `dog1` to `Fido`, natomiast egzemplarza `dog2` to `Rover`.

Egzemplarz można więc uznać za odzwierciedlenie metod egzemplarza danego typu, ale *nie tylko* — to również kolekcja właściwości egzemplarza. Obiekt typu jest odpowiedzialny za zdefiniowanie tego, *co* te właściwości egzemplarza będą przechowywały, ale niekoniecznie za określenie *wartości* dla tych właściwości. Konkretne wartości mogą się zmieniać w trakcie działania programu i mają zastosowanie jedynie dla danego egzemplarza. Innymi słowy: egzemplarz to pewien klaster przechowujący wartości określonych właściwości.

Egzemplarz jest odpowiedzialny nie tylko za wartości, ale również za ich *cykl życiowy*. Przyjmuję założenie o utworzeniu egzemplarza klasy `Dog` i przypisaniu jego właściwości `name` wartości `Fido`. W takim przypadku egzemplarz zachowa tę wartość, dopóki nie zostanie zastąpiona inną wartością, i przez cały czas istnienia danego egzemplarza.

Ujmując rzecz krótko: egzemplarz zawiera kod i dane. Kod pochodzi z typu i pod tym względem jest współdzielony przez wszystkie egzemplarze danego typu. Z kolei dane należą wyłącznie do egzemplarza. Dane nie mogą istnieć dłużej niż egzemplarz. W każdej chwili egzemplarz ma informacje o stanie, czyli pełną kolekcję wartości przechowywanych w swoich właściwościach. Egzemplarz jest to rodzaj urządzenia przeznaczonego do zachowania *informacji o stanie* i przypomina pudełko do przechowywania danych.

Słowo kluczowe self

Egzemplarz jest obiektem, a obiekt może otrzymywać komunikaty. Dlatego też egzemplarz musi mieć możliwość wysyłania komunikatów do samego siebie. Jest to możliwe dzięki użyciu słowa kluczowego `self`. Można je wykorzystać wszędzie tam, gdzie oczekiwany jest egzemplarz odpowiedniego typu.

Przyjmuję założenie, że głos wydawany przez szczekającego psa, np. `hau`, ma być przechowywany we właściwości. W takim przypadku implementacja metody `bark()` musi zawierać odwołanie do pewnej właściwości. Spójrz na przykładowe rozwiązanie.

```
class Dog {
    var name = ""
    var whatADogSays = "hau"
    func bark() {
        print(self.whatADogSays)
    }
}
```

Teraz przyjmuję założenie o konieczności utworzenia metody egzemplarza `speak()` będącej po prostu synonimem dla metody `bark()`. Dlatego też implementacja metody `speak()` może się składać jedynie z wywołania metody `bark()`, jak pokazałem w kolejnym fragmencie kodu.

```
class Dog {
    var name = ""
    var whatADogSays = "hau"
    func bark() {
        print(self.whatADogSays)
    }
    func speak() {
        self.bark()
    }
}
```

Zwróć uwagę na to, że w tym przykładzie słowo kluczowe `self` występuje jedynie w metodach egzemplarza. Gdy w kodzie egzemplarza znajduje się wywołanie `self`, odwołuje się ono do *danego* egzemplarza. Natomiast jeśli wyrażenie `self.name` pojawia się w kodzie metody egzemplarza klasy `Dog`, oznacza to odwołanie do właściwości `name` *danego* egzemplarza klasy `Dog`, który aktualnie wykonuje ten fragment kodu.

Okazuje się, że każde przedstawione tutaj użycie słowa kluczowego `self` jest całkowicie opcjonalne. Można je pominąć, a kod będzie działał w dokładnie taki sam sposób.

```
class Dog {
    var name = ""
    var whatADogSays = "hau"
    func bark() {
        print(whatADogSays)
    }
    func speak() {
        bark()
    }
}
```

Jeżeli pominiesz odbiorcę komunikatu, a on sam może być wysłany do `self`, wówczas kompilator uznaje, że komunikat jest przeznaczony dla `self`. Mimo to *nigdy* (chyba że na skutek błędu) nie polegam na tym rozwiązaniu. Ze względu na styl wolę wyraźnie używać słowa kluczowego `self`. Kod pozbawiony jawnego użycia `self` staje się trudniejszy w odczycie i do zrozumienia. Ponadto zdarzają się sytuacje, w których trzeba wyraźnie użyć słowa kluczowego `self`. Dlatego też preferuję jego stosowanie, gdy mogę to zrobić.

Prywatność

Wcześniej wspomniałem, że przestrzeń nazw sama w sobie nie stanowi bariery uniemożliwiającej dostęp do znajdujących się w niej elementów. Jednak istnienie takiej bariery jest czasami pożądane. Na przykład nie wszystkie dane przechowywane przez egzemplarz są przeznaczone do modyfikowania lub nawet do ich widzenia przez inny egzemplarz. Podobnie nie każda metoda egzemplarza powinna być wywoływana przez inne egzemplarze. Porządny język programowania zorientowanego obiektowo powinien zapewniać elementom składowym egzemplarza pewną *prywatność* — utrudniać innym obiektom dotarcie do tych elementów, jeśli mają pozostać niewidoczne.

Spójrz na kolejny fragment kodu.

```
class Dog {
    var name = ""
    var whatADogSays = "hau"
```

```
    func bark() {
        print(self.whatADogSays)
    }
    func speak() {
        print(self.whatADogSays)
    }
}
```

W tym przykładzie inny obiekt może zmienić wartość właściwości `whatADogSays`. Skoro ta właściwość jest używana przez metody `bark()` i `speak()`, po jej modyfikacji można otrzymać egzemplarz przedstawiający psa, który wydaje głos `miau`. Takie rozwiązanie jest całkowicie niepożądane.

```
let dog1 = Dog()
dog1.whatADogSays = "miau"
dog1.bark() // Miau.
```

Mógłbyś w tym miejscu powiedzieć: trzeba było nie deklarować `whatADogSays` za pomocą słowa kluczowego `var` i zamiast niego użyć `let`. Utwórz stałą, a nikt nie będzie mógł jej zmienić.

```
class Dog {
    var name = ""
    let whatADogSays = "hau"
    func bark() {
        print(self.whatADogSays)
    }
    func speak() {
        print(self.whatADogSays)
    }
}
```

Na pierwszy rzut oka wydaje się to dobrym rozwiązaniem. Takie rozwiązanie jednak rodzi dwa problemy. Co zrobić w sytuacji, gdy *sam* egzemplarz klasy `Dog` ma mieć możliwość zmiany *własnej* właściwości `whatADogSays` za pomocą polecenia `self.whatADogSays`? W takim przypadku właściwość `whatADogSays` musi być zdefiniowana za pomocą słowa kluczowego `var`, ponieważ w przeciwnym razie nawet egzemplarz nie będzie mógł zmienić swojej właściwości. Przyjmuję również założenie, że nie chcemy, aby inne obiekty *poznały* wartość właściwości `whatADogSays`, jeśli nie zostanie wywołana metoda `bark()` lub `speak()`. Mimo zadeklarowania tej właściwości za pomocą słowa kluczowego `let` inne obiekty nadal mogą odczytywać jej wartość. To nie zawsze będzie dobrym wyjściem.

Aby pomóc w rozwiązaniu tego problemu, Swift oferuje słowo kluczowe `private`. Do dokładnego wyjaśnienia sposobu jego działania wrócę w dalszej części książki. W tej chwili wystarczy wiedzieć, że zdefiniowanie elementu za pomocą słowa kluczowego `private` stanowi rozwiązanie wymienionego wcześniej problemu.

```
class Dog {
    var name = ""
    private var whatADogSays = "hau"
    func bark() {
        print(self.whatADogSays)
    }
    func speak() {
        print(self.whatADogSays)
    }
}
```

Teraz `name` to właściwość publiczna, a `whatADogSays` to właściwość prywatna, która pozostaje niedostępna dla innych typów obiektu. Egzemplarz klasy `Dog` może użyć polecenia `self.whatADogSays`, natomiast egzemplarz klasy `Cat` wraz z odwołaniem `fido` do egzemplarza klasy `Dog` nie ma możliwości wywołania `fido.whatADogSays`. Trzeba koniecznie pamiętać o tym, że elementy składowe obiektu są domyślnie publiczne. Jeżeli chcesz zapewnić im pewną prywatność, musisz to wyraźnie zadeklarować za pomocą słowa kluczowego `private`.

Słowa zarezerwowane

Pewne określenia — np.: `class`, `func`, `var`, `let`, `if`, `private` i `import` — są w Swifcie *zarezerwowane* i stanowią część języka. Dlatego też nie można ich używać w charakterze *identyfikatorów*, czyli jako nazw dla klasy, funkcji, zmiennej itd. Jeżeli spróbujesz w taki sposób użyć nazwy zarezerwowanej, kompilator wygeneruje komunikat błędu.

Aby wymusić użycie słowa zarezerwowanego jako identyfikator, ujmij je w odwrotne apostrofy. Przedstawiony tutaj (wyjątkowo mało czytelny) kod jest prawidłowy.

```
class `func` {
    func `if`() {
        let `class` = 1
    }
}
```

Deklaracja klasy definiuje przestrzeń nazw. Uzyskanie dostępu do elementów znajdujących się w przestrzeni nazw wymaga od innych obiektów użycia składni z kropką. Inne obiekty nadal *mogą* odwoływać się do elementów znajdujących się w przestrzeni nazw, która sama w sobie nie stanowi żadnej bariery i nie ogranicza widoczności swoich elementów. Jeżeli chcesz ukryć element przed innymi obiektami, zadeklaruj go wraz z użyciem słowa kluczowego `private`.

Projekt

Jakiego rodzaju obiekty typu są wymagane przez program? Jakie metody i właściwości powinien mieć zdefiniowany obiekt typu? Jak ma zostać utworzony egzemplarz tego typu i do czego powinien być wykorzystywany? To nie są łatwe pytania i na każde z nich nie istnieje tylko jedna dobra odpowiedź. Programowanie oparte na obiektach można uznać za sztukę.

W rzeczywistości podczas programowania na platformie iOS wiele typów obiektów, z którymi pracujesz, to obiekty zdefiniowane przez Ciebie, a nie przez firmę Apple. Swift jest standardowo dostarczany z wieloma użytecznymi typami obiektów, np. `String` i `Int`. Po zaimportowaniu frameworka `UIKit` otrzymujesz dostęp do *ogromnej* liczby obiektów typu, z których każdy może znaleźć zastosowanie w tworzonym programie. Nie zajmujesz się tworzeniem żadnego z tych typów, więc ich projekt nie jest Twoim problemem. Zamiast tego musisz się nauczyć, jak z nich korzystać. Przygotowane przez Apple'a obiekty typów mają na celu dostarczenie *ogólnej* funkcjonalności, która może być wymagana przez dowolną aplikację. Jednocześnie tworzona aplikacja prawdopodobnie będzie miała *konkretną* funkcjonalność unikatową dla jej przeznaczenia i trzeba będzie opracować odpowiednie obiekty typów.

Projekt programu opartego na obiektach musi bazować na bezpiecznych fundamentach natury obiektów. Konieczne jest opracowanie typów obiektu hermetyzujących odpowiednią funkcjonalność (metody) w połączeniu z odpowiednim zestawem danych (właściwości). Po utworzeniu egzemplarzy obiektów trzeba się upewnić, że mają właściwy cykl życiowy, wystarczającą widoczność względem siebie i odpowiednie możliwości w zakresie wzajemnej komunikacji.

Obiekt typu i API

Pliki tworzonego programu będą miały naprawdę niewiele funkcji i zmiennych najwyższego poziomu (albo nie będą ich miały w ogóle). Metody i właściwości obiektu typu — w szczególności metody i właściwości egzemplarza — to te, w których będzie wywoływanych najwięcej akcji. Obiekt typu daje każdemu rzeczywistemu egzemplarzowi jego specjalizowane możliwości. Pomagają one również w sensownym i przejrzystym zorganizowaniu kodu programu.

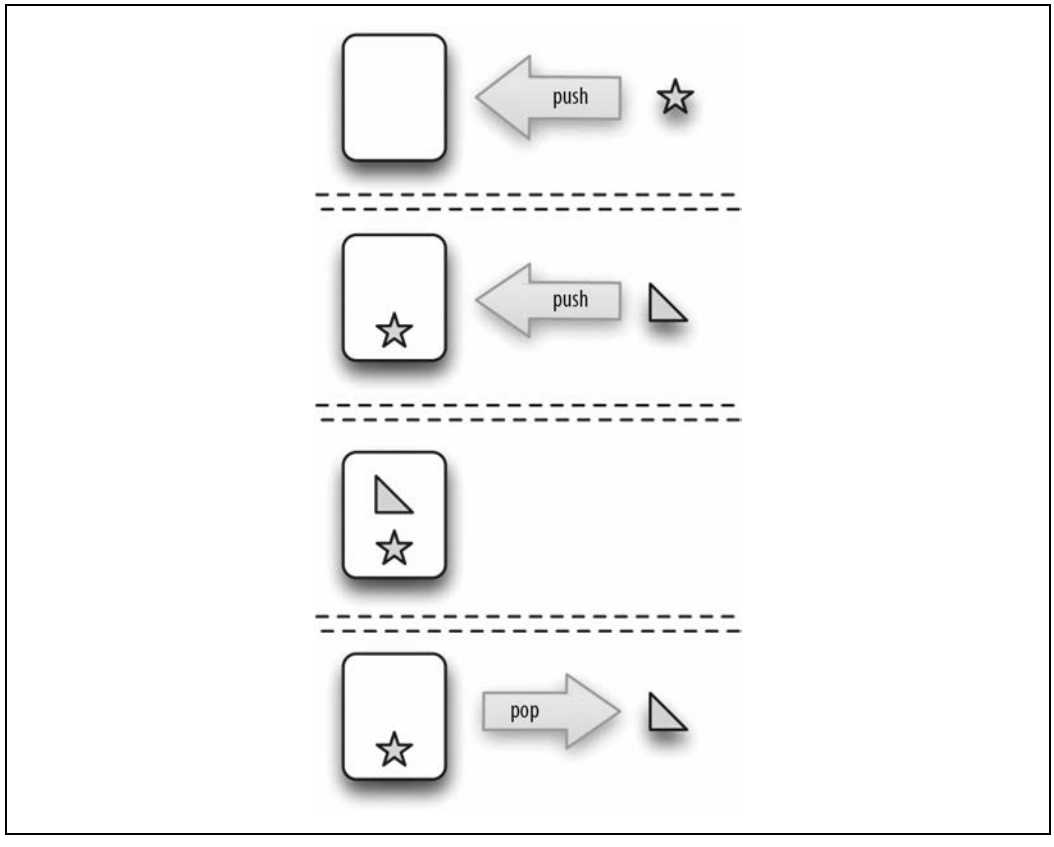
Naturę obiektów można podsumować za pomocą dwóch wyrażeń: hermetyzacja funkcjonalności i zachowanie informacji o stanie. (Po raz pierwszy takiego podsumowania użyłem wiele lat temu w mojej książce *REALbasic: The Definitive Guide*).

Hermetyzacja funkcjonalności
: Każdy obiekt wykonuje swoje zadanie i przekazuje wynik do świata zewnętrznego — innym obiektom, a w pewnym sensie także programiście — znajdującego się za nieprzezroczystą ścianą, w której jedyne drzwi mają postać metod. Obiekt obiecuje reagować na te metody, a akcje są podejmowane po przekazaniu tym metodom właściwych komunikatów. Szczegóły działania obiektu pozostają ukryte w jego wnętrzu i nikt z zewnątrz nie zna rzeczywistego sposobu implementowania akcji. Szczerze mówiąc, żaden inny obiekt nie musi znać takich szczegółów.

Zachowanie informacji o stanie
: Każdy egzemplarz zawiera w sobie pewną ilość danych. Bardzo często są to dane prywatne, pozostają więc hermetyzowane i żaden inny obiekt nie wie o ich istnieniu lub postaci. Jedynym sposobem na otrzymanie z zewnątrz dostępu do danych prywatnych jest zdefiniowanie w przechowującym je obiekcie metody lub właściwości publicznej.

Wyobraź sobie obiekt, którego zadaniem jest implementacja stosu — może to być egzemplarz klasy `Stack`. *Stos* to struktura danych przechowująca pewien zbiór danych z użyciem LIFO (ang. *first in, first out*), czyli pierwszy na wejściu, pierwszy na wyjściu. Ten obiekt odpowiada na dwa komunikaty: `push` i `pop`. Komunikat `push` oznacza umieszczenie określonego fragmentu danych na stosie. Z kolei `pop` oznacza usunięcie ze stosu elementu umieszczonego na nim jako ostatni. Można to porównać do stosu naczyń: układasz je pojedynczo jedno na drugim, a znajdujące się na samym dole nie będzie mogło być użyte, aż do chwili zdjęcia z niego wszystkich położonych na nim (rysunek 1.3).

Obiekt stosu przedstawia hermetyzację funkcjonalności, ponieważ inne obiekty nie mają żadnych informacji na temat sposobu rzeczywistej implementacji stosu. Może to być tablica, lista lub jakakolwiek inna implementacja. Jednak obiekt klienta — czyli ten faktycznie wysyłający komunikaty `push` i `pop` do obiektu stosu — nie ma żadnej wiedzy dotyczącej sposobu implementacji stosu. Prawdę mówiąc, nawet nie potrzebuje takich informacji, pod warunkiem że obiekt stosu działa tak, jak powinien funkcjonować stos. To jest dobre także dla programisty, który w trakcie pracy nad programem może bezpiecznie zastępować jedną implementację inną bez obaw o uszkodzenie programu jako całości.



Rysunek 1.3. Przykład stosu

Obiekt stosu przedstawia również zachowanie informacji o stanie, ponieważ stos to nie tylko brama do jego danych, ale *rzeczywiste* dane. Inne obiekty mogą uzyskać dostęp do tych danych, ale jedynie poprzez sam obiekt stosu i tylko w sposób umożliwiany przez obiekt stosu. Dane stosu znajdują się wewnątrz obiektu i nikt nie może ich zobaczyć. Wszystkie inne obiekty muszą używać komunikatów `push` i `pop`.

Wszystkie komunikaty, które mogą być wysyłane przez inne obiekty — razem te komunikaty noszą nazwę *interfejsu programowania aplikacji* (ang. *application programming interface* — API) — przypominają listę lub menu zadań, o których wykonanie można poprosić obiekt. Obiekty typów dzielą kod, natomiast ich API stanowią podstawę komunikacji między poszczególnymi częściami kodu. To samo dotyczy obiektów nieutworzonych przez Ciebie. Dokumentacja opracowanego przez Apple'a frameworka Cocoa składa się z listy API obiektów. Na przykład jeśli chcesz poznać komunikaty możliwe do wysyłania egzemplarzowi `NSString`, powinieneś zapoznać się z dokumentacją klasy `NSString`. Ta strona to w rzeczywistości długa lista metod i właściwości przedstawiających możliwości obiektu klasy `NSString`. Dlatego też dostarcza większość informacji, które będą Ci potrzebne, aby w tworzonych programach móc używać klasy `NSString`.

Tworzenie, zasięg i cykl życiowy egzemplarzy

Najważniejsze encje w programie Swifta to przede wszystkim egzemplarze. Obiekty typów definiują *rodzaje* egzemplarzy możliwe do utworzenia i ich zachowanie. Jednak rzeczywiste egzemplarze tych typów określają poszczególne zadania wykonywane przez program. Z kolei właściwości i metody egzemplarza pozwalają na zdefiniowanie komunikatów, które mogą być wysyłane egzemplarzom. Dlatego też program *musi* zawierać pewne egzemplarze, aby mógł *cokolwiek* zrobić.

Jednak domyślnie w programie nie ma *żadnych* egzemplarzy. Spójrz raz jeszcze na listing 1.1, na którym zdefiniowałeś obiekt pewnego typu, choć nie utworzyłeś żadnych jego egzemplarzy. Jeżeli uruchomisz ten program, dostępny będzie jedynie obiekt typu. W ten sposób utworzyłeś pewien świat możliwości, w którym *mogą* istnieć wskazane obiekty. W takim świecie tak naprawdę nic się *nie zdarzy*.

Egzemplarze nie pojawiają się na skutek działania jakichś magicznych sił. Konieczne jest utworzenie egzemplarza, aby on powstał i mógł zostać użyty. Spora liczba operacji wykonywanych przez program obejmuje tworzenie egzemplarzy typów. Ponieważ te egzemplarze mają być trwałe, każdy nowo utworzony zostaje przypisany zmiennej (umieszczony w pudełku), otrzymuje nazwę i cykl życiowy. Egzemplarz będzie *istniał* zgodnie z cyklem życiowym zmiennej, która się do niego odwołuje. Ponadto egzemplarz pozostanie *widoczny* dla innych zgodnie z zasięgiem zmiennej, która się do niego odwołuje.

Spora część sztuki związanej z programowaniem opartym na obiektach wiąże się z definiowaniem egzemplarzom wystarczająco długiego cyklu życiowego i udostępnianiem jednych obiektów innym. Bardzo często będziesz umieszczać egzemplarz w określonym pudełku — przypiszesz go pewnej zmiennej, zadeklarujesz w ustalonym zasięgu. Dzięki regułom cyklu życiowego zmiennej i zasięgu ten egzemplarz *będzie istniał* wystarczająco długo, aby był użyteczny dla programu. Jeżeli egzemplarz będzie potrzebny, inny kod może *pobrać do niego odwołanie* i później prowadzić komunikację z tym obiektem.

Planowanie sposobu tworzenia egzemplarzy, ustalenie ich cyklu życiowego i zdefiniowanie komunikacji między egzemplarzami może wydawać się żmudne. Na szczęście w rzeczywistości, gdy zajmujesz się tworzeniem aplikacji na platformę iOS, framework Cocoa dostarcza początkowy szkielet programu. Zanim utworzysz choćby jeden wiersz kodu, framework gwarantuje, że po uruchomieniu aplikacji przez cały czas jej działania będą istniały w niej pewne egzemplarze. W ten sposób otrzymujesz widoczny interfejs aplikacji i miejsce początkowe, w którym możesz rozpocząć tworzenie własnych egzemplarzy i nadawanie im wystarczająco długich cykli życiowych.

Podsumowanie

Jak możesz sobie wyobrazić, podczas tworzenia zorientowanego obiektowo programu przeznaczonego do wykonania konkretnego zadania trzeba wziąć pod uwagę naturę obiektów. Są to typy i egzemplarze. Typ to zbiór metod opisujących to, co będą mogły zrobić wszystkie jego egzemplarze (hermetyzacja funkcjonalności). Zadania powinieneś odpowiednio zaplanować. Obiekty to narzędzie organizacyjne, zbiór pudełek przeznaczonych do hermetyzacji kodu odpowiedzialnego

za wykonanie określonego zadania. To także narzędzie koncepcyjne. Programista zmuszony do myślenia w kategoriach oddzielnych obiektów musi podzielić cele i zachowania programu na poszczególne zadania, a następnie każde z nich przypisać odpowiedniemu obiektowi.

Jednocześnie żaden obiekt nie jest bezludną wyspą. Obiekty mogą współpracować ze sobą, czyli komunikować się, co oznacza wysyłanie wiadomości. Mamy wręcz nieograniczoną liczbę sposobów, na które można przygotować linie komunikacji. Przygotowanie odpowiedniego rozwiązania — tzw. *architektury* — zapewniającego możliwość współpracy i komunikacji między obiektami to jeden z najbardziej wymagających aspektów programowania opartego na obiektach. W przypadku programowania na platformie iOS otrzymujesz ogromną pomoc ze strony frameworka Cocoa, który dostarcza początkowy zbiór obiektów typu i praktyczną architekturę podstawowej aplikacji.

Efektywne wykorzystanie opartego na obiektach programowania do utworzenia programu wykonującego żądane zadania i jednocześnie zachowanie przejrzystości i łatwości jego rozbudowy jest sztuką samą w sobie. Twoje możliwości w tym zakresie będą się poprawiały wraz z doświadczeniem. Ostatecznie będziesz chciał sięgnąć po inne pozycje dotyczące efektywnego planowania i przygotowywania architektury programu opartego na obiektach. Mogę w tym miejscu polecić dwie klasyczne, świetne książki. Pierwsza z nich, *Refaktoryzacja. Ulepszanie struktury istniejącego kodu*, napisana przez Martina Fowlera (wydana przez Helion w 2011 r.), wyjaśnia powody, dla których chciałbyś zmodyfikować metody należące do klas, i pokazuje, jak pokonać strach przed zrobieniem tego. Druga, *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku*, napisana przez Ericha Gammę, Richarda Helma, Ralpha Johnsona i Johna Vlissidesa, znanych także jako „Banda Czterech" (wydana przez Helion w 2010 r.), może być uznana za biblię dotyczącą przygotowywania programów opartych na obiektach. Przedstawia wszystkie sposoby, na jakie można korzystać z obiektów na podstawie ich możliwości.

Skorowidz

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

X

Z

Ź

O'REILLY®

Swift. Doskonałe narzędzie — znakomite efekty!

Język Swift poznaliśmy w 2014 roku. Został opracowany przez Apple specjalnie dla platormy iOS z uwzględnieniem takich założeń jak przejrzystość, bezpieczeństwo, prostota, łatwość stosowania, zorientowanie obiektowe. Kod w tym języku sam zarządza pamięcią i zapewnia ścisłą kontrolę typów. Swift od początku wzbudzał ogromne zainteresowanie wśród programistów, którzy prędko docenili jego zalety. Dziś jest uważany za łatwe do opanowania i bardzo wygodne narzędzie dla profesjonalistów, zwłaszcza że od pewnego czasu Apple wraz ze Swiftem dostarcza aplikację Xcode oraz framework Cocoa. To wszystko sprawiło, że Swift stał się atrakcyjną alternatywą dla Objective-C.

Ta książka zawiera solidne wprowadzenie do tworzenia aplikacji na platformie iOS. Znalazły się tu systematycznie przedstawione informacje na temat Swifta, Xcode i frameworka Cocoa. Podstawy języka wyjaśniono w najużyteczniejszej kolejności, skoncentrowano się bowiem na najczęściej stosowanych i najpraktyczniejszych aspektach Swifta. Sporo miejsca poświęcono środowisku Xcode, w którym odbywa się programowanie na platformie iOS. Omówiono, czym jest projekt, jak zmienić go na aplikację, jak tworzyć, uruchamiać i debugować kod źródłowy, a także jak zgłosić aplikację do sklepu App Store. Bardzo ważną częścią książki jest wprowadzenie do Cocoa Touch, który zapewnia najważniejsze klasy podstawowe, kategorie, protokoły, mechanizmy delegowania i powiadamiania, a także zarządzanie pamięcią.

W tej książce między innymi:

- solidne podstawy koncepcji Swifta
- najnowsze funkcje dostępne podczas programowania na iOS
- cykl życiowy projektu Xcode
- komunikacja między Swiftem a Objective-C
- programowanie techniką klucz-wartość

Matt Neuburg — zaczął programować w 1968 roku jako czternastolatek. Swoją rozprawę doktorską o Ajschylosie napisał w 1981 roku z użyciem komputera typu mainframe na Uniwersytecie Cornella. Uczył języków klasycznych, literatury i kultury na kilku znakomitych uczelniach. W tym czasie wciąż interesował się technologiami informatycznymi, a w 1990 roku przeszedł na platformę Macintosh. Opracował kilka bezpłatnych programów edukacyjnych i użytkowych, redagował serwis internetowy TidBITS oraz magazyn MacTech. Jest autorem kilku książek i wielu artykułów branżowych.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-5257-5
9 788328 352575

Cena: 99,00 zł