

Praktyczny podręcznik  
tworzenia aplikacji  
na systemy  
iOS i Mac OS X!

▼  
*Techniki definiowania klas i sposoby  
wysyłania komunikatów do obiektów*

▼  
*Praca z typami danych, pętlami,  
klasami i obiektami*

▼  
*Korzystanie z bibliotek Foundation  
oraz Cocoa i Cocoa Touch*



Wydanie III

# Objective-C Programowanie

*Vademecum profesjonalisty*

  
ADDISON-WESLEY

Stephen G. Kochan

Helion 

Tytuł oryginału: Programming in Objective-C, Third Edition

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-3667-9

Authorized translation from the English language edition, entitled: Programming in Objective-C, Third Edition, ISBN 0321711394, by Stephen G. Kochan, published by Pearson Education, Inc, publishing as Addison Wesley, Copyright © 2011 by Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education Inc.

Polish language edition published by Helion S.A.  
Copyright © 2012.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION  
ul. Kościuszki 1c, 44-100 GLIWICE  
tel. 32 231 22 19, 32 230 98 63  
e-mail: [helion@helion.pl](mailto: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?gimpbi>  
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

# Spis treści

	<b>O autorze .....</b>	<b>11</b>
	<b>O korektorach merytorycznych .....</b>	<b>12</b>
<b>Rozdział 1.</b>	<b>Wprowadzenie .....</b>	<b>13</b>
	Zawartość książki .....	14
	Struktura książki .....	15
	Pomoc .....	17
	Podziękowania .....	17
<b>Część I</b>	<b>Język Objective-C 2.0 .....</b>	<b>19</b>
<b>Rozdział 2.</b>	<b>Programowanie w języku Objective-C .....</b>	<b>21</b>
	Kompilacja i uruchamianie programów .....	21
	Kompilacja programów przy użyciu Xcode .....	22
	Kompilacja programów przy użyciu terminala .....	28
	Objaśnienie kodu pierwszego programu .....	30
	Wyświetlanie wartości zmiennych .....	34
	Podsumowanie .....	36
<b>Rozdział 3.</b>	<b>Klasy, obiekty i metody .....</b>	<b>39</b>
	Czym tak właściwie jest obiekt .....	39
	Egzemplarze i metody .....	40
	Klasa w języku Objective-C do działań na ułamkach .....	42
	Sekcja @interface .....	45
	Wybór nazw .....	45
	Zmienne egzemplarza .....	47
	Metody klas i egzemplarzy .....	47
	Sekcja @implementation .....	49
	Sekcja programu .....	50
	Zasady dostępu do zmiennych egzemplarzy i hermetyzacja danych .....	56
	Podsumowanie .....	59
<b>Rozdział 4.</b>	<b>Typy danych i wyrażenia .....</b>	<b>61</b>
	Typy danych i stałe .....	61
	Typ int .....	61
	Typ float .....	62
	Typ char .....	62
	Kwalifikatory: long, long long, short, unsigned oraz signed .....	63
	Typ id .....	64

Wyrażenia arytmetyczne .....	65
Kolejność wykonywania działań .....	65
Arytmetyka liczb całkowitych i jednoargumentowy operator minus .....	68
Operator dzielenia modulo .....	70
Konwersja między typami całkowitymi i zmiennoprzecinkowymi .....	71
Operator rzutowania typów .....	72
Operatory przypisania .....	73
Klasa kalkulatora .....	74
<b>Rozdział 5. Pętle .....</b>	<b>79</b>
Pętla for .....	80
Odbieranie danych z klawiatury .....	86
Zagnieżdżanie pętli for .....	88
Warianty pętli for .....	90
Instrukcja while .....	91
Instrukcja do .....	95
Instrukcja break .....	96
Instrukcja continue .....	97
Podsumowanie .....	97
<b>Rozdział 6. Podejmowanie decyzji .....</b>	<b>99</b>
Instrukcja if .....	99
Instrukcja if-else .....	103
Złożone testy relacyjne .....	106
Zagnieżdżanie instrukcji if .....	108
Konstrukcja else if .....	110
Instrukcja switch .....	118
Zmienne logiczne .....	121
Operator warunkowy .....	126
<b>Rozdział 7. Klasy raz jeszcze .....</b>	<b>129</b>
Pisanie interfejsu i implementacji w osobnych plikach .....	129
Automatyczne tworzenie metod dostępowych .....	134
Dostęp do właściwości za pomocą operatora kropki .....	135
Metody przyjmujące po kilka argumentów .....	136
Metody bez nazw argumentów .....	138
Działania na ułamkach .....	138
Zmienne lokalne .....	141
Argumenty metod .....	142
Słowo kluczowe static .....	142
Słowo kluczowe self .....	145
Alokacja i zwracanie obiektów przez metody .....	146
Rozszerzanie definicji klasy i pliku interfejsowego .....	151
<b>Rozdział 8. Dziedziczenie .....</b>	<b>153</b>
Początek wszystkiego .....	153
Wybieranie metody do wywołania .....	157
Rozszerzanie klas poprzez dziedziczenie — dodawanie nowych metod .....	157
Klasa reprezentująca punkt i alokacja pamięci .....	161
Dyrektywa @class .....	162
Klasy jako właściciele swoich obiektów .....	166
Przesłanie metod .....	169
Wybór metody .....	171
Przesłanie metody dealloc i słowa kluczowego super .....	173
Rozszerzanie poprzez dziedziczenie — dodawanie zmiennych egzemplarzowych .....	175
Klasy abstrakcyjne .....	177

<b>Rozdział 9. Polimorfizm oraz typowanie i wiązanie dynamiczne .....</b>	<b>181</b>
Polimorfizm — ta sama nazwa, różne klasy .....	181
Wiązanie dynamiczne i typ id .....	184
Sprawdzanie typów .....	186
Typ id i typowanie statyczne .....	187
Argumenty i typy zwrotne a typowanie dynamiczne .....	188
Pytanie o klasy .....	189
Obsługa wyjątków za pomocą instrukcji @try .....	193
<b>Rozdział 10. Zmienne i typy danych — techniki zaawansowane .....</b>	<b>197</b>
Inicjowanie obiektów .....	197
Zakres dostępności zmiennych — rozszerzenie wiadomości .....	200
Dyrektywy do kontroli zakresu dostępności zmiennych .....	200
Zmienne zewnętrzne .....	201
Zmienne statyczne .....	203
Wyliczenia .....	206
Instrukcja typedef .....	209
Konwersja typów .....	210
Reguły konwersji .....	210
Operatory bitowe .....	211
Operator bitowego I .....	213
Operator bitowego LUB .....	214
Bitowy operator LUB wykluczającego .....	214
Operator uzupełnienia jedynkowego .....	215
Operator przesunięcia w lewo .....	216
Operator przesunięcia w prawo .....	217
<b>Rozdział 11. Kategorie i protokoły .....</b>	<b>219</b>
Kategorie .....	219
Kilka uwag na temat kategorii .....	223
Protokoły i delegacja .....	224
Delegacja .....	227
Protokoły nieformalne .....	228
Obiekty złożone .....	229
<b>Rozdział 12. Preprocesor .....</b>	<b>233</b>
Instrukcja #define .....	233
Inne zaawansowane definicje typów .....	235
Instrukcja #import .....	239
Kompilacja warunkowa .....	241
Instrukcje #ifdef, #endif, #else oraz #ifndef .....	241
Instrukcje #if i #elif .....	243
Instrukcja #undef .....	244
<b>Rozdział 13. Dziedzictwo języka C .....</b>	<b>245</b>
Tablice .....	245
Inicjowanie elementów tablic .....	248
Tablice znaków .....	249
Tablice wielowymiarowe .....	250
Funkcje .....	251
Argumenty i zmienne lokalne .....	253
Wartość zwrotna funkcji .....	255
Funkcje, metody i tablice .....	258

Bloki .....	259
Struktury .....	263
Inicjowanie struktur .....	266
Struktury wewnątrz struktur .....	267
Struktury — uzupełnienie wiadomości .....	268
Nie zapomnij o programowaniu obiektowym! .....	271
Wskaźniki .....	271
Wskaźniki i struktury .....	275
Wskaźniki, metody i funkcje .....	277
Wskaźniki i tablice .....	278
Działania na wskaźnikach .....	288
Wskaźniki i adresy w pamięci .....	290
Unie .....	290
To nie są obiekty! .....	292
Różności .....	293
Literały złożone .....	293
Instrukcja goto .....	293
Instrukcja pusta .....	294
Operator przecinek .....	294
Operator sizeof .....	295
Argumenty wiersza poleceń .....	296
Jak to działa .....	298
Fakt 1. Zmienne egzemplarzowe są przechowywane w strukturach .....	298
Fakt 2. Zmienna obiektowa jest tak naprawdę wskaźnikiem .....	299
Fakt 3. Metody i funkcje oraz wyrażenia wysyłające komunikaty i wywołania funkcji ....	299
Fakt 4. Typ id to ogólny typ wskaźnikowy .....	299

## **Część II Biblioteka Foundation ..... 303**

### **Rozdział 14. Wprowadzenie do biblioteki Foundation ..... 305**

Dokumentacja biblioteki Foundation .....	305
--	-----

### **Rozdział 15. Liczby, łańcuchy i kolekcje ..... 309**

Obiekty liczbowe .....	309
Krótka dygresja na temat puli automatycznej .....	311
Obiekty łańcuchowe .....	314
Funkcja NSLog — rozszerzenie wiadomości .....	315
Metoda description .....	315
Obiekty zmienne i niezienne .....	316
Łańcuchy zmienne .....	322
Gdzie podziewają się te wszystkie obiekty? .....	326
Obiekty tablicowe .....	328
Budowa książki adresowej .....	331
Sortowanie tablic .....	348
Słowniki .....	355
Enumeracja słownika .....	356
Zbiory .....	358
Klasa NSMutableIndexSet .....	362

### **Rozdział 16. Praca z plikami ..... 367**

Praca z plikami i katalogami — klasa NSFileManager .....	368
Klasa NSData .....	372
Praca z katalogami .....	374
Sprawdzanie zawartości katalogów .....	376

Praca ze ścieżkami — plik NSPathUtilities.h .....	378
Najczęściej używane metody do pracy ze ścieżkami do plików .....	380
Kopiowanie plików i używanie klasy NSProcessInfo .....	382
Podstawowe operacje na plikach — klasa NSFileHandle .....	386
Klasa NSURL .....	390
Klasa NSBundle .....	391
<b>Rozdział 17. Zarządzanie pamięcią .....</b>	<b>395</b>
Pula automatyczna .....	395
Liczenie referencji .....	396
Liczenie referencji a łańcuchy .....	399
Zmienne egzemplarzowe .....	401
Przykład automatycznego zwalniania .....	407
Podsumowanie zasad zarządzania pamięcią .....	409
Pętla zdarzeń a alokacja pamięci .....	409
Znajdowanie wycieków pamięci .....	411
Usuwanie nieużytków .....	411
<b>Rozdział 18. Kopiowanie obiektów .....</b>	<b>415</b>
Metody copy i mutableCopy .....	416
Kopiowanie płytkie i głębokie .....	418
Implementacja protokołu <NSCopying> .....	420
Kopiowanie obiektów w metodach ustawiających i sprawdzających .....	423
<b>Rozdział 19. Archiwizacja .....</b>	<b>427</b>
Listy właściwości w formacie XML .....	427
Archiwizacja przy użyciu klasy NSKeyedArchiver .....	429
Pisanie metod kodujących i dekodujących .....	431
Tworzenie archiwów przy użyciu klasy NSData .....	438
Kopiowanie obiektów przy użyciu archiwizatora .....	441
<b>Część III Cocoa, Cocoa Touch i SDK dla systemu iOS .....</b>	<b>443</b>
<b>Rozdział 20. Wprowadzenie do Cocoa i Cocoa Touch .....</b>	<b>445</b>
Warstwy bibliotek .....	445
Cocoa Touch .....	446
<b>Rozdział 21. Pisanie programów dla systemu iOS .....</b>	<b>449</b>
Pakiet SDK dla systemu iOS .....	449
Pierwszy program dla systemu iOS .....	449
Tworzenie nowego projektu programu dla iPhone'a .....	451
Wpisywanie kodu .....	454
Projektowanie interfejsu .....	457
Kalkulator ułamków .....	463
Tworzenie projektu Fraction_Calculator .....	464
Definicja kontrolera widoku .....	467
Klasa Fraction .....	471
Klasa Calculator z obsługą ułamków .....	474
Projekt interfejsu użytkownika .....	476
Podsumowanie .....	476
<b>Dodatki .....</b>	<b>479</b>
<b>Dodatek A Słowniczek .....</b>	<b>481</b>
<b>Skorowidz .....</b>	<b>489</b>

## Rozdział 3.

# Klasy, obiekty i metody

W tym rozdziale poznasz podstawowe pojęcia programowania obiektowego oraz nauczysz się podstaw programowania w języku Objective-C przy użyciu klas. Konieczne będzie przyswojenie terminologii, ale nie będzie to nic strasznego. Objasnienia terminów specjalistycznych są ograniczone do niezbędnego minimum, aby nie przytłoczyć Cię nadmiarem informacji. Jeśli potrzebujesz dokładniejszych definicji, zajrzyj do dodatku A, zawierającego słowniczek pojęć.

## Czym tak właściwie jest obiekt

Obiekt to jakaś rzecz. Programowanie obiektowe polega w pewnym sensie na posiadaniu zbioru takich rzeczy i wykonywaniu na nich różnych operacji. Jest to całkiem odmienne podejście niż w języku C, który jest językiem proceduralnym. W języku C najpierw trzeba się zastanowić, co się chce zrobić, a dopiero potem martwi się o obiekty, czyli odwrotnie niż w programowaniu obiektowym.

Przestudiujemy przykład z życia codziennego. Przyjmijmy, że masz samochód, który oczywiście jest jakimś obiektem należącym do Ciebie. Ten samochód nie jest jakimś tam ogólnie pojazdem, lecz jest konkretnej marki i został gdzieś wyprodukowany, na przykład w Detroit, w Japonii albo jeszcze gdzieś indziej. Ma on też numer VIN, którym można się posługiwać do jego identyfikacji.

W terminologii obiektowej samochód, który posiadasz, nazywa się **egzemplarem** samochodu. Idąc tym tropem, samochód to **klasa**, z której utworzono ten egzemplarz. Zatem kiedy produkowany jest nowy samochód, następuje utworzenie kolejnego egzemplarza klasy samochód. Każdy taki egzemplarz nazywa się **obiektem**.

Samochód może być srebrny, mieć czarną tapicerkę, składany dach itd. Dodatkowo można z nim zrobić różne rzeczy, na przykład prowadzić go, tankować, myć (oby), naprawiać itd. Wszystko to przedstawiono w tabeli 3.1.

Wszystkie czynności wymienione w tabeli 3.1 można wykonać z Twoim samochodem, jak również z każdym innym. Na przykład Twoja siostra również prowadzi swój samochód, myje go, tankuje itd.



**Tabela 3.1.** Działania na obiektach

Obiekt	Działania na nim
[Twój samochód]	Prowadzenie Tankowanie Mycie Naprawianie

## Egzemplarze i metody

Konkretnym wystąpieniem klasy jest egzemplarz, a działania, jakie można na nim wykonywać, nazywają się **metodami**. Niektóre metody można wykonywać na rzecz wybranych egzemplarzy klasy, a inne także na rzecz samej klasy. Na przykład mycie samochodu dotyczy konkretnego egzemplarza (w istocie wszystkie czynności wymienione w tabeli 3.1 to metody egzemplarzowe). Natomiast metoda sprawdzająca, ile typów samochodów dany producent produkuje, miałaby zastosowanie do klasy, a więc byłaby metodą klasową.

Przypuśćmy, że mamy dwa samochody, które zeszły z linii produkcyjnej i są identyczne: oba mają takie same obicie wnętrza, ten sam kolor itd. Początkowo może i są takie same, ale każdy z nich ma innego właściciela, dzięki czemu zmieniają się jego właściwości, czyli **cechy**. Na przykład jeden samochód może mieć zadrapanie na błotniku, a drugi więcej przejechanych kilometrów. Każdy egzemplarz, czyli obiekt, zawiera informacje nie tylko o swoim stanie początkowym, lecz również o bieżących cechach, które mogą się dynamicznie zmieniać. Kiedy jeździsz samochodem, z baku ubywa paliwa, na aucie gromadzi się brud i zużywają się opony.

Zastosowanie metody do obiektu może spowodować zmianę jego **stanu**. Jeśli zastosuje się metodę „napelnij bak”, to po jej zakończeniu bak samochodu będzie pełny. Zatem metoda ta zmieni stan baku samochodu.

Najważniejsze jest to, że obiekty to konkretne reprezentacje klas. Każdy obiekt zawiera pewne informacje (dane), które są jego prywatną własnością. Do sprawdzania i zmieniania tych danych służą metody.

W języku Objective-C do wywoływania metod na rzecz klas i egzemplarzy służy poniższa składnia:

```
[ KlasaLubEgzemplarz metoda ];
```

Otwiera się nawias kwadratowy, wpisuje nazwę klasy lub egzemplarza klasy, przynajmniej jedną spację oraz nazwę metody, która ma być wywołana. Na koniec zamyka się nawias i stawia kończący instrukcję średnik. Wydanie instrukcji wykonania przez klasę lub obiekt jakiejś czynności nazywa się wysłaniem do klasy lub obiektu **komunikatu**. Odbiorca komunikatu to **adresat**. W związku z tym przedstawioną powyżej składnię można także wyrazić w następujący sposób:

```
[ adresat komunikat ] ;
```

Wrócimy do poprzedniej listy i napiszemy wszystko przy użyciu tej nowej składni. Zanim jednak to zrobimy, musimy udać się do fabryki po nowy samochód. Służy do tego następująca instrukcja:

```
yourCar = [Car new];      Tworzenie nowego samochodu
```

Wysłano komunikat `new` do klasy `Car` (adresata komunikatu), aby poprosić ją o wydanie nowego samochodu. W odpowiedzi zostaje utworzony nowy obiekt (reprezentujący nasz samochód), który zostaje zapisany w zmiennej `yourCar`. Od tej pory za pomocą zmiennej `yourCar` można odwoływać się do naszego nowego samochodu, który otrzymaliśmy z fabryki.

Ponieważ aby dostać samochód, udaliśmy się do fabryki, `new` nazywana jest **metodą fabryczną** albo **klasową**. Pozostałe czynności, które będziesz wykonywać na swoim samochodzie, będą metodami egzemplarzowymi, ponieważ będą dotyczyć Twojego egzemplarza samochodu. Oto kilka przykładowych komunikatów, które można by napisać dla samochodu:

```
[yourCar prep];          Przygotowanie do pierwszego użycia
[yourCar drive];         Prowadzenie samochodu
[yourCar wash];          Mycie samochodu
[yourCar getGas];        Zatankowanie samochodu, jeśli jest to konieczne
[yourCar service];       Naprawa samochodu

[yourCar topDown];       Czy samochód jest kabrioletem
[yourCar topUp];
currentMileage = [yourCar odometer];
```

Ostatni wiersz zawiera przykładową metodę egzemplarzową zwracającą informację — zapewne przebieg zgodnie ze wskazaniem licznika. Informacja ta w tym przypadku jest przechowywana w zmiennej o nazwie `currentMileage`.

Oto przykład wywołania metody pobierającej **argument** określający konkretną wartość, która w każdym wywołaniu może być inna:

```
[yourCar setSpeed: 55];  Ustawienie prędkości na 55 mph
```

Twoja siostra Sue może użyć tych samych metod na rzecz swojego samochodu:

```
[suesCar drive];
[suesCar wash];
[suesCar getGas];
```

Możliwość stosowania tych samych metod do różnych obiektów jest jednym z fundamentów programowania obiektowego i dlatego później jeszcze do tego wrócimy.

Najprawdopodobniej w swoich programach nie spotkasz się jednak z obiektami samochodów. Będą to raczej rzeczy ze świata komputerów, a więc okna, prostokąty, fragmenty tekstu albo kalkulatory czy listy odtwarzania muzyki. Metody w tych wszystkich przypadkach mogą być podobne do przedstawionych powyżej metod dla samochodów, na przykład:

<code>[myWindow erase];</code>	<i>Czyści okno</i>
<code>theArea = [myRect area];</code>	<i>Oblicza pole powierzchni prostokąta</i>
<code>[userText spell]Check];</code>	<i>Sprawdza pisownię tekstu</i>
<code>[deskCalculator clearEntry];</code>	<i>Usuwa ostatni wpis</i>
<code>[favoritePlaylist showSongs];</code>	<i>Wyświetla piosenki z listy ulubionych</i>
<code>[phoneNumber dial];</code>	<i>Wybiera podany numer telefonu</i>
<code>[myTable reloadData];</code>	<i>Wyświetla zaktualizowane dane w tabeli</i>
<code>n = [aTouch tapCount];</code>	<i>Wyświetla, ile razy stuknięto w wyświetlacz</i>

## Klasa w języku Objective-C do działań na ułamkach

Czas zbudować prawdziwą klasę i nauczyć się pracować z egzemplarzami.

Ponieważ najpierw przedstawię samą procedurę, przykładowe programy mogą wydawać się mało praktyczne. Do bardziej praktycznych rzeczy dojdziemy nieco później.

Załóżmy, że chcesz napisać program do wykonywania działań na ułamkach — dzielenia, mnożenia, odejmowania itp. Ktoś, kto nie potrafi tworzyć klas, mógłby napisać program podobny do przedstawionego na listingu 3.1:

### Listing 3.1.

---

```
// Prosty program do wykonywania działań na ułamkach

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int numerator = 1;
    int denominator = 3;
    NSLog(@"Ułamek wynosi %i/%i", numerator, denominator);

    [pool drain];
    return 0;
}
```

---

### Listing 3.1 — wynik

---

```
Ułamek wynosi 1/3
```

---

W programie przedstawionym na listingu 3.1 ułamki są reprezentowane przy użyciu licznika i mianownika. Znajdujące się za instrukcją tworzącą pulę automatyczną dwie instrukcje definiują zmienne całkowitoliczbowe `numerator` oraz `denominator` i przypisują im wartości początkowe 1 oraz 3. Kod tych deklaracji można by również zapisać tak:

```
int numerator, denominator;

numerator = 1;
denominator = 3;
```

Ułamek  $1/3$  został zaprezentowany poprzez zapisanie wartości 1 w zmiennej `numerator` i 3 w zmiennej `denominator`. Gdyby w programie trzeba było zapisać dużą liczbę ułamków, ta metoda byłaby bardzo uciążliwa. Każdorazowo, gdy chcielibyśmy odwołać się do jakiegoś ułamka, trzeba by było odwołać się zarówno do jego licznika, jak i mianownika. Również wykonywanie działań byłoby dość nieporadne.

Znacznie lepiej by było, gdyby można było definiować ułamki jako pojedyncze obiekty i odwoływać się do ich liczników i mianowników za pomocą jednej nazwy, na przykład `myFraction`. Można to zrealizować w języku Objective-C, ale potrzebna jest do tego klasa.

Program pokazany na listingu 3.2 robi to samo co program z listingu 3.1, ale przy użyciu nowej klasy o nazwie `Fraction`. Poniżej znajduje się jego kod źródłowy, a dalej szczegółowy opis jego działania.

### Listing 3.2.

```
// Program do wykonywania działań na ułamkach — wersja z użyciem klasy

#import <Foundation/Foundation.h>

//--- Sekcja @interface ---

@interface Fraction: NSObject
{
    int numerator;
    int denominator;
}

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;

@end

//--- Sekcja @implementation ---

@implementation Fraction
-(void) print
{
    NSLog(@"%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

@end
```

```
//--- Sekcja programu ---  
  
int main (int argc, char *argv[])  
{  
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];  
    Fraction *myFraction;  
  
    // Tworzenie egzemplarza klasy Fraction  
  
    myFraction = [Fraction alloc];  
    myFraction = [myFraction init];  
  
    // Ustawienie ułamka na 1/3  
  
    [myFraction setNumerator: 1];  
    [myFraction setDenominator: 3];  
  
    // Wyświetlenie ułamka przy użyciu metody print  
  
    NSLog(@"Wartość mojego ułamka wynosi:");  
    [myFraction print];  
    [myFraction release];  
  
    [pool drain];  
    return 0;  
}
```

---

**Listing 3.2 — wynik**

---

```
Wartość mojego ułamka wynosi:  
1/3
```

---

Zgodnie z komentarzami zamieszczonymi w kodzie źródłowym program jest podzielony na trzy logiczne części:

- sekcję @interface,
- sekcję @implementation,
- sekcję programu.

W sekcji @interface znajduje się opis klasy oraz jej danych i metod. Natomiast w sekcji @implementation zamieszczono kod implementacji tych metod. W sekcji programu znajduje się kod źródłowy wykonujący zadanie, do którego program został stworzony.

Każdy program w języku Objective-C ma wszystkie te sekcje, chociaż nie zawsze trzeba je wpisywać własnoręcznie. Jak się niebawem przekonasz, zazwyczaj są one zapisywane w osobnych plikach. Na razie jednak wszystko będziemy trzymać w jednym pliku.

## Sekcja @interface

Aby zdefiniować klasę, należy wykonać kilka czynności. Przede wszystkim trzeba poinformować kompilator, skąd ona pochodzi, tzn. wyznaczyć jej klasę **macierzystą**. Po drugie, należy określić, jakiego typu dane będą przechowywane w obiektach tej klasy, a więc opisać dane, które będą przechowywane w składowych klasy. Składowe te nazywają się **zmiennymi egzemplarza**. Na koniec trzeba jeszcze zdefiniować typy operacji, czyli **metody**, których będzie można używać do pracy z obiektami tej klasy. Wszystko to robi się w specjalnej sekcji programu o nazwie @interface. Jej ogólna postać jest następująca:

```
@interface NazwaNowejKlasy: NazwaKlasyMacierzystej
{
    deklaracjeSkładowych;
}

deklaracjeMetod;
@end
```

Tradycyjnie nazwy klas zaczyna się wielką literą, mimo iż nie ma takiego formalnego wymogu. Dzięki temu programista czytający kod źródłowy bez trudu odróżnia nazwy klas od innych rodzajów zmiennych — wystarczy, że spojrzy na pierwszą literę nazwy. Zrobimy teraz krótką dygresję od głównego tematu i zastanowimy się nad wyborem nazw w języku Objective-C.

## Wybór nazw

W rozdziale 2. zostało użytych kilka zmiennych do przechowywania liczb całkowitych. Na przykład w programie przedstawionym na listingu 2.4 użyto zmiennej o nazwie `sum` do przechowywania wyniku dodawania liczb 50 i 25.

W języku Objective-C w zmiennych można zapisywać także inne typy danych, pod warunkiem że przed użyciem w programie odpowiednio się te zmienne zadeklaruje. Można przechowywać liczby zmiennoprzecinkowe, znaki, a nawet obiekty (a mówiąc dokładniej — referencje do obiektów).

Reguły tworzenia nazw są bardzo proste: nazwa musi zaczynać się od litery lub znaku podkreślenia (`_`), po których może wystąpić dowolna kombinacja wielkich i małych liter, znaków podkreślenia i cyfr. Wszystkie poniższe nazwy są poprawne:

- `sum`,
- `pieceFlag`,
- `i`,
- `myLocation`,
- `numberOfMoves`,
- `sysFlag`,
- `ChessBoard`.

Natomiast poniższe nazwy są niepoprawne, ponieważ są niezgodne z przedstawionymi wyżej zasadami:

- `sum$value` — znaku `$` nie można używać w nazwach.
- `piece flag` — w nazwach nie może być spacji.
- `3Spencer` — nazwy nie mogą się zaczynać od cyfr.
- `int` — to słowo zarezerwowane.

Słowa `int` nie można użyć jako nazwy, ponieważ ma ono dla kompilatora specjalne znaczenie. Jest to tzw. **nazwa zarezerwowana** lub **słowo zarezerwowane**. W języku Objective-C jest więcej takich słów i żadnego z nich nie można użyć do nazwania czegokolwiek.

Należy również pamiętać, że w języku Objective-C rozróżniane są wielkie i małe litery. W związku z tym nazwy `sum`, `Sum` i `SUM` dotyczą innych zmiennych. Przypomnę, że tradycyjnie nazwy klas zaczyna się wielką literą. Natomiast nazwy zmiennych egzemplarza, obiektów oraz metod zaczyna się małą. Aby tekst był bardziej czytelny, każde kolejne słowo w nazwie zaczyna się wielką literą, jak w poniższych przykładach:

- `AddressBook` — to może być nazwa klasy.
- `currentEntry` — to może być nazwa obiektu.
- `current_entry` — niektórzy wolą oddzielać wyrazy znakami podkreślenia.
- `addNewEntry` — to może być nazwa metody.

Dam Ci jedną radę dotyczącą wybierania nazw: nie spiesz się. Wybieraj takie nazwy, które odpowiadają przeznaczeniu nazywanych elementów. Dobre nazwy — tak jak komentarze — mogą znacznie zwiększyć czytelność kodu programu i na pewno je docenisz podczas usuwania błędów i pisania dokumentacji. Tak naprawdę pisanie dokumentacji będzie wówczas o wiele łatwiejsze, ponieważ wielu części kodu nie trzeba będzie nawet objaśniać.

Poniżej jeszcze raz przedstawiam kod sekcji `@interface` z listingu 3.2.

```
//--- Sekcja @interface ---

@interface Fraction: NSObject
{
    int numerator;
    int denominator;
}

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;

@end
```

Utworzona tu klasa nazywa się `Fraction`, a jej klasą macierzystą jest klasa o nazwie `NSObject` (temat klas macierzystych jest rozwinięty w rozdziale 8.). Definicja klasy `NSObject` znajduje się w pliku `NSObject.h`, który jest automatycznie dołączany do każdego programu, do którego zaimportowany jest nagłówek `Foundation.h`.

## Zmienne egzemplarza

W sekcji *dek1aracjeSkładowych* mieszczą się deklaracje typów danych, jakie będą przechowywane w klasie `Fraction`, oraz ich nazwy. Jak widać, sekcja ta jest zamknięta w nawiasie klamrowym. Z definicji klasy `Fraction` wynika, że jej obiekt będzie zawierał dwie składowe całkowitoliczbowe o nazwach `numerator` i `denominator`:

```
int numerator;  
int denominator;
```

Składowe, których definicje znajdują się w tej sekcji, nazywają się zmiennymi egzemplarza. Jak się niebawem przekonasz, wraz z każdym nowym obiektem tworzony jest nowy zestaw zmiennych egzemplarza. Jeśli zatem zostałyby utworzone dwa obiekty o nazwach `fracA` i `fracB`, to każdy z nich miałby własną parę tych zmiennych, tzn. zarówno obiekt `fracA`, jak i `fracB` miałyby własne liczniki (`numerator`) i mianowniki (`denominator`). System języka Objective-C pilnuje tego automatycznie, co jest jedną z wielkich zalet korzystania z obiektów.

## Metody klas i egzemplarzy

Aby móc coś zrobić z klasą, trzeba zdefiniować jej metody. Przede wszystkim trzeba mieć możliwość ustawiania wartości ułamka. Jako że nie będzie bezpośredniego dostępu do wewnętrznej reprezentacji licznika i mianownika (innymi słowy, nie będzie bezpośredniego dostępu do zmiennych egzemplarza), trzeba napisać metody, które umożliwią zmianę tych wartości. Dodatkowo napiszemy też metodę o nazwie `print` do drukowania wartości ułamków. W pliku interfejsu deklaracja tej metody jest następująca:

```
-(void) print;
```

Znajdujący się na pierwszym miejscu łącznik informuje kompilator Objective-C, że jest to metoda egzemplarza. Zamiast niego może też być znak `+` oznaczający metodę klasy. Metoda klasy to taka, która wykonuje jakieś czynności na samej klasie, a więc na przykład tworzy nowy egzemplarz tej klasy.

Metoda egzemplarza działa na konkretnym egzemplarzu klasy, na przykład ustawia lub pobiera jakąś jego wartość, wyświetla ją itd. Odwołując się do analogii z samochodem, po odebraniu samochodu z fabryki można napełnić jego bak paliwem. Czynność ta jest wykonywana na konkretnym egzemplarzu samochodu, a więc jest odpowiednikiem metody egzemplarza.

## Wartości zwrótne

Deklarując metodę, trzeba poinformować kompilator, czy będzie ona zwracała jakąś wartość, a jeśli tak, to jakiego typu. Definicję typu zwrótnego wpisuje się w nawiasie za łącznikiem lub znakiem plusa na początku definicji metody. Zatem metoda o nazwie `currentAge`, której deklarację widać poniżej, zwraca wartość całkowitoliczbową:

```
-(int) currentAge;
```



Natomiast poniższa metoda zwraca liczbę o podwójnej precyzji (więcej o tym typie danych dowiesz się w rozdziale 4.).

```
-(double) retrieveDoubleValue;
```

Do zwracania wartości przez metodę w języku Objective-C służy instrukcja `return`, a sposób jej użycia jest podobny do zwrotu wartości z funkcji `main`, z którym mieliśmy do czynienia wcześniej.

Jeśli metoda nie zwraca żadnej wartości, zaznacza się to słowem `void`:

```
-(void) print;
```

Powyższa instrukcja to deklaracja metody egzemplarza o nazwie `print`, która nie zwraca żadnej wartości. Na końcu takiej metody nie trzeba wpisywać instrukcji `return`, ale można ją podać bez żadnej wartości:

```
return;
```

## Argumenty metod

Oprócz omówionej metody w sekcji `@interface` na listingu 3.2 znajdują się deklaracje jeszcze dwóch innych metod:

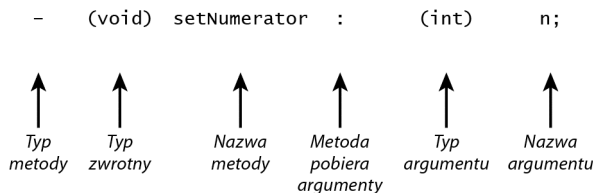
```
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
```

Żadna z nich nie zwraca wartości. Obie pobierają argument w postaci liczby całkowitej, o czym świadczy słowo `int` w nawiasie przed nazwą argumentu. Metoda `setNumerator` ma argument o nazwie `n`. Nazwę argumentu wybiera się dowolnie i jest ona używana przez metodę do odwoływania się do niego. Zatem z deklaracji metody `setNumerator` wynika, że przyjmuje ona jeden argument wywołania, o nazwie `n`, oraz że nie zwraca żadnej wartości. Podobnie jest z metodą `setDenominator`, z tym że jej argument nazywa się `d`.

Przyjrzyj się uważnie składni deklaracji tych metod. Po nazwie każdej z nich znajduje się dwukropek, który oznacza, że wymagają one argumentu. Dalej w nawiasie znajduje się deklaracja typu argumentu, co jest podobne do deklaracji typu zwrotnego. Na końcu jest symboliczna nazwa argumentu służąca do jego identyfikacji wewnątrz metody. Całość kończy średnik. Na rysunku 3.1 przedstawiono tę składnię w sposób schematyczny.

**Rysunek 3.1.**

Deklaracja metody



Jeśli metoda przyjmuje argument, należy do jej nazwy dołączać dwukropek także w odwołaniach do niej. Zatem poprawnie należałoby napisać `setNumerator:` i `setDenominator:`, aby odwołać się do tych pobierających po jednym argumentem metod. Natomiast odwołanie do metody `print` nie ma dwukropka, co oznacza, że nie przyjmuje ona żadnego argumentu. W rozdziale 7. dowiesz się, jak się tworzy metody pobierające więcej argumentów.

## Sekcja @implementation

W sekcji @implementation wpisuje się rzeczywisty kod implementacji metod zadeklarowanych w sekcji @interface. Jeśli chodzi o terminologię, mówi się, że w sekcji @interface metody się **deklaruje**, a w sekcji @implementation się je **definiuje**, tzn. pisze się ich rzeczywisty kod.

Ogólna składnia sekcji @implementation jest następująca:

```
@implementation NazwaNowejKlasy
    definicjeMetod;
@end
```

W miejsce parametru *NazwaNowejKlasy* należy wpisać tę samą nazwę co w sekcji @interface. Podobnie jak poprzednio, za nazwą można wpisać dwukropek i nazwę klasy macierzystej:

```
@implementation Fraction: NSObject
```

Nie jest to jednak obowiązkowe i zazwyczaj się tego nie robi.

W części *definicjeMetod* wpisuje się kod wszystkich metod zadeklarowanych w sekcji @interface. Podobnie jak poprzednio, definicja każdej metody zaczyna się od określnika jej typu (klasy lub egzemplarza), po którym znajduje się typ zwrotny oraz argumenty i ich typy. Jednak dalej zamiast średnika należy wpisać kod metody w nawiasie klamrowym.

Spójrz na sekcję @implementation z listingu 3.2:

```
//--- Sekcja @implementation ---
@implementation Fraction
-(void) print
{
    NSLog(@"%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

@end
```

Metoda print do wyświetlania wartości zmiennych egzemplarza numerator i denominator używa funkcji NSLog. Ale do którego licznika i mianownika ta metoda się odnosi? Do zmiennych, które są własnością obiektu będącego adresatem komunikatu. To bardzo ważne i wkrótce jeszcze do tego wrócimy.

Metoda `setNumerator`: zapisuje argument, któremu nadaliśmy nazwę `n`, w zmiennej egzemplarza `numerator`. Natomiast metoda `setDenominator`: zapisuje wartość argumentu `d` w zmiennej `denominator`.

## Sekcja programu

W sekcji programu znajduje się kod źródłowy rozwiązujący konkretny problem. W razie potrzeby może on być podzielony na wiele plików. Przypomnę, że gdzieś musi się znajdować procedura o nazwie `main`, ponieważ od niej zawsze zaczyna się wykonywanie programu. Przypomnijmy sobie jeszcze raz sekcję programu z listingu 3.2:

```
//--- Sekcja programu ---

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    Fraction *myFraction;

    // Tworzenie egzemplarza klasy Fraction

    myFraction = [Fraction alloc];
    myFraction = [myFraction init];

    // Ustawienie ułamka na 1/3

    [myFraction setNumerator: 1];
    [myFraction setDenominator: 3];

    // Wyświetlenie ułamka przy użyciu metody print

    NSLog(@"Wartość mojego ułamka wynosi:");
    [myFraction print];

    [myFraction release];
    [pool drain];

    return 0;
}
```

Wewnątrz funkcji `main` zdefiniowana jest zmienna o nazwie `myFraction`:

```
Fraction *myFraction;
```

Powyższy wiersz kodu oznacza, że `myFraction` jest obiektem typu `Fraction`. Innymi słowy, w `myFraction` będą przechowywane wartości z klasy `Fraction`. Rola gwiazdki znajdującej się przed nazwą zmiennej została objaśniona trochę dalej.

Mając obiekt do przechowywania ułamka, trzeba go utworzyć, tak jak idzie się do fabryki, aby złożyli nam samochód. Służy do tego poniższy wiersz kodu:

```
myFraction = [Fraction alloc];
```

Słowo `alloc` to skrót od angielskiego słowa `allocate`, czyli **alokować** albo **przydzielać**. Służy ono do rezerwowania w pamięci miejsca dla nowego ułamka. Poniższe wyrażenie wysyła komunikat do nowo utworzonej klasy `Fraction`:

```
[Fraction alloc]
```

Wywołujemy na rzecz klasy `Fraction` metodę `alloc`, której nigdzie nie zdefiniowaliśmy. Skąd więc się ona wzięła? Została odziedziczona po klasie macierzystej, ale szczegółowo dziedziczenie jest opisane dopiero w rozdziale 8.

Po wysłaniu komunikatu `alloc` do klasy w odpowiedzi otrzymuje się jej nowy egzemplarz. W programie przedstawionym na listingu 3.2 zwrócona wartość została zapisana w zmiennej o nazwie `myFraction`. Metoda `alloc` ustawia wartości wszystkich zmienionych egzemplarza na zero, przez co może on nie być od razu gotowy do użytku. Dlatego po alokacji obiekt trzeba zainicjować.

W programie z listingu 3.2 zrobiono to w następnym wierszu kodu:

```
myFraction = [myFraction init];
```

Tu znowu używana jest metoda, której nie zdefiniowaliśmy. Jest to metoda `init`, która służy do inicjacji egzemplarzy klas. Zwróć uwagę, że komunikat `init` został wysłany do `myFraction`, co znaczy, że zainicjowany ma zostać konkretny egzemplarz, a nie klasa. Postaraj się jak najlepiej zrozumieć tę różnicę, zanim przejdziesz do czytania dalszego tekstu.

Metoda `init` zwraca wartość, którą jest zainicjowany obiekt. Została ona zapisana w zmiennej typu `Fraction` o nazwie `myFraction`.

Te dwa wiersze kodu służące do alokacji nowego egzemplarza klasy i jego inicjacji są w języku Objective-C tak często używane, że najczęściej łączy się je w jeden:

```
myFraction = [[Fraction alloc] init];
```

Najpierw wykonywane jest wyrażenie wewnętrzne:

```
[Fraction alloc]
```

Jak już wiesz, jego wynikiem jest alokowany obiekt klasy `Fraction`. Zamiast zapisywać rezultat alokacji w zmiennej, jak to było robione wcześniej, od razu zastosowano do niego metodę `init`. Powtórzmy więc: najpierw alokujemy nowy ułamek, a potem go inicjujemy. Następnie wynik inicjacji przypisujemy do zmiennej `myFraction`.

Innym często stosowanym skrótem jest włączenie alokacji i inicjacji do wiersza deklaracji, jak poniżej:

```
Fraction *myFraction = [[Fraction alloc] init];
```

Tego typu instrukcje będą wielokrotnie używane w dalszej części książki, a więc koniecznie zapoznaj się z ich działaniem. We wszystkich przedstawionych dotychczas programach dokonywano alokacji puli automatycznej:

```
NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
```

Tym razem komunikat `alloc` został wysłany do klasy `NSAutoreleasePool` w celu utworzenia jej nowego egzemplarza. Następnie nowo utworzony obiekt zainicjowano, wysyłając do niego komunikat `init`.

Wracając do programu z listingu 3.2, możemy już ustawić wartość ułamka. Robią to poniższe wiersze kodu:

```
// Ustawienie ułamka na 1/3

[myFraction setNumerator: 1];
[myFraction setDenominator: 3];
```

Pierwsza instrukcja wysłała do obiektu `myFraction` komunikat `setNumerator:`. Jako argument przekazano wartość 1. Zaraz potem następuje przekazanie sterowania do metody `setNumerator:` zdefiniowanej dla klasy `Fraction`. System Objective-C „wie”, że jest to metoda z tej klasy, ponieważ „wie”, że `myFraction` jest jej obiektem.

W metodzie `setNumerator:` następuje zapisanie przekazanej jej wartości 1 w zmiennej `n`. Jedyny wiersz kodu składający się na implementację tej metody zapisuje tę wartość w zmiennej egzemplarza `numerator`. Mówiąc krótko, zmienna `numerator` została ustawiona na 1.

Następny jest komunikat wywołujący metodę `setDenominator:` na rzecz obiektu `myFraction`. Przekazany do niej argument 3 zostaje przypisany do wewnętrznej zmiennej `d`. Następnie wartość ta zostaje zapisana w zmiennej egzemplarza `denominator` i na tym kończy się proces przypisywania wartości 1/3 obiektowi `myFraction`. Teraz można wyświetlić wartość ułamka, do czego służą poniższe wiersze z listingu 3.2:

```
// Wyświetlenie ułamka przy użyciu metody print

NSLog(@"Wartość mojego ułamka wynosi:");
[myFraction print];
```

Funkcja `NSLog` wyświetla następujący tekst:

```
Wartość mojego ułamka wynosi:
```

Poniższe wyrażenie wywołuje metodę `print`:

```
[myFraction print];
```

Metoda `print` wyświetla wartości zmiennych `numerator` i `denominator`, oddzielając je ukośnikiem.

Poniższy komunikat zwalnia pamięć, która była używana przez nasz obiekt klasy `Fraction`:

```
[myFraction release];
```

Powyższy wiersz to bardzo ważny element dobrego stylu programowania. Tworząc obiekt, zawsze prosisz system o alokowanie dla niego obszaru w pamięci. Kiedy już go nie używasz, Twoim obowiązkiem jest zwolnienie tej pamięci. Oczywiście pamięć i tak zostanie zwolniona po zamknięciu programu, ale kiedy zaczniesz pisać bardziej rozbudowane aplikacje używające setek albo nawet tysięcy obiektów, to bardzo szybko może Ci zabraknąć pamięci, jeśli nie będziesz jej na bieżąco zwalniać. Czekanie ze zwalnianiem

pamięci na zamknięcie programu jest marnotrawstwem tego zasobu, może spowolnić działanie programu oraz stanowi przykład bardzo złego stylu programowania. Dlatego najlepiej od razu wyrób sobie nawyk zwalniania pamięci, kiedy tylko się da.

W systemie wykonawczym Apple wbudowany jest specjalny mechanizm nazywany **systemem usuwania nieużytków** (ang. *garbage collector*), który automatycznie zwalnia nieużywaną pamięć. Lepiej jednak umieć zarządzać pamięcią samodzielnie, niż całkowicie zdawać się na ten algorytm. Na niektórych platformach, takich jak iPhone i iPad, w ogóle nie jest on obsługiwany. Dlatego opis tego mechanizmu odłożymy na później.

Patrząc na kod źródłowy programów 3.1 i 3.2, można spostrzec, że ten drugi jest znacznie dłuższy, chociaż robi to samo co pierwszy. Mimo to podstawowym celem używania obiektów jest ułatwienie pisania programów, zarządzania nimi oraz ich rozszerzania. Stanie się to oczywiste nieco później.

Wróćmy jeszcze na chwilę do deklaracji obiektu `myFraction`:

```
Fraction *myFraction;
```

i ustawiania jego wartości.

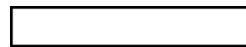


Zwróć uwagę, że Xcode w pierwszym wierszu funkcji `main`, w którym tworzony jest obiekt `NSAutoreleasePool`, automatycznie wstawia spację za znakiem `*`. Spacja ta nie jest potrzebna, a więc tak jak większość programistów nie będziemy jej wstawiać.

Gwiazdka znajdująca się przed nazwą `myFraction` w deklaracji oznacza, że `myFraction` w rzeczywistości jest referencją (**wskaznikiem**) do obiektu typu `Fraction`. Zmienna o tej nazwie tak naprawdę nie przechowuje danych ułamka (tzn. wartości jego zmiennych `numerator` i `denominator`), lecz referencję do nich — będącą adresem w pamięci — która wskazuje miejsce zapisania obiektu w pamięci. Bezpośrednio po zadeklarowaniu zmienna `myFraction` nie ma żadnej wartości, ponieważ nie została ustawiona ani nie ma wartości domyślnej. Zmienną `myFraction` można sobie wyobrazić jako pudełko, w którym znajduje się wartość. Początkowo w pudełku znajduje się niezdefiniowana wartość, ponieważ nic mu nie przypisano. Pokazano to na rysunku 3.2.

### Rysunek 3.2.

Deklaracja `Fraction *myFraction;`



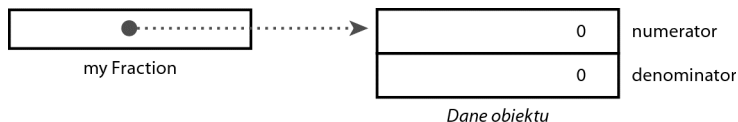
`myFraction`

Kiedy alokowany jest nowy obiekt (na przykład przy użyciu metody `alloc`), następuje zarezerwowanie ilości pamięci trochę większej, niż trzeba do przechowywania jego zmiennych. Następnie metoda `alloc` zwraca informację o miejscu zapisu tego obiektu (referencję do danych) i przypisuje ją do zmiennej `myFraction`. Wszystko to dzieje się w wyniku wykonania poniższej instrukcji z programu 3.2:

```
myFraction = [Fraction alloc];
```

Alokację obiektu i zapisanie referencji do niego w zmiennej `myFraction` pokazano na rysunku 3.3:

**Rysunek 3.3.**  
Relacje między  
zmienną `myFraction`  
i jej danymi

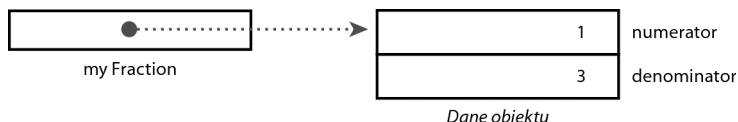


Na rysunku nie pokazano wszystkich danych, które są zapisywane wraz z obiektem, ale na razie nie musisz się tym przejmować. Zwróć uwagę, że zmienne egzemplarza zostały ustawione na 0, co jest dziełem metody `alloc`. Mimo to obiekt nie jest poprawnie zainicjowany i trzeba użyć metody `init`, aby to zmienić.

Zwróć uwagę na strzałkę na powyższym rysunku. Wskazuje ona kierunek połączenia, jakie zostało wykonane między zmienną `myFraction` a alokowanym obiektem (w zmiennej w rzeczywistości zapisany jest tylko adres w pamięci, pod którym znajdują się dane obiektu).

Później następuje ustawienie wartości zmiennych `numerator` i `denominator`. Na rysunku 3.4 pokazano w pełni zainicjowany obiekt klasy `Fraction`, którego zmienna `numerator` jest ustawiona na 1, a `denominator` na 3.

**Rysunek 3.4.**  
Ustawienie wartości  
zmiennych `numerator`  
i `denominator`



W następnym przykładzie zobaczysz, jak można użyć więcej niż jednego ułamka w jednym programie. Program przedstawiony na listingu 3.3 wartość jednego ułamka ustawia na  $2/3$ , a drugiego na  $3/7$ .

### Listing 3.3.

```
// Program do wykonywania działań na ułamkach — kontynuacja

#import <Foundation/Foundation.h>

//--- Sekcja @interface ---

@interface Fraction: NSObject
{
    int numerator;
    int denominator;
}

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;

@end

//--- Sekcja @implementation ---

@implementation Fraction
-(void) print
{
    NSLog(@"%i/%i", numerator, denominator);
}
}
```

```
-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}

@end

//--- Sekcja programu ---

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    Fraction *frac1 = [[Fraction alloc] init];
    Fraction *frac2 = [[Fraction alloc] init];

    // Ustawienie pierwszego ułamka na 2/3

    [frac1 setNumerator: 2];
    [frac1 setDenominator: 3];

    // Ustawienie drugiego ułamka na 3/7

    [frac2 setNumerator: 3];
    [frac2 setDenominator: 7];

    // Wyświetlenie ułamków

    NSLog(@"Pierwszy ułamek:");
    [frac1 print];

    NSLog(@"Drugi ułamek:");
    [frac2 print];

    [frac1 release];
    [frac2 release];

    [pool drain];
    return 0;
}
```

---

**Listing 3.3 — wynik**

---

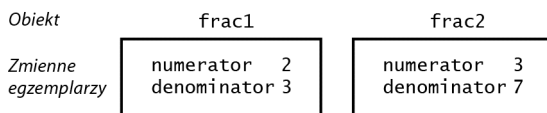
```
Pierwszy ułamek:
2/3
Drugi ułamek:
3/7
```

---



Sekcje `@interface` i `@implementation` pozostają takie same jak w programie 3.2. W treści programu natomiast zostały utworzone dwa obiekty o nazwach `frac1` i `frac2`, którym przypisano odpowiednio wartości  $2/3$  i  $3/7$ . Należy podkreślić, że wywołanie metody `setNumerator`: na rzecz obiektu `frac1` w celu ustawienia jego licznika na 2 powoduje ustawienie na tę wartość zmiennej egzemplarzowej `numerator` właśnie obiektu `frac1`. To samo dotyczy obiektu `frac2`, którego licznik został ustawiony na 3 — na wartość tę została ustawiona jego prywatna zmienna egzemplarzowa o nazwie `numerator`. Każdy nowy obiekt ma własny zestaw zmiennych, co przedstawiono schematycznie na rysunku 3.5.

**Rysunek 3.5.**  
Zmienne  
egzemplarzy



W zależności od tego, który obiekt odbierze komunikat, następuje odwołanie do odpowiednich zmiennych egzemplarzowych. W poniższej instrukcji zostanie ustawiona zmienna `numerator` obiektu `frac1`:

```
[frac1 setNumerator: 2];
```

Dzieje się tak dlatego, że adresatem komunikatu jest obiekt `frac1`.

## Zasady dostępu do zmiennych egzemplarzy i hermetyzacja danych

W poprzednich przykładach pokazano, jak metody dotyczące ułamków ustawiają wartości zmiennych `numerator` i `denominator`, korzystając bezpośrednio z ich nazw. W istocie metoda egzemplarza zawsze ma bezpośredni dostęp do zmiennych egzemplarza. Natomiast przywileju tego nie mają metody klas, ponieważ dotyczą one samej klasy, a nie jej egzemplarzy (zastanów się nad tym przez chwilę). Co jednak zrobić, gdyby trzeba było uzyskać dostęp do zmiennych egzemplarza z innego miejsca, na przykład procedury `main`? Bezpośrednio się nie da, ponieważ zmienne te są ukryte. Ukrywanie zmiennych w taki sposób jest bardzo ważną techniką zwaną **hermetyzacją danych** (ang. *data encapsulation*). Dzięki temu programista piszący klasy rozszerzające i modyfikujące istniejące klasy nie musi się obawiać, że ich użytkownicy będą grzebać w ich wnętrzu. Hermetyzacja danych zapewnia więc izolację programisty od twórcy klasy.

Nieszkodliwy dostęp do zmiennych egzemplarzy można zapewnić poprzez napisanie specjalnych metod umożliwiających odczytywanie i ustawianie ich wartości. Na przykład do ustawiania wartości zmiennych `numerator` i `denominator` w klasie `Fraction` napisaliśmy metody `setNumerator` i `setDenominator`. Aby móc sprawdzać te wartości, trzeba napisać dwie kolejne metody. Napiszemy na przykład dwie metody o nazwach `numerator` i `denominator`, które będą służyły do sprawdzania wartości odpowiednio zmiennych `numerator` i `denominator` obiektu będącego odbiorcą komunikatu. Ich wartością zwrótną będzie liczba całkowita zapisana w tych zmiennych. Poniżej znajduje się kod deklaracji tych dwóch metod:

```
-(int) numerator;  
-(int) denominator;
```

A to są ich definicje:

```
-(int) numerator
{
    return numerator;
}

-(int) denominator
{
    return denominator;
}
```

Zwróć uwagę, że nazwy tych metod są takie same jak nazwy zmiennych egzemplarza, których dotyczą. Nie ma w tym nic złego (choć dla początkującego może to dziwnie wyglądać), a wręcz taki jest zwyczaj. Program przedstawiony na listingu 3.4 pozwala przetestować obie nowe metody.

### Listing 3.4.

```
// Program do wykonywania działań na ułamkach — kontynuacja

#import <Foundation/Foundation.h>

//--- Sekcja @interface ---

@interface Fraction: NSObject
{
    int numerator;
    int denominator;
}

-(void) print;
-(void) setNumerator: (int) n;
-(void) setDenominator: (int) d;
-(int) numerator;
-(int) denominator;

@end

//--- Sekcja @implementation ---

@implementation Fraction
-(void) print
{
    NSLog(@"%i/%i", numerator, denominator);
}

-(void) setNumerator: (int) n
{
    numerator = n;
}

-(void) setDenominator: (int) d
{
    denominator = d;
}
}
```

```
- (int) numerator
{
    return numerator;
}

- (int) denominator
{
    return denominator;
}

@end

//--- Sekcja programu ---

int main (int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Fraction *myFraction = [[Fraction alloc] init];

    // Ustawia ułamek na 1/3

    [myFraction setNumerator: 1];
    [myFraction setDenominator: 3];

    // Wyświetla ułamek przy użyciu dwóch nowych metod

    NSLog (@"Wartość mojego ułamka wynosi: %i/%i",
           [myFraction numerator], [myFraction denominator]);
    [myFraction release];
    [pool drain];

    return 0;
}
```

---

**Listing 3.4** — *wynik*

---

Wartość mojego ułamka wynosi: 1/3

---

Instrukcja wywołująca funkcję `NSLog` wyświetla wynik wysłania dwóch komunikatów do obiektu `myFraction`: pierwszy został wysłany w celu pobrania wartości zmiennej `numerator`, a drugi — `denominator`:

```
NSLog (@"Wartość mojego ułamka wynosi: %i/%i",
      [myFraction numerator], [myFraction denominator]);
```

W pierwszym wywołaniu nastąpiło wysłanie komunikatu `numerator` do obiektu klasy `Fraction` o nazwie `myFraction`. Kod tej metody zwróci wartość zmiennej egzemplarza `numerator` tego ułamka. Przypomnę, że kontekstem wykonywania metody jest obiekt będący adresatem komunikatu. Kiedy zatem metoda `numerator` pobiera i zwraca wartość zmiennej `numerator`, jest to wartość zmiennej o tej nazwie obiektu `myFraction`. Następnie zwrócona liczba całkowita zostaje przekazana do funkcji `NSLog` w celu wyświetlenia.

W drugim wywołaniu metoda `denominator` pobiera i zwraca wartość mianownika obiektu `myFraction`, która następnie również zostaje przekazana do `NSLog` w celu wyświetlenia.

Na metody służące do ustawiania wartości zmiennych egzemplarzowych potocznie mówi się **settery** (ang. *setter* — metoda ustawiająca), natomiast na metody pobierające wartości zmiennych egzemplarzowych — **gettery** (ang. *getter* — metoda pobierająca). W klasie `Fraction` metodami ustawiającymi są `setNumerator:` i `setDenominator:`, a pobierającymi — `numerator` i `denominator`. Zbioreza nazwa, jaką określa się zarówno metody ustawiające, jak i pobierające, to **metody dostępne** (ang. *accessor methods*).



Uwaga

Wkrótce poznasz wygodną funkcję języka Objective-C 2.0 umożliwiającą automatyczne tworzenie metod dostępowych.

Umiejętność rozróżniania metod pobierających i ustawiających jest bardzo ważna. Metody ustawiające nie zwracają żadnej wartości, ponieważ ich zadaniem jest pobieranie argumentów i ustawianie na ich wartości odpowiednich zmiennych egzemplarza. W takim przypadku nie ma potrzeby zwracania czegokolwiek. Natomiast zadaniem metod pobierających jest „pobieranie” wartości zmiennych egzemplarza zapisanych w obiektach i przekazywanie ich do programu. Dlatego metody te muszą zawierać instrukcję `return` służącą do zwracania pobranych wartości zmiennych.

Przypomnę jeszcze, że brak możliwości bezpośredniego ustawiania i sprawdzania wartości zmiennych egzemplarza w inny sposób niż przy użyciu specjalnie do tego przeznaczonych metod dostępowych nazywa się hermetyzacją danych. Aby uzyskać dostęp do danych, które inaczej są ukryte przed „światem zewnętrznym”, trzeba użyć specjalnych metod. W ten sposób tworzy się scentralizowany punkt dostępu do danych i uniemożliwia zmianę ich w żaden inny sposób, co sprawia, że łatwiej jest zrozumieć kod programów, a także go modyfikować i oczyszczać z błędów.

Należy jeszcze dodać, że istnieje także metoda `new`, która łączy w sobie funkcjonalność metod `alloc` i `init`. Można na przykład alokować i zainicjować obiekt klasy `Fraction` za pomocą następującego wiersza kodu:

```
Fraction *myFraction = [Fraction new];
```

Ogólnie rzecz biorąc, lepiej jest jednak korzystać z dwuetapowego procesu alokacji i inicjacji, ponieważ dokładnie wiadomo, że wykonywane są dwie czynności: utworzenie obiektu i jego inicjacja.

## Podsumowanie

Potrafisz już zdefiniować własną klasę, tworzyć jej obiekty, czyli egzemplarze, oraz wysyłać do nich komunikaty. Do klasy `Fraction` wrócimy jeszcze w dalszej części książki. Dowiesz się, jak przekazywać do metod po kilka argumentów, jak dzielić definicje klas na kilka plików oraz jak zrobić użytek z takich podstawowych technik, jak dziedziczenie i wiązanie dynamiczne. Na razie jednak musisz zdobyć nieco więcej wiadomości o typach danych i pisaniu wyrażeń w języku Objective-C. Zanim przejdziesz do kolejnego rozdziału, sprawdź swoją wiedzę, wykonując poniższy zestaw ćwiczeń.

## Ćwiczenia

1. Które z poniższych nazw są niepoprawne? Uzasadnij.

Int	playNextSong	6_05
_calloc	Xx	a1phaBetaRoutine
clearScreen	_1312	z
ReInitialize	_	A\$

2. Przypomnij sobie przykład z samochodem, przytoczony w tym rozdziale, i pomyśl o jakimś obiekcie, którego używasz na co dzień. Wymyśl dla niego klasę i napisz pięć czynności, które można z nim wykonać.
3. Przepisz listę czynności z punktu 2. przy użyciu poniższej składni:

```
[egzemplarz metoda];
```

4. Wyobraź sobie, że oprócz samochodu masz łódź i motocykl. Sporządź listę czynności, jakie można wykonać z każdym z tych sprzętów. Czy niektóre z nich się powtarzają?
5. Odnosząc się do punktu 4., wyobraź sobie, że masz klasę o nazwie `Pojazd` i obiekt o nazwie `mojPojazd`, który może być typu `Samochod`, `Motocykl` lub `Lodz`. Przypuśćmy, że piszesz poniższy kod:

```
[mojPojazd przygotuj];  
[mojPojazd zatankuj];  
[mojPojazd napraw];
```

Czy widzisz jakieś korzyści z tego, że możesz zastosować tę samą czynność do obiektu, który może należeć do jednej z kilku klas?

6. Korzystając z takiego proceduralnego języka programowania jak C, najpierw wymyśla się potencjalne czynności, a potem pisze kod wykonujący je na różnych obiektach. Można by było na przykład napisać procedurę mycia pojazdu, a następnie w jej wnętrzu napisać kod do mycia samochodu, łodzi, motocykla itd. Przypuśćmy, że po jakimś czasie musisz dodać nowy typ pojazdu (zobacz poprzednie ćwiczenie). Czy dostrzegasz jakieś wady lub zalety zastosowania takiego proceduralnego podejścia w stosunku do obiektowego?
7. Zdefiniuj klasę o nazwie `PunktXY` do przechowywania współrzędnych kartezjańskiego układu współrzędnych, gdzie  $x$  i  $y$  są liczbami całkowitymi. Zdefiniuj metody ustawiające i pobierające wartość każdej ze współrzędnych osobno. Następnie napisz program w języku Objective-C implementujący tę klasę i ją przetestuj.

# Skorowidz

#define, instrukcja, 233, 234, 235, 236  
#elif, instrukcja, 241, 243  
#else, instrukcja, 241  
#endif, instrukcja, 241  
#if, instrukcja, 243  
#ifdef, instrukcja, 241  
#ifndef, instrukcja, 241, 242  
#import, instrukcja, 32, 239, 240  
#undef, instrukcja, 244  
@catch, 194, 195  
@class, dyrektywa, 162  
@end, 225  
@finally, 194  
@implementation, sekcja, 44, 49  
@interface, sekcja, 44, 45  
@optional, dyrektywa, 226, 229  
@package, dyrektywa, 200  
@private, dyrektywa, 200, 455  
@property, dyrektywa, 134  
@protected, dyrektywa, 200  
@protocol, dyrektywa, 225  
@public, dyrektywa, 200, 201  
@required, dyrektywa, 226  
@selector, dyrektywa, 190  
@throw, 195  
@try, instrukcja, 193, 194  
\_\_block, modyfikator, 259  
<NSCopying>, protokół  
  implementacja, 420, 421  
<NSMutableCopying>, protokół, 416

## A

abstrakcyjne, klasy, 177  
accessor methods, *Patrz* metody dostępne  
addObject, metoda, 331, 360  
addSubview, metoda, 399  
adresowania, operator, 272, 273  
adresowanie pośrednie, 272  
allKeys, metoda, 357  
alloc, metoda, 51, 53, 204, 411

alokacja, 51, 161  
  metody, 411  
alokowanie pamięci, 51  
aplikacja rodzima, 14  
AppDelegate, klasa, 464  
appendString, metoda, 324  
AppKit, 446  
application bundle, *Patrz* pakiety programów  
Application Kit, 305  
Application Services, *Patrz* usługi programów  
applicationDidFinishLaunchingWithOptions,  
  metoda, 456  
archiveRootObjectToFile, metoda, 429  
archiwizacja, 427  
  przy użyciu klasy NSObjectArchiver, 429  
  z kluczami, 429  
argc, 296  
arguments, metoda, 384  
argumenty metod, 142  
argv, 296  
array, metoda, 331  
arrayWithContentsOfFile, metoda, 428  
arrayWithContentsOfURL, metoda, 391  
arrayWithObjects, metoda, 329  
arytmetyczne, wyrażenia, 65  
atomic, słowo kluczowe, 424  
attributesOfItemAtPath:error, metoda, 372  
attributesOfItemAtPath:traverseLink, metoda, 369  
automatyczne zmienne lokalne, 254  
autorelease pool, *Patrz* pula pamięci zwalnianej  
  automatycznie  
autorelease, komunikat, 409  
autorelease, metoda, 405

## B

bezwzględne, ścieżki, 368  
biblioteka, 15, 305  
bitowe, operatory, 211, 212, 213, 214, 215, 216, 217  
bloki, 259, 260, 262  
błędy krytyczne, 27

BOOL, typ danych, 125  
 break, instrukcja, 96  
 bufor, 372  
   niezmienny, 372  
   zmienny, 372

## C

C, język, 13, 14  
 Caches, katalog, 382  
 caseInsensitiveCompare, metoda, 318  
 categories, *Patrz* kategorie  
 CGFloat, 267  
 CGGeometry.h, 267  
 CGPoint, 267  
 CGRect, 267  
 CGRectMake, 268  
 CGSize, 267  
 CGSizeMake, 268  
 char, typ danych, 61, 62  
 class, metoda, 189  
 Cocoa, 13, 305, 409, 445, 446, 447  
 Cocoa Touch, 305, 445, 446, 447  
 compare, metoda, 313, 318  
 conformsTo, metoda, 226  
 containsObject, metoda, 347  
 contentsOfDirectoryAtPath:error, metoda, 376  
 continue, instrukcja, 97  
 copy, atrybut, 337  
 copy, metoda, 416, 419  
 copyItemAtPath:error, metoda, 371  
 copyWithZone, metoda, 421, 422  
 Core Services, *Patrz* usługi systemu  
 countForObject, metoda, 360  
 Cox, Brad J., 13  
 currentDirectoryPath, metoda, 380

## D

dane, hermetyzacja, 56  
 data encapsulation, *Patrz* hermetyzacja danych  
 dataWithContentsOfFile, metoda, 428  
 dealloc, komunikat, 397, 409  
 dealloc, metoda, 173  
 dealokacja, 397  
 decodeObject:forKey, metoda, 432  
 decyzje, podejmowanie, 99  
 defaultManager, komunikat, 369  
 dekrementacji, operator, 85, 280, 285, 286  
 delegacja, 227  
 deleteCharactersInRange, metoda, 324, 325  
 description, metoda, 315  
 dictionaryWithContentsOfURL, metoda, 391  
 dispatch table, *Patrz* tablice rozdziału

do, instrukcja, 95  
 dostępne metody, 59, 134  
 double, typ danych, 61  
 dynamiczne typowanie, 181, 188, 189  
 działania  
   kolejność, 65, 67  
   na wskaźnikach, 288  
 dziedziczenie, 153, 157, 175  
   zmiennych egzemplarzowych i metod, 155  
 dzielenia modulo, operator, 70

## E

egzemplarz, 39  
 egzemplarzowe, zmienne, 401  
 else if, konstrukcja, 110  
 encodeObject:forKey, metoda, 432, 433, 436  
 encodeWithCoder, metoda, 431  
 enum, słowo kluczowe, 206  
 enumeratorAtPath, metoda, 376  
 etykiety, 293, 294  
 extern, słowo kluczowe, 201, 202, 203

## F

fast enumeration, *Patrz* szybka enumeracja  
 fatal error, *Patrz* błędy krytyczne  
 fileHandleForUpdatingAtPath, metoda, 387  
 fileHandleForWritingAtPath, metoda, 387  
 finishDecoding, komunikat, 439  
 finishEncoding, komunikat, 439  
 float, typ danych, 61, 62  
 for, 80, 81, 83, 84  
   warianty, 90  
   zagnieżdżanie, 88  
 forwardInvocation, metoda, 191  
 Foundation, biblioteka, 305, 309  
   dokumentacja, 305  
   tablice, 328  
   zwalnianie obiektów, 311  
 Foundation.h, 309  
 framework, *Patrz* biblioteka  
 funkcja znaku, 110  
 funkcje, 251, 252, 257, 258  
   argumenty, 253, 256  
   prototyp, 256, 257  
   przekazywanie tablic, 258  
   wartość zwrotna, 255, 256  
   wskaźniki, 277, 288  
   zewnętrzne, 258  
   zmienne lokalne, 253, 254

**G**

garbage collector, *Patrz* system usuwania nieużytków  
 gcc, kompilator, 29  
 gettery, 59  
 globalne zmienne, 201  
 goto, instrukcja, 293

**H**

hermetyzacja danych, 56

**I**

I, operator, 213  
 IBAction, 456  
 IBOutlet, identyfikator, 455  
 id, typ danych, 64, 184, 187, 299  
 if, instrukcja, 99  
   zagnieżdżanie, 108  
 if-else, instrukcja, 103, 105  
 implementacyjny, plik, 133  
 indeks, 246  
 indexOfObject, metoda, 347  
 indexOfObjectPassingTest, metoda, 362, 363  
 indirection, *Patrz* adresowanie pośrednie  
 inicjatory, 198  
   desygnowane, 198  
 init, metoda, 51  
 initWithWritingWithMutableData, komunikat, 439  
 initWith, przedrostek, 312  
 initWithCoder, metoda, 431, 432, 433  
 inkrementacji, operator, 85, 280, 285, 286  
 insertString:atIndex, metoda, 324  
 instrukcje, 32, 33  
 int, typ danych, 61  
 interfejsowy, plik, 132, 133  
 intersection, metoda, 360  
 iOS  
   często używane katalogi, 382  
   pakiet SDK, 449  
   pisanie programów, 449  
 iPhone  
   projektowanie interfejsu, 457  
   symulator, 451  
   szablony programów, 452  
   tworzenie nowego projektu, 451  
 isa, 298  
 isEqualToNumber, metoda, 313  
 isEqualToSet, metoda, 360  
 isEqualToString, metoda, 318  
 isKindOfClass, metoda, 192  
 isMemberOfClass, metoda, 192

**J**

jądro, 445

**K**

katalog główny, 368  
 katalogi  
   praca, 374  
   sprawdzanie zawartości, 376  
   systemu iOS, 382  
 kategorie, 160, 219, 223, 224  
 klasa, 39, 153, 154  
   abstrakcyjna, 177  
   do działań na ułamkach, 42  
   dziedziczenie, 157  
   główna, 153  
   konkretna, 177  
   macierzysta, 45  
   metody, 45, 47  
   nadrzędna, 153  
   nazwy, 45, 46  
   potomna, 153  
   rozszerzanie definicji, 151  
   rozszerzanie przez dziedziczenie, 157  
   zapis w osobnych plikach, 129  
   zmiennie egzemplarza, 45  
 klastr, 177  
 klawiatura, odbieranie danych, 86  
 komentarze, 26, 31  
 kompilacja, 21  
   błędy krytyczne, 27  
   ostrzeżenia, 27  
   przy użyciu Xcode, 22, 26, 28  
   terminal, 28  
   warunkowa, 241, 242  
 komunikat, 40  
 konwersja typów, 71, 210  
   reguły, 210  
 kopiowanie  
   głębokie, 418, 419  
   obiektów, 415  
   plików, 382  
   płytkie, 418, 419  
   przy użyciu archiwizatora, 441  
   w metodach ustawiających i sprawdzających, 423  
 kropki, operator, 135  
 kwalifikatory, 63  
   long, 63  
   long long, 63  
   short, 63, 64  
   signed, 63  
   unsigned, 63



**L**

lastPathComponent, metoda, 380  
 length, metoda, 318  
 liczba trójkątna, 79  
 liczenie referencji, 396  
   łańcuchy, 399  
 listy właściwości, 427  
 literały złożone, 293  
 logiczne, zmienne, 121, 123, 125  
 lokalizacja, 314  
 lokalne zmienne, 141, 142, 253, 254  
   automatyczne, 254  
   statyczne, 254  
 long, 63  
 long long, 63  
 lowercaseString, metoda, 318  
 LUB wykluczającego, operator, 214  
 LUB, operator, 214

**Ł**

łańcuchy formatujące, 36  
 łańcuchy znaków, 249, 284  
   wskaźniki, 282, 284  
 łączność, 65

**M**

M\_PI, 234  
 main, 252  
 mainBundle, metoda, 392  
 makra, 238  
 metody, 40, 45, 258  
   argumenty, 48, 142  
   bez nazw argumentów, 138  
   dekodujące, 431  
   dostępowe, 59, 134  
   egzemplarzy, 47  
   klas, 47  
   kodujące, 431  
   pobierające, 59, 134  
   przekazywanie tablic, 258  
   przesłanie, 169, 173  
   przyjmujące po kilka argumentów, 136  
   ustawiające, 59, 134  
   wartości zwrotne, 47  
   wskaźniki, 277  
   wywoływanie, 40, 41  
   zwracanie obiektów, 146  
 minus, operator, 68  
 moduł, 200  
 moveItemAtPath:toPath, metoda, 372  
 moveItemAtPath:toPath:error, metoda, 371

mutableCopy, metoda, 416, 419  
 mutableCopyWithZone, metoda, 421  
 muteks, 424  
 mutex, *Patrz* muteks

**N**

nadklasa, 153, 154  
   abstrakcyjna, 177  
 nagłówki prekompilowane, 309  
 nagłówkowe pliki, 240  
 NAN, 101  
 nazwy  
   niepoprawne, 46  
   poprawne, 45  
   tworzenie, 45, 46  
   zarezerwowane, 46  
 nieużytki, usuwanie, 411  
 nonatomic, atrybut, 337  
 nonatomic, słowo kluczowe, 424  
 notacja  
   naukowa, 62  
   sigma, 149  
 NSArchiver, klasa, 429  
 NSArray, klasa, 198, 328  
   metody, 352, 353  
 NSBundle, klasa, 367, 391  
 NSCalendarDate, klasa, 271  
 NSCopying, protokół, 225  
 NSCountedSet, klasa, 360  
 NSData, klasa, 372  
   tworzenie archiwów, 438  
 NSDate, klasa, 271  
 NSDictionary, klasa, 357  
   metody, 357  
 NSFileHandle, klasa, 367, 386, 387  
   metody, 386  
 NSFileManager, klasa, 367, 368  
   metody, 369, 374  
 NSFileSize, 370  
 NSHomeDirectory, funkcja, 380  
 NSHomeDirectoryForUser, funkcja, 380  
 NSIndexSet, klasa, 362  
   metody, 364  
 NSKeyedArchiver, klasa, 429  
 NSKeyedUnarchiver, klasa, 433, 439  
 NSLog, funkcja, 32, 34, 35, 65, 315, 331  
 NSMutableArray, klasa, 328  
   metody, 353  
 NSMutableData, klasa, 372  
 NSMutableDictionary, klasa, 357  
   metody, 358  
 NSMutableSet, klasa, 361  
   metody, 361

NSMutableString, klasa, 316, 322  
   metody, 328  
 NSNumber, klasa, 177, 313  
   metody, 312  
 NSObject, klasa, 153  
   metody, 189  
 NSObject.h, 313  
 NSOrderAscending, 313, 318  
 NSOrderDescending, 313, 318  
 NSOrderedSame, 313, 318  
 NSPathUtilities.h, 378  
 NSProcessInfo, klasa, 382, 384  
   metody, 385  
 NSPropertyListSerialization, klasa, 429  
 NSRange, typ danych, 320  
 NSSearchPathForDirectoriesInDomains, funkcja,  
   381, 382  
 NSSet, klasa, 361  
   metody, 361  
 NSString, klasa, 32, 62, 314, 316, 326  
   metody, 326, 327  
 NSTemporaryDirectory, funkcja, 379  
 NSURL, klasa, 367, 390  
 NSValue, klasa, 353, 354  
   metody, 354  
 null statement, *Patrz* pusta instrukcja  
 numberWithInt, metoda, 313, 314  
 numberWithInteger, metoda, 313, 314  
 NWD, algorytm, 92

## O

obiekty, 39, 40  
   identyczne, 346  
   inicjowanie, 197  
   kopiowanie, 415, 423, 441  
   liczbowe, 309  
   łańcuchowe, 314  
   niezmiennicze, 316, 319  
   stan, 40  
   tablicowe, 328  
   złożone, 229  
   zmiennicze, 316  
 objectAtIndex, metoda, 329  
 objectForKey, metoda, 356  
 Objective-C, język, 13  
 obsługa wyjątków, 193  
 odpakowywanie, 354  
 opakowywanie, 354  
 operacje bitowe, 212  
 operatory  
   adresowania, 272, 273  
   bitowe, 211, 212, 213, 214, 215, 216, 217  
   dekrementacji, 85, 280, 285, 286  
   dwuargumentowe, 65

  dzielenia modulo, 70  
   I, 213  
   inkrementacji, 85, 280, 285, 286  
   jednoargumentowe, 69  
   kropki, 135  
   LUB, 214  
   LUB wykluczającego, 214  
   łączność, 65  
   minus, 68  
   porównania, 106  
   priorytet, 65  
   przecinek, 294  
   przesunięcia w lewo, 216, 217  
   przesunięcia w prawo, 217  
   przypisania, 73, 74  
   relacyjne, 82  
   rzutowania typów, 72, 73  
   sizeof, 295, 296  
   trójargumentowe, 126  
   uzupełnienia jedynkowego, 215  
   warunkowe, 126, 127  
   wskaźnikowy struktur, 276  
 ostrzeżenia, 27  
 outlet, 454  
 override, *Patrz* przesłanianie

## P

pakiety programów, 367, 391  
 pamięć  
   adresy, 290  
   przydzielanie, 51  
   wycieki, 148, 411  
   zarządzanie, 395, 409  
   zwalnianie, 52, 53  
 pathComponents, metoda, 380  
 pathExtension, metoda, 380  
 pathForResource ofType, metoda, 392  
 pathsOfResourcesOfType:inDirectory, metoda, 392  
 performSelector, metoda, 190  
 pętle, 79  
   do, 95  
   for, 80, 81, 83, 84, 88, 90  
   while, 91, 92  
 piaskownica, 375  
 pliki  
   dołączane, 240  
   implementacyjne, 133  
   importowanie, 133  
   interfejsowe, 132, 133  
   kopiowanie, 382  
   nagłówkowe, 240  
   podstawowe operacje, 386  
   praca, 367, 368  
   rozszerzenia, 25

pobierające metody, 59, 134  
 podklasa, 153, 154  
 pola bitowe, 269, 270, 271  
 polimorfizm, 181, 184  
 porównania, operator, 106  
 postdekrementacja, 285  
 postinkrementacja, 285, 286  
 predekrementacja, 285  
 preinkrementacja, 285, 286  
 prekompilowane nagłówki, 309  
 preprocesor, 233, 237  
 programowanie  
   obiektywne, 39  
   proceduralne, 39  
 programu, sekcja, 50  
 protokół, 224, 225, 226, 227  
   abstrakcyjny, 228  
   bezklasowy, 226  
   definiowanie, 225  
   formalny, 228  
   nieformalny, 228  
 przecinek, operator, 294  
 przesłanianie, 169, 173  
 przestrzenie nazw, 224  
 przesunięcia w lewo, operator, 216, 217  
 przesunięcia w prawo, operator, 217  
 przydzielanie pamięci, 51  
 przypisania, operatory, 73, 74  
 pula automatyczna, 311, 334, 395, 396, 405, 409  
 pula pamięci zwalnianej automatycznie, 32  
 pusta instrukcja, 294

## Q

qsort, funkcja, 289

## R

rangeOfString, metoda, 321, 325  
 readDataOfLength, metoda, 388  
 readDataToEndOfFile, metoda, 388  
 reference counting, *Patrz* liczenie referencji  
 referencje, 53  
   liczenie, 396  
 relacyjne, operatory, 82  
 release, komunikat, 397, 409  
 release, metoda, 173, 409  
 removeItemAtPath, metoda, 369  
 removeObject, metoda, 360  
 replaceCharacterInRange:withString, metoda, 325  
 respondsToSelector, metoda, 191, 228  
 retain, metoda, 396, 399, 409  
 return, instrukcja, 48, 255  
 Ritchie, Dennis, 13  
 rodzima aplikacja, 14

root class, *Patrz* klasa główna  
 root directory, *Patrz* katalog główny  
 rzutowanie, 72

## S

sandbox, *Patrz* piaskownica  
 scanf, 86, 87, 113  
 seekToEndOfFile, metoda, 390  
 sekcje  
   @implementation, 44, 49  
   @interface, 44, 45  
   programu, 50  
 self, słowo kluczowe, 145, 146  
 setObject:forKey, metoda, 356  
 setString, metoda, 325  
 settery, 59  
 setWithObjects, metoda, 360  
 short, 63, 64  
 sigma, notacja, 149  
 signed, 63  
 sizeof, operator, 295, 296  
 skipDescendants, komunikat, 376  
 słownik atrybutów, 369, 370  
 słowniki, 355  
   enumeracja, 356  
   niezmiennicze, 355  
   zmiennicze, 355  
 słowo zarezerwowane, 46  
 sortowanie szybkie, 289  
 sortUsingSelector, metoda, 349  
 stała, 61  
 static, słowo kluczowe, 142, 254  
 statyczne typowanie, 187  
 statyczne zmiennicze, 203  
 statyczne zmiennicze lokalne, 254  
 sterowniki urządzeń, 445  
 stringByAppendingPathComponent, metoda, 380  
 stringByAppendingString, metoda, 318  
 stringByExpandingTildeInPath, metoda, 380  
 stringByStandardizingPath, metoda, 380  
 stringWithContentsOfFile, metoda, 428  
 stringWithFormat, metoda, 316  
 struct, słowo kluczowe, 291  
 struktury, 263, 265, 268  
   inicjowanie, 266  
   wewnątrz struktur, 267  
   wskaźniki, 275, 276  
 substringFromIndex, metoda, 321  
 substringToIndex, metoda, 321  
 super, słowo kluczowe, 173, 174  
 switch, instrukcja, 118, 119, 121  
 symulator iPhone'a, 451  
 system usuwania nieużytków, 53, 412  
 szybka enumeracja, 330, 341

## Ś

ścieżki, 368  
 bezwzględne, 368  
 najczęściej używane funkcje, 381  
 najczęściej używane metody, 380, 381  
 praca, 378  
 względne, 368  
 środowisko z pamięcią zarządzaną, 411

## T

tablice, 245, 246, 247, 258, 328  
 dwuwymiarowe, 250, 251  
 inicjowanie elementów, 248  
 niezmiennicze, 328  
 rozdziału, 289  
 wielowymiarowe, 250  
 wskaźniki, 278, 279  
 zmienne, 328  
 znaków, 249  
 terminal, kompilacja, 28  
 traverseLink, parametr, 369  
 trójkątna liczba, 79  
 typedef, instrukcja, 197, 209  
 typowanie  
 dynamiczne, 181, 188, 189  
 statyczne, 187  
 typy danych, 61, 65  
 BOOL, 125  
 char, 61, 62  
 double, 61  
 float, 61, 62  
 id, 64, 184, 187, 299  
 int, 61  
 konwersja, 71, 210  
 kwalifikatory, 63  
 metody do kodowania i dekodowania, 432  
 NSRange, 320  
 rzutowanie, 72  
 sprawdzanie, 186  
 wyliczeniowy, 197  
 zakres, 62

## U

UIKit, 447  
 UILabel, klasa, 455  
 UITableView, klasa, 227, 228  
 UITableViewDataSource, protokół, 227  
 UITableViewDelegate, protokół, 228  
 UIViewController, klasa, 465  
 UIWindow, klasa, 455  
 unichar, 314

unie, 290, 291  
 union, metoda, 360  
 union, słowo kluczowe, 291  
 unsigned, 63  
 unwrapping, *Patrz* odpakowywanie  
 uppercaseString, metoda, 318  
 URLWithString, metoda, 390  
 usługi programów, 446  
 usługi systemu, 445  
 ustawiające metody, 59, 134  
 uzupełnienia jedynekowego, operator, 215

## V

void, 48, 252

## W

wartości zmiennoprzecinkowe, 35  
 wartość stała, 61  
 wartość zwrotna, 47, 255  
 typ, 256  
 warunkowa kompilacja, 241, 242  
 warunkowe wyrażenie, 126  
 warunkowy, operator, 126, 127  
 wejście standardowe, 294  
 while, instrukcja, 91, 92  
 wiązanie dynamiczne, 176, 181, 184  
 widoki, 399  
 wiersz poleceń, argumenty, 296  
 wrapping, *Patrz* opakowywanie  
 writeToFile:atomically, metoda, 427, 439  
 wskaźniki, 53, 271, 272, 273, 275  
 adresy w pamięci, 290  
 działania, 288  
 metody i funkcje, 277  
 na funkcje, 288  
 na łańcuchy znaków, 282, 284  
 na struktury, 275, 276  
 na tablice, 278, 279  
 operatory inkrementacji i dekrementacji, 280  
 wskaźnikowy struktur, operator, 276  
 wycieki pamięci, 148, 411  
 wyjątki, 193  
 obsługa, 193  
 wyliczenia, 206, 208, 209  
 wyliczeniowy, typ danych, 197  
 wyrażenia  
 arytmetyczne, 65  
 stałe, 61  
 warunkowe, 126  
 wyrównanie do prawej, 86  
 względne, ścieżki, 368

**X**

Xcode, 21, 22, 28  
kompilacja, 22, 26, 28  
korzystanie z dokumentacji, 306  
tworzenie nowego projektu, 22, 23, 24  
tworzenie nowej klasy, 130, 131  
uruchomienie programu, 26  
znajdowanie wycieków pamięci, 411  
XML, listy właściwości, 427

**Z**

zarządzanie pamięcią, 395, 409  
zbiory, 358  
zewnętrzne zmienne, 201, 202  
złożone testy relacyjne, 106  
zmienne  
egzemplarza, 45, 47, 401  
globalne, 201  
indeksowe, 83  
logiczne, 121, 123, 125  
lokalne, 90, 141, 142, 253, 254  
outlet, 454  
statyczne, 203  
wyświetlanie wartości, 34  
zakres dostępności, 200  
zewnętrzne, 201, 202  
zmiennoprzecinkowe wartości, 35  
znak nowego wiersza, 34  
znaku, funkcja, 110

# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



- 1. ZAREJESTRUJ SIĘ**
- 2. PREZENTUJ KSIĄŻKI**
- 3. ZBIERAJ PROWIZJĘ**

Zmień swoją stronę WWW  
w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>



# Objective-C Programowanie

*Vademecum profesjonalisty*

Zacznij pisać własne, funkcjonalne programy na iPhone'a, iPada oraz iPod'a Touch!

Tak jak iPhone, iPad czy iPod Touch błyskawicznie stały się obiektem pożądania milionów ludzi na całym świecie, tak samo szybko rynek upomniał się o specjalistów od tworzenia aplikacji na te innowacyjne urządzenia. Mimo że od 2007 roku, gdy Apple opublikowało zaktualizowaną wersję języka Objective-C, oznaczoną jako 2.0, minęło już trochę czasu, programistów znających ten język wciąż jest niewiele, a zapotrzebowanie na programy dla systemów iOS i Mac OS X stale rośnie. Warto zatem opanować ten potężny język, zarazem prosty i oferujący ogromne perspektywy zawodowe. Zwłaszcza że można go wykorzystać także na wielu innych platformach z kompilatorem GCC, a więc między innymi w systemach Unix, Linux i Windows.

Oto książka stworzona z myślą o programistach stawiających pierwsze kroki w języku Objective-C. Do wykorzystania zawartej tu wiedzy nie jest potrzebna znajomość języka C ani żadnego innego obiektowego języka programowania. Podstawą do nauki są dziesiątki ciekawych przykładów i ćwiczeń, ilustrujących użycie poszczególnych cech i ułatwiających zrozumienie poznawanych zagadnień. Książkę podzielono na trzy części. Pierwsza zawiera podstawowe wiadomości na temat języka Objective-C 2.0, a w drugiej znajdziesz opis klas dostępnych w bibliotece Foundation. Część trzecia poświęcona została niezwykle ważnym bibliotekom Cocoa i Cocoa Touch. Na końcu książki zamieszczono kurs pisania programów dla systemu iOS przy użyciu Software Development Kit iOS oraz biblioteki UIKit. Poznaj Objective-C i czerp z tego korzyści!

Stephen G. Kochan jest autorem lub współautorem kilku uznanych książek na temat języka C, między innymi *Programming in C*, *Programming in ANSI C* i *Topics in C Programming*, a także kilku publikacji dotyczących Uniksa, np. *Exploring the Unix System* i *Unix Shell Programming*. Pisze programy dla komputerów Macintosh od samego początku ich istnienia, a więc od roku 1984, kiedy pojawił się pierwszy Mac. Napisał książkę *Programming C for the Mac* wydaną w serii Apple Press Library, a także książkę *Beginning AppleScript*.

▼ *Proces tworzenia pierwszego programu w języku Objective-C*

▼ *Klasy, obiekty i metody, czyli podstawy programowania obiektowego*

▼ *Typy danych i wyrażenia oraz sposoby ich używania w programach*

▼ *Rodzaje pętli, których można używać w programach: for, while i do*

▼ *Szczegółowy opis technik pracy z klasami i obiektami*

▼ *Wprowadzenie do pojęcia dziedziczenia*

▼ *Polimorfizm oraz typowanie i wiązanie dynamiczne*

▼ *Techniki inicjacji obiektów, bloki, protokoły, kategorie, preprocesor oraz niektóre elementy języka C*

▼ *Obiekty liczbowe i łańcuchowe, kolekcje, systemy plików*

▼ *Techniki zarządzania pamięcią oraz proces kopiowania i archiwizowania obiektów*

▼ *Wprowadzenie do Cocoa i Cocoa Touch*

▼ *Pisanie programów dla systemu iOS i wprowadzenie do SDK iOS oraz biblioteki UIKit*

**helion.pl**  
księgarnia  
internetowa

Nr katalogowy: 7647

Księgarnia internetowa:  
<http://helion.pl>

Zamówienia telefoniczne:  
**0 801 339900**  
**0 601 339900**



**Helion**

Sprawdź najnowsze promocje:

📍 <http://helion.pl/promocje>

📖 Książki najchętniej czytane:

📍 <http://helion.pl/bestsellery>

📞 Zamów informacje o nowościach:

📍 <http://helion.pl/nowosci>

**Helion SA**

ul. Kościuszki 1c, 44-100 Gliwice

tel.: 32 230 98 63

e-mail: [helion@helion.pl](mailto:helion@helion.pl)

<http://helion.pl>

ślęgnij po WIĘCEJ



KOD KORZYŚCI

ISBN 978-83-246-3667-9



Cena 79,00 zł

Informatyka w najlepszym wydaniu