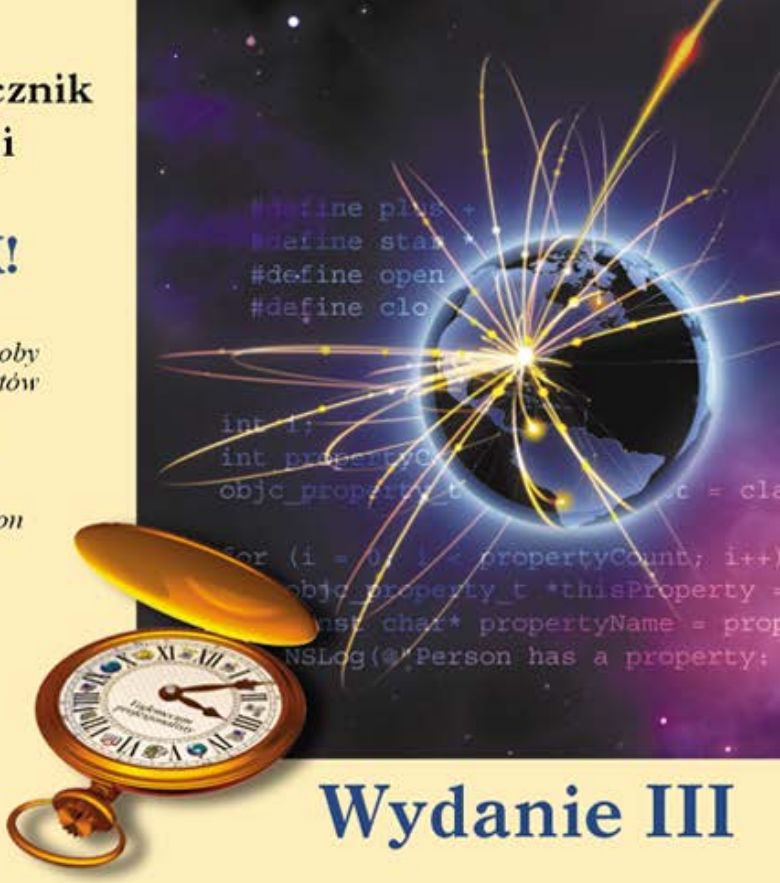
Praktyczny podręcznik tworzenia aplikacji na systemy iOS i Mac OS X!

- *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.

Wydawnictwo HELION
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?gimpbi
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

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. Objaśnienia 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 **egzemplarzem** 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 „napełnij 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:

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

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 *deklaracjeSkładowych* 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 robić 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 zwrotne

Deklarując metodę, trzeba poinformować kompilator, czy będzie ona zwracała jakąś wartość, a jeśli tak, to jakiego typu. Definicję typu zwrotnego 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;
```

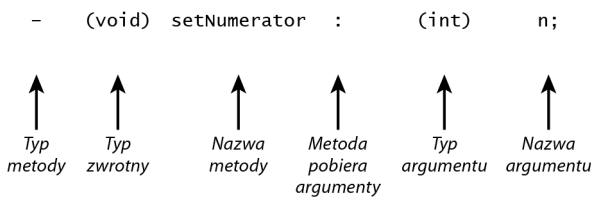
Argumenty metod

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 argumencie 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 zmiennych 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 wysyła 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 ułatwianie pisania programów, zarządzania nimi oraz ich rozszerzania. Stanie się to oczywiste nieco później.

Wróćmy jeszcze na chwilę do deklaracji obiektu `myFraction`:

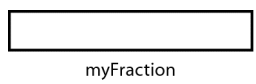
```
Fraction *myFraction;
```

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 (**wskaźnikiem**) 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.

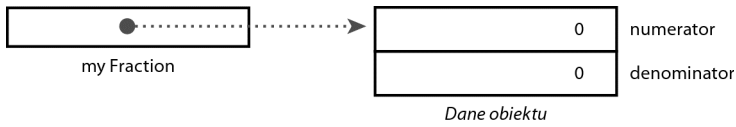
Rysunek 3.2.
*Deklaracja Fraction *myFraction;*



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:

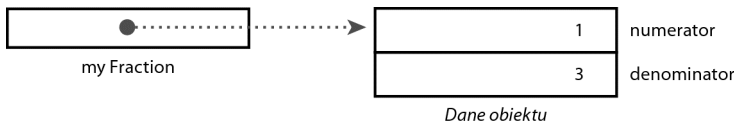


Rysunek 3.3. *Relacje między zmienną myFraction i jej danymi*

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);
}
```

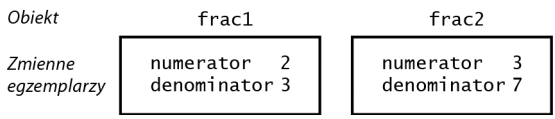
```
-(void) setNumerator: (int) n
{
    numerator = n;
}

-(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 zwrotną 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`. Zbiorcza nazwa, jaką określa się zarówno metody ustawiające, jak i pobierające, to **metody dostępowe** (ang. *accessor methods*).

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 egzemplarzy 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 dostępowy 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              alphaBetaRoutine
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

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Z

Objective-C Programowanie

Vademecum profesjonalisty

Zacznij pisać własne, funkcjonalne programy na iPhone'a, iPada oraz iPoda 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 niewielu, 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!

- *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*

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*.

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
http://helion.pl

ISBN 978-83-246-3667-9

9 788324 636679

Cena 79,00 zł