Stephen G. Kochan 著
林冀 范俊 朱奕欣 译

PEARSON
Broadview
www.broadview.com.cn

Objective-C 程序设计 第4版

Programming in Objective-C Fourth Edition

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

Objective-C程序设计

（第4版）

Programming in Objective-c Fourth Edition

Stephen G. Kochan 著

林冀 范俊 朱奕欣 译

電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

这是一本 Objective-C 编程领域最畅销的书籍，内容涵盖 Xcode 4.2 和自动引用计数（ARC）。

本书详细介绍了 Objective-C 和苹果 iOS、Mac 平台面向对象程序编程的知识。本书作者假设读者没有面向对象程序语言或者 C 语言（Objective-C 基础）编程经验，因此，初学者和有经验的程序员都可以使用这本书学习 Objective-C。读者不需要先学习底层的 C 语言编程，就可以了解面向对象编程。

本书结合独特的学习方法，在每章都编写有大量的小程序例子和练习，使 Objective-C 程序设计适合于课堂教学和自学。

本书已经为 iOS 5 和 Xcode 4.2 中的重大变更做了全面更新，最大的改动是引入了自动引用计数（ARC），并详细说明了如何在 Objective-C 编程过程中使用 ARC 提升和简化内存管理。

版权贸易合同登记号　图字：01-2011-5408

图书在版编目（CIP）数据

Objective-C 程序设计：第 4 版/（美）科昌（Kochan,S.G.）著；林冀，范俊，朱奕欣译. —北京：电子工业出版社，2012.9

书名原文：Programming in Objective-C, Fourth Edition

ISBN 978-7-121-18091-0

Ⅰ. ①O… Ⅱ. ①科… ②林… ③范… ④朱… Ⅲ. ①C 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2012)第 201941 号

策划编辑：张春雨

责任编辑：李利健

印　　刷：

装　　订：三河市双峰印刷装订有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱　邮编 100036

开　　本：787×980　1/16　印张：32.25　字数：614 千字

印　　次：2012 年 9 月第 1 次印刷

定　　价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

目　录

3

类、对象和方法

本章将介绍面向对象程序设计的一些关键概念，并开始使用 Objective-C 中的类。你需要学习少量术语，我们将用不那么正式的形式向你介绍。本章只会讲解一些基本的术语，因为一下讲太多的内容，你可能不容易接受。有关这些术语更精确的定义，可以参见本书附录 A。

3.1 到底什么是对象

对象就是一个物件。面向对象的程序设计可以看成一个物件和你想对它做的事情。这与 C 语言不同，C 语言通常称为过程性语言。在 C 语言中，通常是先考虑要做什么，然后才关注对象，这几乎总是与面向对象的思考过程相反。

我们举一个日常生活中的例子。假定你有一辆汽车，显然它是一个对象，而且是你拥有的一个对象。你并不是拥有任意一辆汽车，而是一辆特定的汽车，它由一家制造厂制造，可能在底特律，可能在日本，也可能在其他地方。你的汽车拥有一个车辆识别号码（vehicle identification number，VIN），在美国它能唯一标识你的汽车。

在面向对象的用语中，你的汽车是汽车的一个实例。如果继续使用术语，car 就是类的名称，这个实例就是从该类创建的。因此，每制造一辆新汽车，就会创建汽车类的一个新实例，而且汽车的每个实例都称为一个对象。

你的汽车可能是银白色的，内部装饰为黑色，是辆敞篷车或者有金属顶盖，等等。此外，你还用这辆汽车执行特定的操作。例如，驾驶汽车、加油、（可能还会）洗车、接受维修，等等，这些情况在表 3.1 中做了描述。

表 3.1　对象的操作

对　　象	使用对象执行的操作
你的汽车	驾驶
	加油
	洗车
	维修

表 3.1 所列的操作可以对你的汽车实现，也可以对其他汽车实现。例如，你姐姐驾驶她的汽车、洗车、加油，等等。

3.2　实例和方法

类的独特存在就是一个实例，对实例执行的操作称为方法。在某些情况下，方法可以应用于类的实例或者类本身。例如，洗车适用于一个实例（事实上，表 3.1 列出的所有方法都将作为实例方法）。找出一家制造厂制造了多少款汽车适用于类，所以它是一个类方法。

假设你有两辆使用装配线制作的汽车，它们看上去是一样的：都有相同的内部设计、相同的喷漆颜色，等等。它们可能同时启动，但是由于每部汽车是由它各自的主人驾驶的，这就使它们获得了自身独特的特征和属性的改变。例如，一辆汽车可能后来有了刮痕，而另一辆汽车可能行驶了更远的距离。每个实例或对象不仅包含从制造商那里获得的有关原始特性的信息，还包含它的当前特性，这些特性可以动态改变。当你驾驶汽车时，油箱的油渐渐耗尽，汽车越来越脏，轮胎也逐渐磨损。

对象使用方法可以影响对象的状态。如果方法是“给汽车加油”，那么执行这个方法之后，汽车的油箱将会加满。这个方法影响了汽车油箱的状态。

这里的关键概念是：对象是类的独特表示，每个对象都包含一些通常对该对象来说是私有的信息（数据）。方法提供访问和改变这些数据的手段。

Objective-C 采用特定的语法对类和实例应用方法：

```
[ ClassOrInstance method ];
```

在这条语句中，左方括号后要紧跟类的名称或者该类的实例名称，它后面

可以是一个或多个空格，空格后面是将要执行的方法。最后，使用右方括号和分号来终止。请求一个类或实例来执行某个操作时，就是在向它发送一条消息，消息的接收者称为接收者。因此，有另一种方式可以表示前面所描述的一般格式，具体如下：

```
[ receiver message ] ;
```

回顾上一个列表，使用这个新语法为它编写所有的方法。在此之前，你需要获得一辆新车，去制造厂购买一辆，语句如下：

```
yourCar = [Car new];    get a new car
```

向 car 类（消息的接收者）发送一条新消息，请求它卖给你一辆新车，获取到的对象（它代表你的独特汽车）将被存储到变量 yourCar 中。从现在开始，可用 yourCar 引用你的汽车实例，就是你从制造厂买的那辆汽车。

因为你到制造厂购买了一辆新汽车，所以这个新方法可叫做制造厂方法，或者类方法。对新车执行的其他操作都将是实例方法，因为它们应用于你的新车。下面是一些你可能为这辆新车编写的示例消息表达式：

```
[yourCar prep];         get it ready for first-time use
[yourCar drive];        drive your car
[yourCar wash];         wash your car
[yourCar getGas];       put gas in your car if you need it
[yourCar service];      service your car

[yourCar topDown];      if it's a convertible
[yourCar topUp];
currentMileage = [yourCar odometer];
```

最后一个示例显示的实例方法可返回信息，即当前的行驶里程，这通过里程表（odometer）可看出来。我们将该信息存储在程序中的 currentMileage 变量内。

这里有一个将特定值作为方法参数的例子，与方法直接调用有所不同：

```
[yourCar setSpeed: 55];  set the speed to 55 mph
```

你姐姐 Sue 可以对她自己的汽车实例使用相同的方法；

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

将同一个方法应用于不同的对象是面向对象程序设计背后的主要概念之

一，稍后将学到这方面更多的内容。

你可能无须在程序中对汽车进行操作。你的对象很可能是面向计算机的对象，例如窗口、矩形、一段文本，甚至是计算器或歌曲的播放列表。就像作用于汽车的方法，这些方法可能看上去类似于下列语句：

```
[myWindow erase];                Clear the window
theArea = [myRect area];         Calculate the area of the rectangle
[userText spellCheck];           Spell-check some text
[deskCalculator clearEntry];     Clear the last entry
[favoritePlaylist showSongs];    Show the songs in a playlist of favorites
[phoneNumber dial];              Dial a phone number
[myTable reloadData];            Show the updated table's data
n = [aTouch tapCount];           Store the number of times the display was tapped
```

3.3 用于处理分数的 Objective-C 类

现在，我们将用 Objective-C 定义一个实际的类，并学习如何使用类的实例。

同样，我们将先学习过程。因此，实际的程序范例可能不是特别实用，那些更加实际的内容将在稍后讨论。

假设要编写一个用于处理分数的程序，可能需要处理加、减、乘、除等运算。如果你还不知道什么是类，那么可以从一个简单的程序开始，代码如下：

代码清单 3-1

```
// 采用分数的简单程序

#import <Foundation/Foundation.h>

int main (int argc, char * argv[])
{
  @autoreleasepool {
    int  numerator = 1;
    int  denominator = 3;
    NSLog (@"The fraction is %i/%i", numerator, denominator);
  }
  return 0;
}
```

代码清单 3-1　输出

```
The fraction is 1/3
```

在代码清单 3-1 中，分数是以分子和分母的形式表示的。在@autoreleasepool 指令之后，main 中的前两行将变量 numerator 和 denominator 都声明为整型，并分别给它们赋予初值 1 和 3。这两个程序与下面的程序行等价：

```
int numerator, denominator;

numerator = 1;
denominator = 3;
```

将 1 存储到变量 numerator 中，将 3 存储到变量 denominator 中，这样就表示分数 1/3。如果需要在程序中存储多个分数，这种方法可能比较麻烦。每次要引用分数时，都必须引用相应的分子和分母，而且操作这些分数也相当困难。

如果能把一个分数定义成单个实体。用单个名称（例如 myFraction）来共同引用它的分子和分母，那就会更好。这种方法可以利用 Objective-C 来实现，从定义一个新类开始。

代码清单 3-2 通过一个名为 Fraction 的新类，重写了代码清单 3-1 中的函数。下面给出这个程序，随后将详细介绍它是如何工作的。

代码清单 3-2

```
// 使用分数的程序——类版本

#import <Foundation/Foundation.h>

//---- @interface 部分 ----

@interface Fraction: NSObject

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

@end

//---- @implementation 部分 ----


@implementation Fraction
{
  int  numerator;
  int  denominator;
}
```

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

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

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

@end

//---- program 部分 ----

int main (int argc, char * argv[])
{
  @autoreleasepool {
    Fraction  *myFraction;

    // 创建一个分数实例

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

    // 设置分数为 1/3

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

    // 使用打印方法显示分数

    NSLog (@"The value of myFraction is:");
    [myFraction print];
  }
  return 0;
}
```

代码清单 3-2　输出

```
The value of myFraction is:
1/3
```

从代码清单 3-2 的注释中可以看到，程序在逻辑上分为以下 3 个部分：

- @interface 部分
- @implementation 部分
- program 部分

其中，@interface 部分用于描述类和类的方法；@implementation 部分用于描述数据（类对象的实例变量存储的数据），并实现在接口中声明方法的实际代码；program 部分的程序代码实现了程序的预期目的。

> **注意**
>
> **也可以在 interface（接口）部分为类声明实例变量。从 Xcode 4.2 开始，已经可以在 implementation（实现）部分添加实例变量，这是为了能够以一种更好的方式来定义类。在后面章节中说明了原因。**

以上 3 个部分存在于每个 Objective-C 程序中，即使你可能不需要自己编写每一部分。你会看到，每一部分通常放在它自己的文件中。然而，目前来说，我们将它们放在一个单独的文件中。

3.4　@interface 部分

定义新类时，首先需要告诉 Objective-C 编译器该类来自何处。也就是说，必须为它的父类命名。其次，还必须定义在处理该类的对象时将要用到的各种操作或方法的类型。在随后的章节中你还会学习到，在@interface 部分中还会列出一些元素，称为属性。这部分的一般格式类似于下列语句：

```
@interface NewClassName: ParentClassName
  propertyAndMethodDeclarations;
@end
```

按照约定，类名以大写字母开头，尽管没有要求这么做。但这种约定能使其他人在阅读你的程序时，仅仅通过观察名称的第一个字母就能把类名和其他变量类型区分开来。让我们暂时转换一下话题，先谈论一些在 Objective-C 中制定名称的有关规则。

3.4.1 选择名称

在第 2 章“Objective-C 编程”中，使用了几个变量来存储整型值。例如，在代码清单 2-4 中使用变量 sum 来存储 50 和 25 两个数相加的结果。

Objective-C 语言还允许变量存储非整型的数据类型，在程序中使用变量之前，只要对它进行适当的声明即可。变量可以用于存储浮点数、字符，甚至是对象（或者更确切地说，是对对象的引用）。

制定名称的规则相当简单：名称必须以字母或下画线（_）开头，之后可以是任何（大写或小写）字母、下画线或者 0～9 之间的数字组合。下面是一些合法的名称：

- sum
- pieceFlag
- i
- myLocation
- numberOfMoves
- sysFlag
- ChessBoard

另外，根据规定，以下名称是非法的：

- sum$value——$是一个非法字符。
- piece flag——名称中间不允许插入空格。
- 3Spencer——名称不能以数字开头。
- int——这是一个保留字。

int 不能用做变量名，因为其用途对 Objective-C 编译器而言有特殊含义，这种用法称为保留名或保留字。一般来说，任何对 Objective-C 编译器有特殊意义的名称都不能作为变量名使用。

应该时刻记住，Objective-C 中的大写字母和小写字母是有区别的。因此，变量名 sum、Sum 和 SUM 均表示不同的变量。前面曾提到，在命名类时，类名通常以大写字母开头。另一方面，实例变量、对象以及方法的名称，通常以小写字母开头。为使程序具有可读性，名称中要用大写字母来表示新单词的开

头，例如下面的例子：

- AddressBook——可能是一个类名。
- currentEntry——可能是一个对象。
- current_entry——一些程序员还使用下画线作为单词的分隔符。
- addNewEntry——可能是一个方法名。

确定名称时，要遵循同样的标准：千万不要偷懒。要找到能反映变量或对象使用意图的名称。原因很明显，就像使用注释语句一样，富有意义的名称可以显著增强程序的可读性，并可在调试和文档编写阶段受益匪浅。事实上，因为程序具有更强的自解释性（self-explanatory），所以编写文档的任务将很可能大大减少。

以下是代码清单 3-2 中的@interface 部分：

```
//---- @interface 部分 ----

@interface Fraction: NSObject

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

@end
```

新类的名称为 Fraction，其父类为 NSObject。（我们将在第 8 章“继承”中讲解有关父类更详细的内容。）NSObject 类在文件 NSObject.h 中定义，导入 Foundation.h 文件时会在程序中自动包括这个类。

3.4.3　类方法和实例方法

必须定义各种方法才能使用 Fraction，且需要将分数的值设为特定的值。因为你不能直接访问分数的内部表示（就是直接访问它的实例变量），因此，必须编写方法来设置分子和分母。还要编写一个名为 print 的方法来显示分数的值。下面是 print 方法的声明，应该位于接口文件中：

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

开头的负号（-）通知 Objective-C 编译器，该方法是一个实例方法。此外，

只有一种选择，就是正号（+），它表示类方法。类方法是对类本身执行某些操作的方法，例如，创建类的新实例。

实例方法能够对类的实例执行一些操作，例如，设置值、检索值和显示值等。在制造出一辆汽车后，引用这个汽车实例时，可能要执行给它加油的操作。这个加油操作是对特定的汽车执行的，因此，它类似于实例方法。

1．返回值

声明新方法时，必须告诉 Objective-C 编译器该方法是否有返回值，如果有返回值，是哪种类型的值。要做到这一点，需要将返回类型放入开头的负号或者正号之后的圆括号。因此，声明

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

指定名为 currentAge 的实例方法将返回一个整型值。

类似地，语句

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

声明了一个返回双精度值的方法（第 4 章“数据型和表达式”将介绍有关这个数据型的更多内容）。

使用 Objective-C 中的 return 语句可以从方法中返回一个值，这与前一个程序例子中从 main 函数返回值的方式类似。

如果方法不返回值，可用 void 类型指明，语句如下：

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

这条语句声明了一个名为 print 的方法，它不返回任何值。在这种情况下，无须在方法的末尾执行一条 return 语句。或者也可以执行一条不带任何指定值的 return 语句，语句如下：

```
return;
```

2．方法的参数

代码清单 3-2 的@interface 部分声明了其他两个方法：

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

它们都是不返回值的实例方法。每个方法都有一个整型参数，这是通过参

数名前面的（int）指明的。就 setNumerator 来说，其参数名是 n。这个名称可以是任意的，它是用来引用参数的方法名称。因此，setNumerator 的声明指定向该方法传递一个名为 n 的整型参数，而且该方法没有要返回的值。这类似于 setDenominator 的声明，不同之处是后者的参数名是 d。

要注意声明这些方法的语法。每个方法名都以冒号结束，这告诉 Objective-C 编译器该方法会有参数。接下来，指定参数的类型，并将其放入一对圆括号中，这与为方法自身指定返回类型的方式十分相似。最后，使用象征性的名称来确定方法所指定的参数。整个声明以一个分号结束，其语法如图 3.1 所示。

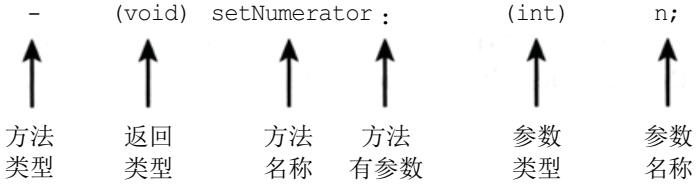

图 3.1　声明一个方法

如果方法接受一个参数，那么在引用该方法时，在方法名之后附加一个冒号。因此，setNumerator:和 setDenominator:是指定这两个方法的正确方式，每个方法都有一个参数。同样，在指定 print 方法时没有使用后缀的冒号，这表明此方法不带有任何参数。在第 7 章“类”中将学习如何指定带有多个参数。

3.5　@implementation 部分

与注释的一样，@implementation 部分包含声明在@interface 部分的方法的实际代码，且需要指定存储在类对象中的数据类型。就像术语指出的那样，在@interface 部分声明方法，并在@implementation 部分定义它们（也就是说，给出实际的代码）。@implementation 部分的一般格式如下：

```
@implementation NewClassName
{
  memberDeclarations;
}
 methodDefinitions;
@end
```

NewClassName 表示的名称与@interface 部分的名称相同。可以在父的名称

之后使用冒号，如同在@interface 部分使用冒号一样：

```
@implementation Fraction: NSObject
```

然而，它是可选的，而且通常并不这么做。

memberDeclarations 部分指定了哪种类型的数据将要存储到 Fraction 中，以及这些数据类型的名称。可以看到，这一部分放入自己的一组花括号内。对于 Fraction 类而言：

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

声明表示 Fraction 对象有两个整型成员，即 numerator 和 denominator。

在这一部分声明的成员称为实例变量。你将看到，每次创建新对象时，将同时创建一组新的实例变量，而且是唯一的一组。因此，如果拥有两个 Fraction，一个名为 fracA，另一个名为 fracB，那么每一个都将有自己的一组实例变量。就是说，fracA 和 fracB 各自将拥有 numerator 和 denominator。Objective-C 系统将自动追踪这些实例变量，对使用对象而言，这是一件令人愉快的事情。@implementation 部分中的 *methodDefinitions* 部分包含在@interface 部分指定的每个方法的代码中。与@interface 部分类似，每种方法的定义通过方法的类型（或者实例）、它的返回值、参数及它们的类型进行标识，我们并没有使用分号来结束该行，而是将之后的方法代码放入一对花括号中。需要注意的是，使用@synthesize 指令能够让编译器自动为你生成一些方法。具体内容在第 7 章中会详细描述。

以下是代码清单 3-2 的@implementation 部分：

```
//---- @implementation 部分 ----
@implementation Fraction
{
  int  numerator;
  int  denominator;
}

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

-(void) setNumerator: (int) n
```

```
{
   numerator = n;
}

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

@end
```

print 方法使用 NSLog 显示实例变量 numerator 和 denominator 的值。但是这个方法引用 numerator 还是 denominator 呢？它引用的实例变量包含在作为消息接收者的对象中。这是一个重要的概念，我们简单回顾一下。

setNumerator：方法带有一个名为 n 的整型参数，并简单地存储到实例变量 numerator 中。

setDenominator：将其参数 d 的值存储到实例变量 denominator 中。

3.6　program 部分

program 部分包含解决特定问题的代码，如果有必要，它可以跨越多个文件。前面提到，必须在其中某个地方有一个名为 main 的函数。通常情况下，这是程序开始执行的地方。以下是代码清单 3-2 的 program 部分：

```
//---- program 部分 ----

int main (int argc, char * argv[])
{
  @autoreleasepool {
    Fraction  *myFraction;

    // 创建一个分数实例并初始化

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

    // 设置分数为 1/3

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

    // 用打印方法显示分数
```

```
    NSLog (@"The value of myFraction is:");
    [myFraction print];
  }

  return 0;
}
```

在 main 中，通过以下程序行定义了一个名为 myFraction 的变量：

```
Fraction *myFraction;
```

这一行表示 myFraction 是一个 Fraction 类型的对象。也就是说，myFraction 用于存储新的 Fraction 类的值。变量名前面的星号（*）在后面会有更详细的描述。

现在，你拥有一个用于存储 Fraction 的对象，需要创建一个分数，就像要求制造厂制造一辆汽车一样，可以用以下程序行实现：

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

alloc 是 allocate 的缩写。因为要为新分数分配内存存储空间，表达式

```
[Fraction alloc]
```

向新创建的 Fraction 类发送一条消息。你请求 Fraction 类执行 alloc 方法，但是你从未定义过这个 alloc 方法，那么它来自何处呢？此方法继承自一个父类。第 8 章“继承”将会详细讨论这个主题。

如果向某个类发送 alloc 消息，便获得该类的新实例。在代码清单 3-2 中，返回值存储在变量 myFraction 中。alloc 方法保证对象的所有实例变量都变成初始状态。然而，这并不意味着该对象已经进行了适当的初始化，从而可以使用。在创建对象之后，还必须对它初始化。

这项工作可以通过代码清单 3-2 中的下一条语句来完成：

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

这里再次使用了一个并非自己编写的方法。init 方法用于初始化类的实例变量。注意，你正将 init 消息发送给 myFraction。也就是说，要在这里初始化一个特殊的 Fraction 对象，因此，它没有发送给类，而是发送给了类的一个实例。继续介绍下面的内容之前，务必理解这一点。

init 方法也可以返回一个值，即被初始化的对象。将返回值存储到 Fraction

的变量 myFraction 中。

代码创建新的实例并进行初始化的两行代码在 Objective-C 中特别常见，所以这两条消息通常组合在一起，语句如下：

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

内部消息表达式

```
[Fraction alloc]
```

将首先求值。可以看到，这条消息表达式的作用是创建实际的 Fraction。对它直接应用 init 方法，而不是像以前那样把创建的结果存储到一个变量中。所以，同样是先创建一个新的 Fraction，然后对它初始化。初始化的结果赋给了变量 myFraction。

作为最终的简写形式，经常把创建和初始化直接合并到声明行，语句如下：

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

回到代码清单 3-2，现在已经可以设置分数的值。程序行

```
// 设置分数为 1/3

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

用于完成这项工作。第一条消息语句向 myFraction 发送 setNumerator:消息，并提供一个值为 1 的参数。于是将控制转到 Fraction 类中定义的 setNumerator:方法。因为 Objective-C 系统知道 myFraction 是 Fraction 类的对象，所以它知道要执行这个类的方法。

在 setNumerator:方法中，传递来的值 1 存储在变量 n 中。该方法中唯一的程序行获得该值，并由实例变量 numerator 存储这个值。因此，myFraction 的分子已经被有效地设置为 1。

其后的消息用于调用 myFraction 的 setDenominator:方法。在 setDenominator:方法中，参数 3 被赋值给变量 d。然后把这个值存储到实例变量 denominator 中，这样就将 myFraction 赋值为 1/3。现在可以用代码清单 3-2 来实现显示此分数的值：

```
// 用打印方法显示分数

NSLog (@"The value of myFraction is:");
```

```
[myFraction print];
```

NSLog 调用仅显示以下文本：

```
The value of myFraction is:
```

使用以下消息表达式调用 print 方法：

```
[myFraction print];
```

在 print 方法中，将显示实例变量 numerator 和 denominator 的值，并用斜杠字符（/）分隔。

> **注意**
>
> **在过去，iOS 程序员需要给对象发送 release 消息，通知系统释放对象，这在内存管理系统中称为手工引用计数。现在由于使用了 Xcode 4.2，程序员不必再担心内存释放的问题，并且可以依靠系统来释放内存。通过自动引用计数（Automatic Reference Counting，简称 ARC）的机制就可以做到。在使用 Xcode 4.2 或以后版本编译新的应用时，ARC 默认会开启。**

似乎需要在代码清单 3-2 中编写大量的代码，才能完成代码清单 3-1 所实现的工作。对于这个简单的例子来说，确实如此，然而，使用对象的最终目的是使程序易于编写、维护和扩展。将来你会认识到这一点。

让我们重新回到 myFraction 的声明

```
Fraction *myFraction;
```

随后为其赋值。

myFraction 前的星号(*)表明 myFraction 是 Fraction 对象的引用(或指钟 。变量 myFraction 实际上并不存储 Fraction 的数 据（即分数的分子和分母），而是存储了一个引用（其实是内存地址），表明对象数据在内存中的位置。在声明 myFraction 时，它的值是未定义的，它没有被设定为任何值，并且没有默认值。在概念上，我们可以认为 myFraction 是一个能够容纳值的盒子。在初始化盒子时包含了一些未定义的值，它并没有被指定任何值，其图形表示如图 3.2 所示。

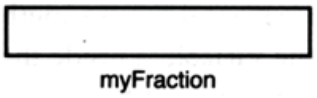

图 3.2　声明 Fraction * myFraction;

如果你创建一个新对象（例如，使用 alloc），就会在内存中为它保留足够的空间用于存储对象数据，这包括它的实例变量的空间，另外再多加了一点。通常，alloc 会返回存储数据的位置（对数据的引用），并赋给变量 myFraction。代码清单 3-2 中有如下语句：

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

图 3.3 中表明创建一个对象，并将对象的引用赋给 myFraction。

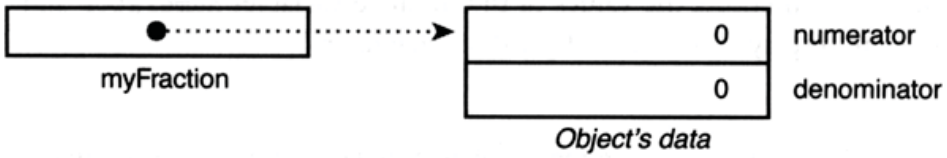

图 3.3　myFraction 及其数据的关系

注意

对象中存储的数据比标明的要多，不过不必在意这些。图中所示实例变量被指定为 0，这正是 alloc 方法做到的。然而对象始终没有被正确地初始化，你仍需使用 init 方法初始化新创建的对象。

注意图 3.3 中的箭头指示，这表明变量 myFraction 和创建的对象之间的关联。（存储在 myFraction 中的是内存地址。对象数据就存储在这个内存地址中。）

代码清单 3-2 中，随后就设定了分数的分子和分母。图 3.4 描述了 Fraction 对象完全初始化的过程，分子被设定为 1，分母被设定为 3。

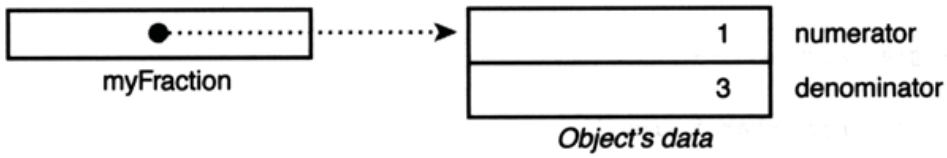

图 3.4　设定分数的分子和分母

下面的一个例子将展示如何在程序中使用多个分数。在代码清单 3-3 中，将一个分数设置为 2/3，另一个设置为 3/7，然后同时显示它们。

代码清单 3-3

```
// 使用分数的程序 - cont'd
```

```
#import <Foundation/Foundation.h>

//---- @interface 部分 ----

@interface Fraction: NSObject

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

@end

//---- @implementation 部分 ----

@implementation Fraction
{
  int  numerator;
  int  denominator;
}

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

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

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

@end

//---- program 部分 ----

int main (int argc, char *argv[])
{
  @autoreleasepool {

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

    // 设置第一个分数为 2/3
```

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

    // 设置第二个分数为 3/7

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

    // 显示分数

    NSLog (@"First fraction is:");

    [frac1 print];

    NSLog (@"Second fraction is:");
    [frac2 print];

  }
  return 0;
}
```

代码清单 3-3　输出

```
First fraction is:
2/3
Second fraction is:
3/7
```

@interface 和@implementation 部分与代码清单 3-2 一样，该程序创建了两个名为 frac1 和 frac2 的 Fraction 对象，然后将它们分别赋值为 2/3 和 3/7。注意，当 frac1 使用 setNumerator:方法将其分子设置为 2 时，实例变量 frac1 也将实例变量 numerator 设置为 2。同样，frac2 使用相同的方法将其分子设置为 3 时，它特有的实例变量 numerator 也被设置为 3。每次创建新对象时，它就获得了自己特有的一组实例变量。图 3.5 描述了这个情况。

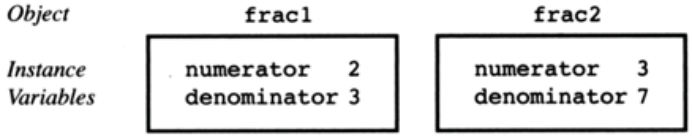

图 3.5　特有的实例变量

根据收到消息的对象，会引用正确的实例变量。因此，在

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

语句中，只要 setNumerator:方法用到名称 numerator，引用的都是 frac1 的 numerator，这是因为 frac1 是此消息的接收者。

3.7 实例变量的访问及数据封装

你已经看到处理分数的方法如何通过名称直接访问两个实例变量 numerator 和 denominator。事实上，实例方法总是可以直接访问它的实例变量的。然而，类方法则不能，因为它只处理本身，并不处理任何类实例（仔细想想）。但是，如果要从其他位置访问实例变量，例如，从 main 函数内部来访问，该如何实现？在这种情况下，不能直接访问这些实例变量，因为它们是隐藏的。将实例变量隐藏起来的这种做法实际上涉及一个关键概念——“数据封装”。它使得编写定义的人在不必担心程序员（即类的使用者）是否破坏类的内部细节的情况下，扩展和修改其定义。数据封装提供了程序员和其他开发者之间的良好隔离层。

通过编写特殊方法来检索实例变量的值，可以用一种新的方式来访问它们。编写 setNumerator:和 setDenominator:方法用于给 Fraction 类的两个实例变量设定值。为了获取这些实例变量的值，我们需要编写新的方法。例如，创建两个名为 numerator 和 denominator 的新方法，用于访问相应的 Fraction 实例变量，这些实例是消息的接收者。结果是对应的整数值，你将返回这些值。以下是这两个新方法的声明：

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

下面是定义：

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

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

注意，它们访问的方法名和实例变量名是相同的，这样做不存在任何问题（虽然似乎有些奇怪）。事实上，这是很常见的情况。代码清单 3-4 用来测试这两个新方法。

代码清单 3-4

```
// 访问实例变量的程序 - cont'd

#import <Foundation/Foundation.h>

//---- @interface 部分 ----

@interface Fraction: NSObject

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

@end

//---- @implementation 部分 ----

@implementation Fraction
{
  int  numerator;
  int  denominator;
}

-(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

//---- program 部分 ----

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

    // 设置分数为 1/3

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

    // 使用两个新的方法显示分数

    NSLog (@"The value of myFraction is: %i/%i",
        [myFraction numerator], [myFraction denominator]);
   }

   return 0;
}
```

代码清单 3-4　输出

```
The value of myFraction is 1/3
```

NSLog 语句显示发送给 myFraction:的两条消息的结果，第一条消息检索 numerator 的值，第二条则检索 denominator 的值。

```
NSLog (@"The value of myFraction is: %i/%i",
   [myFraction numerator], [myFraction denominator]);
```

在第一条消息调用时，numerator 消息会发送给 Fraction 类的对象 myFraction。在这个方法中，分数中 numerator 的实例变量的值被返回。记住，方法执行的上下文环境就是接收到消息的对象。当访问 numerator 方法并且返

回 numerator 实例变量值的时候，会取得 myFraction 的分子并返回，返回的整数传入 NSLog，从而显示出来。第二条消息调用时，denominator 方法会被调用并返回 myFraction 的分母，它仍通过 NSLog 显示。

顺便说一下，设置实例变量值的方法通常总称为设值方法（setter），而用于检索实例变量值的方法叫做取值方法（getter）。对 Fraction 而言，setNumerator：和 setDenominator:是设值方法，numerator 和 denominator 是取值方法。取值方法和设值方法通常称为访问器（accessor）方法。

确定你已经理解了设值方法和取值方法的不同。设值方法不会返回任何值，因为其主要目的是将方法参数设为对应的实例变量的值。在这种情况下并不需要返回值。另一方面，取值方法的目的是获取存储在对象中的实例变量的值，并通过程序返回发送出去。基于此目的，取值方法必须返回实例的值作为 return 的参数。

你不能在类的外部编写方法直接设置或获取实例变量的值，而需要编写设值方法和取值方法来设置或获取实例变量的值，这便是数据封装的原则。你必须通过使用一些方法来访问这些通常对“外界”隐藏的数据。这种做法集中了访问实例变量的方式，并且能够阻止其他一些代码直接改变实例变量的值。如果可以直接改变，会让程序很难跟踪、调试和修改。

这里还应指出，还有一个名为 new 的方法可以将 alloc 和 init 的操作结合起来。因此，程序行

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

可用于创建和初始化新的 Fraction。但用两步来实现创建和初始化的方式通常更好，这样可以在概念上理解正在发生两个不同的事件：首先创建一个对象，然后对它初始化。

3.8　小结

现在，你知道了如何定义自己的类，如何创建该类的对象或实例，以及如何向这些对象发送消息。我们将在随后的章节中介绍 Fraction。你将了解到如何向某个方法传递多个参数，如何将定义划分到不同的文件，同时还将了解到一些关键概念，如继承和动态绑定。然而，现在需要学习更多的数据类型，并

使用 Objective-C 编写表达式的更多内容。首先，请尝试完成以下练习，测试是否已经理解本章所讲的重点。

3.9 练习

1. 下列名称中，哪些是不合法的？为什么？

```
Int             playNextSong   6_05
_calloc         Xx             alphaBetaRoutine
clearScreen     _1312          z
ReInitialize    _              A$
```

2. 根据本章中的汽车示例，举出一个每天都要使用的对象。为这个对象确定一个类，并编写 5 个用于处理该对象的操作。
3. 给出练习 2 中的程序清单，使用以下语法：

```
[instance method];
```

   中的格式重写程序清单。
4. 设想你拥有一艘船、一辆摩托车和一辆汽车。列出对其中每个对象执行的操作。这些操作之间有重叠吗？
5. 根据练习 4，设想有一个名为 vehicle 的类和一个名为 myVehicle 的对象，这个对象可以是汽车、摩托车或船中的任何一个。如果编写以下操作：

```
[myVehicle prep];
[myVehicle getGas];
[myVehicle service];
```

   可以向这几个类的某一个对象执行一个操作，知道这样做的好处吗？
6. 在 C 这样的过程性语言中，思考涉及各种对象的操作，然后编写代码来执行这些操作。参见汽车例子，可用 C 语言编写洗交通工具的过程，然后在该过程中编写代码来处理清洗汽车、清洗船及清洗摩托车等操作。如果采用这种方法，同时希望添加一种新的交通工具（参见以前的练习），那么能指出使用这种过程性的方法比使用面向对象的方法有什么好处和缺点吗？
7. 定义一个名为 XYpoint 的类，用来保存笛卡儿坐标（x，y），其中 x 和 y

均为整数。定义一些方法，分别用来设置点的坐标 *x* 和 *y*，并检索它们的值。编写一个 Objective-C 程序，实现这个新类并测试它。

4

数据类型和表达式

本章将讲解 Objective-C 的基本数据类型，并描述构成算术表达式的一些基本规则。

4.1 数据类型和常量

你已经遇到过 Objective-C 的基本数据类型 int。回顾一下，声明为 int 类型的变量只能用于保存整型值，也就是不包含小数位数的值。

Objective-C 还提供了另外 3 种基本数据类型：float、double 和 char。声明为 float 类型的变量可以存储浮点数（即包含小数位数的值）。double 类型和 float 类型一样，通常，前者表示的范围大约是后者的两倍。char 数据类型可存储单个字符，例如字母 a、数字字符 6 或者一个分号（后面将详细讨论有关内容）。

在 Objective-C 中，任何数字、单个字符或者字符串通常都称为常量。例如，数字 58 表示一个常量整数值，字符串@"Programming in Objective-C is fun."表示一个常量字符串对象。完全由常量值组成的表达式叫做常量表达式。因此，下面的表达式是一个常量表达式，因为该表达式的每一项都是常量值：

```
128 + 7 - 17
```

然而，如果将 i 声明为整型变量，那么表达式就不是一个常量表达式：

```
128 + 7 - i
```

4.1.1 int 类型

整数常量由一个或多个数字的序列组成。序列前的负号表示该值是一个负

数。值 158、-10 和 0 都是合法的整数常量。数字中间不允许插入空格，并且不能使用逗号（因此，12,000 是一个非法的整数常量，它必须写成 12000）。

每个值无论是字符、整数还是浮点数字，都有与其对应的值域。这个值域与系统为特定类型的值分配的内存量有关。一般来说，在语言中没有规定这个量，它通常依赖于所运行的计算机，因此，叫做设备或机器相关量。例如，一个整数可在计算机上占用 32 位，或者可以使用 64 位存储。如果使用 64 位存储，整型变量将能够存储比 32 位更大的数值。

> **注意**
>
> **在 Mac OS X 中，提供了选择应用程序是在 32 位还是 64 位下编译。在前一种情况下，一个 int 占用 32 位；在后一种情况下，一个 int 占用 64 位。**

4.1.2 float 类型

声明为 float 类型的变量可以存储包含小数位的值。要区分浮点常量，可以看它是否包含小数点。值 3.、125.8 及–.0001 都是合法的浮点常量。要显示浮点值，可用 NSLog 转换字符%f 或者%g。

浮点常量也能用所谓的科学计数法来表示。值 1.7e4 就是使用这种计数法来表示的浮点值，它表示值 1.7×10^4。

如上所述，double 类型与 float 类型非常相似，只是 double 类型的变量可存储的范围大概是 float 变量的两倍。

4.1.3 char 类型

char 变量可存储单个字符。将字符放入一对单引号中就能得到字符常量。因此，'a'、';'和'0'都是合法的字符常量。第一个常量表示字母 a，第二个表示分号，第三个表示字符 0，它并不等同于数字 0。不要把字符常量和 C 语言风格的字符串混为一谈，字符常量是放在单引号中的单个字符，而字符串则是放在双引号中的任意个数的字符。正如在第 3 章提及的，前面有@字符并且放在双引号中的字符串是 NSString 字符串对象。

字符常量'\n'（即换行符）是一个合法的字符常量，尽管它似乎与前面提到的规则矛盾。这是因为反斜杠符号被认为是特殊符号。换句话说，Objective-C

编译器将字符'\n'看做单个字符，尽管它实际上由两个字符组成。其他特殊字符也是以反斜杠字符开头的。在 NSLog 调用中可以使用格式字符%c，以便显示 char 变量的值。

在代码清单 4-1 中，使用了基本的 Objective-C 数据类型。

代码清单 4-1

```
#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
  @autoreleasepool {
    int    integerVar = 100;
    float  floatingVar = 331.79;
    double doubleVar = 8.44e+11;
    char   charVar = 'W';

    NSLog (@"integerVar = %i", integerVar);
    NSLog (@"floatingVar = %f", floatingVar);
    NSLog (@"doubleVar = %e", doubleVar);
    NSLog (@"doubleVar = %g", doubleVar);
    NSLog (@"charVar = %c", charVar);
   }
   return 0;
}
```

代码清单 4-1　输出

```
integerVar = 100
floatingVar = 331.790009
doubleVar = 8.440000e+11
doubleVar = 8.44e+11
charVar = W
```

在程序输出的第二行，你会注意到指定给 floatingVar 的值 331.79，实际显示成了 331.790009。事实上，实际显示的值是由具体使用的计算机系统决定的。出现这种不准确值的原因在于，计算机内部使用了特殊的方式表示数字。使用计算器处理数字时，很可能遇到相同的不准确性。如果用计算器计算 1 除以 3，将得到结果.33333333，很可能结尾带有一些附加的 3。这一串 3 是计算器计算 1/3 的近似值。理论上，应该存在无限个 3。然而该计算器只能保存这些位的数字，这就是计算机的不确定性。同样的不确定性也出现在这里：在计算机内存

中不能精确地表示一些浮点值。

4.1.4 限定词：long、long long、short、unsigned 及 signed

如果直接把限定词 long 放在 int 声明之前，那么所声明的整型变量在某些计算机上具有扩展的值域。一个 long int 声明的例子为：

```
long int factorial;
```

这条语句将变量 fractorial 声明为 long 的整型变量。就像 float 和 double 变量一样，long 变量的具体范围也是由具体的计算机系统决定的。

要用 NSLog 显示 long int 的值，就要使用字母 l 作为修饰符，放在整型格式符号之前。这意味着格式符号%li 将用十进制格式显示 long int 的值。

你也可以使用 long long int 变量，甚至是具有更大范围带有浮点数的 long double 变量。

把限定词 short 放在 int 声明之前时，它告诉 Objective-C 编译器要声明的特定变量用来存储相当小的整数。之所以使用 short 变量，主要原因是对节约内存空间的考虑，当程序员需要大量内存而可用的内存量又十分有限时，就可用 short 变量来解决这个问题。

最后一个可以放在 int 变量之前的限定词，是在整数变量只用来存储正数的情况下使用的。以下语句

```
unsigned int counter;
```

向编译器声明，变量 counter 只用于保存正值。通过限制整型变量的使用，让它专门用于存储正整数，可以扩展整型变量的范围。

4.1.5 id 类型

id 数据类型可存储任何类型的对象。从某种意义说，它是一般对象类型。例如，程序行

```
id  graphicObject;
```

将 graphicObject 声明为 id 类型的变量。可声明方法使其具有 id 类型的返回值，如下：

```
-(id) newObject: (int) type;
```

这个程序行声明了一个名为 newObject 的实例方法，它具有名为 type 的单个整型参数并有 id 类型的返回值。

id 类型是本书经常使用的一种重要的数据类型。这里介绍该类型的目的是为了保持本书的完整性。id 类型是 Objective-C 中十分重要的特性，它是多态和动态绑定的基础，这两个特性将在第 9 章“多态、动态类型和动态绑定”中详细讨论。

表 4.1 总结了基本数据类型和限定词。

表 4.1　基础数据类型

类　　型	实　　例	NSLog 字符
char	'a'、'\n'	%c
short int	—	%hi、%hx、%ho
unsigned short int	—	%hu、%hx、%ho %hu、%hx、%ho
int	12、-97、0xFFE0、0177	%i、%x、%o
unsigned int	12u、100U、0XFFu	%u、%x、%o
long int	12L、-2001、0xffffL	%li、%lx、%lo
unsigned long int	12UL、100ul、0xffeeUL	%lu、%lx、%lo
long long int	0xe5e5e5e5LL、500ll	%lli、%llx、&llo
unsigned long long int	12ull、0xffeeULL	%llu、%llx、%llo
float	12.34f、3.1e-5f、0x1.5p10、0x1P-1	%f、%e、%g、%a
double	12.34、3.1e-5、0x.1p3	%f、%e、%g、%a
long double	12.34L、3.1e-5l	%Lf、$Le、%Lg
id	nil	%p

注意

在表 4.1 中，在整型常量中以 0 开头表示常量是八进制（基数 8）的，以 0x 开头或（0X）表示它是十六进制（基数 16）的，数字 0x.1p3 表示十六进制浮点常量。不必担心这些格式，这里只是为了使表格完整进行的总结。此外，前缀 f、l(L)、u(U)和 ll(LL)用来明确表示常量是 float、long、unsigned 和 long long 类型。

4.2 算术表达式

在 Objective-C 中，事实上与所有的程序设计语言一样，在两个数相加时使用加号（+），在两个数相减时使用减号（-），在两个数相乘时使用乘号（*），在两个数相除时使用除号（/）。这些运算符称为二元算术运算符，因为它们运算两个值或项。

4.2.1 运算符的优先级

你已经看到如何在 Objective-C 中执行简单的运算，例如，加法。下面的程序进一步说明了减法、乘法和除法运算。在程序中执行的最后两个运算引入了一个概念，即一个运算符比另一个运算符有更高的优先级。事实上，Objective-C 中的每一个运算符都有与之相关的优先级。

优先级用于确定包含多个运算符的表达式如何求值：优先级较高的运算符首先求值。如果表达式包含优先级相同的运算符，可按照从左到右或从右到左的方向来求值，具体按哪个方向求值取决于运算符。这就是通常所说的运算符结合性。

代码清单 4-2

```
// 说明各种算术运算符的用法

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
  @autoreleasepool {
    int  a = 100;
    int  b = 2;
    int  c = 25;
    int  d = 4;
    int  result;

    result = a - b;     // 减法
    NSLog (@"a - b = %i", result);

    result = b * c;     // 乘法
    NSLog (@"b * c = %i", result);

    result = a / c;     // 除法
    NSLog (@"a / c = %i", result);
```

```
    result = a + b * c;  // 优先级
    NSLog (@"a + b * c = %i", result);

    NSLog (@"a * b + c * d = %i", a * b + c * d);
  }
 return 0;
}
```

代码清单 4-2　输出

```
a - b = 98
b * c = 50
a / c = 4
a + b * c = 150
a * b + c * d = 300
```

在声明整型变量 a、b、c、d 及 result 之后，程序将 a 减 b 的结果指派给 result，然后用恰当的 NSLog 调用来显示它的值。

下一条语句

```
result = b * c;
```

将 b 的值和 c 的值相乘并将其结果存储到 result 中。然后用 NSLog 调用来显示这个乘法的结果。到目前为止，你应该很熟悉该过程了。

之后的程序语句引入了除法运算符——斜杠（/）。100 除以 25 得到 4，可用 NSLog 语句在 a 除以 c 之后立即显示。

在某些计算机系统中，尝试用一个整数除以 0 将导致程序异常终止或出现异常。即使程序没有异常终止，执行这样的除法所得的结果也毫无意义。在第 6 章“选择结构”中，将看到如何在执行除法运算之前检验除数是否为 0。如果除数为 0，可采用适当的操作来避免除法运算。

表达式

```
a + b * c
```

不会产生结果 2550（即 102×25）；相反，相应的 NSLog 语句显示的结果为 150。这是因为 Objective-C 与其他大多数程序设计语言一样，对于表达式中多重运算或项的顺序有自己的规则。通常情况下，表达式的计算按从左到右的顺序执行。然而，为乘法和除法运算指定的优先级比加法和减法的优先级要高。因此，Objective-C 认为表达式

```
a + b * c
```

等价于

```
a + (b * c)
```

（如果采用基本的代数规则，那么该表达式的计算方式是相同的。）

如果要改变表达式中项的计算顺序，可使用圆括号。事实上，前面列出的表达式是相当合法的 Objective-C 表达式。这样，可用表达式

```
result = a + (b * c);
```

替换代码清单 4-2 中的表达式，也可以获得同样的结果。然而，如果用表达式

```
result = (a + b) * c;
```

来替换，则赋给 result 的值将是 2550，因为要首先将 a 的值（100）和 b 的值（2）相加，然后将结果与 c 的值（25）相乘。圆括号也可以嵌套，在这种情况下，表达式的计算要从最里面的一对圆括号依次向外进行。只要确保结束圆括号和开始圆括号的数目相等即可。

从代码清单 4-2 中的最后一条语句可发现，对 NSLog 指定表达式作为参数时，无须将该表达式的结果先指派给一个变量，这种做法是完全合法的。表达式

```
a * b + c * d
```

可根据以上述规则，按照

```
(a * b) + (c * d)
```

即

```
(100 * 2) + (25 * 4)
```

来计算。

求出的结果 300 将传递给 NSLog 函数。

4.2.2 整数运算和一元负号运算符

代码清单 4-3 巩固了前面讨论的内容，并引入了整数运算的概念。

代码清单 4-3

```
// 更多的算术表达式
```

```
#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
  @autoreleasepool {
    int  a = 25;
    int  b = 2;
    float c = 25.0;
    float d = 2.0;

    NSLog (@"6 + a / 5 * b = %i", 6 + a / 5 * b);
    NSLog (@"a / b * b = %i", a / b * b);
    NSLog (@"c / d * d = %f", c / d * d);
    NSLog (@"-a = %i", -a);
  }
  return 0;
}
```

代码清单 4-3　输出

```
6 + a / 5 * b = 16
a / b * b = 24
c / d * d = 25.000000
-a = -25
```

前 3 条语句中，在 int 和 a、b 及 result 的声明之间插入了额外的空格，以便对齐每个变量的声明，使用这种方法书写语句可使程序更容易阅读。还可以注意到，在迄今出现的每个程序中，每个运算符前后都有空格。这种做法同样不是必需的，仅仅是出于美观上的考虑。一般来说，在允许单个空格的任何位置都可以插入额外的空格。如果能使程序更容易阅读，输入空格键的操作还是值得做的。

在代码清单 4-3 中，第一个 NSLog 调用中的表达式巩固了运算符优先级的概念。该表达式的计算按以下顺序执行：

（1）因为除法的优先级比加法高，所以先将 a 的值（25）除以 5。该运算将给出中间结果 5。

（2）因为乘法的优先级也高于加法，所以随后中间结果（5）将乘以 2（即 b 的值），并获得新的中间结果（10）。

（3）最后计算 6 加 10，并得出最终结果（16）。

第二条 NSLog 语句引入了一种新误解。你希望 a 除以 b，再乘以 b 的操作

返回 a（已经设置为 25）。但此操作并不会产生这一结果，在输出显示器上显示的是 24。难道计算机在某个地方迷失了方向？如果这样就太不幸了。其实该问题的实际情况是，这个表达式是采用整数运算来求值的。

如果回头看一下变量 a 和 b 的声明，你会想起它们都是作为 int 类型声明的。当包含两个整数的表达式求值时，Objective-C 系统都将使用整数运算来执行这个操作。在这种情况下，数字的所有小数部分将丢失。因此，计算 a 除以 b，即 25 除以 2 时，得到的中间结果是 12，而不是期望的 12.5。这个中间结果乘以 2，就得到最终结果 24。这样，就解释了出现“丢失”数字的情况。

在代码清单 4-3 的倒数第二个 NSLog 语句中看到，如果用浮点值代替整数来执行同样的运算，就会获得期望的结果。

决定使用 float 变量还是 int 变量应该基于变量的使用目的。如果无须使用任何小数位，就可以使用整型变量。这将使程序更加高效，换言之，它可以在大多数计算机上更快速地执行。另一方面，如果需要精确到小数位，那就很清楚应该选择什么。此时，唯一需要回答的问题是使用 float 还是 double。对此问题的回答取决于使用数据所需的精度以及它们的量级。

在最后一条 NSLog 语句中，使用了一元负号运算符对变量 a 的值取负。这个一元运算符是用于单个值的运算符，而二元运算符作用于两个值。负号实际上扮演了一个双重角色；作为二元运算符，它执行两个数相减的操作；作为一元（或单目）运算符，它对一个值取负。

与其他算术运算符相比，一元负号运算符具有更高的优先级，但一元正号运算符（+）除外，一元正号运算符和算术运算符的优先级相同。因此，表达式

```
c = -a * b;
```

将执行-a 乘以 b。

4.2.3 模运算符

本章介绍的最后一个运算符是模运算符，它由百分号（%）表示。通过分析代码清单 4-4 的输出，请尝试确定这种运算符的工作方式。

代码清单 4-4

```
// 模数运算符
```

```
#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
  @autoreleasepool {
    int a = 25, b = 5, c = 10, d = 7;

    NSLog (@"a %% b = %i", a % b);
    NSLog (@"a %% c = %i", a % c);
    NSLog (@"a %% d = %i", a % d);
    NSLog (@"a / d * d + a %% d = %i", a / d * d + a % d);
  }
  return 0;
}
```

代码清单 4-4　输出

```
a % b = 0
a % c = 5
a % d = 4
a / d * d + a % d = 25
```

注意，main 中的语句定义并初始化了变量 a、b、c 和 d，这些工作均在一条语句内完成。

你已经知道。NSLog 使用百分号之后的字符来确定如何输出下一个参数。然而，如果它后面紧跟另一个百分号，那么 NSLog 函数认为你的目的是想显示百分号，并在程序输出的适当位置插入一个百分号。

如果你总结出用模运算符%的功能是计算第一个值除以第二个值所得的余数，那就对了。在第一个例子中，25 除以 5 所得的余数是 0。如果用 25 除以 10，余数是 5，输出中的第二行语句可以证实。执行 25 除以 7 将得到余数 4，它显示在输出的第三行。

现在，我们把注意力转移到最后一条语句求值的表达式上。前面曾提到，Objective-C 使用整数运算来执行两个整数间的任何运算。因此，两个整数相除所产生的任何余数将被完全丢弃。如果使用表达式 a/d 表示 25 除以 7，将会得到中间结果 3。将这个结果乘以 d 的值（即 7），将会产生中间结果 21。最后，加上 a 除以 b 的余数，该余数由表达式 a%d 来表示，会产生最终结果 25。这个值与变量 a 的值相同并非巧合。一般来说，表达式

```
a / b * b + a % b
```

的值将始终与 a 的值相等。当然，这是在假定 a 和 b 都是整型值的条件下做出的。事实上，定义的模运算符%只用于处理整数。

就优先级而言，模运算符的优先级与乘法和除法的优先级相等。毫无疑问，这意味着表达式

```
table + value % TABLE_SIZE
```

等价于

```
table + (value % TABLE_SIZE)
```

4.2.4 整型值和浮点值的相互转换

若要更有效地开发 Objective-C 程序，必须理解 Objective-C 中浮点值和整型值之间进行隐式转换的规则。代码清单 4-5 表明数值数据类型间的一些简单转换。

代码清单 4-5

```
// Objective-C 中的基本转换

#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
  @autoreleasepool {
    float  f1 = 123.125, f2;
    int    i1, i2 = -150;

    i1 = f1;   // 浮点数到整数的转换
    NSLog (@"%f assigned to an int produces %i", f1, i1);

    f1 = i2;   // 整数到浮点数的转换
    NSLog (@"%i assigned to a float produces %f", i2, f1);

    f1 = i2 / 100;   // 整数除以整数
    NSLog (@"%i divided by 100 produces %f", i2, f1);

    f2 = i2 / 100.0;   // 整数除以浮点数
    NSLog (@"%i divided by 100.0 produces %f", i2, f2);

    f2 = (float) i2 / 100;   // 类型强制转换运算符
    NSLog (@"(float) %i divided by 100 produces %f", i2, f2);
  }
  return 0;
```

```
}
```

代码清单 4-5　输出

```
123.125000 assigned to an int produces 123
-150 assigned to a float produces -150.000000
-150 divided by 100 produces -1.000000
-150 divided by 100.0 produces -1.500000
(float) -150 divided by 100 produces -1.500000
```

在 Objective-C 中，只要将浮点值赋值给整型变量，数字的小数部分都会被删节。因此，在前一个程序中，把 f1 的值指派给 i1 时，数字 123.125 将被删节，这意味着只有整数部分（即 123）存储到了 i1 中。程序输出的第一行验证了上述程序就是这种情况。

把整型变量指派给浮点变量的操作不会引起数字值的任何改变，该值仅由系统转换并存储到浮点变量中。程序输出的第二行验证了这一情况：i2 的值（-150）进行了正确转换并存储到 float 变量 f1 中。

接下来的两行程序输出说明了在编写算术表达式时要记住的两点。第一点与整数运算有关，在前一章已经讨论了这一点。只要表达式中的两个运算数是整型（这一情况还适用于 short、unsigned 和 long 整型），该运算就将在整数运算的规则下进行。因此，由乘法运算产生的任何小数部分都将删除，即使该结果赋给一个浮点变量（如同我们在程序中所做的那样），也是如此。当整型变量 i2 除以整数常量 100 时，系统将该除法作为整数除法来执行。因此，-150 除以 100 的结果是-1，将-1 存储到 float 变量 f1 中。

前一个程序中的下一个除法涉及一个整数变量和一个浮点常量。在 Objective-C 中，对于任何处理两个值的运算，如果其中一个值是浮点变量或常量，那么这一运算将作为浮点运算来处理。因此，当 i2 的值除以 100.0 时，系统将除法作为浮点除法来计算，并产生结果-1.5，该结果将赋给 float 变量 f1。

4.2.5　类型转换运算符

你已经看到，在声明和定义方法时，如何将类型放入圆括号中来声明返回值和参数的类型。在表达式中使用类型时，它表示一个特殊的用途。

代码清单 4-5 中的最后一个除法运算

```
f2 = (float) i2 / 100;   // 类型强制转换运算符
```

引入了类型转换运算符。

为了求表达式的值，类型转换运算符将变量 i2 的值转换成 float 类型。该运算符永远不会影响变量 i2 的值。它是一元运算符，行为和其他一元运算符一样。因为表达式-a 永远不会影响 a 的值，因此，表达式（float）a 也不会影响 a 的值。

类型转换运算符比其他所有的算术运算符的优先级都高，但一元减号和一元加号运算符除外。当然，如果需要，可经常使用圆括号进行限制，以任何想要的顺序来执行一些项。

下面是使用类型转换运算符的另一个例子，表达式：

```
(int) 29.55 + (int) 21.99
```

在 Objective-C 中等价于

```
29 + 21
```

因为将浮点值转换成整数的后果就是舍弃其中的浮点值。表达式

```
(float) 6 / (float) 4
```

得到的结果为 1.5，与下列表达式的执行效果相同：

```
(float) 6 / 4
```

类型转换运算符通常用于将一般 id 类型的对象转换成特定类的对象。例如，

```
id  myNumber;
Fraction *myFraction;
  ...
myFraction = (Fraction *) myNumber;
```

将 id 变量 myNumber 的值强制类型转换成一个 Fraction 对象。转换结果赋给 Fraction 变量 myFraction。

4.3 赋值运算符

Objective-C 语言允许使用以下的一般格式将算术运算符和赋值运算符合并到一起：

```
op=
```

在这个格式中，op 是任何算术运算符，包括+、-、*、/和%。此外，op 还

可以是任何用于移位和屏蔽操作的位运算符，这些内容将在以后讨论。

请考虑下面这条语句：

```
count += 10;
```

通常所说的“加号等号”运算符（+=）将运算符右侧的表达式和左侧的表达式相加，再将结果保存到运算符左边的变量中。因此，上面的语句和以下语句等价：

```
count = count + 10;
```

表达式

```
counter -= 5
```

使用“减号等号”赋值运算符将 counter 的值减 5，它和下面这个语句等价

```
counter = counter - 5
```

下面是一个稍微复杂一些的表达式：

```
a /= b + c
```

无论等号右侧出现何值（或者 b 加 c 的和），都将用它除以 a，再把结果存储到 a 中。因为加法运算符比赋值运算符的优先级高，所以表达式会首先执行加法。事实上，除逗号运算符外的所有运算符都比赋值运算符的优先级高。而所有的赋值运算符的优先级相同。

在这个例子中，该表达式的作用和下列表达式相同：

```
a = a / (b + c)
```

使用赋值运算符的目的有 3 个：首先，程序语句更容易书写，因为运算符左侧的部分没有必要在右侧重写。其次，结果表达式通常容易阅读。最后，这些运算符的使用可使程序的运行速度更快，因为编译器有时在计算表达式时能够产生更少的代码。

4.4　Calculator 类

现在定义一个新类，我们将创建一个 Calculator 类，它是一个简单的四则运算计算器，可用来执行加、减、乘和除运算。类似于常见的计算器，这种计算器必须能够记录累加结果，即通常所说的累加器。因此，方法必须能够执行

以下操作：将累加器设置为特定值、将其清空（或设置为 0），以及在完成时检索它的值。代码清单 4-6 包括这个新类的定义和一个用于试验该计算器的测试程序。

代码清单 4-6

```
// 实现 Calculator 类

#import <Foundation/Foundation.h>

@interface Calculator: NSObject

// 累加方法
-(void)  setAccumulator: (double) value;
-(void)  clear;
-(double) accumulator;

// 算术方法
-(void) add: (double) value;
-(void) subtract: (double) value;
-(void) multiply: (double) value;
-(void) divide: (double) value;
@end

@implementation Calculator
{
   double accumulator;
}

-(void) setAccumulator: (double) value
{
   accumulator = value;
}

-(void) clear
{
   accumulator = 0;
}

-(double) accumulator
{
   return accumulator;
}

-(void) add: (double) value
{
```

```
    accumulator += value;
}

-(void) subtract: (double) value
{
    accumulator -= value;
}

-(void) multiply: (double) value
{
    accumulator *= value;
}

-(void) divide: (double) value
{
    accumulator /= value;
}
@end

int main (int argc, char *argv[])
{
  @autoreleasepool {
    Calculator *deskCalc = [[Calculator alloc] init];

    [deskCalc setAccumulator: 100.0];
    [deskCalc add: 200.];
    [deskCalc divide: 15.0];
    [deskCalc subtract: 10.0];
    [deskCalc multiply: 5];
    NSLog (@"The result is %g", [deskCalc accumulator]);
  }
  return 0;
}
```

代码清单 4-6　输出

```
The result is 50
```

Calculator 类只有一个实例变量，以及一个用于保存累加器值的 double 变量。方法定义的本身非常直观。

要注意调用 multiply 方法的消息：

```
[deskCalc multiply: 5];
```

该方法的参数是一个整数，而它期望的参数类型却是 double。因为方法的数值参数会自动转换以匹配期望的类型，所以此处不会出现任何问题。multiply:

期望使用 double 值，因此调用该函数时，整数 5 将自动转换成双精度浮点值。虽然自动转换过程会自己进行，但在调用方法时提供正确的参数类型仍是一个较好的程序设计习惯。

要认识到与 Fraction 类不同，Fraction 类可能使用多个不同的分数，在这个程序中可能希望只处理单个 Calculator 对象。然而，定义一个新类以便更容易处理这个对象仍是有意义的。在某个时候，你可能会为计算器添加一个图形前端，以便用户能够在屏幕上真正单击按钮，就像系统或手机中已安装的计算器应用程序一样。

在第 10 章“变量和数据类型”中会更多地讨论有关数据类型转换和位操作。

在以后的一些练习中，可以看到定义 Calculator 类的另一个好处，即便于扩展。

4.5 练习

1. 下列常量中，哪些是非法的？为什么？

```
123.456         0x10.5          0X0G1
0001            0xFFFF          123L
0Xab05          0L              -597.25
123.5e2         .0001           +12
98.6F           98.7U           17777s
0996            -12E-12         07777
1234uL          1.2Fe-7         15,000
1.234L          197u            100U
0XABCDEFL       0xabcu          +123
```

2. 编写一个程序，使用以下公式将华氏温度（F）27°转换成摄氏温度（C）：

```
C = (F - 32) / 1.8
```

不需要定义一个类来执行计算。只需要简单地列出表达式就满足要求。

3. 以下程序将输出什么结果？

```
#import <Foundation/Foundation.h>

int main (int argc, char *argv[])
{
  @autoreleasepool {
    char c, d;
```

```
        c = 'd';
        d = c;
        NSLog (@"d = %c", d);
    }
    return 0;
}
```

4．编写一个程序，求以下多项式的值（计算表达式时，只需直接计算，因为在 Objective-C 中没有幂指操作符）：

$3x^3 - 5x^2 + 6$

（令 $x = 2.55$）

5．编写一个程序，求下列表达式的值，并显示其结果（记住要使用指数格式显示结果）：

$(3.31 \times 10^{-8} + 2.01 \times 10^{-7}) / (7.16 \times 10^{-6} + 2.01 \times 10^{-8})$

6．复数包含两个部分：实部和虚部。如果 a 是实部，b 是虚部，那么符号

```
a + bi
```

可用来表示复数。

编写一个 Objective-C 程序，定义一个名为 Complex 的新类。依照为 Fraction 类创建的范例，为该类定义以下方法：

```
-(void) setReal: (double) a;
-(void) setImaginary: (double) b;
-(void) print;      // 显示为 a + bi
-(double) real;
-(double) imaginary;
```

编写一个测试程序测试这个新类和各个方法。

7．假设你正开发操作图形对象的函数库。从定义名为 Rectangle 的新类开始。目前，仅记录矩形的宽和高即可。开发一些方法用于设置矩形的宽和高、检索这些值以及计算矩形的面积和周长。假定这些矩形对象使用整数坐标栅格来描述矩形，例如，一台计算机屏幕。在这种情况下，假定矩形的宽和高都是整数值。

以下是 Rectangle 类的@interface 部分：

```
@interface Rectangle: NSObject
-(void) setWidth: (int) w;
-(void) setHeight: (int) h;
-(int) width;
```

```
-(int) height;
-(int) area;
-(int) perimeter;
@end
```

请编写 implementation 部分，并编写一个测试程序来测试新类的方法。

8．修改代码清单 4-6 中的 add:、subtract:、muntiply:和 divide:方法，使其返回累加器的结果值。测试这些新方法。

9．完成练习 8 后，把以下方法添加到 Calculator 类中并测试它们：

```
-(double) changeSign;  // 改变累加器的正负号
-(double) reciprocal;  // 累加器
-(double) xSquared;    // 累加器的平方
```

10．为代码清单 4-6 中的 Calculator 添加一项存储功能。实现以下方法声明并测试它们：

```
-(double) memoryClear;                          // 清理内存
-(double) memoryStore;                          // 设置内存为累加器
-(double) memoryRecall;                         // 设置累加器到内存
-(double) memoryAdd: (double) value;            // 添加值到内存
-(double) memorySubtract: (double) value;       // 与内存的值相减
```

为最后两组方法设置一个累加器，并能够对内存执行指定的操作。所有的方法都需要返回累加器的值。