Содержание

ГЛАВА 3

Объекты и сообщения Objective-C

Одним из первых объектно-ориентированных языков программирования, получивших широкое распространение, был Smalltalk. Он был разработан в 1970-х годах в Xerox PARC под руководством Алана Кея (Alan Kay) и стал широко известен в начале 1980-х годов. Цель языка Objective-C, созданного Брэдом Коксом (Brad Cox) и Томом Лавом (Tom Love) в 1986 году, заключалась в создании синтаксиса, аналогичного языку Smalltalk, и расширении языка программирования C. Язык программирования Objective-C был лицензирован компанией NeXT в 1988 году и стал основой для интерфейса API каркаса приложений NeXTStep. В конце концов компании NeXT и Apple объединились, каркас приложений NeXT эволюционировал в Cocoa, каркас приложений OS X, по-прежнему основанный на языке Objective-C. Эта история поясняет, почему Objective-C является базовым языком программирования для iOS. (Это также объясняет, почему имена классов Cocoa часто начинаются с "NS" — это всего лишь означает "NeXTStep".)

Усвоив основы C (глава 1) и природу объектно-ориентированного программирования (глава 2), вы готовы к встрече с Objective-C. В этой главе описаны структурные основы Objective-C; следующие две главы предоставляют более подробные сведения о том, как работают классы и экземпляры Objective-C. (Несколько дополнительных возможностей языка обсуждаются в главе 10.) Как и в случае языка программирования C, мое намерение состоит не в том, чтобы полностью описать язык Objective-C, а в том, чтобы, основываясь на собственном опыте и знаниях, обеспечить фундамент для его практического применения, в первую очередь тех аспектов языка, которые необходимы как основа программирования для iOS.

Ссылка на объект является указателем

Ссылка представляет собой именно то, что вы себе представляете, — способ выбора некоторой определенной отдельной сущности. Особенно хороший вид ссылки — имя. Если мы хотим обратиться к Сократу, то утомительно каждый раз описывать его как "толстый лысый парень, который вечно задает дурацкие вопросы". Куда проще указать его по имени — "Сократ". Переменная языка программирования C представляет собой эквивалент имени. Присваивание некоторого значения переменной приводит к тому, что данная переменная (то есть имя) становится ссылкой на это значение.

В языке программирования C каждая переменная должна быть объявлена как имеющая некоторый тип данных. В языке C очень немного фундаментальных типов данных. Эти типы определенно не готовы к роли объектных типов. Для того чтобы добавить к C объекты и тем самым превратить C в объектно-ориентированный язык C, Objective-C использует гибкость указателей C (см. главу 1). Указатель представляет собой тип данных C, но при этом он может

указывать на все, что угодно. Таким образом, в Objective-C каждая ссылка на объект является указателем (и Objective-C сам заботится о выяснении, на что этот указатель указывает).

Тот факт, что ссылки на объекты в Objective-C являются указателями, особенно очевидно в случае ссылки на экземпляр (см. главу 2). В объектно-ориентированном языке, таком как Objective-C, типом экземпляра является его класс. Таким образом, хотелось бы использовать в Objective-C имя класса так, как в C мы используем имя любого типа данных. И позволяют это сделать именно указатели. С одной стороны, указатели удовлетворяют требованию C о том, что ссылка должна быть некоторым определенным типом данных C, а с другой стороны — требованию Objective-C о том, что мы должны иметь возможность указать любой тип класса из огромного их множества. Если в Objective-C переменная явно ссылается на экземпляр класса `MyClass`, то она имеет тип `MyClass*`, т.е. является указателем на `MyClass`. В общем случае в Objective-C ссылка на экземпляр класса является указателем, а имя типа данных, на который указывает этот указатель, представляет собой имя класса этого экземпляра.

Обратите внимание на соглашение об использовании прописных букв. Обычно имена переменных начинаются со строчной буквы, а имена классов — с прописной.

Как упоминалось в главе 1, тот факт, что в Objective-C ссылка на экземпляр является указателем, обычно не вызывает каких-либо трудностей, так как указатели последовательно используются везде в языке. Например, сообщение, направленное экземпляру, отправляется указателю, так что нет необходимости в разыменовании последнего. Действительно, установив, что переменная, представляющая экземпляр, является указателем, можно забыть об этом факте и просто непосредственно работать с этой переменной:

```
NSString* s = @"Hello, world!";
NSString* s2 = [s uppercaseString];
```

Один раз установив, что переменная `s` имеет тип `NSString*`, вы больше никогда не будете разыменовывать `s` (т.е. никогда не будет говорить о `*s`) для доступа к "реальным" строкам `NSString`. Так что все выглядит так, как будто указатель является реальным типом `NSString`. Таким образом, в предыдущем примере, после того как переменная `s` была объявлена как указатель на объект класса `NSString`, сообщение `uppercaseString` отправляется непосредственно этой переменной. (Сообщение `uppercaseString` запрашивает у `NSString` создание и возвращение версии хранимой объектом строки в верхнем регистре; таким образом, после выполнения приведенного кода `s2` представляет собой строку `@"HELLO, WORLD!"`.)

Связь между указателем, экземпляром и классом этого экземпляра столь тесная, что вполне естественно говорить о выражении наподобие `MyClass*` как об означающем "экземпляр `MyClass`", а о значении `MyClass*` как о "`MyClass`". Программист на языке Objective-C, говоря о приведенном выше примере, просто сказал бы, что `s` представляет собой `NSString`, что `uppercaseString` возвращает `NSString` и так далее. Это прекрасно, что можно так говорить, и я постоянно делаю это сам — надо только помнить, что это просто в определенном смысле сокращение. Такое выражение на самом деле означает "экземпляр `NSString`", что, в свою очередь, означает, что поскольку экземпляр представлен в виде указателя C, то он является не чем иным, как `NSString*`, т.е. указателем на `NSString`.

Хотя тот факт, что ссылки на экземпляры в Objective-C являются указателями, не вызывает каких-либо особых трудностей, вы все равно должны осознавать, что такое указатели и как они работают. Как я подчеркивал в главе 1, когда вы работаете с указателями, ваши действия имеют особый смысл. Так что вот некоторые основные факты об указателях, которые вам следует иметь в виду при работе со ссылками на экземпляры в Objective-C.

Распространенная ошибка начинающего — забыть о звездочке в объявлении экземпляра, на что компилятор позже отреагирует загадочными сообщениями типа “Interface type cannot be statically allocated”.

Ссылки на экземпляры, инициализация и nil

Простое объявление типа ссылки на экземпляр не влечет за собой существование соответствующего экземпляра. Например:

```
NSString* s; // Только объявление; экземпляра, на который
             // указывает s, не существует
```

После этого объявления `s` имеет тип указателя на `NSString`, но на самом деле эта переменная не указывает на `NSString`. Вы создали указатель, но не предоставили объект `NSString`, на который он должен указывать. Он пока что ждет, пока вы укажете, на что он должен указывать (обычно путем присваивания, как мы это делали с помощью строки `@"Hello, world!"` выше). Такое присваивание инициализирует переменную, придавая ей первоначальное имеющее смысл значение надлежащего типа.

Можно объявить переменную как ссылку на экземпляр в одной строке кода, а инициализировать ее позже, как в приведенном примере:

```
NSString* s;
// ... Прошло время ...
s = @"Hello, world!";
```

Однако обычно так не делается. Гораздо более распространена практика объявления и инициализации переменной в одной строке кода (если это возможно):

```
NSString* s = @"Hello, world!";
```

Объявление без инициализации до появления механизма ARC (автоматического подсчета ссылок, см. главу 12) приводило к достаточно опасным ситуациям.

```
NSString* s;
```

Без механизма ARC переменная `s` после объявления может иметь *любое* значение. Беда в том, что эта переменная *претендует* на то, чтобы являться указателем на `NSString`. Обманувшись, вы можете *рассматривать* мусор в памяти, на который указывает переменная `s`, как строку `NSString`. Это может привести к серьезным неприятностям, если вы попытаетесь использовать мусор в качестве экземпляра. Отправка сообщения мусору или использование его иным образом может привести к сбою программы. Еще хуже — это может *не* привести к сбою программы, а всего лишь заставить ее работать неверно, — и при этом будет очень, очень сложно выяснить, в чем же причина такого поведения программы.

Назначение ссылке значения `nil` приводит к тому, что, хотя она и не указывает ни на какой реальный экземпляр, она не указывает и на мусор. До появления механизма ARC распространенной защитой от “мусорных указателей” была инициализация каждого указателя в момент его объявления значением `nil`, если инициализация реальным объектом по каким-то причинам оказывается в этот момент невозможной:

```
NSString* s = nil;
```

Небольшим, но очаровательным дополнением к возможностям механизма ARC является то, что такое присваивание выполняется автоматически, неявно и незаметно, в случае объявления переменной без ее инициализации:

```
NSString* s; // ARC присваивает s значение nil
```

Что такое nil? Это разновидность нуля — нуль, который подходит для присваивания ссылке на экземпляр. Значение nil просто означает: "эта ссылка на экземпляр не указывает ни на один реальный экземпляр". Действительно, вы можете проверить, указывает ли ссылка на реальный экземпляр, сравнив ее с nil. Это очень распространенная практика:

```
if (nil == s) // ...
```

Как я упоминал в главе 1, явное сравнение с nil не является строго необходимым; поскольку nil является разновидностью нуля, а также, так как нуль в условии означает ложь, тот же тест можно выполнить проще:

```
if (!s) // ...
```

Я буду использовать именно эту, вторую разновидность проверки, но некоторые программисты будут считать, что у меня плохой стиль программирования. Первая разновидность имеет то преимущество, что ее реальный смысл сразу становится очевиден всем и не приходится полагаться на некоторую неявную функциональность C. В первой разновидности nil также преднамеренно занимает место слева от оператора ==, так что если программист случайно пропустит один знак равенства, выполняя вместо сравнения присваивание, компилятор обнаружит эту ошибку (потому что присваивать что-то nil нельзя).

Многие методы Cocoa используют возвращаемое значение nil вместо ожидаемого экземпляра для указания, что что-то *выполнено* не так, как надо. Чтобы убедиться, что вызов метода прошел успешно, вы должны проверить возвращаемое значение на равенство nil. Например, документация метода stringWithContentsOfFile:encoding:error: класса NSString гласит, что он "возвращает строку, полученную путем считывания данных из файла с именем path, используя кодировку enc. Если файл не может быть открыт или имеется ошибка кодирования, возвращается значение nil". Так что, как я писал в главе 1, этот метод должен вызываться следующим образом:

```
NSString* path =         // Какое-то имя
NSStringEncoding enc =   // Какое-то значение
NSError* err = nil;
NSString* result =
    [NSString stringWithContentsOfFile: path
     encoding: enc error: &err];
```

Почему stringWithContentsOfFile:encoding:error: имеет такой вид? По сути этот метод может возвращать *два* результата. Он возвращает строку, которую мы сохраняем, присваивая ее указателю на NSString, и которую мы называем результатом. Но если произошла ошибка, то метод задает значение другого объекта, а именно объекта NSError. Идея заключается в том, что вы можете впоследствии изучить этот объект NSError и выяснить, что именно пошло не так. (Возможно, не было файла с указанным именем, или он имел неверную кодировку.) Передавая указатель на указатель на NSError, вы позволяете методу выполнить описанное действие. Перед вызовом stringWithContentsOfFile:encoding:error: переменная err была инициализирована значением nil; во время вызова stringWithContentsOfFile:encoding:error:, если произошла ошибка, указатель перенаправляется, тем самым присваивая err значение NSError, описывающее ошибку. (Перенаправление указателя таким образом иногда называют *косвенным* обращением.)

Таким образом, следующий шаг после вызова этого метода и сохранения результата — проверить равенство результата nil, просто чтобы убедиться, что действительно получен экземпляр строки. Если результат не nil, все хорошо; это и есть та строка, которую вы хотели получить. Но если результат *равен* nil, то надо исследовать NSError, чтобы узнать, что пошло неправильно.

```
NSString* path =       // Какое-то имя
NSStringEncoding enc = // Какое-то значение
NSError* err = nil;
NSString* result =
    [NSString stringWithContentsOfFile: path
     encoding: enc error: &err];
if (nil == result) { // Ой! Что-то пошло не так...
    // Из err можно узнать, что именно случилось
}
```

Этот шаблон часто используется в каркасе Cocoa. *Не поймите его неправильно!* Очень распространенная ошибка начинающих программистов — вызов метода `stringWithContentsOfFile:encoding:error:` с немедленной проверкой значения переменной ошибки (в данном случае — `err`). Не делайте этого! Если ошибки не было, значение переменной `err` никак не гарантируется. Вместо этого начните с проверки результата (в данном случае — переменной `result`). И только если результат показывает, что произошла ошибка, то вот тогда и только тогда вы должны исследовать значение `NSError`, которое было установлено путем косвенного обращения.

В исходном тексте на чистом C иногда приходится встречаться с "указателем на ничто", выраженным как `NULL`. Функционально `NULL` и `nil` эквивалентны, так что не стесняйтесь везде использовать только `nil`.

Ссылки на экземпляры и присваивание

Как я уже говорил в главе 1, присвоение указателю не приводит к изменению значения, которое находится "на дальнем конце указателя", на которое он указывает; это просто перенацеливание указателя. Кроме того, присваивание значения одного указателя другому перенацеливает указатель таким образом, что после этого оба указателя указывают на один и тот же объект. Если вы забудете эти простые факты, результаты будут в диапазоне от удивительных до катастрофических.

Предположим, что мы реализовали класс `Stack`, описанный в главе 2, и рассмотрим следующий код:

```
Stack* myStack1 = // ... Создание экземпляра Stack
                  // и инициализация myStack1 ...
Stack* myStack2 = myStack1;
```

Распространенное заблуждение — что присваивание `myStack2 = myStack1` создает новый, отдельный экземпляр, который дублирует `myStack1`. Это вовсе не так. Присваивание не создает новый экземпляр; оно просто приводит к тому, что `myStack2` указывает на тот же экземпляр, что и `myStack1`. Да, можно сделать новый экземпляр, который дублирует данный, но такая возможность не является данностью, а это действие не выполняется путем простого присваивания. (Как создаются экземпляры-дубликаты, описано в главе 10; см. описание протокола `NSCopying` и метода `copy`.)

Кроме того, в общем случае экземпляры изменяемые: они, как правило, имеют переменные экземпляров, которые можно изменить. Если две ссылки указывают на один и тот же экземпляр, то когда экземпляр изменяется с помощью одной ссылки, это изменение становится видимым через другие ссылки.

```
Stack* myStack1 = // ... Создание экземпляра Stack
                  // и инициализация myStack1 ...
Stack* myStack2 = myStack1;
[myStack1 push: @"Hello"];
```

```
[myStack1 push: @"World"];
NSString* s = [myStack2 pop];
```

Когда мы снимем элемент с вершины стека `myStack2`, значением переменной `s` окажется `@"World"`, хотя мы ничего не помещали в стек `myStack2`; стек же `myStack1` после этого содержит только строку `@"Hello"`, хотя мы ничего не удаляли из стека `myStack1`. Это связано с тем, что мы внесли две строки в `myStack1` и удалили одну строку из `myStack2`, а `myStack1` является `myStack2` — в том смысле, что это два указателя на один и тот же экземпляр стека. Это прекрасно до тех пор, пока вы понимаете и правильно используете это поведение.

Впрочем, иногда такое поведение является нежелательным. Если ваша программа имеет более чем одну ссылку на один и тот же экземпляр, можно получить удивительные результаты только потому, что (как и в предыдущем примере) этот экземпляр можно изменить с помощью одной ссылки в то время, когда владелец другой ссылки ничего подобного не ожидает. В реальной жизни проблемы такого рода могут возникнуть, в частности, потому, что экземпляры могут иметь переменные экземпляров, которые указывают на другие объекты, и эти указатели могут сохраняться до тех пор, пока существуют сами экземпляры. Предположим, у нас есть объект `myObject` и мы передаем ему ссылку на объект нашего стека.

```
Stack* myStack = // ... Создание экземпляра Stack
                 // и инициализация myStack ...
[myObject doSomethingWithThis: myStack]; // передача myStack
                                         // в myObject
```

После этого кода `myObject` имеет указатель на тот же экземпляр, на который мы уже указываем с помощью `myStack`. Поэтому мы должны быть очень осторожны и внимательны. Объект `myObject` теперь может изменять наш `myStack` прямо у нас под носом! Более того, этот объект `myObject` может *хранить* свою ссылку на экземпляр стека (например, в переменной экземпляра) и изменить его *позже* — возможно, намного позже, и способом, который нас удивит. Такая ситуация может быть создана преднамеренно, но и в этом случае следует быть внимательным и точно понимать, что вы делаете и зачем (рис. 3.1).

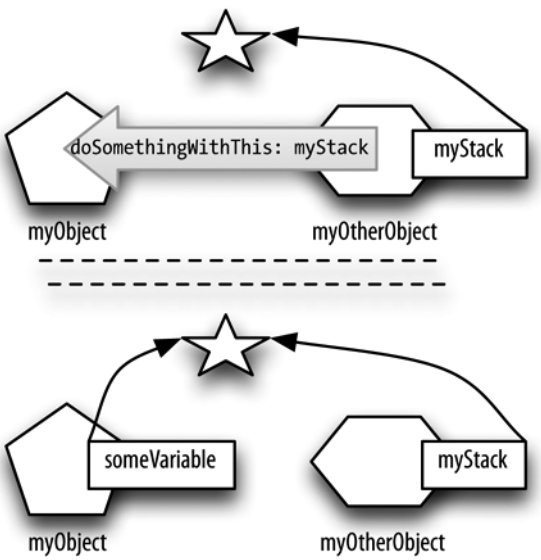

Рис. 3.1. Два экземпляра с указателями на один и тот же третий экземпляр

Ссылки на экземпляры и управление памятью

"Указательная" природа ссылок на экземпляры в Objective-C имеет определенные последствия для системы управления памятью. Правила области видимости и, в частности, времени жизни переменных в чистом C обычно довольно просты: если вы создаете переменную путем ее объявления в некоторой области видимости, то когда эта область перестает существовать, перестает существовать и переменная. Такого рода переменные называются автоматическими (K&R 1.10). Например:

```
void myFunction() {
    int i; // Выделено место для хранения int
    i = 7; // В этом месте размещено значение 7
} // Область видимости закончилась, выделенное
  // место и его содержимое уничтожаются
```

Но в случае указателя есть два вида памяти, о которых следует беспокоиться: сам указатель, который представляет собой целое число, представляющее адрес в памяти, и все, что находится по этому адресу. В языке программирования C при автоматическом уничтожении указателя ничто не ведет к автоматическому уничтожению того, на что он указывает:

```
void myFunction() {
    NSString* s = @"Hello, world!";      // Указатель и NSString
    NSString* s2 = [s uppercaseString];  // Указатель и NSString
}  // Два указателя прекращают существование...
   // ... но что происходит со строками, на которые они указывали?
```

Некоторые объектно-ориентированные языки программирования, в которых ссылка на экземпляр является указателем, автоматически управляют памятью, на которую указывают ссылки на экземпляры (примерами являются REALbasic и Ruby). Но Objective-C не является одним из таких языков. Поскольку язык C ничего не говорит об автоматическом уничтожении того, на что указывают ссылки на экземпляры, Objective-C реализует явный механизм управления памятью. Позже (в главе 12) я расскажу об этом механизме и о том, какие обязанности ложатся на программиста при его использовании. К счастью, при наличии механизма ARC эти обязанности меньшие, чем были ранее; тем не менее памятью по-прежнему необходимо управлять, и вы по-прежнему должны отчетливо понимать, как работает управление памятью экземпляров.

Методы и сообщения

Метод в Objective-C определен как часть класса. Он имеет три аспекта.

Он может быть методом класса или методом экземпляра

Если это метод класса, вы вызываете его с помощью сообщения самому классу. Если это метод экземпляра, вы вызываете его с помощью сообщения экземпляру класса.

Он имеет параметры и возвращаемое значение

Как и функции C (глава 1), методы Objective-C принимают некоторое количество параметров; каждый параметр имеет некоторый конкретный тип. И, как и функции C, методы могут иметь возвращаемое значение, также определенного типа. Если метод не возвращает ничего, его возвращаемый тип объявляется как `void`.

Он имеет имя

Имя метода Objective-C должно иметь столько двоеточий, сколько параметров он принимает; и, если метод принимает параметры, его имя должно завершаться двоеточием.

Вызов метода

Чтобы отправить сообщение объекту, используется выражение сообщения, которое также называют для простоты и по аналогии с функциями C вызовом метода. Как вы уже и сами догадались, синтаксис отправки сообщения объекту включает использование квадратных скобок. Первым в квадратных скобках указывается объект, которому отправляется сообщение; этот объект является получателем сообщения. Затем следует само сообщение:

```
NSString* s2 = [s uppercaseString]; // Отправка сообщения
             // "uppercaseString" объекту s и присваивание
             // результата переменной s2
```

Если сообщение представляет собой метод, который получает параметры, значение каждого соответствующего аргумента указывается после двоеточия:

```
[myStack1 push: @"Hello"]; // Отправка объекту myStack1
    // сообщения "push:" с одним аргументом типа
    // NSString и значением @"Hello"
```

Чтобы отправить сообщение классу (вызов метода класса), класс может быть представлен его именем:

```
NSString* s = [NSString string]; // Сообщение "string"
                                 // для класса NSString
```

Для отправки сообщения экземпляру (вызов метода экземпляра) требуется ссылка на этот экземпляр, которая (как вы знаете) представляет собой указатель:

```
NSString* s                  // s инициализируется как
    = @"Hello, world!";      // экземпляр класса NSString
NSString* s2                 // Отправка сообщения
    = [s uppercaseString];   // "uppercaseString" объекту s
```

Можно отправлять метод класса классу, а метод экземпляра экземпляру, независимо от того, как получен и представлен этот класс или экземпляр. Например, строка `@"Hello, world!"` сама по себе является экземпляром `NSString`, так что вполне можно написать

```
NSString* s2 = [@"Hello, world!" uppercaseString];
```

Если метод не принимает никаких параметров, то его имя не содержит двоеточия, как, например, метод экземпляра `uppercaseString` класса `NSString`. Если метод принимает один параметр, то его имя содержит одно двоеточие, которое представляет собой последний символ имени метода, наподобие метода экземпляра `push:` нашего гипотетического стека. Если метод принимает два и более параметра, его имя содержит соответствующее количество двоеточий и последнее двоеточие является последним символом имени метода. В минимальном случае имя метода заканчивается всеми его двоеточиями. Например, метод, принимающий три параметра, может называться `hereAreThreeStrings:::`. Для того чтобы его вызвать, мы должны разбить имя и поместить после каждого двоеточия аргумент, т.е. получить выражение наподобие следующего:

```
[someObject hereAreThreeStrings: @"string1" : @"string2" : @"string3"];
```

Это вполне корректный вызов метода, хотя и не очень распространенный, главным образом потому, что он не слишком информативный. Обычно применение имени более информативно, так что, как правило, перед каждым двоеточием идет часть имени, описывающая значение, следующее за двоеточием.

Например, вот метод класса `UIColor`, генерирующий экземпляр `UIColor` из четырех чисел `CGFloat`, представляющих красный, зеленый, синий компоненты и прозрачность, и он называется `colorWithRed:green:blue:alpha:`. Обратите внимание на ясность конструкции этого имени. Часть `colorWith` говорит о предназначении метода: он генерирует цвет на основе некоторой информации. Остальная часть имени, `Red:green:blue:alpha:`, описывает смысл каждого параметра. Вы можете вызвать этот метод таким способом:

```
UIColor* c = [UIColor colorWithRed: 0.0 green: 0.5
              blue: 0.25 alpha: 1.0];
```

В этой главе я уже ссылался несколько раз на метод класса `NSString` с именем `stringWithContentsOfFile:encoding:error:`. Это имя весьма информативно. Даже до изучения типов данных параметров мы можем предположить, что первый параметр представляет собой ссылку на файл, второй описывает кодировку, а третий используется для возврата информации об ошибке путем косвенного обращения (и что этот метод возвращает строку).

Правила именования методов Objective-C, вместе с соглашениями, регулирующими применение таких имен (наподобие сделать имя информирующим о предназначении метода и его параметров), приводят к довольно длинным и громоздким именам методов, например, таким: `getBytes:maxLength:usedLength:encoding:options:range:remainingRange:`. Такая многословность является характерной для Objective-C. Вызовы методов и даже объявления методов часто разделены на несколько строк, как для ясности, так и для предотвращения насильственного переноса редактором строки кода.

Объявление метода

Объявление метода представляет собой определяющую открытую инструкцию, указывающую имя метода, тип данных возвращаемого значения и типы данных каждого из его параметров. Например, документация Apple о классе состоит главным образом из списка объявлений его методов. Таким образом, важно уметь читать объявления методов.

Объявление метода состоит из трех частей.

- Знак + или –, означающий, что метод представляет собой метод класса или мл экземпляра соответственно.
- Тип данных возвращаемого значения в скобках.
- Имя метода, разбитое после каждого двоеточия. За каждым двоеточием следует соответствующий параметр, выраженный как тип данных параметра в скобках, за которым следует имя заполнителя для параметра.

Так, например, документация Apple гласит, что объявление метода `colorWithRed:green:blue:alpha:` класса `UIColor` имеет вид

```
+ (UIColor*) colorWithRed: (CGFloat) red  green: (CGFloat) green
                     blue: (CGFloat) blue alpha: (CGFloat) alpha
```

(Обратите внимание, что я разделил объявление на две строки, для удобочитаемости и чтобы поместить его на листе книги. В документации это объявление расположено на одной строке.)

Полезное сокращение

Не редкость, когда за пределами кода, ссылаясь на метод, перед его именем указывают знак + или –, чтобы было ясно, является ли этот метод методом класса или методом экземпляра. Полуофициальное сокращение представляет собой знак + или –, после чего идут квадратные скобки, содержащие имя класса и имя метода. Например, в примечаниях к выпуску iOS 7 сказано, что в iOS 7 добавлен следующий метод, отсутствовавший в iOS 6:

```
-[NSScanner scanUnsignedLongLong:]
```

Это не вызов метода и не объявление. Это вообще не код Objective-C. Это просто метод сокращенного описания метода, указывающий, что это метод экземпляра класса `NSScanner` с именем `scanUnsignedLongLong:`.

Убедитесь, что вы можете прочитать и понять это объявление! Вы должны уметь посмотреть на него и мгновенно сказать себе: "имя метода — `colorWithRed:green:blue:alpha:`. Это метод класса, который принимает четыре параметра типа `CGFloat` и возвращает `UIColor`".

Пробел после каждого двоеточия в объявлении или вызове метода является необязательным. Пробел перед двоеточием также является корректным, хотя на практике редко применяется. Пробел до или после любой из круглых скобок в объявлении метода также является необязательным.

Вложенные вызовы методов

Там, где в вызове метода должен находиться объект определенного типа, можно поместить еще один вызов метода, который возвращает этот тип. Таким образом, вызовы метода могут быть вложенными. Вызов метода может быть приемником в вызове метода:

```
// Глупо, но допустимо:
NSString* s = [[NSString string] uppercaseString];
```

Приведенный код корректен, поскольку результатом метода `string` класса `NSString` является экземпляр `NSString` (формально, значение `NSString*`), так что мы можем отправить метод экземпляра `NSString` этому результату. Аналогично вызов метода может находиться в другом вызове в качестве аргумента:

```
[myStack push: [NSString string]]; // Все в порядке, если метод
                        // push: ожидает параметр типа NSString*
```

Тем не менее я должен предостеречь вас от злоупотребления этой возможностью. Код с большим количеством вложенных квадратных скобок очень трудно читать (и писать). Кроме того, если один из вложенных вызовов возвращает непредвиденное значение, нет способа легко обнаружить этот факт. Зачастую куда лучше писать более подробный код и объявлять временные переменные для хранения результатов каждого вызова метода. Вот пример из моего собственного кода. Вместо чтобы написать

```
NSArray* arr = [[MPMediaQuery albumsQuery] collections];
```

я воспользовался временной переменной:

```
MPMediaQuery* query = [MPMediaQuery albumsQuery];
NSArray* arr = [query collections];
```

Несмотря на то что первая версия короче и достаточно ясна и что во втором варианте переменная `query` никогда не будет использоваться еще раз (она существует исключительно для того, чтобы быть приемником сообщения `collection` во второй строке), ее все же стоило создать. Как минимум, это упрощает пошаговую отладку кода в отладчике, когда я хочу сделать паузу после вызова `albumsQuery` и посмотреть, правильный ли результат возвращается этим вызовом (см. главу 9).

Неверное количество или несоответствие вложенных квадратных скобок может привести к появлению несколько запутанных сообщений от компилятора. Например, слишком большое количество пар квадратных скобок (`[[query collections]]`) или несбалансированное количество левых скобок (`[[query collections]`) влекут за собой сообщение "Expected identifier" ("Ожидается идентификатор").

Отсутствие перегрузки

Тип данных, возвращаемых методом вместе с типами данных каждого из его параметров в порядке передачи этих параметров методу, составляют *сигнатуру* этого метода. Существование двух методов одного и того же типа (метода класса или метода экземпляра) в одном и том же классе с одним и тем же именем недопустимо, даже если они имеют разные сигнатуры.

Так, например, вы не может иметь два метода экземпляра класса `MyClass` с одним именем `myMethod`, один из которых возвращает `void`, а второй — `NSString`. Аналогично нельзя иметь два метода экземпляра класса `MyClass` с именем `myMethod:`, оба возвращающие `void`, но один из которых принимает параметр `CGFloat`, а второй — `NSString`. Попытки нарушить это правило будут заблокированы компилятором, который объявит об ошибке "duplicate declaration" ("повторное объявление"). Причиной появления этого правила является то, что, если бы два таких метода могли существовать, не имелось бы способа определить при получении сообщения, какой именно из этих методов должен быть вызван.

Вы можете подумать, что этот вопрос можно решить на основе типов параметров, участвующих в вызове. Если один `myMethod:` принимает параметр `CGFloat`, а другой — `NSString`, то, когда вызывается `myMethod:`, Objective-C мог бы взглянуть на фактический аргумент и понять, какой из методов должен быть вызван. Но Objective-C таким образом не работает. Да, есть языки программирования, которые обеспечивают такую возможность, которая называется перегрузкой, но Objective-C не является одним из них.

Списки параметров

В языке Objective-C не такая уж редкость методы, требующие неизвестное количество параметров. Хорошим примером является метод `arrayWithObjects:` класса `NSArray`, который выглядит, как будто он принимает один параметр, но на самом деле он принимает любое количество параметров, разделенных запятыми. Параметры являются объектами, которые представляют собой элементы `NSArray`. Хитрость здесь заключается в том, что список должен заканчиваться элементом `nil`. Это значение не является ни одним из объектов, помещаемых в `NSArray` (`nil` не является объектом, поэтому `NSArray` не может его содержать), и предназначено только для того, чтобы указать, где заканчивается список.

Таким образом, вполне корректен следующий вызов этого метода:

```
NSArray* pep = [NSArray arrayWithObjects:@"Manny",
                              @"Moe", @"Jack", nil];
```

В объявлении метода `arrayWithObjects:` используется многоточие, указывающее на то, что здесь может находиться список параметров, разделенных запятыми:

```
+ (id)arrayWithObjects:(id)firstObj, ... ;
```

(Позже в этой главе я объясню смысл встречающегося в объявлении `id`.) Без ограничителя `nil` программа будет не в состоянии определить, где завершается список, и при выполнении программы могут случиться разные неприятности — ведь она заберется в поисках новых параметров туда, где их нет, и запросто заполнит `NSArray` всяческим мусором.

Распространенная ошибка начинающего программиста — забыть об ограничителе `nil`, но современный компилятор Objective-C достаточно умен, чтобы выдать предупреждение об отсутствии ограничителя — "missing sentinel in method dispatch". Несмотря на то что это всего лишь предупреждение, не запускайте такой код на выполнение! Еще один способ не забыть об ограничителе `nil` в этом конкретном примере — избежать вызова `arrayWithObjects:` вообще, что возможно начиная с версии компилятора LLVM 4.0 (Xcode 4.4 или более поздний), где вы можете образовать литеральный объект `NSArray` непосредственно, с помощью синтаксиса `@[...]`:

```
NSArray* pep = @[@"Manny", @"Moe", @"Jack"];
```

Это просто упрощенная запись, которая "за кулисами" разворачивается в вызов `arrayWithObjects:`, но этот вызов корректно оформлен, с ограничителем `nil`, так что шансов не допустить ошибку у программиста существенно больше, чем при явном вызове `arrayWithObjects:` (да и количество вводимых символов при этом куда меньше).

Тем не менее вы будете встречаться со многими другими методами Objective-C, которые принимают параметр, являющийся списком переменной длины с завершающим `nil`. Например, у протокола `UIAppearance` имеется метод класса `appearanceWhenContainedIn:` или метод `initWithTitle:message:delegate:cancelButtonTitle:otherButtonTitles:` в `UIAlertView`. Жаль, что Apple не изменит каким-то образом Objective-C (или эти методы), чтобы избежать использования ограничителя `nil`; но до тех пор, пока это не сделано, следует знать, как правильно вызываются такие методы.

Язык C явным образом обеспечивает функционирование списков аргументов неопределенной длины, чем методы Objective-C наподобие `arrayWithObjects:` пользуются "за сценой". Я не собираюсь пояснять работу механизма языка программирования C, поскольку не думаю, что вам придется писать метод или функцию, для которых он потребуется. Но если вам нужны соответствующие подробности — обратитесь к K&R 7.3.

Когда отправка сообщений не работает

Механизм отправки сообщений имеет фундаментальное значение для Objective-C. По этой же причине это основной источник проблем. Слишком легко может случиться, что, когда сообщение отправляется объекту, все пойдет не так, как следует. Но как что-то может пойти не так в таком простом и понятном механизме? Что ж, рассмотрим следующее, казалось бы, тривиальное, выражение:

```
[s uppercaseString]
```

В этом выражении s представляет собой ссылку, а uppercaseString — сообщение. Сообщение uppercaseString отправляется ссылке s. Что может пойти не так?

Ссылка может указывать на неверный объект

Ссылка на объект является указателем. Предполагается, что она указывает на реальный объект. Ссылка — это всего лишь имя. Как из выражения узнать, на что именно указывает s? Вы не видели, каким значением эта ссылка инициализирована. Она может быть равна nil. Это может не быть строка. Это может быть строка, но не та, которую вы ожидаете увидеть. Ссылка может быть ссылкой вовсе не на то, на что вы думаете.

Сообщение может быть неверным

Эта возможность идет рука об руку с предыдущей. Если ссылка может быть неверной, то объект, на который она указывает, может не принимать отправленное сообщение. Или объект может принять сообщение, но, будучи не тем объектом, что надо, он может дать неверный результат, отличающийся от того, который вы ожидаете.

Это не просто придирки; это распространенная повсеместная реальность. Подавляющее количество вопросов, с которыми я сталкиваюсь в Интернете, о том, почему некоторый код неверно работает, и подавляющее количество ответов на эти вопросы — из-за указанных проблем. Программирование обманчиво; вы можете легко обмануться в своих предположениях. Вы думаете, что знаете, что собой представляют некоторые ссылки, но на самом деле это вовсе не так… В результате выясняется, что все ведет себя не так, как надо, или программа попросту аварийно завершает работу. Понимание, как работает отправка сообщений и что при этом может пойти не так — а я заверяю вас, что это часто так и будет, — это залог успешного написания программы.

В оставшейся части этого раздела мы сосредоточимся на двух особенно коварных причинах, по которым отправка сообщения может пойти не так, как надо: сообщение направляется по ссылке nil и отправляется объекту, которому оно не нравится.

Сообщения для nil

Ссылке на экземпляр очень легко оказаться указывающей на неэкземпляр — то есть на nil. Я уже говорил, что простое объявление экземпляра без инициализации устанавливает ссылку равной nil (при наличии механизма ARC) и что многие методы каркаса Cocoa преднамеренно возвращают nil как способ указать, что что-то пошло не так. Кроме того, экземпляр переменной, объявленной как ссылка на объект, начинает жизнь как nil, и если эта ссылка не будет установлена впоследствии указывающей на реальный объект, как вы ожидаете, то она может так и остаться nil. (В главе 7", мы рассмотрим все наиболее распространенные способы, которыми это может случиться.)

Рассмотрим последствия отправки сообщения неэкземпляру, т.е. по указателю nil. Вы можете отправить сообщение экземпляру NSString следующим образом:

```
NSString * s2 = [s uppercaseString];
```

Этот код посылает сообщение uppercaseString объекту s. Предположительно s представляет собой экземпляр NSString. Но что если s равен nil? В некоторых объектно-ориентированных языках программирования отправка сообщения ссылке nil вызывает ошибку времени выполнения и приводит к преждевременному завершению работы программы (например, в REALbasic и Ruby). Но Objective-C работает иначе. В Objective-C отправка сообщения nil допустима и не прерывает выполнение программы. Более того, если вы сохраните

результат вызова метода, он будет разновидностью нуля — что означает, что если вы присвоите этот результат ссылке на экземпляр, то он получит значение `nil`:

```
NSString* s = nil;                  // Сейчас s равно nil
NSString* s2 = [s uppercaseString]; // Теперь s2 тоже равно nil
```

Вопрос о том, является ли это поведение Objective-C хорошим решением или нет, вызывает целые религиозные войны и является предметом бурных споров среди программистов. С одной стороны, это полезно, с другой — при этом очень легко обмануться. Обычный сценарий — когда вы случайно отправляете сообщение ссылке `nil`, не осознавая этого, и позже ваша программа работает не так, как ожидалось. А поскольку точка, в которой обнаруживается некорректное поведение, находится позже (может быть, даже существенно позже) момента первого появления нулевого указателя, это появление может быть очень трудно отследить (на самом деле программисты часто не считают нулевой указатель первоочередной причиной своих неприятностей).

Имеется не так много вариантов ваших действий — по сути, вы можете только наполнить ваш код сплошными проверками на равенство `nil` используемых ссылок на экземпляры. Описанное поведение в случае сообщения `nil` встроено в язык, и его никто не собирается менять. Поэтому вы просто обязаны знать его и постоянно о нем помнить! Считать, что у вас есть ссылка на реальный экземпляр, в то время как на самом деле у вас ссылка "в никуда" — это очень, очень распространенная ошибка начинающих, отягощенная тем, что при использовании ссылки `nil` никаких жалоб от среды выполнения не поступает, а ссылки `nil` в результате размножаются едва ли не в геометрической прогрессии. При этом ничего страшного не происходит, программа как-то работает, и некому разуверить вас в этом заблуждении (за исключением того, что программа ведет себя совершенно не так, как ожидалось). Если все тихо и таинственно идет не так, в первую очередь следует заподозрить наличие нулевой ссылки и использовать для расследования методы отладки (глава 9).

Словом, если вы заранее знаете, что вызов некоторого метода может вернуть `nil`, не думайте, что все пойдет хорошо и что он не вернет этот `nil`. Наоборот, если что-то может пойти не так — оно пойдет не так с большой вероятностью! Например, опустить проверку на равенство `nil` после вызова `stringWithContentsOfFile:encoding:error:` — это просто тупость. Меня не интересует, что вы точно знаете, что нужный файл имеется и что он именно в той кодировке, которую вы указали, — вы просто **обязаны** проверить результат на равенство `nil`!

Нераспознанные селекторы

Возможна ситуация с отправкой сообщения объекту, у которого нет соответствующего метода. Если такое случится, последствия могут быть фатальными: ваше приложение может просто аварийно завершиться. Единственный защитник от таких непредвиденных обстоятельств — компилятор; но это не слишком сильный защитник. В некоторых случаях компилятор предупредит вас, что, возможно, вы делаете что-то нежелательное; но в других случаях компилятор с радостью позволит вам наступить на эти грабли.

Вот ситуация, в которой компилятор действительно спасет вас от самого себя:

```
NSString* s = @"Hello, world!";
[s rockTheCasbah];
```

Класс `NSString` не имеет метода `rockTheCasbah`. До появления механизма ARC компилятор разрешил бы компиляцию этого кода, но предупредил бы о том, что вы сами ищете себе проблемы. Однако при наличии механизма ARC компилятор с негодованием отвергает вашу попытку с сообщением о фатальной ошибке "No visible `@interface` for '`NSString`'

declares the selector 'rockTheCasbah'". (Компилятор с механизмом ARC гораздо строже, так как механизм ARC должен работать с управлением памятью, а при разрешении отправлять бессмысленные сообщения это не удается делать эффективно.)

Немного замутив воду, мы можем проскользнуть мимо строгого механизма ARC. Предположим, что у нас есть класс `MyClass`, в котором объявлен метод `rockTheCasbah`. Тогда мы можем записать

```
MyClass* m = @"Hello, world!";
[m rockTheCasbah];
```

Мы говорим, что `m` — экземпляр класса `MyClass`, но на деле присваиваем этой ссылке экземпляр `NSString`. Это странное, но не строго запрещенное действие. Компилятор предупредит нас о неполной корректности этого присваивания ("incompatible pointer types"), но программу скомпилирует. Остается ехидно засмеяться и запустить программу на выполнение. И что получится?

Когда программа заработает и класс `NSString` получит сообщение `rockTheCasbah`, программа аварийно завершится с сообщением (в журнале консоли) о случившейся неприятности в виде

```
-[__NSCFConstantString rockTheCasbah]: unrecognized
    selector sent to instance 0x8650.
```

Это сообщение детально описывает происшедшее.

- Фраза о *нераспознанном селекторе* (unrecognized selector) является самой сутью происшедшего. Термин "селектор" грубо эквивалентен термину "сообщение", так что это способ сказать, что определенному объекту послано сообщение, с которым он не в состоянии справиться.
- `-[__NSCFConstantString rockTheCasbah]` описывает сообщение и его получателя, используя сокращение, о котором я говорил ранее: экземпляру `NSCFConstantString` послано сообщение экземпляра `rockTheCasbah`. (Описание `NSString` как `NSCFConstantString` представляет собой внутренние подробности языка.)
- `0x8650` — значение указателя на экземпляр; это адрес в памяти, на который в действительности указывает указатель `m`.

Ситуация нераспознанного селектора приводит к генерации *исключения*, внутреннего сообщения о том, что при работе программы произошло что-то плохое. Код на Objective-C может "поймать" исключение, и тогда аварийное завершение не произойдет. Технически причина завершения работы программы не в том, что объекту отправлено сообщение, с которым тот не смог справиться, а в том, что не было перехвачено сгенерированное при этом исключение. Вот почему журнал сбоев может также гласить "Terminating app due to uncaught exception" (завершение приложения из-за неперехваченного исключения).

В этом примере мы сознательно создали неприятности сами себе, чтобы продемонстрировать, что происходит, когда возникает ситуация нераспознанного селектора. Пример, конечно, достаточно надуманный, но последствия таковыми не являются. Я заверяю, что такая вещь рано или поздно *случится* и с вами, хотя, конечно, не преднамеренно. Это произойдет потому, что, как я сказал в начале этого раздела, ваша ссылка окажется не совсем тем, что вы о ней думаете. Существует множество путей, которыми может возникнуть такая ситуация;

например, как я покажу в следующем разделе, вы можете случайно соврать компилятору о классе объекта, или компилятор может не иметь никакой информации о классе объекта, так что он даже не сможет предупреждать вас о сообщении, которое объект не в состоянии обработать. Следите за фразой "unrecognized selector", когда ваша программа аварийно завершает работу, — теперь вы знаете, что это значит и как интерпретировать такое сообщение! Детали сообщения помогут вам понять, что именно не так и как исправить ситуацию.

Приведение типа и тип `id`

Иногда компилятор, пытаясь спасти вас от ситуации нераспознанного селектора, выдает предупреждение или сообщение об ошибке, в то время как вы очень хорошо знаете, что то, что вы пытаетесь сделать, — безопасно и правильно. Проблема в данном случае заключается в том, что вы знаете больше компилятора о том, что в действительности происходит. Чтобы продолжить работу, вам нужно поделиться своими знаниями с компилятором. Как правило, это делается с помощью *приведения типа* ссылки на объект (см. главу 1). Такое приведение служит в качестве объявления класса, которому объект будет принадлежать при выполнении программы. Компилятор верит в то, что вы говорите ему в выражении приведения; таким образом, вы можете развеять сомнения компилятора в корректности выполняемых вами действий.

Эта ситуация часто возникает в связи с наследованием классов. Пока что мы не будем обсуждать наследование (см. главу 4), но пример я все равно приведу.

Возьмем встроенный класс Cocoa `UINavigationController`. Его метод `topViewController` объявлен как возвращающий экземпляр `UIViewController`. В реальной же жизни он скорее вернет экземпляр некоторого созданного вами подкласса `UIViewController`. Для того чтобы вызов метода созданного вами класса для экземпляра, возвращенного методом `topViewController`, не свел с ума компилятор, последнему следует пояснить, что этот экземпляр на самом деле будет экземпляром созданного вами класса.

Вот пример из одного из моих собственных приложений:

```
[[navigationController topViewController] setAlbums: arr];
```

Эта строка кода не компилируется; компилятор выдает ту же ошибку "no visible `@interface`", о которой я говорил в предыдущем разделе. Встроенный метод `topViewController` возвращает `UIViewController`, а `UIViewController` не имеет метода `setAlbums`.

Однако я знаю, что в данном случае контроллер верхней панели контроллера навигации является экземпляром моего собственного класса `RootViewController`. А мой класс `RootViewController` *имеет* метод `setAlbums`; и именно его я и пытаюсь вызвать. То, что я делаю — разумно, законно и желательно. Мне надо действовать именно таким образом! Чтобы компилятор не мешал мне работать, я должен убедить его, что знаю, что делаю, сообщив, что объект, возвращаемый вызовом метода `topViewController`, на самом деле представляет собой `RootViewController`. Я делаю это с помощью приведения типов:

```
[(RootViewController*)[navigationController topViewController]
        setAlbums: arr];
```

Этого достаточно. Не будет никаких предупреждений и сообщений об ошибках. Приведение типов заставляет компилятор замолчать, когда я собираюсь отправить этому экземпляру сообщение `setAlbums:`, поскольку мой класс `RootViewController` имеет метод экземпляра `setAlbums:`, и компилятор знает об этом. А программа при запуске не завершается аварийно, поскольку я не лгу компилятору: вызов метода `topViewController` действительно возвращает экземпляр `RootViewController`.

Но чем больше власть — тем больше и ответственность. Не лгите компилятору! Вспомните пример из предыдущего раздела:

```
MyClass* m = @"Hello, world!";
[m rockTheCasbah];
```

Первая строка заставляет компилятор предупредить нас о "incompatible pointer types" (несовместимых типах указателей); и в данном случае компилятор совершенно прав: ведь в дальнейшем мы получим аварийное завершение ("unrecognized selector") при попытке отправить сообщение `rockTheCasbah` объекту `NSString`. Мы можем заставить компилятор промолчать с помощью приведения типов:

```
MyClass* m = (MyClass*)@"Hello, world!";
[m rockTheCasbah];
```

Да, предупреждения не будет. Но проблема останется: ведь мы солгали компилятору и будем отправлять сообщение `rockTheCasbah`, как и ранее, экземпляру `NSString`. Мораль проста: поскольку компилятор верит всему, что вы говорите ему с помощью приведения типов, врать ему категорически нельзя.

Приведение типов не меняет чудесным образом реальные типы объектов. Это просто метод подсказки компилятору об информации о типе. На сам приводимый объект это никак не влияет. Выражение приведения `(MyClass*)@"Hello, world!"` не превращает экземпляр `NSString`, который представляет собой `@"Hello, world!"`, в экземпляр `MyClass`! На удивление распространенная ошибка среди новичков — считать, что при этом действительно выполняются какие-то преобразования.

Язык Objective-C предоставляет также специальный тип, предназначенный для того, чтобы все заботы компилятора о типах данных выполнялись в тишине. Это тип `id`. Он представляет собой указатель, так что вы не должны использовать выражение `id*`. Этот тип определен как "указатель на просто объект", без каких бы то ни было дальнейших уточнений. Таким образом, каждая ссылка на экземпляр также представляет собой `id`.

Использование типа `id` заставляет компилятор перестать беспокоиться о взаимосвязи между типами объектов и сообщениями. Компилятор не может ничего знать о том, какого типа в действительности будет тот или иной объект, поэтому он просто поднимает (или опускает — как кому нравится) руки и не предупреждает больше ни о чем. Кроме того, объекту типа `id` может быть присвоено любое объектное значение, и к типу `id` может быть приведен любой другой тип. Значение типа `id` может быть использовано в любом присваивании там, где ожидается значение некоторого конкретного объектного типа. Понятие назначения включает в себя передачу параметров; таким образом, значение типа `id` можно передать в качестве аргумента везде, где ожидается параметр некоторого конкретного объектного типа. (Мне нравится думать об `id` как об аналоге групп крови AB и O: это универсальный реципиент и универсальный донор.) Вот пример применения типа `id`:

```
NSString* s = @"Hello, world!";
id unknown = s;
[unknown rockTheCasbah];
```

Вторая строка корректна, поскольку любое объектное значение может быть присвоено переменной типа `id`. Третья строка не генерирует предупреждения компилятора, поскольку типу `id` может быть послано любое сообщение. (Очевидно, что программа при запуске все равно завершится аварийно, когда выяснится, что `unknown` имеет тип `NSString` — который, конечно же, не может получать сообщения `rockTheCasbah`!)

На самом деле это чрезмерное упрощение. При использовании механизма ARC этот код не может быть скомпилирован. Вместо скомпилированного кода вы получите сообщение об ошибке: "No known instance method for selector 'rockTheCasbah'". Оно означает, что компилятор ничего не знает о методе `rockTheCasbah` ни в каком классе. Однако если `rockTheCasbah` объявляется в заголовочном файле любого класса, импортированном в данный, или если `rockTheCasbah` реализован в текущем классе, то код будет компилироваться без предупреждения.

Если способность типа `id` получать любые сообщения напоминает вам `nil`, то так и должно быть. Я уже говорил, что `nil` представляет собой разновидность нуля; теперь я могу сказать, какой разновидностью нуля является `nil`. Это нуль, приведенный к типу `id`. Конечно, во время выполнения имеется разница, равен ли `id` значению `nil` или некоторому иному; отправка сообщения `nil` не вызовет аварийного завершения программы, но отправка неизвестного сообщения реальному объекту, вероятно, приведет к таковому.

Таким образом, результат применения `id` выражается в полном отключении проверки типов компилятором. Выяснения, каким является тип объекта, откладываются до тех пор, пока программа не будет запущена на выполнение. Но я не рекомендую широко применять `id` таким образом. Компилятор — ваш друг; вы должны позволить ему разведывать и перехватывать возможные ошибки в коде. Поэтому я почти никогда не объявляю переменную или параметр как имеющие тип `id`. Я хочу, чтобы типы моих объектов были максимально конкретными, чтобы компилятор мог максимально проверить мой код. С другой стороны, тип `id` часто используется в интерфейсе API каркаса Cocoa — и это именно то, что, как я предупреждал вас в предыдущем разделе, может приводить к аварийному завершению из-за нераспознанного селектора.

Рассмотрим, например, класс `NSArray`, который представляет собой объектно-ориентированную версию массива. В чистом C вы должны объявить, какого типа объекты находятся в массиве; например, у вас может быть "массив целых чисел `int`". В Objective-C, используя `NSArray`, вы не можете этого сделать. Каждый `NSArray` представляет собой массив элементов типа `id`, что означает, что каждый элемент массива может быть любого объектного типа. Вы можете поместить в массив `NSArray` объект некоторого типа, поскольку любой тип объекта может быть присвоен ссылке на `id` (`id` является универсальным реципиентом). Вы можете получить обратно из `NSArray` любой конкретный тип объекта, так как `id` может быть присвоен объекту любого типа (`id` является универсальным донором).

Метод `lastObject` класса `NSArray`, таким образом, определен как возвращающий тип `id`. Для данного массива `arr` типа `NSArray` последний элемент можно получить следующим образом:

```
id unknown = [arr lastObject];
```

Теперь мы находимся в потенциально опасной ситуации. После этого кода объекту `unknown` можно послать любое сообщение; компилятор не будет этому препятствовать. Следовательно, если мне известен тип элементов массива, я всегда могу выполнить присваивание или приведение типа к типу, который я получаю из массива. Пусть, например, я знаю, что массив `arr` не содержит ничего, кроме экземпляров `NSString` (поскольку я перед этим поместил их туда). Тогда я могу написать

```
NSString* s = [arr lastObject];
```

Компилятор не будет жаловаться на этот код, поскольку тип `id` может быть присвоен любому конкретному типу объекта (`id` является универсальным донором). Кроме того, далее компилятор рассматривает `s` как `NSString` и использует свои возможности проверки типов для того, чтобы убедиться, что я не посылаю `s` никаких сообщений, кроме тех, что можно

посылать NSString. И я не лгу компилятору; во время выполнения s действительно представляет собой объект NSString, так что все в порядке.

Однако есть еще одна опасность: эта ситуация — открытое приглашение случайно солгать компилятору. Предположим, что последний элемент в этом массиве NSArray не является NSString. Компилятор этого не знает; он вообще ничего не знает о том, что NSArray на самом деле будет содержать во время выполнения, а lastObject возвращает id, который компилятор с радостью позволит присвоить ссылке, объявленной как NSString. И тут мы просто напрашиваемся на неприятности. Предположим, например, что в следующей строке я отправлю сообщение uppercaseString объекту s. Компилятор молчит: в конце концов, я объявил эту ссылку как NSString, а uppercaseString является методом NSString. Но если s *не* является NSString, то, вероятно, мы попадем в ситуацию нераспознанного селектора и получим аварийное завершение программы.

Еще одной связанной с id ловушкой являются конфликты имен методов. Ранее я говорил, что один и тот же класс не может определять методы одного и того же типа (методы класса или методы экземпляра) с одинаковыми именами, но различными сигнатурами. Но я не говорил, что происходит, когда два *разных* класса объявляют методы с одинаковыми именами, но различными сигнатурами. Если в коде указан тип объекта-получателя сообщения, никаких проблем нет, потому что нет никаких сомнений в том, какой метод вызывается: он один в классе этого объекта. Но если объектом-получателем сообщения является id, то с использованием механизма ARC вы получите сообщение об ошибке: "Multiple methods named 'rockTheCasbah' found with mismatched result, parameter type or attributes" (найдено несколько методов с именем rockTheCasbah с несоответствующими типами результата, параметров или атрибутами). Это еще одна причина, по которой имена методов так многословны: для того, чтобы сделать каждое имя метода уникальным, предотвращая объявление в разных классах конфликтующих сигнатур для одного и того же имени метода.

Сообщения как тип данных

Предыдущие разделы вращалась вокруг того факта, что Objective-C до времени выполнения не знает, какому объекту посылается сообщение. Но справедливо большее: Objective-C до времени выполнения не должен знать, *какое сообщение* отправляется объекту. Некоторые важные методы в действительности принимают в качестве параметров и сообщение, и его получателя; они не собраны и использованы для формирования выражения фактическое сообщение до времени выполнения.

Рассмотрим, например, такое объявление метода из класса NSNotificationCenter Cocoa:

```
- (void)addObserver:(id)notificationObserver
           selector:(SEL)notificationSelector
               name:(NSString *)notificationName
             object:(id)notificationSender
```

Позже (в главе 11) я поясню, что делает этот метод. Сейчас для понимания важно то, что он составляет команду для отправки некоторого сообщения определенному объекту позже, в подходящее время. Например, наша цель при вызове этого метода может заключаться в том, чтобы иметь сообщение tickleMeElmo: для отправки позже объекту myObject. Для этого нам надо передать соответствующие данные в качестве двух первых аргументов.

Давайте рассмотрим, какие аргументы следует передать. Первый параметр (observer:) представляет собой объект, которому будет отправлено сообщение и который имеет тип id, что позволяет указать в качестве получателя объект любого типа. Так что в нашем случае в

качестве аргумента observer: мы передаем myObject. Само сообщение представляет собой параметр selector:, который имеет специальный тип данных SEL. Теперь вопрос заключается в том, как выразить имя сообщения tickleMeElmo:.

Вы не можете просто набрать tickleMeElmo: как обычный текст: это не сработает синтаксически. Вы можете подумать, что сообщение можно выразить как строку NSString, т.е. как @"tickleMeElmo:", но это тоже не сработает. Оказывается, корректно следует поступить следующим образом:

```
@selector(tickleMeElmo:)
```

Текст @selector() представляет собой директиву компилятору, которая говорит о том, что то, что находится в круглых скобках, представляет собой имя сообщения. Обратите внимание, что находящееся в скобках не является NSString; это просто имя сообщения. А поскольку это имя, оно не должно включать пробелы и должно включать в качестве своей части двоеточия.

Так что правило исключительно простое: там, где ожидается SEL, вы обычно передаете выражение @selector. Ошибка в этом синтаксисе достаточно распространена среди начинающих программистов.

К сожалению, этот синтаксис — также прямое приглашение наделать ошибок при вводе, возможно, с очень неприятными результатами. Если myObject реализует метод tickleMeElmo:, а я случайно наберу текст, скажем, @selector(tickleMeElmo), забыв о двоеточии, то даже если сообщение tickleMeElmo без двоеточия будет отправлено объекту myObject, приложение, вероятно, все равно завершится аварийно из-за исключения нераспознанного селектора.

В Xcode 5, впервые в истории Objective-C, введено предупреждение компилятора в случае, когда селектор не соответствует известному методу ("Undeclared selector 'tickleMeElmo'"). Таким образом, вы можете быть предупреждены о возможности будущего аварийного завершения — или нет. Компилятор не может заглянуть в класс аргумента observer: и выяснить, реализован ли в нем данный метод. Вместо этого компилятор предупреждает только в ситуации, если класса с данным методом вообще нет. Справедливо и обратное: если компилятор знает о наличии метода tickleMeElmo в каком-то классе, то никакого предупреждения вы не получите, даже если объект, которому в действительности во время работы программы будет послано сообщение tickleMeElmo, такого метода не имеет.

Функции C

Хотя ваш код, несомненно, будет вызывать массу методов Objective-C, вероятно, он также будет вызывать и некоторые функции C. Например, я уже упоминал в главе 1, что обычный способ описания CGPoint, основанный на его значениях x и y, заключается в вызове функции CGPointMake, которая объявлена следующим образом:

```
CGPoint CGPointMake (
    CGFloat x,
    CGFloat y
);
```

Убедитесь, что вы с первого взгляда видите, что это функция C, а не метод Objective-C и что вы понимаете разницу в синтаксисе вызова. Для вызова метода Objective-C вы отправляете в квадратных скобках сообщение объекту, с аргументами после двоеточий в имени метода; для вызова функции C используется имя функции, за которым следуют круглые скобки, содержащие аргументы.

У вас даже могут иметься причины для написания вместо методов собственных функций C как части класса. Функции C имеют более низкие накладные расходы, чем полноценные методы; так что, даже несмотря на отсутствие объектно-ориентированных возможностей методов, иногда полезно писать такие функции, в частности, когда некоторые вспомогательные вычисления должны выполняться быстро и часто.

Кроме того, вы можете встретиться с методами или функциями Cocoa, которые требуют передачи им функции C как "функции обратного вызова". Примером является метод `sortedArrayUsingFunction:context:` класса `NSArray`. Его первый параметр имеет тип

```
NSInteger (*) (id, id, void *)
```

Это выражение с помощью довольно сложного синтаксиса C описывает указатель на функцию, которая принимает три параметра и возвращает `NSInteger`. Три параметра функции имеют типы `id`, `id` и указатель на `void` (что означает любой указатель C). В языке программирования C как указатель функции может использоваться ее имя (см. главу 1). Таким образом, чтобы вызвать `sortedArrayUsingFunction:context:`, вам нужно написать функцию C, которая соответствует приведенному описанию, и использовать ее имя в качестве первого аргумента метода.

Для иллюстрации я напишу функцию обратного вызова, которая поможет мне отсортировать массив `NSArray` строк `NSString` по последнему символу каждой строки. (Да, это странное решение, но это ведь всего лишь иллюстрация!) Значение `NSInteger`, возвращаемое функцией, имеет особое значение: оно указывает, меньше ли первый параметр второго, равен ему или больше. Это значение я буду получать с помощью метода `compare:` класса `NSString`, который возвращает `NSInteger` с тем же смыслом. В примере 3.1 определена функция и показано, как должен вызываться метод `sortedArrayUsingFunction:context:` с нашей функцией как функцией обратного вызова.

Пример 3.1. Конструкции управления потоком языка программирования C

```
NSInteger sortByLastCharacter(id string1, id string2,
                              void* context) {
    NSString* s1 = string1;
    NSString* s2 = string2;
    NSString* string1end =
            [s1 substringFromIndex:[s1 length] - 1];
    NSString* string2end =
            [s2 substringFromIndex:[s2 length] - 1];
    return [string1end compare:string2end];
}
// А вот как используется эта функция (предполагается,
// что arr представляет собой массив NSArray строк NSString)
NSArray* arr2 = [arr
        sortedArrayUsingFunction:sortByLastCharacter
        context:nil];
```

CFTypeRef

Многие объектные типы Objective-C имеют низкоуровневые аналоги C, вместе с функциями C для работы с ними.

Например, помимо `NSString`, в Objective-C имеется нечто, именуемое `CFString`; "CF" означает "Core Foundation" и представляет собой низкоуровневый интерфейс API на основе C. `CFString` является "непрозрачной" структурой C; "непрозрачная" означает, что элементы, составляющие эту структуру, держатся в секрете и что вы должны работать с `CFString`

только с помощью соответствующих функций C. Как и в случае NSString или любого другого объекта, в коде вы обычно обращаетесь к CFString через указатель C; указатель на CFString имеет имя типа CFStringRef.

При случае можно решить работать с таким типом даже тогда, когда существует соответствующий объектный тип. Например, может выясниться, что NSString, при всей его мощи, не обеспечивает некоторую необходимую часть функциональности, которая доступна для CFString. К счастью, NSString (значение, типизированное как NSString*) и CFString (значение CFStringRef) являются взаимозаменяемыми: вы можете использовать один из типов там, где ожидается другой, хотя, чтобы успокоить компилятор, может потребоваться приведение типов. Эта взаимозаменяемость описана и в документации.

Для иллюстрации я использую CFString для преобразования строки NSString, представляющей целое значение, в целое число (для этого не так уж необходимо применять CFString, и я делаю это только в иллюстративных целях; тип NSString имеет решающий эту задачу метод intValue):

```
NSString* answer = @"42";
int ans = CFStringGetIntValue((CFStringRef)answer);
```

Объектные типы C, представляющие собой указатели на структуры, обычно имеют имена, заканчивающиеся на "Ref", и которые можно в целом именовать как CFTypeRef, в действительности представляют собой обобщенный указатель на void. Таким образом, можно использовать упомянутую выше взаимозаменяемость типов как приведение типов между объектными и обобщенными указателями, т.е. от id к void* или от void* к id. Даже там, где взаимозаменяемости между *конкретными* типами (как между типами NSString и CFString) нет, всегда можно обеспечить взаимозаменяемость с верхним уровнем иерархии, т.е. между id или NSObject (базовый класс объекта, см. главу 4) и CFTypeRef.

Блоки

Блок представляет собой расширение языка C, введенное в OS X 10.6 и доступное начиная с iOS 4.0. Это способ связывания некоторого кода в единую сущность и передачи его как аргумента функции C или методу Objective-C. Это похоже на то, что мы делали в примере 3.1, передавая в качестве аргумента указатель на функцию; однако теперь мы будем передавать *сам код*. Этот способ имеет некоторые важные преимущества перед передачей указателя, о чем мы поговорим чуть позже.

Для иллюстрации я перепишу пример 3.1 с использованием блока вместо указателя на функцию. Вместо вызова метода sortedArrayUsingFunction:context: я вызываю метод sortedArrayUsingComparator:, который получает блок в качестве параметра. Этот блок типизирован как

```
NSComparisonResult (^)(id obj1, id obj2)
```

Это выражение аналогично синтаксису определения типа указателя на функцию, но с символом ^ вместо *. Здесь описывается блок, который получает два параметра типа id и возвращает NSComparisonResult (который представляет собой просто NSInteger, с тем же смыслом, что и в примере 3.1). Мы можем определить блок прямо в качестве аргумента в вызове sortedArrayUsingComparator:, как в примере 3.2.

Пример 3.2. Использование встроенного блока вместо функции

```
NSArray* arr2 = [arr sortedArrayUsingComparator: ^(id obj1, id obj2) {
    NSString* s1 = obj1;
```

```
    NSString* s2 = obj2;
    NSString* string1end = [s1 substringFromIndex:[s1 length] - 1];
    NSString* string2end = [s2 substringFromIndex:[s2 length] - 1];
    return [string1end compare:string2end];
}];
```

Синтаксис определения встроенного блока имеет вид

```
^❶(id obj1, id obj2)❷ {❸
    //...
}
```

❶ Символ ^

❷ Скобки с параметрами, аналогично параметрам в определении функции C.

❸ Наконец, содержимое блока в фигурных скобках. Фигурные скобки образуют область видимости.

Возвращаемый тип в определении встраиваемого блока обычно опускается. Если он включен, то находится перед символом ^, а не в круглых скобках. Если же он опущен, то вы можете использовать выражение приведения типа в строке `return`, чтобы возвращаемый тип соответствовал ожидаемому.

Встроенный блок, как показанный в примере 3.2, не может использоваться повторно; если бы у нас было два вызова `sortedArrayUsingComparator:`, то мы должны были писать этот блок в полном объеме дважды. Чтобы избежать такого повторения, или просто для ясности, блок может быть присвоен переменной, которая затем передается методу как аргумент (пример 3.3).

Пример 3.3. Присваивание блока переменной

```
NSComparisonResult (^sortByLastCharacter)(id, id) = ^(id obj1, id obj2) {
    NSString* s1 = obj1;
    NSString* s2 = obj2;
    NSString* string1end = [s1 substringFromIndex:[s1 length] - 1];
    NSString* string2end = [s2 substringFromIndex:[s2 length] - 1];
    return [string1end compare:string2end];
};
NSArray* arr2 = [arr sortedArrayUsingComparator: sortByLastCharacter];
NSArray* arr4 = [arr3 sortedArrayUsingComparator: sortByLastCharacter];
```

Возможно, наиболее примечательной возможностью блоков является следующая: переменные в области видимости в точке, где определен блок, сохраняют свои значения в блоке на этот момент, даже несмотря на то, что блок может быть выполнен в некоторый более поздний момент. (Технически мы говорим, что блок является замыканием и что значения переменных вне блока могут захватываться блоком.) Этот аспект блоков делает их полезными для определения функциональности, которая будет выполнена в более позднее время или даже в некотором другом потоке.

Вот пример из реальной практики:

```
CGPoint p = [v center];
CGPoint pOrig = p;
p.x += 100;
void (^anim) (void) = ^{
    [v setCenter: p];
};
void (^after) (BOOL) = ^(BOOL f) {
```

```
    [v setCenter: pOrig];
};
NSUInteger opts = UIViewAnimationOptionAutoreverse;
[UIView animateWithDuration:1 delay:0 options:opts
                 animations:anim completion:after];
```

Этот код делает нечто достаточно удивительное. Ìåòîä `animateWithDuration:delay:options:animations:completion:` настраивает анимацию с помощью блоков. Но сам вывод осуществляется позже; анимация, а следовательно, и блоки, будет выполняться в неопределенный момент в будущем, после завершения вызова метода, когда ваш код будет заниматься совсем другими вещами. Сейчас в области видимости имеется объект `v` типа `UIView`, наряду с `CGPoint p` и еще одним `CGPoint pOrig`. Переменные `p` и `pOrig` — локальные автоматические; они выходят из области видимости и прекращают существование до начала анимации и выполнения блоков. Тем не менее значения этих переменных используются внутри блоков в качестве параметров сообщений, отправляемых `v`.

Поскольку блок может выполняться позже, не совсем корректно выполнять присваивание значений локальным автоматическим переменным, определенным вне блока; поэтому компилятор постарается остановить вас сообщением “variable is not assignable” (переменная не допускает присваивание):

```
CGPoint p;
void (^aBlock) (void) = ^{
    p = CGPointMake(1,2); // Ошибка
};
```

Локальная автоматическая переменная может быть сделана подчиняющейся специальным правилам сохранения путем объявления переменной с помощью квалификатора `__block`. Этот квалификатор обеспечивает существование переменной вместе с блоком, который ее использует, и имеет два основных применения. Вот первое из них: если блок будет выполняться немедленно, квалификатор `__block` позволит ему установить переменную вне блока равной значению, которое будет необходимо после завершения блока.

Например, метод `enumerateObjectsUsingBlock:` класса `NSArray` принимает блок и немедленно вызывает его для каждого элемента массива. Это основанный на блоке эквивалент цикла `for...in`, который циклически проходит по элементам перечислимой коллекции (глава 1). Здесь мы предлагаем циклический проход по циклу до тех пор, пока не будет найдено искомое значение; когда мы найдем его, мы устанавливаем переменную (`dir`) равной этому значению. Однако эта переменная должна быть объявлена вне блока, поскольку она должна использоваться *после* выполнения блока — нам надо, чтобы ее область видимости выходила за пределы фигурных скобок блока. Поэтому мы объявляем ее с квалификатором `__block`, так что можем присваивать ей значение и внутри блока:

```
CGFloat h = newHeading.magneticHeading;
__block NSString* dir = @"N";
NSArray* cards = @[@"N", @"NE", @"E", @"SE",
                   @"S", @"SW", @"W", @"NW"];
[cards enumerateObjectsUsingBlock:^(id obj,
                                    NSUInteger idx, BOOL *stop) {
    if (h < 45.0/2.0 + 45*idx) {
        dir = obj;
        *stop = YES;
    }
}]; // Теперь мы можем использовать dir
```

(Обратите внимание, что присваивание выполняется разыменованному указателю на BOOL. Это способ преждевременного завершения цикла; если мы уже нашли интересующее нас значение, нет смысла продолжать работу цикла. Мы не можем использовать инструкцию break, поскольку на самом деле это не цикл for. Поэтому метод enumerateObjectsUsingBlock: передает блоку параметр, который представляет собой указатель на BOOL и которому блок может косвенно присвоить значение YES как сигнал методу о том, что можно остановить выполнение. Это одна из немногих ситуаций в программировании для iOS, где требуется разыменование указателя.)

Второе из двух применений квалификатора __block обратно первому. Оно реализуется тогда, когда блок будет выполняться через некоторое время после его определения, и мы хотим, чтобы блок использовал значение, которое переменная имеет в момент выполнения, а не захватывал ее значение в момент определения блока. Обычно это связано с тем, что тот же вызов метода, который получает блок (для последующего выполнения) одновременно и устанавливает значение этой переменной (сейчас).

Например, метод beginBackgroundTaskWithExpirationHandler: получает блок, который будет выполнен в некоторый будущий момент времени. Он также генерирует и возвращает UIBackgroundTaskIdentifier, который в действительности представляет собой просто целое число, — и мы хотим использовать это целое число в блоке, если и когда этот блок будет выполнен. Так что мы пытаемся действовать следующим образом: блок передается методу как аргумент, метод вызывается, метод возвращает значение, блок использует это значение. Квалификатор __block делает такую последовательность возможной:

```
__block UIBackgroundTaskIdentifier bti =
    [[UIApplication sharedApplication]
        beginBackgroundTaskWithExpirationHandler: ^{
            [[UIApplication sharedApplication]
                endBackgroundTask:bti];
        }];
```

Тогда же, когда в Objective-C были введены блоки, Apple предоставила системную библиотеку функций C под названием Grand Central Dispatch (GCD), которая интенсивно их использует. Главной целью GCD является управление потоками, но она также оказывается удобной для аккуратного и компактного выражения определенных понятий о том, когда должен быть выполнен код. К примеру, GCD может помочь задержать выполнение нашего кода (отложенное выполнение). В приведенном далее коде блок используется для того, чтобы сказать “измени границы UIView v1, но не прямо сейчас, а через две секунды”:

```
dispatch_time_t popTime =
    dispatch_time(DISPATCH_TIME_NOW, 2 * NSEC_PER_SEC);
dispatch_after(popTime, dispatch_get_main_queue(), ^(void){
    CGRect r = [v1 bounds];
    r.size.width += 40;
    r.size.height -= 50;
    [v1 setBounds: r];
});
```

Последний пример блока в действии представляет собой переписанный код из конца главы 1, где метод класса предоставляет объект-синглтон. Функция GCD dispatch_once — очень быстрая и (в отличие от примера из главы 1) безопасная в смысле потоков, — обеспечивает выполнение ее блока (который в данном случае создает синглтон) только один раз в течение всей работы программы. Таким образом, она гарантирует, что синглтон действительно является синглтоном:

```
+ (CardPainter*) sharedPainter {
    static CardPainter* sp = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sp = [CardPainter new];
    });
    return sp;
}
```

Блок в состоянии выполнить присваивание локальной переменной `sp` без квалификатора `__block`, потому что `sp` имеет квалификатор `static`, который дает тот же результат, но даже еще более сильный: так же, как `__block` продляет время жизни переменной до времени жизни блока, `static` продляет время жизни переменной до времени жизни самой программы. Таким образом, его побочный эффект — обеспечение возможности присваивания значений переменной `sp` внутри блока, так же как если бы она была объявлена с квалификатором `__block`.

Если вы хотите детальнее познакомиться с блоками, обратитесь к документации Apple по адресу `http://developer.apple.com/library/ios/#documentation/cocoa/Conceptual/Blocks/`, или к разделу “Blocks Programming Topics” в окне справки Xcode. Полная техническая спецификация синтаксиса блоков имеется по адресу `http://clang.llvm.org/docs/BlockLanguageSpec.html`.