Objective-Cプログラミング言語

Cocoa > Objective-C Language

2009-10-19

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
U.S.A.

アップルジャパン株式会社
〒163-1450 東京都新宿区西新宿
3 丁目20 番2 号
東京オペラシティタワー
http://www.apple.com/jp/

目次

図、リスト

Objective-Cプログラミング言語

Objective-C言語は、高度なオブジェクト指向プログラミングを可能にするために設計された簡単なコンピュータプログラミング言語です。Objective-Cは、小規模でも強力な、標準のANSI C言語の拡張セットとして定義されます。C言語への追加部分のほとんどは、初期のオブジェクト指向言語の1つであるSmalltalkに基づいています。Objective-Cは、C言語に完全なオブジェクト指向プログラミング機能を、分かりやすい形で追加することを意図して設計されています。

ほとんどのオブジェクト指向開発環境は、いくつかの部分から成ります。

- オブジェクト指向プログラミング言語
- オブジェクトのライブラリ
- 開発ツールスイート
- ランタイム環境

この文書は、開発環境の第1の要素、つまりプログラミング言語に関するものです。この文書では、Objective-C言語について網羅的に説明し、第2の要素であるMac OS XのObjective-Cアプリケーションフレームワーク（総称して「Cocoa」と呼ばれる）について学ぶための基礎を提供します。Cocoaの詳細については、『*Getting Started with Cocoa*』を参照してください。デベロッパが使用する2つの主要な開発ツール、XcodeとInterface Builderについては、『*Xcode Workspace Guide*』および『*Interface Builder*』にそれぞれ記述されています。ランタイム環境については、別の文書の『*Objective-C Runtime Programming Guide*』で説明しています。

重要： この文書は、Mac OS X v10.6でリリースされたバージョンのObjective-C言語について説明しています。このバージョンでは、関連参照(Associative References)が導入されました（「関連参照」（91 ページ）を参照）。Objective-C言語のバージョン1.0（Mac OS X v10.4以前で利用可能）については、『*Object Oriented Programming and the Objective-C Programming Language 1.0*』を参照してください。

対象読者

この文書は次のことに興味のある読者を対象としています。

- Objective-Cによるプログラミング
- Cocoaアプリケーションフレームワークの基礎知識

この文書では、Objective-Cの基盤であるオブジェクト指向モデルの紹介と、言語の完全な解説を行います。C言語に対するObjective-Cの拡張に焦点をあて、C言語そのものの説明は省きます。

この文書はC言語については解説されていないため、C言語にある程度慣れていることが前提となります。しかし、それほど熟達している必要はありません。Objective-Cによるオブジェクト指向プログラミングはANSICの手続き型プログラミングとはかなり違っているので、熟達したCプログラマでなくても、さほど不利にはなりません。

この書類の構成

この文書は複数の章と1つの付録から成ります。

次の各章では、Objective-C言語について説明します。標準C言語の基礎の上にObjective-C言語によって加えられたすべての拡張機能を取り上げます。

- 「オブジェクト、クラス、メッセージ」（13 ページ）
- 「クラスの定義」（37 ページ）
- 「オブジェクトの割り当てと初期化」（51 ページ）
- 「プロトコル」（61 ページ）
- 「宣言済みプロパティ」（71 ページ）
- 「カテゴリと拡張」（85 ページ）
- 「関連参照」（91 ページ）
- 「高速列挙」（95 ページ）
- 「静的な動作の実現」（99 ページ）
- 「セレクタ」（103 ページ）
- 「例外処理」（107 ページ）
- 「スレッド化」（111 ページ）
- 「リモートメッセージング」（113 ページ）

Appleのコンパイラは、GNU Compiler Collectionのコンパイラをベースにしています。Objective-Cの構文はGNU C/C++の構文のスーパーセットで、Objective-Cコンパイラは C、C++、およびObjective-Cのソースコードに対応しています。コンパイラは、Objective-Cのソースファイルをファイル名拡張子 `.m` によって認識し、標準Cの構文のみを含むファイルをファイル名拡張子 `.c` によって認識します。同様に、Objective-Cを使用するC++ファイルを拡張子 `.mm` によって認識します。Objective-CとC++を併用する場合のほかの問題については、「C++とObjective-Cの併用」（119ページ）の章を参照してください。

付録には、Objective-C言語の理解に役立つ以下の参考資料が含まれています。

- 「言語の要約」（125 ページ）では、C言語に対するObjective-Cの拡張のすべてを列挙し、簡単に解説します。

表記規則

この文書では、関数、メソッド、およびほかのプログラミング要素について説明する場合は、リテラル文字と斜体を使います。リテラル文字は、単語または文字を文字通りに受け取るべきこと（文字通りに入力すること）を表します。斜体は、ほかの何かを表す、あるいは変化する語を示します。たとえば、次の構文があるとします。

```
@interface ClassName ( CategoryName )
```

これは、`@interface`と2つの括弧が必須ですが、クラス名とカテゴリ名は選択できることを意味します。

コード例を示す場合、省略記号（...）は、コードの一部（しばしばかなりの量）が省略されていることを表します。

```
- (void)encodeWithCoder:(NSCoder *)coder
{
    [super encodeWithCoder:coder];
    ...
}
```

付録のリファレンスで使用する構文については、付録の中で説明します。

関連項目

オブジェクト指向プログラミングを使用してアプリケーションを作成した経験がない場合は、『*Object-Oriented Programming with Objective-C*』を読む必要があります。C++やJavaなどのほかのオブジェクト指向開発環境を使用したことがある場合も、これらの言語にはObjective-C言語とは異なる想定や規則がたくさんあるため、この文書を読むべきでしょう。『*Object-Oriented Programming with Objective-C*』は、Objective-Cデベロッパの視点からオブジェクト指向開発に習熟できるように作られています。オブジェクト指向設計の意味の一部を解説し、オブジェクト指向プログラムを書くということが、実際にはどのような感じのことであるかを伝えます。

ランタイム

『*Objective-C Runtime Programming Guide*』では、Objective-Cランタイムとその使いかたについて説明しています。

『*Objective-C Runtime Reference*』では、Objective-Cのランタイムサポートライブラリのデータ構造と関数について説明します。プログラムからこれらのインターフェイスを使用して、Objective-Cのランタイムシステムとやり取りすることができます。たとえば、クラスまたはメソッドを追加したり、ロードされているクラスの全クラス定義のリストを取得したりできます。

『*Objective-C Release Notes*』では、Mac OS Xの最新リリースにおけるObjective-Cランタイムの変更点のいくつかを説明しています。

メモリ管理

Objective-Cは、メモリ管理のために、自動ガベージコレクションと参照カウントの2つの環境をサポートしています。

- 『*Garbage Collection Programming Guide*』では、Cocoaで使用されるガベージコレクションシステムを説明しています（この環境はiPhoneでは利用できません。このため、iPhone Dev Centerからこの文書にアクセスすることはできません）。
- 『*Memory Management Programming Guide for Cocoa*』では、Cocoaで使用される参照カウントシステムを説明しています。

オブジェクト、クラス、メッセージ

この章では、Objective-C言語で使用し、実装するオブジェクト、クラス、メッセージの基本について説明します。また、Objective-Cランタイムについても紹介します。

ランタイム

Objective-C言語では、可能な限り多くの決定が、コンパイル時およびリンク時ではなく実行時に行われます。可能な場合は必ず、オブジェクトの作成やどのメソッドを呼び出すかの決定などの操作は動的に実行されます。つまり、Objective-C言語は、コンパイラだけでなく、コンパイルしたコードを実行するランタイムシステムも必要とします。ランタイムシステムは、Objective-C言語にとって一種のオペレーティングシステムとして動作し、言語を機能させるものです。ただし、通常はランタイムと直接やり取りをする必要はありません。ランタイムが提供する機能の詳細については、『*Objective-C Runtime Programming Guide*』を参照してください。

オブジェクト

オブジェクト指向プログラムは、その名称が示すように、**オブジェクト**を中心に構築されます。オブジェクトは、データと、そのデータを使用したりデータに作用したりする特定の操作を関連付けたものです。Objective-Cには、特定のクラスを指定せずにオブジェクト変数を識別するデータ型があります。これによって、動的型定義が可能になります。プログラムでは、通常、不要になったオブジェクトが確実に破棄されるようにする必要があります。

オブジェクトの基礎

オブジェクトは、データと、そのデータを使用したりデータに作用したりする特定の操作を関連付けたものです。Objective-Cでは、このような操作はオブジェクトの**メソッド**と呼ばれています。オブジェクトが作用する対象となるデータは**インスタンス変数**です。要するに、オブジェクトはデータ構造（インスタンス変数）とプロシージャのグループ（メソッド）を、自己完結型のプログラミング単位にまとめたものです。

たとえば、線、円、長方形、テキスト、ビットマップ画像などを組み合わせたイメージを作成できるドロープログラムを作成している場合は、ユーザが操作できるいろいろな基本形状のクラスを作成することが考えられます。たとえば、Rectangleオブジェクトは、図面内での長方形の位置に加えて、幅と高さを識別するインスタンス変数を持つことができます。そのほかにも、長方形の色、塗りつぶし、表示に使用する線パターンなどを定義するインスタンス変数が考えられます。Rectangleクラスは、インスタンスの位置、サイズ、色、塗りつぶし状態、線パターンを設定するメソッドに加えて、インスタンスを表示するメソッドを持つことになります。

Objective-Cでは、オブジェクトのインスタンス変数はオブジェクトに内在し、一般に、オブジェクトのメソッドによってのみオブジェクトの状態にアクセスできます（有効範囲を指定するディレクティブを使えば、サブクラスやほかのオブジェクトからインスタンス変数に直接アクセスできるかどうか指定できます。詳細については、「インスタンス変数の有効範囲」（42 ページ）を参照してください）。他者がオブジェクトに関する情報を得るには、情報を提供するメソッドがなければなりません。たとえば、Rectangleオブジェクトは、そのサイズと位置を示すメソッドを持っています。

さらに、オブジェクトはそのオブジェクト用に設計されたメソッドだけを認識するため、ほかの型のオブジェクトを対象としたメソッドを誤って実行することはありません。ローカル変数の保護のため、C関数がプログラムのほかの部分からローカル変数を隠すのと同じように、オブジェクトもそのインスタンス変数とメソッド実装を隠します。

id

Objective-Cでは、オブジェクト識別子は`id`という明確なデータ型として定められています。これは、クラスに関わらずどの種類のオブジェクトにも対応する汎用的な型です（クラスのインスタンスとクラスオブジェクト自身の両方に使用できます）。`id`は、オブジェクトデータ構造体へのポインタとして定義されています。

```
typedef struct objc_object {
    Class isa;
} *id;
```

このように、どのオブジェクトもそれがどのクラスのインスタンスかを表す`isa`変数を持っています。

用語の定義： Class型はそれ自身がポインタとして定義されています。

```
typedef struct objc_class *Class;
```

このため、`isa`変数は、しばしば「`isa`ポインタ」と呼ばれます。

したがって、C言語の関数や配列のように、オブジェクトはアドレスで識別されます。すべてのオブジェクトは、そのインスタンス変数やメソッドにかかわらず、`id`型です。

```
id anObject;
```

メソッドの戻り値など、Objective-Cのオブジェクト指向構成体では、`id`はデフォルトデータ型として`int`から置き換わりました（関数の戻り値など、Cに限定される構成体については、デフォルトの型は`int`のままです）。

キーワード`nil`は、NULLオブジェクト、すなわち値が0の`id`として定義されます。`id`、`nil`、その他のObjective-Cの基本的な型はヘッダファイル`objc/objc.h`で定義されています。

動的型定義

`id`型は完全に非制限的です。単独では、対象がオブジェクトであること以外の情報は示しません。

しかし、すべてのオブジェクトは同じではありません。Rectangleオブジェクトのメソッドとインスタンス変数は、ビットマップ画像を表すオブジェクトのものと同じではありません。ある時点で、プログラムは含んでいるオブジェクトに関する詳細な情報（オブジェクトのインスタンス変数、実行できるメソッドなど）を検出する必要があります。`id`型指示子はこのような情報をコンパイラに提供できないため、各オブジェクトが実行時にこれらの情報を提供できる必要があります。

`isa`インスタンス変数はオブジェクトの**クラス**（それがどの種類のオブジェクトか）を識別します。すべてのRectangleオブジェクトは、Rectangleであることをランタイムシステムに通知することができます。また、すべてのCircleオブジェクトもCircleであることを通知することができます。同じ振る舞い（メソッド）と同じ種類のデータ（インスタンス変数）を持つオブジェクトは、同じクラスのメンバです。

このように、オブジェクトは実行時に**動的に型定義**されます。必要な場合はいつでも、オブジェクトに要求するだけで、ランタイムシステムはオブジェクトが属している正確なクラスを検出することができます（ランタイムの詳細については、『*Objective-C Runtime Programming Guide*』を参照してください）。Objective-Cの動的型定義は、後述する動的バインディングの基礎となります。

また、`isa`変数によって、オブジェクトが**イントロスペクション**を実行して、オブジェクト自身（またはほかのオブジェクト）に関する情報を得ることが可能になります。コンパイラは、クラス定義に関する情報をデータ構造に記録して、ランタイムシステムが使用できるようにします。ランタイムシステムの関数は、`isa`を使用して、実行時にこの情報を検出します。ランタイムシステムでは、たとえば、オブジェクトが特定のメソッドを実装しているかどうかを突き止めたり、そのスーパークラスの名前を調べたりできます。

オブジェクトクラスの詳細については、「クラス」（24 ページ）を参照してください。

また、ソースコードの中でクラス名を使用して静的に型定義することで、オブジェクトのクラスに関する情報をコンパイラに提供することもできます。クラスは特別な種類のオブジェクトであり、クラス名は型の名前として機能します。詳細については、「クラスの型」（28 ページ）と「静的な動作の実現」（99 ページ）を参照してください。

メモリ管理

Objectiv-Cプログラムでは、不要になったオブジェクトは必ず割り当て解除することが重要です。そうしないと、アプリケーションのメモリ占有率が必要以上に大きくなります。また、まだ使用しているオブジェクトは絶対に割り当て解除しないようにすることも重要です。

Objective-Cではこれらの目標を達成することを可能にする、メモリ管理のための2つの環境を提供しています。

- **参照カウント**。オブジェクトの存続期間の決定についてデベロッパが最終的な責任を負います。

  参照カウントについては、『*Memory Management Programming Guide for Cocoa*』で説明しています。

- **ガベージコレクション**。自動「コレクタ」にオブジェクトの存続期間の決定権を渡します。

  ガベージコレクションについては『*Garbage Collection Programming Guide*』で説明しています（この環境はiPhoneでは利用できません。このため、iPhone Dev Centerからこの文書にアクセスすることはできません）。

オブジェクトメッセージング

このセクションでは、メッセージ送信の構文について説明します。メッセージ式をネストする方法なども取り上げます。また、オブジェクトのインスタンス変数の「可視性」や、ポリモーフィズムと動的バインディングの概念も説明します。

メッセージの構文

オブジェクトに何かを実行させるには、メソッドの適用を指示する**メッセージ**をオブジェクトに送信します。Objective-C では、**メッセージ式**を大括弧で囲みます。

```
[receiver message]
```

receiverはレシーバとなるオブジェクトであり、messageは実行すべきことをオブジェクトに通知します。ソースコードでは、メッセージは単にメソッドの名前とそれに渡される引数です。メッセージを送信すると、ランタイムシステムは、レシーバの持つすべてのメソッドの中から適切なメソッドを選択して呼び出します。

たとえば、次のメッセージは、`myRectangle`オブジェクトに対して、長方形を表示する`display`メソッドを実行するように指示します。

```
[myRectangle display];
```

メッセージの後には、C言語のコード行と同様に“;”が付きます。

メッセージ内のメソッド名はメソッドの実装を「選択する」役割を果たします。このような理由から、メッセージのメソッド名は、**セレクタ**とも呼ばれます。

メソッドは、パラメータ、つまり「引数」をとることもあります。単一の引数を持つメッセージでは、セレクタ名の後にコロン(:)が付き、コロンの直後に引数が付きます。この構造はキーワードと呼ばれます。次の例に示すように、キーワードは末尾にコロンが付き、そのコロンの後に引数が付きます。

```
[myRectangle setWidth:20.0];
```

セレクタ名は、コロンを含むすべてのキーワードを含みますが、それ以外のもの（戻り型やパラメータ型など）は含みません。次のメッセージ例は、`myRectangle`オブジェクトに対して、原点を座標(30.0, 50.0)に設定するように指示します。

```
[myRectangle setOrigin:30.0 :50.0]; // これは、複数の引数の悪い例
```

このコロンはメソッド名の一部であるため、メソッドの名前は`setOrigin::`となります。コロンが2つあるのは引数を2つとるためです。この特定のメソッドは、メソッド名に引数が組み込まれていません。そのため2つ目の引数は事実上ラベル付けされず、メソッドの引数の種類や目的を判断することは難しくなります。

本来は、メソッド名は、メソッドが想定する引数が自然に表現されるようなメソッド名となるように引数を名前に組み込むべきです。たとえば、Rectangleクラスは、2つの引数の目的を明確にする`setOriginX:y:`というようなメソッドを実装することができます。

```
[myRectangle setOriginX: 30.0 y: 50.0]; // これは、複数の引数の良い例
```

重要： メソッド名（セレクタ）を構成する要素は省略可能ではなく、順番も任意ではありません。多くの場合、「名前付き引数」と「キーワード引数」は、メソッドへの引数が実行時に数が異なってもよいこと、デフォルト値を持つこと、異なる順番になってもよいこと、および追加の名前付き引数を持てることを意味します。しかし、Objective-Cの場合は異なります。

Objective-Cにおけるメソッド宣言は、実質的には単に2つの引数が先頭に追加されたC関数です（『*Objective-C Runtime Programming Guide*』の「Messaging」を参照）。これはPythonのような言語で使用できる名前付き引数やキーワード引数とは異なります。

```
def func(a, b, NeatMode=SuperNeat, Thing=DefaultThing):
    pass
```

ここで、Thing（およびNeatMode）は省略可能です。または、呼び出し時に異なる値を持つことが可能です。

メソッドは、引数の数が可変のものもありますが、非常にまれです。付加的な引数は、メソッド名の後に、コンマで区切って指定します（コロンと異なり、コンマは名前の一部と見なされません）。次の例では、`makeGroup:`メソッドに、1つの必須引数(**group**)と3つのオプション引数を渡しています。

```
[receiver makeGroup:group, memberOne, memberTwo, memberThree];
```

標準のC関数と同じように、メソッドは値を返すことができます。次の例では、`myRectangle`を塗りつぶした長方形として描画する場合は`isFilled`変数を`YES`に、輪郭だけ描画する場合は`NO`に設定します。

```
BOOL isFilled;
isFilled = [myRectangle isFilled];
```

変数とメソッドは、同じ名前を持てることに注意してください。

メッセージ式は、別のメッセージ式内にネストすることができます。次の例では、ある長方形の色を別の長方形の色に設定します。

```
[myRectangle setPrimaryColor:[otherRect primaryColor]];
```

また、Objective-Cには、オブジェクトのアクセサメソッドを呼び出すためのコンパクトで便利な構文を提供するドット(.)演算子もあります。これは通常、宣言済みプロパティ機能と組み合わせて使用されます（「宣言済みプロパティ」（71 ページ）を参照）。これについては、「ドット構文」（20 ページ）で説明します。

nilへのメッセージ送信

Objective-Cでは、`nil`へのメッセージ送信が可能です（実行時に影響はありません）。Cocoaでは、これを利用するにはいくつかのパターンがあります。メッセージから`nil`に返される値も有効です。

- メソッドがオブジェクトを返す場合は、`nil`に送信されたメッセージは`0`(`nil`)を返します。次に例を示します。

  ```
  Person *motherInLaw = [[aPerson spouse] mother];
  ```

  `aPerson`の**spouse**が`nil`の場合、`mother`は`nil`に送信され、このメソッドは`nil`を返します。

- メソッドが任意のポインタ型、sizeof(void*)以下のサイズの任意の整数スカラー値、float、double、long double、またはlong longを返す場合、nilに送信されたメッセージは0を返します。
- メソッドがstructを返す場合、『*Mac OS X ABI Function Call Guide*』で定義されているようにレジスタに返されることになっているため、データ構造体のどのフィールドにもnilに送信されたメッセージは0.0を返します。その他のstructデータ型では0のみになることはありません。
- メソッドが前述の値の型以外のものを返す場合、nilに送信されたメッセージの戻り値は不定です。

次のコードの一部はnilへのメッセージ送信の正しい使いかたを示しています。

```
id anObjectMaybeNil = nil;

// これは有効
if ([anObjectMaybeNil methodThatReturnsADouble] == 0.0)
{
    // 実装が続く...
}
```

注： nilへのメッセージ送信の動作は、Mac OS X v10.5では少し異なります。

Mac OS X v10.4以前では、メッセージがオブジェクト、任意のポインタ型、void、またはsizeof(void*)のサイズ以下の任意の整数スカラー値を返す限りはnilへのメッセージは有効で、その場合、nilに送信されたメッセージはnilを返します。nilに送信されたメッセージが前述の値の型以外のものを返す場合（たとえば、任意のstruct型、任意の浮動小数点型、または任意のベクトル型を返す場合）、戻り値は不定です。したがって、メソッドの戻り値型がオブジェクト、任意のポインタ型、またはsizeof(void*)のサイズ以下の任意の整数スカラー値でない場合は、nilに送信されたメッセージの戻り値に依存すべきではありません。

レシーバのインスタンス変数

メソッドは、受信側オブジェクトのインスタンス変数に自動的にアクセスできます。インスタンス変数を引数としてメソッドに渡す必要はありません。たとえば、上記のprimaryColorメソッドは引数を持ちませんが、otherRectのプライマリカラーを検出して返すことができます。すべてのメソッドはレシーバとそのインスタンス変数を引き継ぐため、それらを引数として宣言する必要はありません。

このような規則により、Objective-Cのソースコードは簡素化されます。また、オブジェクトとメッセージに対するオブジェクト指向プログラマの考えかたもサポートされています。手紙が家庭に配達されるように、メッセージはレシーバに送信されます。メッセージ引数は、外部の情報をレシーバにもたらします。レシーバを取得してくる必要はありません。

メソッドは、レシーバのインスタンス変数にのみ自動的にアクセスできます。メソッドが別のオブジェクトに格納されている変数に関する情報を必要とする場合は、その変数の内容を明らかにするように要求するメッセージを該当するオブジェクトに送信する必要があります。上記のprimaryColorおよびisFilledメソッドは、まさにこの目的で使用しています。

インスタンス変数への参照の詳細については、「クラスの定義」（37 ページ）を参照してください。

ポリモーフィズム（多態性）

上記の例に示したように、Objective-Cのメッセージは、標準のC関数呼び出しと同じ構文上の位置に指定されます。しかし、メソッドはオブジェクトに属するため、メッセージは関数呼び出しとは異なる振る舞いをします。

特に、オブジェクトは、オブジェクト用に定義されたメソッドによってのみ操作できます。ほかのオブジェクトが同じ名前のメソッドを持っていても、ほかのオブジェクト用に定義されたメソッドと混同されることはありません。つまり、2つのオブジェクトは、同じメッセージに対して異なる応答をすることができます。たとえば、`display`メッセージを送信された各オブジェクトは、それぞれ独自の方法で自身を表示することができます。CircleとRectangleは、カーソルを追跡する同じ指示に対して異なる応答をします。

この機能は**ポリモーフィズム**と呼ばれ、オブジェクト指向プログラムの設計において重要な役割を果たします。動的バインディングとポリモーフィズムにより、さまざまな種類の多数のオブジェクトに適用できるコードを作成でき、コードを書く時点ではオブジェクトの種類を選択する必要はありません。たとえば、ほかのプロジェクトに取り組んでいる別のプログラマが後で開発するオブジェクトであってもかまいません。`id`変数に`display`メッセージを送信するコードを書く場合は、`display`メソッドを持つオブジェクトすべてが潜在的なレシーバになります。

動的バインディング

関数呼び出しとメッセージの大きな違いは、関数とその引数がコンパイル時にコードの中で結合されるのに対して、メッセージと受信側のオブジェクトはプログラムを実行してメッセージが送信されるまで結合されないことです。したがって、メッセージに応答するために呼び出される実際のメソッドは、コードのコンパイル時ではなく、実行時にのみ知ることができます。

メッセージが実際に呼び出すメソッドは、レシーバによって決まります。さまざまなレシーバが、同じメソッド名の異なるメソッド実装を持つことができます（ポリモーフィズム）。コンパイラが、メッセージに対する正しいメソッド実装を見つけるには、レシーバがどのようなオブジェクトなのか（どのクラスに属するか）を知る必要があります。この情報は、レシーバが実行時にメッセージを受信するとき提示できますが（動的型定義）、ソースコード内の型宣言からは分かりません。

メソッド実装の選択は、実行時に行われます。メッセージが送信されると、ランタイムメッセージングルーチンが、レシーバとメッセージに指定されたメソッド名を確認します。このルーチンは、メソッド名の一致するレシーバのメソッド実装を検出して、そのメソッドを呼び出し、レシーバのインスタンス変数へのポインタを渡します（このルーチンの詳細については、『*Objective-C Runtime Programming Guide*』の「Messaging」を参照してください）。

このような、メッセージに対するメソッドの**動的バインディング**は、ポリモーフィズムと連携することにより、オブジェクト指向プログラミングに対してより高い柔軟性と能力を提供します。各オブジェクトが独自のメソッドを持つことができるため、メッセージ自身を変えるのではなく、メッセージを受信するオブジェクトを変えることで、プログラムはさまざまな結果を得ることができます。これは、プログラム実行時に行えるため、レシーバを「その場」で決めることができ、ユーザ操作などの外部要因に応じて決まるようにもできます。

たとえば、Application Kitをベースにしたコードを実行している場合は、どのオブジェクトが「カット」、「コピー」、および「ペースト」などのメニューコマンドからメッセージを受信するかはユーザが決めます。メッセージは、現在の選択を制御している任意のオブジェクトに送信されます。テキストを表示するオブジェクトは、`copy`メッセージに対して、スキャン画像を表示するオブジェクトとは異なる反応を示します。一連の形状を表すオブジェクトは、Rectangleとは異なる応答

をします。メッセージが実行時までメソッドを選択しない（メソッドがメッセージにバインドされない）ため、応答の違いはメッセージに応答するメソッド内に隔離されます。メッセージを送信するコードは、それらの違いを考慮する必要はありませんし、可能性を列挙する必要もありません。アプリケーションはそれぞれに、`copy`メッセージに独自の方法で応答する、独自のオブジェクトを作成することができます。

Objective-Cでは動的バインディングをさらに一歩進めて、送信するメッセージ（メソッドセレクタ）も、実行時に決定する変数にすることができますこれについては、『*Objective-C Runtime Programming Guide*』の「Messaging」で解説しています。

動的メソッド解決

動的メソッド解決を使用して、クラスメソッドおよびインスタンスメソッドの実装を実行時に指定できます。詳細については、『*Objective-C Runtime Programming Guide*』の「Dynamic Method Resolution」を参照してください。

ドット構文

Objective-Cは、アクセサメソッドを呼び出す記法として、大括弧(`[ ]`)の代わりに使用できる、コンパクトで便利なドット(.)演算子を提供します。この記法は、別のオブジェクトのプロパティであるプロパティ（または、そのまた別のオブジェクトのプロパティ）にアクセスしたり、その変更をしたりする場合に特に役立ちます。

ドット構文の使用

概要

ドット構文を使うと、次の例に示すように、構造体の要素にアクセスするときと同じパターンを使ってアクセサメソッドを呼び出せます。

```
myInstance.value = 10;
printf("myInstance value:%d", myInstance.value);
```

ドット構文は純粋に「構文上の便宜」であり、コンパイラによってアクセサメソッドの呼び出しに変換されます（したがって、実際にはインスタンス変数に直接的にアクセスするわけではありません）。上記のコード例は、次のコードと完全に同じです。

```
[myInstance setValue:10];
printf("myInstance value:%d", [myInstance value]);
```

一般的な使用法

次の例に示すように、ドット(.)構文を使ってプロパティを読み書きできます。

リスト 1-1　ドット構文を使ったプロパティへのアクセス

```
Graphic *graphic = [[Graphic alloc] init];

NSColor *color = graphic.color;
CGFloat xLoc = graphic.xLoc;
```

```
BOOL hidden = graphic.hidden;
int textCharacterLength = graphic.text.length;

if (graphic.textHidden != YES) {
    graphic.text = @"Hello";
}
graphic.bounds = NSMakeRect(10.0, 10.0, 20.0, 120.0);
```

(@"Hello"は、定数のNSStringオブジェクトです（「コンパイラのディレクティブ」（126 ページ）を参照）。

*property*プロパティにアクセスすると、そのプロパティに対応するgetメソッド（デフォルトでは*property*）が呼び出され、プロパティを設定するとプロパティのsetメソッド（デフォルトでは`set`*Property*:）が呼び出されます。呼び出されるメソッドは、宣言済みプロパティの機能を使って変更できます（「宣言済みプロパティ」（71 ページ）を参照してください）。ドット構文では、そのように見えないかもしれませんが、カプセル化は維持されます。つまり、インスタンス変数に直接アクセスすることはありません。

次の文は、リスト 1-1（20 ページ）に示した文とまったく同じコードにコンパイルされますが、大括弧構文を使用しています。

リスト 1-2　大括弧構文を使ったプロパティへのアクセス

```
Graphic *graphic = [[Graphic alloc] init];

NSColor *color = [graphic color];
CGFloat xLoc = [graphic xLoc];
BOOL hidden = [graphic hidden];
int textCharacterLength = [[graphic text] length];

if ([graphic isTextHidden] != YES) {
    [graphic setText:@"Hello"];
}
[graphic setBounds:NSMakeRect(10.0, 10.0, 20.0, 120.0)];
```

ドット構文の利点は、コンパイラが読み取り専用プロパティへの書き込みを検出するとエラーを知らせることができる点ですが、存在しない`set`*Property*:メソッドを呼び出した場合は（実行時に失敗します）せいぜい未宣言メソッドの警告を生成できるだけです。

適切なC言語タイプのプロパティの場合、複合代入の意味は明確です。たとえば、次の複合代入を使って、NSMutableDataのインスタンスの長さプロパティを更新できます。

```
NSMutableData *data = [NSMutableData dataWithLength:1024];
data.length += 1024;
data.length *= 2;
data.length /= 4;
```

次のように記述しても同じです。

```
[data setLength:[data length] + 1024];
[data setLength:[data length] * 2];
[data setLength:[data length] / 4];
```

プロパティを使用できないケースが1つあります。次のコードを考えてみましょう。

```
id y;
x = y.z;  // zは宣言されていないプロパティ
```

yは型なしであり、zプロパティは宣言されていません。これは何通りかの解釈ができます。曖昧であるため、この文は未宣言のプロパティエラーとして扱われます。zが宣言されている場合、現在のコンパイル単位の中にzプロパティの宣言が1つだけあれば、コードは曖昧ではなくなります。zプロパティの宣言が複数ある場合は、それらすべてが同じ型（BOOLなど）であればコードは正当です。また、プロパティ宣言の1つがreadonlyと宣言されている場合も、曖昧さの原因の1つになります。

nil値

プロパティをたどっているときにnil値に遭遇した場合、その結果は、同等のメッセージをnilに送った場合と同じです。たとえば、次の組み合わせはすべて同じです。

```
// パスの各メンバはオブジェクト
x = person.address.street.name;
x = [[[person address] street] name];

// パスにはCの構造体が含まれおり、
// windowがnilの場合、または-contentViewがnilを返した場合はクラッシュする
y = window.contentView.bounds.origin.y;
y = [[window contentView] bounds].origin.y;

// setterの使用例....
person.address.street.name = @"Oxford Road";
[[[person address] street] setName:@"Oxford Road"];
```

self

アクセサメソッドを使ってselfのプロパティにアクセスする場合は、次の例に示すようにselfを明示的に呼び出す必要があります。

```
self.age = 10;
```

self.を使用しない場合は、インスタンス変数に直接アクセスします。次の例では、ageプロパティのsetアクセサメソッドは呼び出されません。

```
age = 10;
```

パフォーマンスとスレッド

ドット構文では、標準的なメソッド呼び出し構文に相当するコードが生成されます。したがって、ドット構文を使ったコードは、アクセサメソッドを使って直接記述したコードとまったく同じ動作をします。ドット構文はメソッドを単に呼び出すだけであるため、使用した結果、スレッドの依存性がさらに発生することはありません。

使用方法のまとめ

```
aVariable = anObject.aProperty;
```

aPropertyメソッドを呼び出し、戻り値をaVariableに代入します。プロパティaPropertyの型とaVariableの型は互換性がなくてはなりません。互換性がないとコンパイラ警告が生じます。

```
anObject.name = @"New Name";
```

setName:メソッドをanObjectに対して呼び出し、引数として@"New Name"を渡します。

setName:が存在しないかプロパティnameが存在しない場合、あるいはsetName:がvoid以外を返した場合はコンパイラ警告が生じます。

```
xOrigin = aView.bounds.origin.x;
```

boundsメソッドを呼び出し、xOriginがboundsから返されたNSRectのorigin.x構造体の値になるように代入します。

```
NSInteger i = 10;
anObject.integerProperty = anotherObject.floatProperty = ++i;
```

anObject.integerPropertyとanotherObject.floatPropertyの両方に11を代入します。つまり代入の右辺が先に評価され、結果がsetIntegerProperty:とsetFloatProperty:に渡されるということです。先に評価された結果は、代入の各ポイントで必要に応じて変換されます。

不正な使用

以下に示すパターンの使用は勧められません。

```
anObject.retain;
```

コンパイラ警告が生じます(warning: value returned from property not used.)。

```
/* メソッド宣言 */
- (BOOL) setFooIfYouCan:(MyClass *)newFoo;

/* コード */
anObject.fooIfYouCan = myInstance;
```

setFooIfYouCan:は、(void)を返さないためsetterメソッドであるように見えないことを示すコンパイラ警告が生じます。

```
flag = aView.lockFocusIfCanDraw;
```

lockFocusIfCanDrawを呼び出し、値をflagに代入します。この場合、flagの型がメソッドの戻り値の型と不一致でない限り、コンパイラ警告は生じません。

```
/* プロパティ宣言 */
@property(readonly) NSInteger readonlyProperty;
/* メソッド宣言 */
- (void) setReadonlyProperty:(NSInteger)newValue;

/* コード */
self.readonlyProperty = 5;
```

プロパティがreadonlyとして宣言されているため、このコードではコンパイラ警告が生じます(warning: assignment to readonly property 'readonlyProperty')。setterメソッドが存在するため実行時には動作しますが、プロパティのsetterを単に追加してもreadwriteの意味を持たせることはできません。

クラス

オブジェクト指向プログラムは、通常、さまざまなオブジェクトで作られています。Cocoaフレームワークをベースにしたプログラムでは、NSMatrixオブジェクト、NSWindowオブジェクト、NSDictionaryオブジェクト、NSFontオブジェクト、NSTextオブジェクト、その他多くのオブジェクトを使用します。また、しばしば、同じ種類（クラス）の複数のオブジェクト（たとえば、NSArrayオブジェクトやNSWindowオブジェクト）を使用します。

Objective-Cでは、クラスを定義することでオブジェクトを定義します。クラス定義は、一種のオブジェクトのプロトタイプです。クラスのあらゆるメンバの一部になるインスタンス変数を宣言し、クラスの全オブジェクトが使用できるメソッドのセットを定義します。

コンパイラにより、クラスごとに1つだけアクセス可能なオブジェクトが作成されます。これが**クラスオブジェクト**で、そのクラスに属する新しいオブジェクトの構築方法を知っています（そのため、伝統的に「ファクトリオブジェクト」と呼ばれています）。クラスオブジェクトはコンパイル済みのクラスであり、それによって構築されるオブジェクトがクラスの**インスタンス**です。プログラムの主な作業を実行するオブジェクトは、実行時にクラスオブジェクトによって作成されたインスタンスです。

クラスの全インスタンスは同じメソッドのセットを持っており、すべてが同じ鋳型に基づくインスタンス変数のセットを持っています。オブジェクトはそれぞれ固有のインスタンス変数を持ちますが、メソッドは共有されます。

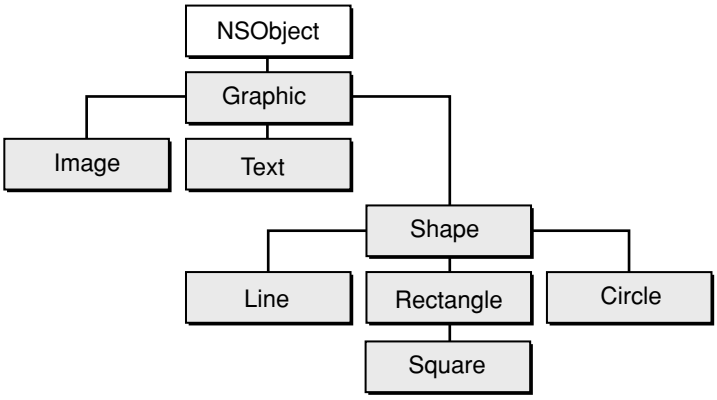
慣習的に、クラス名は大文字で始まり（「Rectangle」など）、インスタンス名は通常小文字で始まります（「myRectangle」など）。

継承

クラス定義は追加的に定義していきます。つまり、定義する新しいクラスはすべて別のクラスをベースにしており、そのメソッドとインスタンス変数を**継承**します。新しいクラスでは、継承したものに追加や変更を加えるだけです。継承したコードを複製する必要はありません。

継承によって、すべてのクラスがリンクされ、1つのクラスをルートに持つ階層ツリーを形成します。Foundationフレームワークをベースにしたコードを書いている場合、そのルートクラスは一般的にNSObjectです。（ルートクラスを除く）すべてのクラスには、ルートに1ステップ近い**スーパークラス**があります。また、（ルートクラスを含む）どのクラスも、ルートから1ステップ遠い任意の**サブクラス**のスーパークラスになります。図 1-1はドロープログラムで使用されるいくつかのクラスの階層を示したものです。

図 1-1 ドロープログラムのクラス



この図に示すように、SquareクラスはRectangleのサブクラス、RectangleクラスはShapeのサブクラス、ShapeはGraphicのサブクラス、GraphicはNSObjectのサブクラスです。継承は累積的なものです。したがって、Squareオブジェクトは、特にSquareのために定義されたメソッドとインスタンス変数だけでなく、Rectangle、Shape、Graphic、およびNSObjectのために定義されたメソッドとインスタンス変数も持っています。これは簡単にいえば、SquareオブジェクトはSquareであるだけでなく、Rectangle、Shape、Graphic、およびNSObjectでもあるということです。

そのため、NSObject以外のすべてのクラスは、別のクラスを特殊化または適合化したものと考えることができます。下位のサブクラスはそれぞれ継承したものの累計に変更を加えます。たとえば、Squareクラスでは、RectangleをSquareに変えるのに必要な最小限のものだけが定義されます。

クラスを定義する際には、そのスーパークラスを宣言することでクラスを階層にリンクします。作成するすべてのクラスは、（新しいルートクラスを定義しないかぎり）別のクラスのサブクラスである必要があります。スーパークラスとして利用できるクラスは多数あります。Cocoaには、NSObjectクラスと、250以上の追加クラスの定義を含む複数のフレームワークがあります。そのまま使用できる（そのままプログラムに組み込める）クラスもあります。また、サブクラスを定義して自分自身のニーズに合わせられるものもあります。

フレームワーククラスには、必要なものをほとんど定義しながら、一部の詳細はサブクラスの実装に任されているものもあります。したがって、コードを少しだけ記述して、フレームワークのプログラマによる成果を再利用すれば、非常に高度なオブジェクトを作成することができます。

NSObjectクラス

NSObjectはルートクラスなので、スーパークラスはありません。Objective-Cオブジェクトとオブジェクトの対話の基本的フレームワークは、NSObjectで定義されています。NSObjectは、自身を継承するクラスとクラスのインスタンスに、オブジェクトとして動作し、ランタイムシステムと連携する能力を与えます。

特別な動作を別のクラスから継承する必要のないクラスでも、NSObjectクラスのサブクラスにするべきです。クラスのインスタンスには、少なくとも実行時にObjective-Cオブジェクトのように動作する能力が必要です。このような能力をNSObjectクラスから継承するほうが、新しいクラス定義で新たに作成することに比べると、より簡単で信頼性も高くなります。

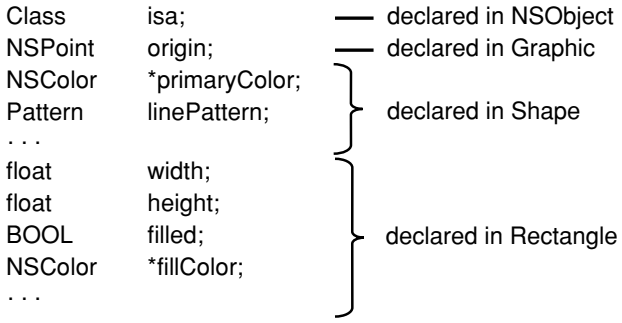
注： 新しいルートクラスを実装するのは、慎重な作業が必要で、多くの危険が潜んでいます。そのクラスは、インスタンスを割り当て、クラスに接続し、ランタイムシステムで識別するなど、NSObjectクラスが実行する多くのことを複製しなければなりません。そのため、通常は、ルートクラスとしてCocoaに提供されているNSObjectクラスを使用してください。詳細については、Foundationフレームワークドキュメントで`NSObject`クラスと`NSObject`プロトコルを参照してください。

インスタンス変数の継承

クラスオブジェクトが新しいインスタンスを作成すると、新しいオブジェクトにはそのクラスに定義されたインスタンス変数だけでなく、そのスーパークラスと、スーパークラスのスーパークラスに定義されたインスタンス変数も、ルートクラスまで遡って含まれます。したがって、`NSObject`クラスで定義された`isa`インスタンス変数は、あらゆるオブジェクトの一部になります。`isa`は各オブジェクトをそのクラスに結び付けます。

図 1-2はRectangleの個々の実装に定義できるインスタンス変数と、それらがどこから継承されているかを示しています。オブジェクトをRectangleにする変数がオブジェクトをShapeにする変数に追加され、オブジェクトをShapeにする変数がオブジェクトをGraphicにする変数に追加されるというようになっている点に注目してください。

図 1-2 Rectangleのインスタンス変数



クラスでのインスタンス変数の宣言は必須ではありません。新しいメソッドを単に定義して、何らかのインスタンス変数が必要な場合は、継承するインスタンス変数に依存することができます。たとえば、Squareクラスは、自身の新しいインスタンス変数を宣言しなくてもかまいません。

メソッドの継承

オブジェクトは、そのクラスに定義されたメソッドだけでなく、そのスーパークラスと、スーパークラスのスーパークラスに定義されたメソッドにも、階層のルートクラスまで遡ってアクセスできます。たとえば、Squareオブジェクトは、自身のクラスで定義されたメソッドはもちろん、Rectangle、Shape、Graphic、および`NSObject`クラスで定義されたメソッドも使用することができます。

したがって、プログラムで定義する新しいクラスは、階層で上位にあるすべてのクラスのために書かれたコードを利用することができます。このような継承は、オブジェクト指向プログラミングの主要なメリットです。Cocoaの提供するオブジェクト指向フレームワークのいずれかを使用すると、プログラムはフレームワーククラスのコードとして実装されている基本機能を利用することができます。追加する必要があるのは、標準機能をアプリケーションに合わせてカスタマイズするコードだけです。

クラスオブジェクトも、階層で上位にあるクラスを継承します。ただし、クラスオブジェクトはインスタンス変数を持たないため（インスタンスだけが持ちます）、メソッドだけを継承します。

メソッドのオーバーライド

継承には1つの便利な例外があります。新しいクラスを定義する際に、階層の上位にあるクラスで定義されたメソッドと同じ名前で、新しいメソッドを実装することができます。新しいメソッドはオリジナルをオーバーライドします。新しいクラスのインスタンスはオリジナルではなく新しいメソッドを実行し、新しいクラスのサブクラスもオリジナルではなく新しいメソッドを継承します。

たとえば、Rectangleは独自の`display`を定義することによって、Graphicに定義された`display`メソッドをオーバーライドします。Graphic版のメソッドはGraphicクラスを継承するあらゆるオブジェクトで利用可能ですが、Rectangleオブジェクトでは利用できません。Rectangleオブジェクトでは代わりに、独自版の`display`が実行されます。

メソッドをオーバーライドするとオリジナルを継承できなくなりますが、新しいクラスで定義されたほかのメソッドは、再定義されたメソッドをスキップして、オリジナルを見つけることができます（詳細については、「selfとsuperに対するメッセージ」 （45 ページ）を参照してください）。

また、再定義したメソッドには、オーバーライド対象メソッドを組み込むことができます。この場合、新しいメソッドはオーバーライド対象メソッドを完全に置き換えるのではなく、改良または変更するにすぎません。階層内の複数のクラスで同じメソッドを定義し、それぞれの新しいバージョンでオーバーライド対象メソッドを組み込んでいる場合、実質的には元のメソッドの実装がすべての対象クラスに拡散されていることになります。

サブクラスは継承したメソッドをオーバーライドできますが、継承したインスタンス変数はオーバーライドできません。オブジェクトは継承するすべてのインスタンス変数にメモリを割り当てるため、同じ名前で新しいインスタンス変数を宣言して、継承した変数をオーバーライドすることはできません。オーバーライドを試みると、コンパイラがエラーを表示します。

抽象クラス

クラスの中には、ほかのクラスに継承されることのみを目的としているものや、主にほかのクラスに継承されることを目的としているものもあります。このような**抽象クラス**は、さまざまなサブクラスが使用できるメソッドとインスタンス変数を共通の定義にグループ化します。抽象クラスは通常、単独では不完全ですが、サブクラスを実装する負担を軽減するのに役立つコードが含まれています（抽象クラスを使用するにはサブクラスが必要であるため、**抽象スーパークラス**と呼ばれることもあります）。

ほかの言語とは異なり、Objective-Cには、クラスを抽象クラスとしてマークする構文はありません。また、抽象クラスのインスタンスを作成することも妨げられません。

`NSObject`クラスはCocoaでの抽象クラスの標準的な例です。`NSObject`クラスのインスタンスをアプリケーションで使用することはありません。これは何かに使用できるものではなく、特に何かを行う機能のない汎用オブジェクトです。

これに対して、`NSView`クラスは、場合によっては直接使用する可能性のある抽象クラスの一例です。

抽象クラスには多くの場合、アプリケーションの構造を定義するのに役立つコードが含まれています。抽象クラスのサブクラスを作成すると、新しいクラスのインスタンスは特に問題なくアプリケーション構造に適合し、ほかのオブジェクトと自動的に連携します。

クラスの型

クラス定義は特定の種類のオブジェクトの仕様です。したがって、クラスは、実質的にデータ型を規定します。このデータ型は、クラスで定義されるデータ構造（インスタンス変数）だけでなく、定義に含まれている動作（メソッド）もベースにしています。

`sizeof`演算子の引数などのように、C言語において型指定子を指定できる場所ならば、クラス名をソースコードに記述できます。

```
int i = sizeof(Rectangle);
```

静的な型定義

`id`の代わりにクラス名を使用して、オブジェクトの型を指定することができます。

```
Rectangle *myRectangle;
```

このような方法でのオブジェクト型の宣言は、オブジェクトの種類に関する情報をコンパイラに提供するため、**静的な型定義**と呼ばれています。`id`が実際にはポインタであるのと同様に、オブジェクトはクラスへのポインタとして静的に型定義されています。オブジェクトは必ずポインタとして型定義されます。静的な型定義はポインタを明示的なものにし、`id`はポインタを隠蔽します。

静的な型定義により、コンパイラは型チェックを行うことができ（たとえば、オブジェクトがメッセージを受信しても応答できないと警告する）、一般的に`id`として型定義されたオブジェクトに適用される制限を緩和することができます。さらに、ほかの人に対して、ソースコードの意図を明らかにします。ただし、静的な型定義は動的バインディングを無効化するものではなく、実行時におけるレシーバのクラスの動的な決定を変更するものでもありません。

オブジェクトは、自身のクラスまたは継承するクラスとして静的に型定義することができます。たとえば、継承によりRectangleはGraphicの一種であるため、Rectangleインスタンスは`Graphic`クラスに合わせて静的に型定義できます。

```
Graphic *myRectangle;
```

これが可能なのは、RectangleがGraphicであるためです。ShapeとRectangleのインスタンス変数とメソッドの機能も持っているため、RectangleはGraphicを越える機能を持ちますが、それでもやはりGraphicクラスであることに変わりありません。型チェックのために、コンパイラは`myRectangle`がGraphicであると見なしますが、実行時にはRectangleとして扱われます。

静的な型定義とそのメリットの詳細については、「静的な動作の実現」（99 ページ）を参照してください。

型のイントロスペクション

インスタンスは、実行時にその型を明らかにすることができます。`isMemberOfClass:`メソッドは`NSObject`クラスで定義されており、レシーバが特定のクラスのインスタンスかどうかをチェックします。

```
if ( [anObject isMemberOfClass:someClass] )
    ...
```

isKindOfClass:メソッドもNSObjectクラスで定義されており、レシーバが特定のクラスを継承しているか、またはそのクラスに属しているか（レシーバの継承パス内にそのクラスがあるかどうか）を、もっと広くチェックします。

```
if ( [anObject isKindOfClass:someClass] )
    ...
```

isKindOfClass:がYESを返すクラスのセットは、レシーバを静的に型定義できるものと同じセットです。

イントロスペクションで調べられる情報は型情報に限られていません。この章の後のセクションでは、クラスオブジェクトを返したり、オブジェクトがメッセージに応答できるかどうかを報告したり、その他の情報を明らかにするメソッドについて説明します。

isKindOfClass:、isMemberOfClass:および関連するメソッドの詳細については、Fundationフレームワークリファレンスの`NSObject`クラス仕様を参照してください。

クラスオブジェクト

クラス定義には、次のようなさまざまな情報が含まれており、その大部分はクラスのインスタンスに関するものです。

- クラスとそのスーパークラスの名前
- インスタンス変数のセットを記述したテンプレート
- メソッド名およびその戻り値と引数の型の宣言
- メソッドの実装

これらの情報はコンパイルされて、ランタイムシステムで利用できるデータ構造に記録されます。コンパイラは、クラスを表すオブジェクト、すなわち**クラスオブジェクト**を1つだけ作成します。クラスオブジェクトは、クラスに関するすべての情報にアクセスできます。これは、主にクラスのインスタンスを表す情報です。クラス定義によって規定されるプランに従って、新しいインスタンスを生成することができます。

クラスオブジェクトはクラスインスタンスのプロトタイプを保持していますが、それ自体はインスタンスではありません。クラスオブジェクトは独自のインスタンス変数を持っていませんし、クラスのインスタンスを対象としたメソッドを実行することができません。しかし、クラス定義は特にクラスオブジェクトを対象としたメソッド、すなわち**インスタンスメソッド**ではなく、**クラスメソッド**を含むことができます。インスタンスがインスタンスメソッドを継承するのと同じように、クラスオブジェクトは階層の上位にあるクラスからクラスメソッドを継承します。

ソースコードでは、クラスオブジェクトはクラス名で表されます。次の例では、Rectangleクラスが`NSObject`クラスから継承したメソッドを使用してクラスのバージョン番号を返します。

```
int versionNumber = [Rectangle version];
```

ただし、クラス名は、メッセージ式の単なるレシーバとしてのクラスオブジェクトを表します。その他の場合は、インスタンスまたはクラスに対して`id`クラスを返すように要求する必要があります。次のどちらの文も`class`メッセージに応答します。

```
id aClass = [anObject class];
id rectClass = [Rectangle class];
```

上記の例に示すように、クラスオブジェクトはほかのオブジェクトと同様に、idとして型定義できます。しかし、クラスオブジェクトは、Classデータ型としてより明示的に型定義することもできます。

```
Class aClass = [anObject class];
Class rectClass = [Rectangle class];
```

すべてのクラスオブジェクトはClass型です。この型名をクラスに使用するのは、クラス名を使用してインスタンスを静的に型定義するのと同じです。

つまり、クラスオブジェクトは、動的に型定義したり、メッセージを受信したり、ほかのクラスからメソッドを継承することができる完全なオブジェクトです。クラスオブジェクトが特別であるのは、コンパイラによって作成され、クラス定義に基づいて構築されるものを除けば自身のデータ構造（インスタンス変数）がなく、実行時にインスタンスを生成するエージェントであるという点だけです。

注： コンパイラは、各クラスの「メタクラスオブジェクト」も構築します。クラスオブジェクトがクラスのインスタンスを示すのと同じように、メタクラスオブジェクトはクラスオブジェクトを示します。ただし、インスタンスやクラスオブジェクトにメッセージを送信できるのに対し、メタクラスオブジェクトはランタイムシステムによって内部的に使用されるだけです。

インスタンスの作成

クラスオブジェクトの主要な機能は、新しいインスタンスを作成することです。次のコードは、Rectangleクラスに対して新しいRectangleインスタンスを作成して、myRectangle変数に割り当てるように指示します。

```
id  myRectangle;
myRectangle = [Rectangle alloc];
```

allocメソッドは、新しいオブジェクトのインスタンス変数に動的にメモリを割り当て、すべてを0に初期化します。ただし、新しいインスタンスをそのクラスに結び付けるisa変数は除きます。オブジェクトが有用であるためには、通常は完全に初期化する必要があります。その初期化を行うのがinitメソッドの機能です。初期化は、通常、割り当ての直後に行います。

```
myRectangle = [[Rectangle alloc] init];
```

この章のこれまでの例で示したメッセージをmyRectangleで受信するには、先に上記のようなコードが必要です。allocメソッドは新しいインスタンスを返し、そのインスタンスがinitメソッドを実行して初期状態に設定します。すべてのクラスオブジェクトに、新しいオブジェクトを生成するメソッド（allocなど）が少なくとも1つはあり、すべてのインスタンスは、それを使えるように準備するメソッド（initなど）が少なくとも1つあります。初期化メソッドは多くの場合、特定の値を渡せる引数を持っており、引数にラベルを付けるキーワードを持っていますが（たとえば、新しいRectangleインスタンスを初期化するinitWithPosition:size:のようなメソッドなど）、それらはすべて「init」という文字列で始まります。

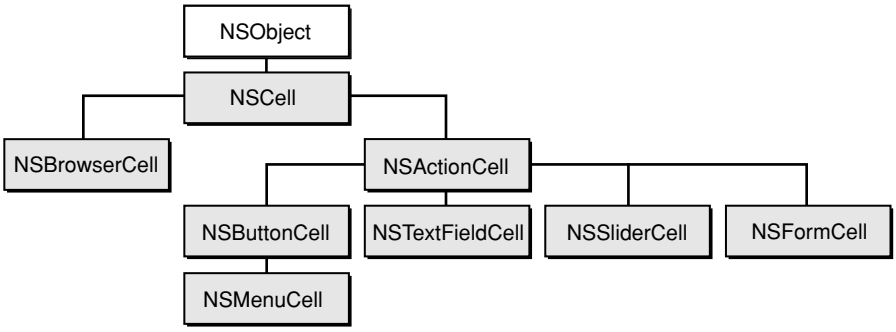
クラスオブジェクトによるカスタマイズ

クラスをオブジェクトとして扱うのは、Objective-C言語では偶然ではありません。それは、意図的で、ときには意外な設計上のメリットを持つ選択なのです。たとえば、クラスが制限のないセットに属している場合は、クラスでオブジェクトをカスタマイズすることができます。Application Kitでは、たとえば、特定の種類のNSCellオブジェクトを使用してNSMatrixオブジェクトをカスタマイズできます。

NSMatrixオブジェクトは、セルを表す個々のオブジェクトを作成することができます。オブジェクトの作成は、行列を最初に初期化する際と、あとで新しいセルが必要なときに可能です。NSMatrixオブジェクトが画面上に描画する目に見える行列は、実行時にユーザの操作に応じて拡大／縮小することができます。拡大する場合、行列は、追加される新しいスロットを満たすため、新しいオブジェクトを生成する必要があります。

それらを、どのようなオブジェクトにすべきかについて次に説明します。それぞれの行列は1種類のNSCellのみを表示しますが、種類はさまざまにあります。図 1-3の継承階層は、Application Kitによって提供されるものの一部です。すべてが汎用NSCellクラスを継承します。

図 1-3 NSCellの継承階層



行列がNSCellオブジェクトを作成するとき、それらは一連のボタンやスイッチを表示するNSButtonCellオブジェクトであるべきでしょうか。それともユーザがテキストの入力や編集ができるフィールドを表示するNSTextFieldCellオブジェクトであるべきでしょうか。あるいはほかの種類のNSCellにするべきでしょうか。NSMatrixオブジェクトは、あらゆる種類のセルに対応しなければなりません。まだ作成されていない型についても考慮しなければなりません。

この問題に対する1つの解決策は、NSMatrixクラスを抽象クラスとして定義し、それを使用する全員にサブクラスの宣言と、新しいセルを生成するメソッドの実装を義務付けることです。NSMatrixクラスのユーザは必要なメソッドを実装するため、作成したオブジェクトが適切な型であることを保証できます。

しかし、これは、NSMatrixクラスでなされているべき作業を他人に求めることになるため、クラスの数が不必要に増えます。アプリケーションは複数のNSMatrixを必要とし、それぞれが異なる種類のNSCellを持つ可能性があるため、NSMatrixのサブクラスで雑然とする可能性があります。新しいNSCellを作成するたびに、新しいNSMatrixも定義する必要があります。さらに、別々のプロジェクトのプログラマが、同じ処理を実行するほとんど同じようなコードを書くことになります。すべてはNSMatrixがしないことを補うためです。

より優れた解決策、すなわちNSMatrixクラスが実際に採用している解決策は、NSMatrixインスタンスを一種のNSCellで（クラスオブジェクトで）初期化することです。NSMatrixは、NSMatrixが空のスロットを満たすために使用するNSCellオブジェクトを表すクラスオブジェクトを渡すsetCellClass:メソッドを定義します。

```
[myMatrix setCellClass:[NSButtonCell class]];
```

NSMatrixオブジェクトは、最初に初期化するとき、および、より多くのセルを含むようにサイズ変更するときに、クラスオブジェクトを使用して新しいセルを生成します。クラスが、メッセージで渡したり、変数に割り当てることのできるオブジェクトでなければ、このようなカスタマイズは困難です。

変数とクラスオブジェクト

新しいクラスを定義する際、インスタンス変数を指定することができます。クラスのあらゆるインスタンスが、宣言された変数のコピーをそれぞれに保持することができ、各オブジェクトが自身のデータを制御します。ただし、インスタンス変数に対応する「クラス変数」はありません。クラスに提供されるのは、クラス定義に基づいて初期化された内部データ構造だけです。また、クラスオブジェクトは、どのインスタンスのインスタンス変数にもアクセスできません。すなわち、インスタンス変数の初期化、読み取り、または変更ができません。

クラスのすべてのインスタンスがデータを共有するには、何らかの外部変数が必要になります。最も簡単に実現するには、次のコードで示すようにクラス実装ファイルで変数を宣言します。

```
int MCLSGlobalVariable;

@implementation MyClass
// 実装が続く
```

より洗練された実装では、static変数を宣言し、それらを管理するクラスメソッドを提供します。static変数を宣言すると、その有効範囲は当該クラスのみ、厳密にはそのファイルに実装されたクラスの部分に限定されます（したがって、インスタンス変数と異なり、静的変数をサブクラスによって継承したり、サブクラスで直接操作したりできません）。このパターンは、クラスの共有インスタンス（シングルトンなど）の定義によく使用されます（シングルトンについては、『*Cocoa Fundamentals Guide*』の「Creating a Singleton Instance」を参照してください）。

```
static MyClass *MCLSSharedInstance;

@implementation MyClass

+ (MyClass *)sharedInstance
{
    // 共有インスタンスの有無を確認する
    // 必要なら作成する
    return MCLSSharedInstance;
}
// 実装が続く
```

静的な変数は、クラスオブジェクトに対して、インスタンスを生成する「ファクトリ」以上の機能を与える役割も持ち、それ自体で完全で多目的なオブジェクトになることができます。クラスオブジェクトは、作成するインスタンスの間を取り持ったり、作成済みオブジェクトのリストからインスタンスを削除したり、アプリケーションに不可欠なほかの処理を管理するために使用することができます。特定クラスのオブジェクトが1つだけ必要な場合は、オブジェクトの状態をすべて静的な変数に入れて、クラスメソッドのみを使用するようにできます。これにより、インスタンスの割り当てと初期化のステップを省けます。

注： staticとして宣言されていない外部変数を使用することもできますが、別々のオブジェクトにデータをカプセル化するには、静的な変数によって有効範囲を限定するほうが有効です。

クラスオブジェクトの初期化

クラスオブジェクトをインスタンスの割り当て以外に使用する場合は、インスタンスのように初期化する必要がある場合もあります。プログラムはクラスオブジェクトを割り当てませんが、Objective-Cはプログラムがそれらを初期化する手段を提供します。

クラスが静的な変数またはグローバル変数を利用する場合は、initializeメソッドがそれらの初期値の設定に適しています。たとえば、クラスがインスタンスの配列を保持する場合、initializeメソッドでその配列を準備し、さらに1つか2つのデフォルトインスタンスを割り当てて用意しておくことができます。

ランタイムシステムは、クラスがほかのメッセージを受信する前、およびそのスーパークラスがinitializeメッセージを受信した後に、initializeメッセージをすべてのクラスオブジェクトに送信します。これにより、クラスは自身が使用される前に、ランタイム環境を準備する機会を与えられます。初期化が不要な場合は、メッセージに応えるinitializeメソッドを書く必要はありません。

継承があるため、スーパークラスがすでにinitializeメッセージを受信していても、initializeメソッドを実装していないクラスへ送信されたinitializeメッセージは、スーパークラスへ転送されます。たとえば、クラスAはinitializeメソッドを実装しており、クラスBはクラスAを継承しているけれどもinitializeメソッドは実装していないと仮定します。クラスBが最初のメッセージを受信する直前に、ランタイムシステムはクラスBへinitializeを送信します。ただし、クラスBはinitializeを実装していないため、クラスAのinitializeが代わりに実行されます。そのため、クラスAでは、初期化ロジックが1回だけ、適切なクラスに対して実行されるようにしなければなりません。

初期化ロジックが複数回実行されるのを防ぐには、initializeメソッドを実装する際にリスト 1-3に示すテンプレートを使用します。

リスト 1-3　initializeメソッドの実装

```
+ (void)initialize
{
  if (self == [ThisClass class]) {
        // ここで初期化を実行する
        ...
    }
}
```

注： ランタイムシステムはクラスごとにinitializeを送信します。そのため、クラス内のinitializeメソッド実装で、スーパークラスへinitializeメッセージを送信する必要はありません。

ルートクラスのメソッド

クラスオブジェクト、インスタンスオブジェクトを問わず、オブジェクトはすべて、ランタイムシステムに対するインターフェイスが必要です。クラスオブジェクトとインスタンスはどちらも、その能力についてのイントロスペクションを可能にし、継承階層における位置を報告できる必要があります。このインターフェイスを提供するのはNSObjectクラスの役割です。

NSObjectのメソッドを二度（一度はインスタンスにランタイムインターフェイスを提供するため、もう一度はそのインターフェイスをクラスオブジェクトにコピーするため）実装する必要がないように、クラスオブジェクトには、ルートクラスで定義されているインスタンスメソッドを実行する特別許可が与えられます。クラスオブジェクトがクラスメソッドで応えられないメッセージを受信すると、ランタイムシステムは、メッセージに応えられるルートインスタンスメソッドがあるかどうかを調べます。クラスオブジェクトが実行できるインスタンスメソッドは、ルートクラスに定義されているものだけであり、指定の作業を実行できるクラスメソッドがない場合にのみ実行できます。

ルートインスタンスメソッドを実行する、クラスオブジェクトの特別な能力の詳細については、Foundationフレームワークリファレンスの`NSObject`クラス仕様を参照してください。

ソースコードにおけるクラス名

ソースコードでは、まったく異なる2つのコンテキストでのみクラス名を使用することができます。これらのコンテキストは、データ型およびオブジェクトとしてのクラスの、二重の役割を反映しています。

- クラス名は、オブジェクトの種類を示す型名として使用することができます。たとえば、次のようになります。

  ```
  Rectangle *anObject;
  ```

  この場合、`anObject`は、Rectangleのポインタとなるように静的に型定義されています。コンパイラは、対象がRectangleインスタンスのデータ構造と、Rectangleクラスで定義されそこから継承したインスタンスメソッドを持っているものと想定します。静的な型定義により、コンパイラの型チェックを強化し、ソースコードの自己文書化をさらに進めることができます。詳細については、「静的な動作の実現」（99 ページ）を参照してください。

  静的に型定義できるのはインスタンスだけです。クラスオブジェクトは、クラスのメンバではなくClassデータ型に属するため、静的に型定義できません。

- メッセージ式のレシーバとしてのクラス名は、クラスオブジェクトを表します。このような使用法は、これまでのいくつかの例で示しました。クラス名は、メッセージのレシーバとしてのみクラスオブジェクトを表すことができます。それ以外のコンテキストでは、クラスオブジェクトに（`class`メッセージを送信して）`id`を明らかにするように要求する必要があります。次の例では、Rectangleクラスを`isKindOfClass:`メッセージの引数として渡しています。

  ```
  if ( [anObject isKindOfClass:[Rectangle class]] )
      ...
  ```

引数として単純に「Rectangle」という名前を使用するのは正しくありません。このクラス名はレシーバとしてのみ指定できます。

コンパイル時にクラス名が分からなくても、実行時に文字列として持っていれば、`NSClassFromString`を使用してクラスオブジェクトを返すことができます。

```
NSString *className;
    ...
if ( [anObject isKindOfClass:NSClassFromString(className)] )
    ...
```

渡された文字列が有効なクラス名でない場合、この関数は`nil`を返します。

クラス名は、グローバル変数や関数名と同じネームスペースに存在します。クラスとグローバル変数は、同じ名前を持つことができません。クラス名は、Objective-Cでグローバルに認識できるほぼ唯一の名前です。

クラスの等価性のテスト

ポインタを直接比較することによって、2つのクラスオブジェクトが等しいかどうかをテストできます。ただし、適切なクラスを取得することが重要です。Cocoaフレームワークには、機能を拡張するために既存のクラスのサブクラスを動的かつ透過的に作成する機能がいくつかあります（たとえば、キー値監視やCore Dataなど。これらの機能については、『*Key-Value Observing Programming Guide*』および『*Core Data Programming Guide*』をそれぞれ参照してください）。このようなサブクラス化が生じると、一般的に`class`メソッドがオーバーライドされ、動的に作成されたサブクラスが元のクラスのように振る舞います。したがって、クラスの等価性をテストする場合は、低レベルの関数の戻り値ではなく、`class`メソッドの戻り値を比較する必要があります。APIの構文で表すと次のようになります。

```
[object class] != object_getClass(object) != *((Class*)object)
```

したがって、2つのクラスが等しいかどうかは、次のようにしてテストすべきです。

```
if ([objectA class] == [objectB class]) { //...
```

クラスの定義

オブジェクト指向プログラミングの大部分は、新しいオブジェクトのコードを書くこと、つまり新しいクラスを定義することに費やされます。Objective-Cでは、クラスを2つに分けて定義します。

- クラスのメソッドとインスタンス変数を宣言し、そのスーパークラスを指定する**インターフェイス**
- 実際にクラスを定義する**実装**（メソッドを実装するコードを含む）

これらは通常2つのファイルに分けられていますが、クラス定義は「カテゴリ」と呼ばれる機能を使用することで、複数のファイルにまたがる場合もしばしばあります。カテゴリによって、クラス定義の区分や、既存のクラス定義の拡張を行うことができます。カテゴリについては「カテゴリと拡張」（85 ページ）で説明します。

ソースファイル

コンパイラによって要求されているわけではありませんが、インターフェイスと実装は、通常2つのファイルに分けられています。インターフェイスファイルは、クラスを使用する全員が利用できるようにしなければなりません。

1つのファイルで、複数のクラスを宣言または実装することができます。しかし、クラスごとに別々のインターフェイスファイルを持つのが通例で、実装ファイルも別々です。クラスインターフェイスを別々にしておくことは、それらが互いに独立の構成要素であることをより的確に反映します。

インターフェイスファイルと実装ファイルには、通常、クラスにちなんだ名前を付けます。実装ファイルの名前には、Objective-Cのソースコードを含んでいることを示す拡張子`.m`が付けられます。インターフェイスファイルには、ほかの任意の拡張子を割り当てることができます。インターフェイスファイルはほかのソースファイルにインクルードされるため、通常、その名前にはヘッダファイルの典型的な拡張子である`.h`が付けられます。たとえば、Rectangleクラスは、`Rectangle.h`で宣言され、`Rectangle.m`で定義されます。

オブジェクトのインターフェイスと実装を分けることは、オブジェクト指向プログラムの設計によく合致します。オブジェクトは自己完結型の構成要素であり、外部からはほとんど「ブラックボックス」と見なすことができます。プログラムのほかの要素に対するオブジェクトの対話方法をいったん決めたら（つまり、インターフェイスを宣言したら）、アプリケーションのほかの部分に影響を与えることなく、その実装を自由に変更することができます。

クラスインターフェイス

クラスインターフェイスの宣言は、コンパイラディレクティブ`@interface`で始まり、ディレクティブ `@end`で終わります（コンパイラに対するObjective-Cのディレクティブはすべて「@」で始まります）。

```
@interface ClassName :ItsSuperclass
{
    instance variable declarations
}
method declarations
@end
```

宣言の1行目では、新しいクラス名を指定し、それをスーパークラスにリンクします。「継承」（24 ページ）で説明したように、スーパークラスによって継承階層における新しいクラスの位置が決まります。コロンとスーパークラス名を省略すると、新しいクラスはルートクラス（NSObjectクラスのライバル）として宣言されます。

クラス宣言の最初の部分に続いて、**インスタンス変数**の宣言が大括弧で囲まれています。インスタンス変数は、クラスの各インスタンスの一部をなすデータ構造です。次に、Rectangleクラスで宣言できるインスタンス変数の一部を示します。

```
float width;
float height;
BOOL filled;
NSColor *fillColor;
```

次に、インスタンス変数を囲んだ大括弧の後から、クラス宣言の終わりまでの間に、クラスのメソッドを宣言します。クラスオブジェクトに使用されるメソッドの名前、つまり**クラスメソッド**の前にはプラス記号を付けます。

```
+ alloc;
```

クラスのインスタンスが使用できるメソッド、つまり**インスタンスメソッド**の前にはマイナス記号を付けます。

```
- (void)display;
```

一般的な方法ではありませんが、クラスメソッドとインスタンスメソッドを同じ名前で定義することができます。メソッドに、インスタンス変数と同じ名前を付けることもできます。これが一般的なのは、特にメソッドが変数の値を返す場合です。たとえば、Circleが`radius`インスタンス変数に一致する`radius`メソッドを持つことができます。

メソッドの戻り型は、標準Cの型キャストの構文を使って宣言します。

```
- (float)radius;
```

引数型も同じ方法で宣言します。

```
- (void)setRadius:(float)aRadius;
```

戻り型や引数型を明示的に宣言しないと、メソッドやメッセージのデフォルト型、`id`と見なされます。前述の`alloc`メソッドは`id`を返します。

複数の引数がある場合は、メソッド名の中でコロンの後に引数を宣言します。引数は、メッセージの場合と同様に、宣言でも名前が区切られます。たとえば、次のようになります。

```
- (void)setWidth:(float)width height:(float)height;
```

可変引数を持つメソッドは、関数と同様に、コンマと省略記号を使って引数を宣言します。

```
- makeGroup:group, ...;
```

インターフェイスのインポート

インターフェイスファイルは、当該クラスインターフェイスに依存するすべてのソースモジュールにインクルードする必要があります。対象となるソースモジュールとしては、当該クラスのインスタンスを作成したり、クラスに宣言したメソッドを呼び出すメッセージを送信したり、クラスで宣言したインスタンス変数を記述するモジュールがあります。インターフェイスは、通常、`#import`ディレクティブでインクルードされます。

```
#import "Rectangle.h"
```

このディレクティブは`#include`と同じですが、同じファイルが2回以上はインクルードされないことが保証されています。そのため、使用が推奨されており、すべてのObjective-C関連ドキュメントのコード例の中で、`#include`の代わりに使用されています。

クラス定義が派生クラスの定義に基づいて構築されることを反映して、インターフェイスファイルはスーパークラスのインターフェイスをインポートすることで始まります。

```
#import "ItsSuperclass.h"

@interface ClassName :ItsSuperclass
{
    instance variable declarations
}
method declarations
@end
```

この規則は、あらゆるインターフェイスファイルが、すべての派生クラスのインターフェイスファイルを間接的にインクルードすることを意味します。ソースモジュールがあるクラスインターフェイスをインポートすると、そのクラスのベースとなっている継承階層全体のインターフェイスが得られます。

スーパークラスをサポートする*precomp*（プリコンパイルされたヘッダ）がある場合は、代わりにprecompをインポートすることもできます。

ほかのクラスの参照

インターフェイスファイルでクラスを宣言すると、そのスーパークラスをインポートすることで、`NSObject`からスーパークラスに至るまで、すべての派生クラスの宣言を暗黙のうちに含みます。インターフェイスがその階層以外のクラスを記述している場合は、それらを明示的にインポートするか、`@class`ディレクティブで宣言する必要があります。

```
@class Rectangle, Circle;
```

このディレクティブは、「Rectangle」と「Circle」がクラス名であることをコンパイラに知らせるだけです。インターフェイスファイルをインポートするものではありません。

インスタンス変数、戻り値、および引数を静的に型定義するときに、インターフェイスファイルにクラス名を記述します。たとえば、次の宣言をご覧ください。

```
- (void)setPrimaryColor:(NSColor *)aColor;
```

この宣言には、`NSColor`クラスが記述されています。

このような宣言は、単にクラス名を型として使用しているだけで、クラスインターフェイスの詳細（メソッドとインスタンス変数）には依存しないため、引数として何が期待されているかをコンパイラに予告するには@classディレクティブで十分です。しかし、クラスのインターフェイスを実際に使用する場面では（インスタンスの作成、メッセージの送信）、クラスインターフェイスをインポートする必要があります。通常、インターフェイスファイルで@classを使ってクラスを宣言し、（それらのクラスのインスタンスを作成したり、メッセージを送信する必要があるため）対応する実装ファイルでそれらのインターフェイスをインポートします。

@classディレクティブは、コンパイラとリンカによって参照されるコードの量を最小限に抑えるため、クラス名の前方宣言を行う最も簡潔な方法です。簡潔であるため、ほかのファイルをインポートするファイルのインポートに伴う潜在的な問題が回避されます。たとえば、あるクラスが別のクラスの静的に型定義されたインスタンス変数を宣言していて、それぞれのインターフェイスファイルが互いをインポートすると、どちらのクラスも正しくコンパイルされない可能性があります。

インターフェイスの役割

インターフェイスファイルの目的は、新しいクラスをほかのソースモジュール（およびほかのプログラマ）に対して宣言することです。インターフェイスファイルには、クラスを使用するのに必要なすべての情報が含まれています（いくらか文書化されていればプログラマにも歓迎されるでしょう）。

- インターフェイスファイルは、クラスが継承階層にどのように結び付いていて、どのようなクラスがほかに必要となるか（継承するか、クラスのどこかで参照するか）をユーザに知らせます。
- また、インターフェイスファイルはオブジェクトに含まれているインスタンス変数をコンパイラに知らせ、サブクラスが継承している変数をプログラマに知らせます。インスタンス変数は、インターフェイスではなくクラスの実装の問題と考えるのが最も自然ですが、それでもやはり、インターフェイスファイルで宣言する必要があります。これは、コンパイラが、オブジェクトが定義されている場所だけでなく、オブジェクトが使用される場所でもオブジェクトの構造を認識している必要があるからです。しかし、一般にプログラマは、サブクラスを定義する場合を除いて、使用するクラスのインスタンス変数を無視することができます。
- 最後に、メソッド宣言のリストを通して、インターフェイスファイルはほかのモジュールに、どのようなメッセージをクラスオブジェクトとクラスインスタンスに送信できるかを知らせます。クラス定義の外部で使用できるすべてのメソッドを、インターフェイスファイルで宣言します。クラス実装の内部で使用するメソッドは省略できます。

クラス実装

クラスの定義は、宣言と非常によく似た構造になります。@implementationディレクティブで始まり、@endディレクティブで終わります。

```
@implementation ClassName :ItsSuperclass
{
    instance variable declarations
}
method definitions
@end
```

ただし、すべての実装ファイルは、自身のインターフェイスをインポートする必要があります。たとえば、Rectangle.mはRectangle.hをインポートします。実装はインポートする宣言を繰り返す必要がないため、次のものは省略しても支障ありません。

- スーパークラスの名前
- インスタンス変数の宣言

これで実装が簡素化されるため、大部分をメソッドの定義に割けます。

```
#import "ClassName.h"

@implementation ClassName
method definitions
@end
```

クラスのメソッドは、C関数のように一対の中括弧内に定義します。中括弧の前には、インターフェイスファイルの場合と同じ方法でメソッドを宣言しますが、セミコロンは不要です。たとえば、次のようになります。

```
+ (id)alloc
{
    ...
}

- (BOOL)isFilled
{
    ...
}

- (void)setFilled:(BOOL)flag
{
    ...
}
```

可変引数を持つメソッドは、関数と同様に引数を処理します。

```
#import <stdarg.h>

 ...

- getGroup:group, ...
{
    va_list ap;
    va_start(ap, group);
    ...
}
```

インスタンス変数の参照

デフォルトでは、インスタンスメソッドの定義は、有効範囲内にあるオブジェクトのインスタンス変数をすべて持っています。インスタンスメソッドは、名前だけでインスタンス変数を参照することができます。コンパイラはインスタンス変数を格納するためにCの構造体と同等のものを作成し

ますが、構造体の詳細は隠されています。オブジェクトのデータを参照するのに、どちらの構造体演算子（.または->）も必要ありません。たとえば、次のメソッド定義はレシーバのfilledインスタンス変数を参照します。

```
- (void)setFilled:(BOOL)flag
{
    filled = flag;
    ...
}
```

受信側オブジェクトも、そのfilledインスタンス変数も、このメソッドの引数として宣言されていませんが、このインスタンス変数は有効範囲内に入っています。このようなメソッド構文の簡素化によって、Objective-Cコードの記述は非常に簡潔で分かり易くなっています。

インスタンス変数がレシーバでないオブジェクトに属する場合は、オブジェクトの型を静的な型定義によってコンパイラに明示しなければなりません。静的に型定義したオブジェクトのインスタンス変数を参照するには、構造体ポインタ演算子（->）を使用します。

たとえば、Siblingクラスで静的に型定義したオブジェクトtwinをインスタンス変数として宣言するとします。

```
@interface Sibling :NSObject
{
    Sibling *twin;
    int gender;
    struct features *appearance;
}
```

静的に型定義したオブジェクトのインスタンス変数がクラスの有効範囲内にあれば（twinが同じクラスに型定義されているため、この例ではインスタンス変数が有効範囲内にあります）、Siblingメソッドはインスタンス変数を直接設定することができます。

```
- makeIdenticalTwin
{
    if ( !twin ) {
        twin = [[Sibling alloc] init];
        twin->gender = gender;
        twin->appearance = appearance;
    }
    return twin;
}
```

インスタンス変数の有効範囲

インスタンス変数はクラスインターフェイスで宣言しますが、それらは、クラスを使用する方法よりも、クラスを実装する方法の問題です。オブジェクトのインターフェイスは、内部データ構造ではなく、そのメソッドを基盤にします。

多くの場合、メソッドとインスタンス変数は、次の例に示すように1対1で対応しています。

```
- (BOOL)isFilled
{
    return filled;
}
```

しかし、必ずしもそうである必要はありません。インスタンス変数に格納されていない情報を返すメソッドもありますし、オブジェクトが公開しない情報を格納するインスタンス変数もあります。

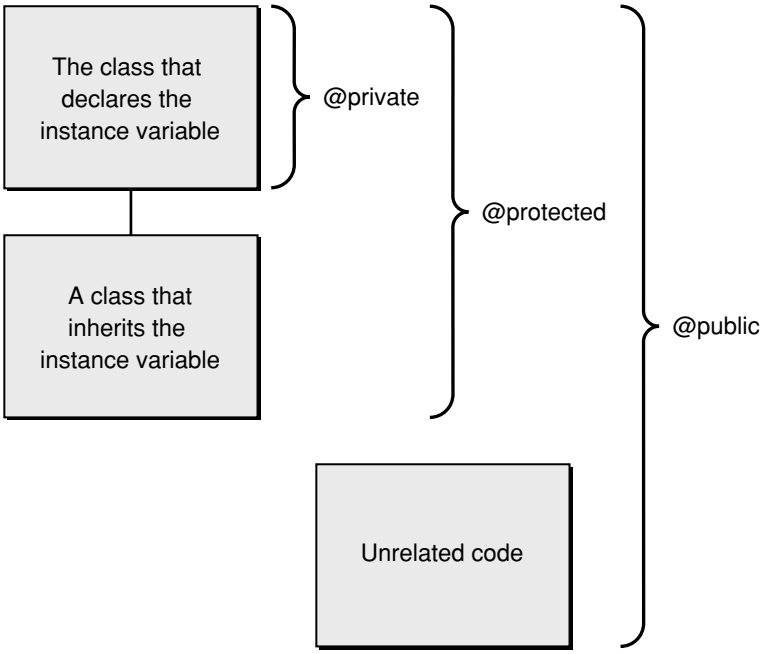
クラスにその時々で変更を加えていくとき、クラスで宣言するメソッドは同じままでも、使用するインスタンス変数は変わる可能性があります。メッセージがクラスのインスタンスと対話するための伝達手段であるかぎり、このような変化は実際にはインターフェイスに影響しません。

オブジェクトがそのデータを隠すことができるように、コンパイラはインスタンス変数の有効範囲を制限します。つまり、プログラム内でのインスタンス変数の可視性を制限します。しかし、柔軟性を提供するために、有効範囲を3段階で明示的に設定することもできます。各段階はコンパイラディレクティブで指定します。

ディレクティブ	意味
`@private`	インスタンス変数を宣言するクラス内でのみ当該変数にアクセス可能。
`@protected`	インスタンス変数を宣言するクラス内および継承するクラス内で当該変数にアクセス可能。
`@public`	対象インスタンス変数にどこからでもアクセス可能。
`@package`	最新のランタイムを使用すると、`@package`インスタンス変数は、クラスを実装するイメージ内では`@public`のように振舞い、クラスを実装するイメージの外側では`@private`のように振舞います。 これは変数と関数の`private_extern`に似ています。クラス実装のイメージの外側のコードからインスタンス変数を使用しようとすると、すべてリンクエラーとなります。これはフレームワーククラス内のインスタンス変数に最も役立ちます。フレームワーククラスでは、`@private`では制限が厳しすぎるが、`@protected`や`@public`では制限が緩すぎるという場合があります。

これを図 2-1に図示します。

図 2-1 インスタンス変数の有効範囲



ディレクティブは、それ以降、次のディレクティブまたはリストの終わりまでの間に記述されたインスタンス変数に適用されます。次の例では、`age`および`evaluation`インスタンス変数は@private、`name`、`job`、および`wage`は@protected、`boss`は@publicです。

```
@interface Worker :NSObject
{
    char *name;
@private
    int age;
    char *evaluation;
@protected
    id job;
    float wage;
@public
    id boss;
}
```

デフォルトでは、無指定のインスタンス変数（上記の`name`など）はすべて`@protected`です。

クラスで宣言するインスタンス変数はすべて、どのような指定をされていても、クラス定義の有効範囲内にあります。たとえば、上記のWorkerクラスのように、`job`インスタンス変数を宣言するクラスは、メソッド定義で当該変数を参照することができます。

```
- promoteTo:newPosition
{
    id old = job;
    job = newPosition;
    return old;
}
```

言うまでもなく、クラスが自身のインスタンス変数にアクセスできなければ、インスタンス変数は何の意味もありません。

通常は、クラスは継承したインスタンス変数にもアクセスできます。インスタンス変数を参照する能力は、通常、変数とともに継承されます。クラスがデータ構造全体をその有効範囲内に保つことは、クラス定義を継承元のクラスを詳細化するものと考えている場合には特に意味があります。上記の`promoteTo:`メソッドは、Workerクラスから`job`インスタンス変数を継承するクラスであれば同様に定義することができます。

しかし、継承先のクラスによるインスタンス変数への直接アクセスを制限するべき場合があるそれなりの理由があります。

- サブクラスの中で継承したインスタンス変数にアクセスすると、当該変数を宣言するクラスがサブクラスの実装の一部に縛られるようになります。後のバージョンで、サブクラスを不用意に壊すことなく当該変数をなくしたり、その役割を変更することはできません。
- さらに、サブクラスにおいて継承したインスタンス変数にアクセスしてその値を変更すると、特に変数がクラス内部の依存関係に関わっている場合は、変数を宣言したクラスに不用意にバグが持ち込まれる可能性があります。

インスタンス変数の有効範囲を、当該変数を宣言するクラスに限定するには、そのインスタンス変数を`@private`として指定する必要があります。`@private`として指定されたインスタンス変数は、パブリックアクセサメソッドが存在する場合に、それらを呼び出すことによってのみサブクラスから利用できます。

逆に、変数を`@public`として指定すると、その変数を継承したり宣言したりするクラス定義の外でも広く利用可能になります。通常、ほかのオブジェクトがインスタンス変数内の情報を取得するには、情報を要求するメッセージを送信する必要があります。しかし、パブリックインスタンス変数は、C構造体のフィールドであるかのように、どこからでもアクセスすることができます。たとえば、次のようになります。

```
Worker *ceo = [[Worker alloc] init];
ceo->boss = nil;
```

オブジェクトは静的に型定義する必要があることに注意してください。

インスタンス変数を`@public`として指定すると、オブジェクトによる当該データの隠蔽が無効になります。これは、表示や不用意な間違いからデータを保護するためオブジェクト内にカプセル化するという、オブジェクト指向プログラミングの原則に反します。したがって、特別な場合を除いて、パブリックインスタンス変数の使用は避けるべきです。

selfとsuperに対するメッセージ

Objective-Cでは、メソッドを実行するオブジェクトを参照するためにメソッド定義内で使用できる2つのキーワード、`self`と`super`が提供されています。

たとえば、操作対象となるすべてのオブジェクトの座標を変更する必要がある、`reposition`メソッドを定義するとします。このメソッドは、変更を行う`setOrigin::`メソッドを呼び出すことができます。必要な処理は、`reposition`メッセージ自体の送信先と同じオブジェクトに`setOrigin::`メッセージを送信することだけです。repositionのコードを記述する際には、そのオブジェクトを`self`または`super`のいずれかとして参照することができます。`reposition`メソッドは次のいずれかのように記述できます。

```
- reposition
{
```

```
    ...
    [self setOrigin:someX :someY];
    ...
}
```

または

```
- reposition
{
    ...
    [super setOrigin:someX :someY];
    ...
}
```

この場合、どのようなオブジェクトであっても、selfとsuperはどちらもrepositionメッセージを受信するオブジェクトを参照します。ただし、この2つのキーワードはまったく異なるものです。selfは、メッセージングルーチンがすべてのメソッドに渡す隠し引数の1つであり、インスタンス変数の名前と同じように、メソッド実装内で自由に使用できるローカル変数です。superは、メッセージ式のレシーバとして使われる場合にのみselfの代わりに使用できるキーワードです。レシーバとして、この2つのキーワードは、主にメッセージング処理に与える影響が異なります。

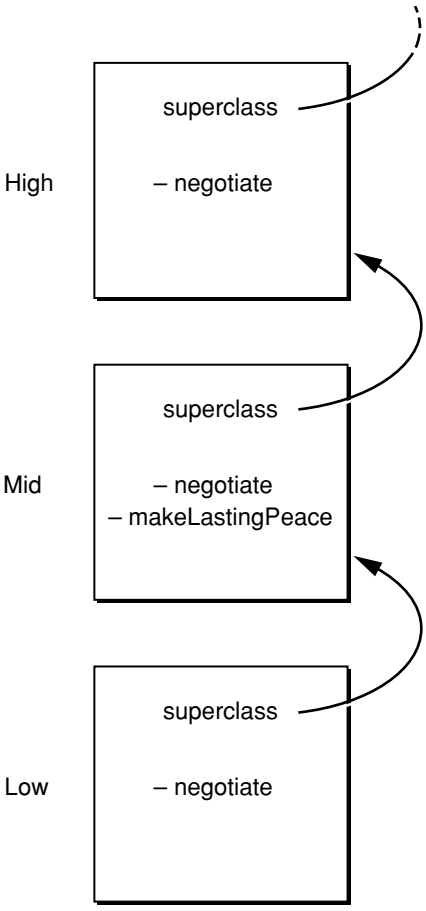
- selfは受信側オブジェクトのクラスのディスパッチテーブルから始まり、通常の方法でメソッド実装を検索します。上記の例では、repositionメッセージを受信するオブジェクトのクラスから検索を始めます。
- superは、まったく異なる場所でメソッド実装の検索を開始します。検索は、superが出現するメソッドを定義しているクラスのスーパークラスから始まります。上記の例では、repositionが定義されているクラスのスーパークラスから検索が始まります。

superがメッセージを受信した場合は常に、コンパイラはobjc_msgSend関数の代わりに別のメッセージングルーチンを使用します。この代替ルーチンは、メッセージを受信したオブジェクトのクラスではなく、定義クラスのスーパークラス（superにメッセージを送信したクラスのスーパークラス）を直接参照します。

例

selfとsuperの違いは、3つのクラスの階層で明確になります。たとえば、Lowというクラスに属するオブジェクトを作成するとします。Lowのスーパークラスは Midで、Midのスーパークラスは High です。3つのクラスすべてにnegotiateというメソッドを定義し、さまざまな用途に使用します。さらに、Midにはは makeLastingPeaceという高度なメソッドを定義します。このメソッドはnegotiate メソッドも必要とします。図 2-2にこれを図示します。

図 2-2 High、Mid、Low



ここで`makeLastingPeace`メソッドを実行するために、Lowオブジェクトにメッセージを送信します。すると、`makeLastingPeace`は同じLowオブジェクトに`negotiate`メッセージを送信します。ソースコードの中で送信対象のオブジェクトを`self`にすると、次のようになります。

```
- makeLastingPeace
{
    [self negotiate];
    ...
}
```

この場合、メッセージングルーチンはLow（`self`のクラス）に定義されているバージョンの`negotiate`を探します。一方、Midのソースコードの中で送信対象のオブジェクトを`super`にすると、次のようになります。

```
- makeLastingPeace
{
    [super negotiate];
    ...
}
```

この場合、メッセージングルーチンは、Highで定義されているバージョンのnegotiateを探します。makeLastingPeaceはMidで定義されているため、メッセージングルーチンは受信側オブジェクトのクラス(Low)を無視して、Midのスーパークラスにスキップします。どちらのメッセージも、Midバージョンのnegotiateを探しません。

この例に示すように、superは別のメソッドをオーバーライドするメソッドをバイパスする手段を提供します。ここではこれによって、makeLastingPeaceは、元のHighバージョンのものを再定義したMidバージョンのnegotiateを回避することができました。

Midバージョンのnegotiateに到達できないのは欠陥のように見えるかもしれませんが、このような状況ではそれを回避するほうが適切です。

- Lowクラスの作成者が意図的にMidバージョンのnegotiateをオーバーライドして、Lowクラス（およびそのサブクラス）のインスタンスが、再定義されたバージョンのメソッドを代わりに呼び出すようにしています。Lowの設計者は、Lowオブジェクトに継承メソッドを実行させないようにしたわけです。
- superにメッセージを送信することで、Midのmake LastingPeaceメソッドの作成者は、Midバージョンのnegotiate（およびMidを継承するLowなどのクラスで定義するバージョン）を意図的にスキップして、Highクラスに定義されているバージョンを実行するようにしました。Midの設計者は、negotiateのHighバージョンだけを使用するようにしたいわけです。

それでもMidバージョンのnegotiateを使用することはできますが、そのためにはMidインスタンスに直接メッセージを送信する必要があります。

superの使用

superへのメッセージにより、メソッド実装を複数のクラスに分散することができます。既存のメソッドをオーバーライドして変更や追加を行う一方で、元のメソッドをその変更に組む込むことができます。

```
- negotiate
{
    ...
    return [super negotiate];
}
```

処理によっては、継承階層の各クラスで作業の一部を行い、残りの作業についてはメッセージをsuperに渡して処理するメソッドを実装することができます。新たに割り当てられたインスタンスを初期化するinitメソッドは、このように動作するように設計されています。それぞれのinitメソッドは、クラスに定義されているインスタンス変数を初期化する役割を持っています。しかし、初期化の前に、initメッセージをsuperに送信して、継承元のクラスにインスタンス変数を初期化させます。initの各バージョンがこの手続きに従うため、クラスは継承の順序に従ってインスタンス変数を初期化することになります。

```
- (id)init
{
    if (self = [super init]) {
        ...
    }
}
```

イニシャライザメソッドには、ほかにもいくつかの制約があります。詳細については、「オブジェクトの割り当てと初期化」（51 ページ）を参照してください。

また、中核的な機能をスーパークラスで定義された1つのメソッドに集中させ、サブクラスにおいてsuperへのメッセージを使用してそのメソッドを組み込むこともできます。たとえば、インスタンスを作成するすべてのクラスメソッドは、新しいオブジェクトにデータ記憶域を割り当て、isa変数をクラス構造体に初期化する必要があります。これは、通常、NSObjectクラスに定義されているallocメソッドとallocWithZone:メソッドに任されます。別のクラスでこれらのメソッドをオーバーライドする場合も（まれなケース）、そのメソッドでメッセージをsuperに送信することによって基本機能を利用することができます。

selfの再定義

superは実行するメソッドの検索を始める場所をコンパイラに伝える単なるフラグで、メッセージのレシーバとしてのみ使用します。しかし、selfは変数名で、いろいろな方法で使用でき、新しい値を代入することもできます。

クラスメソッドの定義では、まさにそうすることがよくあります。クラスメソッドは多くの場合、クラスオブジェクトではなく、クラスのインスタンスを対象としています。たとえば、多くのクラスメソッドがインスタンスの割り当てと初期化を結合し、多くの場合、同時にインスタンス変数値も設定します。このようなメソッドでは、インスタンスメソッドの場合と同様に、新たに割り当てられたインスタンスにメッセージを送信して、selfインスタンスを呼び出すことも考えられます。しかし、これはエラーになります。selfとsuperは、どちらも受信側オブジェクト（メソッドを実行するように指示するメッセージを取得するオブジェクト）を参照します。インスタンスメソッド内ではselfはインスタンスを参照しますが、クラスメソッド内でselfはクラスオブジェクトを参照します。次の例は、してはいけないことを示します。

```
+ (Rectangle *)rectangleOfColor:(NSColor *) color
{
    self = [[Rectangle alloc] init]; // だめ
    [self setColor:color];
    return [self autorelease];
}
```

混乱を避けるために、通常はクラスメソッド内のインスタンスを参照するとき、selfではなく、変数を使用するほうが適切です。

```
+ (id)rectangleOfColor:(NSColor *)color
{
    id newInstance = [[Rectangle alloc] init]; // よい
    [newInstance setColor:color];
    return [newInstance autorelease];
}
```

実際、クラスメソッドの中でクラスにallocメッセージを送信するよりも、allocをselfに送信するほうが有効です。こうしておけば、クラスをサブクラス化し、サブクラスがrectangleOfColor:メッセージを受信した場合、返されるインスタンスはそのサブクラスと同じ型になります（たとえば、NSArrayのarrayメソッドは、NSMutableArrayによって継承されます）。

```
+ (id)rectangleOfColor:(NSColor *)color
{
    id newInstance = [[self alloc] init]; // 最良
    [newInstance setColor:color];
    return [newInstance autorelease];
}
```

オブジェクト割り当ての詳細については、「オブジェクトの割り当てと初期化」（51 ページ）を参照してください。

オブジェクトの割り当てと初期化

オブジェクトの割り当てと初期化

Objective-Cを使ってオブジェクトを作成するには、次の2つのステップを実行する必要があります。

- 新しいオブジェクトに動的にメモリを割り当てる
- 新たに割り当てられたメモリを適切な値に初期化する

2つのステップを完了するまで、オブジェクトは完全には機能しません。各ステップを実行するのは別々のメソッドですが、通常は1行のコードに記述します。

```
id anObject = [[Rectangle alloc] init];
```

割り当てと初期化を分離すると、各ステップを個別に制御できるため、それぞれを他方から切り離して変更することができます。以降のセクションでは、割り当てと初期化の順に説明し、それらを制御、変更する方法について説明します。

Objective-Cでは、`NSObject`クラスで定義されたクラスメソッドを使用して、新しいオブジェクトのメモリを割り当てます。このために、`NSObject`では2つの主要なメソッド、`alloc`と`allocWithZone:`を定義しています。

これらのメソッドは、受信側クラスに属するオブジェクトに、すべてのインスタンス変数を格納するために十分なメモリを割り当てます。これらのメソッドを、サブクラスでオーバーライドして変更する必要はありません。

`alloc`および`allocWithZone:`メソッドは、新たに割り当てられたオブジェクトの`isa`インスタンス変数を初期化して、オブジェクトのクラス（クラスオブジェクト）を指すようにします。ほかのインスタンス変数は、すべて`0`に設定されます。通常、オブジェクトは明確に初期化しないと、安全に使用できません。

このような初期化は、慣例により、省略形「init」で始まるクラス固有のインスタンスメソッドの役割です。メソッドが引数を持たない場合、メソッド名はそれらの4文字のみ、つまり`init`になります。引数を持つ場合、「init」プレフィックスの後に引数のラベルが続きます。たとえば、`NSView`オブジェクトは`initWithFrame:`メソッドで初期化することができます。

インスタンス変数を宣言するどのクラスも、`init...`メソッドを提供して、インスタンス変数を初期化する必要があります。`NSObject`クラスでは、`isa`変数を宣言し、`init`メソッドを定義します。しかし、`isa`はオブジェクトにメモリを割り当てるときに初期化されるため、`NSObject`の`init`メソッドが実行することは`self`を返すことだけです。`NSObject`では、主に前述の命名規則を確立するためにメソッドを宣言します。

返されるオブジェクト

init...メソッドは、通常、レシーバのインスタンス変数を初期化して、それを返します。エラーなしで使用できるオブジェクトを返すのは、このメソッドの役割です。

しかし、場合によっては、この役割は、レシーバではなく別のオブジェクトを返すことを意味する場合もあります。たとえば、命名されたオブジェクトのリストをクラスが保持している場合、新しいインスタンスを初期化するinitWithName:メソッドを提供することができます。同じ名前のオブジェクトが1つしか存在できない場合は、initWithName:で同じ名前を2つのオブジェクトに割り当てることを拒否できます。別のオブジェクトにすでに使用されている名前を新しいインスタンスに割り当てるように要求すると、このメソッドは新たに割り当てられたインスタンスを解放し、他方のオブジェクトを返すことがあります。こうすることで、名前の一意性を確保しながら、要求されたもの、つまり要求された名前の付いたインスタンスを返します。

まれなケースですが、init...メソッドが要求されたことを実行できない場合もあります。たとえば、initFromFile:メソッドは、引数として渡されたファイルから必要なデータを取得する場合があります。渡されたファイル名が実際のファイルと一致していないと、初期化を完了することができません。このような場合、init...メソッドはレシーバを解放して、nilを返し、要求されたオブジェクトを作成できないことを示せます。

init...メソッドは新たに割り当てられたレシーバ以外のオブジェクトを返したり、nilを返したりすることがあるため、プログラムではallocまたはallocWithZone:が返す値だけでなく、初期化メソッドが返す値を使用することが重要です。次のコードはinitが返す値を無視しているため、非常に危険です。

```
id anObject = [SomeClass alloc];
[anObject init];
[anObject someOtherMessage];
```

この代わりに、オブジェクトを安全に初期化するには、割り当ておよび初期化メッセージを1行のコードに結合する必要があります。

```
id anObject = [[SomeClass alloc] init];
[anObject someOtherMessage];
```

init...メソッドがnilを返す可能性がある場合は（「初期化エラーの処理」（54 ページ）を参照）、先へ進む前に戻り値をチェックします。

```
id anObject = [[SomeClass alloc] init];
if ( anObject )
    [anObject someOtherMessage];
else
    ...
```

イニシャライザの実装

新しいオブジェクトを作成するときは、メモリのすべてのビット（isaを除く、すべてのインスタンス変数の値）を0に設定します。オブジェクトを初期化するときに必要な処理は、これだけで十分な場合もあります。しかし、多くの場合、オブジェクトのインスタンス変数にほかのデフォルト

値を与えたり、イニシャライザの引数として値を渡したりします。後者の場合は、カスタムイニシャライザを記述する必要があります。Objective-Cでは、カスタムイニシャライザは、ほかのほとんどのメソッドよりも多くの制約と規則に従わなければなりません。

制約と規則

イニシャライザメソッドには、ほかのメソッドには適用されない制約や規則がいくつかあります。

- 慣習的に、カスタムイニシャライザメソッドの名前は`init`で始まる。

  この例としては、Foundationフレームワークの`initWithFormat:`、`initWithObjects:`、`initWithObjectsAndKeys:`があります。

- イニシャライザメソッドの戻り値の型は`id`とする必要がある。

  その理由は、`id`は、クラスが意図的に想定されていない（呼び出しコンテキストによって、クラスが指定されなかったり、変化したりする）ことを示すためです。たとえば、`NSString`には`initWithFormat:`メソッドがあります。このメッセージを`NSMutableString`（`NSString`のサブクラス）のインスタンスに送信すると、`NSString`ではなく、`NSMutableString`のインスタンスが返されます（「割り当てと初期化の結合」（59 ページ）で示すシングルトンの例も参照してください）。

- カスタムイニシャライザの実装では、最終的には**指定イニシャライザ**を呼び出す必要がある。

  指定イニシャライザについては「指定イニシャライザ」（57 ページ）を参照してください。この問題の網羅的な説明は、「クラスの調整」（56 ページ）で示します。

  要するに、新しい指定イニシャライザを実装する場合は、スーパークラスの指定イニシャライザを呼び出す必要があります。その他のイニシャライザを実装する場合は、そのクラス自身の指定イニシャライザ、または最終的にはその指定イニシャライザを呼び出す別のイニシャライザを呼び出す必要があります。

  デフォルトでは（`NSObject`と同様に）、指定イニシャライザは`init`です。

- イニシャライザの戻り値には`self`を割り当てる必要がある。

  これは、イニシャライザが元のレシーバとは異なるオブジェクトを返す場合があるからです。

- インスタンス変数の値を設定する場合は、通常、アクセサメソッドを使用するのではなく直接代入を使用する。

  これによって、アクセサ内で望まない副作用が生じる可能性を避けることができます。

- イニシャライザでエラーが発生して`nil`を返す場合以外は、イニシャライザの最後で`self`を返す必要がある。

  イニシャライザにおけるエラーについては、「初期化エラーの処理」（54 ページ）で詳しく解説します。

次の例は、`NSObject`を継承し、オブジェクトの作成日時を表す`creationDate`というインスタンス変数を持つクラスのカスタムイニシャライザの実装を示しています。

```
- (id)init {
    // superの指定イニシャライザの戻り値にselfを設定する
    // NSObjectの指定イニシャライザはinit
    if (self = [super init]) {
        creationDate = [[NSDate alloc] init];
    }
    return self;
```

```
}
```

（`if (self = [super init])`パターンを使用した理由については、「初期化エラーの処理」（54 ページ）で解説します。）

イニシャライザでは、変数ごとに引数を提供する必要はありません。たとえば、あるクラスがそのインスタンスに名前とデータソースを持つことを要求する場合は、`initWithName:fromURL:`メソッドを用意することも考えられますが、重要でないインスタンス変数は任意の値に設定したり、デフォルトでNULL値に設定することができます。そして初期化段階が完了した後で、デフォルト値を変更するために、`setEnabled:`、`setFriend:`、および`setDimensions:`のようなメソッドに頼ることができます。

次の例に、1つの引数をとるカスタムイニシャライザの実装を示します。この例では、クラスは`NSView`を継承します。ここでは、スーパークラスの指定イニシャライザを呼び出す前に何らかの処理を実行できることを示します。

```
- (id)initWithImage:(NSImage *)anImage {

    // 画像から新しいインスタンスのサイズを得る
    NSSize size = anImage.size;
    NSRect frame = NSMakeRect(0.0, 0.0, size.width, size.height);

    // superの指定イニシャライザの戻り値にselfを設定する
    // NSViewの指定イニシャライザはinitWithFrame:
    if (self = [super initWithFrame:frame]) {

        image = [anImage retain];
    }
    return self;
}
```

この例では初期化中に問題が発生した場合の処理を示していません。これについては、次のセクションで解説します。

初期化エラーの処理

一般に、初期化メソッドの中で問題が発生した場合は、`[self release]`を呼び出して`nil`を返すべきです。

このポリシーの結果は主に次の2つです。

- イニシャライザメソッドから`nil`を受け取るすべてのクラス（独自のクラス、サブクラス、または外部の呼び出し元であれ）は、それを処理できなければなりません。呼び出し元が呼び出しの前にこのオブジェクトへの外部参照を確立しているような珍しいケースの場合、この処理にはすべての接続の解消が含まれます。
- 部分的に初期化されたオブジェクトが存在しても`dealloc`メソッドが安全であることを保証しなければなりません。

注： `[self release]`は、エラーが発生した時点でのみ呼び出すべきです。スーパークラスのイニシャライザ呼び出しから`nil`が返った場合は、`release`を呼び出すべきではありません。単に、`dealloc`の中で処理していないセットアップ済みの参照をすべてクリーンアップしてから`nil`を返すべきです。通常、これは、スーパークラスのイニシャライザの戻り値のテストに基づくブロック内で、初期化を実行するパターンによって処理されます（前の例を参照）。

```
- (id)init {
    if (self = [super init]) {
        creationDate = [[NSDate alloc] init];
    }
    return self;
}
```

「制約と規則」（53 ページ）の例を基にした次の例は、パラメータとして渡された不適切な値を処理する方法を示しています。

```
- (id)initWithImage:(NSImage *)anImage {

    if (anImage == nil) {
        [self release];
        return nil;
    }

    // 画像から新しいインスタンスのサイズを得る
    NSSize size = anImage.size;
    NSRect frame = NSMakeRect(0.0, 0.0, size.width, size.height);

    // superの指定イニシャライザの戻り値にselfを設定する
    // NSViewの指定イニシャライザはinitWithFrame:
    if (self = [super initWithFrame:frame]) {

        image = [anImage retain];
    }
    return self;
}
```

次の例は、問題が発生した場合に、参照で返される`NSError`オブジェクトの形式で有用な情報を返すことができるというベストプラクティスを示しています。

```
- (id)initWithURL:(NSURL *)aURL error:(NSError **)errorPtr {

    if (self = [super init]) {

        NSData *data = [[NSData alloc] initWithContentsOfURL:aURL
                                        options:NSUncachedRead error:errorPtr];

        if (data == nil) {
            // この場合はNSDataのイニシャライザでエラーオブジェクトが作成される
            [self release];
            return nil;
        }
        // 実装が続く...
```

通常、この種のエラーを表すために例外を使用するべきではありません。詳細については、『*Error Handling Programming Guide For Cocoa*』を参照してください。

クラスの調整

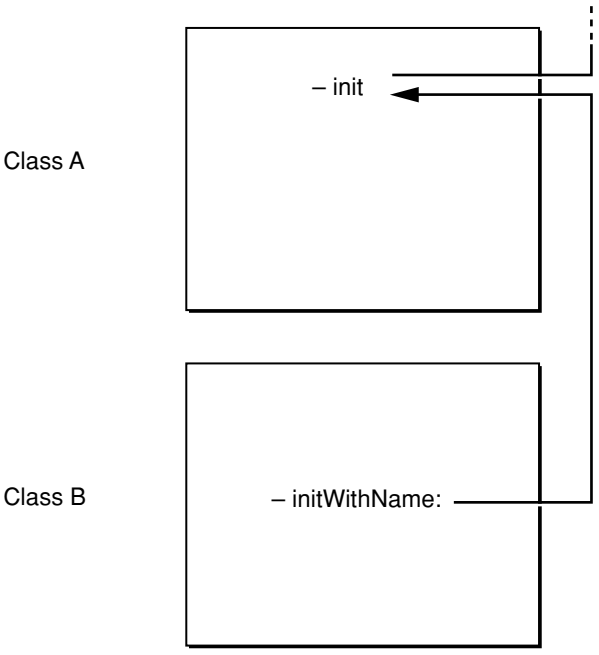
クラスで定義するinit...メソッドは、通常、そのクラスで宣言されている変数のみを初期化します。継承したインスタンス変数は、superにメッセージを送信して、継承階層の上位のどこかで定義された初期化メソッドを実行することで初期化します。

```
- (id)initWithName:(NSString *)string {
    if ( self = [super init] ) {
        name = [string copy];
    }
    return self;
}
```

superへのメッセージは、継承元のすべてのクラスの初期化メソッドを連鎖させます。これが先に実行されるため、スーパークラスの変数がサブクラスで宣言されている変数よりも先に初期化されることが保証されます。たとえば、RectangleオブジェクトはRectangleとして初期化される前に、NSObject、Graphic、およびShapeとして初期化される必要があります。

前述のinitWithName:メソッドと、そこに組み込まれている継承したinitメソッドとの関係を、図3-1に示します。

図 3-1 継承した初期化メソッドの組み込み

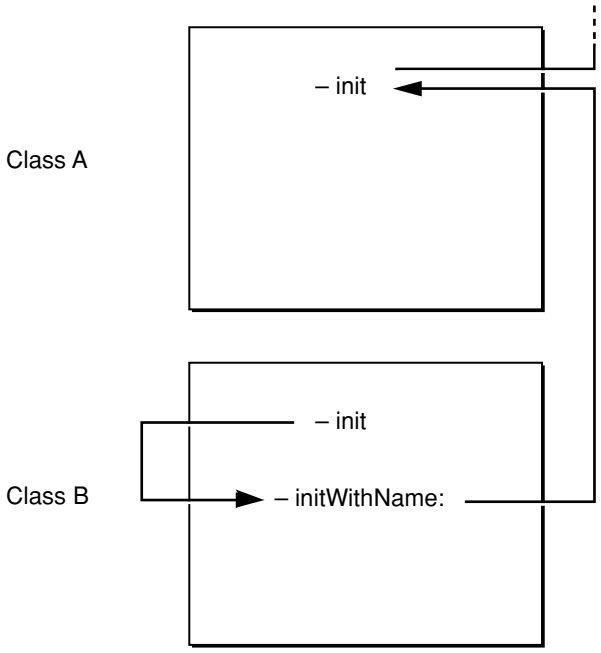


また、クラスでは、継承したすべての初期化メソッドが必ず機能するようにしなければなりません。たとえば、図 3-1で示すように、クラスAでinitメソッドを定義し、そのサブクラスBでinitWithName:メソッドを定義する場合、サブクラスBでもinitメッセージがBのインスタンスを確実に初期化できる必要があります。これを実現する最も簡単な方法は、継承したinitメソッドを、initWithName:を呼び出すバージョンに置き換えることです。

```
- init {
    return [self initWithName:"default"];
}
```

前述のように、initWithName:メソッドも、継承したメソッドを呼び出します。図 3-2は、Bバージョンのinitを示しています。

図 3-2 継承した初期化モデルのオーバーライド



継承した初期化メソッドをオーバーライドすると、定義したクラスをほかのアプリケーションに移植しやすくなります。継承したメソッドをオーバーライドしないでおくと、ほかの誰かがそれを使用して、間違った初期化が行われたクラスのインスタンスを作成する可能性があります。

指定イニシャライザ

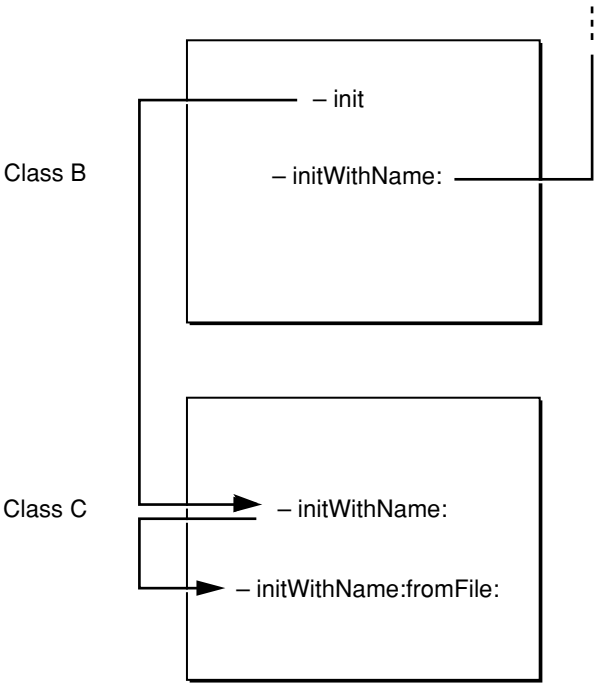
「クラスの調整」（56 ページ）の例では、initWithName:が対象クラス（クラスB）の**指定イニシャライザ**になります。指定イニシャライザは、（superにメッセージを送信して継承元のメソッドを実行することによって）継承したインスタンス変数の初期化を保証する各クラスのメソッドです。指定イニシャライザはまた、作業の大部分を実行するメソッドであり、同じクラスのほかの初期化メソッドが呼び出すメソッドでもあります。指定イニシャライザは常に、新しいインスタンスの特性を決めるための最大の自由を与えるメソッドであるというのがCocoaの規約です（通常、これは最も多くの引数を持つメソッドですが、いつもそうとは限りません）。

サブクラスを定義する際には、指定イニシャライザを知っていることが重要です。たとえば、クラスBのサブクラスとしてクラスCを定義し、initWithName:fromFile:メソッドを実装するとします。このメソッドに加えて、継承したinitおよびinitWithName:メソッドも、Cのインスタンスに対して機能するようにしなければなりません。これを実現するには、initWithName:fromFile:を呼び出すバージョンで、Bのinitwithname:をオーバーライドします。

```
- initWithName:(char *)string {
    return [self initWithName:string fromFile:NULL];
}
```

Cクラスのインスタンスでは、継承したinitメソッドはinitWithName:fromFile:を呼び出す、initWithName:の新しいバージョンを呼び出します。これらのメソッド間の関係を図 3-3に示します。

図 3-3 指定イニシャライザのオーバーライド



この図では、重要な部分が省略されています。initWithName:fromFile:メソッドは、Cクラスの指定イニシャライザであるため、superにメッセージを送信して、継承した初期化メソッドを呼び出します。しかし、Bクラスのメソッドのうち、それが呼び出すのはinitまたはinitWithName:のどちらでしょう。次の2つの理由により、initを呼び出すことはできません。

- 循環が生じる（initがCのinitWithName:を呼び出し、これがinitWithName:fromFile:を呼び出し、さらにこれがinitを再度呼び出します）。
- BクラスのバージョンのinitWithName:の初期化コードを利用することはできない。

したがって、initWithName:fromFile:はinitWithName:を呼び出す必要があります。

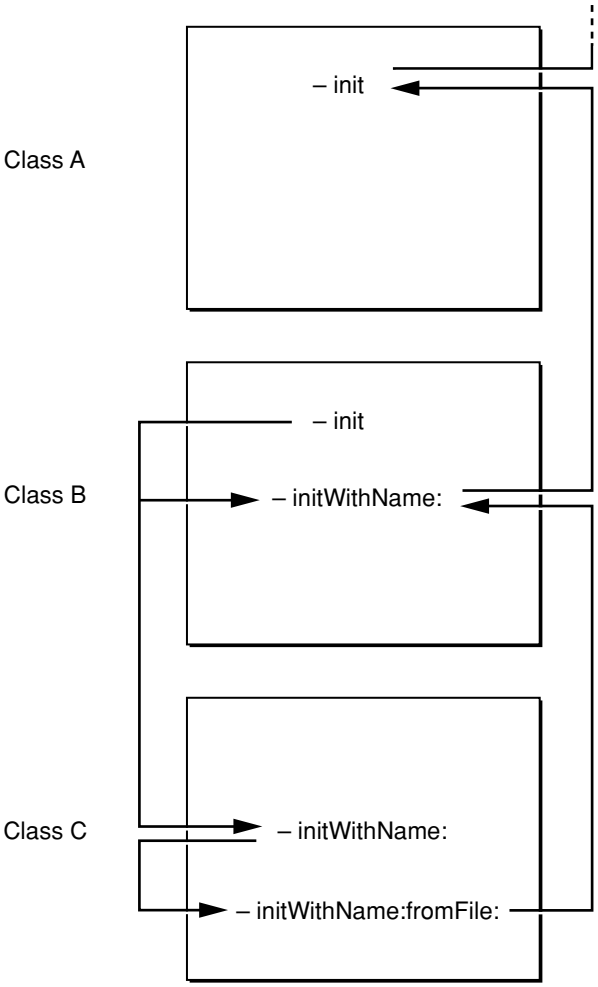
```
- initWithName:(char *)string fromFile:(char *)pathname {
    if ( self = [super initWithName:string] )
        ...
}
```

一般原則： クラスの指定イニシャライザは、superへのメッセージを通じて、スーパークラスの指定イニシャライザを呼び出さなければなりません。

指定イニシャライザはsuperへのメッセージを通して相互に結び付いており、ほかの初期化メソッドはselfへのメッセージを通して指定イニシャライザに結び付いています。

図3-4は、クラスA、B、およびCのすべての初期化メソッドがどのようにリンクしているかを示しています。selfへのメッセージが左側、superへのメッセージが右側に示されています。

図 3-4 初期化チェーン



Bバージョンのinitは、selfにメッセージを送信してinitWithName:メソッドを呼び出すことに注目してください。したがって、レシーバがBクラスのインスタンスである場合は、BバージョンのinitWithName:が呼び出され、レシーバがCクラスのインスタンスである場合は、CのバージョンのinitWithName:が呼び出されます。

割り当てと初期化の結合

Cocoaでは、割り当てと初期化の2つのステップを結合し、クラスの初期化された新しいインスタンスを返す、作成メソッドをいくつかのクラスで定義しています。これらのメソッドは、**簡易コンストラクタ**とも呼ばれ、通常は+ *className*...（*className*はクラス名）の形式をとります。たとえば、NSStringには（ほかを含め）次のメソッドがあります。

```
+ (id)stringWithCString:(const char *)cString encoding:(NSStringEncoding)enc;
```

```
+ (id)stringWithFormat:(NSString *)format, ...;
```

同様に、NSArrayでは、割り当てと初期化を結合する次のクラスメソッドを定義しています。

```
+ (id)array;
+ (id)arrayWithObject:(id)anObject;
+ (id)arrayWithObjects:(id)firstObj, ...;
```

重要： ガベージコレクションを使用しない場合は、これらのメソッドの使用から生じるメモリ管理上の影響を理解することが重要です（「メモリ管理」（15 ページ）を参照）。これらの簡易コンストラクタに適用されるポリシーを理解するには、『*Memory Management Programming Guide for Cocoa*』を読む必要があります。

これらのメソッドの戻り値型がidであることに注意してください。これは、イニシャライザメソッドと同じ理由からです（「制約と規則」（53 ページ）を参照）。

割り当てと初期化を結合するメソッドは、割り当てが何らかの形で初期化によって通知される必要がある場合に特に役立ちます。たとえば、初期化のためのデータをファイルから取得する場合に、そのファイルに複数のオブジェクトを初期化するのに十分なデータが含まれていても、ファイルを開くまではいくつのオブジェクトを割り当てるか分かりません。このような場合は、ファイル名を引数として受け取るlistFromFile:メソッドを実装することができます。このメソッドは、ファイルを開いて、割り当てるオブジェクトの数を確認し、新しいオブジェクトをすべて格納するのに十分な大きさのListオブジェクトを作成します。次に、ファイル内のデータに基づいてオブジェクトを割り当てて初期化し、Listに入れ、最終的にそのListを返します。

また、使用しない可能性のある新しいオブジェクトに対して無駄にメモリを割り当てるステップを回避したい場合は、割り当てと初期化を1つのメソッドに結合することには意味があります。「返されるオブジェクト」（52 ページ）で説明したように、init...メソッドではレシーバの代わりに別のオブジェクトを使用できます。たとえば、すでに割り当てられている名前がinitWithName:に渡されると、レシーバが解放され、その代わりに、先にその名前が割り当てられたオブジェクトを返すことが考えられます。これはもちろん、オブジェクトを割り当て、使用せずにすぐ解放することを意味します。

レシーバを初期化するかどうかを決定するコードが、init...の中ではなく、割り当てを実行するメソッド内にあれば、不要な新しいインスタンスを割り当てるステップを回避することができます。

次の例では、soloistメソッドが、Soloistクラスのインスタンスが1つしかないことを保証しています。このメソッドは単一の共有インスタンスを割り当てて初期化します。

```
+ (Soloist *)soloist {
    static Soloist *instance = nil;

    if ( instance == nil ) {
        instance = [[self alloc] init];
    }
    return instance;
}
```

このケースでは戻り値型がSoloist *であることに注意してください。このメソッドはシングルトン共有インスタンスを返すため、強い型定義が適切です（このメソッドがオーバーライドされることは決してありません）。

プロトコル

プロトコルによって、すべてのクラスが実装できるメソッドを宣言します。プロトコルは少なくとも次の3つの状況で役に立ちます。

- ほかのクラスが実装するものと期待されるメソッドの宣言
- クラスが隠蔽されているオブジェクトのインターフェイスの宣言
- 階層的な関係のないクラス間の類似性の取得

ほかのクラスが実装できるインターフェイスの宣言

クラスおよびカテゴリインターフェイスでは、特定のクラスに関連付けられるメソッド、主にクラスが実装するメソッドを宣言します。これに対して、非形式および形式**プロトコル**は、特定のクラスには依存していないながらも、任意の（おそらく多数の）クラスによって実装される可能性のあるメソッドを宣言します。

プロトコルはメソッド宣言の単なるリストで、クラス定義とは結び付いていません。たとえば、マウスに対するユーザ操作を報告する次のメソッドは、プロトコルにまとめることができます。

```
- (void)mouseDown:(NSEvent *)theEvent;
- (void)mouseDragged:(NSEvent *)theEvent;
- (void)mouseUp:(NSEvent *)theEvent;
```

マウスイベントに応答しなければならないクラスは、このプロトコルを採用して、そのメソッドを実装することができます。

プロトコルはメソッド宣言をクラス階層への依存から解放するため、クラスとカテゴリでは使用できない方法でメソッドを使用できます。プロトコルはどこかに実装されている（またはその可能性のある）メソッドをリストしますが、メソッドを実装するクラスを知る必要がありません。知る必要があるのは、特定のクラスがプロトコルに**準拠する**かどうか、プロトコルに宣言されているメソッドをクラスが実装しているかどうかということです。したがって、同じクラスを継承することによる類似性だけでなく、同じプロトコルに準拠することによる類似性に基づいて、オブジェクトを型に分類することができます。継承階層において互いに関係のない分岐にあるクラスも、同じプロトコルに準拠するため、類似のものとして型定義することができます。

プロトコルはオブジェクト指向設計において重要な役割を果たします。特に、プロジェクトを多数の実装者が分担したり、ほかのプロジェクトで開発されたオブジェクトを組み込んだりする場合に重要となります。Cocoaソフトウェアは、Objective-Cのメッセージを通じたプロセス間通信をサポートするため、プロトコルを大いに利用しています。

しかし、Objective-Cプログラムでは、プロトコルを使用する必要はありません。クラス定義やメッセージ式とは異なり、プロトコルの使用は任意です。Cocoaフレームワークにも、プロトコルを使用するものと、使用しないものがあります。プロトコルを使用するかどうかは、行うべき作業によって決まります。

ほかのクラスが実装するメソッド

オブジェクトのクラスが分かっていれば、そのインターフェイス宣言（および、継承元のクラスのインターフェイス宣言）を参照して、そのオブジェクトが応答する対象となるメッセージを調べることができます。これらの宣言は、受信できるメッセージを提示します。プロトコルは、送信するメッセージを提示する方法も提供します。

通信は双方向で機能し、オブジェクトはメッセージを受信するだけでなく、送信もします。たとえば、あるオブジェクトは特定操作の責任を別のオブジェクトにデリゲートするかもしれませんし、あるいは、別のオブジェクトに情報を要求するだけかもしれません。場合によっては、オブジェクトがその動作をほかのオブジェクトに積極的に通知し、ほかのオブジェクトが必要な措置を取れるようにすることも考えられます。

同じプロジェクトの一部として送信側のクラスとレシーバのクラスを開発する場合（あるいは、ほかからレシーバとそのインターフェイスファイルが提供される場合）、この通信は簡単に調整できます。送信側は単に、レシーバのインターフェイスファイルをインポートするだけです。インポートしたファイルには、送信側が送信するメッセージで使用する、メソッドセレクタが宣言されています。

しかし、まだ定義されていないオブジェクト（ほかの人に実装を任せているオブジェクト）にメッセージを送信するオブジェクトを開発する場合は、レシーバのインターフェイスファイルがありません。メッセージで使用する、自分では実装しないメソッドを宣言するには、別の方法が必要になります。プロトコルは、このような目的に使用できます。プロトコルは、クラスが使用するメソッドをコンパイラに通知し、オブジェクトを連携させるために定義する必要があるメソッドをほかの実装者に知らせます。

たとえば、`helpOut:`などのメッセージを送信することで、別のオブジェクトに支援を要請するオブジェクトを開発するとします。これらのメッセージのアウトレットを記録する`assistant`インスタンス変数を用意し、このインスタンス変数を設定するための付随メソッドを定義します。このメソッドにより、ほかのオブジェクトは、オブジェクトのメッセージの潜在的レシーバとして自身を登録することができます。

```
- setAssistant:anObject
{
    assistant = anObject;
}
```

次に、メッセージを`assistant`に送信するたびに、メッセージに応えるメソッドがレシーバに実装されていることをチェックします。

```
- (BOOL)doWork
{
    ...
    if ( [assistant respondsToSelector:@selector(helpOut:)]) {
        [assistant helpOut:self];
        return YES;
    }
    return NO;
}
```

このコードを記述する時点では、`assistant`としてどのようなオブジェクトが登録されるかは分からないため、`helpOut:`メソッドのプロトコルを宣言することしかできません。このメソッドを実装するクラスのインターフェイスファイルはインポートできません。

匿名オブジェクトのインターフェイスの宣言

プロトコルを使って、**匿名オブジェクト**、つまり未知のクラスのオブジェクトのメソッドを宣言することができます。匿名オブジェクトは、サービスを示したり、限られた数の関数から成るセットを処理することができます。特にその種類のオブジェクトが1つだけ必要な場合に使用します（アプリケーションのアーキテクチャを定義する際に基本的な役割を果たすオブジェクトや、使用する前に初期化しなければならないオブジェクトは、匿名オブジェクトには適していません）。

もちろん、当該オブジェクトのデベロッパにとっては匿名ではありませんが、デベロッパがオブジェクトをほかの誰かに提供するときは匿名です。たとえば、次のような状況があるとします。

- ほかから使用されるフレームワークや一組のオブジェクトを提供するデベロッパは、クラス名やインターフェイスファイルによって識別されないオブジェクトを含めることができます。名前とクラスインターフェイスがないので、ユーザにはクラスのインスタンスを作成する方法がありません。代わりに、供給側は事前に作成しておいたインスタンスを提供する必要があります。通常、別のクラスのメソッドは、使用可能なオブジェクトを返します。

  ```
  id formatter = [receiver formattingService];
  ```

  メソッドによって返されたオブジェクトはクラス識別情報を持たないオブジェクト、少なくとも供給側が積極的に公開するような識別情報を持たないオブジェクトです。しかし、多少なりとも役に立つように、供給側は対応できるメッセージの少なくとも一部を積極的に提示する必要があります。これを行うには、プロトコルで宣言したメソッドのリストとオブジェクトを関連付けます。

- Objective-Cのメッセージは、**リモートオブジェクト**（ほかのアプリケーションのオブジェクト）に送信することができます（詳細については、「リモートメッセージング」（113 ページ）を参照してください）。

  各アプリケーションは、独自の構造、クラス、および内部ロジックを持ちます。しかし、アプリケーションと通信するためにその動作や構成要素を知っている必要はありません。部外者として知っている必要があるのは、送信可能なメッセージ（プロトコル）とメッセージの送信先（レシーバ）だけです。

  リモートメッセージの潜在的レシーバとしてオブジェクトの1つを公開するアプリケーションは、オブジェクトが受信したメッセージに対応するために使用するメソッドを宣言するプロトコルも公開しなければなりません。オブジェクトに関して、ほかには何も公開する必要がありません。送信側アプリケーションは、オブジェクトのクラスを知っていたり、自身の設計の中でそのクラスを使用する必要はありません。必要なのはプロトコルだけです。

プロトコルにより、匿名オブジェクトが可能になります。プロトコルがなければ、クラスを特定せずに、オブジェクトのインターフェイスを宣言する方法はありません。

注： 匿名オブジェクトの供給側はそのクラスを公開しませんが、オブジェクト自体は実行時に明らかになります。classメッセージは匿名オブジェクトのクラスを返します。しかし、通常はこの付加的な情報を知る意味はほとんどなく、プロトコルの情報だけで十分です。

階層以外の類似性

複数のクラスが一組のメソッドを実装する場合、それらのクラスは多くの場合、共通のメソッドを宣言する抽象クラスの下にグループ化されます。各サブクラスは独自の方法でメソッドを再実装できますが、継承階層および抽象クラスの共通の宣言によってサブクラス間の本質的な類似性が確保されます。

しかし、共通のメソッドを抽象クラスにグループ化できないこともあります。それにもかかわらず、ほとんどの点で関連のないクラスが、いくつかの類似メソッドを実装する必要があるかもしれません。このような限られた類似性は、階層関係の正当な理由になりません。たとえば、アプリケーション内のオブジェクトのXML表現を作成し、XML表現からオブジェクトを初期化するためのサポートを追加する場合は次のようになります。

```
- (NSXMLElement *)XMLRepresentation;
- initFromXMLRepresentation:(NSXMLElement *)xmlString;
```

これらのメソッドをプロトコルとしてグループ化し、クラスをすべて同じプロトコルに準拠させることで、それらの類似性を反映できます。

オブジェクトはそれらのクラスではなく、このような類似性（クラスが準拠するプロトコル）に応じて型定義することができます。たとえば、NSMatrixインスタンスはセルを表すオブジェクトと通信しなければなりません。matrixは、これらの各オブジェクトがNSCell（クラスをベースにした型）の一種であることを要求し、NSCellクラスを継承するすべてのオブジェクトがNSMatrixメッセージに応えるために必要なメソッドを持っているものと想定することができます。もう1つの方法として、NSMatrixオブジェクトはセルを表すオブジェクトに、特定のメッセージセットに対応できるメソッドを持つことを要求することができます（プロトコルをベースにした型）。この場合、NSMatrixオブジェクトはセルオブジェクトがメソッドさえ実装していれば、どのようなクラスに属しているかは問題にしません。

形式プロトコル

Objective-C言語には、メソッドのリスト（宣言済みプロパティを含む）をプロトコルとして形式的に宣言する方法があります。形式プロトコルは、言語とランタイムシステムによってサポートされます。たとえば、コンパイラはプロトコルに基づいて型をチェックすることができ、オブジェクトはプロトコルに準拠しているかどうかを実行時にイントロスペクションを実行して報告することができます。

プロトコルの宣言

形式プロトコルは、@protocolディレクティブを使って宣言します。

```
@protocol ProtocolName
```

```
method declarations
@end
```

たとえば、次のようなXML表現プロトコルを宣言できます。

```
@protocol MyXMLSupport
- initFromXMLRepresentation:(NSXMLElement *)XMLElement;
- (NSXMLElement *)XMLRepresentation;
@end
```

クラス名と異なり、プロトコル名にはグローバルな可視性がありません。プロトコル名は自身のネームスペースに属します。

任意のプロトコルメソッド

プロトコルメソッドは、`@optional`キーワードを使用してオプションとして指定できます。`@optional`キーワードに対応して、デフォルトの振る舞いのセマンティクスを形式的に示す`@required`キーワードがあります。`@optional`と`@required`を使用して、プロトコルを適切と思われるセクションに分割できます。キーワードを指定しない場合のデフォルトは、`@required`です。

```
@protocol MyProtocol

- (void)requiredMethod;

@optional
- (void)anOptionalMethod;
- (void)anotherOptionalMethod;

@required
- (void)anotherRequiredMethod;

@end
```

注： Mac OS X v10.5では、プロトコルにオプションの宣言済みプロパティは含まれません。Mac OS X v10.6以降では、この制約はなくなっています。

非形式プロトコル

形式プロトコルのほかに、カテゴリ宣言の中のメソッドをグループ化することで**非形式**プロトコルを定義できます。

```
@interface NSObject ( MyXMLSupport )
- initFromXMLRepresentation:(NSXMLElement *)XMLElement;
- (NSXMLElement *)XMLRepresentation;
@end
```

非形式プロトコルは、`NSObject`を継承する任意のクラスとメソッド名を緩やかに関連付けるため、通常は、`NSObject`クラスのカテゴリとして宣言します。すべてのクラスはルートクラスを継承するため、メソッドの対象は継承階層のどの部分にも限定されません（非形式プロトコルを別のクラスのカテゴリとして宣言し、継承階層の特定の分岐に適用対象を限定することも可能ですが、そうするべき理由はほとんどありません）。

プロトコルの宣言に使用する場合、カテゴリインターフェイスには対応する実装がありません。その代わりに、プロトコルを実装するクラスは、自身のインターフェイスファイルでもう一度メソッドを宣言し、実装ファイルでほかのメソッドと一緒に定義します。

非形式プロトコルはカテゴリ宣言の規則に反して、メソッドのグループをリストアップする一方で、それらを特定のクラスや実装と関連付けません。

カテゴリで宣言したプロトコルは非形式プロトコルであるため、言語によるサポートはほとんど受けません。コンパイル時の型チェックも、オブジェクトがプロトコルに準拠しているかどうかを確認する実行時のチェックもありません。これらのメリットを得るには、形式プロトコルを使用する必要があります。非形式プロトコルは、デリゲートのためなどですべてのメソッドが任意である場合に役立つかもしれませんが、（Mac OS X v10.5以降では）通常は、任意のメソッドにも形式プロトコルを使用するほうがよいでしょう。

Protocolオブジェクト

実行時にクラスがクラスオブジェクトによって表され、メソッドがセレクタコードによって表されるように、形式プロトコルは特別なデータ型、すなわちProtocolクラスのインスタンスによって表されます。プロトコルを処理するソースコード(型指定で使用する場合を除く）は、Protocolオブジェクトを参照する必要があります。

さまざまな意味で、プロトコルはクラス定義と似ています。どちらもメソッドを宣言し、実行時にオブジェクトによって表されます。つまり、クラスはクラスオブジェクトによって、プロトコルはProtocolオブジェクトよって表されます。クラスオブジェクトのように、Protocolオブジェクトはソースコードにある定義と宣言から自動的に作成され、ランタイムシステムによって使用されます。プログラムソースコードでの割り当てと初期化は行われません。

ソースコードでは、`@protocol()`ディレクティブを使用してProtocolオブジェクトを参照することができます。このディレクティブは、プロトコルを宣言するディレクティブと同じですが、後に丸括弧が付いています。この丸括弧にはプロトコル名を入れます。

```
Protocol *myXMLSupportProtocol = @protocol(MyXMLSupport);
```

これは、ソースコードでProtocolオブジェクトを呼び出せる唯一の方法です。クラス名と異なり、プロトコル名はオブジェクトを指定しません（`@protocol()`内は除く）。

コンパイラはプロトコル宣言に遭遇するたびにProtocolオブジェクトを作成しますが、それは次の場合だけです。

- クラスでプロトコルを採用している
- プロトコルが（`@protocol()`を使って）ソースコードのどこかで参照されている

宣言したものの使用されていないプロトコル（後述のように型チェックの場合は除く）は、実行時にProtocolオブジェクトによって表されません。

プロトコルの採用

プロトコルの採用は、ある意味でスーパークラスの宣言に似ています。どちらもメソッドをクラスに割り当てます。スーパークラス宣言は継承メソッドをクラスに割り当て、プロトコルはプロトコルリストに宣言されているメソッドをクラスに割り当てます。クラスが形式プロトコルを**採用**するという場合は、クラスの宣言の中でスーパークラス名の後の不等号括弧内にそのプロトコルがリストされているはずです。

```
@interface ClassName :ItsSuperclass < protocol list >
```

カテゴリもほぼ同じ方法でプロトコルを採用します。

```
@interface ClassName ( CategoryName ) < protocol list >
```

クラスは複数のプロトコルを採用できます。その場合はプロトコルリストにプロトコル名をコンマで区切って指定します。

```
@interface Formatter :NSObject < Formatting, Prettifying >
```

プロトコルを採用するクラスまたはカテゴリは、プロトコルが宣言するすべての必須メソッドを実装しなければなりません。そうしないとコンパイラから警告が発せられます。上記のFormatterクラスでは、自身で宣言したものに加えて、採用した2つのプロトコルで宣言されている必須メソッドをすべて定義します。

プロトコルを採用するクラスまたはカテゴリは、プロトコルを宣言するヘッダファイルをインポートする必要があります。採用したプロトコルで宣言されているメソッドは、クラスまたはカテゴリインターフェイスのほかの場所では宣言されていません。

クラスではプロトコルを採用するだけで、ほかのメソッドを宣言しないことも可能です。たとえば、次のクラス宣言では、FormattingおよびPrettifyingプロトコルを採用していますが、インスタンス変数や自身のメソッドは宣言していません。

```
@interface Formatter :NSObject < Formatting, Prettifying >
@end
```

プロトコルへの準拠

プロトコルを採用したクラス、またはプロトコルを採用した別のクラスを継承したクラスは、形式プロトコルに**準拠**していると言われます。クラスのインスタンスは、そのクラスが準拠しているものと同じプロトコルのセットに準拠していると言います。

クラスは採用するプロトコルで宣言されたすべての必須メソッドを実装する必要があり、クラスまたはインスタンスがプロトコルに準拠するというのは、そのレパートリーの中にプロトコルで宣言されているすべてのメソッドがあるというのと同じです。

オブジェクトがプロトコルに準拠しているかどうかをチェックするには、`conformsToProtocol:`メッセージを送信します。

```
if ( ![receiver conformsToProtocol:@protocol(MyXMLSupport)]  ) {
    // オブジェクトがMyXMLSupportプロトコルに準拠していない
    // MyXMLSupportプロトコルで宣言されているメソッドを実装する
    //  レシーバを期待している場合は、これはおそらくエラー
```

```
}
```

（同じ名前のクラスメソッド、`conformsToProtocol:`もあることに注意。）

`conformsToProtocol:`テストは、単独のメソッドを対象とする`respondsToSelector:`テストによく似ています。ただし、特定のメソッドが実装されているかどうかではなく、プロトコルが採用されているか（そして結果的に、宣言されているメソッドがすべて実装されているか）どうかをテストします。プロトコル内のすべてのメソッドをチェックするため、`conformsToProtocol:`のほうが`respondsToSelector:`より効率が高いこともあります。

`conformsToProtocol:`テストは、`isKindOfClass:`テストにも似ています。ただし、継承階層をベースにした型ではなく、プロトコルをベースにした型をテストします。

型チェック

オブジェクトの型宣言は、形式プロトコルを含むように拡張することができます。したがって、プロトコルによってコンパイラによるもう1つのレベルの型チェックが可能になります。プロトコルは特定の実装に結び付いていないので、より抽象的な型チェックになります。

型宣言では、プロトコル名はタイプ名の後の不等号括弧内に記述します。

```
- (id <Formatting>)formattingService;
id <MyXMLSupport> anObject;
```

静的型定義では、コンパイラがクラス階層に基づいて型をテストできるのと同様に、この構文では、コンパイラはプロトコルに準拠しているかどうかに基づいて型をテストすることができます。

たとえば、次の宣言で、Formatterが抽象クラスであるとします。

```
Formatter *anObject;
```

上記の宣言はFormatterを継承するすべてのオブジェクトを1つの型にグループ化するため、コンパイラはその型を対象に割り当てをチェックすることができます。

次の宣言も同様です。

```
id <Formatting> anObject;
```

上記の宣言は、Formattingプロトコルに準拠するすべてのオブジェクトをクラス階層の位置に関係なく、1つの型にグループ化します。コンパイラは、プロトコルに準拠するオブジェクトだけがこの型に割り当てられることを保証できます。

いずれの場合も、共通の継承を共有するか、共通のメソッドセットを中心にまとまるかの違いはありますが、型によって類似のオブジェクトがグループ化されます。

2つの型を1つの宣言で一体化することができます。

```
Formatter <Formatting> *anObject;
```

プロトコルは、クラスオブジェクトの型定義には使用できません。クラスとして静的に型定義できるのはインスタンスだけであるのと同じように、プロトコルとして静的に型定義できるのもインスタンスだけです（ただし、実行時には、クラスとインスタンスはどちらも`conformsToProtocol:`メッセージに応答します）。

プロトコル内のプロトコル

クラスでプロトコルを採用するときに使うのと同じ構文を使用して、プロトコルの中にほかのプロトコルを組み込むことができます。

```
@protocol ProtocolName < protocol list >
```

不等号括弧内に記述されたすべてのプロトコルが、*ProtocolName*プロトコルの一部と見なされます。たとえば、次のようにして、Pagingプロトコルに Formattingプロトコルを組み込みます。

```
@protocol Paging < Formatting >
```

この場合、Pagingプロトコルに準拠するオブジェクトは、Formattingにも準拠します。次の型宣言を参照してください。

```
id <Paging> someObject;
```

また、次の`conformsToProtocol:`メッセージも参照してください。

```
if ( [anotherObject conformsToProtocol:@protocol(Paging)] )
    ...
```

これらはどちらも、Pagingプロトコルを記述するだけで、Formattingへの準拠もテストされます。

クラスでプロトコルを採用するときは、前述したように、プロトコルに宣言されている必須メソッドを実装する必要があります。さらに、採用したプロトコルに組み込まれているすべてのプロトコルにも準拠しなければなりません。組み込まれているプロトコルにさらにほかのプロトコルが組み込まれている場合、クラスはそれらにも準拠する必要があります。クラスが組み込まれたプロトコルに準拠するには、次のどちらかの方法があります。

- プロトコルが宣言するメソッドを実装する。
- プロトコルを採用し、メソッドを実装しているクラスを継承する。

たとえば、PagerクラスでPagingプロトコルを採用しているとします。Pagerが`NSObject`のサブクラスの場合、次のようになります。

```
@interface Pager :NSObject < Paging >
```

組み込まれたFormattingプロトコルで宣言されているメソッドを含め、すべてのPagingメソッドを実装する必要があります。これは、Pagingに加えてFormattingプロトコルも採用します。

これに対して、PagerがFormatter（Formattingプロトコルを独自に採用しているクラス）のサブクラスの場合、次のようになります。

```
@interface Pager :Formatter < Paging >
```

Pagingプロトコル自体に宣言されているすべてのメソッドを実装する必要がありますが、Formattingに宣言されているメソッドは実装の必要がありません。PagerがFormatterからFormattingプロトコルへの準拠を継承するからです。

クラスは、プロトコルを形式的に採用しなくても、単にプロトコルに宣言されているメソッドを実装するだけでプロトコルに準拠できることに注意してください。

ほかのプロトコルの参照

複雑なアプリケーションに取り組んでいるときに、次のようなコードを記述している場合があります。

```
#import "B.h"

@protocol A
- foo:(id <B>)anObject;
@end
```

ここで、プロトコルBが次のように宣言されているとします。

```
#import "A.h"

@protocol B
- bar:(id <A>)anObject;
@end
```

このような状況では、循環が生じ、どちらのファイルも正しくコンパイルされません。このような再帰的循環を中断するには、プロトコルが定義されているインターフェイスファイルをインポートするのではなく、`@protocol`ディレクティブを使って必要なプロトコルを前方参照する必要があります。次のコードの抜粋は、これをどのように行うかを示しています。

```
@protocol B;

@protocol A
- foo:(id <B>)anObject;
@end
```

`@protocol`ディレクティブをこのように使用すると、「B」が後で定義するプロトコルであることがコンパイラに単純に通知されます。プロトコルBが定義されているインターフェイスファイルはインポートしません。

宣言済みプロパティ

Objective-Cの「宣言済みプロパティ」の機能は、オブジェクトのアクセサメソッドの宣言と実装を簡単に行う方法を提供します。

概要

この言語機能には2つの側面があります。すなわち、**宣言済みプロパティ**を指定したり任意で合成したりするための構文要素と、「ドット構文」（20 ページ）で説明した関連する構文要素です。

（属性および関係という意味では）オブジェクトのプロパティへのアクセスは通常、一組のアクセサメソッド(getter/setter)を通じて行います。アクセサメソッドを使うことにより、カプセル化の原則が守られます（『*Object-Oriented Programming with Objective-C*』 > 「The Object Model」 > 「Mechanisms Of Abstraction」を参照）。APIのクライアントを実装の変更から隔離しつつ、getter/setterペアの動作と、基盤となる状態管理を厳格に制御できます。

アクセサメソッドの使用には大きな利点がありますが、それでもなおアクセサメソッドの記述は手間のかかる作業であり、ガベージコレクション環境と、参照カウント環境の両方をサポートするコードを記述しなければならない場合は特に面倒です。さらに、APIのコンシューマにとって重要と考えられるプロパティの側面（アクセサメソッドがスレッドセーフかどうか、設定時に新しい値がコピーされるかどうかなど）は不明瞭なままです。

宣言済みプロパティは、次の機能を提供することによって、標準アクセサメソッドの問題に対処します。

- プロパティ宣言により、アクセサメソッドの動作方法の明瞭で明示的な仕様を指定できます。
- コンパイラは、宣言で指定された仕様に従ってアクセサメソッドを合成できます。これは、コードの記述と保守が少量で済むことを意味します。
- プロパティは構文上は識別子として表現され、有効範囲を持つため、コンパイラは宣言されていないプロパティの使用を検出できます。

プロパティの宣言と実装

宣言済みプロパティには、宣言と実装という2つの部分があります。

プロパティの宣言

プロパティの宣言はキーワード`@property`から始まります。`@property`は、クラスの`@interface`にあるメソッド宣言リスト内の任意の場所に置くことができます。`@property`は、プロトコルやカテゴリの宣言の中に置くこともできます。

`@property(attributes) type name;`

`@property`はプロパティを宣言します。オプションの括弧内の属性セットは、格納方法のセマンティクスやプロパティのその他の振る舞いについて、追加情報を指定します。可能な値については、「プロパティ宣言属性」（72 ページ）を参照してください。Objective-Cのほかの型と同様に、各プロパティには型指定と名前があります。

リスト 5-1に、簡単なプロパティの宣言を示します。

リスト 5-1　簡単なプロパティの宣言

```
@interface MyClass :NSObject
{
    float value;
}
@property float value;
@end
```

プロパティの宣言は、2つのアクセサメソッドを宣言することと同等であると考えることができます。したがって、次のようなプロパティ宣言があるとします。

```
@property float value;
```

これは、次の記述と同等です。

```
- (float)value;
- (void)setValue:(float)newValue;
```

ただし、プロパティの宣言は、アクセサメソッドをどのように実装するかについての追加情報を提供します（「プロパティ宣言属性」（72 ページ）で説明します）。

プロパティ宣言属性

`@property(attribute [, attribute2, ...])`の形式を使ってプロパティを属性で装飾することができます。メソッドと同様に、プロパティの有効範囲はそれを囲んでいるインターフェイス宣言内です。コンマ区切りの変数名リストを使うプロパティ宣言の場合、プロパティ属性は名前付きプロパティのすべてに適用されます。

コンパイラにアクセサメソッドの作成を指定する`@synthesize`ディレクティブを使う場合、生成されるコードはキーワードによって指定された仕様に合致します。アクセサメソッドを自身で実装する場合は、そのアクセサメソッドが仕様に合致していることを確認する必要があります（たとえば`copy`を指定した場合は、setterメソッドで入力値をコピーしていることを確認する必要があります）。

アクセサメソッドの名前

プロパティに対応するgetterメソッドとsetterメソッドのデフォルトの名前は、それぞれ*propertyName*と`set`*PropertyName*`:`です。たとえば、「foo」というプロパティの場合、アクセサメソッドは`foo`と`setFoo:`になります。次の属性を使用すると、デフォルト名の代わりに独自の名前を指定できます。これらの属性は両方とも任意です。また、ほかの任意の属性と一緒に使用できます（ただし、`setter=`は`readonly`と一緒には使用できません）。

`getter=getterName`
: プロパティのgetアクセサの名前を指定します。getterはプロパティの型と一致する型を返し、引数は取りません。

`setter=setterName`
: プロパティのsetアクセサの名前を指定します。setterメソッドはプロパティの型と一致する型の引数を1つ取り、`void`を返します。

  プロパティが`readonly`の場合に`setter=`でsetterも指定すると、コンパイラ警告が発生します。

通常、アクセサメソッドの名前には、キー値コーディングに準拠する名前を指定する必要があります（『*Key-Value Coding Programming Guide*』を参照）。`getter`デコレータを使う一般的な理由は、ブール値に対する`is`*PropertyName*規則に従うためです。

書き込み可能性

これらの属性は、プロパティがsetアクセサを持つかどうかを指定します。これらは排他的に使われます。

`readwrite`
: プロパティを読み取り／書き込み可能として扱うべきであることを示します。これはデフォルトで適用されます。

  `@implementation`ではgetterとsetterの両方のメソッドが必須です。実装ブロックで`@synthesize`を使う場合は、getterメソッドとsetterメソッドが合成されます。

`readonly`
: プロパティが読み取り専用であることを示します。

  `readonly`を指定する場合、`@implementation`ではgetterメソッドだけが必須です。実装ブロックで`@synthesize`を使う場合は、getterメソッドだけが合成されます。また、ドット構文を使って値を代入しようとすると、コンパイラエラーが発生します。

setterのセマンティクス

これらの属性は、setアクセサのセマンティクスを指定します。これらは排他的に使われます。

`assign`
: setterで、単純代入を使用することを指定します。これはデフォルトで適用されます。

  通常は、`NSInteger`や`CGRect`などのスカラ型、または（参照カウント環境の場合は）デリゲートのような所有していないオブジェクトに対して、この属性を使用します。

  ガベージコレクション環境では、`retain`と`assign`は事実上同じです。

`retain`
: 代入時にオブジェクトに対して`retain`を呼び出す必要があることを指定します（デフォルトは`assign`です）。

  以前の値には`release`メッセージが送信されます。

  Mac OS X v10.6より前では、この属性はObjective-Cのオブジェクト型に対してのみ有効です（したがって、Core Foundationのオブジェクトに`retain`を指定することはできません。「Core Foundation」（79 ページ）を参照）。

  Mac OS X v10.6以降では、__attribute__キーワードを使用して、メモリ管理においてCore FoundationプロパティをObjective-Cオブジェクトのように扱うように指定できます。次に例を示します。

```
@property(retain) __attribute__((NSObject)) CFDictionaryRef myDictionary;
```

`copy`
: 代入にオブジェクトのコピーを使用することを指定します（デフォルトは`assign`です）。

以前の値には`release`メッセージが送信されます。

コピーは、`copy`メソッドを呼び出すことによって作成されます。この属性はオブジェクト型に対してのみ有効であり、その場合は`NSCopying`プロトコルを実装する必要があります。詳細については、「コピー」（78 ページ）を参照してください。

ガベージコレクションを利用するかどうかに応じて、適用される制約が次のように異なります。

- ガベージコレクションを利用しない場合、オブジェクトプロパティ対して`assign`、`retain`、または`copy`のいずれかを明示的に指定する必要があります。指定しないとコンパイラ警告が発生します（必要なメモリ管理動作について検討し、それを明示的に指定することが促進されます）。

  どれを選択すべきかを判断するには、Cocoaのメモリ管理ポリシーを理解する必要があります（『*Memory Management Programming Guide for Cocoa*』を参照）。

- ガベージコレクションを利用する場合、プロパティの型が`NSCopying`に準拠しているクラスでない限り、デフォルトを使う（つまり、`assign`、`retain`、または`copy`のどれも指定しない）と警告は発生しません。通常はデフォルトを使用します。ただし、プロパティ型のコピーが可能な場合は、カプセル化を守るためにオブジェクトのプライベートなコピーを作成することもできます。

アトミック性

この属性は、アクセサメソッドがアトミックでないことを指定します（アトミックであることを示すキーワードはありません）。

`nonatomic`
: アクセサが非アトミックになるように指定します。*デフォルトでは、アクセサはアトミックです。*

プロパティがデフォルトではアトミックであるため、合成されたアクセサがマルチスレッド環境においてプロパティへの堅牢なアクセスを可能にしています。つまり、getterから返される値やsetterを通じて設定される値は、ほかのスレッドが同時に実行しているかどうかに関係なく必ず完全に取得または設定されます。詳細については、「パフォーマンスとスレッド」（82 ページ）を参照してください。

`nonatomic`を指定しない場合、参照カウント環境では、オブジェクトプロパティ用に合成されたgetterアクセサは、ロックを使用し、戻り値の保持と自動解放を行います。その実装は、次のようになります。

```
[_internal lock]; // オブジェクトレベルのロックを使用してロックする
id result = [[value retain] autorelease];
[_internal unlock];
return result;
```

`nonatomic`を指定した場合は、オブジェクト用に合成されたアクセサは、単に値を直接返すだけです。

マークアップと非推奨化

プロパティはCスタイルのデコレータをすべてサポートします。次の例に示すように、プロパティを非推奨にして、__attribute__スタイルのマークアップをサポートすることができます。

```
@property CGFloat x
AVAILABLE_MAC_OS_X_VERSION_10_1_AND_LATER_BUT_DEPRECATED_IN_MAC_OS_X_VERSION_10_4;
@property CGFloat y __attribute__((...));
```

プロパティがInterface Builderのアウトレットであることを指定する場合は、IBOutlet識別子を使用できます。

```
@property (nonatomic, retain) IBOutlet NSButton *myButton;
```

ただし、IBOutletは形式的には属性リストの一部ではありません。

ガベージコレクションを使用する場合は、プロパティ宣言にストレージ修飾子__weakや__strongを使用できます。

```
@property (nonatomic, retain) __weak Link *parent;
```

ただし、この場合もこれらは形式的には属性リストの一部ではありません。

プロパティの実装ディレクティブ

@implementationブロックで@synthesizeディレクティブと@dynamicディレクティブを使って、特定のコンパイラ動作が行われるように指定できます。このディレクティブは、@property宣言ではどちらも*必須ではありません*。

重要： 特定のプロパティに@synthesizeや@dynamicを指定しない場合は、そのプロパティのgetterメソッドとsetterメソッド（readonlyのプロパティの場合はgetterのみ）の実装を用意する必要があります。

@synthesize

@implementationブロック内でsetterメソッドとgetterメソッドを指定しない場合に、そのプロパティのsetterメソッドとgetterメソッドを合成する必要があることをコンパイラに伝えるには@synthesizeキーワードを使います。

リスト 5-2 @synthesizeの使用

```
@interface MyClass :NSObject
{
    NSString *value;
}
@property(copy, readwrite) NSString *value;
@end

@implementation MyClass
@synthesize value;
@end
```

property=ivarの形式を使って、プロパティに対して特定のインスタンス変数を使用するように指示することもできます。次に例を示します。

```
@synthesize firstName, lastName, age = yearsOld;
```

これは、firstName、lastName、ageに対するアクセサメソッドを合成して、プロパティageをインスタンス変数yearsOldによって表すことを指定しています。合成されたメソッドの残りの側面は、オプションの属性によって決まります（「プロパティ宣言属性」（72 ページ）を参照）。

インスタンス変数の名前を指定するかどうかにかかわらず、@synthesizeでは、スーパークラスではなく、現在のクラスのインスタンス変数のみを使用できます。

次のようにランタイムに依存する動作に関していくつか相違があります（「ランタイムの相違」（82 ページ）を参照）。

- 従来のランタイムの場合、インスタンス変数は現在のクラスの@interfaceブロックですでに宣言されていなければなりません。プロパティと同じ名前、および互換性のある型のインスタンス変数が存在していれば、そのインスタンス変数が使われます。それ以外の場合、コンパイラエラーとなります。
- 最新のランタイムでは（『*Objective-C Runtime Programming Guide*』の「Runtime Versions and Platforms」を参照）、インスタンス変数は必要に応じて合成されます。同じ名前のインスタンス変数がすでに存在していれば、それが使用されます。

`@dynamic`

プロパティによって暗黙に示されるAPIコントラクトを満たすために、メソッド実装を直接提供したり、コードの動的なロードや動的なメソッド解決など、ほかのメカニズムを使って実行時に用意することをコンパイルに伝えるには、@dynamicキーワードを使います。リスト 5-3の例は、直接的なメソッド実装の使用を示しています。これはリスト 5-2（75 ページ）で取り上げた例と同じです。

リスト 5-3　直接的なメソッド実装の場合の@dynamicの使用

```
@interface MyClass :NSObject
{
    NSString *value;
}
@property(copy, readwrite) NSString *value;
@end

// ガベージコレクションの使用を想定
@implementation MyClass
@dynamic value;

- (NSString *)value {
    return value;
}

- (void)setValue:(NSString *)newValue {
    if (newValue != value) {
        value = [newValue copy];
    }
}
@end
```

プロパティの使用

サポートされる型

プロパティは、任意のObjective-Cクラス、Core Foundationのデータ型、またはPOD (plain old data)型（「C++ Language Note: POD Types」を参照）として宣言できます。Core Foundationの型を使用する場合の制約については、「Core Foundation」（79 ページ）を参照してください。

プロパティの再宣言

サブクラスでプロパティを再宣言できますが、（`readonly`を`readwrite`に変更することを除き）プロパティの属性全体をそのサブクラスで繰り返す必要があります。あるカテゴリまたはプロトコルで宣言されたプロパティについても同じことがいえます。つまりあるカテゴリまたはプロトコルでプロパティが再宣言されているときには、そのプロパティの属性を全体で繰り返す必要があります。

あるクラスでプロパティを`readonly`として宣言した場合、そのプロパティをクラス拡張（「拡張」（88 ページ）を参照）、プロトコル、またはサブクラス（「プロパティを使ったサブクラス化」（81 ページ）を参照）で`readwrite`として宣言しなおせます。クラス拡張で再宣言する場合は、`@synthesize`文よりも前でプロパティを再宣言することによって、setterが合成されます。読み取り専用のプロパティを読み／書き可能として宣言しなおせる機能により、2つの一般的な実装パターンが可能になります。すなわち不変クラスの可変サブクラス（`NSString`、`NSArray`、`NSDictionary`など）と、パブリックAPIが`readonly`であっても、クラスの内部に対してはプライベートな`readwrite`実装を有するプロパティです。次の例では、クラス拡張を使って、パブリックなヘッダで読み取り専用として宣言されたプロパティを、読み／書き可能としてプライベートに宣言しなおしたプロパティを提供する例を示します。

```
// パブリックなヘッダファイル
@interface MyObject :NSObject {
    NSString *language;
}
@property (readonly, copy) NSString *language;
@end

// プライベートな実装ファイル
@interface MyObject ()
@property (readwrite, copy) NSString *language;
@end

@implementation MyObject
@synthesize language;
@end
```

コピー

copy宣言属性を使うと、代入時に値がコピーされます。対応するアクセサを合成する場合、合成されたメソッドはcopyメソッドを使います。これは、文字列オブジェクトなどの属性のように、setterで渡された新しい値が可変である可能性があり（たとえば、NSMutableStringのインスタンスなど）、オブジェクトに独自に不変のプライベートなコピーを持たせたい場合などに役立ちます。たとえば、プロパティを次のように宣言したとします。

```
@property (nonatomic, copy) NSString *string;
```

この場合、合成されたsetterメソッドは次のようになります。

```
-(void)setString:(NSString *)newString {
    if (string != newString) {
        [string release];
        string = [newString copy];
    }
}
```

これは属性が文字列であれば十分に機能しますが、配列や集合などのコレクションである場合は問題となることがあります。通常、これらのコレクションは可変にしますが、copyメソッドは、このコレクションの*不変*バージョンを返します。このような状況では、次の例に示すようなsetterメソッドの実装を独自に用意する必要があります。

```
@interface MyClass :NSObject {
    NSMutableArray *myArray;
}
@property (nonatomic, copy) NSMutableArray *myArray;
@end

@implementation MyClass

@synthesize myArray;

- (void)setMyArray:(NSMutableArray *)newArray {
    if (myArray != newArray) {
        [myArray release];
        myArray = [newArray mutableCopy];
    }
}

@end
```

dealloc

宣言済みプロパティは、基本的にアクセサメソッドの宣言の代わりになります。プロパティを合成するときに、コンパイラは欠けているアクセサメソッドのみを作成します。deallocメソッドとの直接のやり取りはありません。つまり、プロパティが自動的に解放されることは*ありません*。ただし、宣言済みプロパティは、deallocメソッドの実装をクロスチェックする便利な手段を提供しています。ヘッダファイル内のすべてのプロパティ宣言を探して、assignマークが付いていないオブジェクトプロパティを解放し、assignマークが付いているオブジェクトプロパティを解放しないようにします。

注： 通常、deallocメソッドでは、次の例に示すように、（setアクセサを呼び出して、パラメータとしてnilを渡すのではなく）オブジェクトのインスタンス変数を直接解放すべきです。

```
- (void)dealloc {
    [property release];
    [super dealloc];
}
```

ただし、最新のランタイムを使用していてインスタンス変数を合成している場合は、そのインスタンス変数に直接アクセスできないため、アクセサメソッドを呼び出す必要があります。

```
- (void)dealloc {
    [self setProperty:nil];
    [super dealloc];
}
```

Core Foundation

「プロパティ宣言属性」（72 ページ）で説明したように、Mac OS X v10.6より前では、非オブジェクト型に対してretain属性を指定できません。このため、次の例に示すように型がCFTypeのプロパティを宣言してアクセサを合成すると、

```
@interface MyClass :NSObject
{
    CGImageRef myImage;
}
@property(readwrite) CGImageRef myImage;
@end

@implementation MyClass
@synthesize myImage;
@end
```

参照カウント環境では、生成されたsetアクセサは、単にインスタンス変数に新しい値を代入するだけになります（新しい値は保持されず、古い値は解放されません）。これは通常は適切でないため、メソッドを合成せずに自分自身でメソッドを実装する必要があります。

ガベージコレクション環境では、変数が__strongとして宣言されている場合、

```
...
__strong CGImageRef myImage;
...
@property CGImageRef myImage;
```

アクセサは適切に合成されます。つまり、このイメージはCFRetainされず、setterは書き込みバリアを実行します。

例

ここではプロパティの使用例をいくつか示します。

- Linkプロトコルは、nextプロパティを宣言します。

- MyClassはLinkプロトコルを採用しているため、暗黙的にnextプロパティも宣言します。MyClassは、ほかにもいくつかのプロパティを宣言します。
- creationTimestampとnextは合成されますが、異なる名前の既存のインスタンス変数を使います。
- nameは合成され、インスタンス変数の合成を使います（先述のとおり、従来のランタイムではインスタンス変数の合成はサポートされません。詳しくは「プロパティの実装ディレクティブ」（75 ページ）と「ランタイムの相違」（82 ページ）を参照）。
- gratuitousFloatにはdynamicディレクティブがあります。つまり直接的なメソッド実装を使ってサポートされます。
- nameAndAgeにはdynamicディレクティブはありませんが、これはデフォルトで適用されます。つまり指定された名前（nameAndAgeAsString）の直接的なメソッド実装を使ってサポートされます（読み取り専用であるためgetterのみ必要）。

リスト 5-4　クラスに対するプロパティの宣言

```
@protocol Link
@property id <Link> next;
@end


@interface MyClass :NSObject <Link>
{
    NSTimeInterval intervalSinceReferenceDate;
    CGFloat gratuitousFloat;
    id <Link> nextLink;
}
@property(readonly) NSTimeInterval creationTimestamp;
@property(copy) NSString *name;
@property CGFloat gratuitousFloat;
@property(readonly, getter=nameAndAgeAsString) NSString *nameAndAge;

@end


@implementation MyClass

@synthesize creationTimestamp = intervalSinceReferenceDate, name;
// 従来のランタイムでは、'name'を合成するとエラーになる
// 最新のランタイムでは、インスタンス変数が合成される

@synthesize next = nextLink;
// 格納用にインスタンス変数"nextLink"を使用

@dynamic gratuitousFloat;
// このディレクティブは厳密には必要ない

- (CGFloat)gratuitousFloat {
    return gratuitousFloat;
}
- (void)setGratuitousFloat:(CGFloat)aValue {
    gratuitousFloat = aValue;
}
```

```
- (NSString *)nameAndAgeAsString {
    return [NSString stringWithFormat:@"%@ (%fs)", [self name],
                [NSDate timeIntervalSinceReferenceDate] -
intervalSinceReferenceDate];
}


 - (id)init {
    if (self = [super init]) {
        intervalSinceReferenceDate = [NSDate timeIntervalSinceReferenceDate];
    }
    return self;
}

- (void)dealloc {
    [nextLink release];
    [name release];
    [super dealloc];
}

@end
```

プロパティを使ったサブクラス化

readonlyプロパティをオーバーライドして、これを書き込み可能にすることができます。たとえば、readonlyプロパティ、valueを持つMyIntegerというクラスを定義します。

```
@interface MyInteger :NSObject
{
    NSInteger value;
}
@property(readonly) NSInteger value;
@end

@implementation MyInteger
@synthesize value;
@end
```

次に、このプロパティを書き込み可能になるように再定義した、サブクラスMyMutableIntegerを実装します。

```
@interface MyMutableInteger :MyInteger
@property(readwrite) NSInteger value;
@end

@implementation MyMutableInteger
@dynamic value;

- (void)setValue:(NSInteger)newX {
    value = newX;
}
@end
```

パフォーマンスとスレッド

独自のメソッド実装を用意する場合、プロパティを宣言したという事実は、効率化やスレッドの安全性にはまったく影響しません。

合成プロパティを使う場合、コンパイラによって生成されるメソッドの実装は、指定する仕様によって異なります。パフォーマンスとスレッド処理に影響する宣言の属性は、`retain`、`assign`、`copy`、`nonatomic`です。最初の3つは設定メソッドの代入部分の実装にのみ影響し、次に示すような実装になります（実装は正確にこのとおりでない場合もあります）。

```
// 代入
property = newValue;

// 保持
if (property != newValue) {
    [property release];
    property = [newValue retain];
}

// コピー
if (property != newValue) {
    [property release];
    property = [newValue copy];
}
```

`nonatomic`属性の影響は環境によって異なります。デフォルトでは、合成されるアクセサはすべてアトミックです。参照カウント環境では、「アトミック性」（74 ページ）で示したように、アトミックな動作を保証するためにはロックを使用する必要があります。また、返されるオブジェクトは保持され自動解放されます。このようなアクセサが頻繁に呼び出されると、アトミックな動作はパフォーマンスに重大な影響をもたらすことがあります。ガベージコレクション環境では、ほとんどの合成メソッドはこうしたオーバーヘッドなしでアトミックになります。

アトミックな実装の目的は*堅牢*なアクセサを提供することであり、コードの*正確性*を保証することではないことを理解することが重要です。「アトミック」とは、*プロパティ*へのアクセスがスレッドセーフであるという意味ですが、単にクラス内のすべてのプロパティをアトミックにするだけでそのクラス（一般にはオブジェクトグラフ）が「スレッドセーフ」になるということではありません。スレッドの安全性を個々のアクセサメソッドのレベルで表現することはできません。マルチスレッドの詳細については、『*Threading Programming Guide*』を参照してください。

ランタイムの相違

プロパティの一般的な動作は、すべてのランタイム上で同じです（『*Objective-C Runtime Programming Guide*』の「Runtime Versions and Platforms」を参照）。ただし、最新のランタイムはインスタンス変数の合成をサポートしますが、従来のランタイムはそれをサポートしないという、重要な違いが1つあります。

従来のランタイムで`@synthesize`が機能するためには、同じ名前を持ち、そのプロパティと互換性のある型を持つインスタンス変数を用意するか、`@synthesize`文で既存の別のインスタンス変数を指定する必要があります。最新のランタイムでは、インスタンス変数がない場合は、コンパイラがそれを追加します。たとえば、次のようなクラス宣言と実装があるとします。

```
@interface MyClass :NSObject {
    float sameName;
    float otherName;
}
@property float sameName;
@property float differentName;
@property float noDeclaredIvar;
@end

@implementation MyClass
@synthesize sameName;
@synthesize differentName=otherName;
@synthesize noDeclaredIvar;
@end
```

従来のランタイム用のコンパイラでは、`@synthesize noDeclaredIvar;`の箇所でエラーが発生します。最新のランタイム用のコンパイラでは、`noDeclaredIvar`を表すインスタンス変数が追加されます。

カテゴリと拡張

カテゴリを使用すると、メソッドを既存のクラス（自分がソースを持っていないクラス）に追加できます。これは、サブクラス化を行わずに既存のクラスの機能を拡張できる強力な機能です。カテゴリを使用すれば、自身のクラスの実装を複数のファイル間で分けることもできます。クラス拡張も同様ですが、追加の*必須*APIをプライマリクラスの`@interface`ブロック内を除く場所でクラスに宣言できます。

メソッドのクラスへの追加

クラスにメソッドを追加するには、インターフェイスファイルでカテゴリ名の下にメソッドを宣言し、実装ファイルで同じ名前の下にメソッドを定義します。カテゴリ名は、メソッドが、新しいクラスではなく、どこかほかの場所で宣言されたクラスへの追加であることを示します。ただし、カテゴリを使って、クラスにインスタンス変数を追加することはできません。

カテゴリが追加するメソッドは、クラス型の一部になります。たとえば、カテゴリを使用して`NSArray`クラスに追加したメソッドは、コンパイラが`NSArray`インスタンスのレパートリーに含まれていると想定するメソッドの一部となります。`NSArray`クラスのサブクラスに追加されたメソッドは、`NSArray`型には含まれません（静的な型定義はコンパイラがオブジェクトのクラスを認識できる唯一の方法であるため、これが問題になるのは静的に型定義されたオブジェクトに関してのみです）。

カテゴリメソッドは、クラス自体で定義したメソッドで可能なことはすべて実行できます。実行時には、まったく違いがありません。カテゴリがクラスに追加するメソッドは、ほかのメソッドと同様に、すべてのクラスのサブクラスによって継承されます。

カテゴリインターフェイスの宣言は、クラスインターフェイス宣言とよく似ています。ただし、クラス名の後にカテゴリ名を丸括弧で囲んでリストアップし、スーパークラスを記述しない点が異なります。カテゴリのメソッドがクラスのインスタンス変数にアクセスする場合は、カテゴリでは拡張元のクラスのインターフェイスファイルをインポートする必要があります。

```
#import "ClassName.h"

@interface ClassName ( CategoryName )
// メソッド宣言
@end
```

この実装は、通常どおり、自身のインターフェイスをインポートします。共通の命名規則により、カテゴリの基本的なファイル名は、カテゴリが及ぶクラス名の後に「+」が付きその後にカテゴリ名が続きます。したがって、カテゴリ実装（`ClassName+CategoryName.m`というファイル内）は次のようになります。

```
#import "ClassName+CategoryName.h"

@implementation ClassName ( CategoryName )
// メソッド定義
@end
```

カテゴリでは、クラスに追加するインスタンス変数を宣言できないことに注意してください。カテゴリはメソッドだけを含みます。ただし、クラスの有効範囲内にあるすべてのインスタンス変数は、カテゴリの有効範囲内にもあります。これには、クラスで宣言されているすべてのインスタンス変数のほか、`@private`として宣言されているインスタンス変数も含まれます。

クラスに追加できるカテゴリの数には制限がありませんが、各カテゴリの名前は異なっている必要があり、それぞれが異なるメソッドのセットを宣言して定義する必要があります。

カテゴリの使いかた

カテゴリには、次のような使いかたがあります。

- ほかの実装者が定義したクラスを拡張するため。

  たとえば、Cocoaフレームワークで定義されているクラスにメソッドを追加できます。追加したメソッドはサブクラスに継承され、実行時にはクラスのオリジナルのメソッドと区別がつきません。

- サブクラスの代替手段として。

  既存のクラスを拡張するサブクラスを定義するのではなく、カテゴリによって、クラスに直接的にメソッドを追加することができます。たとえば、`NSArray`などのCocoaクラスにカテゴリを追加できます。サブクラスの場合のように、拡張するクラスのソースコードは必要ありません。

- 新しいクラスの実装を別々のソースファイルに分散させるため。

  たとえば、大規模なクラスのメソッドをいくつかのカテゴリにグループ化し、カテゴリごとに別のファイルに保存します。このように使用することで、カテゴリは、次のようなさまざまな点で開発プロセスに恩恵をもたらします。

  - 関連するメソッドをグループ化する簡単な方法を提供する。異なるクラスで定義されている類似のメソッドを、同じソースファイルにまとめておくことができます。
  - 複数のデベロッパがクラス定義に取り組むような大きなクラスの管理を簡素化できる。
  - 非常に大きなクラスについて、インクリメンタルコンパイルのメリットがある程度得られる。
  - よく使用するメソッドについて、参照の局所性を向上するのに役立つ。
  - 個々のアプリケーションに合わせてクラスを異なるように設定することが可能となり、同じソースコードのさまざまなバージョンを維持する必要性をなくす。

- 非形式プロトコルを宣言するため。

  「非形式プロトコル」（65 ページ）を参照してください。「ほかのクラスが実装できるインターフェイスの宣言」（61 ページ）で解説しているとおりです。

現在のところ、カテゴリを使用して、クラスが継承したメソッドや、クラスインターフェイスで宣言されているメソッドをオーバーライドできますが、この機能の使用はできるだけ避けてください。カテゴリはサブクラスの代わりではありません。カテゴリには、次のような重大な短所があります。

- カテゴリで継承元のメソッドをオーバーライドした場合、そのカテゴリ内のメソッドは、通常どおり、superへのメッセージによって継承元の実装を呼び出すことができます。しかし、カテゴリが、そのカテゴリのクラス内にすでに存在するメソッドをオーバーライドしている場合は、元の実装を呼び出す方法はありません。
- 同じクラスの別のカテゴリで宣言されているメソッドを確実にオーバーライドすることはできません。

  Cocoaクラスの多くがカテゴリを使用して実装されているため、この問題は特に重要です。オーバーライドしようとするフレームワーク定義のメソッド自身がカテゴリで実装されている可能性があるため、どの実装が優先されるかが明確になりません。
- メソッドの存在そのものが、すべてのフレームワークに影響する動作変更を引き起こす場合もあります。たとえば、NSObjectにwindowWillClose:の実装を追加すると、すべてのウインドウデリゲートが、そのメソッドに応答するようになり、NSWindowのすべてのインスタンスの動作が変わる場合があります。その結果、不可解な動作変更が生じて、クラッシュに至る場合もあります。

ルートクラスのカテゴリ

カテゴリは、ルートクラスを含め、任意のクラスにメソッドを追加できます。NSObjectに追加したメソッドは、コードにリンクするすべてのクラスで利用可能になります。メソッドの追加は役立つこともありますが、かなり危険な場合もあります。カテゴリによる変更内容は熟知されていて、影響も限定的であるように思えますが、継承によって影響が広範囲に及びます。未知のクラスに予期しない変更を加えてしまう可能性があります。つまり、行ったことの結果をすべて知ることができないかもしれないのです。さらに、ほかのデベロッパも変更があったことを知らなければ、変更による影響も理解できません。

さらに、そのほかにも、ルートクラスにメソッドを実装する際の注意点が2つあります。

- superへのメッセージは無効です（スーパークラスがありません）。
- クラスオブジェクトは、ルートクラスに定義されているインスタンスメソッドを実行することができます。

通常、クラスオブジェクトはクラスメソッドのみを実行できます。しかし、ルートクラスに定義されているインスタンスメソッドは特例です。それのメソッドは、すべてのオブジェクトが継承するランタイムシステムへのインターフェイスを定義します。クラスオブジェクトは完全なオブジェクトで、同じインターフェイスを共有する必要があります。

この機能は、NSObjectクラスのカテゴリで定義するインスタンスメソッドが、インスタンスだけでなくクラスオブジェクトによっても実行される可能性を考慮しなければならないことを意味します。たとえば、メソッドの本文では、selfがインスタンスだけでなく、クラスオブジェクトを指すこともあります。ルートインスタンスメソッドへのクラスアクセスの詳細については、Foundationフレームワークリファレンスの`NSObject`クラス仕様を参照してください。

拡張

クラス拡張は「匿名」のカテゴリに似ていますが、宣言するメソッドを対応するクラスのメイン`@implementation`ブロックで実装しなければならない点が異なります。

クラスには、パブリックに宣言されたAPIと、そのクラスまたはそのクラスが含まれるフレームワークだけで使用するためのプライベートに宣言された追加のAPIがあるのが一般的です。プライベートヘッダファイル内または上記で説明した実装ファイル内のカテゴリ（または複数のカテゴリ）内でこうしたAPIを宣言できます。これは有効ですが、コンパイラは宣言されたメソッドがすべて実装されていることを検証できません。

たとえば、コンパイラは次の宣言と実装をエラーなしでコンパイルします。

```
@interface MyObject :NSObject
{
    NSNumber *number;
}
- (NSNumber *)number;
@end

@interface MyObject (Setter)
- (void)setNumber:(NSNumber *)newNumber;
@end


@implementation MyObject

- (NSNumber *)number {
    return number;
}
@end
```

`setNumber:`メソッドの実装がないことに注意してください。これが実行時に呼び出されると、エラーが発生します。

クラス拡張を使用すると、プライマリクラスの`@interface`ブロック内以外の場所でクラスに追加の*必須*APIを定義できます。次の例を参照してください。

```
@interface MyObject :NSObject
{
    NSNumber *number;
}
- (NSNumber *)number;
@end

@interface MyObject ()
- (void)setNumber:(NSNumber *)newNumber;
@end


@implementation MyObject

- (NSNumber *)number {
    return number;
}
- (void)setNumber:(NSNumber *)newNumber {
```

```
    number = newNumber;
}
@end
```

この例では、次の点に注意してください。

- 2番目の`@interface`ブロックの括弧内に名前が指定されていない。
- `setNumber:`メソッドの実装が、クラスのメイン`@implementation`ブロックに含まれている。

`setNumber:`メソッドの実装は、クラスのメイン`@implementation`ブロック内に*必ず*存在しなければなりません（カテゴリ内では実装できません）。そうでなければ、コンパイラから`setNumber:`のメソッド定義が見つからないという警告が出されます。

関連参照

関連参照を使用すると、既存のクラスへのオブジェクトインスタンス変数を擬似的に追加できます。

関連参照はMac OS X v10.6以降でのみ利用できます。

クラス定義の外でのストレージの追加

関連参照を使用すると、クラス宣言を変更せずにオブジェクトにストレージを追加できます。これは、クラスのソースコードにアクセスできない場合や、バイナリ互換維持のためにオブジェクトのレイアウトを変更できない場合に役立ちます。

関連はキーに基づいているため、任意のオブジェクトにいくつでも関連を追加できますが、それぞれに別のキーを使用します。また、関連によって、関連先のオブジェクトが、少なくともソースオブジェクトが存続する間は、（ガベージコレクション環境において回収不能サイクルを生じさせることなく）有効であることを保証できます。

関連の作成

あるオブジェクトと別のオブジェクトの間に関連を作成するには、Objective-Cランタイム関数`objc_setAssociatedObject`を使用します。この関数は、ソースオブジェクト、キー、値、関連ポリシー定数の4つの引数を取ります。この中のキーと関連ポリシーについて、詳しく解説します。

- キーは`void`ポインタです。各関連のキーは一意でなければなりません。`static`変数を使用するのが典型的なパターンです。
- 関連ポリシーは、関連先のオブジェクトを代入するか、保持するか、コピーするかを指定したり、関連の作成をアトミックに行うか、非アトミックに行うを指定します。これは、宣言済みプロパティの属性と同様のパターンに従っています（「プロパティ宣言属性」（72 ページ）を参照）。関係のポリシーは、定数を使用して指定します（`objc_AssociationPolicy`を参照）。

次の例は、配列と文字列の間に関連を作成する方法を示しています。

リスト 7-1　配列と文字列の間の関連の作成

```
static char overviewKey;

NSArray *array = [[NSArray alloc] initWithObjects:@"One", @"Two", @"Three",
nil];
// 例を示す目的で、文字列を割り当て解除できるようにinitWithFormat:を使用する
NSString *overview = [[NSString alloc] initWithFormat:@"%@", @"First three
numbers";
```

```
objc_setAssociatedObject(array, &overviewKey, overview, OBJC_ASSOCIATION_RETAIN);

[overview release];
// (1) overviewは有効
[array release];
// (2) overviewは無効
```

(1)の時点では、OBJC_ASSOCIATION_RETAINポリシーによって、配列が関連先のオブジェクトを保持する設定になっているため、文字列overviewは有効です。しかし、配列が割り当て解除されると（(2)の時点）、overviewは解放され、割り当ても解除されます。たとえば、overviewの値を記録しようとすると、ランタイム例外が発生します。

関連先のオブジェクトの取得

関連先のオブジェクトは、Objective-Cランタイム関数objc_getAssociatedObjectを使用して取得します。リスト 7-1 （91 ページ）の例に続けて、次のコードを使用すると配列からoverviewを取得できます。

```
NSString *associatedObject = (NSString *)objc_getAssociatedObject(array,
&overviewKey);
```

関連の解消

関連を解消するには、通常nilを値として渡してobjc_setAssociatedObjectを使用します。

リスト 7-1 （91 ページ）の例に続けて、次のコードを使用すると、配列と文字列overviewの間の関連を解消できます。

```
objc_setAssociatedObject(array, &overviewKey, nil, OBJC_ASSOCIATION_ASSIGN);
```

（関連先のオブジェクトがnilに設定されている場合は、ポリシーは実際には重要ではありません。）

オブジェクトに対するすべての関連を解消するには、objc_removeAssociatedObjectsを使用できます。ただし、一般に、この関数はすべてのクライアントのすべての関連を解消するため、この関数の使用はお勧めできません。オブジェクトを「きれいな状態」に戻す必要がある場合のみ、この関数を使用します。

完全な例

次のプログラムは、ここまでのセクションのコード例を結合したものです。

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

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

```
    static char overviewKey;

    NSArray *array = [[NSArray alloc] initWithObjects:@"One", @"Two", @"Three",
 nil];
    // 例を示す目的で、割り当て解除可能な文字列を取得できるように、
    // initWithFormat:を使用する
    NSString *overview = [[NSString alloc] initWithFormat:@"%@", @"First three
 numbers";

    objc_setAssociatedObject(array, &overviewKey, overview,
OBJC_ASSOCIATION_RETAIN);
    [overview release];

    NSString *associatedObject = (NSString *)objc_getAssociatedObject(array,
&overviewKey);
    NSLog(@"associatedObject:%@", associatedObject");

    objc_setAssociatedObject(array, &overviewKey, nil, OBJC_ASSOCIATION_ASSIGN);
    [array release];

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

高速列挙

高速列挙は、簡潔な構文を使ってコレクションの内容を効率よく安全に列挙できる言語機能です。

for…in機能

高速列挙は、コレクションの内容を列挙できる言語機能です。構文は次のように定義されています。

```
for ( Type newVariable in expression ) { statements }
```

または

```
Type existingItem;
for ( existingItem in expression ) { statements }
```

どちらの場合も、*expression*は、`NSFastEnumeration`プロトコルに準拠するオブジェクトを返します（「高速列挙の採用」 （95 ページ）を参照）。反復変数には、戻されたオブジェクト内の各要素が順番に設定され、`statements`で定義されたコードが実行されます。オブジェクトのソースプールが空になってループの最後までくると、反復変数が`nil`に設定されます。ループが途中で終了した場合、反復変数は最後の反復要素を指したままになります。

高速列挙を使用する利点は次のようにいくつかあります。

- `NSEnumerator`を直接使うなどの手法よりもはるかに効率がよい。
- 構文が簡単である。
- 「安全」である。列挙子には変更防止の機能があるため、列挙中にコレクションを変更しようとすると例外が発生します。

反復中のオブジェクトの変更が禁止されているため、複数の列挙を同時に実行できます。

高速列挙の採用

ほかのオブジェクトのコレクションにアクセスできるインスタンスを持つクラスはすべて、`NSFastEnumeration`プロトコルを採用できます。Cocoaのコレクションクラス（`NSArray`、`NSDictionary`、`NSSet`）は、`NSEnumerator`と同様に、このプロトコルを採用しています。`NSArray`と`NSSet`の場合は、列挙は自身の内容が対象であることは明白です。それ以外のクラスでは、反復の対象のプロパティを明確にする必要があります。たとえば`NSDictionary`と、Core Dataクラスの`NSManagedObjectModel`は高速列挙に対応しており、`NSDictionary`はそのキーを`NSManagedObjectModel`はそのエンティティを列挙します。

高速列挙の使用

次のコード例は、NSArrayオブジェクトとNSDictionaryオブジェクトによる高速列挙の使用を示しています。

```
NSArray *array = [NSArray arrayWithObjects:
        @"One", @"Two", @"Three", @"Four", nil];

for (NSString *element in array) {
    NSLog(@"element:%@", element);
}

NSDictionary *dictionary = [NSDictionary dictionaryWithObjectsAndKeys:
    @"quattuor", @"four", @"quinque", @"five", @"sex", @"six", nil];

NSString *key;
for (key in dictionary) {
    NSLog(@"English:%@, Latin:%@", key, [dictionary valueForKey:key]);
}
```

次の例に示すように、高速列挙機能を備えたNSEnumeratorオブジェクトを使うこともできます。

```
NSArray *array = [NSArray arrayWithObjects:
        @"One", @"Two", @"Three", @"Four", nil];

NSEnumerator *enumerator = [array reverseObjectEnumerator];
for (NSString *element in enumerator) {
    if ([element isEqualToString:@"Three"]) {
        break;
    }
}

NSString *next = [enumerator nextObject];
// next = "Two"
```

明確な順序を持つコレクションや列挙子（配列から派生したNSArrayインスタンスやNSEnumeratorインスタンス）の場合、列挙はその順序に従って行われるため、必要な場合は反復をカウントするだけでコレクション内を指す正しいインデックスが得られます。

```
NSArray *array = /* これが存在するものと仮定する */;
NSUInteger index = 0;

for (id element in array) {
    NSLog(@"Element at index %u is:%@", index, element);
    index++;
}
```

その他の点では、この機能は標準的なforループに似た動作をします。breakを使用して反復を中断できます。要素をスキップしたい場合は、次の例のようにネストした条件文を使用できます。

```
NSArray *array = /* これが存在するものと仮定する */;

for (id element in array) {
    if (/* 要素のテスト */) {
        // テストをパスした要素のみに適用される文
    }
```

```
}
```

最初の要素をスキップしてそれに続く5つの要素だけを処理したい場合は、次の例のようにします。

```
NSArray *array = /* これが存在するものと仮定する */;
NSUInteger index = 0;

for (id element in array) {
     if (index != 0) {
          NSLog(@"Element at index %u is:%@", index, element);
     }

     if (++index >= 6) {
          break;
     }
}
```

静的な動作の実現

この章では、静的な型定義の仕組みについて説明し、Objective-C本来の動的な振る舞いを一時的に回避する方法など、Objective-Cのその他機能についても説明します。

デフォルトの動的動作

Objective-Cのオブジェクトは動的であるように作られています。オブジェクトに関する決定の可能なかぎり多くが、コンパイル時ではなく実行時に行われます。

- オブジェクト用のメモリは、新しいインスタンスを作成するクラスメソッドによって、実行時に**動的に割り当て**られます。
- オブジェクトは**動的に型定義**されます。ソースコードでは（コンパイル時）、オブジェクトのクラスに関係なく、オブジェクト変数は`id`型にできます。`id`変数の実際のクラス（および個々のメソッドとデータ構造体）はプログラムが実行されるまで決まりません。
- メッセージとメソッドは、「動的バインディング」（19 ページ）で説明されているように、**動的にバインド**されます。ランタイムプロシージャは、メッセージのメソッドセレクタをレシーバに「属する」メソッド実装と対応付けます。

これらの機能はオブジェクト指向プログラムに非常に高い柔軟性と能力をもたらしますが、支払うべき代償もあります。特に、コンパイラは、`id`変数の正確な型（クラス）をチェックできません。適切なコンパイル時の型チェックを可能にし、コードを自己文書化するために、Objective-Cでは、オブジェクトを汎用的に`id`として型定義するのではなく、クラス名で静的に型定義できます。また、操作を実行時からコンパイル時に戻すために、オブジェクト指向機能の一部をオフにすることもできます。

注： 通常は、実行される実際の作業に比べてオーバーヘッドはほんのわずかですが、メッセージは関数呼び出しよりもやや遅くなります。Objective-Cの動的な振る舞いを回避してもよい非常に数少ないケースは、SharkやInstrumentsのような分析ツールを使用して裏付けることができます。

静的な型定義

オブジェクト宣言において`id`の代わりに、クラス名のポインタを使用する場合は、次のようになります。

```
Rectangle *thisObject;
```

コンパイラは、宣言した変数の値を、宣言で指定されたクラスのインスタンス、または指定されたクラスを継承するクラスのインスタンスのいずれかに限定します。上記の例では、`thisObject`は何らかのRectangleに限定されます。

静的に型定義したオブジェクトは、idとして宣言されたオブジェクトと同じ内部データ構造を持ちます。型はオブジェクトには影響しません。コンパイラに提供するオブジェクトに関する情報の量と、ソースコードを読むほかの人が得られる情報の量にのみ影響します。

静的な型定義は、実行時のオブジェクトの処理方法にも影響しません。静的に型定義したオブジェクトは、id型のインスタンスを作成するのと同じクラスメソッドによって動的に割り当てられます。SquareがRectangleのサブクラスの場合、次のコードはRectangleのインスタンス変数だけでなく、Squareのインスタンス変数をすべて持ったオブジェクトを生成します。

```
Rectangle *thisObject = [[Square alloc] init];
```

静的に型定義したオブジェクトに送信するメッセージは、idとして型定義したオブジェクトと同じように、動的にバインドされます。さらに、静的に型定義したレシーバの実際の型は、実行時にメッセージング処理の一環として決定されます。次の例は、thisObjectに送信されるdisplayメッセージです。

```
[thisObject display];
```

これはRectangleスーパークラスのメソッドではなく、Squareクラスに定義されているメソッドのバージョンを実行します。

オブジェクトに関する詳細な情報をコンパイラに提供することで、静的な型定義にはidとして型定義したオブジェクトにはない可能性が広がります。

- ある状況において、静的な型定義はコンパイル時の型チェックを可能にします。
- 静的な型定義により、同じ名前のメソッドは同一の戻り型と引数型を持つ必要がある、という制限からオブジェクトが解放されます。
- また、構造体ポインタ演算子を使用して、オブジェクトのインスタンス変数に直接アクセスすることもできます。

最初の2つのトピックについては、以降のセクションで説明します。3番目のトピックは「クラスの定義」（37 ページ）で説明します。

型チェック

静的な型定義によって追加情報が提供されるため、コンパイラは次の2つの状況下において、より適切な型チェックサービスを行うことができます。

- 静的に定義したレシーバにメッセージを送信すると、コンパイラはレシーバが応答できることを確認できます。レシーバがメッセージに指定されているメソッドにアクセスできない場合は、警告が発せられます。
- 静的に型定義したオブジェクトを静的に型定義した変数に割り当てると、コンパイラは型の互換性があるかどうかを確認します。互換性がない場合は、警告が発せられます。

代入するオブジェクトのクラスが、代入先の変数のクラスと同じであるか、そのクラスを継承する場合には、代入を行っても警告は出ません。次の例は、これを示しています。

```
Shape     *aShape;
Rectangle *aRect;
```

```
aRect = [[Rectangle alloc] init];
aShape = aRect;
```

ここで、RectangleはShapeの一種である（Rectangleクラスは Shapeを継承する）ため、aRectをaShapeに代入することができます。ただし、2つの変数の役割を逆にしてaShapeをaRectに代入した場合、コンパイラが警告を発します。すべてのShapeがRectangleであるとはかぎらないからです（図 1-2 （26 ページ）も参照してください。この図は、ShapeとRectangleを含むクラス階層を示しています）。

代入演算子のどちらかの側にある式がidである場合は、チェックが行われません。静的に型定義したオブジェクトをidに自由に代入したり、idを静的に型定義したオブジェクトに自由に代入することができます。allocやinitのようなメソッドはidを返すため、コンパイラは静的に型定義した変数に対して互換性のあるオブジェクトが返されることを保証しません。次のコードはエラーが起きる可能性はありますが、指定は可能です。

```
Rectangle *aRect;
aRect = [[Shape alloc] init];
```

戻り型と引数型

一般に、異なるクラスで同じセレクタ（同じ名前）を持つメソッドは、戻り型と引数型も同じでなければなりません。この制約は動的バインディングを可能にするためにコンパイラによって課せられます。メッセージレシーバのクラス（そして実行を要求されるメソッドに関するクラス固有の詳細）はコンパイル時には分からないため、コンパイラは同じ名前のメソッドをすべて同様に処理する必要があります。コンパイラがランタイムシステムのためにメソッドの戻り型と引数型に関する情報を準備する際に、メソッドセレクタごとにメソッド記述を1つだけ作成します。

しかし、メッセージを静的に型定義したオブジェクトに送信する場合、コンパイラはレシーバのクラスを認識しています。コンパイラはメソッドに関するクラス固有の情報にアクセスできます。したがって、メッセージは戻り型と引数型に関する制限から解放されます。

継承元クラスへの静的な型定義

インスタンスは、自身のクラスまたは継承元のクラスに静的に型定義することができます。たとえば、すべてのインスタンスをNSObjectとして静的に型定義できます。

ただし、コンパイラは型指定のクラス名にのみ基づいて、静的に型定義されたオブジェクトのクラスを把握し、それに従って型チェックを実行します。したがって、インスタンスを継承元クラスとして型定義すると、コンパイラが実行時に起こると予測したことと、実際に起こることとの間に矛盾が生じる可能性があります。

たとえば、Rectangleインスタンスを Shapeとして静的に型定義する場合、次のようになります。

```
Shape *myRectangle = [[Rectangle alloc] init];
```

この場合、コンパイラはRectangleをShapeとして処理します。ここでRectangleメソッドを実行するメッセージをオブジェクトに送信します。

```
BOOL solid = [myRectangle isFilled];
```

この場合、コンパイラがエラーを報告します。`isFilled`メソッドはShapeクラスではなく、Rectangleクラスで定義されているからです。

しかし、今度はShapeクラスが認識するメソッドを実行するメッセージを送信したとします。

```
[myRectangle display];
```

Rectangleがメソッドをオーバーライドしていても、コンパイラはエラーを報告しません。しかし実行時には、Rectangleバージョンのメソッドが実行されます。

同様に、Upperクラスで、`double`を返す`worry`メソッドを宣言したとします。

```
- (double)worry;
```

また、UpperのMiddleサブクラスでメソッドをオーバーライドし、新しい戻り型を宣言します。

```
- (int)worry;
```

インスタンスをUpperクラスとして静的に型定義した場合、コンパイラは`worry`メソッドが`double`を返すと予測し、インスタンスをMiddleクラスとして型定義した場合、`worry`が`int`を返すと予測します。Middleインスタンスを`Upper`クラスとして型定義すると、明らかにエラーが生じます。コンパイラは、オブジェクトに送信される`worry`メッセージが`double`を返すことをランタイムシステムに通知しますが、実行時には実際に返されるのが`int`なのでエラーが発生します。

静的な型定義は、名前が同じメソッドを、同じ戻り型と引数型を持たなければならないという制限から解放しますが、確実に実行できるのはメソッドがクラス階層の異なる分岐で宣言されている場合だけです。

セレクタ

Objective-Cでは、「セレクタ」には2つの意味があります。オブジェクトに送信されるソースコードのメッセージで使われる場合は、セレクタは単にメソッドの名前を指します。しかし、ソースコードがコンパイルされるときは、セレクタはメソッド名に代わる一意の識別子を指します。コンパイル済みのセレクタは`SEL`型です。同じ名前のメソッドは、すべて同じセレクタを持ちます。セレクタを使用して、オブジェクトに対してメソッドを呼び出すことができます。これが、Cocoaのターゲット／アクションデザインパターンの実装の基礎になります。

メソッドとセレクタ

効率化のため、コンパイル済みのコードでは完全なASCII名をメソッドセレクタとして使用しません。その代わりに、コンパイラは各メソッド名をテーブルに書き込み、その名前を実行時にメソッドを示す一意の識別子と組み合わせます。ランタイムシステムによって、各識別子が一意であることが保証されます。2つのセレクタが同じであることはなく、同じ名前のメソッドはすべて同じセレクタに対応します。

SELと@selector

コンパイル済みのセレクタは、ほかのデータと区別するため、特別な型`SEL`に割り当てられます。有効なセレクタは`0`であることはありません。メソッドへの`SEL`識別子の割り当てはシステムに任せる必要があります。任意に割り当てても無駄になります。

`@selector()`ディレクティブを使用すると、完全なメソッド名ではなく、コンパイル済みのセレクタを参照することができます。次の例では、`setWidth:height:`のセレクタを、`setWidthHeight`変数に代入しています。

```
SEL setWidthHeight;
setWidthHeight = @selector(setWidth:height:);
```

コンパイル時に`@selector()`ディレクティブを使用して、`SEL`変数に値を代入するのが最も効率的です。しかし、場合によっては実行時に文字列をセレクタに変換する必要があります。これを実行するには、`NSSelectorFromString`関数を使います。

```
setWidthHeight = NSSelectorFromString(aBuffer);
```

逆方向の変換も可能です。`NSStringFromSelector`関数はセレクタに対応するメソッド名を返します。

```
NSString *method;
method = NSStringFromSelector(setWidthHeight);
```

メソッドとセレクタ

コンパイル済みのセレクタは、メソッド実装ではなく、メソッド名を識別します。たとえば、あるクラスのdisplayメソッドは、ほかのクラスで定義されたdisplayメソッドと同じセレクタを持ちます。これはポリモーフィズム（多態性）と動的バインディングには不可欠です。これにより、さまざまなクラスに属するレシーバに、同じメッセージを送信することができます。メソッドの実装ごとに1つのセレクタがあったとしたら、メッセージは関数呼び出しと変わりません。

クラスメソッド、および同じ名前のインスタンスメソッドには、同じセレクタが割り当てられます。しかし、それらは別々のドメインに属するため、2つの間に混乱は生じません。1つのクラスで、displayインスタンスメソッドに加えて、displayクラスメソッドを定義することができます。

メソッドの戻り値と引数の型

メッセージングルーチンはセレクタを通してのみメソッド実装にアクセスできるため、同じセレクタに対応するすべてのメソッドを同等に扱います。ルーチンは、セレクタに基づいてメソッドの戻り型と引数のデータ型を知ります。したがって、静的に型定義されたレシーバに送信されたメッセージを除いて、動的バインディングでは、同じ名前を持つメソッドのすべての実装で、戻り型と引数型が同じである必要があります（コンパイラはクラス型からメソッド実装を知ることができるため、静的に型定義されたレシーバはこの規則の例外です）。

同じ名前のクラスメソッドとインスタンスメソッドは、同じセレクタで表されますが、引数と戻り型が異なる可能性があります。

実行時のメッセージ変更

performSelector:、performSelector:withObject:、およびperformSelector:withObject:withObject:メソッドはNSObjectプロトコルで定義されており、初期引数としてSEL識別子をとります。3つのメソッドはすべて、メッセージング関数に直接マップされます。たとえば、次のように記述します。

```
[friend performSelector:@selector(gossipAbout:)
    withObject:aNeighbor];
```

これは、次の記述と同等です。

```
[friend gossipAbout:aNeighbor];
```

これらのメソッドにより、メッセージを受信するオブジェクトを変更できるのと同じように、実行時にメッセージを変更できます。メッセージ式を構成する両方の要素に変数名を使用することができます。

```
id   helper = getTheReceiver();
SEL  request = getTheSelector();
[helper performSelector:request];
```

上記の例では、レシーバ(helper)は実行時に（架空のgetTheReceiver関数によって）選択され、レシーバが実行するように要求されるメソッド(request)も実行時に（同様に架空のgetTheSelector関数によって）決められます。

注： performSelector:とその関連メソッドはidを返します。実行されるメソッドが別の型を返す場合は、適切な型にキャストする必要があります（ただし、キャストがすべての型に有効なわけではありません。メソッドはポインタまたはポインタと互換性のある型を返す必要があります）。

ターゲット／アクションデザインパターン

ユーザインターフェイス制御の処理において、Application Kitはレシーバとメッセージの両方を変更する方法を有効に利用しています。

NSControlオブジェクトは、アプリケーションへの指示を指定するのに使用できるグラフィカルデバイスです。大部分はボタン、スイッチ、ノブ、テキストフィールド、ダイヤル、メニュー項目など、現実世界の制御デバイスに似ています。ソフトウェアでは、これらのデバイスがアプリケーションとユーザの間に介在します。これらのデバイスは、キーボードやマウスなどのハードウェアデバイスから発されるイベントを解釈し、アプリケーション固有の命令に変換します。たとえば、「検索」というラベルの付いたボタンはマウスクリックを、アプリケーションが何かの検索を開始する命令に変換します。

Application Kitは制御デバイスを作成するためのテンプレートを定義し、「そのまま」使用できる独自のデバイスもいくつか定義します。たとえば、NSButtonCellクラスは、NSMatrixインスタンスに割り当てて、サイズ、ラベル、ピクチャ、フォント、およびキーボードショートカットで初期化できるオブジェクトを定義しています。ユーザがボタンをクリックすると（あるいは、キーボードショートカットを使用すると）、NSButtonCellオブジェクトはアプリケーションに何かを実行するよう指示するメッセージを送信します。これを行うには、NSButtonCellオブジェクトを画像、サイズ、およびラベルだけでなく、どんなメッセージを誰に送信するかに関する指示についても初期化する必要があります。したがって、NSButtonCellインスタンスは、アクションメッセージ、送信するメッセージで使用するメソッドセレクタ、ターゲット（メッセージを受信するオブジェクト）で初期化することができます。

```
[myButtonCell setAction:@selector(reapTheWind:)];
[myButtonCell setTarget:anObject];
```

ボタンセルは、NSObjectのperformSelector:withObject:メソッドを使用してメッセージを送信します。すべてのアクションメッセージは1つの引数、つまりメッセージを送信する制御デバイスのidを受け取ります。

Objective-Cでメッセージを変更できないとしたら、すべてのNSButtonCellオブジェクトが同じメッセージを送信しなければならず、メソッドの名前がNSButtonCellソースコードで固定されます。ユーザ操作をアクションメッセージに変換するメカニズムを単に実装するのではなく、ボタンセルなどのコントロールでメッセージの内容に制約を課す必要が生じます。これでは、どのようなオブジェクトも、複数のボタンセルに対応するのが困難になります。ボタンごとに1つのターゲットを用意するか、ターゲットオブジェクトがメッセージを送信したボタンを検出し、それに応じて動作する必要が生じます。ユーザインターフェイスを再配置するたびに、アクションメッセージに応答するメソッドも実装し直す必要が生じます。幸いにも、Objective-Cでは、このように不要な複雑さが回避されています。

メッセージングエラーの回避

オブジェクトが、レパートリーにないメソッドを実行するメッセージを受信すると、エラーが発生します。これは、存在しない関数を呼び出した場合と同じ種類のエラーです。しかし、メッセージングは実行時に行われるため、多くの場合、プログラムを実行するまでエラーが明らかになりません。

メッセージセレクタが不変で、受信側オブジェクトのクラスが分かっている場合、このようなエラーを回避するのは比較的簡単です。プログラムを記述するときに、レシーバが応答できることを確認しておけばよいからです。レシーバが静的に型定義されていれば、コンパイラがそのテストを実行してくれます。

しかし、メッセージセレクタまたはレシーバのクラスが可変の場合、そのテストを実行時まで延期しなければならない場合もあります。`NSObject`クラスで定義されている`respondsToSelector:`メソッドを使用してレシーバがメッセージに応答できるかどうかを調べることができます。引数としてメソッドセレクタを受け取り、レシーバがセレクタに一致するメソッドにアクセスできるかどうかを返します。

```
if ( [anObject respondsToSelector:@selector(setOrigin::)])
    [anObject setOrigin:0.0 :0.0];
else
    fprintf(stderr, "%s can't be placed\n",
        [NSStringFromClass([anObject class]) UTF8String]);
```

`respondsToSelector:`テストが特に重要なのは、コンパイル時に制御の及ぶ範囲内にないオブジェクトにメッセージを送信する場合です。たとえば、ほかから設定可能な変数によって指定されるオブジェクトにメッセージを送信するコードを記述する場合は、必ずメッセージに応答できるメソッドをレシーバが実装するようにします。

注： また、オブジェクトは、自身で直接応答できない場合、受信したメッセージをほかのオブジェクトに転送するように準備しておくこともできます。その場合、メッセージを別のオブジェクトに割り当てて間接的に応答しているにも関わらず、オブジェクトがメッセージを処理できるように見えます。詳細については、『*Objective-C Runtime Programming Guide*』の「Message Forwarding」を参照してください。

例外処理

Objective-C言語には、JavaやC++に似た例外処理構文があります。`NSException`、`NSError`、またはカスタムクラスと併用すると、強力なエラー処理をプログラムに追加することができます。この章では、例外構文と例外処理の概要について説明します。詳細については、『*Exception Programming Topics for Cocoa*』を参照してください。

例外処理の有効化

GCC (GNU Compiler Collection)バージョン3.3以降を使用すると、Objective-Cは言語レベルで例外処理をサポートします。これらの機能のサポートを有効にするには、GCC (GNU Compiler Collection) v3.3以降の`-fobjc-exceptions`スイッチを使用します（この機能を使用すると、Mac OS X v10.3以降でのみ実行可能なアプリケーションになります。それ以前のバージョンのソフトウェアには、例外処理と同期に対するランタイムサポートがないからです）。

例外処理

例外は、通常のプログラム実行フローが中断される特殊な状態です。例外が生成される（通常、例外は**発生する**または**スローする**と言われます）理由には、ソフトウェアだけでなくハードウェアに起因するものなど、さまざまなものがあります。たとえば、演算エラー（0による除算、アンダーフロー、オーバーフローなど）、未定義の命令の呼び出し（未実装のメソッドを呼び出そうとした場合など）、範囲外のコレクション要素にアクセスしようとした場合などがあります。

Objective-Cの例外サポートは、`@try`、`@catch`、`@throw`、および`@finally`の4つのコンパイラディレクティブが中心になります。

- 例外をスローする可能性のあるコードは`@try`ブロックに入れます。
- `@catch()`ブロックには、`@try`ブロックでスローされた例外の例外処理ロジックが含まれています。異なるタイプの例外をキャッチするために、複数の`@catch()`ブロックを持つこともできます（これについては、「異なるタイプの例外のキャッチ」（108 ページ）で説明します）。
- `@finally`ブロックには、例外がスローされたかどうかに関係なく、実行しなければならないコードが含まれています。
- 例外をスローするには、`@throw`ディレクティブを使用します。例外は本質的にはObjective-Cオブジェクトです。通常は`NSException`オブジェクトを使用しますが、必ずしもその必要はありません。

次の例は簡単な例外処理アルゴリズムを示しています。

```
Cup *cup = [[Cup alloc] init];

@try {
```

```
    [cup fill];
}
@catch (NSException *exception) {
    NSLog(@"main:Caught %@:%@", [exception name], [exception reason]);
}
@finally {
    [cup release];
}
```

異なるタイプの例外のキャッチ

@tryブロックでスローされた例外をキャッチするには、@tryブロックに続いて、1つまたは複数の@catch()ブロックを使用します。@catch()ブロックは、最も特殊性が高いものから最も特殊性の低いものの順に並べるべきです。そうすることで、リスト 11-1に示すように、例外処理をグループとしてまとめることができます。

リスト 11-1　例外ハンドラ

```
@try {
    ...
}
@catch (CustomException *ce) {  // 1
    ...
}
@catch (NSException *ne) {  // 2
    // このレベルで必要な処理を行う。
    ...

}
@catch (id ue) {
    ...
}
@finally {  // 3
    // 例外が発生したかどうかにかかわらず行う必要のある処理を実行する。
    ...
}
```

次のリストは、番号の付いたコード行の説明です。

1. 最も特殊なタイプの例外をキャッチする。
2. より一般的なタイプの例外をキャッチする。
3. 例外がスローされたかどうかに関係なく、必ず実行しなければならないクリーンアップ処理を実行する。

例外のスロー

例外をスローするには、例外名や例外をスローする理由など、適切な情報を持ったオブジェクトをインスタンス化する必要があります。

```
NSException *exception = [NSException exceptionWithName:@"HotTeaException"
                                 reason:@"The tea is too hot"  userInfo:nil];
@throw exception;
```

重要： 多くの環境で、例外の使用はかなり一般的です。たとえば、ルーチンが正常に実行できない（ファイルが見つからない、データが正しく解析できないなど）ことを表すために例外をスローできます。例外は、Objective-Cではリソースをかなり消費します。一般的なフロー制御や単にエラーを示すために例外を使用するべきではありません。代わりに、メソッドや関数の戻り値を使用してエラーが発生したことを示し、エラーオブジェクトで問題に関する情報を提供すべきです。詳細については、『*Error Handling Programming Guide For Cocoa*』を参照してください。

`@catch()`ブロック内では、`@throw`ディレクティブを引数なしで使用することで、キャッチした例外を再スローすることができます。これはコードをより読みやすくするのに役立ちます。

スローできるのは、`NSException`オブジェクトに限定されていません。任意のObjective-Cオブジェクトを例外オブジェクトとしてスローできます。`NSException`クラスは例外処理に役立つメソッドを提供しますが、必要があれば独自のものを実装することもできます。`NSException`をサブクラス化して、ファイルシステム例外や通信例外など、特殊なタイプの例外を実装することもできます。

スレッド化

Objective-Cでは、スレッド同期と例外処理をサポートしています。これらについては、この章と「例外処理」（107 ページ）で説明されています。これらの機能のサポートを有効にするには、GCC (GNU Compiler Collection) v3.3以降の`-fobjc-exceptions`スイッチを使用します。

注： プログラムでこれらの機能のいずれかを使用すると、Mac OS X v10.3以降でのみ実行可能なアプリケーションになります。それ以前のバージョンのソフトウェアでは、例外処理と同期のランタイムサポートがないからです。

スレッド実行の同期

Objective-Cでは、アプリケーションのマルチスレッド処理をサポートしています。これは、2つのスレッドが同時に同じオブジェクトを変更しようとする可能性があること、プログラムに深刻な問題を引き起こす可能性がある状況を意味します。同時に複数のスレッドによって実行されないようにコードの一部を保護できるように、Objective-Cでは`@synchronized()`ディレクティブを提供しています。

`@synchronized()`ディレクティブは、コードの一部を1つのスレッドで使用するようにロックします。スレッドが保護コードを抜け出すまで、つまり、`@synchronized()`ブロックの最後の文を通過するまで、ほかのスレッドはブロックされます。

`@synchronized()`ディレクティブは、唯一の引数として`self`を含む任意のObjective-Cオブジェクトを受け取ります。このオブジェクトは、*相互排他(mutual exclusion)*セマフォまたはミューテックス*(mutex)*と呼ばれています。これにより、スレッドはコードの一部をほかのスレッドに使用されないようにロックできます。プログラムの個別のクリティカルセクションを保護するには、別々のセマフォを使用する必要があります。アプリケーションのマルチスレッド処理を開始する前に、すべての相互排他オブジェクトを作成して、競合状態を回避するのが最も安全です。

リスト 12-1に、ミューテックスに`self`を使って、カレントオブジェクトのインスタンスメソッドへのアクセスを同期するコード例を示します。`self`の代わりにクラスオブジェクトを使用して、同様の方法で対応するクラスのクラスメソッドを同期できます。もちろん、後者の例では、すべての呼び出し側で共有しているクラスオブジェクトは1つのみのため、クラスメソッドを実行できるスレッドは一度に1つだけです。

リスト 12-1 selfを使ってメソッドをロックする

```
- (void)criticalMethod
{
    @synchronized(self) {
        // クリティカルコード。
        ...
    }
}
```

リスト 12-2では、一般的な方法を示します。コードでは、クリティカルなプロセスを実行する前に、Accountクラスからセマフォを取得し、これを使ってクリティカルセクションをロックしています。Accountクラスは、自身の`initialize`メソッド内にセマフォを作成できます。

リスト 12-2　カスタムセマフォを使ってメソッドをロックする

```
Account *account = [Account accountFromString:[accountField stringValue]];

// セマフォを取得する。
id accountSemaphore = [Account semaphore];

@synchronized(accountSemaphore) {
    // クリティカルコード。
    ...
}
```

Objective-Cの同期機能は、再帰コードおよび再入可能コードをサポートしています。スレッドは1つのセマフォを再帰的に複数回使用できます。そのスレッドが取得したすべてのロックを解放するまで（つまり、すべての`@synchronized()`ブロックが正常終了するか、例外によって終了するまで）、ほかのスレッドはその使用をブロックされます。

`@synchronized()`ブロックのコードが例外をスローすると、Objective-Cランタイムがその例外をキャッチし、セマフォを解放して（その結果、ほかのスレッドが保護コードを実行できます）、その例外を次の例外ハンドラに再スローします。

リモートメッセージング

ほかの多くのプログラミング言語と同様に、Objective-Cは当初、単一のアドレス空間でシングルプロセスとして実行されるプログラム用に設計されました。

しかし、実行時に解決されるメッセージを介して比較的自己完結型の単位の間で通信が行われるオブジェクト指向モデルは、プロセス間通信にも非常に適していると考えられます。異なるアドレス空間（つまり、異なるタスク）や、同一タスクを実行する異なるスレッドに存在するオブジェクトどうしのObjective-Cメッセージのやり取りを想像するのは難しくありません。

たとえば、典型的なサーバークライアント間の対話では、クライアントタスクはサーバ内の指定されたオブジェクトに要求を送信し、サーバは特定のクライアントオブジェクトをターゲットにして通知やほかの情報の送信を行います。あるいは、ユーザコマンドを実行するために大量の計算を行う必要のある、対話型アプリケーションがあるとします。このアプリケーションは、処理の間ユーザに待つように通知するダイアログを単純に表示するか、ユーザ入力を受け付けるアプリケーションの主要部を解放しておくために処理作業を切り離して従属タスクに任せることができます。2つのタスク内のオブジェクトは、Objective-Cメッセージを介して通信を行います。

分散オブジェクト

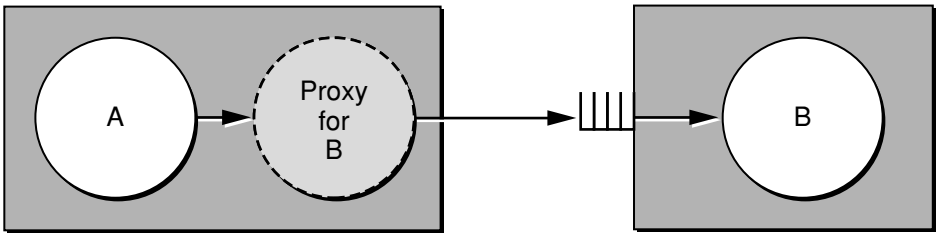
Objective-Cのリモートメッセージングには、異なるアドレス空間にあるオブジェクトどうしの接続の確立、リモートアドレス空間内のオブジェクトを対象とするメッセージの検出、アドレス空間どうしでのデータ転送を行うことのできるランタイムシステムが必要です。また、2つのタスクの間のスケジュール調整を行う必要があります。つまり、リモートレシーバが処理から解放されて応答できるようになるまでメッセージを保持できる必要があります。

Cocoaには、ランタイムシステムにまさにこのような拡張を加える、**分散オブジェクト**アーキテクチャが組み込まれています。分散オブジェクトを使用すると、Objective-Cメッセージを別のタスクのオブジェクトに送信したり、メッセージを同じタスクの別のスレッドに実行させることができます（リモートメッセージを同一タスクの2つのスレッド間で送信する場合、スレッドは別々のタスクのスレッドどうしの場合とまったく同様に処理されます）。Cocoaの分散オブジェクトシステムは、ランタイムシステムの基礎の上に構築されており、Cocoaオブジェクトの基本的な動作を変更しないことに注意してください。

リモートメッセージを送信するには、アプリケーションはまずリモートレシーバとの接続を確立する必要があります。接続を確立すると、リモートオブジェクトに対するプロキシがアプリケーション自身のアドレス空間に用意されます。以降、アプリケーションはプロキシを通してリモートオブジェクトと通信します。プロキシは、リモートオブジェクトのアイデンティティを自分のアイデンティティとして使用し、独自のアイデンティティは持ちません。アプリケーションは、プロキシをリモートオブジェクトのように見なすことができ、多くの場合、プロキシはリモートオブジェクトも同然です。

リモートメッセージングについて、図 13-1 （114 ページ） に示します。この場合、オブジェクトAはプロキシを通してオブジェクトBと通信し、Bへのメッセージは Bが応答できるようになるまでキュー内で待機します。

図 13-1 リモートメッセージ



送信側と受信側は異なるタスクにあり、互いに関係なくスケジュールされます。そのため、送信側がメッセージを送信する準備ができていても、受信側の準備ができている保証はありません。したがって、到着メッセージはキュー内に置かれ、受信側アプリケーションの都合に合わせて取り出されます。

プロキシは、リモートオブジェクトに代わって動作するものではなく、そのクラスへのアクセスを必要とするものでもありません。また、プロキシは、オブジェクトのコピーではなく、軽量の代理です。ある意味において、プロキシは透過的です。受信したメッセージをリモートレシーバに受け渡して、プロセス間通信を管理するだけです。その主な機能は、ローカルアドレスを持っていないオブジェクトのためにローカルアドレスを提供することです。しかし、プロキシは完全に透過的なわけではありません。たとえば、プロキシでは、オブジェクトのインスタンス変数の設定や取得を直接行うことはできません。

リモートレシーバは一般的に匿名です。そのクラスはリモートアプリケーション内に隠蔽されています。送信側アプリケーションは、アプリケーションの設計や使用するクラスを知っている必要はありません。また、送信側で同じクラスを使用する必要もありません。知っている必要があるのは、リモートオブジェクトがどのようなメッセージに応答するかということだけです。

そのため、リモートメッセージを受信するように指定されたオブジェクトは、形式プロトコルとしてインターフェイスを公開します。送信側および受信側アプリケーションの双方でプロトコルを宣言します。どちらも同じプロトコル宣言をインポートします。リモートオブジェクトはプロトコルに準拠する必要があるため、受信側アプリケーションでそのプロトコルを宣言します。送信側アプリケーションでプロトコルを宣言するのは、送信するメッセージに関してコンパイラに通知するためであり、また、`conformsToProtocol:`メソッドと`@protocol()`ディレクティブを使用してリモートレシーバをテストするためです。送信側アプリケーションは、そのプロトコルにいかなるメソッドも実装する必要はありません。プロトコルを宣言するのは、単にリモートレシーバに対してメッセージを開始するからです。

`NSProxy`や`NSConnection`クラスなど、分散オブジェクトアーキテクチャについては、Foundationフレームワークリファレンスの『*Distributed Objects Programming Topics*』を参照してください。

言語サポート

リモートメッセージングは、プログラム設計に多くの魅力をもたらす可能性があるだけでなく、Objective-C言語のいくつかの興味深い問題も提起します。問題の大半は、リモートメッセージングの効率と、2つのタスクが互いに通信している間に保っているべき分離の程度に関するものです。

プログラマがリモートメッセージの目的に関する明示的な指示を与えられるように、Objective-Cでは、形式プロトコル内でメソッドを宣言する際に使用できる次の6つの型修飾子を定義しています。

```
oneway
```

```
in
out
inout
bycopy
byref
```

これらの修飾子の使用は形式プロトコルに限定され、クラスおよびカテゴリ宣言に使用することはできません。ただし、クラスまたはカテゴリでプロトコルを採用する場合、プロトコルメソッドの実装では、対応するメソッドの宣言に使用されている修飾子を使用することができます。

以降のセクションでは、これらの修飾子の使いかたについて説明します。

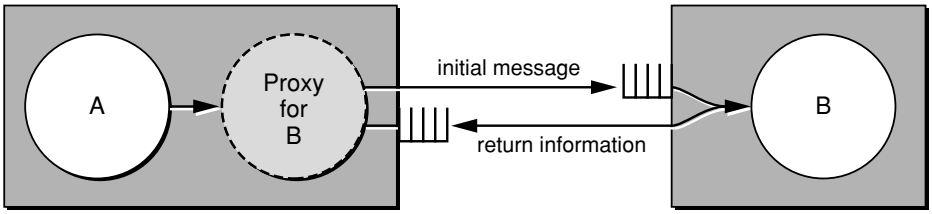
同期および非同期メッセージ

まず、ごく単純な戻り値を持つメソッドについて考えてみます。

```
- (BOOL)canDance;
```

`canDance`メッセージを同じアプリケーションのレシーバに送信すると、このメソッドが呼び出され、戻り値が送信側に直接提供されます。しかし、レシーバがリモートアプリケーションにある場合は、2つの基礎的なメッセージが必要になります。1つはリモートオブジェクトを取得してメソッドを呼び出すメッセージで、もう1つはリモート計算の結果を返送するメッセージです。これについては、次の図に示します。

図 13-2　往復メッセージ



ほとんどのリモートメッセージは基本的に、次に説明するような、双方向（または「往復」）のリモートプロシージャ呼び出し(RPC)です。送信側アプリケーションは、受信側アプリケーションがメソッドを呼び出して処理を終了し、終了したことを示すメッセージを、要求された情報があればそれらとともに返送するまで待機します。情報が返されなかったとしても、レシーバが終了するのを待機するため、2つの通信アプリケーションを調整して、両者の「同期」を保てるという利点があります。そのため、往復メッセージは、**同期**メッセージとも呼ばれます。同期メッセージがデフォルトです。

しかし、応答を待機することは必ずしも必要なわけではなく、得策でないこともあります。場合によっては、リモートメッセージを単純に送信して戻るだけでよいこともあります。そして、レシーバが可能なときにタスクに取り掛かるのに任せます。その間、送信側はほかのことを続けられます。Objective-Cは戻り型修飾子`oneway`を指定して、メソッドが**非同期**メッセージ専用に使用されることを示します。

```
- (oneway void)waltzAtWill;
```

onewayは（constのような）型修飾子であり、oneway floatあるいはoneway idのように、特定の型名と組み合わせて使用できますが、唯一意味のある組み合わせはoneway voidだけです。非同期メッセージは、有効な戻り値を持つことができません。

ポインタ引数

次に、ポインタ引数を受け取るメソッドについて考えます。ポインタは、参照によってレシーバに情報を渡すために使用できます。呼び出されたメソッドは、渡されたアドレスに格納されているものを参照します。

```
- setTune:(struct tune *)aSong
{
    tune = *aSong;
    ...
}
```

また、同じ種類の引数を使用して、参照によって情報を返すこともできます。メソッドはポインタから、メッセージによって要求された情報の格納場所を知ることができます。

```
- getTune:(struct tune *)theSong
{
    ...
    *theSong = tune;
}
```

ポインタの使いかたによって、リモートメッセージの処理方法が変わります。いずれの場合も、ポインタをそのままの状態でリモートオブジェクトに渡すことはできません。ポインタは、送信側のアドレス空間内のメモリ領域を指しているため、リモートレシーバのアドレス空間では意味を持ちません。リモートメッセージングのランタイムシステムは、多少の調整を行う必要があります。

参照によって情報を渡すために引数を使用する場合、ランタイムシステムはポインタをたどり、ポインタが指す値をリモートアプリケーションに送信して、その値をアプリケーションのローカルアドレスに格納し、そのアドレスをリモートレシーバに渡す必要があります。

一方、参照によって情報を返すためにポインタを使用する場合は、ポインタが指す値をほかのアプリケーションに送信する必要はありません。その代わりに、ほかのアプリケーションからの値の返送を受け、ポインタが示す場所に書き込む必要があります。

初めのケースでは、情報を往路で渡しています。次のケースでは、情報を復路で返しています。このような場合、リモートメッセージングのためのランタイムシステム側では動作が大きく異なるため、Objective-Cはプログラマの意図を明示できる型修飾子を提供します。

- 型修飾子inは、メッセージに情報を含めて渡すことを示します。

  ```
  - setTune:(in struct tune *)aSong;
  ```

- 修飾子outは、参照によって情報を返すために引数を使用することを示します。

  ```
  - getTune:(out struct tune *)theSong;
  ```

- 第3の修飾子inoutは、情報の提供と返送のために引数を使用することを示します。

  ```
  - adjustTune:(inout struct tune *)aSong;
  ```

Cocoa分散オブジェクトシステムでは、inがデフォルトであるconstとして宣言されたポインタ引数を除き、すべてのポインタ引数のデフォルト修飾子がinoutとなります。inoutは最も安全な条件ですが、双方向で情報を渡す必要があるため、最も時間がかかるものでもあります。値（ポインタ以外）によって渡される引数にとって、意味のある唯一の修飾子はinです。inはどのような種類の引数でも使用できますが、outとinoutはポインタに対してのみ意味があります。

C言語では、複合値を表すためにポインタを使用する場合もあります。たとえば、文字列は文字ポインタ(char *)として表します。表記と実装においては、間接のレベルが存在しますが、概念上はありません。概念的には、文字列はそれ自体がエンティティであり、ほかのものへのポインタではありません。

このような場合には、分散オブジェクトシステムはポインタを自動的にたどり、ポインタが指すものを値渡しのように渡します。したがって、outおよびinout修飾子は、単純な文字ポインタでは意味をなしません。参照によって文字列を渡したり、返したりすると、リモートメッセージに間接のレベルが生まれます。

```
- getTuneTitle:(out char **)theTitle;
```

同じことがオブジェクトにも当てはまります。

```
- adjustRectangle:(inout Rectangle **)theRect;
```

これらの規則はコンパイラによってではなく、実行時に適用されます。

プロキシとコピー

最後に、オブジェクトを引数として受け取るメソッドについて考えます。

```
- danceWith:(id)aPartner;
```

danceWith:メッセージは、オブジェクトidをレシーバに渡します。送信側と受信側が同じアプリケーションにある場合は、どちらも同じ*aPartner*オブジェクトを参照することができます。

受信側がリモートアプリケーションにあったとしても、これは変わりません。ただし、受信側はプロキシを通してオブジェクトを参照する必要があります（オブジェクトがそのアドレス空間に存在しないため）。danceWith:がリモートレシーバに提供するポインタは、実際にはプロキシへのポインタです。プロキシに送信するメッセージは、接続を介して実際のオブジェクトへ渡され、返される情報があれば、それらはリモートアプリケーションに返されます。

プロキシがあまりにも非効率的である場合、オブジェクトのコピーをリモートプロセスに送信して、そのアドレス空間で直接対話できるようにしたほうがよい場合があります。このような意図を示す方法をプログラマに提供するために、Objective-Cにはbycopy型修飾子が用意されています。

```
- danceWith:(bycopy id)aClone;
```

bycopyは戻り値にも使用できます。

```
- (bycopy)dancer;
```

この型修飾子をoutと併用すると、参照が返すオブジェクトをプロキシの形で提供するのではなく、コピーするように指示できます。

```
- getDancer:(bycopy out id *)theDancer;
```

注： オブジェクトのコピーを別のアプリケーションに渡す際は、匿名にはできません。オブジェクトを受信するアプリケーションでは、オブジェクトのクラスをそのアドレス空間にロードする必要があります。

bycopyは、クラスによっては非常に有用であるため（たとえば、ほかのオブジェクトのコレクションを含むことを目的としたクラスなど）、そのようなクラスでは、通常の参照の代わりにコピーをリモートレシーバに送信するように記述されている場合があります。しかし、このような動作をbyrefでオーバーライドして、メソッドから返されるメソッドやオブジェクトに渡すオブジェクトを、参照で渡したり返すように指定することもできます。Objective-Cオブジェクトのほどんどでは参照渡しがデフォルトの動作であるため、byrefキーワードを実際に使用することはほとんどありません。

bycopyまたはbyrefで変更する意味のある唯一の型はオブジェクトです。idで動的に型定義するか、クラス名で静的に型定義するかは関係ありません。

bycopyとbyrefはクラスおよびカテゴリ宣言内では使用できませんが、形式プロトコル内では使用できます。たとえば、形式プロトコルfooは次のように記述できます。

```
@Protocol foo
- (bycopy)array;
@end
```

以降、クラスまたはカテゴリでプロトコルfooを採用することができます。これによって、プロトコルに記述されたメソッドがオブジェクトをどのように渡したり返したりするべきかを示す「ヒント」を提供するプロトコルの構築が可能になります。

C++とObjective-Cの併用

AppleのObjective-Cコンパイラでは、C++のコードとObjective-Cのコードを同じソースファイルに混在させて記述することが可能です。このようなObjective-CとC++のハイブリッド言語は、Objective-C++と呼ばれています。Objective-C++を使えば、Objective-Cアプリケーションから既存のC++ライブラリを利用することができます。

Objective-CとC++の機能の混在

Objective-C++では、C++コードとObjective-Cメソッドのどちらの言語からでもメソッドを呼び出すことができます。どちらの言語でも、オブジェクトのポインタは単なるポインタであるため、どこでも使用できます。たとえば、Objective-Cオブジェクトのポインタを C++クラスのデータメンバとして含めることができ、C++オブジェクトのポインタをObjective-Cクラスのインスタンス変数として含めることができます。リスト 14-1にこれを示します。

注： Xcodeでは、Objective-C++拡張をコンパイラで利用できるように、拡張子「.mm」を持ったファイル名にする必要があります。

リスト 14-1　C++とObjective-Cインスタンスをインスタンス変数として使用

```
/* Hello.mm
 * Compile with:g++ -x objective-c++ -framework Foundation Hello.mm  -o hello
 */

#import <Foundation/Foundation.h>
class Hello {
    private:
        id greeting_text;  // NSStringを保持
    public:
        Hello() {
            greeting_text = @"Hello, world!";
        }
        Hello(const char* initial_greeting_text) {
            greeting_text = [[NSString alloc]
initWithUTF8String:initial_greeting_text];
        }
        void say_hello() {
            printf("%s\n", [greeting_text UTF8String]);
        }
};

@interface Greeting :NSObject {
    @private
        Hello *hello;
}
- (id)init;
```

```
- (void)dealloc;
- (void)sayGreeting;
- (void)sayGreeting:(Hello*)greeting;
@end

@implementation Greeting
- (id)init {
    if (self = [super init]) {
        hello = new Hello();
    }
    return self;
}
- (void)dealloc {
    delete hello;
    [super dealloc];
}
- (void)sayGreeting {
    hello->say_hello();
}
- (void)sayGreeting:(Hello*)greeting {
    greeting->say_hello();
}
@end

int main() {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    Greeting *greeting = [[Greeting alloc] init];
    [greeting sayGreeting];                          // > Hello,  world!

    Hello *hello = new Hello("Bonjour, monde!");
    [greeting sayGreeting:hello];                    // > Bonjour,  monde!

    delete hello;
    [greeting release];
    [pool release];
    return 0;
}
```

Objective-Cインターフェイスで C構造体を宣言できるように、Objective-Cインターフェイスで C++クラスを宣言することもできます。C構造体と同様に、Objective-Cインターフェイスで定義した C++クラスは、Objective-Cのクラス内にネストされず、有効範囲がグローバルになります（これは、C++ではなく、標準Cにおいてネストされた構造体定義の有効範囲がファイル単位になるのと同等です）。

言語に応じてコードの条件定義が行えるように、Objective-C++コンパイラでは、C++およびObjective-C言語標準でそれぞれ規定されている、__cplusplusと__OBJC__プリプロセッサ定数が共に定義されています。

前述したように、Objective-C++ではObjective-Cオブジェクトから C++クラスを継承できず、C++オブジェクトからObjective-Cクラスを継承することもできません。

```
class Base { /* ...*/ };
@interface ObjCClass:Base ...@end // エラー!
class Derived:public ObjCClass ...// エラー!
```

Objective-Cと異なり、C++のオブジェクトは静的に型定義されており、例外的なケースとして実行時にポリモーフィズムが利用できます。そのため、2つの言語のオブジェクトモデルは、直接的には互換性がありません。さらに、より根本的な要素として、メモリ内でのObjective-Cオブジェクトと C++オブジェクトのレイアウトは相互に互換性がありません。つまり、両言語に対して有効なオブジェクトインスタンスを作成するのは、一般的に不可能です。したがって、2つのタイプの階層を混合することはできません。

Objective-Cのクラス宣言の中で、C++クラスを宣言することはできます。次のように、コンパイラは、グローバルなネームスペースで宣言されたものとして、このようなクラスを処理します。

```
@interface Foo {
 class Bar { ... } // OK
}
@end

Bar *barPtr; // OK
```

Objective-Cでは、（Objective-C宣言の中で宣言されたかどうかに関係なく）C構造体をインスタンス変数として使用できます。

```
@interface Foo {
   struct CStruct { ...};
   struct CStruct bigIvar; // OK
} ... @end
```

Mac OS X 10.4以降では、`fobjc-call-cxx-cdtors`コンパイラフラグを設定すれば、仮想関数およびユーザ定義の引数なしコンストラクタとデストラクタを含むC++クラスのインスタンスを、インスタンス変数として使用できます（`fobjc-call-cxx-cdtors`コンパイラフラグは、gcc-4.2ではデフォルトで設定されます）。コンストラクタは、それらをメンバとして含むObjective-Cオブジェクトが割り当てられた直後に、宣言の順番に、`alloc`メソッド（正確には、`class_createInstance`内）で呼び出されます。使われるコンストラクタは、「パブリックな引数なしのインプレースコンストラクタ」です。デストラクタは、それらをメンバとして含むObjective-Cオブジェクトが割り当て解除された直後に、宣言とは逆の順番に、`dealloc`メソッド（正確には、`object_dispose`内）で呼び出されます。

Mac OS X v10.3以前のバージョン： 次の注意は、Mac OS X v10.3およびそれ以前のバージョンにのみ適用されます。

Objective-C++も同様に、C++クラスインスタンスをインスタンス変数として機能させることを目指しています。該当するC++クラス（およびそのスーパークラスすべて）が仮想メンバ関数を定義しないかぎり、これは可能です。仮想メンバ関数がある場合、当該C++クラスはObjective-Cのインスタンス変数として機能しません。

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

struct Class0 { void foo(); };
struct Class1 { virtual void foo(); };
struct Class2 { Class2(int i, int j); };

@interface Foo :NSObject {
    Class0 class0;      // OK
    Class1 class1;      // エラー!
    Class1 *ptr;        // OK—Fooのinitから'ptr = new Class1()'を呼び出し、
                        // Fooのdeallocから'delete ptr'を呼び出す
    Class2 class2;      // 警告—コンストラクタを呼び出さない!
...
@end
```

C++では、仮想関数を含むクラスの各インスタンスが、適切な仮想関数テーブルポインタを含む必要があります。しかし、Objective-Cランタイムは C++オブジェクトモデルを知らないため、仮想関数テーブルポインタを初期化することができません。同様に、Objective-Cランタイムは、それらのオブジェクトのC++コンストラクタまたはデストラクタに対して呼び出しを発行できません。C++クラスにユーザ定義コのンストラクタやデストラクタがあっても、それらは呼び出されません。そのような場合、コンパイラは警告を発します。

Objective-Cには、ネストされたネームスペースという概念がありません。C++のネームスペースではObjective-Cクラスを宣言できず、Objective-Cクラスでネームスペースを宣言することもできません。

Objective-Cクラス、プロトコル、およびカテゴリはC++テンプレート内で宣言することはできません。また、C++テンプレートをObjective-Cインターフェイス、プロトコル、またはカテゴリの有効範囲内で宣言することもできません。

しかし、Objective-Cクラスは、C++テンプレートのパラメータとして使用することが可能です。また、Objective-Cメッセージ式では、C++テンプレートのパラメータを（セレクタとしてではなく）レシーバまたはパラメータとして使用することもできます。

C++の曖昧性と競合

すべてのObjective-Cプログラムがインクルードする必要のあるObjective-Cヘッダファイルには、いくつかの識別子が定義されています。これらの識別子は、`id`、`Class`、`SEL`、`IMP`、`BOOL`です。

Objective-Cメソッドの内部では、コンパイラはC++のキーワード`this`と同様に、識別子`self`と`super`を事前宣言します。しかし、C++のキーワード`this`とは異なり、`self`と`super`はコンテキスト依存であるため、Objective-Cメソッドの外部でも通常の識別子として使用できます。

プロトコル内のメソッドのパラメータリストには、さらに5つのコンテキスト依存キーワードがあります(oneway、in、out、inout、bycopy)。これらは、ほかのコンテキストではキーワードになりません。

Objective-Cプログラマの観点から見ると、C++には新しいキーワードが多数追加されています。それでもC++キーワードはObjective-Cセレクタの一部として使用できるため、影響はそれほど大きくありません。しかし、Objective-Cクラスやインスタンス変数の指定にC++キーワードを使用することはできません。たとえば、classがC++キーワードであったとしても、依然としてNSObjectメソッドclassを使用できます。

```
[foo class]; // OK
```

ただし、これはキーワードであるため、classを変数名として使用することはできません。

```
NSObject *class; // エラー
```

Objective-Cでは、クラスとカテゴリの名前は別々のネームスペースに属します。つまり、@interface fooと@interface(foo)は、同じソースコード内に存在できます。Objective-C++では、C++クラスや構造体の名前と同じカテゴリを持つこともできます。

プロトコルとテンプレートの指定子は、異なる用途に同じ構文を使用します。

```
id<someProtocolName> foo;
TemplateType<SomeTypeName> bar;
```

このような曖昧さを避けるために、コンパイラではidをテンプレート名として使用することを認めていません。

最後に、次のように、ラベルの後にグローバル名を使用した式を続けた場合に、C++の構文解釈上の曖昧さが生じます。

```
label: ::global_name = 3;
```

最初のコロンの後のスペースは必須です。Objective-C++にも同様の場合があり、やはりスペースが必須です。

```
receiver selector: ::global_c++_name;
```

制限事項

Objective-C++はObjective-Cのクラスに C++の機能を追加するものではなく、C++のクラスにObjective-Cの機能を追加するものでもありません。たとえば、Objective-C構文を使用してのC++オブジェクトの呼び出し、Objective-Cオブジェクトへのコンストラクタやデストラクタの追加、キーワードのthisとselfの置き換え使用などを行うことはできません。クラス階層も異なります。C++のクラスはObjective-Cのクラスを継承できませんし、Objective-Cのクラスも C++のクラスを継承することはできません。さらに、複数言語の例外処理もサポートされていません。つまり、Objective-Cコードでスローされた例外はC++コードではキャッチできず、逆に、C++コードでスローされた例外もObjective-Cコードではキャッチできません。Objective-Cの例外に関する詳細については、「例外処理」（107 ページ）を参照してください。

言語の要約

Objective-Cは、C言語にいくつかの構成体を加え、ランタイムシステムと効果的にやり取りするためのいくつかの規則を定義したものです。ここでは、追加されたすべての要素をリストしますが、詳細については取り上げません。詳細については、この文書のほかの章を参照してください。.

メッセージ

メッセージ式は、次のように大括弧で囲んで表現します。

```
[receiver message]
```

receiverは次のいずれかです。

- オブジェクトとして評価される変数または式（変数`self`も含みます）
- クラス名（クラスオブジェクトを示します）
- `super`（メソッド実装検索の代替の起点を示します）

messageは、メソッドの名前と、メソッドに渡される任意の引数です。

定義済みの型

Objective-Cで使われる主要な型は、`objc/objc.h`に定義されています。次のものが含まれます。

型	定義
`id`	オブジェクト（そのデータ構造体へのポインタ）。
`Class`	クラスオブジェクト（クラスデータ構造体へのポインタ）。
`SEL`	セレクタ、すなわちコンパイラによって割り当てられる、メソッド名を識別するコード。
`IMP`	`id`を返すメソッド実装へのポインタ。
`BOOL`	ブール値。`YES`または`NO`のどちらか。 `BOOL`の型は`char`です。

`id`を使用して任意の種類のオブジェクト、クラス、インスタンスを型定義できます。さらに、クラス名を型名として使用して、クラスのインスタンスを静的に型定義することができます。静的に型定義されたインスタンスは、そのクラスへのポインタ、またはその継承元である任意のクラスへのポインタになるように宣言されます。

objc.hヘッダファイルには、次に示す有用なキーワードも定義されています。

型	定義
nil	nullオブジェクトポインタ、(id)0。
Nil	nullクラスポインタ、(Class)0。
NO	falseブール値、(BOOL)0。
YES	trueブール値、(BOOL)1。

プリプロセッサのディレクティブ

プリプロセッサは、次の特殊な表記を解釈します。

表記	定義
#import	ヘッダファイルを読み込みます。このディレクティブは#includeと同じですが、同じファイルを2回以上はインクルードしません。
//	行末まで続くコメントを開始します。

コンパイラのディレクティブ

コンパイラへのディレクティブは“@”から始まります。次のディレクティブを使って、クラス、カテゴリ、プロトコルの宣言と定義を行います。

ディレクティブ	定義
@interface	クラスまたはカテゴリのインターフェイスの宣言を開始します。
@implementation	クラスまたはカテゴリの定義を開始します。
@protocol	形式プロトコルの宣言を開始します。
@end	クラス、カテゴリ、プロトコルの宣言または定義を終了します。

次の相互に排他的なディレクティブは、インスタンス変数の可視性を指定します。

ディレクティブ	定義
@private	インスタンス変数の有効範囲を、それを宣言しているクラスに限定します。
@protected	インスタンス変数の有効範囲を、宣言しているクラスと派生クラスに限定します。

ディレクティブ	定義
`@public`	インスタンス変数の有効範囲についての制約を除去します。

デフォルトは`@protected`です。

次のディレクティブは、例外処理をサポートします。

ディレクティブ	定義
`@try`	例外をスローできる範囲を示すブロックを定義します。
`@throw`	例外オブジェクトをスローします。
`@catch()`	先述の`@try`ブロック内でスローされた例外をキャッチします。
`@finally`	例外が先述の`@try`ブロックでスローされたかどうかに関係なく実行されるコードのブロックを定義します。

次のディレクティブは宣言済みプロパティ機能をサポートします（「宣言済みプロパティ」（71 ページ）を参照）。

ディレクティブ	定義
`@property`	宣言済みプロパティの宣言を開始します。
`@synthesize`	このディレクティブに続く名前を持つプロパティに対応するアクセサメソッドのカスタム実装が存在しない場合に、そのアクセサメソッドを生成するようにコンパイラに指示します。
`@dynamic`	このディレクティブに続く名前を持つプロパティに対応するアクセサメソッドの実装が見つからない場合に、警告を出さないようにコンパイラに指示します。

さらに、次のような特定の目的のためのディレクティブがあります。

ディレクティブ	定義
`@class`	ほかの場所で定義されたクラスの名前を宣言します。
`@selector(method_name)`	*method_name*に指定されたメソッドを識別するコンパイル済みのセレクタを返します。
`@protocol(protocol_name)`	*protocol_name*プロトコル（Protocolクラスのインスタンス）を返します（`@protocol`は、前方宣言の場合は (*protocol_name*) なしでも有効です）。
`@encode(type_spec)`	*type_spec*に指定された型構造体をエンコードする文字の並びを生成します。

ディレクティブ	定義
`@"string"`	現在のモジュールの中で`NSString`オブジェクト定数を定義し、指定された文字列でオブジェクトを初期化します。 Mac OS X v10.4以前では、この文字列は7ビットのASCIIでエンコードされていなければなりません。Mac OS X v10.5以降（Xcode 3.0以降を含む）では、UTF-16でエンコードされた文字列も使用できます（Mac OS X v10.2以降のランタイムは、UTF-16でエンコードされた文字列をサポートします。したがって、Mac OS X v10.5を使用してMac OS X v10.2以降用のアプリケーションをコンパイルすると、UTF-16でエンコードされた文字列を使用できます）。
`@"string1" @"string2" ... @"stringN"`	現在のモジュールの中で、`NSString`オブジェクト定数を定義します。作成される文字列は、2つのディレクティブで指定された文字列を結合した結果です。
`@synchronized()`	一度に1つのスレッドによってのみ実行されなければならないコードのブロックを定義します。

クラス

新しいクラスは、`@interface`ディレクティブを使用して宣言します。そのクラスのスーパークラスのインターフェイスファイルをインポートする必要があります。

```
#import "ItsSuperclass.h"

@interface ClassName :ItsSuperclass < protocol_list >
{
    instance variable declarations
}
method declarations
@end
```

コンパイラのディレクティブとクラス名以外は、すべて省略可能です。コロンとスーパークラス名を省略すると、そのクラスは新しいルートクラスとして宣言されます。プロトコルをリスト指定した場合は、それらが宣言されているヘッダファイルもインポートする必要があります。

クラス定義を含んでいるファイルは、自身のインターフェイスをインポートします。

```
#import "ClassName.h"

@implementation ClassName
method definitions
@end
```

カテゴリ

カテゴリは、クラスとほとんど同じ方法で宣言します。クラスを宣言しているインターフェイスファイルをインポートする必要があります。

```
#import "ClassName.h"

@interface ClassName ( CategoryName ) < protocol list >
method declarations
@end
```

プロトコルリストとメソッドの宣言は省略可能です。プロトコルをリスト指定した場合は、それらが宣言されているヘッダファイルもインポートする必要があります。

クラス定義と同様に、カテゴリ定義を含んでいるファイルは、自身のインターフェイスをインポートします。

```
#import "CategoryName.h"

@implementation ClassName ( CategoryName )
method definitions
@end
```

形式プロトコル

形式プロトコルは、@protocolディレクティブを使って宣言します。

```
@protocol ProtocolName < protocol list >
declarations of required methods
@optional
declarations of optional methods
@required
declarations of required methods
@end
```

取り込むプロトコルのリストとメソッドの宣言は省略可能です。プロトコルは、それが取り込むプロトコルを宣言しているすべてのヘッダファイルをインポートする必要があります。

@optionalディレクティブは、それに続くメソッドが任意であることを指定します。@requiredディレクティブは、それに続くメソッドが、そのプロトコルを採用するクラスによって実装されなければならないことを指定します。デフォルトは@requiredです。

次の方法で@protocolディレクティブを使って、プロトコルの前方参照を作成できます。

```
@protocol ProtocolName;
```

ソースコード内では、プロトコルは同様の@protocol()ディレクティブを使って参照します（括弧内にプロトコル名を指定します）。

不等号括弧（<...>）で囲んで指定したリストのプロトコル名には、次の3つの異なる用途があります。

- プロトコルの宣言においては、ほかのプロトコルを組み込む（前述の通り）
- クラスまたはカテゴリの宣言においては、プロトコルを採用する（「クラス」（128 ページ）および「カテゴリ」（128 ページ）を参照）
- 型の指定においては、型を、そのプロトコルに準拠するオブジェクトに限定する

プロトコルの宣言内では、次の型修飾子がリモートメッセージングをサポートします。

型修飾子	定義
oneway	対象メソッドは非同期メッセージ用であり、有効な戻り値の型はありません。
in	対象引数は、リモートのレシーバに情報を受け渡します。
out	対象引数は、参照によって返される情報を取得します。
inout	対象引数は、情報の受け渡しと取得の両方を行います。
bycopy	オブジェクトの代理ではなくコピーを渡します（または返します）。
byref	オブジェクトのコピーではなく参照を渡します（または返します）。

メソッドの宣言

メソッドの宣言においては、次の規則が使用されます。

- クラスメソッドの宣言の前には「+」を付けます。
- インスタンスメソッドの宣言の前には「-」を付けます。
- 引数と戻り値の型は、型キャストのためのCの構文を使って宣言します。
- 引数はコロン（:）の後に宣言します。

  ```
  - (void)setWidth:(int)newWidth height:(int)newHeight
  ```

  通常は、引数を説明するラベルをコロンの前に付けます。次の例は、有効ですが望ましくない形式といえます。

  ```
  - (void)setWidthAndHeight:(int)newWidth :(int)newHeight
  ```

  ラベルもコロンも、メソッド名の一部であると見なされます。
- メソッドのデフォルトの戻り値と引数の型は、関数の場合のintではなく、idです（ただし、型の指定を続けずに修飾子unsignedを使用した場合は、必ずunsigned intの意味になります）。

メソッドの実装

メソッドの実装には次の2つの隠し引数が渡されます。

- 受信側オブジェクト(self)。
- メソッドのセレクタ(_cmd)。

実装内では、selfとsuperはどちらも受信側のオブジェクトを参照します。その実装によって継承されているメソッドだけをメッセージに対応して実行させることを示す場合、メッセージのレシーバとしてselfの代わりにsuperを使用します。

ほかに有効な戻り値のないメソッドは、通常は`void`を返します。

非推奨構文

次の構文は、メソッドに非推奨のマークをつけるために使用します。

```
@interface SomeClass
-method __attribute__((deprecated));
@end
```

または

```
#include <AvailabilityMacros.h>
@interface SomeClass
-method DEPRECATED_ATTRIBUTE;  // または、ほかのデプロイメントターゲット固有のマクロ
@end
```

この構文は、Objective-C 2.0以降でのみ使用可能です。

命名規則

Objective-Cソースコードが記述されているファイルの名前には、拡張子`.m`が付きます。クラスとカテゴリのインターフェイスを宣言しているファイル、またはプロトコルを宣言しているファイルには、ヘッダファイルの標準的な拡張子である`.h`が付きます。

クラス、カテゴリ、プロトコルの名前は一般的に大文字から始まり、メソッドとインスタンス変数の名前は通常は小文字から始まります。インスタンスを保持する変数の名前も、通常は小文字から始まります。

Objective-Cでは、同一の名前でも用途が異なれば競合は生じません。クラス内では、名前は自由に割り当てることができます。

- クラスにおいては、ほかのクラスにあるメソッドと同じ名前のメソッドを宣言できます。
- クラスにおいては、ほかのクラスにある変数と同じ名前のインスタンス変数を宣言できます。
- インスタンスメソッドには、クラスメソッドと同じ名前を付けることができます。
- メソッドには、インスタンス変数と同じ名前を付けることができます。
- 1文字のアンダースコア（_）で始まるメソッド名は、Appleでの使用のために予約されています。

同様に、同じクラスのプロトコルとカテゴリには、保護されたネームスペースがあります。

- プロトコルには、クラス、カテゴリ、その他のものと同じ名前を付けることができます。
- 1つのクラスのカテゴリには、ほかのクラスのカテゴリと同じ名前を付けることができます。

ただしクラス名は、グローバル変数および定義済みの型と同じネームスペースにあります。プログラムには、クラスと同じ名前のグローバル変数が存在することはできません。

書類の改訂履歴

この表は「*Objective-C プログラミング言語*」の改訂履歴です。

日付	メモ
2009-10-19	関連参照についての説明を追加しました。
2009-08-12	細かい誤りをいくつか修正しました。
2009-05-06	Objective-CとC++の混在についての章を更新しました。
2009-02-04	カテゴリについての説明を更新しました。
2008-11-19	いくつかのセクションを新しいランタイムガイドに移動したことに伴って、大幅に再編成しました。
2008-10-15	誤植を修正しました。
2008-07-08	誤植を修正しました。
2008-06-09	細かいバグ修正と説明の明確化（特に「プロパティ」の章）。
2008-02-08	プロパティの説明に加筆し可変オブジェクトを含めました。
2007-12-11	細かい誤りをいくつか修正しました。
2007-10-31	辞書の高速列挙の例を示し、プロパティの説明を強化しました。
2007-07-22	Objective-C 2.0の新機能を説明する文書への参照を追加しました。
2007-03-26	細かい誤植を修正しました。
2007-02-08	nilへのメッセージ送信の説明を分かりやすくしました。
2006-12-05	コードリスト3-3の説明を分かりやすくしました。
2006-05-23	メモリ管理の説明を『Memory Management Programming Guide for Cocoa』に移動しました。
2006-04-04	細かい誤植を修正しました。
2006-02-07	細かい誤植を修正しました。
2006-01-10	クラスで使用するグローバル変数の静的な指定子の使用法を分かりやすくしました。
2005-10-04	nilへのメッセージ送信の効果を分かりやすくしました。コンパイラに対し、Objective-C++であることを示す".mm"拡張子の使用方法を記述しました。

日付	メモ
2005-04-08	言語の構文仕様の誤植を修正し、サンプルコードを修正しました。
	外部宣言のプロトコル宣言リストの宣言に関する構文を修正しました。
	リスト 14-1 （119 ページ）の例を分かりやすくしました。
2004-08-31	関数とデータ構造体の参照を削除しました。例外と同期の構文を追加しました。技術的な修正と若干の編集上の変更をしました。
	関数とデータ構造体の参照を『*Objective-C Runtime Reference*』へ移動しました。
	「スレッド実行の同期」にスレッド同期方法の例を追加しました。
	「クラスオブジェクトの初期化」で、`initialize`メソッドの呼び出しのタイミングを分かりやすくし、メソッドを実装するためのテンプレートを記述しました。
	「構文」に、例外と同期の構文を追加しました。
	文書全体をとおして、`conformsTo:`を`conformsToProtocol:`に置き換えました。
2004-02-02	「例外ハンドラ」の誤植を修正しました。
2003-09-16	`id`の定義を修正しました。
2003-08-14	「例外処理とスレッド同期」に、Mac OS X version 10.3以降で利用可能なObjective-Cの例外と同期のサポートについて文書化しました。
	「コンパイラのディレクティブ」（126 ページ）に定数文字列を連結するための言語サポートについて文書化しました。
	「オブジェクトの所有権」を「オブジェクトの保持」の前に移動しました。
	`Ivar`構造体と`objc_ivar_list`構造体の説明を修正しました。
	`class_getInstanceMethod`と`class_getClassMethod`の*関数の結果*のフォントを変更しました。
	用語解説の「*準拠（conform）*」の定義を修正しました。
	`method_getArgumentInfo`の定義を修正しました。
	文書の名前を『*Inside Mac OS X: The Objective-C Programming Language*』から『*The Objective-C Programming Language*』に変更しました。
2003-01	定数文字列を宣言するための言語サポートについて文書化しました。誤植をいくつか修正しました。索引を追加しました。
2002-05	Mac OS X 10.1にObjective-C++のコンパイラが導入され、Objective-CクラスからのC++要素の呼び出し、およびその逆が可能になりました。

日付	メモ
	ランタイムライブラリのリファレンスを追加しました。
	Objective-C言語におけるインスタンス変数宣言の構文の説明に誤りがあり、修正しました。
	あいまいな表現、受身表現、古い言い回しを少なくするため、文書全体を通して、文体とセクション名を更新しました。統一性を高めるため、いくつかのセクションを再編成しました。
	文書の名前を『*Object Oriented Programming and the Objective-C Language*』から『*Inside Mac OS X: The Objective-C Programming Language*』に変更しました。

書類の改訂履歴

用語解説

抽象クラス(abstract class) もっぱらほかのクラスが継承できるように定義されたクラス。プログラムでは抽象クラスのインスタンスは使用せず、そのサブクラスだけを使用します。

抽象スーパークラス(abstract superclass) 抽象クラス(abstract class)と同じ。

採用(adopt) Objective-C言語では、あるクラスで特定のプロトコルのすべてのメソッドを実装すると宣言している場合、そのクラスはそのプロトコルを採用していると言います。プロトコルを採用するには、クラス宣言またはカテゴリ宣言で不等号括弧内にそれらの名前を列挙します。

匿名オブジェクト(anonymous object) 未知のクラスのオブジェクト。匿名オブジェクトのインターフェイスは、プロトコル宣言によって公開されます。

Application Kit アプリケーションのユーザインターフェイスを実装するCocoaフレームワーク。Application Kitは、画面上で描画を行い、イベントに応答するアプリケーションの基本プログラム構造を提供します。

アーカイブ(archiving) データ構造、特にオブジェクトを後で使用するために保存する処理。アーカイブされたデータ構造はたいていファイルに保存されますが、メモリに書き込んだり、ペーストボードにコピーしたり、別のアプリケーションに送信することができます。Cocoaでは、アーカイブは`NSData`オブジェクトへのデータの書き込みを伴います。

非同期メッセージ(asynchronous message) メッセージを受信するアプリケーションが応答するのを待たずにすぐに戻るリモートメッセージ。送信側アプリケーションと受信側アプリケーションは独立して機能するため、「同期」していません。同期メッセージ(synchronous message)も参照。

カテゴリ(category) Objective-C言語では、クラス定義のほかの部分から分離されたメソッド定義のセット。カテゴリを使用して、クラス定義をグループに分割したり、既存のクラスにメソッドを追加したりできます。

クラス(class) Objective-C言語では、特定のオブジェクトのプロトタイプ。クラス定義では、インスタンス変数を宣言し、クラスの全メンバのメソッドを定義します。同じタイプのインスタンス変数を持ち、同じメソッドにアクセスできるオブジェクトは同じクラスに属します。クラスオブジェクト(class object)も参照。

クラスメソッド(class method) Objective-C言語では、クラスのインスタンスではなくクラスオブジェクトを操作できるメソッド。

クラスオブジェクト(class object) Objective-C言語では、クラスを表し、クラスの新しいインスタンスを作成する方法を知っているオブジェクト。クラスオブジェクトはコンパイラによって作成され、インスタンス変数を持たず、静的に型定義できませんが、それ以外ではほかのすべてのオブジェクトのように動作します。メッセージ式のレシーバとして、クラスオブジェクトはクラス名によって表されます。

Cocoa Mac OS X上の高度なオブジェクト指向開発プラットフォーム。Cocoaは、主要なプログラミングインターフェイスがObjective-Cで提供されるフレームワークセットです。

コンパイル時(compile time) ソースコードをコンパイルするとき。コンパイル時になされる決定は、ソースファイルにエンコードされた情報の量や種類によって制約されます。

準拠(conform) Objective-C言語では、あるクラス（またはスーパークラス）が特定のプロトコルに宣言されているメソッドを実装している場合、そのクラスはプロトコルに準拠していると言い

ます。クラスがプロトコルに準拠すれば、インスタンスもプロトコルに準拠します。したがって、プロトコルに準拠するインスタンスは、プロトコルに宣言されているインスタンスメソッドを実行できます。

コンテンツビュー(content view) Application Kitでは、ウインドウのコンテンツ領域に関連付けられている`NSView`オブジェクト。タイトルバーと境界を除く、ウインドウの全領域。ウインドウ内にあるほかのすべてのビューは、コンテンツビューの下位の階層に配置されます。

デリゲート（委任、delegate） 別のオブジェクトに代わって同じ役割を果たすオブジェクト。

指定イニシャライザ(designated initializer) クラスの新しいインスタンスを初期化する主たる責任を持っている`init...`メソッド。各クラスは、自身の指定イニシャライザを定義するか継承します。`self`へのメッセージを通して、同じクラスにあるほかの`init...`メソッドは直接的または間接的に指定イニシャライザを呼び出し、指定イニシャライザは`super`へのメッセージを通して、スーパークラスの指定イニシャライザを呼び出します。

ディスパッチテーブル(dispatch table) メソッドセレクタと、それらが識別するメソッドのクラス固有のアドレスを関連付けるエントリを含むObjective-Cランタイムテーブル。

分散オブジェクト(distributed objects) 異なるアドレス空間にあるオブジェクト間の通信を容易にするアーキテクチャ。

動的割り当て(dynamic allocation) オペレーティングシステムがアプリケーションの起動時ではなく、動作中に必要に応じてメモリを提供するCベースの言語で使用される技法。

動的バインディング(dynamic binding) コンパイル時ではなく、実行時にメソッドをメッセージにバインドすること。メッセージに応じて呼び出すメソッド実装を探します。

動的型定義(dynamic typing) コンパイル時ではなく、実行時にオブジェクトのクラスを検出すること。

カプセル化(encapsulation) ユーザから操作の実装を抽象的なインターフェイスの背後に隠蔽するプログラミング技法。これにより、インターフェイスのユーザに影響を与えることなく、実装を更新または変更することができます。

イベント(event) 外部のアクティビティ、特にキーボードとマウスに対するユーザアクティビティに関する直接的または間接的報告。

ファクトリ(factory) クラスオブジェクト(class object)と同じ。

ファクトリメソッド(factory method) クラスメソッド(class method)と同じ。

ファクトリオブジェクト(factory object) クラスオブジェクト(class object)と同じ。

形式プロトコル(formal protocol) Objective-C言語では、`@protocol`ディレクティブで宣言したプロトコル。クラスは形式プロトコルを採用することができ、オブジェクトは実行時に形式プロトコルに準拠するかどうかの照会に対して応答することができ、インスタンスは準拠する形式プロトコルを使用して型定義できます。

フレームワーク(framework) 互いに論理的に関連するクラス、プロトコル、および関数のセットとともに、ローカライズした文字列、オンライン文書、その他の関連ファイルをパッケージ化する方法。Cocoaは、Foundationフレームワークと Application Kitフレームワークを含む多数のフレームワークを提供します。フレームワークは「キット」と呼ばれることもあります。

gdb Mac OS Xの標準デバッグツール。

id Objective-C言語では、クラスに関係なく、任意のオブジェクトのための汎用の型。`id`は、オブジェクトデータ構造体へのポインタとして定義されています。クラスオブジェクトとクラスのインスタンスに使用できます。

実装(implementation) クラスの実装を定義するObjective-Cクラス仕様の一部。このセクションでは、publicメソッドとprivateメソッドの両方（クラスのインターフェイスで宣言していないメソッド）を定義します。

非形式プロトコル(informal protocol) Objective-C言語では、カテゴリとして（通常は`NSObject`クラスのカテゴリとして）宣言するプロトコル。

Objective-C言語は形式プロトコルを明示的にサポートしていますが、非形式プロトコルは明示的にはサポートしていません。

継承(inheritance) オブジェクト指向プログラミングでは、スーパークラスがその特性（メソッドとインスタンス変数）をそのサブクラスに渡す能力。

継承階層(inheritance hierarchy) オブジェクト指向プログラミングでは、スーパークラスとサブクラスの配置によって定義されるクラスの階層。あらゆるクラス（`NSObject`などのルートクラスを除く）にはスーパークラスがあり、どのクラスもサブクラスの数には制限がありません。スーパークラスを通して、各クラスは階層の上位クラスを継承します。

インスタンス(instance) Objective-C言語では、特定クラスに属する（そのメンバである）オブジェクト。インスタンスは、クラス定義の仕様に従って実行時に作成されます。

インスタンスメソッド(instance method) Objective-C言語では、クラスオブジェクトではなく、クラスのインスタンスが使用できるメソッド。

インスタンス変数(instance variable) Objective-C言語では、インスタンスの内部データ構造の一部をなす変数。インスタンス変数はクラス定義で宣言され、そのクラスに属するかそのクラスを継承するすべてのオブジェクトの一部になります。

インターフェイス(interface) パブリックインターフェイスを宣言するObjective-Cクラス仕様の一部で、スーパークラス名、インスタンス変数、およびpublicメソッドのプロトタイプが含まれています。

Interface Builder アプリケーションのユーザインターフェイスをグラフィカルに指定できるツール。対応するオブジェクトを設定し、これらのオブジェクトと必要な独自コードとの間で簡単に接続を確立できるようにします。

イントロスペクション(introspection) オブジェクトとして自身に関する情報（そのクラスとスーパークラス、応答できるメッセージ、準拠するプロトコルなど）を明らかにできるオブジェクトの能力。

キーウインドウ(key window) キーボードイベントを受信し、ユーザアクティビティの中心となるアクティブアプリケーションのウインドウ。

リンク時(link time) 異なるソースモジュールからコンパイルしたファイルを単一のプログラムにリンクするとき。リンカの下す決定はコンパイル済みのコードによって、そして根本的にはソースコードに含まれる情報によって制約されます。

ローカライズ(localize) アプリケーションがさまざまなローカル条件の下で動作するように適合させること。特に、ユーザが選択した言語を使用できるようにすること。ローカライズでは、アプリケーションコードから言語固有および文化固有の参照を取り除き、ローカライズしたリソース（文字列、画像、サウンドなど）をインポートできるようにする必要があります。たとえば、スペイン語にローカライズされたアプリケーションでは、アプリケーションメニューに「Salir」と表示されます。イタリア語では「Esci」、ドイツ語では「Verlassen」、英語では「Quit」になります。

メインイベントループ(main event loop) イベントによって駆動されるアプリケーションの主要な制御ループ。起動から終了まで、アプリケーションはWindow Managerからキーボードまたはマウスイベントを次から次へと受け取って応答し、次のイベントがレディー状態でない場合はイベント間で待機します。Application Kitでは、`NSApplication`オブジェクトがメインイベントループを実行します。

メニュー(menu) コマンドのリストを表示する小さなウインドウ。アクティブアプリケーションのメニューだけが画面上に表示されます。

メッセージ(message) オブジェクト指向プログラミングでは、メッセージ式の中で、実行すべきことを受信側オブジェクトに伝えるメソッドセレクタ（名前）とそれに付随する引数。

メッセージ式(message expression) オブジェクト指向プログラミングでは、メッセージをオブジェクトに送信する式。Objective-C言語では、メッセージ式は大括弧で囲まれ、レシーバとその後に続くメッセージ（メソッドセレクタと引数）で構成されます。

メソッド(method) オブジェクト指向プログラミングでは、オブジェクトが実行できるプロシージャ。

多重継承(multiple inheritance) オブジェクト指向プログラミングでは、1つのクラスが複数のスーパークラスを持てること。異なるソースを継承し、別々に定義された動作を単一のクラスに結合できること。Objective-Cでは多重継承をサポートしていません。

ミューテックス（相互排他、mutex） 相互排他セマフォとも呼ばれています。スレッド実行を同期するために使用します。

名前空間(name space) すべての名前が一意的でなければならないプログラムの論理的区分。ある名前空間におけるシンボルは、別の名前空間にある同名のシンボルとは競合しません。たとえば、Objective-Cで、各クラスのインスタンスメソッドは、クラスメソッドやインスタンス変数と同様に、個別の名前空間に属します。

nil Objective-C言語では、0の値を持ったオブジェクト`id`。

オブジェクト(object) データ構造（インスタンス変数）と、データを使用したり変更したりする操作（メソッド）をまとめたプログラミング単位。オブジェクトは、オブジェクト指向プログラムの主要な構成要素。

アウトレット(outlet) 別のオブジェクトを指すインスタンス変数。アウトレットインスタンス変数は、オブジェクトがメッセージの送信先となるほかのオブジェクトを追跡する方法です。

ポリモーフィズム（多態性、polymorphism） オブジェクト指向プログラミングでは、さまざまなオブジェクトがそれぞれの方法で同じメッセージに応答できる能力。

手続き型プログラミング言語(procedural programming language) C言語のように、明確な開始と終了のある手続きのセットとしてプログラムを構成する言語。

プロトコル(protocol) Objective-C言語では、特定のクラスに関連付けられていないメソッドのグループの宣言。形式プロトコル(formal protocol)と 非形式プロトコル(informal protocol)も参照。

レシーバ(receiver) オブジェクト指向プログラミングでは、メッセージの送信先となるオブジェクト。

参照カウント(reference counting) オブジェクトの所有権を主張する各エンティティがオブジェクトの参照カウントをインクリメントし、その後でデクリメントするメモリ管理技法。オブジェクトの参照カウントがゼロになると、オブジェクトの割り当てが解除されます。この手法により、オブジェクトの1つのインスタンスを、複数のオブジェクト間で安全に共有することができます。

リモートメッセージ(remote message) あるアプリケーションから別のアプリケーションのオブジェクトに送信されるメッセージ。

リモートオブジェクト(remote object) リモートメッセージのレシーバになりうる、別のアプリケーションのオブジェクト。

実行時(runtime) プログラムが起動し、動作している時。実行時になされる決定は、ユーザの選択によって影響を受ける可能性があります。

セレクタ(selector) Objective-C言語では、オブジェクトへのソースコードメッセージで使用されるメソッドの名前、またはソースコードをコンパイルするときにその名前を置き換える固有の識別子。コンパイル済みのセレクタは`SEL`型です。

静的型定義(static typing) Objective-C言語では、クラスへのポインタとして型定義することで、インスタンスがどのようなオブジェクトかを示す情報をコンパイラに提供すること。

サブクラス(subclass) Objective-C言語では、継承階層において別のクラスより1段階下にあるクラス。時には、より一般的に別のクラスから継承する任意のクラスを指したり、動詞として別のクラスのサブクラスを定義する処理を指したりします。

スーパークラス(superclass) Objective-C言語では、継承階層において別のクラスより1段階上にあるクラス。サブクラスがメソッドやインスタンス変数を継承するクラス。

代理（サロゲート、surrogate） 別のオブジェクトの代理をし、当該オブジェクトにメッセージを転送するオブジェクト。

同期メッセージ(synchronous message) 受信側アプリケーションがメッセージへの応答を終了するまで戻らないリモートメッセージ。メッセージを送信するアプリケーションは受信側アプリ

ケーションの確認応答や戻り情報を待つので、2つのアプリケーションの「同期」が維持されます。非同期メッセージ(asynchronous message)も参照。

索引

Symbols

A

B

C

D

E

F

G

H

I

M

N

O

P

R

S

T

U

V

Y