Objective-Cプログラミング言語

Cocoa > Objective-C Language

2011-10-12

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

はじめに

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。また、第2の要素であるObjective-Cのアプリケーションフレームワーク（総称して「**Cocoa**」と呼びます）について学ぶための基礎についても説明しています。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。

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。Objective-Cによるオブジェクト指向プログラミングはANSI C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

次の各章で、標準C言語に対してObjective-C言語によって加えられたすべての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」（33 ページ）
- 「プロトコル」（47 ページ）
- 「宣言済みプロパティ」（57 ページ）
- 「カテゴリと拡張」（65 ページ）
- 「関連参照」（67 ページ）
- 「高速列挙」（71 ページ）
- 「静的な動作の実現」（73 ページ）
- 「セレクタ」（77 ページ）
- 「例外処理」（81 ページ）
- 「スレッド化」（85 ページ）

この文書の最後の用語解説では、Objective-Cとオブジェクト指向プログラミングに特有の用語の定義をまとめています。

表記規則

この文書では、リテラル文字と斜体を使用しています。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];
    ...
}
```

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

想定や規則が多数あるため、この文書をお読みください。『*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をいくつか詳しく説明し、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ことなのかを説明します。

ランタイムシステム

『*Objective-C Runtime Programming Guide*』では、Objective-Cランタイムとその使い方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。

メモリ管理

Objective-Cは、メモリ管理の3つの仕組みをサポートしています。1つは自動ガベージコレクションで、もう1つは参照カウントです。

- **自動参照カウント**（ARC、Automatic Reference Counting）。コンパイラがオブジェクトの存続期間を推論します。
- **非自動参照カウント**（MRC、Manual Reference Counting、あるいはMRR、Manual Retain/Release）。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。

  非自動参照カウントについては『*Advanced Memory Management Programming Guide*』を参照してください。
- **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*』を参照してください（この環境はiOSでは利用できません。このため、iOS 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、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。

ランタイムシステム

Objective-C言語では、可能な限り多くの決定が、**コンパイル時**と**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。このため、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。これが提供する機能について詳しくは、『*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。

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、これらの操作は、オブジェクトの**メソッド**として知られています。メソッドが作用するデータのことを**インスタンス変数**といいます（ほかの環境では、*ivars*またはメンバ変数と呼ばれることもあります）。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。

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」（38 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、矩形（Rectangleオブジェクト）であれば、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。この型は、クラスに関係なくどの種類のオブジェクトにも対応する汎用的な型であり、クラスのインスタンスとクラスオブジェクトのインスタンスに対して使用できます。

```
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。

`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、どのオブジェクトにもそれがどのクラスのインスタンスかを表す`isa`変数があります。`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。

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。ある時点で、プログラムは通常、プログラム内に含まれているオブジェクトに関する詳細な情報を検出する必要があります。`id`型指示子はこの特定の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。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（ランタイムについて詳しくは、『*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」（20 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」（24 ページ）と「静的な動作の実現」（73 ページ）を参照してください。

メモリ管理

いずれのプログラムにおいても、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3つの仕組みを提供しています。

- **自動参照カウント**（ARC、Automatic Reference Counting）。コンパイラがオブジェクトの存続期間を推論します。
- **非自動参照カウント**（MRC、Manual Reference Counting、あるいはMRR、Manual Retain/Release）。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。

  非自動参照カウントについては『*Advanced Memory Management Programming Guide*』を参照してください。
- **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*』を参照してください（この環境はiOSでは利用できません。このため、iOS 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有効範囲、すなわち「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パラメータです。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のステートメントと同様に";"が付きます。

メッセージ内のメソッド名はメソッド実装を「選択する」役割を果たすため、メッセージ内のメソッド名はしばしば**セレクタ**と呼ばれます。

メソッドは、パラメータ（*引数*とも呼ばれます）をとることもあります。単一のパラメータを持つメッセージでは、名前の後にコロン(:)が付き、コロンの直後にパラメータが付きます。

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

複数のパラメータをとるメソッドの場合、Objective-Cのメソッド名は名前の中にパラメータが挟み込まれ、その名前によって想定されるパラメータが必然的に分かるようになっています。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 setOriginX: 30.0 y: 50.0]; // 複数のパラメータを
                                        // 適切に扱っている例
```

セレクタ名には、コロンも含め、名前のすべての部分が含まれるため、上記の例のセレクタの名前は`setOriginX:y:`です。コロンが2つあるのはパラメータを2つとるためです。ただし、セレクタ名にはそれ以外のもの（戻り型やパラメータ型など）は含まれません。

重要： Objective-Cの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。いくつかのプログラミング言語では、「名前付きパラメータ」と「キーワードパラメータ」という言葉は、そのパラメータが実行時に数が異なってもよいこと、デフォルト値があること、順番が異なっていてもよいこと、および追加の名前付きパラメータを持てることを意味します。パラメータに関するこれらの特徴は、Objective-Cについては当てはまりません。

Objective-Cメソッド宣言は、大まかに言えば、単純に2つの引数を先頭に追加したC関数です（『*Objective-C Runtime Programming Guide*』の「Messaging」を参照してください）。そのため、Objective-Cのメソッド宣言の構造は、次の例に示すような、Pythonなどの言語における名前付きパラメータやキーワードパラメータとは異なります。

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

このPythonの例では、`Thing`と`NeatMode`は省略可能であるか、または呼び出し時に異なる値を持つことが可能です。

原則上は、`Rectangle`クラスは、第2パラメータのラベルなしで`setOrigin::`メソッドを実装することもでき、その場合は次のように呼び出されます。

```
[myRectangle setOrigin:30.0 :50.0]; // 複数のパラメータを適切に扱っていない例
```

構文的には間違いではありませんが、setOrigin::はメソッド名にパラメータが組み込まれていません。そのため第2パラメータは事実上ラベル付けされず、メソッドのパラメータの種類や目的を判断することが難しくなります。

メソッドは可変個のパラメータをとることもできますが、非常にまれです。付加的なパラメータは、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つの必須パラメータ(group)と3つのオプションパラメータを渡しています。

```
[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。ドット演算子は通常、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」（57 ページ）を参照）。これについては、「ドット構文」（19 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];
  ```

  ここではspouseオブジェクトが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、nilに送信されたメッセージは、structのどのフィールドにも、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方を示しています。

```
id anObjectMaybeNil = nil;

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

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パラメータ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メソッドはパラメータを持ちませんが、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、それらをパラメータ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方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。メッセージパラメータは、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。前記の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」（33 ページ）を参照してください。

ポリモーフィズム（多態性）

前述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。そのため、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メッセージを受信する各オブジェクトは、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構いません。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、関数とそのパラメータ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、Objective-Cのステートメントはさまざまな結果を得ることができます。レシーバはプログラムの実行時に決めることができます。このためレシーバの選択はユーザの操作などの要因に応じて行うことができます。

たとえば、Application Kit(**AppKit**)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、`copy`に対しては`Rectangle`とは異なる応答をすることがあります。メッセージは実行時までメソッドを選択しないため（視点を変えると、メソッドとメッセージのバインドは実行時まで行われないため）、動作におけるこれらの違いは、メソッド自身から切り離されています。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なく、可能性を列挙する必要もありません。アプリケーションのオブジェクトはそれぞれ、`copy`メッセージ独自の方法で応答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の仕組みについては『*Objective-C Runtime Programming Guide*』の「Messaging」を参照してください。

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には、アクセサメソッドを呼び出す大括弧(`[]`)の代わりとして、ドット(`.`)演算子があります。ドット構文は、Cの構造体要素にアクセスするときと同じパターンを使用します。

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

ただし、オブジェクトに関して使用される場合、ドット構文は「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。ドット構文が直接的にインスタンス変数の取得や設定を行うわけでは*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]);
```

当然ながら、アクセサメソッドを使用してオブジェクトの独自のインスタンス変数にアクセスする場合は、`self`を明示的に呼び出す必要があります。以下に例を示します。

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

またはこれと同等の

```
[self setAge: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`の**set**アクセサメソッドは呼び出され*ません*。

```
age = 10;
```

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の構造体が含まれている。
// 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"];
```

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に、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ます。

図 1-1　ドロープログラムのクラス

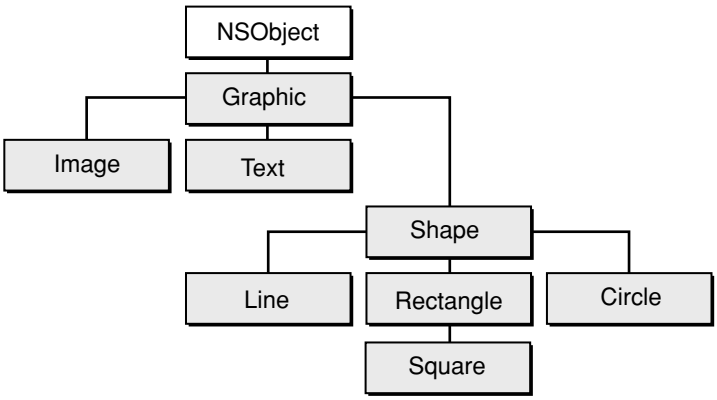


図1-1では、`Square`クラスは`Rectangle`クラスのサブクラス、`Rectangle`クラスは`Shape`のサブクラス、`Shape`は`Graphic`のサブクラス、`Graphic`は`NSObject`のサブクラスであることが示されています。継承は累積的なものです。そのため`Square`オブジェクトは、特に`Square`のために定義されたメソッドとインスタンス変数だけでなく、`Rectangle`、`Shape`、`Graphic`、`NSObject`のために定義されたメソッドとインスタンス変数も持っています。これは簡単にいえば、`Square`オブジェクトは単なる正方形であるだけでなく、型が`NSObject`の矩形(Rectangle)、図形(Shape)、グラフィック(Graphic)、オブジェクト(Object)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（矩形）をSquare（正方形）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限り）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。

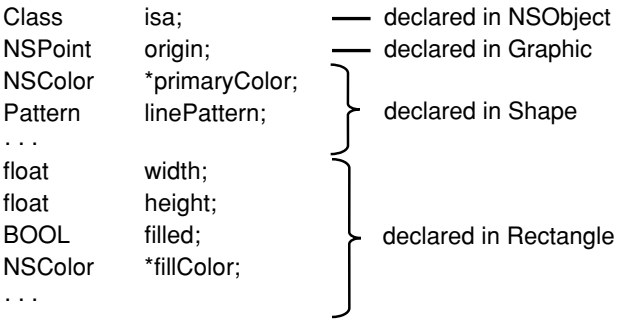
注： 新しいルートクラスを実装するのは、慎重な作業が必要で、多くの危険が潜んでいます。そのクラスは、インスタンスを割り当て、クラスに接続し、ランタイムシステムで識別するなど、NSObjectクラスが実行する多くのことを複製しなければなりません。そのため、通常は、ルートクラスとしてCocoaに提供されているNSObjectクラスを使用してください。詳細については『*NSObject Class Reference*』および『*NSObject Protocol Reference*』を参照してください。

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クラスの特定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構いません。

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`、`NSObject`のクラスで定義されたメソッドも使用できます。

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`Graphic`は`display`メソッドを定義していますが、`Rectangle`は独自のバージョンの`display`メソッドを定義することによってこれをオーバーライドしています。`Graphic`メソッドは`Graphic`クラスから派生するあらゆる種類のオブジェクトで利用できますが、`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、独自に定義した`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」 （41 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、（図 1-1 （21 ページ）のサンプルの階層に示したとおり）継承によりRectangleオブジェクトはGraphicオブジェクトの一種になるため、Rectangleインスタンスは Graphicクラスに合わせて静的に定義できます。

```
Graphic *myRectangle;
```

RectangleオブジェクトはGraphicオブジェクトであるため、ここではスーパークラスとして静的に型定義することができます。さらに、ShapeオブジェクトとRectangleオブジェクトのインスタンス変数とメソッド機能も持っているのでこれだけではありませんが、それでもRectangleオブジェクトはGraphicオブジェクトです。型チェックの目的のため、ここに示した宣言が与えられると、コンパイラはmyRectangleの型はGraphicであると見なします。しかし、実行時にmyRectangleオブジェクトがRectangleのインスタンスとして割り当てられ初期化されると、オブジェクトはそのとおりに扱われます。

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」 （73 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:メソッド、isMemberOfClass:メソッド、および関連するメソッドの詳細については、『*NSObject Class Reference*』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パラメータの型の宣言
- 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`クラスから派生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、特定の値を渡せるパラメータをとり、パラメータ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メソッドである`initWithPosition:size:`など）、初期化メソッドはすべて「`init`」から始まります。

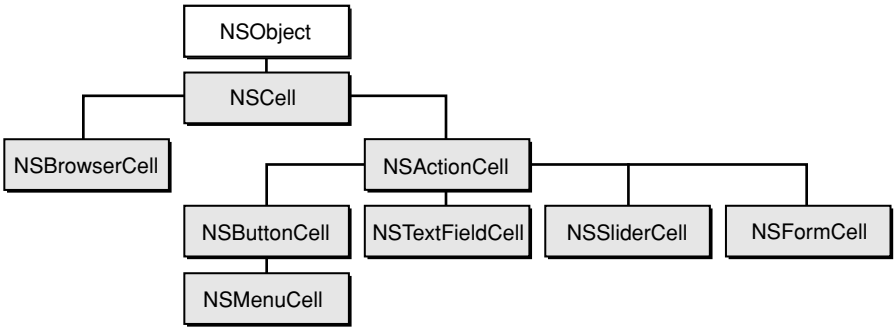
クラスオブジェクトによるカスタマイズ

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。AppKit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、AppKitによって提供される継承階層の一部を示しています。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



ある行列が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。使用者側が新しいセルを生成するメソッドを実装することになるため、作成したオブジェクトを確実に適切な型にすることができます。

しかしこの解決法では、NSMatrixクラス自身で行われるべき作業を、NSMatrixクラスの使用者に行わせることになり、クラスの数も無用に増えます。アプリケーションは複数の種類の行列を必要とし、それぞれが異なる種類のセルを持つ可能性があるため、NSMatrix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。

より優れた解決策であり、NSMatrixクラスが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クラスはまた、空のスロットを満たすためにNSMatrixが使用すべきNSCellオブジェクトを表すクラスオブジェクトを渡すsetCellClass:メソッドを定義しています。

```
[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。これを実行する最も簡単な方法は、クラス実装ファイルで変数を宣言することです。

```
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、シングルトンのようなクラスの共有インスタンスの宣言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」 in *Cocoa Fundamentals Guide*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。この一連の処理により、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 1-1 に示すテンプレートを使用します。

リスト 1-1　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のメソッドを実行するクラスオブジェクトのこの特別な能力の詳細については、『*NSObject Class Reference*』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。次に例を示します。

  ```
  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`クラスによって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」（73 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]] )
      ...
  ```

  パラメータとして単純に「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はこれを行います。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）。動的に作成されたサブクラスでは、`class`メソッドは一般にオーバーライドされ、作成されたサブクラスは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、動的サブクラスは次の不等式で表せます。

```
[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プロパティを宣言し、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）

これらの部分のそれぞれは、通常は個別のファイルに書かれます。しかし、場合によっては、*カテゴリ*と呼ばれる機能の使用を通じて、クラス定義が複数のファイルに及んでいることもあります。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」（65 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、クラスインターフェイスと実装は、通常2つの異なるファイルに書かれます。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
// メソッド宣言、プロパティ宣言。
@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」（21 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。

次に、クラス宣言の終わりまでの間に、クラスのメソッドおよびプロパティを宣言します。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。メソッドの名前をインスタンス変数と同じ名前にすることもでき、特にメソッドが変数に値を返す場合には、同じ名前にするのが一般的です。たとえば、`Circle`は、インスタンス変数`radius`と一致する`radius`というメソッドを持っているなどです。

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;
```

パラメータ型も同じ方法で宣言します。

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

戻り型やパラメータ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。

複数のパラメータがある場合は、メソッド名の中でコロンの後にパラメータを宣言します。パラメータは、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。次に例を示します。

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

可変パラメータを持つメソッドは、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, ...;
```

プロパティ宣言は次のような形式です。

```
@property (attributes) Type propertyName;
```

プロパティについては「宣言済みプロパティ」（57 ページ）で詳しく説明しています。

注： 歴史的には、インターフェイスにはクラスの**インスタンス変数**宣言が必要でした。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。これは、`@interface`宣言とメソッド宣言の間に、波括弧ではさんで記述します。

```
@interface ClassName : ItsSuperclass
{
    // インスタンス変数の宣言。
}
// メソッド宣言、プロパティ宣言。
@end
```

インスタンス変数は実装詳細であり、通常、クラス自身の外からアクセスされることはありません。さらに、実装ブロック内に宣言すること、あるいは宣言済みプロパティから自動生成させることも可能です。したがって通常は、インスタンス変数宣言をパブリックインターフェイスで行うべきではないので、波括弧も省略してください。

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
// メソッド宣言、プロパティ宣言。
@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、およびパラメータ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ディレクティブで充分です。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。
- 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。クラスにはさらに、@implementationディレクティブのあとに、波括弧ではさんでインスタンス変数を宣言しても構いません。

```
@implementation ClassName
{
    // インスタンス変数の宣言。
}
// メソッド定義。
@end
```

インスタンス変数は宣言済みプロパティで指定することもよくあります（「宣言済みプロパティ」（57 ページ）を参照）。ほかにインスタンス変数を宣言しないのであれば、波括弧は省略しても構いません。

```
@implementation ClassName
// メソッド定義。
@end
```

注： 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、スーパークラス名は省略しても支障ありません。

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。次に例を示します。

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

- (BOOL)isFilled {
    ...
}

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

可変個パラメータをとるメソッドは、関数が行うのと同じようにパラメータを処理します。

```
#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、このメソッドのパラメータ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やすくなっています。

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;
}
```

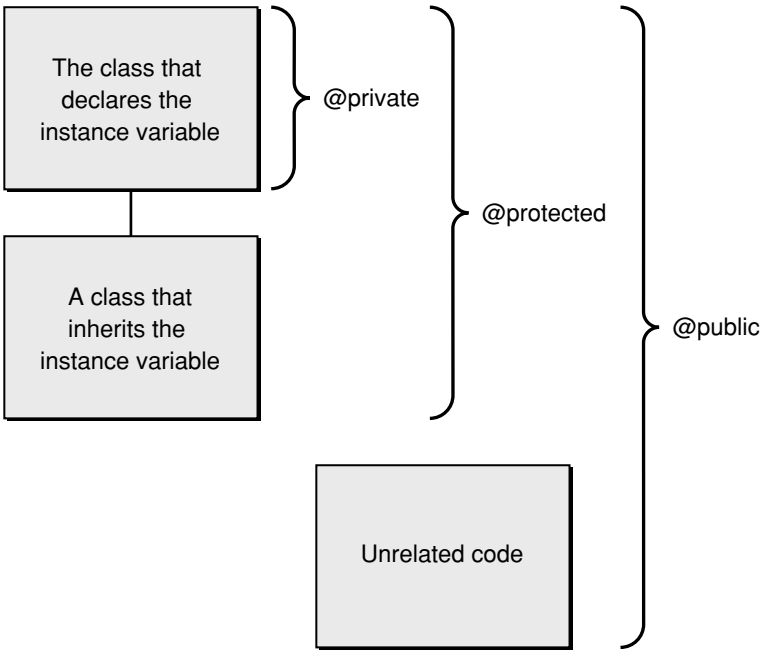
インスタンス変数の有効範囲

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4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`	インスタンス変数は、それを宣言するクラス内でのみアクセスできます。
`@protected`	インスタンス変数は、それを宣言するクラス内および継承するクラス内でアクセスできます。明示的な有効範囲ディレクティブのないインスタンス変数の有効範囲はすべて`@protected`です。
`@public`	インスタンス変数は、どこからでもアクセスできます。
`@package`	最新のランタイムを使用すると、`@package`インスタンス変数の有効範囲は、そのクラスを実装する実行可能イメージ内では`@public`であり、クラスを実装するイメージの外側では`@private`として作用します。 Objective-Cインスタンス変数の有効範囲`@package`は、Cの変数と関数の`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、すべてリンクエラとなります。 この有効範囲は、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。

図 2-1に、有効範囲のレベルを示します。

図 2-1　　インスタンス変数の有効範囲（`@package`は図示していません）



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。次に例を示します。

```
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パラメータの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。

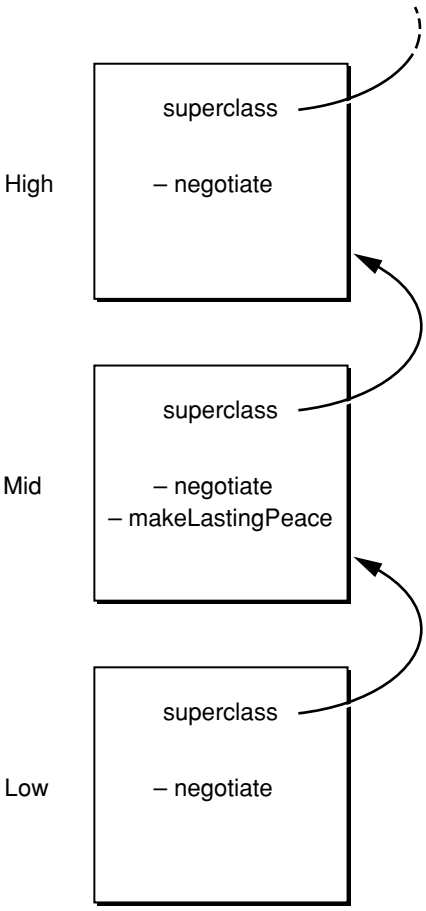
- superは、まったく異なる場所でメソッド実装の検索を開始する旨を、コンパイラに指示するフラグです。検索は、superが出現するメソッドを定義しているクラスのスーパークラスから始まります。上記の例では、repositionが定義されているクラスのスーパークラスから検索が始まります。

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の使用

selfとsuperの違いは、3つのクラスの階層を使用するときに明確になります。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であり、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、各クラスはそれぞれの目的でこのメソッドを使用します。また、Midには makeLastingPeaceという高度なメソッドを定義します。このメソッド自身は negotiateメソッドを使用します。これらのクラスとそのメソッドを、図 2-2に図示します。

図 2-2 High、Mid、Lowの階層



makeLastingPeace（Midクラス）の実装は、selfを使用して negotiateメッセージの送信先のオブジェクトを指定しているとします。

```
- makeLastingPeace
{
```

```
    [self negotiate];
    ...
}
```

あるメッセージがLowオブジェクトに送信され、makeLastingPeaceメソッドが実行されると、makeLastingPeaceはnegotiateメッセージを同じLowオブジェクトに送信します。メッセージングルーチンは、selfのクラスであるLowで定義されているバージョンのnegotiateを探します。

しかし、makeLastingPeaceの実装は代わりにsuperをレシーバに使用します。

```
- 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が定義されている場所はMidであるため、メッセージングルーチンはmakeLastingPeaceメッセージを受信したオブジェクトのクラス（Low）を無視してMidのスーパークラスへスキップします。実装ではMidバージョンのnegotiateも見つかりません。

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、superを使用することでmakeLastingPeaceは、negotiateメソッドのHighバージョン再定義したMidバージョンのnegotiateを飛び越えることができました。

説明したとおり、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、この状況下では意図したとおりの動作です。

- Lowクラスの作成者は意図的に、Midバージョンのnegotiateをオーバーライドして、Low（とそのサブクラス）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。

- Midのmake LastingPeaceメソッドの作成者は、（2番目の実装に示したように）superにnegotiateメッセージ送信することで、Midバージョンのnegotiate（およびMidから派生したLowなどのクラスで定義されるバージョン）を意図的にスキップして、Highクラスで定義されているバージョンを実行するようにしました。makeLastingPeaceの2番目の実装の設計者は、Highバージョンのnegotiateだけを使用する必要がありました。

Midバージョンのnegotiateも依然として使用される可能性はありますが、Midのインスタンスへ直接メッセージを送信することで解決できます。

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

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。割り当ては、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]; // BAD
    [self setColor:color];
    return self;
}
```

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]; // GOOD
    [newInstance setColor:color];
```

```
    return newInstance;
}
```

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]; // EXCELLENT
    [newInstance setColor:color];
    return newInstance;
}
```

初期化メソッドやその関連メソッドの実装について詳しくは、「オブジェクトの作成と初期化」を参照してください。

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との**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（恐らく多数の）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、同じクラスを継承した結果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、可能なのは`helpOut:`メソッドのプロトコルを宣言することのみです。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。メッセージは、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ることによって識別されます。

- 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

  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充分です。

非階層的な類似性

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、継承階層と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を含めることはできません。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、対応する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。クラスは`Class`のインスタンスによって、プロトコルは`Protocol`のインスタンスによって表されます。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、これは恐らくエラー
```

```
}
```

（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。次のような型宣言と、

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

次のような`conformsToProtocol:`メッセージは、

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

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。次のどちらかのテクニックを使って、組み込まれたプロトコルにクラスを準拠させることができます。

- 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。次に示すように、`Pager`が`NSObject`のサブクラスである場合、

```
@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。

```
@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手段となります。

概要

（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*』の「Mechanisms Of Abstraction」in *Object-Oriented Programming with Objective-C*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。

アクセサメソッドにはこのように大きな利点がありますが、その記述は手間のかかる作業です。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、こ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。
- 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`ブロック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ディレクティブ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」（58 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 4-1に、簡単なプロパティの宣言を示します。

リスト 4-1　簡単なプロパティの宣言

```
@interface MyClass : NSObject
@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」（58 ページ）で説明します）。

プロパティ宣言はクラス拡張に記述することもできます（「拡張」（66 ページ）を参照）。たとえば先に挙げたvalueプロパティは、次のように宣言できます。

```
@interface MyClass : NSObject
@end

@interface MyClass ()
@property float value;
@end
```

これはプライベートプロパティの宣言を隠蔽したい場合に有用です。

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（「プロパティの実装ディレクティブ」（61 ページ）を参照）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、パラメータはとりません。

`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パラメータを1つとり、`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している名前を指定する必要があります（『*Key-Value Coding Programming Guide*』を参照）。この`getter`デコレータを使う一般的な理由の1つには、ブール値の`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。この属性はデフォルトです。

  `@implementation`ブロックではgetterとsetterの両方のメソッドが必須です。実装ブロックで`@synthesize`ディレクティブを使う場合は、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`ブロックではgetterメソッドだけが必須です。実装ブロックで`@synthesize`ディレクティブを使う場合は、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。

`strong`
: 対象オブジェクトに対する強い（所有権を伴う）関係がある旨を指定します。

`weak`
: 対象オブジェクトに対する弱い（所有権を伴わない）関係がある旨を指定します。

  対象オブジェクトが割り当て解除されると、プロパティ値は自動的に`nil`になります

  （弱いプロパティは、OS X v10.6やiOS 4では未対応なので、代わりに`assign`を使ってください）。

`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
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。

`assign`
setterで、単純代入を使用することを指定します。この属性はデフォルトです。
この属性は、`NSInteger`、`CGRect`などのスカラ型に使います。

`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
以前の値には`release`メッセージが送信されます。
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。

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

アトミック性

この属性を使用して、アクセサメソッドがアトミックでないことを指定できます（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であることを指定します。*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。

`strong`、`copy`、`retain`のいずれかを指定し、`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。次のように、プロパティを破棄して`__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__((...));
```

プロパティがアウトレット（iOSについては「outlet」、OS Xについては「outlet」を参照）であることを指定する場合は、IBOutlet識別子を使用します。

```
@property (nonatomic, weak) 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。アウトレットプロパティの宣言について詳しくは、「Nibファイル」を参照してください。

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。用意しないと、コンパイラは警告を発します。

@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ディレクティブを使います。@synthesizeディレクティブがあれば、インスタンス変数が宣言されていなくても自動的に生成されます。

リスト 4-2 @synthesizeの使用

```
@interface MyClass : NSObject
@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」（58 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ディレクティブでは、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アクセサ合成の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」（64 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のインスタンス変数が存在しており、なおかつその型がプロパティの型と互換性のある型であれば、そのインスタンス変数が使われます。それ以外の場合、コンパイラエラとなります。

- 最新のランタイム（『*Objective-C Runtime Programming Guide*』の「Runtime Versions and Platforms」を参照）では、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。このキーワードは、適切な実装が見つからない場合に発せられるはずのコンパイラの警告を抑止します。そのためこのキーワードは、メソッドが実行時に利用可能であることが分かっている場合のみ使用するべきです。

リスト 4-3に、`NSManagedObject`のサブクラスでの`@dynamic`の使用例を示します。

リスト 4-3　NSManagedObjectでの@dynamicの使用

```
@interface MyClass : NSManagedObject
@property(nonatomic, retain) NSString *value;
@end

@implementation MyClass
@dynamic value;
@end
```

`NSManagedObject`は、Core Dataフレームワークに含まれています。マネージドオブジェクトクラスには、そのクラスの属性と関係を定義するスキーマが対応付けられており、実行時にCore Dataフレームワークが必要に応じてこれらに対するアクセサメソッドを生成します。そのため、通常は属性と関係のプロパティを宣言しますが、アクセサメソッドをデベロッパ側で実装する必要はなく、コンパイラにもアクセサメソッドの実装を依頼してはなりません。しかし、実装を用意せずにプロパティを単に宣言すると、コンパイラは警告を発する可能性があります。`@dynamic`を使用すると、コンパイラの警告が抑止されます。

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」（63 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）そのサブクラスすべてにおいてプロパティの属性を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」（66 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」（63 ページ）を参照）において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
@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
```

Core Foundation

「プロパティ宣言属性」（58 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
@property(readwrite) CGImageRef myImage;
@end

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

参照カウント環境では、合成された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）。単純な代入はCore Foundationオブジェクトに対しては一般的に適切でないため、メソッドを合成せずにデベロッパ自身でメソッドを実装する必要があります。

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
@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
```

ランタイムの相違

プロパティの一般的な動作は、最新のラインタイムと従来のランタイムのどちらにおいてでも同じです（『*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
@property float noDeclaredIvar;
@end

@implementation MyClass
@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。カテゴリ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。カテゴリを使用すると、デベロッパの独自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、あるカテゴリの`NSArray`クラスに追加されたメソッドは、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。ただし、`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 )
// method declarations
@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。

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。Clang/LLVM 2.0コンパイラを使うと、クラス拡張でプロパティやインスタンス変数も宣言できます。

クラス拡張の代表的な用途として、読み込み専用（readonly）としてパブリックに宣言されたプロパティを、読み書き両用（readwrite）としてプライベートに宣言し直す、というものがあります。

```
@interface MyClass : NSObject
@property (retain, readonly) float value;
@end

// プライベートな拡張（通常はメインの実装ファイル内に隠蔽）。
@interface MyClass ()
@property (retain, readwrite) float value;
@end
```

（カテゴリと違い、）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ことに注意してください。

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メソッドがあるのも一般的です。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*メソッドを定義できます。次の例を参照してください。

```
@interface MyClass : NSObject
- (float)value;
@end


@interface MyClass () {
    float value;
}
- (void)setValue:(float)newValue;
@end

@implementation MyClass

- (float)value {
    return value;
}

- (void)setValue:(float)newValue {
    value = newValue;
}

@end
```

`setValue:`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`setValue:`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（Mac OS X v10.6以降で使用可）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ます。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、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つのパラメータを取ります。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。このパターンは、宣言済みプロパティの属性のパターンに似ています（「プロパティ宣言属性」 （58 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`と`objc_AssociationPolicy`を参照）。

リスト 6-1に、配列と文字列の間に関連を確立する方法を示します。

リスト 6-1　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、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];
//  (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 6-1 （67 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を呼び出します。

リスト 6-1 （67 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を呼び出すことができます。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、この関数の使用は推奨されません。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、この関数を使用してください。

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[]) {

    @autoreleasepool {
        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];

    }
    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構文

高速列挙の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」（71 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。

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。次の要素をスキップしたい場合は、`continue`を使用できます。

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。Foundationフレームワークのコレクションクラス（`NSArray`、`NSSet`、`NSDictionary`）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 objectForKey:key]);
}
```

下の例に示すように、`NSEnumerator`オブジェクトと高速列挙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 = <#Get an array#>;
NSUInteger index = 0;

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

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限り多くが、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`は何らかの種類の`Rectangle`オブジェクトに限定されます。

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オブジェクトのインスタンス変数だけでなく、Squareオブジェクト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へのメッセージと同じように、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オブジェクトに送信される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パラメータ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つの可能性については、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」（33 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から派生する）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であるとは限らないからです（図 1-2 （23 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型である場合は、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オブジェクトに自由に代入したり、idオブジェクト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型のオブジェクトを返すため、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];
```

戻り型とパラメータ型

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、戻り型とパラメータ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パラメータ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パラメータ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インスタンスとして扱います。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、コンパイラは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インスタンスが Upperクラスとして型定義されると、結果はエラーとなります。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であるためエラーが発生します。

静的な型定義は、同じ名前のメソッドの戻り型とパラメータ型は同じでなければならないという制限を解除しますが、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。

メソッドの戻り型とパラメータ型

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パラメータのデータ型を見つけます。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、戻り型とパラメータ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、パラメータ型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、初期パラメータとして`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型のオブジェクトを返します。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、AppKitはレシーバとメッセージの両方を実行時に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。

AppKit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];
```

ユーザが対応するボタンをクリックすると、ボタンセルはNSObjectプロトコルの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つのパラメータ、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。制約が課せられたメッセージングでは、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。動的なメッセージングが行われないと、Objective-Cがうまく回避している不要な複雑さを生じさせることになります。

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判定できます。パラメータ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:`ランタイム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、自身で直接応答しない場合、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、呼び出し側の視点からは、メッセージを別のオブジェクトに転送することで間接的に処理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。この構文を`NSException`、`NSError`、またはカスタムクラスと併用することで、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*』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。（このスイッチを使用すると、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の例外サポートとしては、`@try`、`@catch`、`@throw`、`@finally`の4つのコンパイラディレクティブがあります。

- 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（コード例については、「異なるタイプの例外のキャッチ」（82 ページ）を参照してください）。
- 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。
- `@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。

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 10-1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 10-1　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*』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。この場合は引数を除外すると、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」（81 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。

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。そのため、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り、プログラムに深刻な問題を引き起こす可能性があります。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 11-1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 11-1　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 11-2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 11-2　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。

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
2011-10-12	現行の宣言済みプロパティの使用に関する細かな更新と訂正を行いました。
2010-12-08	内容を編集し、表現を明確にしました。
2010-07-13	初期化パターンの改訂を反映するように更新しました。
2009-10-19	関連参照についての説明を追加しました。
2009-08-12	細かな誤りを訂正しました。
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	誤字を訂正しました。
2008-07-08	誤字を訂正しました。
2008-06-09	細かなバグ修正と説明の明確化をいくつか行いました（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	細かな誤りを訂正しました。
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	細かな誤字を訂正しました。
2007-02-08	nilへのメッセージ送信の説明を明確にしました。
2006-12-05	コードリスト3-3の説明を明確にしました。
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	細かな誤字を訂正しました。

日付	メモ
2006-02-07	細かな誤字を訂正しました。
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方法を明確にしました。
2005-10-04	nilへのメッセージ送信の効果を明確にしました。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。
2005-04-08	言語の構文仕様の誤字を訂正し、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。
	「C++とObjective-Cインスタンスをインスタンス変数として使用」の例を分かりやすくしました。
31.08.04	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*』に移動しました。
	「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。
	文書全体を通して、`conformsTo:`を`conformsToProtocol:`に置き換えました。
02.02.04	「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。
14.08.03	「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」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。

日付	メモ
	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。
	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、そのサブクラスのインスタン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。

AppKit *Application Kit*とも呼ばれます。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。AppKit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。

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における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。

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 AppKit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。

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) publicメソッド（クラスのインターフェイスで宣言されるメソッド）とprivateメソッド（クラスのインターフェイスで宣言されないメソッド）の両方を規定した、Objective-Cクラス仕様の一部。

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 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。

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。

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）とそれに付随するパラメータ。

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パラメータ）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。

ミューテックス（mutex） *mutual exclusion semaphore*（相互排他）の省略形。スレッド実行を同期するために使用されるオブジェクトです。

名前空間(namespace) 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、あるクラスのインスタンスメソッドは、そのクラスにとって一意となる名前空間に属します。同様に、あるクラスのクラスメソッドはそのクラスメソッド固有の名前空間に属し、インスタンス変数はそのインスタンス変数に固有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 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。より一般的には、別のクラスから派生したクラスを指すときに使われます。また、別のクラスのサブクラスを定義する処理を指すときに、動詞としても使われます。

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。

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。