Become an Xcoder

Start Programming the Mac Using Objective-C

By Bert Altenberg, Alex Clarke
and Philippe Moughin
日本語訳：関根 延篤

ライセンス

■CocoaLab

CocoaLabは本書の英語、中国語、アラビア語版をwikiおよびpdfドキュメントの形で、インターネット上で無償で提供しています。

http://www.cocoalab.com/cocoalab/index.php

目次

はじめに

素晴らしいCocoaアプリケーションを作るために必要な全てのツールを、アップルは無償で提供しています。Xcodeの名で知られるこれらのツールは、Mac OS Xの全てのパッケージに付属しています。またアップルの開発者向けウェブサイトからダウンロードすることもできます。Macのプログラミングを学ぶための良書がいくつか発行されていますが、それらはどれもあなたがいくらかのプログラミングの経験を持っていることを前提に書かれています。しかしこのガイドは違います。このガイドでプログラミングの基礎、特にXcodeを使ったObjective-Cプログラミングの基本を学ぶことができます。このガイドの5章までを読めば、グラフィカルユーザーインターフェース（GUI）を持たない基本的なプログラムを作れるようになります。さらにもう少し読み進めれば、ウィンドウなどのGUIを持った簡単なプログラムを作れるようになるでしょう。そしてこのガイドを読み終えた時には、もっと専門的なCocoaプログラミングについての本に進むことができます。それらの本には多くの学ぶべき知識がつまっており、プログラミングを続けるにはいずれそれらを読む必要があります。でも安心してください。このガイドはもっと気楽に読むことができます。

■このガイドの使い方

いくつかの段落は以下のようなボールド書体で書かれています。

コラム

各章を少なくとも二回以上読むことをお勧めします。一回目はボールドの部分は無視してください。二回目に読む時にボールドの部分も読んでください。一回目に読んでいる時は、興味深いコラムも逆に混乱の種になります。このように二回に分けて読むことで、学習曲線をよりなだらかにし、より学びやすくすることができます。

このガイドは複数行にまたがるプログラムコードのサンプルをたくさん含んでいます。文中の各コードがどのサンプルのものなのかは、[4]というように大括弧でくくられた数字のラベルでわかるようになっています。ほとんどのサンプルは二行以上のコードを持っており、大括弧内の二つ目の番号は行番号を示します。例えば[4.3]はサンプル[4]の三行目を指します。また長いコードの一部を抜粋するために、各コードの後ろにラベルを置いてあります。例えば以下のとおりです。

```
volume = baseArea * height;     // [4.3]
```

プログラミングは簡単な作業ではありません。かなりの忍耐と、このガイドで学んだ全てを自分で実際に試してみることが必要です。ピアノの弾き方や車の運転の仕方は、本を読むだけでは学ぶことはできません。プログラミングの学習も同じです。この本は電子フォーマットなので、Xcodeにいつでも切り替えることができます。だから、5章以降は各章三回ずつ読むことをお勧めします。二回目に読む時は各サンプルを実際に作ってみて、さらにコードをいろいろ変更してみて、実際どのように結果が変わるか試してみてください。

00：始める前に

私たちは、あなたのために、このガイドを書きました。無償でこのガイドを提供する代わりに、Macの宣伝を少しさせてください。全てのMacユーザーはちょっとの努力で、自分の好きなコンピュータの宣伝をすることができます。こんな感じに・・・

あなたのMacをもっと便利にすれば、より多くの人をMacに惹き付けることができます。だから、Macの情報を集めたウェブサイトを訪れ、Macの雑誌を読んで、新しい情報を集めてください。もちろん、Objective-CやAppleScriptを学んでそれらを使うのも素晴らしいことです。ビジネスにおいては、AppleScriptを使うことで多くのお金と時間を節約することができます。Bertの初心者向けの無料のAppleScriptガイドを読んでみてください（http://www.macscripter.net/books）。

Macをもっと目立たせることで、全員が全員PCを使っているわけじゃないんだということを世界中にアピールしましょう。格好良いMac Tシャツを着て外を歩きまわるのも一つの方法ですが、家の中にいながらMacをアピールする方法だってあります。アクティビティモニタ（アプリケーションフォルダ内のユーティリティフォルダに入っています）を起動すると、あなたのMacはCPUの全パワーをほんのたまにしか使いきっていないことに気がつくでしょう。世界の科学者たちはいくつかの分散コンピューティングプロジェクト（DC）、例えばFolding@home（http://folding.stanford.edu/japanese/）やSETI@home（http://www.planetary.or.jp/setiathome/home_japanese.html）などを行っています。それらのプロジェクトはこのような、使われていないCPUパワーを利用して、公共の利益に役立てています。まずはDCクライアントと呼ばれる小さな、無料のプログラムをダウンロードして起動してみてください。これらのDCクライアントは最も低い優先順位で動作します。もしあなたがMac上で何かのプログラムを実行し、そのプログラムが全CPUパワーを必要とするときは、DCクライアントは直ちに停止します。なのであなたはDCクライアントが動いていることに気がつかないでしょう。では、これがどうしてMacの役に立つのでしょうか？ほとんどのDCプロジェクトは作業グループの処理量ランキングをウェブサイトに乗せています。もしあなたがMacチームに参加したら（ランキング上に名前があることに気付くと思いますが）、Macチームの順位を上げる手助けになります。そしてMac以外のコンピュータのユーザーは、Macがどれだけ役に立っているか気がつくことでしょう。世界には数学や、難病治療など様々なDCプロジェクトがありますので、http://distributedcomputing.info/projects.htmlで好みのプロジェクトを選んでみてください。ただし一つだけ問題があります。DCへの参加ははまりすぎて中毒してしまうことです!

Macを世界最高のソフトを有するプラットフォームにしましょう。すごいソフトを自分自身で作り出す、それだけではありません。あなたが使っているソフトを作った開発者に対して、丁寧なフィードバックを返すことを習慣にしましょう。たとえソフトをちょっと触ってみて気に入らなかったとしても、なぜ気に入らなかったかを開発者に伝えましょう。バグレポートは、バグに遭遇した時に行った処理を出来る限り正確に伝えましょう。

あなたが使っているソフトの対価を払いましょう。Macのソフトウェア市場で利益をあげられる限り、開発者は素晴らしいソフトを作り続けます。

少なくとも三人以上のプログラミングに興味を持ちそうなMacユーザーに、このガイドとガイドをダウンロード出来る場所を教えてあげてください。もしくは彼らに以上四つのアドバイスをしてください。

OK、あなたがDCクライアントをダウンロードしている間に、勉強を始めましょうか！

01：プログラムは命令の連続

■はじめに

あなたが車の運転を習う時、同時に複数の作業をこなす練習をする必要があります。クラッチとガソリンとブレーキペダルと、それぞれについて知っていなくてはいけません。プログラミングも同じで、たくさんのことを心に留めておく必要があります。さもなくばあなたのプログラムはクラッシュしてしまうことでしょう。もしかしたらあなたは運転を学び始める前から車の内部構造を知っていたかもしれませんが、Xcodeを使ってプログラミングを始める前にこのようなアドバンテージはありません。あなたを混乱させないように、実際の開発環境についての説明は後回しにして、まずはあなたがよく知っているであろう簡単な数学についてのObjective-Cのコードから勉強を始めることにしましょう。

小学校で以下のような計算式の穴埋め問題をしたと思います。

```
2 + 6 =  ...
... = 3 * 4 （「*」記号はコンピュータで乗算（x）を表す一般的な記号です）
```

中学に入るとこのような空白を使うのは時代遅れになり、xとyと呼ばれる変数（あるいは「代数」という呼び方）を使うようになります。思い返してみると、このような記号のちょっとした変更で当時はなぜ混乱と恐怖を感じたのか、あなたは不思議に感じているかもしれません。

```
2 + 6 = x
y = 3 * 4
```

■変数

Objective-Cでも同じように変数を使用します。変数は、例えば数字のような、特定のデータを表すための便利な名前にすぎません。以下に変数に特定の値をセットする（代入する）ためのObjective-Cの命令文、すなわちコードがあります。

```
[1]
x = 4;
```

■セミコロン（;）

xという名前の変数に4という値がセットされます（「xは4である」という意味では無いので注意。あくまでも値のセット「代入」です）。命令文の最後にセミコロン（;）があることに注意してください。全ての命令文の最後にはセミコロンを置かなければいけません。なぜでしょうか？サンプル[1]のコードは機械的に見えるかもしれませんが、コンピュータはこのコードでも何をすべきか全く理解できません。あなたが入力したこのような文章を、Macが理解できる0と1の二進数に変換するためには、コンパイラと呼ばれる特殊なプログラムが必要です。人間が入力した文章を読み、理解するのはコンパイラにとってとても大変な作業です。なのであなたは特定の手がかり、例えば特定の命令がどこで終わっているのかという情報をコンパイラに与えてやる必要があります。それがセミコロンの表す意味です。

もしセミコロンを一つでもあなたのコードに置き忘れたら、そのコードはコンパイル（ソースコードをMacが実行できる形式に変換すること）できませんし、Macが実行可能なプログラムを作ることはで

きません。ただし、コンパイラがあなたのコードをコンパイルできないときはあなたに文句を言いますので、あまり心配しすぎなくても大丈夫です。後々の章で解説しますが、コンパイラはコードのどこが間違っているのか見つける手助けをしてくれます。

■変数の名前をつける

変数の名前自体はコンパイラにとっては特別な意味はありませんが、わかりやすい変数の名前はプログラムをより読みやすく、よりわかりやすくします。そしてコードの中のエラーを探す時に大きな助けになります。

プログラムのエラーは一般的にバグと呼ばれています。またバグを探し修正することをデバッグと呼びます。

そのため、本当のコードを書く際、例えばxというような、何を示しているのかわかりにくい変数の名前は避けるべきです。例えばイメージの横幅を表す変数ならば、pictureWidthという名前をつけるべきでしょう [2]。

```
[2]
pictureWidth = 8;
```

セミコロンの付け忘れがコンパイラにとって大問題になるということから、プログラミングは細部が重要だということがわかるはずです。注意すべき細かい決まり事の一つとして、コードは大文字と小文字を区別するということがあげられます。pictureWidthという変数名はpictureWIDTHやPictureWidthという名前とは同じではありません。一般的な規約に従って、私はいくつかの単語の組合せを変数名として使い、最初の単語の先頭文字は小文字で、それ以外の単語は大文字で始めるようにしています。サンプル[2]のようにです。このような決まり事を守ることで、プログラミング中に大文字小文字の間違いを減らすことができます。

変数の名前は一つの単語（時には一文字）から作り出すように気をつけてください。

プログラマは自由に変数の名前を選ぶことができますが、守るべきいくつかのルールも存在します。そのルール全てについて説明することもできますが、現時点でそれを読むのは少し退屈なことでしょう。守るべき最初のルールは、Objective-Cによって予約されている名前（つまりObjective-Cにとって特別な意味を持つ単語）を使ってはならないということです。単語の組合せで変数名を作れば（例えばpictureWidth）、その変数名は問題ありません。変数名を読みやすくするために、変数名の中で適宜大文字を使うことが推奨されます。この規則を守ることで、あなたの作るプログラムのバグを減らすことができるでしょう。

もういくつかのルールを学んでこの段落を終えることにしましょう。文字の他に数字も使うことができますが、変数名を数字から始めることはできません。またアンダースコア（ _ ）を使うこともできます。以下にいくつか変数名のサンプルを示します。

正しい変数名の例
door8k
do8or
do_or

間違った変数名の例
door 8 （空白を含んでいる）
8door （数字から始まっている）

推奨されない例
Door8 （大文字から始まっている）

■計算の中で変数を使う

これで、変数にどうやって値を与えれば良いかわかったので、計算を実行することができます。写真の面積を計算するコードを見てみましょう[3]。

```
[3]
pictureWidth=8;
pictureHeight=6;
pictureSurfaceArea=pictureWidth*pictureHeight;
```

驚いたことにコンパイラは空白の有無は無視します（ただし変数名やキーワードは除く）。コードをもっと見やすくするために、空白を使うことができます。

```
[4]
pictureWidth = 8;
pictureHeight = 6;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

次にサンプル[5]、特に最初の二行に注意して見てみましょう。

```
[5]
pictureWidth = 8;
pictureHeight = 4.5;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

■整数値（integer）と浮動小数値（float）

数値は大まかに分けて二種類に区別することができます。整数値（integer）と小数値（float）です。[5.1]と[5.2]がそれぞれの数値の例です。整数値は、例えば何かの命令を特定の回数繰り返すため回数を数える（第7章参照）ときなどに使われます。また小数値、もしくは浮動小数値は、例えば野球の打率などを表すのに使います。

サンプル[5]はおそらく正しく動きません。問題は、あなたが使おうとしている変数の名前と、その変数が持つデータのタイプ（データの「型」と呼びます）、つまり整数値か小数値かを、事前にコンパイラに伝えなければならない、という点です。専門的にいうとこれは「変数の宣言」と呼ばれます。

```
[6]
int pictureWidth;
float pictureHeight, pictureSurfaceArea;
pictureWidth = 8;
```

```
pictureHeight = 4.5;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

[6.1]の行の最初の「int」という単語は、変数pictureWidthが整数値int型であることを示しています。次の行では、変数名をカンマで区切ることで、一度に二つの変数を宣言しています。もっと詳しくいうと [6.2]の記述は二つの変数どちらも小数値float型であること示しています。この場合、pictureWidthだけを他の二つの変数と別の型にするのは少しおかしいのですが、intとfloatを掛け合わせるとその結果はfloatになるため、[6.2]ではpictureSurfaceAreaという変数をfloatとして宣言しているのだということを理解してください。

コンパイラはなぜ、変数が整数なのかそれとも小数を伴う数値なのかを知る必要があるのでしょうか？コンピュータのプログラムは、実行するときデータを保存するためコンピュータのメモリの一部を必要とします。そのためコンパイラは各変数のためにメモリ（バイト）領域を確保しています。異なった型のデータ、この場合intとfloatは、それぞれ異なったメモリの量と表現（メモリ上にどのように書込むか）が必要なため、コンパイラはそれぞれの変数に必要なだけのメモリと正しい表現を準備してやる必要があります。もしとても大きな数値やとても精度の高い小数の数値（小数の場合は値の大きい小さいではなく、より細かい数値を表現できる、ということで精度が高い低いと言います）を使っていると、どうなるでしょうか？それらの大きな値はコンパイラが準備した少しのメモリ容量に入りきらなくなってしまう？その通り。これには二つの解決方法があります。一つはint、floatよりも大きな（もしくはより精度の高い）数値を持つことができる別の変数を使うことです。多くのシステムはlong longとdoubleというより大きな変数のタイプを持っています。もしこの変数にすら入りきらない場合、二つ目の解決方法ですが、プログラマとしてその問題を警戒するのがあなたの仕事です。いずれにせよこれはこのような入門ガイドの内容にはふさわしくないのでこの辺にしておきます。

ちなみに、整数値も小数値も例えばあなたの銀行口座のように、負の値を扱うことができます。もし変数が負の値になり得ないと分かっているのであれば、その変数が扱うことができる数値の範囲を広げることができます。

[7]

```
unsigned int chocolateBarsInStock;
```

チョコバーの数に負の値はあり得ないので、ここではunsigned intを使うことができます。unsigned intタイプの変数は0以上の値を扱うことができます。

■変数の宣言

変数の宣言と同時にその変数に値を代入することも可能です。

[8]

```
int x = 10;
float y= 3.5, z = 42;
```

こうすることでタイピングの量を少しだけ減らすことができます。

■数学的演算

以前のサンプルで、乗算命令を試してみました。以下の記号（正式には演算子と呼びます）によって、基本的な数学的計算を行うことができます。

```
+  加算
-  減算
*  乗算
/  除算
```

演算子を使うことでいろいろな種類の計算を行うことができます。もしプロのObjective-Cプログラマのコードを読む機会があったら、変わった点に気がつくことでしょう。
x = x + 1;と書く代わりにプログラマは[9]や[10]のような書き方をよく使います。

```
[9]
x++;

[10]
++x;
```

どちらの意味も「xを1増やす」です。ただし場合によっては、++が変数の前にあるか後ろにあるかによって大きな違いがでてきます。[11]と[12]のサンプルを見てください。

```
[11]
x = 10;
y = 2 * (x++);

[12]
x = 10;
y = 2 * (++x);
```

[11]のコードが実行されたら、yは20に、xは11になります。つまり2*xがyに代入されたあと、xの値が1増やされます。対照的に[12.2]では、2をかける前にxの値が1増やされます。だから実行後はxは11にyは22になります。サンプル[12]は次のサンプル[13]と同じ意味です。

```
[13]
x = 10;
x++;
y = 2 * x;
```

したがってプログラマは二つの行を一行にまとめることができます。個人的には[11]のような書き方はプログラムを読みにくくすると思います。このような省略をするのもかまいませんが、そこにはバグの可能性もあるということに注意してください。

■括弧

あなたがもうすでになんとか高校を卒業した後であれば、これはもうすっかり昔の記憶になっているかもしれませんが、括弧は計算を実行する順番を指定するのに使われる、ということを思い出してください。また*と/は+と-よりも優先されます。つまり4 + 2 * 3は10になります。括弧を使うことで優先順位の低い+や-を先に実行することができます。例えば(3 + 4) * 2は14になります。

■除算

除算演算子を使うときはちょっとした注意が必要です。intに対して使う場合とfloatの場合では結果が異なるからです。次の[14, 15]の例を見てください。

```
[14]
int x = 5, y = 12, ratio;
ratio =  y / x;

[15]
float x = 5, y = 12, ratio;
ratio =  y / x;
```

[14]の例では結果は2になります。[15]のサンプルでのみ、おそらくあなたが期待しているであろう結果2.4を得ることができます。

■余り

あなたがおそらくあまり馴染みがないであろう演算子として%（余り）があります。この演算子はパーセンテージを計算するものではありません。%の結果は前の数字を後ろの数字で割ったときの余りです（もし後ろの数字が0、つまり0で割ったときの%の動作は定義されていません）。

```
[16]
int x = 13, y = 5, remainder;
remainder =  x % y;
```

変数remainderの結果は3になります。

以下にいくつか、%計算の結果を示します。

```
21 % 7 is equal to 0
22 % 7 is equal to 1
23 % 7 is equal to 2
24 % 7 is equal to 3
27 % 7 is equal to 6
30 % 2 is equal to 0
31 % 2 is equal to 1
32 % 2 is equal to 0
33 % 2 is equal to 1
34 % 2 is equal to 0
50 % 9 is equal to 5
```

60 % 29 is equal to 2

　%演算子は時に便利に使うことができますが、int型の値にのみ使うことができるという点に注意してください。

　%のよくある使い方の一つとして偶数か奇数かの判定があります。偶数であれば、%2の結果が0になります。そうでなければ奇数です。例として以下のサンプルを紹介します。

[17]
```
int anInt;
//anIntの値を何かセットする
if ( (anInt % 2) == 0)
{
	NSLog(@"anInt is even");	//anIntは偶数
}
else
{
	NSLog(@"anInt is odd");	//anIntは奇数
}
```

02: ノーコメント? それはダメ!

■はじめに

わかりやすい変数の名前を使うことで、コードをより読みやすく、わかりやすくすることができるということは既に学びました。

```
[1]
float pictureWidth, pictureHeight, pictureSurfaceArea;
pictureWidth = 8.0;
pictureHeight = 4.5;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

今のところ数行の長さしかないサンプルを使っていますが、実際にはもっともっとずっと長いコードを書くことになります。プログラムを書く際は、実際にあなたがどんなものを作りたいのかを考えるだけではなく、正しくドキュメント化されているかも考慮する必要があります。しばらくプログラミングから遠ざかったあとでコードに変更を加えたくなったとき、コードの各部分がどんな役割を果たしていて、なぜこのような順番にコードが並んでいるのかを素早く理解する（思い出す）ために、コメント（コードの中の注釈）が大きな手助けとなります。まずはコメントから書き始めるプログラマだっているくらいです。

とにかく、あなたのコードにコメントを書込む作業にいくらかの時間を割くことを強くお薦めします。コメントに費やした時間はいずれいろいろな形で取り戻せることは保証します。また、あなたのコードを別の人と共有するとき、あなたが提供するコメントによってその人は素早くあなたのコードを理解できるでしょう。

コメントを入力するには、二つのスラッシュをコメントの文章の頭に置きます。

```
//これがコメントです
```

Xcodeではコメントは普通緑色で表示されます。もしコメントが長く複数行にまたがるときは、/*と*/の間に置きます。

```
/*これが複数行に渡る
コメントです。*/
```

■コメントアウト

ここでは少しだけプログラムのデバッグについて説明します。Xcodeはデバッグのための素晴らしい機能を持っています。デバッグのための古いやり方の一つにコメントアウトがあります。あなたのコードの一部を/*　*/で囲むことで、その部分の機能を一時的に無効にする（コメントアウトする）ことが出来ます。問題のありそうな部分を無効にすることでその他の部分が正しく動作しているかどうかを確認することができ、バグの場所を探すことができます。もしコメントアウトした部分が特定の変数に値を代入する役割を持っている場合などは、変数に適当な値を代入する一行を一時的に追加することで、コードの残りの部分に問題がないか調べることができます。

■なぜコメントが必要？

　コメントの重要性はいくら強調してもし足りません。コードが何を行っているのかを普通の言葉で長々と書くことも有用です。これによりコードが何をしているのか推測する必要がなくなり、もしその部分で問題を経験したのであれば、その問題もすぐに把握することができるようになります。たとえばもし何かの本に説明されている特定の数学的機能を使っている場合、関係のあるコードの近くにその本の書誌参照を置くべきでしょう。

　時には、実際のコードを書く前にコメントを書いてしまうのも有益です。コメントを書くことであなたの考えをまとめることができ、その結果コードもより簡単なものになるでしょう。

　このガイドのサンプルコードは、本文内に多くの説明があるため、普段私が書いているようなコメントは記入していません。

03: 関数

■はじめに

今まで見てきたサンプルコードは、最も長いものでもたった5行しかありませんでした。数千行に渡るプログラムを書くのはまだまだ先のことになりますが、Objective-Cの特徴として、プログラムの構成の方法について早い段階で説明することにします。

もしプログラムが長い、切れ目のない命令の連続でできていたとしたら、バグを見つけ修正するのはとても困難になります。また、特定の同じ処理がプログラム内の複数の場所に存在する場合、そこにバグがあったら、複数の同じバグを修正しなくてはなりません。そんなとき、一つや二つは必ず見落とすものです。そのためコードを構造化し、バグの修正をより簡単にする方法について考慮すべきです。

この問題の解決方法として、機能に応じて各命令をグループ化することがあげられます。例えば円の表面積を計算する命令があったとします。このコードが正しく機能することを一度確認したら、それ以降はそこにバグがあるかどうかを見直す必要はありません。「関数」と呼ばれるコードのセットは、それぞれ名前を持ち、その名前を呼ぶことで関数が持つ機能を使うことができます。この、関数を使うという考え方はとても基礎的なもので、各プログラムは少なくとも一つは関数を持っています。main()関数です。コンパイラはまずこのmain()関数を探し、ここからプログラムの実行を開始します。

■main()関数

ではmain()関数をもっと詳しく見ていきましょう。

```
[1]
main()
{
    // main()の本体。ここにコードを書く。
}
```

[1.1]の行は関数の名前、つまりmainというのが関数の名前です。括弧が名前に続きます。mainは予約語であり、main関数は各プログラムに必ず必要です。もし自分の関数を作るときは、その関数に自由に名前をつける（mainのような予約語を除く）ことができます。括弧にも意味がありますが、それについてはあとで説明します。続いて行[1.2,1.4]には中括弧{}があります。関数のコードはこの中括弧の中に書きます。1章で使ったサンプルを見直してみましょう。

```
[2]
main()
{
    // 以下に変数の宣言
     float pictureWidth, pictureHeight, pictureSurfaceArea;
    // 変数の初期化（各変数に適当な値をセットする）
     pictureWidth = 8.0;
     pictureHeight = 4.5;
    // 実際の計算処理
     pictureSurfaceArea = pictureWidth * pictureHeight;
}
```

■初めての関数

main()関数に引き続きコードを追加する場合、デバッグがより難しくなってしまいます。このような構造化されていないコードは避けるべきだということは既に学びました。ここではもう少し構造化されたプログラムを書いてみましょう。プログラムに必須のmain()関数とは別に、circleArea()関数を作ってみます [3]。

```
[3]
main()
{
    float pictureWidth, pictureHeight, pictureSurfaceArea;
    pictureWidth = 8.0;
    pictureHeight = 4.5;
    pictureSurfaceArea = pictureWidth * pictureHeight;
}
circleArea()   // [3.8]
{
}
```

簡単ですが、[3.8]から始まる自作の関数はまだ何の機能もありません。関数を作るとき、main()関数の外に置くということをまず覚えてください。別の言葉で言い換えると、関数はネスト（nest=「入れ子」とも訳されます）されません。

新しく作ったcircleArea()関数はmain()関数から呼ばれます。関数を呼ぶ（実行する）にはどうすれば良いか[4]を見てください。

```
[4]
main()
{
     float pictureWidth, pictureHeight, pictureSurfaceArea,
            circleRadius, circleSurfaceArea; // [4.4]
     pictureWidth = 8.0;
     pictureHeight = 4.5;
     circleRadius = 5.0;// [4.7]
     pictureSurfaceArea = pictureWidth * pictureHeight;

   // ここで関数の呼び出し!
    circleSurfaceArea = circleArea(circleRadius);// [4.11]
}
（プログラムの残りの部分に関しては[3]のサンプルを見てください）
```

■引き数を渡す

[4.4]で新たに二つのfloat型の変数を追加し、[4.7]で変数circleRadiusを初期化（値を代入）しました。最も重要なのは[4.11]、circleArea()関数を読んでいる部分です。見てわかる通り、変数circleRadiusが括弧の中に置かれています。これはcircleArea()の「引き数」と呼ばれます。circleRadiusの持つ値が関数circleArea()に渡されます。そして、circleArea()が実際の計算処理を終え

ると、関数は計算結果を返します。[4.11]では関数circleArea()の結果を変数circleSurfaceAreaに代入しています。[3]で見たcircleArea()関数を[5]のように書き直してみましょう（circleArea()関数のみ以下に載せます）。

```
[5]
circleArea(float theRadius) // [5.1]
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius; // 円周率×半径の二乗　[5.4]
    return theArea;
}
```

[5.1]行で、circleArea()関数の入力（引き数）としてfloat型の値が必要であることを宣言しています。この入力を受け取ったら、その値をtheRadiusという名前の変数にセットします。そして[5.4]の計算結果を保存しておくための変数が必要なので、main()関数[4.4]と同じように、[5.3]で変数theAreaを宣言しています。変数theRadiusの宣言は[5.1]の括弧の中で終わっていることに注意してください。[5.5]では計算結果を、この関数を呼び出した元の関数に返しています（関数の結果を返り値といいます）。結果として、[4.11]で変数circleSurfaceAreaに計算結果がセットされます。

[5]の関数のサンプルにはあと一つするべきことが残っています。関数が返す値（返り値）のデータの型を指定しなくてはいけません。コンパイラがそれを必要としているからです。そこで、[6.1]のように返り値の型としてfloatを指定します。

```
[6]
float circleArea(float theRadius)  //[6.1]
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius;
    return theArea;
}
```

[6.1]の先頭のfloatは、関数によって返されるデータ（つまりここでは変数theAreaの値）がfloat型であることを示しています。main()関数[4.8]でcircleArea()から返り値を受け取る変数circleSurfaceAreaも同じ型（つまりfloat）であるということを、プログラマとしてしっかり確認してください。そうすればコンパイラからあとで文句を言われる心配はありません。

全ての関数が引き数を必要としているわけではありません。もし引き数が無くても中身が空の括弧()は必要です。

```
[7]
int throwDice()
{
    int noOfEyes;
   // 1から6の値をランダムで作るコード
    return noOfEyes;
}
```

■返り値

全ての関数が返り値を持つわけではありません。関数に返り値がない場合、void型を指定します。またreturn命令はあっても無くてもかまいません。返り値が無い関数でreturnを使う場合、returnの後には変数名や値などは書きません。

```
[8]
void beepXTimes(int x);
{
   // ビープ音をx回鳴らすコード
    return;
}
```

次のpictureSurfaceArea()のように複数の引き数を持つ関数の場合、各引き数はカンマで区切ります。

```
[9]
float pictureSurfaceArea(float theWidth, float theHeight)
{
   // コードを書く
}
```

main()関数はint型の値を返すものと決まっています。もちろんreturnが必要です。基本的にはプログラムに問題が発生しなかったことを示すため、0を返します[10.9]。main()関数の返り値はintなので、[10.1]のようにmain()の前にintを書きます。では今までのコードをまとめてみましょう。

```
[10]
int main()
{
    float pictureWidth, pictureHeight, pictureSurfaceArea,
        circleRadius, circleSurfaceArea;
    pictureWidth = 8;
    pictureHeight = 4.5;
    circleRadius = 5.0;
    pictureSurfaceArea = pictureWidth * pictureHeight;
    circleSurfaceArea = circleArea(circleRadius);     // [10.9]
    return 0;     // [10.10]
}

float circleArea(float theRadius)           // [10.13]
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius;
    return theArea;
}
```

■動くようにする

[10]を見てわかる通り、[10.1]にmain()関数が、[10.13]に自分で定義した関数があります。もしこのコードをコンパイルしようとすると、コンパイラはまだうまくコンパイルできません。[10.9]行で、circleArea()という名前の関数は知らない、と文句を言われてしまいます。なぜでしょうか？どうもコンパイラは、main()から読み始めたところ、なんだかよくわからない言葉に出くわしてしまったということのようです。そしてコンパイラはここで読むのをやめて、あなたに警告を出します。コンパイラを満足させるためには、main()の前に関数の定義（専門的には「関数プロトタイプ宣言」といいます）を追加してやればOKです[11.1]。特に難しいことはありません。行の最後にセミコロンを置くこと以外は[10.13]行と全く同じです。これで、コンパイラは関数を見つけても驚いて警告を出すことは無くなります。

```
[11]
float circleArea(float theRadius); // function declaration
int main()
{
      以降[10]に同じ
```

もうすぐこのプログラムを実際にコンパイルしますが、まずはいくつか残りの作業を片付けましょう。

プログラムを書くときは、後でコードを再利用できるように注意しましょう。このプログラムも[12]のようにrectangleArea()関数を作って、main()関数から呼ぶように変更できます。もし関数を一度しか使わない場合でも、関数にまとめるのは有益なことです。main()関数はより読みやすくなり、デバッグするときもバグがどこにあるのか、より見つけやすくなります。長々とした命令を全部チェックする必要は無くなり、中括弧で囲われた関数の中身だけをチェックすれば良くなります。

```
[12]
float rectangleArea(float length, float width)
{
    return (length * width);
}
```

このような簡単な関数の場合、[12.3]のように計算と結果を返すのを一つにまとめることができます。[10.15]ではよりわかりやすくするために、実際には不要な変数theAreaを定義しています。

この章で作成した関数はとるに足らないものですが、関数の定義そのもの（つまり最初の行）を変更しない限り、呼び出す側の関数に何の影響も与えること無く関数の内容を変更できる、という点はとても重要ですので覚えておいてください。

例えば関数内の変数の名前を自由に変更でき、また変更しても正しく動作し、他のコードに影響することもありません。誰かが作成した関数を、中で何をしているのかを知らなくても使うこともできます。その際知っておくべきことは関数をどのようにして使えば良いか、つまり：

・関数の名前
・関数がとる引き数の数、順番、データの型
・返り値として何のデータを返すか、及びそのデータの型

サンプル[12]に当てはめると

・rectangleArea
・引き数は二つ、最初の引き数が縦幅、次が横幅、型はどちらもfloat
・関数は何か返り値を返し、その型はfloat（[12.1]の行からわかります）

■変数の保護

関数内部のコードはmain()関数やその他の関数から分離されています。だからある関数の内部の変数の値は、たとえ他の関数内部で同じ名前の変数を使っていても、通常はその変更の影響を受けることはありません。

これはObjective-Cの最も重要な機能です。5章でこの機能について再び説明します。しかしその前に、サンプル[11]をXcodeで動かしてみましょう。

04: 画面に表示する

■はじめに

サンプルプログラムは大分進歩しましたが、この計算結果をどのようにして画面上に表示するかについてはまだ説明していません。Objective-C言語そのものにはそのような機能はありませんが、幸運なことに、画面に表示する関数を作った人がいるのでそれを利用しましょう。結果を画面上に表示する方法はいろいろな選択肢がありますが、ここではアップルのCocoa環境によって提供されるNSLog()を利用します。これなら何の心配も無く、結果を画面に表示することができます。

> **NSLog()関数はもともと処理結果ではなくエラーメッセージを表示するために作られています。しかしとても簡単に使うことができるため、このガイドではNSLog()を結果の表示に利用することにします。もう少しCocoaに熟達すれば、もっと凝った表示方法を使うことができるようになることでしょう。**

■NSLogを使う

NSLog()の使い方を見てみましょう。

```
[1]
int main()
{
    NSLog(@"Julia is a pretty actress.");
    return 0;
}
```

実行すると「Julia is a pretty actress.」というテキストが表示されます。「@"」と「"」の間の文を文字列（string）と呼びます。

文字列そのものの他に、NSLog()関数は現在の時間とアプリケーション名など、様々な追加情報を一緒に表示します。例えば私のシステム上では、サンプル[1]の結果は以下のように表示されます。

```
2005-12-22 17:39:23.084 test[399] Julia is a pretty actress.
```

文字列は0個以上の文字を持ちます。

注意：以下のサンプルではmain()関数の中で重要な部分のみを抜粋しています。

```
[2]
NSLog(@"");
NSLog(@" ");
```

[2.1]は0個の文字を含みます。これは空の文字列（つまり長さが0の文字列）と呼ばれます。[2.2]は見た目に反して空の文字列ではありません。この文字列は空白を一つ含んでいます。つまり長さが1の文字列です。

いくつかの特殊な文字の組合せは、文字列の中で特別な意味を持っています。このような文字の組合せをエスケープシーケンスと呼びます。

例えば、上記サンプルの文の最後の単語を改行して次の行に表示したいとき、[3.1]のように特殊な文字\nを追加します。\nは"new line character"の略です。

```
[3]
NSLog(@"Julia is a pretty \nactress.");
```

画面への表示は以下のとおりです（必要な部分だけ抜粋しています）。

```
Julia is a pretty
actress.
```

[3.1]のバックスラッシュ（\、オプションキー+¥キーで入力できます）はエスケープ文字と呼ばれ、\の後ろの文字は普通の文字ではなく特別な意味を持っている、ということをNSLog()関数に伝えています。この場合「n」は改行を意味します。

ではもしバックスラッシュそのものを表示したいとき、どうすれば良いでしょうか？その場合バックスラッシュをバックスラッシュの前にもう一つ追加してやれば良いのです。これで二番目のバックスラッシュを表示するようにNSLog()に指示することができます。サンプルは以下のとおりです。

```
[4]
NSLog(@"Julia is a pretty actress.\\n");
```

[4.1]の結果は以下のようになるでしょう。

```
Julia is a pretty actress.\n
```

■変数を表示する

今のところ、固定の文字列の表示しか行いませんでした。次は計算の結果得られた値を画面上に表示してみましょう。

```
[5]
int x, integerToDisplay;
x = 1;
integerToDisplay = 5 + x;
NSLog(@"The value of the integer is %d.", integerToDisplay);
```

NSlogの括弧の間にまず文字列があり、次にカンマと変数名があるということに注意してください。そして文字列には何やらおかしな記号%dが含まれています。バックスラッシュと同じように%文字も特別な意味を持っています。もし%の次にd（decimal number、つまり十進数の略）がある場合、その文字は実行時にカンマの次の値、つまりここでは変数integerToDisplayの値に置き換えられます。[5]の実行結果は以下のようになるはずです。

```
The value of the integer is 6.
```

floatを表示させるなら%dの代わりに%fを使います。

```
[6]
float x, floatToDisplay;
x = 12345.09876;
floatToDisplay = x/3.1416;
NSLog(@"The value of the float is %f.", floatToDisplay);
```

小数点以下何桁まで表示させるか指定することもできます。
小数点以下二桁まで表示するには、%とfの間に2を挟みます。

```
[7]
float x, floatToDisplay;
x = 12345.09876;
floatToDisplay = x/3.1416;
NSLog(@"The value of the float is %.2f.", floatToDisplay);
```

後ほど、計算の繰り返し処理を覚えたら、変数のリストを作りたいと思うかもしれません。華氏と摂氏の変換テーブルを想像してみてください。もし値をきれいに表示したいなら、固定幅の二つのコラムに値を表示したくなると思います。%とf（あるいは%とd）の間に置くことで、その幅を指定することができます。ただしあなたの指定幅が数字の実際の幅よりも狭い場合、実際の値の幅が優先されます。

```
[8]
int x = 123456;
NSLog(@"%2d", x);
NSLog(@"%4d", x);
NSLog(@"%6d", x);
NSLog(@"%8d", x);
```

サンプル[8]の出力（表示結果）は以下のとおりです。

```
123456
123456
123456
  123456
```

[8.2, 8.3]の二行では小さすぎる桁数を指定していますが、それでも全て表示できるだけの桁数がとられています。[8.5]では実際の数字よりも大きな桁を指定しているので、余分なスペースが数字の前に表示されています。

数字の幅と小数点以下の桁数を組み合わせて指定することもできます。

```
[9]
float x=1234.5678
NSLog(@"Reserve a space of 10, and show 2 significant digits.";
NSLog(@"%10.2d", x);
```

■複数の値を表示する

二つ以上の変数を表示することももちろん可能です[10.3]。ただし%dと%fが変数の型と正しく呼応しているか、きちんと確認してください。

```
[10]
int x = 8;
float pi = 3.1416;
NSLog(@"The integer value is %d, whereas the float value is %f.", x, pi);
```

■シンボルと値の対応

初心者に最も多い間違いの一つに、NSLog()等の関数で正しくない型を指定してしまうミスがあげられます。結果がおかしかったり、理由も無くプログラムがクラッシュしてしまう場合、データ型を確認してください。

例えば最初の値が間違っている場合、二つ目の値も正しく表示されないかもしれません。

```
[10b]
int x = 8;
float pi = 3.1416;
NSLog(@"The integer value is %f, whereas the float value is %f.", x, pi);
//正しくはNSLog(@"The integer value is %d, whereas the float value is %f.", x, pi);
```

結果は以下のとおりになります。

```
The integer value is 0.000000, whereas the float value is 0.000000.
```

■Foundationをリンクする

最初のプログラムを実行する前にもう一つだけ解決すべき問題が残っています。

サンプルプログラムはこの便利なNSLog()関数を、関数のコードを何も書いていないのにどうやって実行するのでしょうか？実はプログラマがヒントを与えないと、サンプルプログラムはNSLog()関数を実行できません。実行できるようにするには、NSLog()を含むいろいろな道具を集めたライブラリ（フレームワークと呼ばれます）を読み込んでやる（インポートする）必要があります。インポートするには以下の命令を使います。

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

この命令はプログラムのコードの最初に置く必要があります。ここまで学んだ内容を全てまとめると、以下のようなコードが出来上がります。次の章では実際にこのプログラムを動かしてみます。

```
[11]
#import <Foundation/Foundation.h>
float circleArea(float theRadius);
float rectangleArea(float width, float height);
int main()
{
    float pictureWidth, pictureHeight, pictureSurfaceArea,
        circleRadius, circleSurfaceArea;
    pictureWidth = 8.0;
    pictureHeight = 4.5;
    circleRadius = 5.0;
    pictureSurfaceArea = rectangleArea(pictureWidth, pictureHeight);
    circleSurfaceArea = circleArea(circleRadius);
    NSLog(@"Area of circle: %10.2f.", circleSurfaceArea);
    NSLog(@"Area of picture: %f. ", pictureSurfaceArea);
    return 0;
}

float circleArea(float theRadius)              // 初めての自作関数
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius;
    return theArea;
}

float rectangleArea(float width, float height) // ニ番目の自作関数
{
    return width*height;
}
```

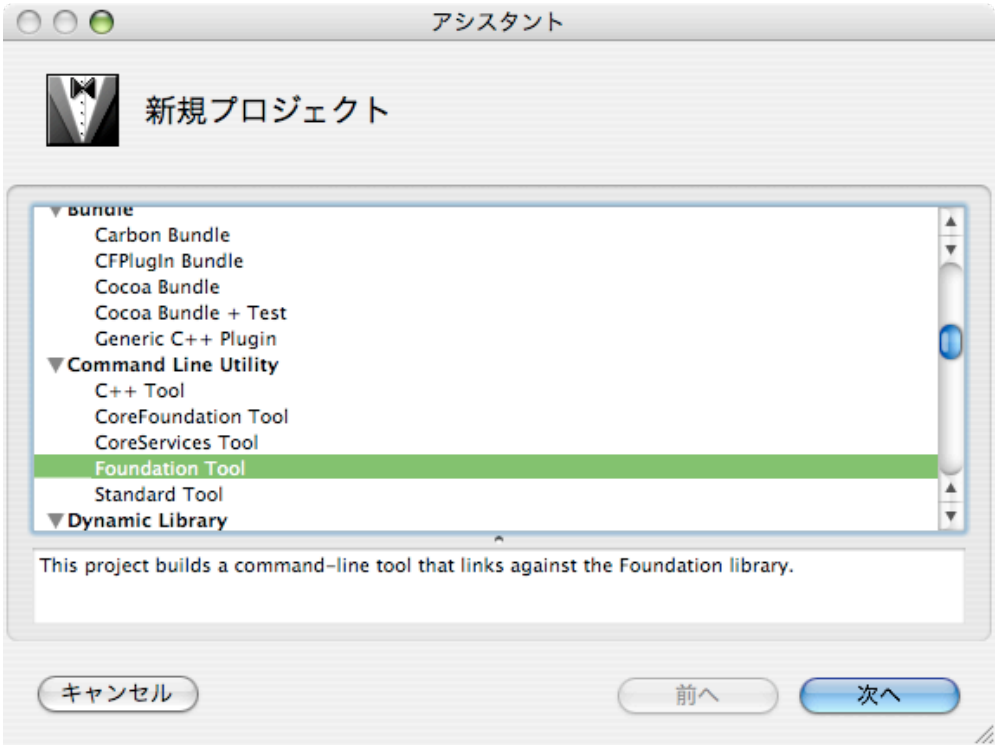
05: プログラムをコンパイルして実行する

■はじめに

ここまでで作ったコードは我々人間が読むことができる単なるテキストに過ぎません。しかしそれは完全な人間の文章とも言えず、かといってMacが理解できるものでは到底ありません。これだけではなんの価値もありません。あなたが書いたプログラムコードを、Macが実行可能なランタイムコードに変換するためには、コンパイラと呼ばれる特殊なプログラムが必要です。それはアップルが無償で提供しているXcode開発環境に含まれています。まずはMac OS XディスクからXcodeをインストールしてください。そしてhttp://developer.apple.comから常に最新版をダウンロードするように心がけてください（ダウンロードには無料の登録が必要です）。

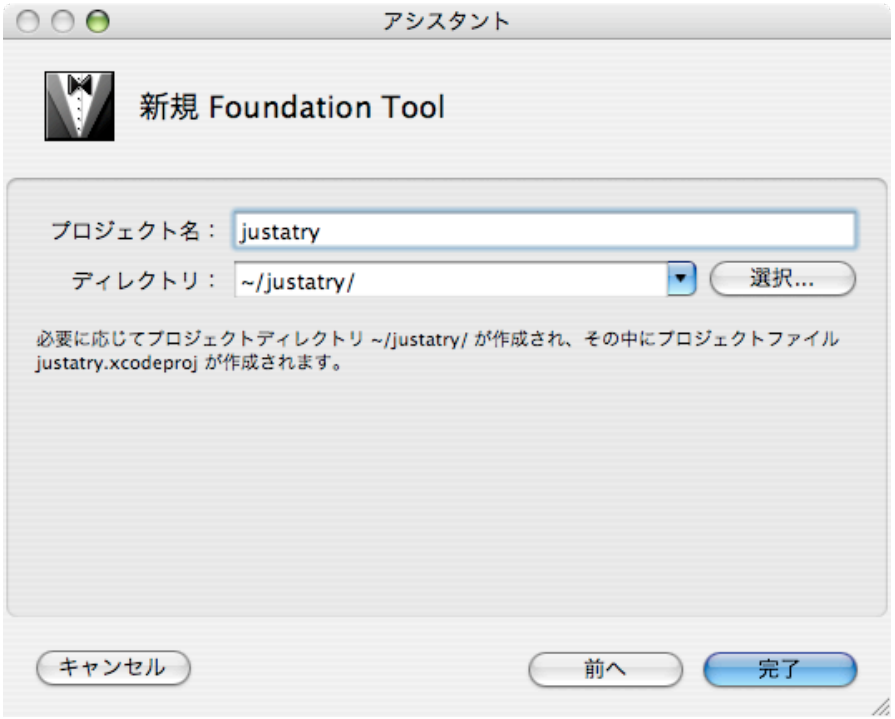
■プロジェクトを作成する

インストールしたらDeveloperフォルダ内のApplicationsフォルダに入っているXcodeを起動してください。初めて起動するときは、Xcodeはあなたに二、三の質問をするはずです。全ては環境設定で後から変更できますので、とりあえずはデフォルトの設定を選んでおけば問題ありません。そして次に「ファイル」メニューから「新規プロジェクト...」を選択します。現在使用可能なプロジェクトタイプのリストがダイアログで表示されます。



Xcodeアシスタントを使ってプロジェクトを作成する

ここではGUIを持たない簡単なObjective-Cプログラムを作りたいので、ウィンドウ下の方にある「Command Line Utility」の下から「Foundation Tool」を選択してください。



名前と保存場所を選択する

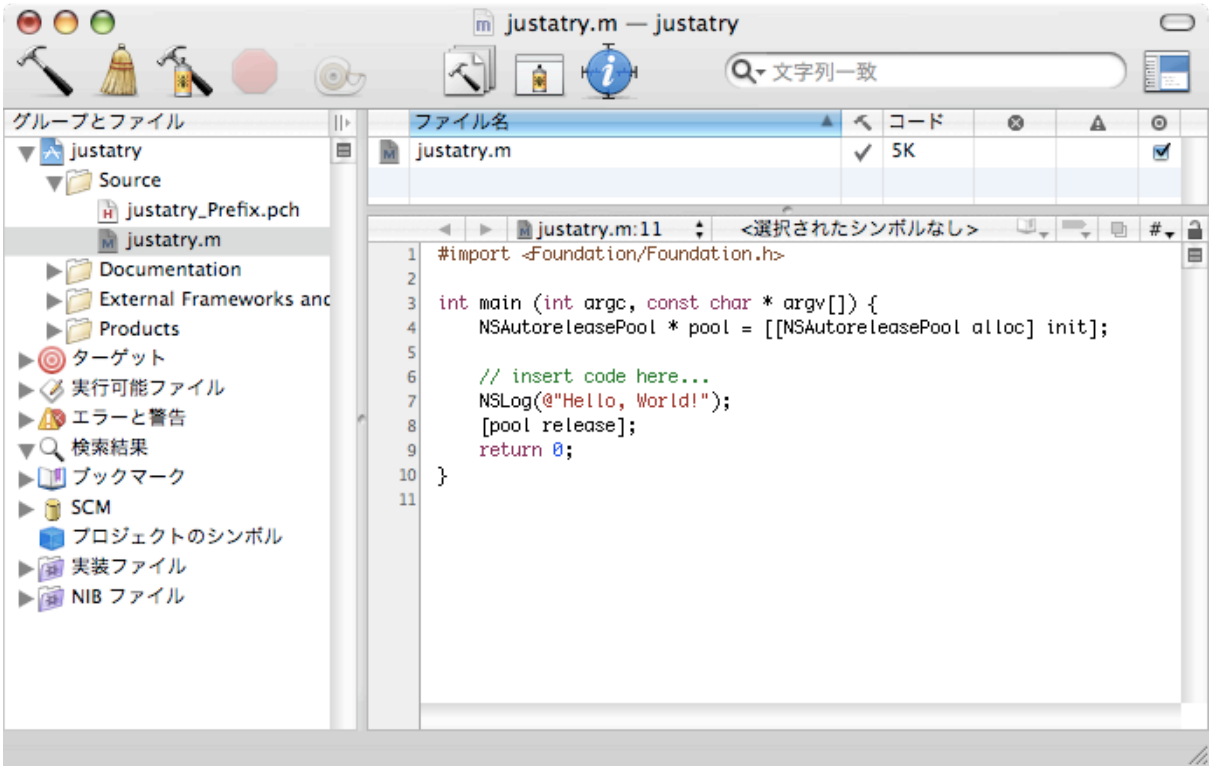
「justatry」とかなんとか、あなたのアプリケーションの名前を入れてください。次にプロジェクトファイルを保存する場所を選択したら、「完了」ボタンをクリックします。

今作ったプロジェクトはターミナルから実行することができます。もしそうしたい時、そしていくらかの手間を省きたい場合は、プロジェクトの名前は単語一つだけにしてください。また慣習的にターミナルから実行するプログラムの先頭文字は、小文字を使うことになっています。逆にGUIを持つプログラムの名前は大文字から始めます。

■Xcodeを使ってみる

さて、あなたは今、プログラマとしてこれから何度も目にすることになるウィンドウを開いています。ウィンドウには二つのフレームがあります。左側はプログラムを作り上げるための全てのファイルを操作するための「グループとファイル」フレームです。今のところそこに表示されているファイルは多くありませんが、多言語対応のGUIプログラムを作るときは、様々な言語のGUIのためのたくさんのファイルがここに置かれます。ファイルはグループ化されフォルダの中に収められますが、このフォルダはあくまでXcodeの中だけのものであり、あなたのMacのハードディスクの中に対応するフォルダはありません。Xcodeはあなたのファイル類を整理するために仮想的なフォルダ機能を提供しています。

「グループとファイル」フレームから「justatry」グループを開き、さらに「Source」というグループを開いてください。中に「justatry.m」というファイルがあります[1]。全てのプログラムはmain()という名前の関数を持っている、ということを覚えていますか？このファイルがmain()関数を持っています。この章の後半でこのコードを変更します。justatry.mファイルをダブルクリックで開いたら、あなたは少し驚くのではないかと思います。アップルはあなたのために既にmain()関数を用意してくれています。



Xcodeでmain()関数を表示しているところ

```
[1]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) // [1.3]
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];// [1.5]

// insert code here...
    NSLog(@"Hello, World!");
   [pool release];//[1.9]
    return 0;
}
```

プログラムを一眺めして、理解できそうな部分を探して見ましょう。

- ・NSLog()等の関数を使うために必要なimport命令はシャープ記号#から始まる
- ・main()関数
- ・このプログラムの中身を記述するための中括弧
- ・コードを書くように指示するコメント
- ・画面上に文字列を表示するためのNSLog()命令
- ・返り値を返すreturn 0;命令

まだあなたが理解できない行もいくつかあります。

- ・main()関数の括弧の中にある二つの妙な引き数 [1.3]
- ・NSAutoreleasePoolから始まる命令 [1.5]
- ・poolとreleaseという単語を含むもう一つの命令 [1.9]

もし自分が読者だったら、よくわからないところだらけのコードを見せられて、「あとできっとわかるから」と言われてもあまりうれしくありません。だから私はこのガイドで事前に関数の概念について説明しておきました。なので、今、わからないことはそれほど多くないのではないかと思います。

関数はプログラムを構造化する手段であり、全てのプログラムはmain()を持ち、関数とはどのようなものか、ということは既に学びました。しかし残念ながらサンプル[1]の内容を今すぐ、全て説明することはできません。申し訳ありませんが、これらの命令（つまり[1.3, 1.5, 1.9]）については当分無視してください。簡単なプログラムを書けるようになるために、覚えておくべきObjective-Cの文法が他にいくつかあります。しかし幸いなことに、既にあなたは特に難しい二つの章を既にクリアしており、次の三つの章はとても簡単です（その先再び難しい内容に入ります）。

> **もしあなたが本当に、何の説明も無しに進むのがいやであれば、この要約を読んでください。main関数の二つの引き数はプログラムをターミナルから実行するために必要になります。プログラムは実行時にメモリを確保します。プログラムの実行が終わったら、そのプログラムが確保していたメモリは、メモリを必要としている他のアプリに渡されます。あなたのプログラムが必要としているメモリを確保するのは、プログラマとしてのあなたの仕事です。また作業が終わったら確保していたメモリを解放する（他のアプリが使える状態にする）のも、同じくらい大事な仕事です。このメモリの処理がpoolを含む二つの命令が行っていることです。**

■ビルドして実行

アップルが準備してくれたプログラムをとりあえず実行してみましょう。「ビルドして実行」という名前のついた金槌のアイコンをクリックして、プログラムをビルド（コンパイル）し実行してみます。

「ビルドして実行」ボタン

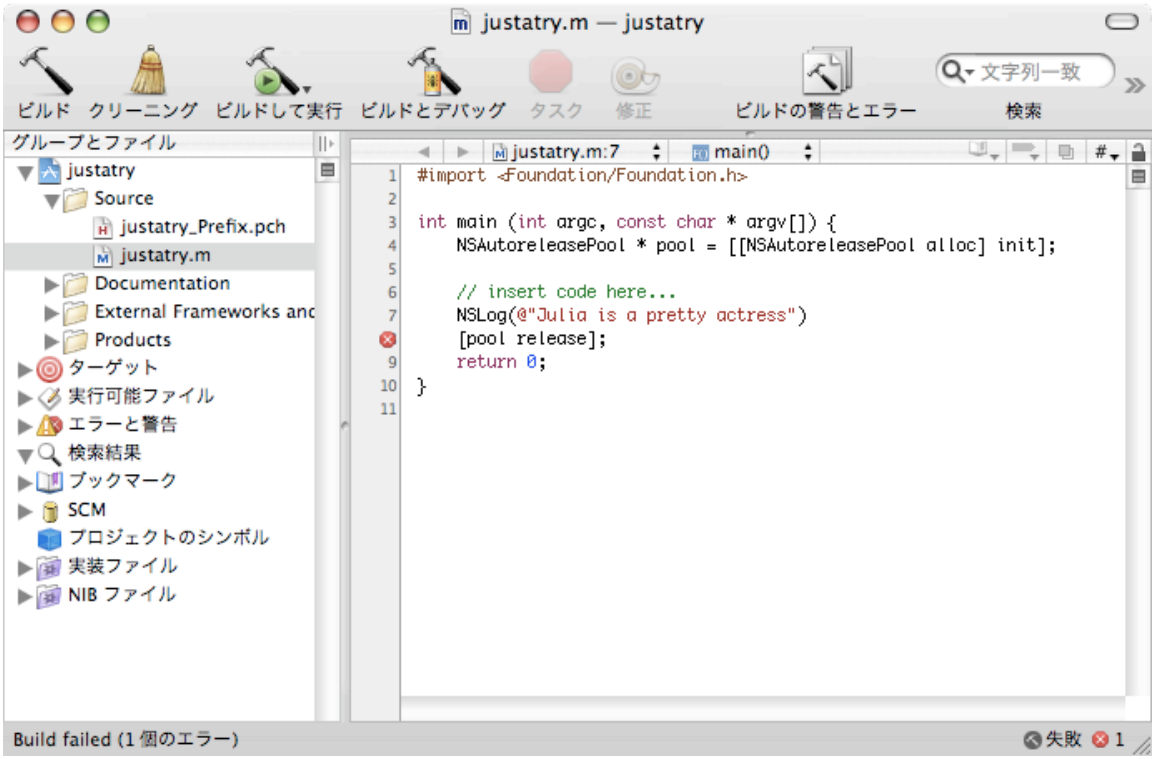
プログラムが実行され、いくつかの情報とともに結果が「実行ログ」ウィンドウに表示されます。ログの最後に「プログラムはステータス値：0 で終了しました。」と表示されプログラムは終了します。3章[7.9]で、main()関数は0を返す、と学びましたが、このプログラムも終了する前に最後の部分でちゃんと0を返しています。ここまでは問題無いようです。

■バグテスト

サンプル[1]に戻り、プログラムにバグがあったらどうなるか試してみましょう。例えばNSLog()の中身を別のものに差し替えて、最後のセミコロンをわざと打ち忘れてみます。

```
[2]
#import <Foundation/Foundation.h>

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

// insert code here...
    NSLog(@"Julia is a pretty actress")   //おっと、セミコロンを忘れた!
    [pool release]; //[2.9]
    return 0;
}
```

ツールバーの「ビルド」アイコンをクリックしてアプリケーションをビルドします。赤い丸が[2.9]行の前に表示されるはずです。



Xcodeはコンパイルでエラーが発生したことを警告している

赤丸をクリックするとウィンドウの一番下に警告の簡単な説明が表示されます。

```
error: parse error before "release".
```

「parse」はコンパイラが始めにすることで、ソースコードを最初から最後まで調べて、コンパイラが正しく理解できるかどうかをチェックする作業のことです。コンパイラがコードを正しく理解できるように、いろいろな手がかりを用意するのはあなたの責任です。たとえばimport命令にはシャープ記号#を付けてあげなくてはいけません。そして[2.8]行の終わりを明確にするために、文の最後にはセミコロン;

を付けてやる必要があります。コンパイラは[2.9]行まで来ると何かおかしいことに気がつきます。しかし、問題はこの行ではなく、セミコロンを付け忘れたその前の行にあるということまでは、コンパイラにはわかりません。ここで重要なことは、コンパイラはなるべくわかりやすいフィードバックをしようとするものの、実際のところその警告は（かなり正確に近いものの）必ずしもバグの正確な場所と内容を示しているとは限らない、ということです。

セミコロンを追加して、プログラムが今度は正しく動作することを確認してください。

■初めてのアプリケーション

では前回の章で作ったコードを持ってきて、アップルが準備してくれたコード[1]に嵌め込んでみましょう。サンプル[3]のようになります。

```
[3]
#import <Foundation/Foundation.h>

float circleArea(float theRadius);   // [3.3]
float rectangleArea(float width, float height);   // [3.4]

int main (int argc, const char * argv[])    // [3.6]
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int pictureWidth;
    float pictureHeight, pictureSurfaceArea,
            circleRadius, circleSurfaceArea;
    pictureWidth = 8;
    pictureHeight = 4.5;
    circleRadius = 5.0;
    pictureSurfaceArea = pictureWidth * pictureHeight;
    circleSurfaceArea = circleArea(circleRadius);    // [3.16]
   NSLog(@"Area of picture: %f.  Area of circle: %10.2f.",
            pictureSurfaceArea, circleSurfaceArea);
  [pool release];
   return 0;
}

float circleArea(float theRadius)          // [3.23]
{
   float theArea;
   theArea = 3.1416 * theRadius * theRadius;
    return theArea;
}

float rectangleArea(float width, float height)  // [3.30]
{
   return width*height;
```

```
}
```

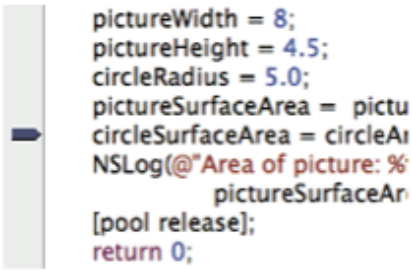
まずは時間をかけて、このプログラムの構造を把握するように勧めてください。自作の関数circleArea() [3.23]とrectangleArea() [3.30]のヘッダ（関数プロトタイプ宣言）がmain()関数 [3.6]の前に置かれています。それらの自作関数はmain()関数[3.5]の中括弧の外に置いてあります。そして以前に作ったmain()関数の中身を、アップルがそこに置くようにコメントで指示している場所に置きました。

このコードを実行すると、以下のような結果が表示されるはずです。

```
Area of picture: 36.000000.  Area of circle:      78.54.
justatry has exited with status 0.
```

■デバッグ

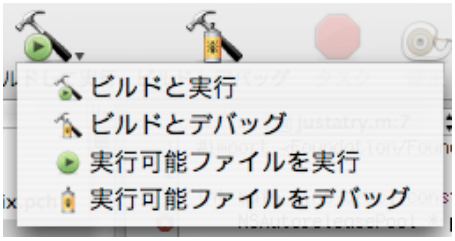
プログラムがもっと複雑になってくると、デバッグがより難しくなります。そのため、プログラムが実行中に、中で一体何が起こっているのか調べる必要が出てきます。Xcodeなら簡単にそれが出来ます。変数の値を知りたい箇所の、コード左側のグレーの余白をクリックするだけです。Xcodeは紺色の矢印アイコンをそこに挿入し、「ブレークポイント」（プログラムの実行を一時的に停止して中身をチェックするためのポイント）を設定します。ここでは以下のように[3.16]にブレークポイントを置いてみましょう。



ブレークポイントをセットする

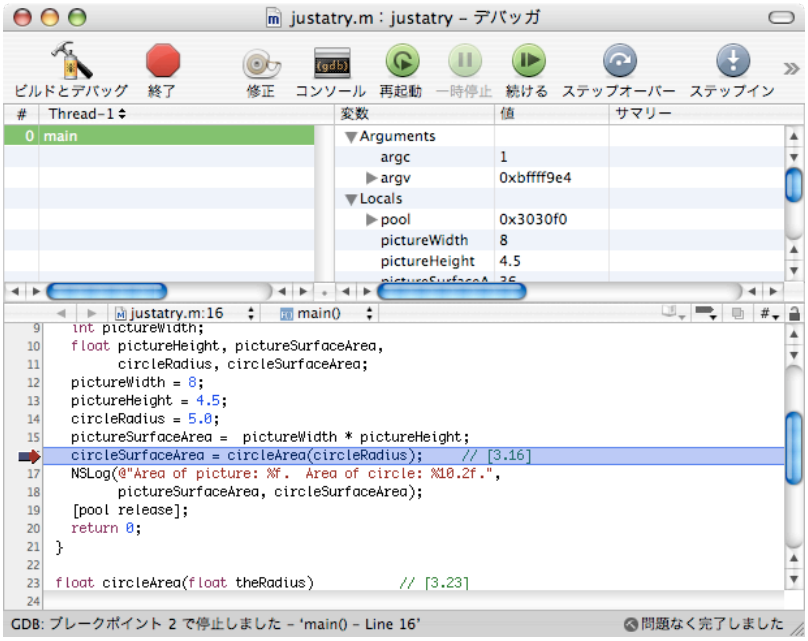
ブレークポイントで処理が停まった場合、停まった行の処理はまだ実行されていません。つまりある計算結果を確認したい場合、その計算の行にブレークポイントを置くと、停まった時にはまだ計算が実行されていない、ということに注意してください。そのため多くの場合、確認したい計算の次の行にブレークポイントを置くことになります。

それでは次に金槌アイコンをクリックしたまま長押ししてください。メニューが表示されます。



ビルドと実行ポップアップメニュー

そこで「ビルドとデバッグ」を選択してください。以下のようなウィンドウが表示されます。



Xcodeのデバッガを使うことで一行一行プログラムを実行して変数の変化を確認できる

プログラムは最初のブレークポイントまで実行されます。ウィンドウ右上のペインでは、様々な変数の値を確認することができます。ブレークポイントで停まったあとに変更された値は赤色で表示されます。プログラムを続けて実行するには「続ける」ボタンを押します。このデバッガはとても強力なツールです。しばらくいろいろ試してみて、使い方に慣れてください。

■まとめ

Mac OS X用の簡単なプログラムを書いて、デバッグして実行する内容はこれで終わりです。

もしあなたがGUIを持つプログラムを作らないのであれば、これからあなたがするべきことは、より洗練された非GUIプログラムを開発できるようにObjective-Cに関する知識を増やすことだけです。この先の数章で我々が行うのはまさにそれです。その後、ついにGUIベースのアプリケーションに挑戦します。がんばりましょう!

06: 条件分岐

■if()

特定の条件に合致するときだけある処理を行いたい場合があります。このような条件を指定するために特殊な構文があります[1.2]。

```
[1]
// ageはint型の変数で、ユーザーの年齢がセットされています
if (age > 30)    // > 記号の意味は「より大きい」
{
    NSLog(@"age is older than thirty."); //[1.4]
}
NSLog(@"Finished.");//[1.6]
```

[1.2]にif()という命令があります。これは「条件分岐」命令と呼ばれます。この括弧の中には条件に指定したい論理式（正しいか正しくないかを判定する式）を指定します。その条件に合致すれば、例えばここでは「age > 30」（変数ageの値が30より大きい）であれば、次の中括弧{}の中のコードが実行され、[1.4]の文字列が表示されます。そして[1.6]はifの中括弧の外にあるため、条件に合致してもしなくても[1.6]の文字列が表示されます。なおif()の行の最後にはセミコロン;は必要ありません。付けると動作しませんので注意してください。なおこの先のコードは、5章で作ったサンプルプログラムを使って実際に試しながら読むことをお勧めします。

■if() else()

if…else構文を使うことで、条件に合致しなかった場合の処理を指定することもできます[2]。

```
[2]
// ageはint型の変数で、ユーザーの年齢がセットされています
if (age > 30)
{
    NSLog(@"age is older than thirty."); //[2.4]
}
else
{
    NSLog(@"age is not older thirty."); //[2.8]
}
NSLog(@"Finished.");
```

条件に合致しない（変数ageが30より大きくない、つまり30以下の）場合、[2.8]の文字列が表示されます。

■比較

[2.2]の「より大きい」（>）記号の他に、以下の比較のための記号（このような記号を「比較演算子」と呼びます）を使うことができます。

```
== 等しい
 > より大きい
 <  より小さい
 >=  以上
 <=  以下
 !=  等しくない
```

「等しい」演算子は特に注意してください。=が二つです。はじめのうちはそれを忘れて=を一つだけ使ってしまうことが良くありますが、一つの=は「代入」の演算子であり、左辺に右辺の値をセットするための記号です。これは初心者によくある間違いの元です。「二つの値が同じかどうか調べるときは==を使う」ということをよく覚えておいてください。

比較の演算子は、ある処理を特定の回数だけ繰り返す時に、特に便利です。それは次の章のテーマになります。まずは、あとで役に立つかもしれないifの別の使い方について学びましょう。

■練習

比較についてもう少し詳しく学びましょう。比較演算の結果は二種類しかありません。真（正しい=true）か偽（正しくない=false）です。

Objective-CではtrueとfalseはとBOOL型と呼ばれる、trueとfalseを表現するための特殊なデータ型もあります。trueを表現するために1もしくはYESと書くこともできます。falseであれば0もしくはNOです。

```
[3]
int x = 3;
BOOL y;
y = (x == 4); // yは0になる
```

より多くの条件を指定することもできます。一つ以上の条件が全て真である（正しい）という条件を指定するには、論理積ANDを利用します。演算子は&&です。一つ以上の条件のいずれかが真であるという条件を指定するには、論理和ORを利用します。演算子は||（縦棒二つ）です。[4.1]は「ageが18以上で、かつ、ageが65より小さい」という意味です。

```
[4]
if ( (age >= 18) && (age < 65) )
{
    NSLog(@"Probably has to work for a living.");
}
```

もし

```
if ( (age >= 18) || (age < 65) )
```

と変更すると「ageが18以上もしくは、ageが65より小さい」（つまり全ての数字が合致します）という意味になります。

また条件式をネスト（組み合わせる）ことも可能です。単に、ある条件式の中括弧{}の中に別の条件式を入れるだけです。一番最初の条件が合致するかどうか調べられ（専門的には「評価する」と言います）、もしtrue（合致する）なら、続けて中の条件が評価されます。

```
[5]
if (age >= 18)
{
    if (age < 65)
    {
        NSLog(@"Probably has to work for a living.");
    }
}
```

07: 繰り返し処理

■はじめに

今まで学んできたコードは全て、各命令とも、一度きりしか実行しませんでした。もちろん[1]のように同じコードを繰り返すことで、同じ処理を繰り返すことはできます。

```
[1]
NSLog(@"Julia is a pretty actress.");
NSLog(@"Julia is a pretty actress.");
NSLog(@"Julia is a pretty actress.");
```

しかし一行以上の処理を何度も繰り返す必要が出て来ることがよくあります。他のプログラミング言語同様、Objective-Cもいくつか繰り返し処理の方法があります。

■for()

もし繰り返す回数があらかじめ分かっているのなら、サンプル[2]のように回数を直接指定することもできます。2.7回繰り返すことはできませんので、繰り返す回数は当然int型（などの整数型）で指定します。

```
[2]
int x;
for (x = 1;x <= 10;x++)  //[2.2]
{
    NSLog(@"Julia is a pretty actress.");   //[2.4]
}
NSLog(@"The value of x is %d", x);
```

サンプル[2]では、[2.4]の文字列が10回表示されます。[2.2]のfor()というのが繰り返しの命令です。まずifと同様for()の後ろにセミコロン;は必要ありません。次にfor()の括弧の中に注目してください。括弧の中はセミコロン;で区切られた三つの式に分かれています。左から「x = 1;」、「x <= 10;」、「x++」です。左から順番に見ていきましょう。まず、変数xに1が代入されます（x = 1）。そしてコンピュータは次に書かれている条件（x <= 10）を評価します。もしこの条件が真ならば[2.3]から始まる中括弧{}の中のコードが実行されます。そして実行したあと、最後にforの一番右の文「x++」が実行されます。つまりxの値が1増えます。「x++」が実行されると再びforに戻り、真ん中の条件式が評価され、真ならば（x=2なので2 <= 10は真ですね）再び{}内が実行され、という繰り返しになります。xが11になると条件x <= 10は成り立たなくなりますのでforの繰り返し（一般的にforループと呼ばれます）は終わり、最後の処理[2.6]が実行されます。最終的にxは10ではなく11であることに注意してください。

時にはx++より複雑な条件で繰り返すこともありますが、必要なのは条件式を適当なものに交換することだけです。次のサンプル[3]は華氏を摂氏に変換するプログラムです。

```
[3]
float celsius, templnFahrenheit;
```

```
for (tempInFahrenheit = 0;tempInFahrenheit <= 200;
                                        tempInFahrenheit = tempInFahrenheit + 20)
{
    celsius = (tempInFahrenheit - 32.0) * 5.0 / 9.0;
    NSLog(@"%10.2f -> %10.2f", tempInFahrenheit, celsius);
}
```

このサンプルプログラムの出力（計算結果）は以下のとおりです。

```
     0.00 ->     -17.78
    20.00 ->      -6.67
    40.00 ->       4.44
    60.00 ->      15.56
    80.00 ->      26.67
   100.00 ->      37.78
   120.00 ->      48.89
   140.00 ->      60.00
   160.00 ->      71.11
   180.00 ->      82.22
   200.00 ->      93.33
```

■while()

Objective-Cにはあと二つ、繰り返しの方法があります。

```
while () { }
```

と

```
do {} while ()
```

です。

前者は先ほど学んだforループと基本的に同じです。条件式の評価から始まります。もし評価結果がはじめから偽（false）であればループの中の処理は一度も実行されません。

```
[4]
int counter = 1; //[4.1]
while (counter <= 10)
{
    NSLog(@"Julia is a pretty actress.\n");
    counter = counter + 1;      //[4.5]
}
NSLog(@"The value of counter is %d", counter);
```

for()構文ではcounterの初期化（先のサンプルではx=1;）とcounterのインクリメント（値を1増やすこと。先のサンプルのx++）は、forの()の中でまとめて実行されましたが[2.2]、whileでは別々に実行する必要があります[4.1][4.5]。このサンプルを実行すると、counterは11になります。

do {} while () 構文では評価の前にまず中括弧の中の処理されます。つまり{}の中の処理が必ず最低一度は実行されます。

```
int counter = 1;
do
{
    NSLog(@"Julia is a pretty actress.\n");
    counter = counter + 1;
}
while (counter <= 10);
NSLog(@"The value of counter is %d", counter);
```

do{} while()構文の場合、while()のあとにはセミコロン;が必要なことに注意してください。このサンプルを実行すると、counterは11になります。

あなたのプログラミングの知識は大分増えましたので、次はもう少し難しい課題に挑戦してみましょう。次の章ではいよいよグラフィカルユーザーインターフェース（GUI）付きのプログラムを作ってみます。

08: GUIプログラム

■はじめに

ここまでObjective-Cについて多くを学んできて、そろそろグラフィカルユーザーインターフェース（GUI）付きプログラムの作り方について学ぶ準備も整いました。Objective-Cは実際のところC言語と呼ばれるプログラミング言語の拡張版です。ここまで習ってきたことのほとんどは実は普通のC言語のものです。ではObjective-CはCとどこが違うのでしょうか？違いはこの「Objective」の部分にあります。Objective-Cは「オブジェクト」という名前で知られる抽象的な概念を普通のC言語に取り入れたものです。

ここまでのところ、我々は主に数値のみを扱ってきました。今まで学習した通り、Objective-Cは数値という考え方をサポートしています。なので、メモリ上に数値を作り、数学的な関数や演算子を使ってそれらを処理できたのです。あなたが数値を処理するアプリケーションを作る時（つまり計算プログラム）、これは大きなメリットです。では、もし曲やプレイリスト、アーティストなどを扱うジュークボックスプログラムを作りたい時、果たしてどうでしょうか？あるいは飛行機やフライト、空港を管理する航空管制システムを作るときはどうすれば良いでしょう？これまであなたが数値を扱ってきたくらい簡単に、Objective-Cを使ってそれらのデータを扱うことができたら素晴らしいことですよね？

ここでオブジェクトの出番です。Objective-Cを使えば、あなたが管理したいそれらの「もの」（オブジェクト）を定義して、それらを管理し操作するアプリケーションを作ることができるのです。

■オブジェクトの動作

サンプルとして、Objective-Cで書かれたプログラム、例えばSafariなどがどのようにウィンドウを管理しているかを見てみましょう。Safariのウィンドウを開いてみてください。ウィンドウの左上に三つのボタンがあります。赤は「閉じる」ボタンです。ではもしウィンドウを閉じるために赤いボタンをクリックしたらどうなるのでしょう？「メッセージ」がボタンからウィンドウに送信されます。このメッセージに答えて、ウィンドウは自分自身を閉じるためのコードを実行するのです。

ボタンから「閉じる」メッセージがウィンドウに対して送られる

このウィンドウが「オブジェクト」です。あなたはそれをドラッグすることができます。三つのボタンも「オブジェクト」です。あなたはそれらをクリックすることができます。これらのオブジェクトは画面上の視覚表現、つまり画面上に表示される形や色を持っていますが、必ずしも全てのオブジェクトがそうではありません。例えばSafariとあるウェブサイトとの間のコネクションを表すオブジェクトなどは、画面上の視覚表現を持っていません。

あるオブジェクト（例えばウィンドウ）は別のオブジェクト（例えばボタン）を中に持つ（内包する）ことができます。

オブジェクト（ウィンドウ）は別のオブジェクト（ボタン）を内包できる

■クラス

あなたはSafariのウィンドウを好きなだけ作ることができます。あなたはアップルのプログラマがどうやってウィンドウを作っていると思いますか？

1. あなたがいくつのウィンドウを作るか、その天才的頭脳であらかじめ予想して、必要なだけのウィンドウを先にプログラムしておく

2. ひな形のようなものを準備しておいて、必要に応じてウィンドウのオブジェクトをSafariに作らせる

もちろん答えは2です。ウィンドウとは何か、どういう形でどのような動作をするのかを定義するクラスと呼ばれるコードを彼らは作成し、あなたが新しいウィンドウを開こうとしたら、このクラスにウィンドウを作らせます。クラスはウィンドウの概念（形や動作）を表現し、それぞれのウィンドウはこの概念を実体化したもの（インスタンスと呼びます）なのです。同様に「76」というのは「数値」という概念の実体化したものです。現実に例えれば、クラスというのは形や動作などを定義した「車の設計図」、そしてインスタンスというのは設計図を元に作った「車」そのものということです。

■インスタンス変数

作ったウィンドウは、Macの画面上の適当な位置に表示されます。もしウィンドウをDockにしまい、もう一度表示したら、ウィンドウは前と全く同じ位置に表示されます。これはどうやっているのでしょうか？クラスはスクリーン上の位置を覚えるための変数を定義しています。クラスのインスタンス、つまりオブジェクトは個別にこれらの変数の実際の値を持っています。なので、異なるウィンドウはそれぞれ異なる値（ウィンドウの位置など）を持っているのです。先ほどの車の例でいえば設計図には単に「色」という項目があり、実際に生産した車はそれぞれ「赤」や「黒」という別々の色情報を持っている、というイメージです。

■メソッド

クラスはウィンドウを作るだけではなく、オブジェクトが実行できる様々な処理へのアクセス方法も提供しています。これらの処理の一つに「閉じる」処理があります。ウィンドウの「閉じる」ボタンをクリックすると、ボタンはそのウィンドウオブジェクトに対し「閉じる」メッセージを送ります。オブジェクトによって実行されるこの処理のことを「メソッド」と呼びます。見ればわかると思いますが、それは関数ととてもよく似ています。だから、これからメソッドについて学びますが、大きな困難は感

じないはずです。

■メモリ内のオブジェクト

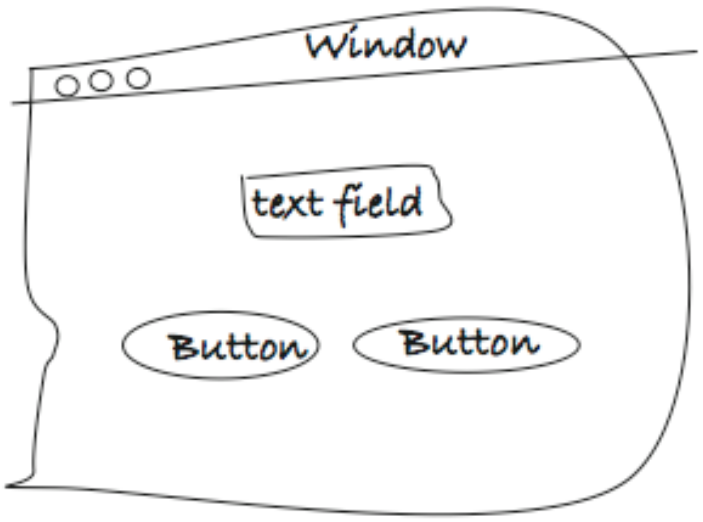
クラスがウィンドウオブジェクトを作るとき、ウィンドウの位置やその他の情報を収納するためのメモリ（RAM）領域を確保します。しかしウィンドウを閉じるためのコードのコピーまでは作りません。コードはどのウィンドウにも共通のものであり、いちいちコピーするのはメモリの無駄です。ウィンドウを閉じるために必要なコードは一つだけであり、各ウィンドウオブジェクトはクラスが一つだけ持っているそのコードへのアクセス方法さえ知っていれば問題ありません。

このメモリの確保や解放に関するコードを含むサンプルを以前の章の中で見かけましたが、少し上級レベルのこのテーマについてはずっとあとで学ぶことにします。

練習

■アプリケーション

ここでは二つのボタンと一つのテキストフィールドを持つアプリケーションを作ってみます。ボタンを押すと値がテキストフィールドに表示されます。もう一つのボタンを押すと、別の値が表示されます。ボタンが二つある計算機（実際には計算しませんが）だと考えてください。もちろん、いろいろ勉強すれば本当の計算機を作ることもできますが、まずは一歩一歩進んでいきましょう。



これから作るアプリケーションのイメージ

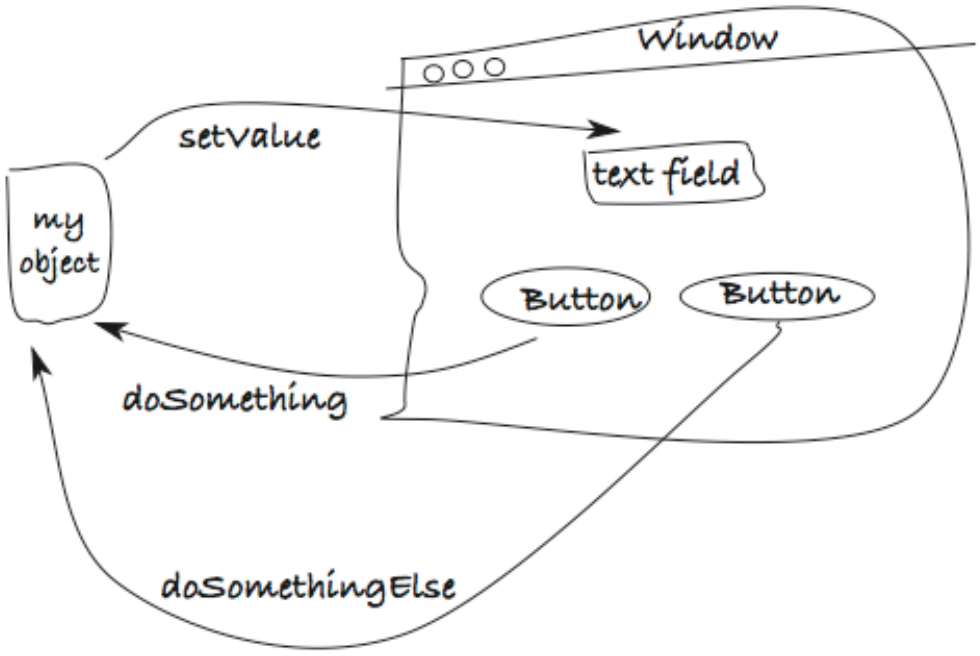
アプリケーションのボタンが押されると、ボタンはメッセージを送ります。このメッセージは実行されるべきメソッドの名前を持っています。このメッセージは・・・何に対して送信されるのでしょう？ウィンドウの場合、ウィンドウクラスのインスタンスであるウィンドウオブジェクトに対して「閉じる」メッセージは送信されます。では今我々が必要なものは？それは二つのボタンから送られたメッセージを受け取り、テキストフィールドに値を表示するように指示できるオブジェクトです。

■初めてのクラス

ではまず我々の初めてのクラスを作り、次にそのインスタンスを作ってみましょう。そのオブジェクトは二つのボタンからのメッセージを受け取ることになります。ウィンドウオブジェクトと同じように我々が作るインスタンスもオブジェクトですが、ウィンドウオブジェクトとは違い、プログラムを実行しても我々のインスタンスは画面上に表示されません。それはMacのメモリ上に存在するだけの目に見えないものです。

二つのボタンのどちらかから送信されたメッセージを受け取ると、適切なメソッドが実行されます。メソッドの中身のコードはクラスの中（インスタンスそのものではなく）に保存されています。メソッドが実行されるとテキストフィールドのテキストが置き換えられます。

ではそのメソッドはテキストフィールドのテキストを変える方法をどうして知っているのでしょうか。実際のところ、その方法は知りません。しかしテキストフィールド自身が、自分のテキストを変える方法を知っています。だから我々はテキストフィールドオブジェクトにメッセージを送り、テキストを変えるように頼むのです。ではどのようなメッセージを送れば良いのでしょうか。もちろん、まずメッセージを受け取る相手の名前（つまりウィンドウの中のテキストフィールド）を指定してやる必要があります。そしてもちろん、受け取る相手に何をして欲しいのかを告げる必要もあります（当然テキストフィールドがどんなメソッドを実行でき、それがどんな名前なのかを調べなくてはいけません）。さらにはテキストフィールドに何を表示させたいのか（内容は押されたボタンによって変わります）も指示する必要があります。つまりメッセージが送るべき内容はオブジェクトの内容やメソッドの名前だけでなく、テキストフィールドのメソッドが必要としている引き数も含んでいます。



アプリケーション内部のメッセージやり取りのイメージ

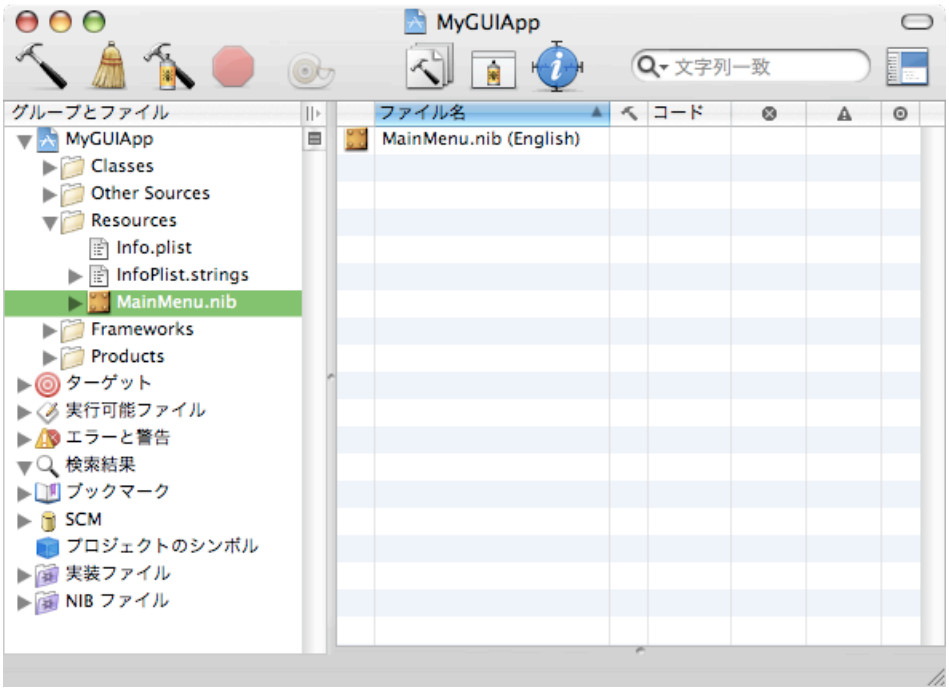
これはObjective-Cの一般的なメッセージのフォーマットです。[1.1]は引き数無し、 [1.2]は引き数ありの例です。

```
[1]
[receiver message];
[receiver messageWithArgument:theArgument];
```

これらの命令を見てわかる通り、文は全てブラケット（大括弧）[]の間に置かれ、行の最後には例のごとくセミコロンがついています。ブラケットの間には、まずメッセージを受け取る相手（一般的に「レシーバ」と呼ばれます）の名前が最初に書かれ、次にメソッドの名前が置かれます。そしてメソッドに一つ以上の引き数がある場合は[1.2]のようにメソッドの名前のあとに書きます。

■プロジェクトを作る

ではこれが実際どのように動作するのか見てみましょう。Xcodeを起動し新しいプロジェクトを作ります。Applicationという項目の下にある「Cocoa Application」をひな形に選んでください。そしてプロジェクトの名前をつけます（慣習的に、GUIアプリケーションの名前は大文字から始めます）。Xcodeのウィンドウが表示されたら、左の「グループとファイル」の中から「Resources」フォルダを開き、「MainMenu.nib」ファイルをダブルクリックします。

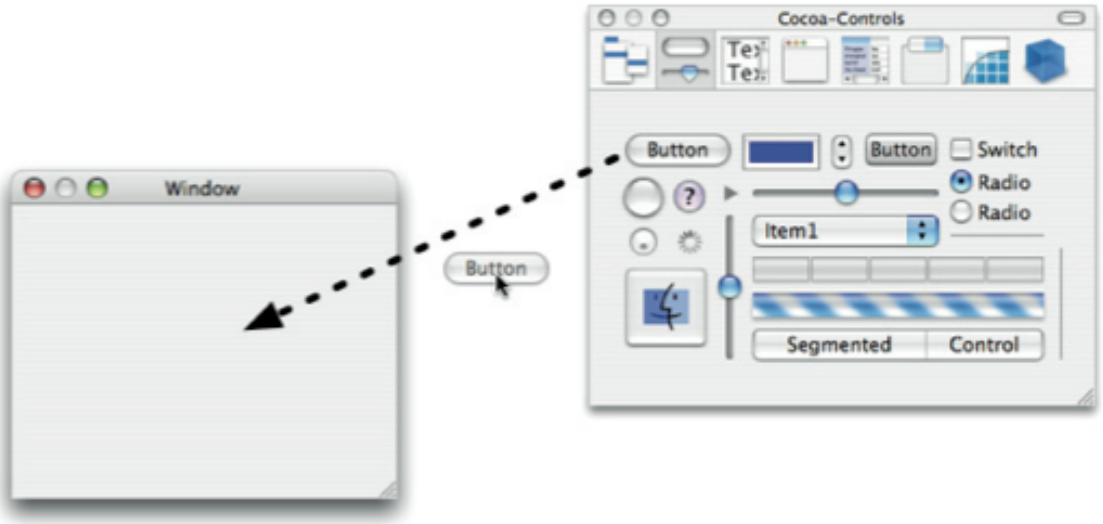


XcodeでMainMenu.nibファイルをブルクリックする

■GUIを作る

XcodeとはのもうーつのCocoaアプリ開発用アプリケーションInterface Builderが起動します。これは主にアプリケーションのGUIをデザインするソフトです。たくさんのウィンドウが表示されるので、メニューから「Hide Others」を選択した方が良いかもしれません。Interface Builderは三つの画面を表示します。「Window」という名前のウィンドウは、あなたのアプリケーションのユーザーが目にす

るウィンドウです。少し大きすぎるので、若干小さくした方が良いでしょう。「Window」ウィンドウの右隣には、「Cocoa-」から始まる名前のウィンドウがあります。このウィンドウはGUIで使うことの出来るいろいろなオブジェクトの倉庫のようなもので、「パレットウィンドウ」と呼ばれます。このウィンドウのツールバーの左から二番目のアイコンをクリックし、ボタンを「Window」ウィンドウの上にドラッグして配置します。もう一つボタンをドラッグして、ウィンドウ上に二つのボタンを配置してください。そしてツールバー左から三番目のボタンをクリックし、「System Font Text」というテキストフィールドを同じようにウィンドウ上にドラッグし、配置します。



GUIオブジェクトをパレットからあなたのアプリケーションのウィンドウにドラッグする

パレットウィンドウからあなたのアプリケーションのウィンドウにボタンをドラッグする作業の裏で、Interface Builderは新しいボタンオブジェクトを生成しあなたのウィンドウに配置しています。テキストフィールドや、パレットウィンドウからドラッグしてウィンドウに配置するその他のオブジェクトについても、同じことをしています。

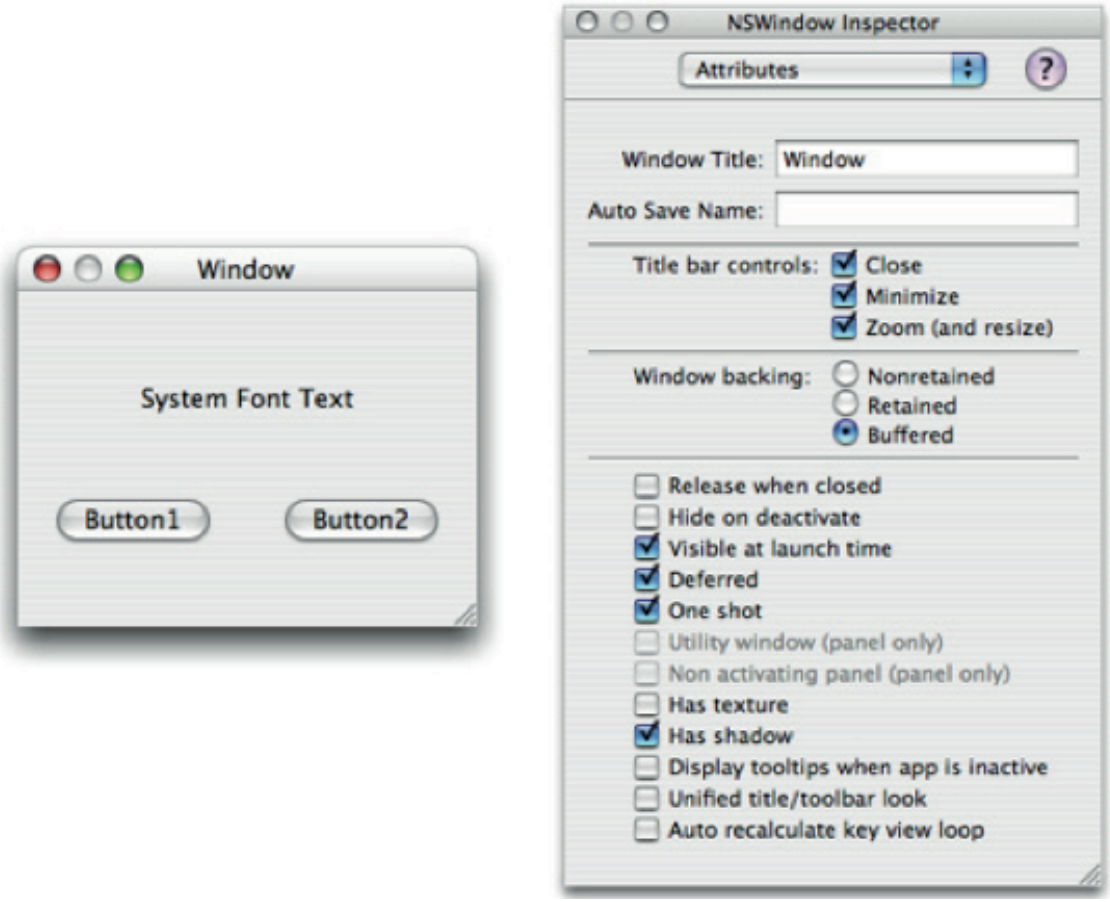
カーソルをパレットウィンドウのアイコンの上にしばらくかざしておくと、NSButtonやNSTextViewといった名前が表示されます。これらはアップルによって提供されているクラスの名前です。次の章では、我々のプログラムで処理を実行するための、これらのクラスのメソッドをどうやって見つければ良いのかを学びます。

Windowウィンドウにドラッグしたオブジェクト（ボタンとテキストフィールド）を見栄えよくデザインしましょう。それらの大きさをうまいこと変えてウィンドウにフィットさせます。ボタンをダブルクリックして、ボタンの名前を変えます。このプロジェクトが終わったら、その他のオブジェクトをどうやってウィンドウに追加すれば良いか学ぶために、パレットウィンドウをいろいろ調べてみることにします。

■Interface Builderの機能をいろいろ試してみる

オブジェクトの状態を変更するためには、そのオブジェクトをクリックで選択してコマンド＋シフト＋iキーを押します。例えばWindowウィンドウを選択して（左下のウィンドウのInstancesタブでウィンドウが選択されていることがわかります）コマンド＋シフト＋iキーを押してください。右上に表示されたウィンドウ一番上のポップアップボタンで「Attributes」を選択して、下の「Has texture」ボタンを

クリックしてください。こうするとWindowウィンドウがメタル調に変わります。つまり一行もコードを書くこと無く、アプリケーションの見た目を簡単に変更することができるのです！



作成したウィンドウとInspectorウィンドウ

■クラスのバックグラウンド

　では約束した通り、そろそろクラスを作ってみましょう。ただしその前に、クラスというものがどう機能するかをもう少し詳しく見てみます。何もかもを0から作るのではなく、既に作ったものを使いまわすことで、プログラミングの手間を大きく省くことができます。例えばもし特殊な機能を持つウィンドウを作りたい場合、あなたはただこの特別な機能のコードをウィンドウに追加してやるだけで良いのです。ウィンドウを最小化したり閉じたりといったその他の機能についてのコードを書く必要はありません。あなたはほかのプログラマたちが作ったこれらの機能を自由に受け継いで（専門的には「継承する」といいます）、それに自分のコードを付け足すことができるのです。そしてこの機能こそが普通のCとObjective-Cをわける決定的な違いです。

　これは一体どういうことでしょうか。たとえばNSWindowというウィンドウのクラスがありますが、あなたはこのクラスの機能を 継承した（受け継いだ）新しいクラスを自分で作ることができます。あなたは自作のウィンドウクラスにいくつか新しい機能を追加したとしましょう。ではあなたのウィンドウが「閉じる」メッセージを受け取ったらどうなるでしょうか？あなたはまだ「閉じる」に関する何のコードも書いていないし、そのような機能のコードをコピーしてもいません。もしあなたの特別なウィンドウが「閉じる」のような特定のメソッドに対応するコードを持っていない場合、そのメッセージはウィンドウが機能を受け継いだ元のクラス（「親クラス」「スーパークラス」といいます）自動的に転送されます。そしてもしスーパークラスもコードを持っていなかったら、コードが見つかるまでスーパークラスのスーパークラスへと転送されます。

対応するコードが結局見つからなかった場合、あなたが送信したメッセージは処理されません。それは自動車修理工場にそりのタイヤ交換を頼むようなものです。工場のボスだってそんなことはできません。そういう場合はObjective-Cはエラーを報告します。

■カスタムクラス

ではスーパークラスから既に受け継いだメソッドに、あなたの独自の処理を加えたい時はどうでしょうか？簡単です。そのメソッドを上書き（オーバーライド）すれば良いのです。例えば閉じるボタンがクリックされた時、ウィンドウが実際に閉じられる前に画面上から見えなくするようなコードを書くことができます。その場合、アップルによって定義されたウィンドウを閉じるためのメソッド名をそのまま使います。そうすれば閉じるメッセージを受け取った時、実行されるメソッドはアップルのものではなく、あなたが書いたものになります。そして、ウィンドウは実際に閉じられる前に一度画面上から消えてくれます。

そうそう、ウィンドウを実際に閉じる処理はアップルによって既に提供されていますよね。だからあなたが自作する「閉じる」メソッドでは、スーパークラスが既に持っているコードを使えば、ウィンドウを閉じることができます。ただし「閉じる」メソッドを実行したら、メソッドの中で「閉じる」メソッドを再び呼び出し（「再帰」といいます）、永久にそれが続く危険性があるので（テレビの中にテレビを写すようなイメージです）、それを避けるためにちょっと違ったコードの書き方をする必要があります。

[2]
```
//ウィンドウを画面上から見えなくするコードをここに書く
[super close];// スーパークラスのメソッドを呼び出す
```

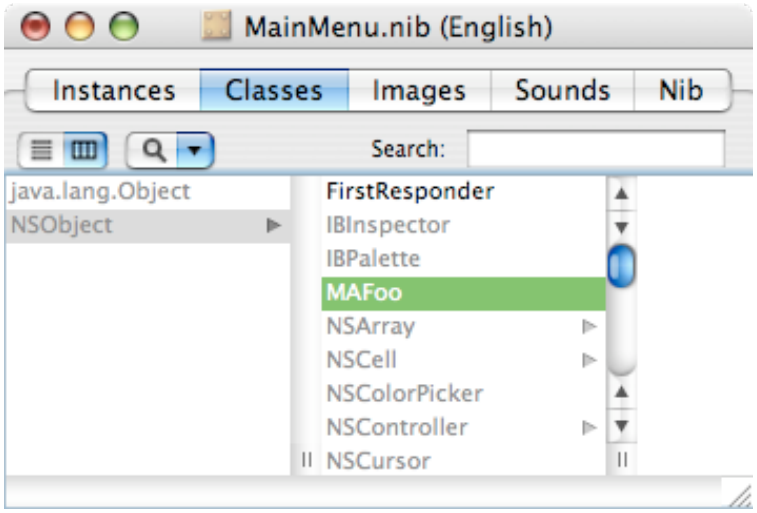
これはこの入門ガイドにはちょっと難しすぎるので、現時点では理解しなくて結構です。

■全てを統治するクラス

丘の上にそびえる城の中の王のごときクラスの中の王、それがNSObjectと呼ばれるクラスです。あなたが作ったり使ったりするであろうほぼ全てのクラスは、直接的・間接的にNSObjectのサブクラス（NSObjectを継承する子供のクラス）になります。例えばNSWindowクラスはNSResponderクラスのサブクラスであり、NSResponderはNSObjectのサブクラスです。そしてNSObjectクラスは全てのオブジェクトに共通のメソッド（例えばオブジェクトの状態を表示したり、特定のメソッドに対応できるかどうかを調べたり）を定義しています。では能書きはこのくらいにして、クラスを作ってみましょう。

■クラスを作る

まずMainMenu.nibを開いて、左下のウィンドウでClassesタブを選択してください。ブラウザーの一番左、最初のコラムにNSObjectというクラス名があるのがわかります。NSObjectをクリックして選択し、メニューバーで「Classes」メニューを開き、「Subclass NSObject」を選びます。左下の「MainMenu.nib」ウィンドウに戻り、新しく出来たクラスにかっこいい名前をつけます（私はMAFooと名付けました）。



MAFooクラスを作る

MAFooの最初の二文字はMy Applicationの略です（ちなみにプログラミングの世界では何か適当な名前をつける際、fooとかbarあるいはhogeなどという言葉がよく使われます。これらに特別な意味はありません）。クラス名は好きなように付けることができます。アプリケーションを書き始めたら、自分なりのルールを作ることをお勧めします（例えば「MAWindow」のように二、三文字をクラス名の頭に付けて、他のクラスとの重複を避ける、など）。ただし混乱を避けるためにNSは使わないでください。NSはアップルのクラス名に使われています。NSはNextStepの略であり、それはアップルがNext社を買収した時手に入れたMac OS Xの基礎となっているOSの名前です（ちなみに買収のボーナスとしてSteve Jobsもアップルに取り戻しました）。CocoaDevのwikiには避けるべき接頭辞のリストがあります。自分の接頭辞を選ぶ際、チェックした方が良いでしょう。

http://www.cocoadev.com/index.pl?ChooseYourOwnPrefix

クラスを作るときは、そのクラスの内容がわかるような名前をつけるべきでしょう。例えばウィンドウを表示・操作するためのクラスであればNSWindowという名前がついています。あるいは色を表現するクラスであればNSColorという名前です。このサンプルの場合、MAFooクラスはアプリケーション内で複数のオブジェクトがどのようにお互いにやり取りしているかを学ぶためのクラスですので、特に意味のない一般的な名前を付けました。

■インスタンスを作る

ではInterface Builderに戻り、「Classes」メニューから「Instantiate MAFoo」を選択してください（「MainMenu.nib」ウィンドウで「MAFoo」が選択されていることを確認してください）。すると「Instances」タブに「MAFoo」という新しいアイコンが出来ます。このアイコンは今作ったばかりの新しいインスタンスを表しています。

MAFooのインスタンスを作る

■コネクションを張る

次の作業は（メッセージを送信する）ボタンとMAFooオブジェクトとの間にコネクションを作ることです。ついでにテキストフィールドにメッセージを送るために、MAFooからテキストフィールドへのリンクも張ります。別のオブジェクトの参照（相手のオブジェクトの住所のようなもの）を持っていないと、別のオブジェクトに対してメッセージを送ることはできません。ボタンからMAFooへのコネクションを作ることで、ボタンにMAFooオブジェクトへの参照を提供することができ、ボタンはこの参照を使ってMAFooオブジェクトにメッセージを送信することができます。同様にMAFooからテキストフィールドへコネクションを張ることで、前者から後者へメッセージを送ることができるようになります。アプリケーションが何をすべきかもう一度確認しましょう。

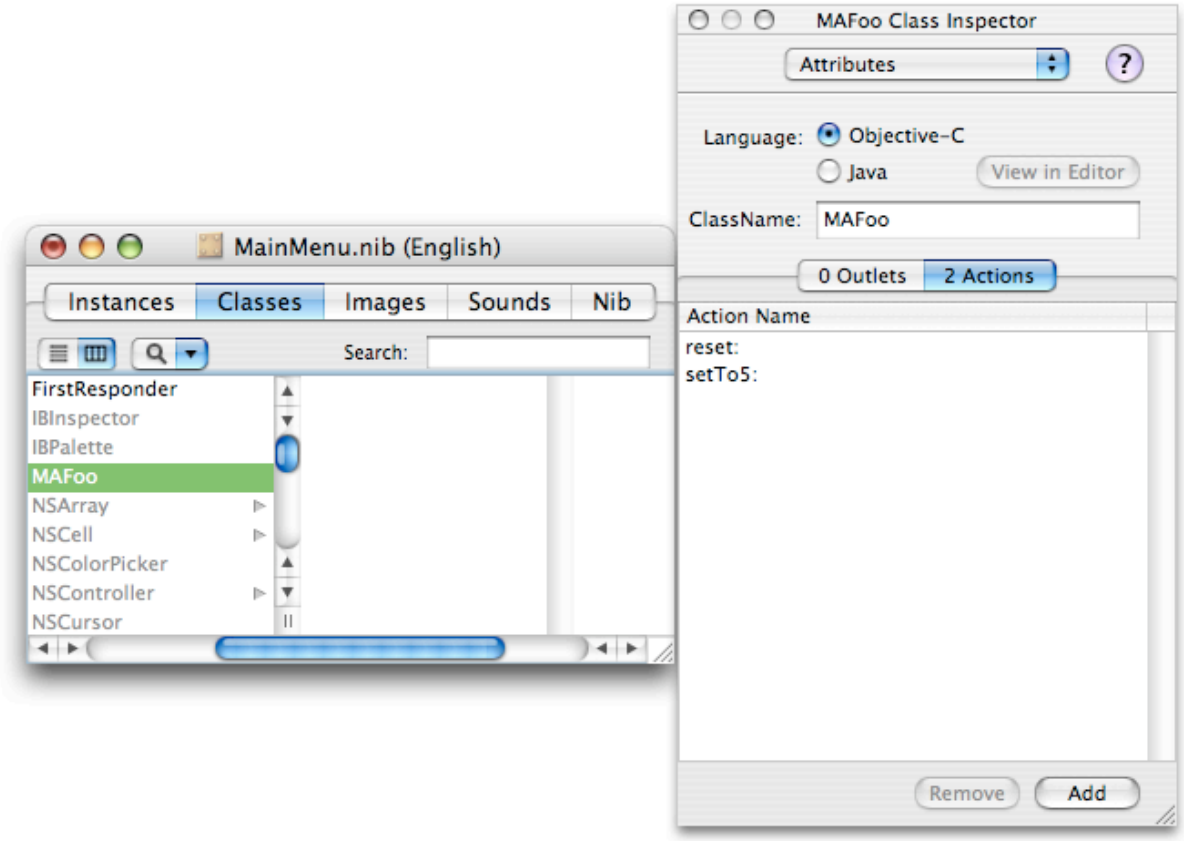
各ボタンはクリックされると、アクションに応じて適切なメッセージを送信します。このメッセージはMAFooが実行すべきメソッドの名前を含んでいます。そしてさっき作ったばかりのMAFooクラスのインスタンス、つまりMAFooオブジェクトに対して送信されます（オブジェクトのインスタンスそのものは実行すべきコードそのものは持っていないということを思い出してください。コードはクラスが持っています）。そしてMAFooオブジェクトに送られたメッセージが引き金となって、MAFooクラスが何かの処理（この場合ではテキストフィールドオブジェクトに対するメッセージの送信）を行います。そして全てのメッセージと同様、このテキストフィールドに対するメッセージも、テキストフィールドが実行すべきメソッドの名前を持っています。この場合テキストフィールドオブジェクトが実行すべきメソッドは値を画面に表示することであり、その値（引き数という名前でしたよね。覚えてますか？）はメッセージの一部として送信されます。

我々のクラスは、二つのボタンオブジェクトから呼ばれる二つのアクション（メソッド）が必要です。

我々のクラスは、一つのアウトレット、つまりメッセージを送信するべきオブジェクトを覚えておく変数が必要です。

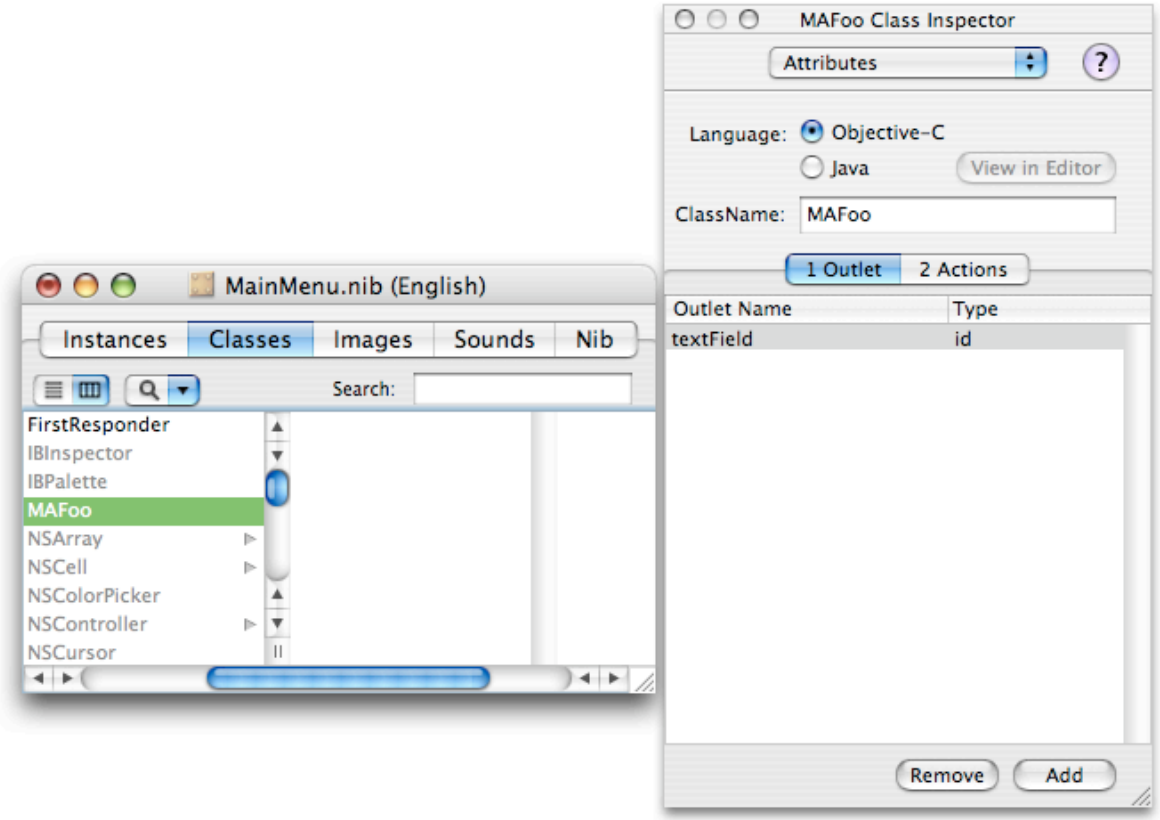
次にMainFile.nibウィンドウのClassesタブでMAFooが選択されていることを確認してから、コマンド＋シフト＋iキーを押してInspectorウィンドウを表示させます。InspectorウィンドウでActionタブを選択し、addボタンをクリックして、MAFooクラスにアクション（つまりメソッド）を追加してください。Interface Builderが自動で付けるデフォルトの名前（myAction:）を、もっとわかりやすい名前（このメソッドではテキストフィールドに「5」を表示させるので、例えば「setTo5:」など）に変更し

ます。もう一つメソッドを追加して、同じように名前（こちらはテキストフィールドに0をセットするので例えば「reset:」など）を付けます。これらのメソッドはどちらもコロン（:）で終わることに注意してください（ただし:を入力しなくてもInterface Builderが自動的に追加します）。これについては後ほど触れます。



MAFooクラスにアクションメソッドを追加する

インスペクタウィンドウで「Outlet」タブを選択し、アウトレット（他のオブジェクトへの参照）を作成して名前をつけます（例えばtextFieldなど）



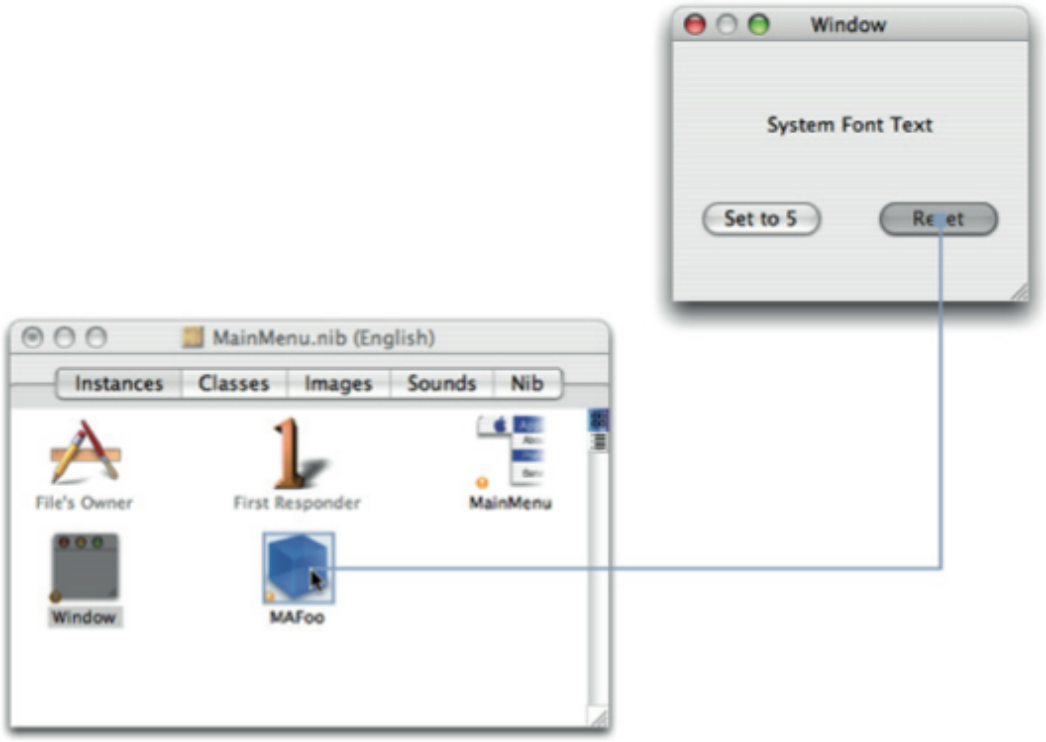
MAFooクラスにアウトレットを追加する

オブジェクト間のコネクションを作成する前に、二つのボタンにわかりやすい名前をつけます。最初のボタンはMAFooインスタンスに「5」を表示するようにメッセージを送信するので、ボタンの名前を「Set to 5」（「5をセット」など日本語でも可）に変えます（ボタンの名前の変え方についてはもう勉強しましたね。ボタンをダブルクリックして、名前を編集するのです）。同様に二つ目のボタンの名前を「Reset」に変えます。ただしボタンの名前を変えても変えなくてもプログラムは正しく動く、ということは覚えておいてください。これは単に、ユーザーがよりわかりやすいインターフェースにするのが目的です。

これで各コネクションを作成する準備はできました。必要なコネクションは以下の三つです。

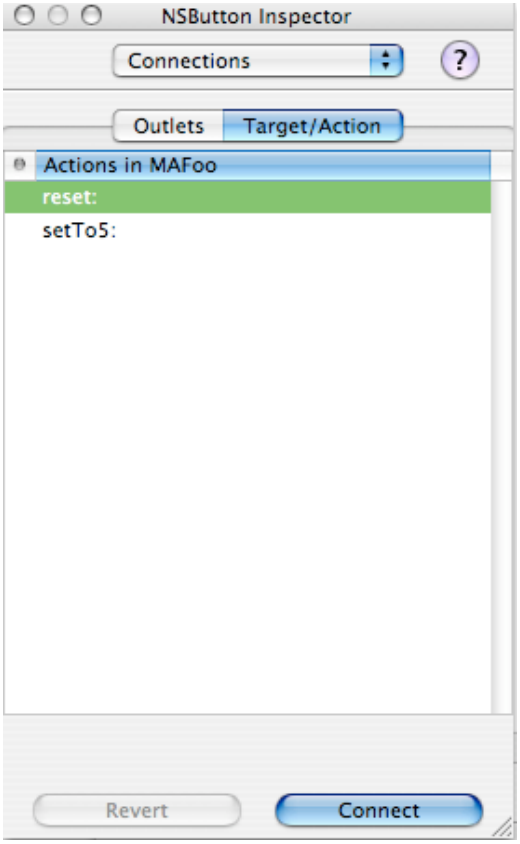
a)「Reset」ボタンからMAFooインスタンスへ
b)「Set to 5」ボタンからMAFooインスタンスへ
c)MAFooインスタンスからテキストフィールドへ

コネクションを作成するためにMainFile.nibウィンドウのInstancesタブをクリックします。そしてcontrolキーを押したまま、「Reset」ボタンからMAFooインスタンスにマウスでドラッグします（他の方法でやらないでください！）。ドラッグするとコネクションを表す線が画面上に表示されます。ボタンからMAFooインスタンスに線を引いて、つながったらマウスボタンを離します。



ボタンからMAFooインスタンスへのコネクションを作成する

マウスボタンを話すと、inspectorウィンドウはMAFooオブジェクトで利用可能なアクションメソッドの一覧をコネクションパネルに表示します。適切なアクション（つまりこの場合は「reset:」）を選択し、「Connect」ボタンをクリックして、コネクションの作成を終わらせます。



inspectorウィンドウでコネクションを確立する

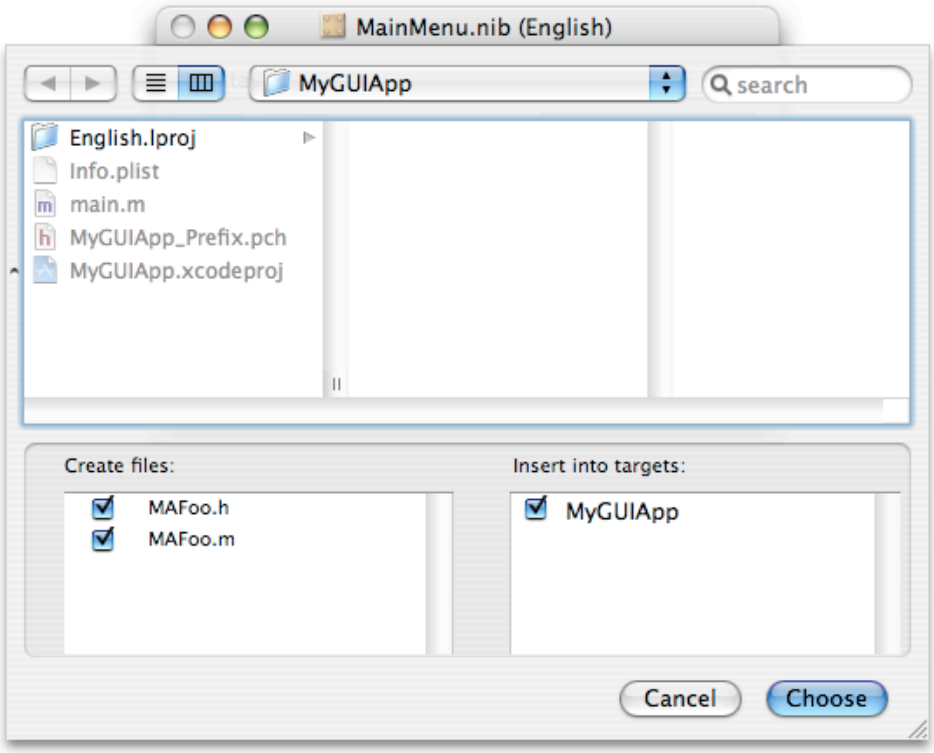
これでボタンはMAFooオブジェクトの参照を持つことができ、ボタンがクリックされるとMAFooインスタンスに対して「reset:」メッセージを送信します。同じように「Set to 5」ボタンからMAFooオブジェクトにコネクションを作成してください。

MAFooオブジェクトからテキストフィールドへのコネクションを作成するには、MAFooからテキストフィールドに向けてコントロールキー＋ドラッグを行い、「Connect」ボタンをクリックします。

この作業は一体どういうことでしょうか？すぐにわかりますが、この作業によって、あなたは一行も書くこと無く、たくさんのコードを生成することができるのです。

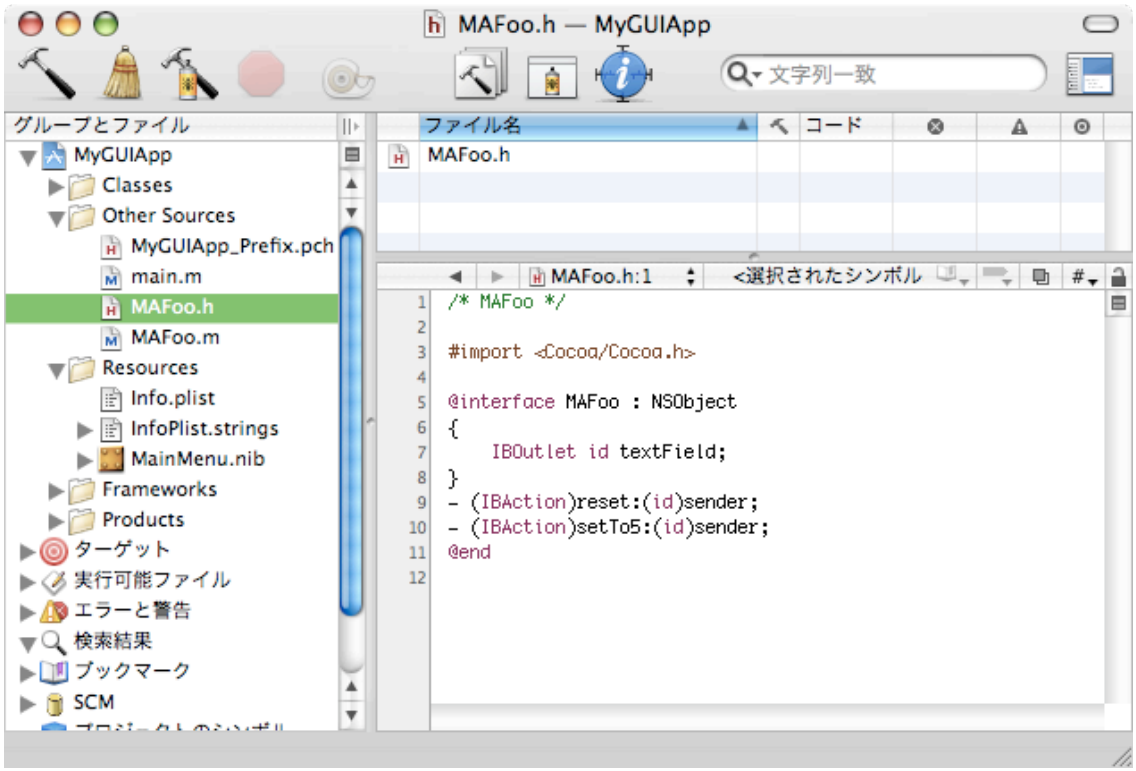
■コードを生成する

MainMenu.nibウィンドウでMAFooインスタンスを選択してから、「Classes」タブに切り替えます。たくさんあるクラスの一覧の中でMAFooクラスが選択されていることを確認してください。「Classes」メニューから「Create Files for MAFoo」を選択します。Interface Builderは生成したファイルをどこに保存するか聞いてきます。デフォルトでは作成中のプロジェクトフォルダの中が選択されますので、必要に応じて適当な所を選択してください。



MAFooクラスのコードを生成する

ここでXcodeに切り替えると、プロジェクトウィンドウの「Other Sources」グループの中に、今生成されたファイルを見つけることができるはずです（グループの中からファイルを移動しても、実際のハードディスク上のファイルの位置は変わりません。グループ分けはXcode上の便宜上のものです）。



生成されたファイルがXcode上で表示されます

ここで4章で学んだ関数について少し思い出してみましょう。4章[11.1]の関数プロトタイプ宣言について覚えていますか？これは、どのような関数がその後使われるかについてコンパイラに知らせる警告のようなものでしたね。生成された二つのファイルのうちMAFoo.hという名前のファイルは、MAFooクラスについての情報をまとめた「ヘッダファイル」で、プロトタイプ宣言同様いろいろな情報をコンパイラに伝えるためのものです。例えば[3.5]行に「NSObject」という単語があるのに気がつくと思いますが、これはMAFooクラスはNSObjectを継承したクラスである、ということを示しています。

```
[3]
/* MAFoo */

#import <Cocoa/Cocoa.h>  // [3.3]

@interface MAFoo : NSObject
{
    IBOutlet id textField; // [3.7]
}
- (IBAction)reset:(id)sender;
- (IBAction)setTo5:(id)sender;
@end
```

また[3.7]にはテキストフィールドオブジェクトへのアウトレットがあることに気がつくと思います。この「id」はこの変数（アウトレット）がオブジェクトであるということを示しています。「IB」はInterface Builderの略です。

[3.9, 3.10]の「IBAction」は「void」に相当します。このメソッドは、メッセージを送信したオブジェクトに何も返り値を返さない、つまりボタンはMAFooにメッセージを送っても、何の返事も受け取ることはできない、ということを意味しています。

またこのファイルの中に二つのInterface Builderアクションがあることがわかると思います。この二つはMAFooクラスのメソッドです。メソッドは既に勉強した関数ととてもよく似ていますが、少し違います。この違いについてはあとで学びます。

[3.3]のような書き方の代わりに、このガイドの前の方で#import <Foundation/Foundation.h>というのを学びましたが、これは非GUIアプリ向けのもので、[3.3]はGUIアプリ向けの書き方です。

ではさらに二つ目のファイル「MAFoo.m」を眺めてみましょう。たくさんのコードが自動的に生成されています。

```
[4]
#import “MAFoo.h”

@implementation MAFoo

- (IBAction)reset:(id)sender    // [4.5]
{
}
- (IBAction)setTo5:(id)sender
{
}
@end
```

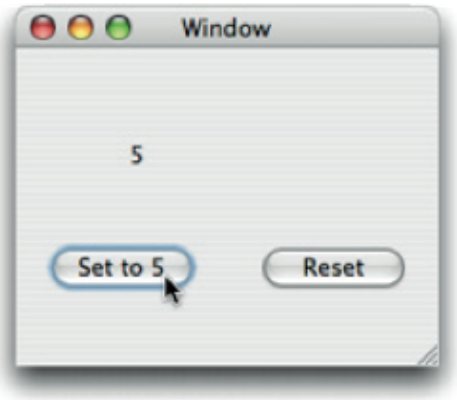
まず最初にヘッダファイルMAFoo.hがインポートされます[4.1]。これによりコンパイラはクラスについて必要な情報を知ることができます。その後に「reset:」と「setTo5:」の二つのメソッドがあります。これらはこのMAFooクラスのメソッドです。関数と同じように、コードをこの中括弧の中に書いていきます。このアプリケーションでは、ボタンが押されるとボタンはMAFooクラスのインスタンスにメッセージを送り、これらのメソッドのどちらかを実行するように要求します。そのメッセージ送受信の処理に関してはコードを書く必要はありません。Interface BuilderでボタンとMAFooオブジェクトとのコネクションを作ればそれで終わりです。しかしこれら二つのメソッドを実装する、つまり各機能を実行するためのコードを書く必要はあります。この場合これらのメソッドがすべきことは他でもない、このMAFooオブジェクトからテキストフィールドオブジェクトにメッセージを送ることです。以下のようにコードを書きます[5.7, 5.12]。

```
[5]
#import “MAFoo.h”

@implementation MAFoo

- (IBAction)reset:(id)sender
{
    [textField setIntValue:0];   // [5.7]
}

- (IBAction)setTo5:(id)sender
{
    [textField setIntValue:5];   // [5.12]
}
@end
```

見ての通り、textFieldアウトレットとして参照を持っているオブジェクトに対してメッセージを送っています。Interface Builderを使って、このアウトレットからテキストフィールドにコネクションを作ったため、「setIntValue:」というメソッドは画面上のテキストフィールドオブジェクトに正しく送られます。「setIntValue:」メソッドは引き数にint（整数）の値をとります。「setIntValue:」メソッドはテキストフィールドオブジェクトに整数の値を表示することができます。次の章では、どのようにしてこのような適切なメソッドを探せば良いかについて学びます。

■準備完了

　これであなたのアプリケーションをコンパイルして実行する準備は終わりました。いつものようにXcodeのツールバーで「ビルドと実行」ボタンを押します。アプリケーションをビルドして起動するには少々の時間が必要です。そのうちアプリケーションが画面上に表示されますので、動作をいろいろ試すことができます。



動作中のアプリケーション

　このように、自分ではたった二行のコードを書くだけで、（とても単純な）アプリケーションを作ることができました。

09: メソッドを調べる

■はじめに

前の章ではメソッドについて学びました。自分で二つのメソッド（の中身）を書きましたが、その中でアップルによって提供されているメソッドを使いました。「setIntValue:」は整数の値をテキストフィールドオブジェクトに表示するメソッドでしたね。ではこのようなメソッドをどうやって探したら良いのでしょうか？

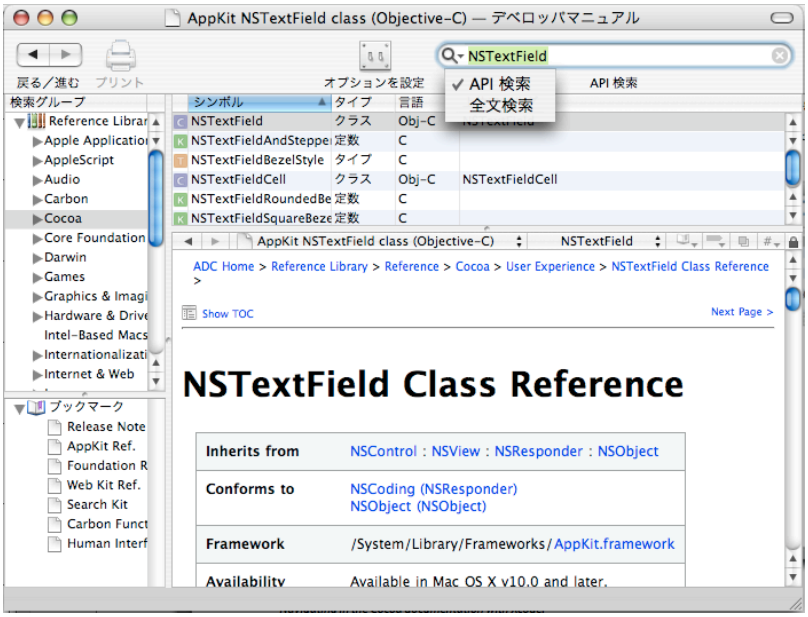
> **アップルによって作られたメソッドを使う場合は、自分でコードを書く必要は無い、ということをもう一度思い出してください。しかもそれらのメソッドはほとんどバグはありません。なので、自分で何かのコードを書き始める前に、その機能にふさわしい機能が既に存在していないかしっかり時間をかけて調べるべきです。**

■練習

Interface Builderでパレットウィンドウのオブジェクトの上にマウスカーソルをあわせておくと、小さなラベルが表示されます。例えばボタンアイコンの上にカーソルをあわせれば、「NSButton」というラベルが表示されるでしょう。System Font Textと書かれたテキストフィールドであれば「NSTextField」です。これらはどれもクラス名です。ではNSTextFieldクラスにはどんなメソッドがあるのか調べてみましょう。

Xcodeを起動し、「ヘルプ」メニューで「マニュアル」を選択します。左側のフレームで「Cocoa」を選択し、右上の検索フィールドに「NSTextField」と入力してください（スクリーンショットのように「API検索」が選択されていることを確認してください）。アルファベットをタイプ中に検索結果がどんどん絞り込まれ、すぐに「NSTextField」が一番上に表示されます。

NSTextField（タイプは「クラス」）の行を選択すると、下のフレームにNSTextFieldの情報が表示されます（現状では全て英語なのが日本人にはつらいところです）。



XcodeでCocoaドキュメントを調べる

ここでまず確認すべきことは、このクラスがどのクラスから機能を継承しているか（「inherits from」の欄）、という点です。この継承関係の先祖は、丘の上の城の主、NSObjectです。少し下には

Methods by Task

という見出しがあります。ここからメソッドの情報を調べ始めます。このリストをざっと眺めると、我々が今探している機能、テキストフィールドに値を表示するメソッドはここにないことがわかります。では次にすべきことは、継承の原則によりNSTextFieldクラスのすぐ上のスーパークラス、NSControlの中を探すことです（ここでも見つからなければ、さらにその上のクラスNSViewへとさかのぼって調べていかなくてはなりません）。全てのドキュメントはHTMLで書かれているため、（「inherits from」の欄の）「NSControl」のリンクをクリックすれば、NSControlに移動することができます。ではNSControlクラスの情報を見てみましょう。

NSControl Class Reference

Inherits from NSView : NSResponder : NSObject

これでクラスの階層を一段階上に登ったことがわかります。このメソッドのリストの中に

Setting the control’s value

という項目が見つかるはずです。これが、我々の探していたものです。この副見出しの下に

– setIntValue:

という項目があります。どうもこれが探していたもののようなので、リンクをクリックしてメソッドの説明を読んでみましょう。

setIntValue:
Sets the value of the receiver’s cell using an integer.
- (void)setIntValue:(int)anInt

Parameters
anInt
The value of the cell interpreted as an integer.

Discussion
If the cell is being edited, this method aborts all editing before setting the value. If the cell does not inherit from NSActionCell, the method marks the cell’s interior as needing to be redisplayed;NSActionCell performs its own updating of cells.

Availability
Available in Mac OS X v10.0 and later.

See Also

– intValue
– setDoubleValue:
– setFloatValue:
– setObjectValue:
– setStringValue:

先ほど作成したアプリでは、NSTextFieldオブジェクトはメッセージのレシーバ（受取手）であり、引き数に整数値を渡してやる必要があります。このことはメソッドのシグネチャ（メソッドの返り値の型、名前、引き数の型、全てを表示したもの）からも読み取ることができます。

- (void)setIntValue:(int)anInt

Objective-Cでは、メソッドの先頭のマイナス（-）記号は、そのメソッドがインスタンスメソッド（クラスメソッドの反対。これについてはあとで説明します）であることを示しています。「void」は、返り値が何も無いことを意味しています。つまりtextFieldに対してsetIntValue:を送っても、MAFooオブジェクトはテキストフィールドオブジェクトから何の値も受け取れない、ということです。コロン:の次の(int)は、引き数として渡す値は整数値でなければならない、ということを表しています。我々のサンプルアプリでは我々が送信する値は整数の5か0ですので、問題ありません。

どのメソッドを使うのが適当か判断するのが難しいことがときどきあります。このドキュメントをよく読むことで判断に迷うことが減りますので、よく読んでおいてください。

ではテキストフィールドに表示されている値を取り出したい時はどうでしょうか？オブジェクトが持つ全ての値は、オブジェクトの中に保存され、隠されている（隠蔽されている）、ということを思い出してください。しかし大概の場合、オブジェクトはアクセッサと呼ばれる、値の取り出しと値の設定の、対になっているメソッドを用意しています。取り出しメソッドはこんなものです。

[1]
- (int) intValue

見ての通り、このメソッドは整数値を返します。なのでテキストフィールドオブジェクトが表示している整数値を取り出したい場合はこのようにメッセージを送ります。

[2]
resultReceived = [textField intValue];

繰り返しますが、関数（及びメソッド）では全ての変数は隠蔽されています。これはとても便利な特徴です。なぜならある部分で変数を変更しても、別の関数内の同じ名前の変数に影響が出る心配は無いからです。ただし関数の名前は重複してはいけません。Objective-Cはもう少し進んでおり、メソッド名はクラス内では重複できませんが、別のクラスは同じ名前のメソッドを持つことができます。メソッド名の衝突を心配すること無く、それぞれに相互に独立したクラスを作成することができるため、これは巨大なプログラムにはとても有益な機能です。

しかしメリットはそれだけではありません。異なったクラスの異なったメソッドが同じ名前を持つことを、専門的には「ポリモーフィズム（多態性）」といいますが、これこそオブジェクト指

向プログラムの最大の特徴の一つです。これにより、プログラマは自分が操作しているオブジェクトが一体どのクラスのものなのかを知らなくても、コードを書くことができます。実行時にメッセージを受け取ったオブジェクトが、そのメッセージを理解できさえすれば問題無く動作するのです。

　ポリモーフィズムの先進性を利用することで、クラスのデザイン次第ではより柔軟で拡張性の高いアプリケーションを作ることができます。たとえば先ほど作成したGUIアプリでは、もしテキストフィールドを、setIntValue:を実行できる別のクラスのオブジェクトと交換しても、コードを書き換えたり、コンパイルしなおさなくても問題無く実行することができます。実行中にオブジェクトを差し替えることだって可能です。これはオブジェクト指向プログラムの先進性のおかげです。

10: awakeFromNib

■はじめに

プログラムの作成をより簡単にするために、アップルはたくさんの作業をしてくれています。あなたの小さなアプリではウィンドウやボタンを描画するなど、細かい作業をする必要はありません。

これらの機能のほとんどは二つのフレームワーク（いろいろなクラスや関数を集めたもの）を通じて提供されています。4章[12]のサンプルでインポートしたFoundation Kitフレームワークは、GUIを伴わない多くの機能を提供しています。もう一つのApplication Kitと呼ばれるフレームワークは、画面に描画されたり、ユーザーの操作を受け付けたりする様々なクラスを提供しています。どちらのフレームワークもきちんとしたドキュメントが提供されています（ただし英語のみ）。

ではGUIアプリに戻りましょう。アプリケーションが起動してウィンドウが表示されたとき、すぐにテキストフィールドに何か値を表示させるように変更します。

■練習

ウィンドウに関する全ての情報はnibファイル（「nib」は「Next Interface Builder」の略）に収められています。つまり今我々が必要なメソッドはGUIに関するフレームワークであるApplication Kitの一部である、ということが推測できます。このフレームワークに関する情報をどうやって見つけたら良いか調べてみましょう。

Xcodeでヘルプメニューから「マニュアル」を選択します。マニュアルウィンドウの検索フィールドで「全文検索」を選択してください（検索フィールドの虫眼鏡ボタンをクリックしてメニューの中から選びます）。そして「Application Kit」と入力してリターンキーを押します。

Xcodeは複数の検索結果を表示しますので、その中から「Application Kit Framework Reference」を選択してください。このフレームワークが提供している機能の一覧を読むことができます。
「Protocol References」の中に「NSNibAwaking」というリンクがあります。

NSNibAwaking Protocol Reference

(informal protocol)

Framework /System/Library/Frameworks/AppKit.framework

Declared in AppKit/NSNibLoading.h

Companion guide Loading Resources

Protocol Description

This informal protocol consists of a single method, awakeFromNib. Classes can implement this method to perform final

initialization of state after objects have been loaded from an Interface Builder archive.

awakeFromNibを実装すると、nibファイルからそのオブジェクトが読み込まれた時、このメソッドが呼び出されます。したがってこのメソッドを使うことでこの章のゴール、起動時にテキストフィールドに値を表示することができます。

このような方法で正しいメソッドを探すことが常に良い方法だとはいえません。ドキュメントを眺めたり検索するために適当なキーワードを入力したりして、適当なメソッドを見つけることもあります。しかし重要なことは二つのフレームワークのドキュメントをよく読んでおくことです。そうすればどのクラス、どのメソッドが適当な解決方法なのかがすぐにわかるようになります。現時点でそこまでは必要ありませんが、いずれプログラムを作成する時にそのような知識は大きな助けとなることでしょう。

OK、必要なメソッドは見つかりました。やるべきことはMAFoo.mファイルにこのメソッドを追加して、コードの中身を実装することです[1.15]。

```
[1]
#import “MAFoo.h”

@implementation MAFoo

- (IBAction)reset:(id)sender
{
    [textField setIntValue:0];
}

- (IBAction)setTo5:(id)sender
{
    [textField setIntValue:5];
}

- (void)awakeFromNib    // [1.15]
{
    [textField setIntValue:0];
}

@end
```

ウィンドウが開くと、awakeFromNibメソッドが自動的に呼び出されます。その結果、開かれたウィンドウを見ると、テキストフィールドは既に0を表示しています。

11: ポインタ

■注意！

この章ではC言語の重要で先進的な概念を学ぶのですが、ここはC言語を学ぶ際多くの初心者が壁にぶつかる難所になっています。もし一度読んだだけで理解できなくても心配しないでください。ポインタの概念はプログラミング上とても便利ですが、幸いなことにObjective-Cでプログラムを始める時点では必ずしも必須の知識ではありません。今はわからなくても何度も書いていれば次第に理解することができるでしょう。

■はじめに

あなたが変数を定義すると、変数の値を保存するためにMacはこの変数にメモリを割り当てます。

例えば以下のようなコードを見てみましょう。

```
[1]
int x = 4;
```

これを実行するためにMacはまだ使われていないメモリスペースを探し、変数xの値をここに保存するように指定します（もちろん変数名は何でもかまいません）。変数の型（ここではint）から、Macはxの値を保存するのにどれだけのメモリが必要なのか判断します。もしxがlong longやdouble型であれば、もっとたくさんのメモリが確保されます。

x=4のような代入により、確保されたメモリに4という値が保存されます。もちろんコンピュータはxという名前の変数の値がメモリのどこ、専門的にいうとメモリのどのアドレスに保存されているかを記憶しています。こうして、あなたが変数xをプログラム内で使うたびに、コンピュータはメモリ上の正しい位置（正しいアドレス）を参照し、xの値を見つけることができるのです。

「ポインタ」とは**ある変数のメモリアドレスを示す変数のこと**です（「ポインタ」という名前は「ある変数を指し示す＝ポイントするもの」ということです）。

■変数を参照する

変数の前に&を付けることで、その変数のアドレス、つまりポインタを取り出すことができます。例えば変数xのアドレスを取り出す場合は&xと書きます。

コンピュータがxを使用する時、xの値、上記サンプルでは4を返します。対照的に&xは変数xの値ではなく変数xのアドレスを返します。アドレスは、コンピュータの特定のメモリの場所を指し示す数値です（ホテル内の特定の部屋を指し示すルームナンバーと同じです）。

■ポインタを使う

ポインタはこのように宣言します。

```
[2]
int *y;
```

この命令はint型の変数の『メモリアドレス』を保存するためのyという変数を宣言しています。繰り返します。yはint型の『値』は持っておらず、int型の変数の『メモリアドレス』を保存します。変数yに変数xの値を保存する（変数xのアドレスをポインタyに代入する）ためには以下のようにします。

```
[3]
y = &x;
```

これでyはxのアドレスを指すようになりました。yを利用することで変数xを見つけることができます。与えられたポインタからそのポインタが指している変数の値を取り出すには、ポインタの前にアスタリスク*を付けます。例えば

```
*y
```

は4になります。これはxの値を使うのと同じことです。

```
*y = 5
```

は、x=5と同じです。

ここで[2]の宣言を見直してみてください。[2]では突然*記号がでてきて驚いたかもしれませんが、*y（つまりポインタyが指し示している変数）の型はintである、と考えれば今までに勉強してきた「int x;」といった書き方と同じことだと理解できると思います。

変数の値ではなく変数のアドレスを参照して何か処理を行うとき、ポインタはとても便利です。例えば変数に1を加える関数を作成するとします。では、こういう風に書くことはできるでしょうか？

```
[4]
void increment(int x)
{
    x = x + 1;
}
```

答えはNoです。この関数を実行しても、あなたが望んだ通りの結果を得ることはできません。

```
[5]
int myValue = 6;
increment(myValue);
NSLog(@"%d:\n", myValue);
```

このコードは6を表示するはずです。なぜでしょうか？関数increment()を呼ぶことでmyValueの値を増やしたはずでは？いいえ、実は増やしてはいません。関数[4]は変数myValueの『値』（つまり6）を取り出して1増やし・・・基本的には増やした値を捨ててしまっています。関数というのはあなたが渡した『値』を使って作業をするだけで、その値を保存していた変数に対しては何もしません（myValueとxは値は同じでも、全く別々の変数である、ということに注意してください。xを1増やしていますが、myValueを1増やしているわけではないのです）。[4]のように変数xを加工しても、変数xが受け取った値を加工しているだけです。そしてそのような変更は関数の処理が終わったら失われてしまいます。そもそもxは変数である必要すらありません。もしincrement(5);を実行したら、一体何の値が増やされるのでしょうか？

正しく動作する関数increment()を書きたいのであれば、つまり変数を受け取ってその変数の値を増やしたいのであれば、変数の『アドレス』を関数に渡してやる必要があります。そうすることで、一時的な値ではなく、変数が保存している値を加工することができます。以下のようにポインタを引き数として渡してやります。

```
[6]
void increment(int *y)
{
    *y = *y + 1;
}
```

そしてこの関数はこのようにして呼び出します。

```
[7]
int myValue = 6;
increment(&myValue);// 変数のアドレスを渡す
// これでmyValueは7になる
```

12: 文字列

■はじめに

ここまでいくつか基本的なデータ型：int、long、float、double、BOOLについて勉強しました。そして前の章ではポインタを紹介しました。文字列に関しては、NSLog()関数との関連でのみ学びました。この関数には%記号から始まるいくつかの文字、たとえば%dなどを値と差し替えて、それを画面上に表示する機能があります。

```
[1]
float piValue = 3.1416;
NSLog(@”Here are three examples of strings printed to the screen.\n”);
NSLog(@”Pi approximates %10.4f.\n”, piValue);
NSLog(@”The number of eyes of a dice is %d.\n”, 6);
```

文字列に関してはデータ型としてはまだ学んでいません。なぜかというと、intやfloatとは違い、文字列はNSStringクラスやNSMutableStringクラスを使って作る、本物のオブジェクトだからです。ではこれらのクラスについて学びましょう。まずはNSStringからです。

NSString

■ポインタ再び

```
[2]
NSString *favoriteComputer;
favoriteComputer = @”Mac!”;
NSLog(favoriteComputer);
```

[2.2]の記述は理解できるのではないかと思いますが、[2.1]は少し説明が必要です。ポインタを宣言するときは、それが何の型のデータへのポインタなのかを明示する必要がある、ということをもう一度思い出してください。11章[2]のサンプルはこうでした。

```
[3]
int *y;
```

[3]はポインタyはint型のデータが保存されているメモリのアドレスを指している、ということをコンパイラに指示しています。

同様に[2.1]では、favoriteComputerというポインタはNSString型のデータが保存されているメモリアドレスを指している、ということをコンパイラに伝えています。Objective-Cではオブジェクトは直接ではなく、必ずポインタを通して操作することに決まっているため、文字列を保持するためにポインタを使っています。

この意味を完全に理解できなくても心配しないでください。そこまで重要な問題ではありません。重要なことは、NSStringやNSMutableStringのインスタンスを参照するときは、常に*記号を使う、という点です。

■@記号

では[2.2]の妙な@記号は一体なんでしょうか。C言語の拡張版のObjective-Cはは独自の文字列処理の方法を持っています。C言語の文字列型とは区別するために、完全なオブジェクトである文字列にはObjective-Cは@記号を使います。

■新しいタイプの文字列

C言語の文字列をObjective-Cはどのように改良したのでしょうか。Objective-Cは文字列の文字コードとして、ASCIIではなくUnicodeを使っています（文字コードとはデジタルデータで文字を表現する方法です）。ASCIIではアルファベットと平仮名片仮名程度しか扱うことができませんでしたが、Unicodeの文字列は、日本語や中国語などいかなる言語の文字もアルファベットと同様に表示することができます。

■練習

もちろん[4]のように一行で文字列を宣言し初期化（最初に値を代入）することも出来ます。

```
[4]
NSString *favoriteActress = @”Julia”;
```

ポインタfavoriteActressは、「Julia」という文字列を表すオブジェクトが保存されているメモリの位置を指し示しています。

変数favoriteComputerを初期化したあとで違う値を代入することはできますが、文字列そのものを変更することはできません。これについてはもう少しあとで説明します。

```
[5]
#import <Foundation/Foundation.h>

int main (int argc, const char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSString *x;
    x = @”iBook”; // [5.7]
    x = @”MacBook Pro Intel”;// ノートをもっと新しいものに換えてみました
    NSLog(x);

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

このプログラムを実行すると、以下のように表示されます。

MacBook Pro Intel

■NSMutableString

NSStringクラスの文字列は変更（編集）することができないため、immutable（不変）と呼ばれます。

変更できないと何が良いのでしょうか？変更できない文字列は、OSにとって扱いやすく、プログラムの実行速度が早くなります。そして実際に自分のプログラムを作ってみると、ほとんどの場合文字列を変更する必要は無い、ということがわかるでしょう。

もちろん時には文字列を変更する必要もあります。そのような場合のために変更可能な文字列オブジェクトを作るためのクラスNSMutableStringが用意されています。これについては後ほど説明します。

■練習

まずは文字列がオブジェクトである、ということをしっかりと理解しましょう。オブジェクトですので、文字列に対してメッセージを送ることができます。例えばlengthメッセージを文字列オブジェクトに送信することができます[6]。

[6]

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

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

    foo = @"Julia!";
    theLength = [foo length]; // [6.10]
    NSLog(@"The length is %d.", theLength);

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

このプログラムを実行すると、以下のように表示されます。

```
The length is 6.
```

プログラマは説明用のサンプルによくfooとbarという変数名を使います。xなどという変数名と同様意味が分からないので、このような名前は良くありません。ここでは、「よくそういう使

い方がされる」という紹介のためにfooを使ってみました。インターネット上の議論などでこういう名前を見かけても、xやyと同様特に意味はありませんので深く考えないでください。

[6.10]でfooオブジェクトにlengthメッセージを送信しています。lengthメソッドはNSStringクラスのメソッドで、以下のように定義されています。

- (unsigned int)length
The number of Unicode characters in the receiver.
（レシーバ（文字列オブジェクト）内のUnicode文字の数を返します。）

[7]のように文字列内の文字を全て大文字に変更することもできます。そのためには文字列オブジェクトに適切なメッセージ、ここではuppercaseStringメッセージを送る必要があります。このようなメソッドはドキュメントの中から自分で探し出すことができるようにしてください（この場合はNSStringクラスのメソッドをチェックします）。このメソッドを受け取ると、文字列オブジェクトは各文字を大文字に変えた新しい文字列オブジェクトを作って返します（自分自身の文字列を変更するわけではありません。新しい別のオブジェクトを作って返します）。

```
[7]
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSString *foo, *bar;
    foo = @"Julia!";
    bar = [foo uppercaseString];
    NSLog(@"%@ is converted into %@.", foo, bar);
    [pool release];
    return 0;
}
```

実行すると以下のように表示されます。

```
Julia! is converted into JULIA!
```

新しい文字列を作成するのではなく、文字列の内容を変更したい場合もあります。そのような場合はNSMutableStringクラスのオブジェクトを使います。NSMutableStringクラスは、文字列の内容を編集するためのメソッドをいくつか提供しています。例えばappendString:メソッドを使えば、レシーバの文字列の後ろに引き数で渡した文字列を追加することができます。

```
[8]
#import <Foundation/Foundation.h>

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

    NSMutableString *foo;                    // [8.7]
    foo = [@"Julia!" mutableCopy];           // [8.8]
    [foo appendString:@" I am happy"];
    NSLog(@"Here is the result: %@.", foo);
    [pool release];
    return 0;
}
```

実行すると以下のように表示されます。

```
Here is the result: Julia! I am happy.
```

[8.8]でmutableCopyメソッド（NSStringクラスが提供しています）は、レシーバと同じ内容の、変更可能な文字列を作成して返します。[8.8]の処理の結果、fooは「Julia!」という文字列を含む変更可能な文字列オブジェクトを指し示します。

■もう一度ポインタの話

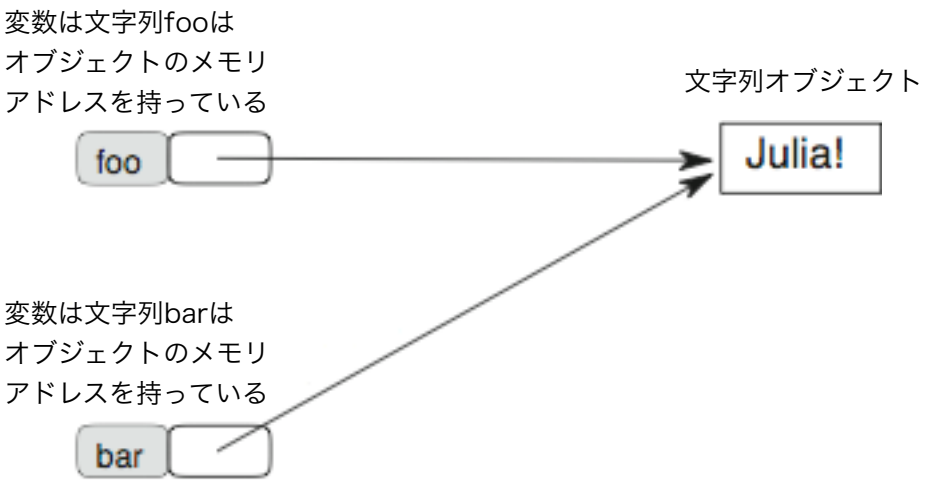
この章の始めに、Objective-Cはオブジェクトを直接操作することは決して無く、必ずポインタを通して操作する、ということを述べました。このため、例えば[8.7]のようにポインタを利用します。実際、Objective-Cで「オブジェクト」という単語を使った場合、一般的には「オブジェクトのポインタ」のことを意味します。ただしいちいちポインタであることを明示しなくても違いは無いため、普通は単に「オブジェクト」と呼びます。オブジェクトは常にポインタを通じて操作するということは、覚えておくべき重要な意味があります。複数の変数が同時に一つのオブジェクトを指し示すことがある、ということです。例えば[8.7]を実行後、ポインタfooは　「Julia!」という文字列を表すオブジェクトを参照します。下図のように表現できます。

オブジェクトは常にポインタを通して操作します

次にfooの値を変数barに代入します。

```
bar = foo;
```

この結果fooとbarはどちらも同じオブジェクトを指し示します。



複数の変数は同じオブジェクトへのポインタを持つことができます

このような状況でfooにメッセージを送る（例えば[foo dosomething];など）のと、barに送る（例えば[bar dosomething];など）のとは全く同じ結果になります。次のサンプルを見てください。

```
[9]
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSMutableString *foo = [@"Julia!" mutableCopy];
    NSMutableString *bar = foo;
    NSLog(@"foo points to the string: %@.", foo);
    NSLog(@"bar points to the string: %@.", bar);
    NSLog(@"----------");
    [foo appendString:@" I am happy"];
    NSLog(@"foo points to the string: %@.", foo);
    NSLog(@"bar points to the string: %@.", bar);
    [pool release];
    return 0;
}
```

実行すると以下のように表示されます。

```
foo points to the string: Julia!
bar points to the string: Julia!
----------------------------
foo points to the string: Julia! I am happy
bar points to the string: Julia! I am happy
```

複数のポインタが同時に同じオブジェクトの参照を持つことができる、というのはオブジェクト指向言語の大きな特徴の一つです。実はこの機能を前の章で既に利用しています。例えば8章では二つの異なるボタンが同時にMAFooオブジェクトのリファレンスを持っています。

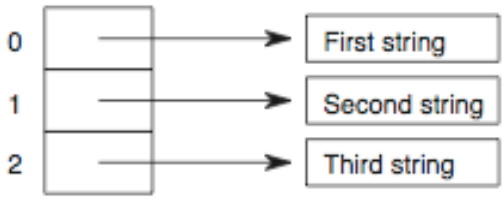
13: 配列

■はじめに

プログラムではデータの集合を管理することが良くあります。例えば文字列のリストを管理する必要があるかもしれません。しかし複数の文字列それぞれに変数を用意するのはとても面倒です。もちろんもっと便利な方法があります。配列（array）です。

配列はオブジェクト（正確にはオブジェクトのポインタ）が順番に並んだリストです。オブジェクトを配列に追加したり、取り除いたり、特定のインデックス（つまり特定の場所）に保存されているオブジェクトを取り出したりすることができます。また配列の中にいくつのオブジェクトが入っているのかを調べることもできます。

オブジェクトの順番を数える時、普通は1から始めますが、配列では一番最初のオブジェクトは0番で二番目が1番となります。



例：三つの文字列を保存する配列

配列のインデックスは0番から始まるということがわかるサンプルコードを、後ほど紹介します。

配列の機能は二種類のクラス、NSArrayとNSMutableArrayによって提供されています。文字列と同じように変更できるもの（mutable）と出来ないもの（immutable）の二種類がありますが、この章では変更可能なものを取り上げます。

> **これらの配列はObjective-C及びCocoa環境独自のものです。他にも、よりシンプルな機能のC言語の配列（Objective-Cにも含まれています）もありますが、ここではそれは取り上げません。いずれCの配列について学ぶ機会もあると思いますが、それはNSArrayやNSMutableArrayほど多機能ではないということだけ覚えておいてください。**

■クラスメソッド

配列を作成する一つの方法は以下のように命令することです。

```
[NSMutableArray array];
```

このコードが実行されるとからの配列を作成して返します。しかし・・・ちょっと待ってください。このコードはちょっと変じゃないですか？このコードでは「NSMutableArray」クラスの名前をメッセージのレシーバに指定しています。しかしこれまではメッセージは常にクラスではなくインスタンスに送ってきましたよね。

ではここで新しいことを一つ学びましょう。Objective-Cではメッセージをクラスにも送ることができます（実はクラスもオブジェクトの一種であり、「メタクラス」と呼ばれるもののインスタンスなのです。しかしこの入門編ではこれ以上詳しくは説明しません）。

このようにして作られたオブジェクトは自動的に解放される、ということは覚えておいてください。このオブジェクトはNSAutoreleasePoolというクラスのオブジェクトに取り込まれ、それを作成したクラスのクラスメソッド（クラスが実行できるメソッド。これに対しインスタンスが実行できるメソッドはインスタンスメソッドと呼びます）で解放されます。このクラスメソッドを呼ぶのは以下の命令と同じです。

```
NSMutableArray *array = [[[NSMutableArray alloc] init] autorelease];
```

NSAutoreleasePoolのライフスパンよりも長く配列を保持したい場合は、配列のインスタンスにretainメッセージを送ってやる必要があります。

Cocoaのドキュメントでは、クラスに対して送ることができるメソッド（クラスメソッド）は、例えば8章[4.5]のサンプルのような-記号ではなく、先頭に+記号がついています。例えばarrayメソッドのドキュメントを見ると以下のように説明されています。

array

Creates and returns an empty array.

+ (id)array

Discussion
This method is used by mutable subclasses of NSArray.

Availability
Available in Mac OS X v10.0 and later.

See Also
+ arrayWithObject:
+ arrayWithObjects:

■練習

ではプログラミングに戻りましょう。以下のプログラムは空の配列を作成して、三つの文字列をその中に格納し、配列内の項目の番号を表示します。

```
[1]
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSMutableArray *myArray = [NSMutableArray array];
    [myArray addObject:@"first string"];
    [myArray addObject:@"second string"];
    [myArray addObject:@"third string"];
    int count = [myArray count];
    NSLog(@"There are %d elements in my array", count);
    [pool release];
    return 0;
}
```

実行すると以下のように表示されます。

```
There are 3 elements in my array
```

以下のプログラムはインデックス0番（配列の先頭の項目）を表示する、という点以外は前のサンプルと同じです。配列内の項目を取り出すには「objectAtIndex:」メソッドを使用します[2.13]。

```
[2]
#import <Foundation/Foundation.h>

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

    NSMutableArray *myArray = [NSMutableArray array];

    [myArray addObject:@"first string"];
    [myArray addObject:@"second string"];
    [myArray addObject:@"third string"];

    NSString *element = [myArray objectAtIndex:0]; // [2.13]

    NSLog(@"The element at index 0 in the array is: %@", element);
    [pool release];
    return 0;
}
```

実行すると以下のように表示されます。

```
The element at index 0 in the array is: first string
```

配列内の項目それぞれに対して何か処理を行うために、配列内の項目をひとつひとつ取り出していくことが良くあります。そのためには7章で学んだ繰り返し処理を使います。次のサンプルはインデックス順に配列内の各項目を取り出して表示させるサンプルです。

[3]

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

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

    NSMutableArray *myArray = [NSMutableArray array];

    [myArray addObject:@"first string"];
    [myArray addObject:@"second string"];
    [myArray addObject:@"third string"];

    int i;
    int count;

    for (i = 0, count = [myArray count];i < count;i = i + 1)
    {
        NSString *element = [myArray objectAtIndex:i];
        NSLog(@"The element at index %d in the array is: %@", i, element);
    }

    [pool release];
}
```

実行すると以下のように表示されます。

```
The element at index 0 in the array is: first string
The element at index 1 in the array is: second string
The element at index 2 in the array is: third string
```

配列は文字列に限らずどんなオブジェクトでも保持（格納）することができます。

NSArrayとNSMutableArrayクラスは他にも様々なメソッドを持っているので、他のメソッドについて学ぶために、ぜひこれらのクラスのドキュメントを読んでください。最後に特定のインデックスのオブジェクトを、他のオブジェクトと差し替えるメソッドを紹介します。「replaceObjectAtIndex:withObject:」というメソッドです。

今までメソッドに渡す引き数は多くて一つでした。しかしこのメソッドは二つの引き数をとります（なのでここで紹介するのです）。メソッド名が二つのコロン:を含んでいるので、二つの引き数をとるということがわかるはずです。Objective-Cではメソッドはいくつでも引き数をとることができます。このメソッドは以下のように使います。

```
[4]
[myArray replaceObjectAtIndex:1 withObject:@”Hello”];
```

このメソッドを実行するとインデックス1（つまり前から二番目）のオブジェクトは@"Hello"に変わります。もちろんメソッドには正しいインデックスを渡さなくてはいけません。つまりメソッドが正しくオブジェクトを差し替えられるように、既にオブジェクトが保持されているインデックスを渡さなくては行けません（例えば二つしか項目のない配列にIndex:2を渡してはダメです）。

■まとめ

見ての通り、Objective-Cのメソッド名は所々に空欄（空欄の前には:がついています）のある文章のようなものです。メソッドを使う際はこの空欄を実際の値で埋めて、よりわかりやすい文章にします。このようなメソッドの記述方法は（Objective-Cの元となっているプログラミング言語の）Smalltalkから由来しており、Objective-Cの大きな長所の一つでもあります。これによってコードをとてもわかりやすく記述することができます。あなたがメソッドを自作するときは、よりわかりやすい（叙述的な）メソッド名を付けるように努力してください。そうすればObjective-Cのコードをより読みやすくすることができ、ひいてはあなたのプログラムをよりメンテナンスしやすく保つことができます。

14: メモリ管理

■はじめに

いくつもの章において、サンプルコード中のいくつかの行の説明を省いてきました。これらはメモリに関するコードです。あなたのプログラムはMac上で動作している唯一のプログラムではありませんし、プログラムが使うRAMは有限の、大切なリソースです。だから、もしあなたのプログラムがメモリの一部を必要としなくなったら、そのメモリはシステムに返してやる必要があります。お母さんから、礼儀正しく行動し、周りの人々と協調しなさいと言われたことがあると思いますが、それはプログラムに関しても同じことなのです！たとえあなたのプログラムがMac上で唯一のプログラムだったとしても、メモリを解放しなかったら最終的にはプログラムは困った状態に陥り、Macの動作速度が遅くなってしまうでしょう。

■オブジェクトのライフサイクル

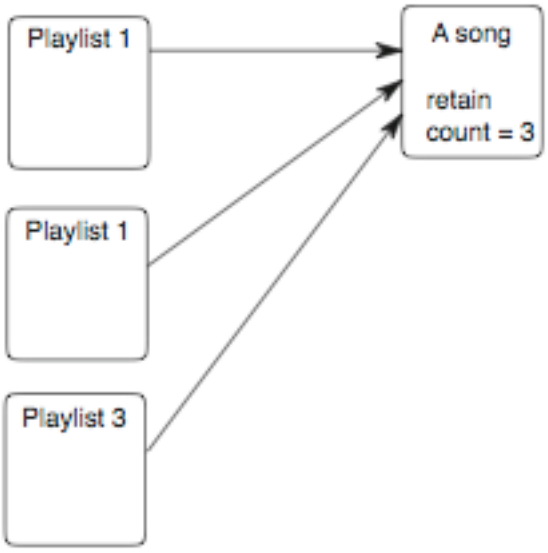
プログラムがオブジェクトを作ると、オブジェクトはメモリ上の一部を使用（「メモリを占有する」といいます）します。なのでオブジェクトがもう必要なくなったら、そのメモリスペースを解放してやる必要があります。つまりオブジェクトを必要としなくなったら、そのオブジェクトを破棄するのです。しかしオブジェクトが必要かどうか判断するのは決して簡単なことではありません。

例えばプログラムの実行中、あなたのオブジェクトは他の多くのオブジェクトから参照されているかもしれません。だから他のオブジェクトから操作される可能性があるうちは、そのオブジェクトを破棄してしまうことはできません（破棄されたオブジェクトを操作しようとすると、プログラムはクラッシュしたり、予期できない動作をしたりします）。

■retainカウント

必要なくなったオブジェクトを破棄するために、Cocoaは各オブジェクトにretainカウントと呼ばれるカウンタを準備しています。プログラム中、もしあなたがあるオブジェクトへの参照を持ちたいとき、retainカウントを1増やすことで、そのオブジェクトを自分が参照している、ということを知らせなくてはいけません。そしてそのオブジェクトの参照を破棄するとき（もうオブジェクトを操作する必要がなくなったとき）、retainカウントを1減らして、そのオブジェクトをもう参照しない、ということを知らせます。そしてretainカウントが0になったら、オブジェクトはもうどこからも参照されていない、ということがわかり、安全に破棄されるのです。オブジェクトが破棄されると、関連するメモリ領域も自動的に解放されます。

例えば、あなたのプログラムがジュークボックスソフトウェアで、曲とプレイリストを表すオブジェクトがあるとします。ある曲が三つのプレイリストオブジェクトから参照されていて、それ以外からは参照されていない、とすると、その曲オブジェクトのretainカウントは3になるはずです。



オブジェクトはretainカウントのおかげで、自分がいくつのオブジェクトから参照されているかわかります

■retainとrelease

オブジェクトのretainカウントを増やすためには、retainメッセージを送信します。

```
[anObject retain];
```

オブジェクトのretainカウントを減らすためには、releaseメッセージを送信します。

```
[anObject release];
```

■Autorelease

Cocoaはautorelease poolと呼ばれる機能も提供しています。これを使うと、すぐではなく、後でオブジェクトにreleaseメッセージを送ることができます。この機能を使うには、オブジェクトに対してautoreleaseメッセージを送ってautorelease poolにオブジェクトを登録してやる必要があります。

```
[anObject autorelease];
```

autorelease poolは登録されたオブジェクトに対し、後でreleaseメッセージを送ってくれます。前に勉強したいくつかのサンプルコードで出てきたautorelease poolに関連する記述は、autorelease poolの機能の準備をするための命令だったのです。

■ガベージコレクション

この章で説明したCocoaのメモリ管理技術を、一般的に「Reference Counting」と呼びます。より高度な本や記事で、Cocoaのメモリ管理システムのきちんとした説明を読むことができます。

そして、「Automatic Garbage Collection」（自動ガベージコレクション）と呼ばれる、新しいメモリ管理システムがもうすぐCocoaに導入されます。このシステムは今のシステムよりもずっと強力で、より使いやすく、そしてエラーも少なくなっています。このシステムはMac OS X 10.5 Leopardから導入される予定です。

15: 情報源

このガイドのささやかなゴール地点は、Xcode環境で使うObjective-Cの基本をあなたに学んでもらうことです。もしこのガイドを二度繰り返して読んで、サンプルを自分なりに変更して試してみたのであれば、あなたが作りたいと考えているキラーアプリの作り方を学ぶ準備はもう完了です。問題に直面したとき、それを素早く解決するための十分な知識も学びました。この章まで読み通したなら、その他の本などのリソースを利用することも可能です。以下にあげる記事や本はあなたの興味を引くことでしょう。

■ウェブサイト

コードを書き始める前に、大事なアドバイスがあります。あわてて始めてはいけません！少しコードを書くだけで使うことができるたくさんの便利なクラスを、アップルはすでに提供してくれています。まずはフレームワークをチェックしてください。また誰か別の人があなたの代わりにもう既に同じ機能を作って、ソースコードを公開しているかもしれません。だから、マニュアルやインターネット上でそういう情報を探して、いちいち自分で作る手間を減らしましょう。まずはアップルの開発者向けのサイトを覗いてください。

http://www.apple.com/developer

また以下のサイトをブックマークすることをお勧めします。

http://osx.hyperjeff.net/reference/CocoaArticles.php
http://www.cocoadev.com
http://www.cocoadevcentral.com
http://www.cocoabuilder.com
http://www.stepwise.com
http://www.cocoalab.com

これらのサイトには、別のサイトや情報へのたくさんのリンクがあります。またcocoa-devにも登録しましょう（ただし英語）。

http://lists.apple.com/mailman/listinfo/cocoa-dev

質問があったら、ここにポストすることができます。親切なメンバーがあなたを助けてくれるはずです。そのかわり過去の質問と回答（http://www.cocoabuilder.com）を検索して、自分の質問に対する答えを見つけられるかどうかを最初に調べてください。またメーリングリストに質問を投稿する際の注意点がまとめられていますので、「How To Ask Questions The Smart Way」（http://www.catb.org/~esr/faqs/smart-questions.html）というドキュメントにも目を通してください。

またCocoa開発に関する素晴らしい本もいくつか出版されています。Stephen Kochan氏の『Programming in Objective-C』は初心者向けです。いくつかの本を読むには、この本で得られる程度の最低限の知識が必要です。Xcode開発のセミナーの講師を仕事にしているBig Nerd RanchのAaron Hillegass氏の『Cocoa Programming for Mac OS X』（日本語訳 『MacOS X Cocoaプログ

ラミング』ピアソンエデュケーション）はおすすめです。またO'Reillyから出版されているJames Duncan Davidson氏とアップルによる『Cocoa with Objective-C』もおすすめできます。

■自作のアプリケーション

最後に少し注意点を。あなたはMacのプログラムを作っています。プログラムを公開する前に、バグが無いのを確認するだけでなく、アプリの外観が整理されていて、かつ「Apple human interface guidelines」（英語）に準拠していることを確認してください。

> http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/index.html

確認し終えたら、もう公開を恥ずかしがらないでください。他の人からのコメントは、あなたのプログラムを改良し、拡張する上で大きな助けになります。

あなたがこのガイドを楽しんで読むことができ、またXcodeでのプログラミングを続けてくれることを願っています。そうそう、0章のお願いも忘れないでくださいね。

Bert, Alex, Philippe.

■訳者あとがき

全く知識の無い人向けのガイド、ということでなるべく簡単な説明を目指したのですが、読み返してみるとやはりまだまだ難しいなあと反省しています。私のホームページの方でいくらかの補足や練習問題を掲載していますので、読み終わったらぜひチェックしてみてください。またこのドキュメントの間違いやわかりにくかった点の指摘、感想などありましたら掲示板の方に書込んでください。あと、このガイドの内容に限らず、プログラミングに関する質問（初歩的なものに限りますが）がありましたら書込んでください。可能な限りお答えします。

http://homepage.mac.com/nsekine/SYW/SYWSoft/learnCocoa

日本語版改訂履歴

- 2007/06/17　誤記、分かりにくい表現等を修正
- 2007/06/09　初版公開