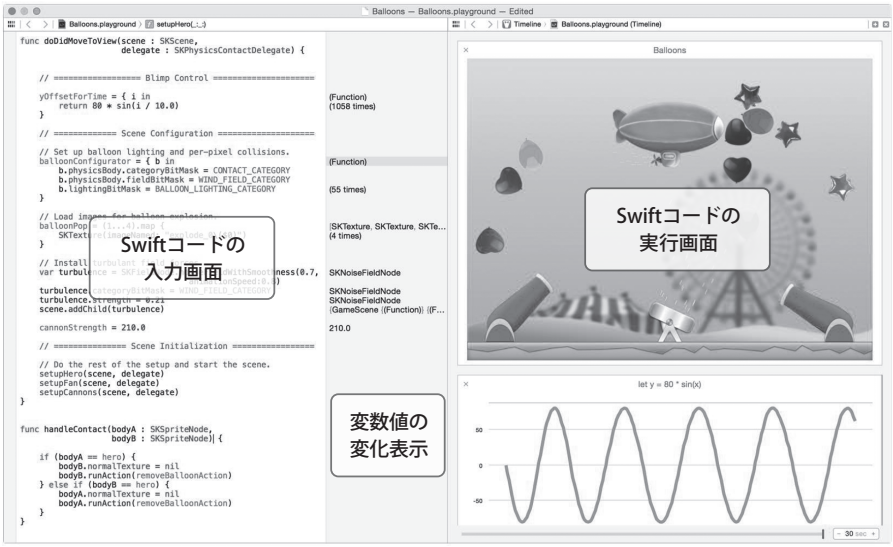
しかし、既に Swift のスーパークラスが多数作られて Swift ライブラリが構築され、Objective-C の実行環境の代わりに Swift 実行環境が作られている。2014 年秋にリリースされれば、将来的にも Swift コードは動作するとされている。

1.4 統合開発環境 Xcode

Swift 言語を動かすためには、2014 年秋に無料でリリースされる Xcode 6 が必要となる。しかし、秋まで待てない人は、iOS Developer 有料会員（8,400 円程度）となり、秘密保持契約（NDA：Non-Disclosure Agreement）を結び、Xcode 6 ベータ版を取得する必要がある。また、7 月 12 日には Xcode 6 ベータ 3 の無料配布が始まった。当然、Xcode 6 ベータ版を動かすためには、OS X（10.9.3 以上）の Mac が必要となる。2014 年 10 月 20 には、Xcode 6.1 が無料でリリースされ、OS X 10.10 Yosemite や 10.9 Marvericks, iOS 8.1 のための SDK が含まれている。本書（第 3 刷）では、Xcode 6.1 に対応するようにプログラム・コードを見直した。

Apple の Swift の Web サイトで Xcode 6 の Playgrounds 画面が公開されているが、これを見る限り、右画面に Swift コードを入力すると、中央に対話的に実行された結果が行ごとに見えており、コンパイル操作することなくインタプリタのように、コードを作成している矢先から変数の値の変化を見ることができ、対話的なコードの動作確認が可能となる。また、右画面にはその変数をグラフ化したり、コンソール画面が見られる。本書では、Swift コード例を多数掲載したので、この Swift コードの入力画面にそれらを入力すれば、すぐに実行結果を確かめることができる。

Apple SwiftのWebサイトで公開されているPlaygrounds画面
（https://developer.apple.com/swift/）



1.5 Swift 言語の参考資料

本書は、Apple が公開している言語仕様 The Swift Programming Language のドキュメントを主に参考にして、独自に調査して、プログラム例を作成し、執筆した。そのために、実際の Xcode 6 のリリースまで、いろいろな変更点があると予想されるが、本書では反映されない点が出てくる可能性があり得ることをご了承願いたい。変更点は逐次、つぎの Swift のブログで報告されるので、フォローしていただきたい。本書（第 3 刷）では増刷にともない、Xcode 6.1 に対応したプログラムコードに変更した。Apple の Swift 言語関連として、つぎのような Web サイトがある。

さらに、String 型には、文字列を整数値にする toInt() メソッドがあるが、数字の文字列でなければ整数に変えられないので、戻り値はオプショナルになっている。そのために、オプショナルを開いて整数があればその整数を取り出すために、！を後置して開く必要がある。

つぎのように 2 つの数字の文字列を toInt() メソッドで整数にして、足し算ができるか確かめてみよう。

```
//Sw3-11.swift　数字の文字列を整数にする方法
var n = "12"; var m = "45"                ←数字の含まれた文字列を生成
println( n.toInt()! + m.toInt()! )        ←オプショナルを!で開き、足すと57と表示される
```

ここで、文字列が数字でなく、整数に変換できないときは、nil となり、実行エラーが出る。

3.2.3　文字列演算

文字列に使える演算子があり、たとえば、2 つの文字列を重ねる**文字列連結（string concatenation）**には、2 つの文字列を + 演算子で足すだけでよい。つぎのように、文字列の足し算や同じ文字列かどうかを調べたり、文字列を辞書順に大小関係を比較できる。

表3.2●文字列Stringに使える演算子

	演算子	意味
○	+	2 つの文字列を連結し、連結文字列を生成する
○	+=	文字列の最後に文字列を追加し、追加文字列を生成する
○	==	2 つの文字列が同じであれば true、異なれば false を返す
○	<	2 つの文字列を辞書順に大小比較し、真なら true 偽なら false を返す

演算子 + では、オペランドに String 型が使えて、つぎのように文字列の連結が可能となる。Character 型同士の連結は、エラーとなりできなくなった。

```
//Sw3-12.swift　文字列や文字の足し算で合成文字列の作り方（一部エラーとなる）
let c: Character = "S"                ←文字Character型定数に文字Sを割り当て
let s: String = "wift"                ←文字列String型定数に文字列wiftを割り当て
println( c + c )                      ←Character型同士の連結はエラーとなる
println( c + s )                      ←Character型とString型との連結はエラーとなる
println( s + s )                      ←String型同士が連結されてString型のwiftwiftとなる
```

ただし、Character 型には 1 文字しか入らないので、Character 型変数には連結できないことに注意すること。

また、演算子 += でも、オペランドに String 型が使えて、つぎのように文字列の追加をしてみよう。

```
//Sw3-13.swift　文字列に文字列を追加する方法(一部エラーとなる)
var s = "Swift"
s += " is "                          ←文字列sの最後に文字列" is "が追加される
println(s)                           ←Swift is と表示される
let a = "great!"                     ←文字列定数aを設定
s += a                               ←文字列変数sの最後に文字列定数aが追加される
println(s)                           ←Swift is great!と表示される
a += s                               ←文字列定数aに文字列を追加できないので、エラーとなる
```

２つの文字列を比較して、それぞれの文字列が全く同じ文字列であるかどうかを調べるとき、つぎのように、文字列同士の等価性の比較は == 演算子を用い、文字列の比較を行う。== 演算子による文字列の比較結果は、文字列が同じであれば true を返し、異なれば false を返す。

```
//Sw3-14.swift　同じ文字列かを調べる方法
let s = "Swift"
println(s == "Swift")                ←文字列が同じなのでtrueと表示される
println(s == "swift")                ←大文字・小文字は区別するので、falseと表示される
```

最後に、関係演算 <(小なり)は、2つの文字列をアルファベットの辞書順に大小関係を比較し、真なら true を返し、偽なら false を返す。たとえば、つぎのような文字列を使って、辞書順での大小関係を真偽判定で調べてみよう。

```
//Sw3-15.swift　文字列をアルファベットの辞書順で大小を比較する方法
let a = "ABC"
let z = "XYZ"
println( a < z )                     ←アルファベットの辞書順なので、大小関係はtrueと表示される
println( "z" < "a" )                 ←直接に文字列を入れてもよく、falseと表示される
```

```
}
println(w == .Sun)                          ←メンバー値は同じなので、trueと表示される
```

7.2 連想値

列挙型のメンバーは、それ自体が値として取り扱われて、メンバー値ともなっていた。しかし、それ以外に、**連想値（associated value）**をメンバー値に付与することができる。この連想値は、メンバーやメンバー値を連想させるように、1 つのメンバーに複数個付与してもよい。

たとえば、つぎのプログラムは、列挙型の各メンバーの後ろに丸括弧 () を付けて、連想値として、文字列を付与したプログラム例である。メンバーに連想値が付けられたので、文字列を代入できる。

```
//Sw7-5.swift  列挙型のメンバーに連想値を付ける仕方
enum Color {
    case Red(String)
    case Green(String, Int)
    case Blue(String, Double)
}
var r = Color.Red("R")                      ←列挙型メンバーRedに連想値"R"を追加できる
```

7.3 素値

列挙型メンバーにいろいろな連想値が追加できたが、同じ型をメンバーに初期化して付けるときには、つぎのように生の値として**素値（raw value）**を付けられる。素値には、Character や String、Int、Double などの型を指定でき、これらを列挙型名の後ろに型注釈として付ける。各メンバーから素値を取り出すためには、つぎのように rawValue プロパ

ティを使う。素値からメンバー値を得るには、イニシャライザー init?(rawValue:) を使う。

```
//Sw7-6.swift  列挙型のメンバーに素値を付けてメソッドの使う方法
enum Color: Character {                          ←列挙型の素値の型注釈を付ける
    case Red = "R"
    case Green = "G"
    case Blue = "B"
}
var c = Color.Red.rawValue                       ←メンバーRedから素値を取り出す
println(c)                                       ←素値が出力され、Rと表示される
c = Color.Blue.rawValue                          ←メンバーBlueから素値を取り出す
println(c)                                       ←素値が出力され、Bと表示される
println(Color.Blue == Color(rawValue: "B"))      ←メンバーは一致しtrueと表示される
```

ここで注意することは、各メンバーの素値はすべて同じ型で、列挙型の宣言のときにのみ、デフォルト値として素値を付けられることである。

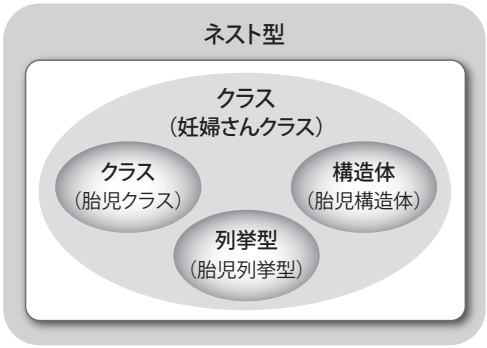
当然、列挙型の中で同じ素値があるとエラーとなる。そのために、自動的に数値を付けてくれると便利なときもある。列挙型の素値の型が Int 型のときには、つぎのように最初の整数の素値を指定すれば、後は自動的に数値が 1 ずつ増分し、Green は 2 となり、Blue は 3 となる。そして、これらの整数の素値は、つぎのように、先に示した rawValue プロパティやイニシャライザー init?(rawValue:) でやり取りできる。

```
//Sw7-7.swift  列挙型のメソッドを使ってswitch文で処理をする方法
enum Color: Int {
    case Red = 1
    case Green
    case Blue
}
var c = Color.Red.rawValue
println(c)                                       ←素値が出力され、1と表示される
c = Color.Blue.rawValue
println(c)                                       ←素値が出力され、3と表示される
if let s = Color(rawValue: 2) {                  ←素値からメンバー値を取り出す
    switch s {                                   ←メンバー値と一致したメンバーを探す
        case .Red: println("R")
        case .Blue: println("B")
        case .Green: println("G")
        default: println("No")
    }                                            ←メンバーGreenが選択されて、Gと表示される
}
```

このように、スーパークラスには、アプリケーションやアルゴリズムを解くための共通性のある基本のプロパティ（半径や高さ）やよく使うメソッド（半径の２乗）を定義する。そのサブクラスには、面積計算などのメソッドを定義し、１つの共通性のあるスーパークラスから円の面積や球の面積、円錐の面積などいくつもの特徴のあるサブクラスを作成できる。さらに、個々のサブクラスから体積計算を行うようなサブクラスを生成して、大きなアプリケーションを開発するためのクラス階層を形成できる。

9.2.2　ネスト型

ネスト型（nested type）とは、構造体やクラスの入れ子状態を意味する。人形の中に人形が入っているロシアのマトリョーシカのように入れ子状態であり、構造体やクラスの中に列挙型、構造体、そして、クラスも設定することが可能である。



たとえば、つぎのような簡単な例でネスト型を作成してみよう。クラスの中でネストされた列挙型と構造体を設定し、外部からクラスや構造体のインスタンスを生成することなく、直接に列挙型名や構造体名で呼び出せるので、便利なときもある。

```
//Sw9-4.swift　クラス中に列挙型と構造体を設定したネスト型の作り方
class C {                                    ←structでも可能
    enum E : Int { case x = 1 }              ←列挙型Eの設定
    struct S { static let y = 2 }            ←構造体Sで型プロパティyを設定
}
println(C.E.x.rawValue)                      ←列挙型EからrawValueで生のxの値を取得し1と表示される
println(C.S.y)                               ←クラスCの構造体Sの型プロパティyを呼び出し、2と表示される
```

また、構造体の中でプロパティとして、ネストされた構造体やクラスのインスタンスを設定

して、プロパティを介して外部から呼び出すこともできる。

```
//Sw9-5.swift  構造体でネスト構造体やネストクラスの作り方
struct C {
    struct S { let y = 2 }          ←構造体Sでインスタンスプロパティyを設定
    class CC { var z = 3 }          ←クラスCCでインスタンスプロパティzを設定
    var s = S()                     ←クラスCのプロパティsとして構造体Sを設定
    var cc = CC()                   ←クラスCのプロパティccとしてクラスCCを設定
}
var c = C()                         ←構造体Cのインスタンスcの生成
println(c.s.y)                      ←cからインスタンスsのプロパティyを呼び出し、2と表示される
println(c.cc.z)                     ←cからインスタンスccのプロパティzを呼び出し、3と表示される
```

9.3 イニシャライザー（初期化）

9.3.1 指名イニシャライザー

これまで、構造体やクラスのイニシャライザーとして簡単に説明してきたが、ここで、もっと詳しくイニシャライザーについて説明しよう。

インスタンスの初期化に使われるイニシャライザーは、キーワード **init** が使われる。正確にはメソッドではないが、インスタンスメソッドのように設定する。つまり、形はインスタンスメソッドに似ているが、キーワード func や戻り値の型、メソッド内で値・式を返すような return は使用しない。このイニシャライザーは、Java 言語の構築子 constructor に相当する。

また、クラスのイニシャライザーの定義の前に required 修飾子を付けると、そのすべてのサブクラスでそのイニシャライザーに required 修飾子を付けて定義する必要がある。