# Java入門

(2003.07.14 updated)

国際文化学部　村尾　元

# 目 次

# 第1章　プログラミング言語Java

## 1.1　プログラミング言語Javaの特徴

プログラミング言語 Java (以下，単に Java と呼ぶ) は主に Smalltalk と C 言語をルーツとするオブジェクト指向プログラミング言語である．`const` の代わりに `final` を使うといったような意識的な構文の違いはあるが，基本的な構文は ANSI C を元にしている．既存のプログラミング言語 (主に C 言語) に比べ，Java は以下のような特徴を有している．

- **オブジェクト指向**：Java はオブジェクト指向言語である．オブジェクト指向とは，巨大なプログラムを，細かなプログラム単位 (オブジェクト) に分解して作成するプログラミング手法である．Java はオブジェクト指向でプログラムを作成するために，クラスという重要な機構を含んでいる．
- **アーキテクチャ中立**：Java のプログラムは，プラットフォームに依存しない**バイトコード**としてコンパイルされ，Java インタプリタを用いて実行される．このことは，Java のプログラムがコンパイルされた後でさえ，Java インタプリタが動作するあらゆるプラットフォームで，再コンパイルする必要なしに実行可能であることを意味している．このため，Java のプログラムでは移植という作業が不必要である．
- **動的結合（ダイナミックバインド，ダイナミックリンク）**：Java は実行に必要なバイトコードを，実行時に探しだし利用する．これにより，プログラマは，プログラムの一部を改変したときに，その全体を再コンパイルする必要がなくなる．
- **コンパイル時の型チェック**：Java は静的な型を持つため，コンパイル時に可能な限りのエラーを見つけることができる．また，静的な型を利用してコンパイル時に最適化を行うことができる．
- **メモリに関する堅牢性**：Java はメモリ管理にガベージコレクションの機構を採用している．実行中に使用されなくなったメモリは自動的に解放されるため，C 言語の系列に見られるメモリリークという質の悪いバグがない．また，Java ではポインタに関する演算がないため，メモリの操作に関して致命的なバグが発生することが少ない．
- **マルチスレッド**：Java はマルチスレッド対応の言語である．今日のネットワークアプリケーションや高度な GUI を有するアプリケーションは複数の処理を同時に行なう必要があるが，Java の有するマルチスレッド機構により，これを容易に記述することができる．

その他にも，ネットワーク環境への親和性，高い安全性，強力な例外処理とデバッグ機構といった点で既存の言語にはない優れた特徴を有している．

## 1.2　Java プログラミング環境

Java のプログラムは，まず「`.java`」で終わる名前のテキストファイルに書かれる．これを**ソースファイル**と呼ぶ．ソースファイルを Java コンパイラによってバイトコードと呼ばれる機種依存しない中間言語

の形態にコンパイルすることで「.class」で終わる名前の**クラスファイル**が生成される．Java インタプリタはクラスファイルに含まれるバイトコードを実行する．そのプログラムの作成手順を図 1.1 に示す．

コンソールで実行されるプログラムは，エディタを用いてソースファイルを作成し，それを javac というコンパイラでコンパイルすることで作成する．実行する際には java というインタプリタを用いる．グラフィックを扱う場合には，アプレット (applet) と呼ばれる形式のプログラムを作成する．これを実行するためにはアプレット実行用のインタプリタ，もしくは Java 対応の Web ブラウザを用いる．

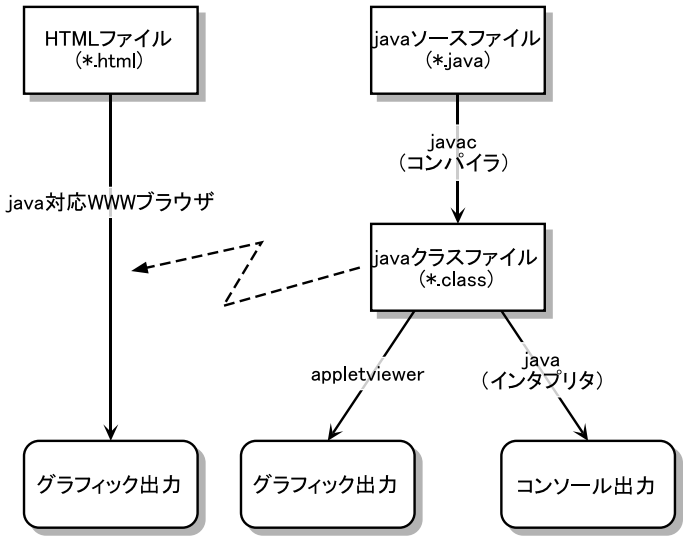


図 1.1: Java プログラミングの流れ

## 1.3　はじめての Java プログラミング

### 1.3.1　準備：Java プログラミングのその前に …

まず最初に開発環境のインストールおよび環境設定をしなくては Java プログラミングに必要なコンパイラやインタプリタの実行ができない．以下の手順に従い Java プログラミングをするのに十分な設定がされているか確認しておこう．必要ならば Java の開発環境のインストールを行う．なお，利用しようとする計算機に Java の開発環境がインストールされているかどうか分からない場合は，計算機の管理者に聞いてみるとよい．

### 1.3.2　Java 開発環境のダウンロード

Java はプログラムの実行に必要な環境（Java 2 Runtime Environment: JRE）と開発環境（Java 2 Software Development Kit: SDK）が別々に配布されている．プログラムの実行だけ出来ればよいという場合には JRE のみをインストールすれば，実行に不要なファイルのインストールはされないのでディスクの節約ができる．SDK には JRE も含まれているので SDK をインストールする場合には JRE をインストールする必要はない．ここでは開発環境 Java 2 SDK のインストールを行う．

まず，http://java.sun.com/j2se/1.4.2/download.html にアクセスし，必要なファイルのダウンロードを行う．

ページが開いたら，“Download J2SE v 1.4.2” となっているグループから，Windows 用であれば，“Windows Offline Installation” を，Linux 用であれば，ディストリビューションに合わせて “Linux RPM in self-extracting file” か “Linux self-extracting file” を選択し，ダウンロードしよう．Linux 用の場合，以下では “Linux self-extracting file” についてのみ説明を行うので，RPM 版をダウンロードした場合は，自助努力でなんとかするということで頑張って欲しい．なお，いずれの場合も間違いなく “SDK” をダウンロードするところだけは間違わないようにしよう．

さて，“DOWNLOAD” というリンクをクリックすると，続いて “Keep Informed” というページに飛ぶ．ここで個人情報を入力することにより，Java に関する新しい情報や，製品情報を送ってもらうようにメイリングリストに登録することができる．ただし，特に設定しなくても，ページの先頭の “download” というリンクをクリックすると次のライセンス承諾画面，さらに次のダウンロード画面へと進むことができる．

### 1.3.3 Java 開発環境のインストール

ダウンロードが終了したら，次は開発環境のインストールを行う．これは OS によって異なるので，それぞれの OS でのインストール方法を参照すること．

#### Windows の場合

Windows の場合，ダウンロードしたファイルをダブルクリックするとインストールが開始する．あとは画面の指示に従ってインストールを行う．ほとんどの場合,「＞次へ」のボタンをクリックしていれば問題ない．この場合，開発環境そのものは `C:\j2sdk1.4.2` にインストールされる．

#### Linux の場合

Linux の場合，まずダウンロードしたファイル（この場合 `j2sdk-1_4_2-linux-i586.bin` のはず）のあるディレクトリで以下のコマンドを実行し，ダウンロードしたファイルを実行可能にする．

```
% chmod +x j2sdk-1_4_2-linux-i586.bin Enter
```

続いて，以下のようにして，`j2sdk-1_4_2-linux-i586.bin` を実行する．

```
% ./j2sdk-1_4_2-linux-i586.bin Enter
```

すると，ライセンスの承諾が確認されたあと，**カレントディレクトリ**に `j2sdk-1_4_2` という名前のサブディレクトリが作成され，その中に全てのファイルが展開される．

標準の設定では，開発環境のシステム設定はこのサブディレクトリに納められるが，Java の開発環境そのものをネットワークを介して共有している場合のように，システム設定は別の場所に納めたい場合は`-localinstall` オプションをつけて以下のようにインストールを実行する．このようにすると，システム設定は `/etc` ディレクトリに作成される．

```
% ./j2sdk-1_4_2-linux-i586.bin -localinstall Enter
```

もし，この開発環境を，同じ計算機を利用している他のユーザと同時に使いたいのであれば `/usr/local` などにインストールすると良いだろう．このためには，`/usr/local` に移動してからインストールを実行すれば良い．

### 1.3.4　環境変数の設定

インストールが終了したら環境変数の確認，設定を行おう．Java で主に用いられる環境変数は以下の通りである．

- PATH：実行ファイルを納めたディレクトリを指定する．複数のディレクトリを指定することができるが，その中に Java のコンパイラやインタプリタを含むディレクトリを指定しておかなければ，Java は実行できない．
- CLASSPATH：Java のコンパイル済みクラスの置き場所を指定する．複数のディレクトリを指定できる．Java のコンパイラやインタプリタは，このディレクトリから，コンパイル時や実行時に必要なファイルを探してくる．特に設定していない場合は，システムの標準ディレクトリから探してくるので，当面，設定してなくても問題はない．

#### Windows の場合

WindowsNT 系 OS（Windows2000 や WindowsXP の場合）は，デスクトップの「マイコンピュータ」のプロパティから「詳細設定」を選び，環境変数に指定する．

- PATH：標準のインストール状態であれば C:\j2sdk1.4.2\bin に設定すれば良い．

  すでに PATH に何かが設定してある可能性もあるが，この場合は，PATH がすでにリストアップされているはずなので，これを選択し,「編集」ボタンを押して，Java の実行ファイルを納めたディレクトリを追加する．この場合，ディレクトリの区切り文字は「;」（セミコロン）を用いる．

  まだ設定されていないようであれば,「新規」ボタンを押し,「変数名」を「PATH」に,「変数値」を「C:\j2sdk1.4.2\bin」に指定する．
- CLASSPATH：「新規」ボタンを押し,「変数名」を「CLASSPATH」に,「変数値」には**自分でクラスファイルを納めようとするディレクトリ**を指定する．通常は，自分のホームディレクトリ（Windows の場合は My Documents だったり）の中に新たなディレクトリを作成し，ここを指定する．

#### Linux の場合

まずターミナルウィンドウ内で下線のように実行し，二つの環境変数 PATH および CLASSPATH が表示されるか，そしてそれらが正しく設定されているかどうか確認しよう．

##### 1) csh 系の場合

利用しているシェルが csh または tcsh と呼ばれるシェルの場合には setenv コマンドで確認を行い，.cshrc または.tcsh という設定ファイルに設定を行う．

```
% setenv | grep PATH Enter
CLASSPATH=.:/home/murao/java                    （設定されているか？）
PATH=...:/usr/local/jdk1.2.2/bin:...   （java の実行パスが含まれているか？）
```

もし，それらが正しく設定されていなければ，ホームディレクトリの下の.cshrc というファイルの最後に次の二行を追加し，ターミナルを起動しなおしてみよう．ただし，下線部は**自分の Java プログラム**

**作成用ディレクトリ**に合わせて変更すること．この例では，自分のホームディレクトリ（$HOME という環境変数に自動的に設定されるのでそれを利用している）の直下に java という名前のディレクトリを作り，その中にプログラムを作成することにしている．

```
setenv PATH ${PATH}:/usr/local/jdk1.2.2/bin
setenv CLASSPATH .:${HOME}/java
```

**2) bash の場合**

利用している OS が Linux の場合，デフォルトのシェルが bash と呼ばれるシェルである場合も多い．この場合には以下のように export コマンドで環境変数の確認を行い，.bashrc というファイルに設定を行う．

```
% export | grep PATH Enter
declare -x CLASSPATH=".:/home/murao/java"          （設定されているか？）
declare -x PATH="...:/usr/local/jdk1.2.2/bin:..."  （java の実行パスが含まれているか？）
```

もし，それらが正しく設定されていなければ，ホームディレクトリの下の.bashrc というファイルの最後に次の二行を追加し，ターミナルを起動しなおしてみよう．

```
export PATH="${PATH}:/usr/local/jdk1.2.2/bin"
export CLASSPATH=".:${HOME}/java"
```

### 1.3.5 はじめての Java プログラムを作成する

さて，多くのプログラミング言語入門の例にもれず，まずは「Hello World」を表示するプログラムを作ってみよう．Java でのソースファイルは以下のようになる．

```
public class HelloWorld {
    public static void main( String[] args )
    {
        System.out.println("Hello World");
    }
}
```

上記のプログラムは**メソッド** main のみを持つ**クラス** HelloWorld を定義している．main メソッドは Java における特殊なメソッドで，クラスをアプリケーションとして実行した場合に最初に呼び出される．これは以下のような書式で記述する．

```
public static void main(String[] args) {
    .....
}
```

これを emacs などのテキストエディタを用いて作成する．ここでは分かりやすくするという意味でも，**ファイル名はクラスの名前に一致する**ように作成することにするので，このプログラムは HelloWorld.java

という名前で作成する．これを以下のように Java コンパイラを用いてコンパイルする．

```
% javac HelloWorld.java Enter
```

すると HelloWorld.class という名前のクラスファイルが生成されるので，これを以下のように Java インタプリタを用いて実行する．

```
% java HelloWorld Enter
```

**演習：環境設定の確認**

HelloWorld.java を作成し，コンパイル，実行してみよう．うまくコンパイル，実行ができない場合は，環境設定がうまく行われていないと考えられる．もう一度確認してみよう．

## 1.4　Java ひとめぐり

### 1.4.1　基本型と変数

ここでは Fibonacci(フィボナッチ) 数列を生成するプログラムを作成する．Fibonacci 数列は，Fibonacci が著した算術書 *Liber abaci*(1202 年) に出て来る次のような問題を解くことで得られる．

> 1 つがいのうさぎは，毎月 1 つがいの子を生む．新しく生まれたうさぎは，一カ月後から子を生み始める．最初 1 つがいのうさぎがいたとすると，一年後には何つがいになるか．

このとき $n$ カ月目のつがいの数 $F_n$ は，$F_1 = F_2 = 1$ より始め，以下のような漸化式で計算される．

$$F_{n+2} = F_{n+1} + F_n, \qquad n = 1, 2, 3, ... \tag{1.1}$$

このとき得られる以下のような数列 $F_n$ が Fibonacci 数列である．

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... \tag{1.2}$$

最も単純なアルゴリズムで作成されたプログラムは以下のように書くことができる．

```
public class Fibonacci {
    /** 100 以下の Fibonacci 数列を表示する */
    public static void main( String[] args )
    {
        int f1 = 1;
        int f2 = 1;

        System.out.println(f1);
        while( f2 < 100 ) {
            System.out.println(f2);
            int t = f2;      // 一時的に変数を退避
            f2 = f1 + f2;
            f1 = t;          /* 古い変数を復帰 */
        }
    }
}
```

このプログラムでも HelloWorld と同様，main メソッドのみを有するクラス Fibonacci を定義している．main の始めの 2 行は 2 つの変数 f1 と f2 を宣言している．全ての変数は型指定が必要であり，C 言語と同様，型名は変数名の前に書く．この例では f1 と f2 の型は int である．int 型は C 言語と異なり，32 ビット符号付き整数と決められている．

また，C 言語と異なり (C++言語と同様)，変数はプログラム中のどこででも宣言が可能である．クラス Fibonacci では変数 t が while 文の途中で宣言されている．

Java には整数，浮動小数点，真偽値，文字 (文字列ではない) をサポートするための基本的な型が用意されている．これらをまとめると以下のようになる．特に **char 型が 8bit 整数ではない**こと，**int 型が 32 ビット**であること，**long 型が 64 ビット**であることに注意しておこう．

| 型 | 取りうる値 |
|---|---|
| boolean | true（真）または false（偽） |
| char | 16 ビット Unicode1.1 規格の文字 |
| byte | 8 ビット符号付き整数 |
| int | 32 ビット符号付き整数 |
| long | 64 ビット符号付き整数 |
| float | 32 ビット符号付き浮動小数点 |
| double | 64 ビット符号付き浮動小数点 |

### 1.4.2 式と演算子

### 1.4.3 制御構造

### 1.4.4 クラスとオブジェクト

Java のプログラムは**クラス**から構成される．クラスから，その**インスタンス**が何個でも生成され，これらがお互いに協調してプログラム全体が動作する．

クラスは，オブジェクトを抽象化したものであり，現実世界では，実体がなくても説明できる事物の集合に対応させることができる．より具体的には，商品などの設計図や操作手順を示した説明書と考えると分かりやすい．例えば,「A 社のテレビ」や「B 社のコンピュータ」といったものはクラスに対応する．また，特定の会社の製品を指さない「テレビ」や「コンピュータ」といったものも，テレビやコンピュータが持つべき特徴を，漠然とではあるが指し示していると考えられるので，これらもクラスに対応すると考えられる．

これに対して，インスタンスは実際に製造された製品であり，操作対象となるものである．目の前に「製造番号 A5061 という A 社のテレビ」がある場合には，これは「A 社のテレビ」というクラスのインスタンスである．インスタンスは，それぞれに固有の状態を持ち，説明書に書かれた操作手順で状態を変更させることができる.「製造番号 A5061 という A 社のテレビ」は「製造番号 A1201 という A 社のテレビ」とは同一のクラスのインスタンスであり，同一の操作でチャンネルを変更することができるが，それぞれに異なるインスタンスであり，異なるチャンネルの番組を表示することが可能である．

クラスは Java のプログラムの基本単位で，クラスやそのクラスのインスタンスの状態を表す**フィールド**と，これらの操作方法を示す**メソッド**を**メンバ**として持つ．具体的には，メソッドはフィールドなどを操作する実行文の集まりであり，クラスやインスタンスの状態を変化させる．

### 1.4.5　コメント

Java ではプログラム中に利用できるコメントの種類として次の 3 種類がある．

| 表記 | 意味 |
|---|---|
| `/* ... */` | `/*` と `*/` で挟まれた部分をコメントとして扱う |
| `// ...` | `//` から行末までをコメントとする |
| `/** ... */` | `/**` と `*/` で挟まれた部分をドキュメントコメントとして扱う． |

ドキュメントコメントは直後の宣言を説明するために利用される．先の `Fibonacci` では，`main` メソッドの説明になっている．`javadoc` というプログラムはプログラム中のドキュメントコメントを抜きだし，HTML 文書を生成する．

### 1.4.6　名前付き定数

名前付き定数は C 言語で言うところの `const` に相当する．Java では `final` によって名前付き定数を定義する．

名前付き定数は大きく言って 2 つの理由から利用が推奨される．第一に，定数の名前によってその意味を表現することができるからである．定数名によって，その値がどのような意味で用いられるかを表すことが可能となる．第二の理由は，プログラムの保守性の問題である．定数値を変更する場合，名前付き定数を定義している場所で値を変更するだけでよく，プログラム中におけるそれを利用している全ての場所における値の変更が不要となる．例えば，$\pi$ の値を定義するには以下のように記述する．

```
public class Circle {
    final double pi = 3.141592;
     ...
    length = 2 * pi * r;
     ...
}
```

ただし，main 関数のように static なメソッドから参照する場合には，変数そのものも static 宣言しなくてはならない．もちろん，関数内に記述する場合にはその限りではない．すなわち，static 関数で定数を利用する場合には以下のような二種類の記述が可能である．

```
public class Circle {
    static final double pi = 3.141592;

    static double calcCircumference() {
         ...
        length = 2 * pi * r;
         ...
    }
}
```

```
public class Circle {
    static double calcCircumference() {
        final double pi = 3.141592;
         ...
        length = 2 * pi * r;
         ...
    }
}
```

上の例では $\pi$ の値は，一つのクラスの中で利用することにしているが，現実には $\pi$ の値はただ一つ存在し，これを全てのクラスで共通に利用するように定義することもできる．この場合には，$\pi$ だけを定義するような特別なクラスを用意し，以下のように定義すると良い．

```
public class CircleStuff {
    static final double pi = 3.141592;
}
```

static は，それがインスタンスのメンバではなく，クラスのメンバであることを意味する．クラスのメンバはインスタンスには複製されないので，**プログラム全体でただ一つ**存在することになる．参照する場合にはクラスのメンバとして参照することになるので，$\pi$ の値を参照したければ CircleStuff.pi という

形で参照を行なう．例えば上のクラス Circle を書き換えるならば以下のように書くことができる．

```
public class Circle {
     ...
    length = 2 * CircleStuff.pi * r;
     ...
}
```

# 第2章　制御構造

## 2.1　制御の流れ

Java は**文**を単位としてプログラムを実行する．文は，**セミコロン**（;）で終了する**式**か，**ブロック**である．ただし，全ての式をセミコロンで終了させたとしても，意味のある文にはならない．文として成立するのは以下のような式である．

- 変数の宣言
- 代入式（前置演算子，後置演算子（++や--など）を含む）
- メソッド呼び出し
- インスタンスの生成式（new によってインスタンスを生成する式）

ブロックは 0 個以上の文を**中括弧**（{}）で囲んだものである．
以上をまとめると以下のようになる．

文　:=　式;　または　{文}

これにより文は再帰的に定義され，

```
1. { 式; { 式; } }
2. { 式; 式; 式;}
3. { 式; 式; { 式; { 式; }}}
```

はいずれも文として成立することに注意しよう．

さて，基本的に，Java のプログラムは main メソッドからはじめ，文を順番に実行していく．この章で学ぶ**制御構造**は，この文の実行順序を変更するための特殊な文である．このような文の仲間には if 文，switch 文（case 文），for 文，while 文，do-while 文，break 文，continue 文，return 文がある．また，これらを補助するための**ラベル**という文の修飾子も用意されている．次節以降は，これらについて順に説明していこう．

## 2.2　if 文

```
if( 条件式 )
    文 1
else
    文 2
```

if 文は条件に従って，次に来る文を実行するか否かを選択する．すなわち，まず**条件式**が評価される．条件式は「Boolean を返す式」である．この返値が true であれば「文 1」が，そうでないとき，もし else 節（else 以降の部分）があれば「文 2」が実行される．

「文 1」や「文 2」の部分には任意の文を用いることが可能なので，if 文自身や，この後に出てくる for 文を用いて以下のような文を作成することも可能である．

```
if( count < 0 )
    System.err.println("Unexpected condition!!");
else if( MAX < count )
    System.err.println("Too large.");
else for( int i = 0 ; i < count ; i++ ) {
    doTheJob(aParameter);
}
```

ここで注意したいのは，if 文は「返値が true でなく，かつ実行できる else 節があるのなら，else 節の文を実行する」という点である．簡単に言えば，if 文は「最も近いくて実行可能な else 節を実行する」ということである．それゆえ，以下のメソッドは思い通りには実行されない．

```
public double sumPositive(double[] values) {
    double sum = 0.0;

    if( values.length > 1 ) // (A)
        for( int i = 0 ; i < values.length ; i++ )
            if( values[i] > 0 ) // (B)
                sum += values[i];
    else  // (C)
        sum = values[0];
    return sum;
}
```

プログラマは，(C) の else 節を，(A) の if に対応させたかったのだが，この例では (C) の else 節は，もっとも近い (B) の if に対応して実行されてしまう．

このように不用意に制御構造を用いると，文が思いも寄らない順番で実行されることになる．このような問題が発生しないように，制御構造を用いる場合には，できるかぎりブロックを用いて実行単位を明確にしておいたほうが良いだろう．例えば，上のメソッドは以下のように書くべきである．

```
public double sumPositive(double[] values) {
    double sum = 0.0;

    if( values.length > 1 ) {
        for( int i = 0 ; i < values.length ; i++ ) {
            if( values[i] > 0 ) {
                sum += values[i];
            }
        }
    } else {
        sum = values[0];
    }
    return sum;
}
```

## 2.3 ラベル

```
ラベル:
    文
```

ラベルは文の修飾子である．文の前にラベル（名前）に続けてコロン（:）を付けることで，当該の文に固有の名前を付けることができる．

例えば，以下の文は続くブロックに block というラベルを付ける．このようなラベル付きブロックは，後述する break 文や continue 文と共に用いると便利である．

```
block:
{
    .....
}
```

## 2.4 switch 文

```
switch( 式 ) {
    case 整数 1:
        ...
    case 整数 2:
        ...
    case 整数 3:
        .
        .
        .
    default:
        ...
}
```

switch 文は整数を返す式を評価し，その値によって，直後のブロック内の case で指定されたラベルに実行順序を移す．ブロック内に対応する case ラベルがない場合，ブロック内に default ラベルがあれば，default ラベルに実行順序を移す．case ラベルや default ラベルは switch 文の中で利用される特殊なラベルで，switch 文によって実行順序を移すために用いられる．

ここで注意したいのは，これらのラベルは実行順序を移すための目印として用いられているのであり，実行中に次の case ラベルに遭遇したとしても**実行順序は変わらない**という点である．例えば以下のような文を考えてみよう．

```
switch( aNumber ) {
    case 1:
        System.out.println("One.");
    case 2:
        System.out.println("Two.");
    default:
        System.out.println("neither One nor Two.");
}
```

これは，一見すると数字に応じたメッセージが表示されるように見えるが，実際には，数字が「1」ならば以下の 3 行が，

```
One.
Two.
neither One nor Two.
```

数字が「2」ならば

```
Two.
neither One nor Two.
```

の 2 行が出力されてしまう．

任意の場所で実行をやめるのであれば break 文によってブロックを抜け出す必要がある．

```
switch( aNumber ) {
    case 1:
        System.out.println("One.");
        break;
    case 2:
        System.out.println("Two.");
        break;
    default:
        System.out.println("neither One nor Two.");
}
```

## 2.5 while 文，do-while 文

```
while( 条件式 )
    文
```

while 文は，ある条件下での文の繰り返し実行のために用いられる．while 文が実行されると，まず条件式が評価され，その返値が true の場合は続く文が実行され，また条件式が評価される．文の実行は条件式の返値が false になるまで繰り返される．

while 文では，最初に条件式が評価されるため，文が全く実行されない可能性もある．すなわち，最初に条件式を評価する際に，その返値が false であれば，文は一度も実行されないまま，while 文は終了する．

文を一度は実行したいという場合には，do-while 文を用いる．do-while 文は以下のような書式である．

```
do
    文
while( 条件式 );
```

do-while 文では，条件式は文が実行された後に評価される．条件式の返値が trule の間，文が繰り返し実行されるのは while 文と同様である．

## 2.6 for 文

```
for( 初期化式 ; 条件式 ; 繰り返し式 )
    文
```

for 文は，ある条件下での文の繰り返し実行のために用いられる．while 文と異なるのは，実行の前に初期化式が評価される点と，文の実行が終わると必ず繰り返し式が評価される点である．

for 文は以下のように書き換えることができる．これは，continue 文に遭遇した場合の処理を除き，for

文と同等の動作をする．

```
{
    初期化式;
    while( 条件式 )
    {
        文
        繰り返し式;
    }
}
```

初期化式と繰り返し式はコンマ（,）で区切られた式のリストであっても構わない．コンマで区切られた式は，左から右に評価される．これを用いることにより，初期化式および繰り返し式において，複数の式を実行することができる．例えば，配列を最後から調べる文は次のように書くことができる．

```
for( i = 0, j = arr.length - 1 ; i < arr.length ; i++, j-- ) {
    if( arr[j] == ... )
        ....
}
```

for 文の式は全てなくても良い．初期化式や繰り返し式がない場合は，その部分は無視される．条件式がない場合は，その値は常に true として扱われる．これを利用して無限ループ（永遠に文の実行を繰り返す文）は以下のように記述することができる．

```
for(;;)
    文
```

## 2.7　break 文

```
break ラベル;
```

break 文はラベルで指定された switch 文，for 文，while 文，do-while 文を終了させるために利用される．ラベルは省略することができ，省略した場合には**最も内側**の switch 文，for 文，while 文，do-while 文が終了させられる．

例えば，以下の文では配列の中を探索し，引数である変数に一致する要素を見つけた時点で探索を終了する．

```
public boolean searchValue( int c ) {
    int x, y;
    boolean found = false;

    search:
        for( x = 0 ; x < Matrix.length ; x++ ) {
            for( y = 0 ; y < Matrix[x].length ; y++ ) {
                if( Matrix[x][y] == c ) {
                    found = true;
                    break search;
                }
            }
        }

    // 作業する
    ....
}
```

このようにラベルを利用した break 文により，多重ループを抜け出すことが可能になる．C 言語では，この目的のためにしばしば goto 文が利用されてきたが，goto 文は，任意の場所への分岐に利用され，制御の流れを見にくくする．ラベルを用いた break 文や後ほど説明する continue 文は，任意の深さのループを抜け出す目的のためだけに利用されるため，制御の流れは理解しやすい．

## 2.8 continue 文

```
continue ラベル;
```

continue 文はラベルで指定された for 文，while 文，do-while 文における，文の実行を終了する．ラベルは省略することができ，省略した場合には**最も内側**の for 文，while 文，do-while 文における文の実行が終了させられる．

break 文と異なり，for 文，while 文，do-while 文そのものを終了させるわけでは**ない**ことに注意しよう．すなわち，for 文で continue 文を用いた場合には，ブロック内の continue 文以降の文の実行は中止され，即座に繰り返し式および条件式の評価が行われる．while 文，do-while 文で用いた場合には，ブロック内の continue 文以降の文の実行は中止され，即座に条件式の評価が行われる．

意味のある例ではないが，以下のプログラムを考えてみる．このプログラムでは，配列の中を探索し，引数である変数に一致する要素を見つけた時点で (B) の for 文を終了し，(A) の for 文の繰り返し式および条件式が評価される．このとき (C) の文は実行されないことに注意しよう．そして，再び (A) の for 文の実行文，すなわち (B) の for 文が実行される．

```
public boolean searchValue( int c ) {
    int x, y;
    boolean found = false;

    search:
        for( x = 0 ; x < Matrix.length ; x++ ) { // (A)
            for( y = 0 ; y < Matrix[x].length ; y++ ) { // (B)
                if( Matrix[x][y] == c ) {
                    continue search;
                    found = true; // (C)
                }
            }
        }

    // 作業する
    ....
}
```

## 2.9　return 文

```
return 返値;
```

return 文はメソッドの実行を終了し，制御を呼び出し側に移す．その際，返値が指定されていれば，返値を返す．メソッドが値を返さない場合は，return 文の返値を省略すればよい．