情報科学 2007　久野クラス　＃12

久野　靖*

2008.1.18

はじめに

今回は「さまざまなプログラミング言語」の話題ですが、その一環として Java 言語を取り上げ実際にプログラムを書いてみて頂きます。Ruby と違うところが色々あってとまどうかも知れませんが、変数と代入、順次実行、制御構造などの概念は一緒ですのでご心配なく。

1　さまざまなプログラミング言語

1.1　機械語、アセンブリ言語、高水準言語

皆様はここまで散々コンピュータを使って来たが、またさらに Ruby でプログラムを書いて来たが、コンピュータの内部でプログラムが実行されるしくみについてはあまり立ち入って来なかった。そこを少し見てみよう (図 1)。

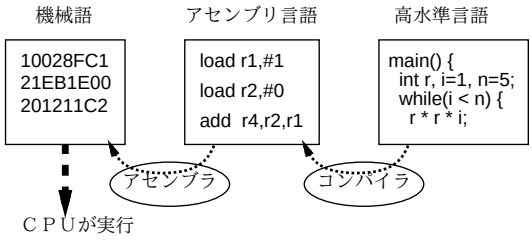

図 1: 機械語、アセンブリ言語、高水準言語

コンピュータ内部の **CPU**(central processing unit、命令を実行する部分) が直接実行できる命令はもちろん、「メモリ上に格納された 0/1 の並び」である。このようなメモリ上の語の列としてのプログラムを**機械語**のプログラムと呼ぶ。もちろん、全部数字なので、見ても何がなんだか分からない。初期のコンピュータ (1945 年ころ〜) ではプログラムを機械語で直接開発していたこともあるが、これは非常に大変である。

そこで、それぞれの命令、変数、レジスタ (CPU 内で値を保持する場所) などを名前で表すような記法が使われるようになった。これを**アセンブリ言語**と呼ぶ。アセンブリ言語のプログラムは**アセンブラ**と呼ばれる言語処理系で機械語に変換して、それを CPU が実行する。しかしアセンブリ言語でも、CPU に備わっている個々の命令 (加算命令、ロード命令、ジャンプ命令) などを連ねてプログラムを書くのは非常に大変である。また、機械語やアセンブリ言語では、CPU の種類が違うと命令も違うので、機種を入れ替えるとプログラムが作り直しになるという問題もあった。このような、CPU の種類に依存するプログラミング言語を**低水準言語**と呼ぶ。

このためやがて (1950 年代〜)、もっと一般的な人間に分かりやすい、特定の機種に依存しない書き方 (言語) でプログラムを書くようになった。このようなプログラミング言語のことを**高水準言語**と呼ぶ。高水準言語は、**コンパイラ**と呼ばれるプログラムにより (場合によってはアセンブリ言語を経由して) 機械語に変換して実行するか、または**インタプリタ**と呼ばれるプログラムで直接解釈実行する (これは既におもちゃ言語でやりましたね)。

ちょっとだけ具体例を見てみよう。次のコードは C 言語で階乗の計算を記述したもの。

*筑波大学大学院経営システム科学専攻

```
#include <stdio.h>
main() {
  int n, i = 1, r = 1;
  printf("? "); scanf("%d", &n);
  while(i <= n) {
    r = r * i; i = i + 1;
  }
  printf("%d\n", r);
}
```

まあ入出力のところはさておき、階乗の計算を行うループのところは Ruby と同じですから。これを iMac(PPC) のアセンブリ言語に直すには…

```
% gcc -S -O4 test.c  ← test.c に上のプログラム
```

これでファイル test.s にアセンブリ言語プログラムができる。その中核部分の抜粋を示そう。

```
LC0:
      .ascii "? \0"    ← printf のパラメタ文字列
      .align 2
LC1:
      .ascii "%d\0"    ← scanf のパラメタ文字列
      .align 2
LC2:
      .ascii "%d\12\0"  ← printf のパラメタ文字列
(中略)
      addis r3,r31,ha16(LC0-"L00000000001$pb")  ← LC0 参照
      la r3,lo16(LC0-"L00000000001$pb")(r3)
      bl L_printf$LDBLStub$stub   ← printf を呼ぶ
      addis r3,r31,ha16(LC1-"L00000000001$pb")  ← LC1 参照
      addi r4,r1,56               ←変数 n の番地参照
      la r3,lo16(LC1-"L00000000001$pb")(r3)
      bl L_scanf$LDBLStub$stub    ← scanf を呼ぶ
      lwz r0,56(r1)               ← r0 が n
      li r4,1                     ← r4 が r
      cmpwi cr7,r0,0              ← n が 0 なら
      ble cr7,L5                  ← L5 へ飛ぶ
      li r2,1                     ← r2 が i
      .p2align 4,,15
L4:
      mullw r4,r4,r2              ← r = r * i
      addi r2,r2,1                ← i = i + 1
      cmpw cr7,r2,r0              ← r と n を比較
      ble cr7,L4                  ← <= なら L4 へ飛ぶ
L5:
      addis r3,r31,ha16(LC2-"L00000000001$pb")  ← LC2 参照
      la r3,lo16(LC2-"L00000000001$pb")(r3)
      bl L_printf$LDBLStub$stub   ← printf を呼ぶ
```

このように確かに高水準言語の動作に対応する命令が作られていることがわかる (でも大変でしょう?)。これを機械語にして実行してみる。

```
% gcc -O4 test.c
% a.out
? 5
120
%
```

まあこれだけですけど。

1.2 プログラミング言語の歴史

最初に作られた高水準言語は FORTRAN と呼ばれる数値計算向け言語であるが、その後、用途によって言語に求められる機能が違うことが認識され、さまざまな言語が作られるようになった。年代別に簡単に見てみよう。

1.2.1 1950 年代

初期のプログラミング言語の時代。さまざまな専用目的の言語が作られた。

- FORTRAN — 上述の通り、数値計算むけ言語。ただしコンピュータといえば理工系で理工系は数値計算を必要とするので、理工系のコンピュータ入門教育でも広く教えられた。今でも存在しているが、数値計算をやる人だけが使うのでマイナー化。
- COBOL — 事務処理言語。計算を 10 進表現でやるので億でも兆でも京でも正確に扱える。しかし現在では汎用のオブジェクト指向言語に見られる多様な機能の取り込みが遅れたことからマイナー化。日本では事務処理業界に多数の COBOL プログラマがいてそれをオブジェクト指向に転換させるのが頭の痛い問題となっているい。
- Lisp — 記号処理言語。人工知能研究に広く使われた (今でも使われている)。たとえば Emacs の中は大部分 Lisp で書かれている。

このほか、数値計算やアルゴリズム記述に使われた Algol、システム記述言語の BCPL と Bliss など多くの言語が存在していた。

1.2.2 1960〜70 年代

新しい言語メカニズム探求の時代。オブジェクト指向、論理型、関数型など言語の基盤となる枠組みがいろいろ模索されそれに応じた言語が作られた。

- Simula、Smalltalk — オブジェクト指向の黎明期の言語。
- BASIC、Pascal、LOGO — 教育むけに分かりやすい/使いやすい言語を作ろうとする試みの成果。
- C — システム記述言語であり、Unix の記述に使われたことから Unix の普及とともに広がって現在に至る。
- Prolog — 論理型言語の最初のものであり、今でも使われている。
- ML — 関数型言語として実用に使われた最初のものであり、今でも使われている。

1.2.3 1980〜

それまでに考案された枠組み、とくにオブジェクト指向を土台としてパッケージした「使いやすい」言語の登場と普及。とくに、コンピュータの能力が向上したことから、インタプリタ実行による柔軟性を持った**スクリプト言語**が増えた。

- C++、Java、C# — オブジェクト指向の汎用言語
- Perl、Python、Ruby、JavaScript — スクリプト言語
- Haskell、OCaml — 関数型の発展

これらは超メジャーなものだけで、この他にも多数の言語が存在している。これは、言語処理系の作り方が分かって来て誰でも簡単に作れるようになったから、ということの影響が大きい。

1.3 プログラミング言語の種別

プログラミング言語を分類するとき、決まった分け方があるわけではなく、さまざまな分類の軸が存在している。どのような分類軸があるかについて簡単に見ていく。

1.3.1 命令型と宣言型

命令型 (ないし**手続き型**) 言語とは、コンピュータが行うべき動作を順に命令として記述するようなもの。「変数+代入+順次実行+制御構造」。古くから現在に至るまで、プログラミング業界では主流。

宣言型言語とは、ものごとの性質や関係を記述し、その要件を満たすような解を処理系側で探す。つまり実行順序は手続き型の「人間が順序を決める」のと対照的に処理系が定める。代表的な 2 つの方式。

- 関数型 — 関数を記述し、その適用を通じて計算を行う。ML、Haskell、OCaml など。
- 論理型 — 変数を含んだ述語の群を記述し、その述語群を満たすような変数の値を求めることで計算結果が求まる。

1.3.2 コンパイラとインタプリタ

コンパイラにより機械語に翻訳する方式は、実行速度が高くしやすいため、実用的なソフトウェア開発のための言語で中心的に使われて来た。Fortran、C、C++など。ただしコンパイラは CPU の機種が違うと (出力する機械語が異なるため) 作り直す必要がある (図 2 左)。

インタプリタにより解釈実行する方式は、実行時の柔軟性が高めやすいので、教育用、記号処理、文字列処理など特殊目的の言語で広く使われて来た。BASIC、LOGO、Lisp、SNOBOL など。しかし最近になって、CPU の実行速度が向上したので、実用的な言語でも採用されるようになった。スクリプト言語の大半がそう。インタプリタは CPU ごとに別のものではあるが、高水準言語で書いてあるので単に各 CPU 用にコンパイルし直せば済むことが多い (図 2 中)。

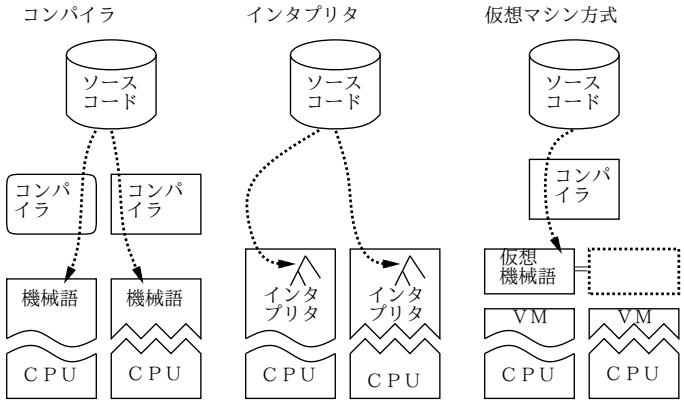

図 2: コンパイラとインタプリタ

なお、1 つの言語についてコンパイラもインタプリタも存在するという場合もある。だから言語の特性というより、処理系の作り方と考えるべき。とはいえ、「この言語は主にこちら」という傾向はある。

さらに、両者を組み合わせた**仮想マシン**方式と呼ばれるものもある (図 2 右)。つまり、コンパイルした結果が特定 CPU の機械語ではなく**仮想機械語**であり、それを仮想マシン (VM、virtural machine) と呼ばれる一種のインタプリタで実行するという方式。この方式はコンパイラを CPU に合わせて書き直さなくて済むわりに速度が向上させやすいという利点がある。近年では仮想マシンの実装技術が向上したため機械語を出力するコンパイラにさほど遜色ない速度で実行できることも多い。Java と C#が代表的。Ruby も 1.9.x から内部実行が仮想マシン方式になった。

1.3.3 オブジェクト指向言語

既に学んだ通り、「もの」を中心としてプログラムを構成するという考え方がオブジェクト指向であり、それをサポートする言語機構を持った言語がオブジェクト指向言語。クラス、メソッド、インスタンス変数などの概念を持つ (このほか、クラスを持たない方式もある)。

多様なライブラリ部品を用意する場合、それらをオブジェクト (ないしクラス) として用意するのが自然なので、今日の言語は多くがオブジェクト指向機能を提供するようになっている。

1.3.4 スクリプト言語

スクリプトとは「台本」つまり「やりたいことを書いたもの」という意味。もともとは Unix のコマンド (シェルコマンド) をファイルに入れておいて、そのファイルを実行すると書いてあるコマンドが順番に実行される、という仕組み (シェルスクリプト) でプログラムが書けるという話から始まった。

その後、Larry Wall による Perl 言語が、さまざまなシステム機能やファイル処理が自在に呼び出せ、またパターンマッチなどの機能が充実していてやりたいことが簡潔に書けることから、広く普及し、スクリプト言語という概念が確立した。

Perl 自体はオブジェクト指向の実現において弱点があり、このためこれを改良した新世代のスクリプト言語として Python、Ruby が普及している。このほか、ブラウザに搭載されるスクリプト言語として JavaScript、Web サーバのサーバ側プログラミングに使われるスクリプト言語として PHP など、各種用途ごとにそれむけの言語が作られるという面もある (Ruby でサーバ側プログラミングをやることも多い)。

1.4 プログラミング言語と型

プログラミング言語において、**型** (type) とは、データの種別のこと。たとえば、同じ「+」であっても、CPU の命令では整数の足し算と浮動小数点の足し算は違う命令。また、整数は 32 ビット、浮動小数点は 64 ビットで扱うことが多いので確保する領域も違う→型に応じて扱いを変える必要。また、オブジェクト指向言語では、「オブジェクトの種類=所属するクラス」なので、クラスと型が対応していると考えることができる。

プログラミング言語における型の扱い型として、次の 2 種類がある。

- 強い型 — 変数や式やメソッドの引数・返値などにおいて、すべて型が決まっているような言語
- 弱い型 — 変数や式やメソッドの引数・返値などにおいて、型が決まっていなくて、実行時にどのような型でも受け入れられる言語

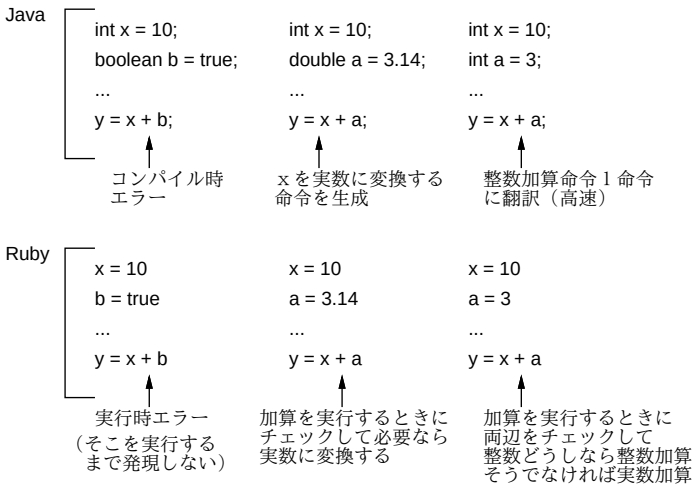

図 3: 強い型と弱い型

弱い型のことを「型のない言語」と呼ぶこともあるが、実際には実行時にデータ本体に「これはこういう型」という情報が付属していて、それに基づいて適切な演算や処理を行うので、型がないわけではない。(アセンブラのように本当に型がなくて、どの値はどのようなデータかをプログラマが全部覚えていて間違いなく使うような場合はこれに相当しないが、そのような高水準言語は今はない。)

弱い型の言語では、どの変数にどんな種類のデータを入れてもよいのは便利ではあるが、その代わりに実行時に種類の間違ったものを入れたりするとよく分からないエラーになったりして苦労する。さらに困るのは、複雑なプログラムになると、全部の箇所が実行されるとは限らないので、実際に間違いのある箇所が実行されるまでエラーが分からないことである。さらに、実行時に型情報をチェックするため実行速度が遅くなる。(図 3 下)

強い型の言語では、すべての変数やメソッドの引数・返値について「これはこのような型」ということを予め書く必要がある。これは慣れないと煩わしい。その一方で、型が間違っているという状況はコンパイル時に検出されるので、実行している途中で型のエラーが出ることはなくなる (オブジェクト指向の部分で多少問題が残ることはあるが)。また、弱い型の言語と異なり、実行時に型の情報を保持していてチェックしつつ実行する必要がなくなるため、実行が高速にしやすい。(図 3 上)

このため、大規模なソフトウェアを整合性を持って組み立てるなどの用途には強い型の言語 (C++、Java) が主に使われる。もっとも、強い型が嫌いで大規模ソフトでも弱い型の言語でばりばり書くのがいい、と主張する人も少なからず存在はしている。

2 Java 言語入門

2.1 Java のプログラムを動かす

ここまででもうお話には飽きたと思うので、さっさと Java のプログラムを動かすことにしよう。いきなりプログラムを示す。

```
import java.util.*;        ←おまじない、ライブラリの取り込み

public class Sample21 {    ←クラス開始
  public static void main(String[] args) {  ← main 開始
    Scanner sc = new Scanner(System.in);     ←入力用オブジェクト用意
    System.out.print("w> ");                 ←プロンプト
    double w = sc.nextDouble();              ← double を読む
    System.out.print("h> ");                 ←プロンプト
    double h = sc.nextDouble();              ← double を読む
    System.out.println("area = " + (w*h)/2.0); ←表示
  }                          ← main 終了
}                            ←クラス終了
```

Java では irb のようなものはないので、すべて入出力機能を使って読み書きする必要がある。そして、変数はすべて型を指定してまず定義する必要がある (通常はその後すぐ初期値を代入する)。では、これをさっさと打ち込んで動かして頂くための最低限の説明。

- Java のプログラムはすべてクラスが単位となる。そして、ファイル名と最外側の (public な) クラス名を一致させないといけない (窮屈!)。たとえば「public class Sample21 ...」なプログラムであれば、か・な・ら・ず「Sample21.java」というファイルに入れなければ怒られてしまう!!!
- プログラムの実行は必ずクラス内の「public static void main(String[] args)」なメソッドから開始される。この意味は「クラスの外部から呼べる、(インスタンスメソッドではなく) クラスに直接付随するメソッドで、名前は main で、パラメタ args を受け取る」ということ。
- プログラムを作ったらまずコンパイルする。

```
javac Sample21.java   ←コンパイルにはファイル名を指定
```

コンパイル方式だと、ここで色々エラーが出ることがあると思う。そのときはプログラムを直して再度コンパイル。「何も表示なしに終わった時」は OK。

- エラーなしにコンパイルできたら実行に進む。

```
java Sample21          ←実行時にはクラス名を指定
```

やってみよう。

```
% javac Sample21.java
% java Sample21
w> 5
h> 7
area = 17.5
%
```

さて最後にプログラム内部の説明だが、Scanner オブジェクトは入力から整数 nextInt()、実数 nextDouble()、文字列 nextLine() など色々読み取るのに便利な機能を提供する。標準入力オブジェクト (System.in) を与えて Scanner を作るので、キーボードからの読み込み用となる。System.out.println() は文字列出力。文字列定数は必ず"..."で囲むこと。そして Java では「+」演算は左右いずれかが文字列のときは他方も文字列に変換した上で文字列連結を行う (わりと便利)。

演習 1 例題の三角形の面積計算プログラムをそのまま打ち込み、実行させてみよ。数字でないものを与えたりするとどうなるかも試せ。

演習 2 三角形の面積計算で、割る数の指定を「2.0」でなくただの「2」にした場合に何か違いがあるか試せ。

演習 3 次のような計算をするプログラムを作って動かせ。なお、平方根は Math.sqrt(x) で求められます。

a. 2 つの実数を与え、その和を返す (ついでに、差、商、積も)。何か気づいたことがあれば述べよ。
b. 「%」という演算子は剰余を求める演算である。上と同様に剰余もやってみよ。何か気づいたことがあれば述べよ。
c. 円錐の底面の半径と高さを与え、体積を返す。
d. 円錐の底面の半径と高さを与え、表面積を返す。
e. 実数 x を与え、x を 10 で割った結果を返す。また、同様だが x の 0.1 倍を返す。これらを比較し、何か気がついたことがあれば述べよ。
f. 実数 x を与え、x の平方根を出力する。さまざまな値について計算し、何か気がついたことがあれば述べよ。
g. その他、自分が面白いと思う計算を行うプログラムを作って動かせ。

2.2 Java 言語の由来と特徴

では少しだけお話を。Java 言語は 1990 年代に Sun Microsystems 社が、ネットワーク経由でコンパイル済みの仮想機械コードをさまざまなマシンに転送して実行させることができるようなシステムのための言語として開発した。その直後、WWW(Web) が爆発的に普及したとき、当時は HTML による「止まった」ページしか作れなかったので (まだ Flash も JavaScript もなかった)、Sun によってページ内にアプレットを入れる技術が公開されたとき、世の中はこれを熱狂的に歓迎して「Java ブーム」が生まれ、あっという間に Java をメジャーなプログラミング言語にしてしまった。まあ、Java 自体もそれが土台とした C++よりも「安全で」「簡潔で」「スマートな」設計になっていたということもある。

今日ではアプレットについては Flash などに駆逐されてしまったが、Java は上記の特性を持つ「汎用のオブジェクト指向言語で」「仮想機械方式によりさまざまなプラットフォーム上で動作する」という特徴により一定の支持を得て広く使われている。

2.3 Java 言語のデータモデル

Java は構文は C や C++に似せてあるが、そのデータモデルはどちらとも違っている。まず、Java ではデータは次の 2 種類に画然と分かれている。

- 値 — 予め定義された値の型として整数 (int、byte、short、long)、実数 (double、float)、文字 (char)、論理値 (boolean) があり、これらはオブジェクトではない。加減乗除、大小比較などの演算はすべて値に対してだけ行える。
- オブジェクト — 配列、文字列、その他のあらゆるデータはオブジェクトであり、クラスによって定義されている。オブジェクトに対しては演算は行えず、すべてメソッド呼び出しによって操作する (例外として、==/!=により等しいオブジェクトか否かを調べることができ、また文字列に限って+により連結できるが、これらは本当に例外中の例外)。

このように画然と分かれていると不便なこともある。たとえば整数をオブジェクトとして渡そうとするとそのままではできない。そこで、int に対して Integer、double に対して Double のような対応関係にあるクラス (**包囲クラス**) が用意されていて、これらの間での変換は自動的に行ってくれる。このあたりはおいおい。

2.4 Java 言語の制御構造

Java の制御構造は C や C++から来た伝統的なものだが、Ruby が革新的なのでとまどうこともあると思う。さっさと説明しておく。

```
while(条件) {  ← while 文
  文…
}

if(条件) {     ← then だけの if 文
  文…
}

if(条件) {     ← 2 方向の if 文
  文…
} else {
  文…
}

if(条件) {     ← if-else if の連鎖 (Ruby の elsif)
  文…
} else if(条件) {
  文…
} else if(条件) {
  文…
} else {
  文…
}

for(初期化; 条件; ステップ) {
  文…
}
```

while、if、if-else if の連鎖は Ruby と書き方が違うだけ。最後の for 文というのは Ruby の for 文 (説明しなかったけど) と全く違うので説明しておく。Java の for 文はまず「初期化」を実行し、それから「条件」が成り立つ間繰り返し while 文と同様に反復するが、1 回本体を実行するごとに「ステップ」を実行する。例を見てもらった方が早いが、0 から 100 まで出力するというのは次のようになる。

```
for(int i = 0; i <= 100; ++i) {  ←「++i」は「i を 1 増やす」
  System.out.println(i);
}
```

演習 4 枝分かれと繰り返しを用いて、次の動作をする Java プログラムを作成せよ。なお、整数を読んで変数に入れるときは先の例を少し直して「`int i = sc.nextInt();`」等とすればよいでしょう。

a. 2 つの異なる整数 a、b を読み込み、より大きい方を表示。
b. 3 つの異なる整数 a、b、c を読み込み、最も大きいものを表示。
c. 整数を 1 つ読み込み、正なら positive、負なら negative、零なら zero と表示。
d. 2 つの正の整数を読み込み、最小公倍数を表示。
e. 正の整数 n を読み込み、n の階乗を表示。
f. 正の整数 n を読み込み、n 以下の素数を列挙。

なお 1 つ注意しておきますが、`{...}`の「中で」宣言した変数はその範囲内でしか使えないので、外まで持ち出したい変数は「外で」(もっと前で) 定義してください。

2.5 Java 言語の配列

Java にも Ruby と同様、配列はあるが、次の点が大きく異なっている。

- 強い型の言語であるので、「型名 `[]`」という書き方で「何型の配列か」を必ず指定する。たとえば「`int[] a;`」ならば変数 `a` は整数の配列。「`double[][] m`」ならば `m` は実数の配列の配列。
- 配列の大きさは最初に作ったときに決まり、後から変更することができない (すごく不便だが効率はよい)。配列 `a` に対して `a.length` で大きさ (要素数) が指定できるのは同じ。

だから実は、メソッド `main` の引数は文字列の配列なのだった。これは次のように `java` コマンドで実行開始時に指定できる。

```
java クラス名 引数 0 引数 1 … 引数 N
```

プログラム中で実際に配列を作る時、大きさを指定して作る方法と中身を指定して作る方法の 2 種類があるのは Ruby と同様 (前者の場合初期値は数値型なら 0、論理型なら `false`、その他のオブジェクト型なら `nil` に固定)。

```
int[] a = new int[100];  ← 100 要素の整数の配列
double[][] m = new double[]{{1.0, 0.0}, {0.0, 1.0}}; ← 2x2 単位行列
```

配列の大きさ固定が不便なので、大きさが変化する配列を実現するオブジェクトが別にあり、そちらを使うことも多い (今回は説明しない)。

とりあえず簡単な例として、ファイルの各行を配列に読み込んで「上下逆転」して打ち出すプログラムを示そう。いくつかおまじないの追加があるが、詳しく説明している余裕はないのでご勘弁を。

とりあえず、I/O など例外を投げるメソッドを呼ぶ可能性がある場合は※のように `main` に「`throws Exception`」を指定してください。

```
import java.util.*;
import java.io.*;     ← I/O パッケージも取り込む
```

```
public class Sample22 {                    ↓ I/O などを行う場合※
  public static void main(String[] args) throws Exception {
    Scanner sc = new Scanner(new FileReader(args[0]));  ←ファイルを開く
    String[] lines = new String[10000];   ←とりあえず 1 万行までとする
    int count = 0;
    while(sc.hasNextLine()) {   ←まだ行があれば
      lines[count] = sc.nextLine(); ++count;  ←追加
    }
    for(int i = count-1; i >= 0; --i) {  ←逆順に
      System.out.println(lines[i]);      ←出力
    }
  }
}
```

FileReader オブジェクトはファイル名を指定して生成し、そのファイルから読み込む機能を提供する。行単位で読みたいのでこれを渡して Scanner を作っている。各行を入れておく配列は大きさ固定なのでとりあえず大きめに作り、現在いくつまで使っているかを別の変数で持つようにしている (不便!)。出力時に逆順に数えるのとかは for 文の場合、自前で適切な初期値、条件、ステップ (1 減らす操作) を書く必要があるので注意。

演習 5 Java で次のようなプログラムを作ってみよ。

a. 好みの方法で素数列挙を作り、同一アルゴリズムの Ruby 版と所要時間を比較してみる。
b. 好みの整列アルゴリズムを実現し、同一アルゴリズムの Ruby 版と所要時間を比較してみる。
c. その他好みのアルゴリズムを実現し、同一アルゴリズムの Ruby 版と所要時間を比較してみる。

演習 6 Java で好きなプログラムを作ってみよ。

上記演習を行う際に参考になる情報をいくつか。

- 乱数を生成したいときは `Math.random()` でできます (0 以上 1 未満の実数の一様乱数が生成される)。
- 整数を実数に変換するのは自動でできる。逆に実数を整数に変換するときは「`(int)(式)`」という構文を使う。たとえば 0〜9999 の整数一様乱数が必要なら「`(int)(Math.random()*10000)`」とする。
- `main` だけでなく複数のメソッドに分けたいこともあると思う。その場合は次のような感じで。

```
public static int sub(int x, int[] a) {
  ...          ↑返値は int   ↑パラメタも全て型を指定する
}
public static void sub2(double x) {
  ...           ↑返値を返さないメソッドの場合 void と指定
}
```

返値を返すのは `return 式;` で、これは Ruby と同じ。返値を `void` としたメソッドでは式は指定しない (できない)。

そのほか、Java に関する入門としてはいぱーワークブックのプログラミングの項がありますので (下記 26 番)、よかったらどうぞ。

`http://hwb.ecc.u-tokyo.ac.jp/current/`

英語でよければ Java Tutorials も定評あります。

`http://java.sun.com/docs/books/tutorial/`

あとは本ですが、これは色々あるので… (久野の書いた本もありますが Java のバージョンが変わっているのでちょっとね。) まあ、自分で本屋さんで眺めてみてください。

2.6 Javaの文法

Javaでだいぶ色々書いてもらったが、コンパイルエラーが沢山出ますね? ここでJavaの文法を拡張BNFで示しておこう (簡略化してあります):

プログラム = クラス定義 …
クラス定義 = 修飾… `class` クラス名 { 定義… }
修飾 = `public` | `private` | `static` | …(以下略)
定義 = クラス定義 | 変数定義 | メソッド定義 | コンストラクタ
変数定義 = 型指定 (変数名 [`=` 式]),.. ;
型指定 = 型名 | 型指定 `[]`
メソッド定義 = 修飾… 型指定 メソッド名 (引数,..) ブロック
コンストラクタ = 修飾… クラス名 (引数,..) ブロック
引数 = 型指定 引数名
文 = ; | 式 ; | ブロック | `if` 文 | `while` 文 | `for` 文 | …(以下略)
ブロック = { (変数定義 | 文)… }
`if` 文 = `if` (式) 文 [`else` 文]
`while` 文 = `while` (式) 文
`for` 文 = `for` (式 ; 式 ; 式) 文 | `for` (変数定義 ; 式 ; 式) 文
式 = 定数 | 配列定数 | 呼び出し | 配列 | 演算 | (式)
呼び出し = (式 | クラス名) . メソッド名 (式,..)
配列 = 式 `[` 式 `]`
演算 = 式 演算子 式 | 式 演算子 | 演算子 式

すべてのJavaプログラムはこの構文規則に従っていなければならない。構文エラーの場合は、この書き方に合っていない箇所がある、ということになる。拡張BNFの書き方について少し説明しておく。

- $\alpha = \beta$ — 左の内容を右辺で定義。
- $\alpha \mid \beta$ — α または β。
- [α] — α があってもなくてもよい。
- α… — α の繰り返し。
- α ,.. — α をカンマで区切った並び。

A 本日の課題 12A

演習問題から1つ選び、動かしたプログラムを含む小レポートを今日中に久野までメールで送ってください。

1. Subject: は「Report 12A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 動かしたプログラムどれか1つのソース。
4. 以下のアンケートの回答。

   Q1. 「さまざまなプログラミング言語」の内容についてどれくらい知っていましたか?
   Q2. Javaを動かして見て、Rubyとの違いについてどうでしたか? Rubyで学んだことがらはどのくらいそのまま使える/使えないと感じましたか?
   Q3. その他、感想、要望等どうぞ。

次回までの課題はありません。なお、Javaを自分のPCで動かしたい人は `http://java.sun.com/javase/downloads/` から「JDK 6 Update N」(N はいくつでも最新のでよい) を (自分の使ってるシステム用のを) 持って来て入れるといいでしょう。