文教大学女子短期大学部研究紀要52集，1－16，2009

Javaプログラミング教育に関する一考察

太　田　信　宏

Study on Java Programming Education

Nobuhiro Ohta

Abstract

The purpose of this study is to consider the content and key points for inclusion in a Java programming course for beginners. The Java programming language has a variety of functions and has the largest application field of all such languages, containing many themes that are appropriate for any such programming course. The multifunctional and wide-ranging functions of Java, however, may actually act as a barrier to study for beginners. The core content of a programming class for beginners should contain elements such as assembling algorithms and basic programming techniques. For any Java programming course it is important to clarify two points: “What is taught?” and “What is not taught?”

This study considers the following subjects.

(1) Content of programming course for beginners

(2) Execution environment for Java programming

(3) Grammar and coding rules

(4) Algorithms

(5) Data expression and information technology, which are assumed knowledge for programming

(6) Java’s specific expression methods

１．はじめに

インターネットの進化とともにウェブを利用したプログラミング環境が多く出現している。従来型のプログラミング教育ではC、COBOL、BASIC、Fortranなどの手続き型言語が中心であったが、最近ではオブジェクトプログラミングが可能なJavaやVisual Basicなどがよく利用されるようになっている。実行環境の面では、汎用機やパソコンを「閉じた世界」で利用する環境から、ウェブを活用した実行環境へと変化している。スクリプト系言語とHTMLを組み合わせることによって、比較的簡単にプログラムを体験することも可能になっている。その多くは無償で利用できるものである。こうした状況から、最近では一般のユーザがプログラムに触れる機会も多くなっているといえる。

筆者は文教大学情報学部において、初心者を対象としたプログラミング授業を担当して

いる。言語としては2003年度～2006年度まではC言語、2007年度からはJavaを使用している。言語の分類でいえば、C言語は手続き型言語であり、Javaはオブジェクト型言語である。言語仕様は異なるが、初級プログラミングのレベルにおいては、両者に文法的な共通点が多く見られる。また一方では、ごく簡単なプログラムであっても言語仕様の違いから、指導上注意すべき点も少なくない。初級者がJavaのプログラムを学習する上でどのような点に疑問を抱くのか、本論では、一部C言語との比較を試みながら指導上のポイントなどを考察していくこととする。

2．プログラムの実行環境

授業において活用したコンパイラおよびプログラミングの開発環境は表2-1のとおりである。開発環境のCPad[1)]はC/C++、Java、Fortran、Pascel、C#で利用可能なGUIベースの開発支援ソフトウェアである。JavaもCも、コンパイラを含め必要なソフトウェアはすべて無償で入手可能なものである。したがって受講者自身が、自分の所有するPCで学校と同等の環境を実現することが可能となっている。

表2-1　本学におけるプログラム実行環境（2008年度）

	Java 言語	C 言語
コンパイラ	Java2 SDK　1.5.0	Borland C++ Compiler　5.5
プログラミング開発環境	CPad for Java2 SDK	CPad for Borland C++Compiler

3．初級プログラミングにおける指導内容

筆者が担当した授業は初心者向けのものであり、プログラミングの基礎とアルゴリズムの基本技法を習得させることが主なねらいである。授業は1セメスター週2コマで行う。2コマは連続して開講され、講義と実習がセットになっている。プログラミング教育には、対象者に応じていろいろなレベル設定が可能であるが、初級レベルの内容を挙げると、おおよそ表3-1のようになる。筆者の授業も基本的にこの内容にしたがって計画・実施している。この表にある通り、JavaとCでは指導する項目に共通点が多く見られる。ただしグラフィカルな処理に関しては若干事情が異なる。Javaではアプレットやframeを用いてウィンドウ型アプリケーションやウェブ画面への図形描画を比較的簡単に実現することができる。これに対し、C言語ではグラフィックス用のライブラリが別途必要になるためJavaに比べると敷居が高い。表3-1の項番11はJavaアプレットを利用したグラフィックスの内容である。C言語の授業ではこの項目は範囲外とし、代わりにより実践的なアルゴリズム（ソート、探索、データ構造等）についての指導を行った。なお表3-1はシラバスそのもの

表3-1　初級プログラミングで指導すべき内容

項番	指導項目	Java 言語	C 言語
1	プログラミングとは ・プログラミング言語の種類 ・プログラムの基本構成	Java の特徴 オブジェクト指向言語	C 言語の特徴 UNIX 開発言語 手続き型言語
2	ソフトウェアのしくみと動作 ・プログラムの開発環境 ・ソースプログラムの作成 ・コンパイルと実行 ・PC 教室実行環境の説明 ・CPadの使い方	Java コンパイラ JVM JavaAPI class ファイル	コンパイルと実行 exe ファイル

3	プログラミングの基礎 ・ソースプログラムの書き方 ・定数と変数 ・整数型 ・実数型（単精度と倍精度） ・文字型（文字と文字列） ・基数表現（10 進数、8 進数、16 進数） ・コメントの付け方	クラス main メソッド 基本データ型 参照型 char と String	main 関数 include
4	データ入出力の基礎 ・標準出力 ・標準入力	データの表示 　System.out.print キーボード入力 　BufferedReader ラッパークラス 改行（¥n） DecimalFormat の指定	データの表示 　printf 関数 キーボード入力 　scanf 関数 書式指定の方法 　%d,%f,%o,%x,%c,%s 改行（¥n） 表示桁数の指定方法
5	基本的な演算 四則演算（＋－*/）剰余演算（%） 割り算における商の型、キャスト 演算子の種類と使用法 ・インクリメントと代入演算子	（Java／C 共通）	
6	文字と文字列 ・文字コード ・大文字 / 小文字変換	Unicode toUpperCase toLowerCase	ASCII コード toupper tolower
7	アルゴリズムの基礎 制御構造とフローチャート ・if 文,for 文,while 文,switch 文 ・関係演算子と論理演算子 ・合計、平均、最大、最小 ・多重ループ	（Java／C 共通）	
8	データ構造　－配列－ ・配列要素と添字 ・数値配列と文字配列 ・文字配列の表示と暗号化 ・２次元配列、添字と行・列の関係 ・２次元配列のプログラム例	（Java／C 共通）	
9	ユーザ定義処理	ユーザ定義メソッド 引数と戻り値 return、void	ユーザ定義関数 プロトタイプ宣言 引数と戻り値 return、void
10	実践的なプログラム ・整列アルゴリズム ・線形探索アルゴリズム ・二分探索アルゴリズム ・乱数の生成 ・ゲームプログラム	（Java／C 共通）	
11	ブラウザとの連携 ・Java アプレット ・アプレットビューア 基本図形の作成 カラー表示の原理 いろいろなグラフィック模様	Java アプレット applet タグ RGB による色指定 drawRect drawLine drawOval drawString setColor　など	（範囲外）

ではないため、各項目が１週分の授業に対応しているわけではないことを付け加えておく。

４．Javaプログラミングで指導すべきポイント

筆者が担当した授業は、プログラミング経験を持っていない受講生が大多数を占めている。初心者はプログラムの完成に至るまでに、さまざまな場面でつまずくことになる。たとえ数行程度の小さなプログラムであっても、正しく実行させるのは大変なことである。彼らはどのような点につまずくのであろうか。本章ではJavaプログラミング教育の指導方法や留意点について、さまざまな角度から考察していく。

4.1　プログラムのコンパイル・実行方法に関わること

Windows環境でJavaを実行する方法はいろいろあるが、もっともベーシックな形は、コマンドプロンプトを起動してコンパイル・実行を行う方法である。はじめにこの流れを確認しておく。

```
a. ソースプログラムの作成（テキストエディタ使用）
      ↓
b. コンパイル（コマンド例：javac Prog1.java ）
      ↓
c. 実行　　（コマンド例：java Prog1 ）
```

図4.1.1　コンパイル＆実行の流れ

慣れの問題はあるにせよ、初心者にとってこのようなコマンドベースでプログラムを作成していくことは、一般的に困難であると思われる。実際には上記のコマンド以外にもディレクトリやパスを設定するためのコマンド操作が必要になる。GUI操作に慣れた者にとっては、コマンド入力よりもマウスで操作できる方が望ましい。前述したように、筆者の授業ではプログラムの実行環境としてCPadを利用した。このソフトウェアはGUIベースの開発環境が提供されており、コマンド操作による煩雑さという問題点はほとんどクリアされていた。初心者の場合、コンパイルを終了するまでには幾度となくソースリストを修正することになる。したがってコンパイルから実行までの一連の流れが、簡単なマウス操作で処理できることは、初級プログラミングの実行環境として必要な要件であると考える。

前置きがやや長くなったが、コンパイル・実行の操作に関連して、よく見かけたエラーについて述べる。圧倒的に多いのは、ソースプログラムのファイル名とクラス名の不一致によるエラーである。Javaでは、ソースプログラムのファイル名は、クラス名.javaでなければならない。以下に挙げるエラーは頻出のものであり、指導上留意すべき点であると考える。

(1) 拡張子が「java」でないケース

入力ミスなどによってファイル名の拡張子が「java」でないソースプログラムに対しては、コンパイル自体が実行されない。たとえばファイル名を「Prog1.jav」とすると、コンパイルしたときに「javac: Prog1.jav は無効なフラグです」のようなエラーメッセージが表示される。メッセージの意味は多少わかりにくいが、注意してみれば比較的見つけやすいエラーといえる。

(2) ソースファイル名とクラス名の不一致

ソースファイル名が「Prog1.java」でクラス名が「Prog」のようなケースである。ソース自体に文法エラーがなければ、コンパイルは完了し「Prog.class」の名前でクラスファイルが生成される。このときCpadで「コンパイル＆実行」を行うと次のようなメッセージが出る。

```
コンパイルに失敗しました
ファイル Prog1.class は存在しません
```

図4.1.2　ソースファイル名とクラス名の不一致

この時表示されるエラーメッセージは受講者にとって少しわかりにくいものである。実際にはコンパイルは成功しており、実行ができないわけである。エラーの状況はコンパイルと実行を別々に行うとより明確になる。クラスファイルを直接実行してみると、コマンド「java Prog」では正しく実行されるのに対して、コマンド「java Prog1」では「Exception in thread “main” java.lang.NoClassDefFoundError: Prog1」となるからである。この例外はJavaでは頻出のエラーの1つであるが、Cpadの「コンパイル＆実行」では見ることができない。ソースファイル名とクラス名を一致させることは、学生への指導の際、何回も繰り返して説明しているが、エラーメッセージからは読み取りにくいエラーとなっている。なお、C言語の場合はクラスという概念がないためこのエラーは発生しない。

(3) 大文字・小文字の不一致

Javaは大文字と小文字を区別するため、正しい使い分けをしないとエラーになる。たとえばソースファイル名が「Prog1.java」でクラス名が「prog1」のような場合である。本質的にはソースファイル名とクラス名が不一致のケースと似ているが、エラーメッセージの出方が少し異なる。「コンパイル＆実行」を行うと、エラーメッセージ「Exception in thread “main” java.lang.NoClassDefFoundError: Prog1（wrong name: prog1）」が出る。wrong nameと表示されるため、前記（2）のケースに比べて原因を見つけるのは比較的容易であるといえる。ただWindowsに慣れている者の場合、ファイル名の大文字・小文字の違いはあまり意識しないことが多く、意外と見つけにくいエラーでもある。なお、このエラーもC言語では発生しない。

(4) クラス名として使用できない文字

ハイフンやピリオドなど識別子として使用できない文字が含まれているとエラーになる。たとえばクラス名にハイフン文字を入れて「Prog-1」のように書くと、図4.1.3のようなコンパイルエラーが出る。

```
Prog_1.java:9: ’{’ がありません。
class Prog-1 {
          ^
Prog_1.java:31: ’}’ がありません。
^
```

図4.1.3　クラス名のエラー（ハイフン）

実際にはクラス名の間違いであるが、「カッコがありません」という文字に目がいってしまうため、発見しにくいエラーといえる。ハイフン以外ではピリオドを付けてしまう間違いもよく見受けられる。たとえばクラス名を「Prog_1」とすべきところを、拡張子付きで「Prog_1.java」と書いてしまうケースである。この場合もクラス名に使用できないピリオドが含まれていることがエラーの原因になっている。この場合も「カッコがありません」というエラー表記になる。前記（2）のケースと同様、クラス名の間違いに気づきにくい要因となっている。

4.2　文法やコーディング規則に関わること

続いてプログラミングの文法や表記に関する留意点について述べる。前述の「コンパイル＆実行」の操作に関わるエラーは、ほとんどがファイル名とクラス名の不一致に起因するものであった。一部コンパイルエラーとなるものも含まれていたが、以下に述べるケースもコンパイル時に留意すべき点が多く含まれている。

(1) 日本語文字の扱い

JavaではUnicodeを用いて日本語文字を表現する。ソースプログラムの中で使用するキーワードはすべてASCII文字でなければならない。一方、識別子はUnicodeによる日本語表現が可能である。すなわち変数名、クラス名、メソッド名などにいわゆる全角文字を使

うことができる。識別子に全角文字を積極的に用いるべきかどうかは議論の分かれるところである。漢字表記を入れることによって可読性が高まるのであれば、全角文字の使用は効果的である。たとえば図4.2.1のようなケースでは、変数名が日本語で表現されているため処理内容が判別しやすくなっている。

```
int 単価,数量,売上合計;
売上合計 = 単価 * 数量;
```

図4.2.1　識別子の日本語表記（1）

その反面、短所も考えられる。一つ目はソースコード入力の手間の問題である。上記の例で言えば型名のint、カンマ、セミコロン、演算記号などはASCII文字である。1つのセンテンスを入力するのに「全角／半角」の切り替え操作が何度も必要になる。現実問題としてこれは結構な手間であり、コンパイルエラーを誘発させる原因にもなる。また使い方によっては逆に可読性が低下する場合がある。たとえば図4.2.2のケースである。ソースコード中に複数の「合計」という文字があるが、変数名、文字列リテラル、コメントの区別が一見しただけではわかりにくい。これは極端な例であるが、結果的に可読性を低下させてしまっている。

```
int 合計;
System.out.print("合計＝"+合計); //合計表示
```

図4.2.2　識別子の日本語表記（2）

このように識別子に日本語を使用することについては、一長一短がある。筆者の授業では、識別子に日本語を用いない指導方法を採用した。これはプログラミング初心者にとって、前述の短所の影響が大きく出ると判断したためである。

ソースプログラム中に全角文字が入る事例について、さらに考察をすすめる。基本的に日本語を指定できない箇所に全角文字を書いた場合はコンパイルエラーとなる。よく見かけるのは、次のように全角の空白、セミコロン、ダブルコーテーション、カッコなどを入れてしまうケースである。

```
int□kazu;                          //①空白
kazu=100；                          //②セミコロン
System.out.print("¥n"); 」          //③閉じカッコ
System.out.print("文教大学”);       //④引用符
```

図4.2.3　全角文字のエラー事例

全角文字は不正な文字として認識される。このときのエラー表記は全角文字がどのように指定されているかによって異なる。図4.2.3のケース①～③は、ほぼ共通でどれも不正な文字のUnicode番号が出力される（全角文字の混入位置によっては、他のエラーメッセージが併記されることもある）。たとえば①では「￥12288 は不正な文字です」のようなエラーになる。Unicode番号は10進表記になるため見慣れない数値であるが、12288＝0x3000＝「空白」を表している。Unicode番号で文字を判断することは困難であるが、ソースコードの行位置が併せて示されるため、エラーを発見することは比較的容易である。なお①のような空白文字を発見するためには、エディタに全角スペース、半角スペース、タブ文字を「可視化表示」する機能が必須となる。この点Cpadは問題ないが、Windowsの「メモ帳」にはこの機能がない。Javaプログラミングに限ったことではないが、一般に「メモ帳」はソースプログラムやHTMLの編集には不向きといえる。

④のケースについては「不正な文字」というエラー表記は出ない。エラー箇所が文字列リテラルの中にあるため、全角文字「”」が含まれていてもコンパイラは文字列が継続していると判断してしまう。その結果「文字列リテラルが閉じられていません」「 ')' がありません」といった表示になる。ただし、この場合でもエラーとなったソースコードの行位置は示されるため、エラーを発見することは

それほど困難ではない。

やや面倒なのは、クラス名に全角文字が混入するケースである。このケースは前節「4.1 (2) ソースファイル名とクラス名の不一致」とほぼ同様に考えてよい。たとえばソースファイル名が「Kanji.java」でクラス名が全角文字「Ｋａｎｊｉ」のような場合である。プログラム自体に文法エラーがなければ、コンパイルは正しく完了する。その結果「Ｋａｎｊｉ.class」の名前でクラスファイルが生成されるが、ソースファイル名とクラス名が一致していないため実行はできない。このときのエラー表示は次のようなものである。

```
コンパイルに失敗しました
ファイル kanji.class は存在しません
```

図4.2.4　クラス名の全角文字

前節でも触れたが、わかりにくさの第一は、メッセージの内容である。コンパイルは正常に完了しているにも関わらず「コンパイルに失敗しました」というメッセージが出てしまう（なおコマンドプロンプトから実行した場合は、No Class Def Found Errorのエラーメッセージが出力される）。第二は全角文字と半角文字の違いを画面上で見分けることが非常に困難なことである。目で見て文字の違いを判別できればあまり問題にならないが、実際には判別がきわめて難しい。PCの標準フォントがプロポーショナルフォントの場合、目視によって両者を区別することはほとんど不可能である。筆者の授業においてもよく見かけられたが、学生にとっては非常に発見しにくいエラーであった。なお補足しておくと、クラス名だけでなくソースファイル名を「Ｋａｎｊｉ.java」のような全角文字にしておけば、ソースファイル名とクラスファイル名の不一致は起こらないため正しく実行される（ただし拡張子は半角文字でなければならない）。

(2) カッコの組み合わせ

ソースプログラムの入力において、初心者がよく間違えるミスにカッコの組み合わせがある。Javaで扱うカッコには、小カッコ「( ) パーレン」と中カッコ「{ } ブレース」がある。小カッコが用いられる箇所は、主にif文、for文の条件指定やメソッドの指定時である。小カッコは１行から数行の範囲で完結することが多く、比較的ミスは起こりにくい。中カッコはクラスやメソッドの範囲を示すために使われる。また処理の一部をブロック化する場合にも用いられる。中カッコの対象範囲は、ソースプログラム数十行（あるいはそれ以上）に及ぶこともあり、小カッコに比べてかなり大きな範囲となる。したがって初心者のミスとして目立つのは、中カッコの場合が多い。以下、いくつかの事例を挙げる。

①右カッコが足りないケース

if文やfor文でブロックを構成する場合に、カッコを入れ忘れるケースがよく見受けられる。図4.2.5は右カッコが足りないケースである。for文の右カッコ（*2）がない場合、コンパイラはカッコの対応関係を点線で示したよう判断する。その結果、左カッコ（*1）に対応する右カッコが存在しないと判断され、エラーメッセージは最終行の位置に出力される。そのときのメッセージは「 '}' がありません」という内容になる。多くの場合、このようにカッコを入れ忘れた位置に関わらず、最終行にエラーが表示されることになる。エラー原因は容易に判別できるが、場所を特定するためにはカッコの対応関係を注意深く見ていく必要がある。

```
class Prog {  *1
    public static void main(String[] args) {
        for (ループ指定) {
          処理
        }  *2 → このカッコがない場合
    }
}
```

図4.2.5　右カッコが足りないケース

②右カッコが多いケース

次に右カッコが多いケースを考える。図4.2.6のようなケースであるが、この場合は少し面倒である。for文に余分な右カッコ（*3）が入っているため、コンパイラはカッコの対応関係を点線で示したように判断する。その結果、最終行位置にある右カッコ（*4）に対応する左カッコが存在しないと判断される。その結果、①のケースと同様、エラーメッセージは最終行の位置に出力される。

このとき表示されるエラーメッセージは「'class' または 'interface' がありません」となる。コンパイラから見れば、「Prog」はクラスとして正しく完結している。したがって*4の右カッコは「Prog」とは無関係であり、それゆえ該当するクラス（またはインタフェース）がないというわけである。①のような「カッコがありません」というメッセージが出るわけではない。

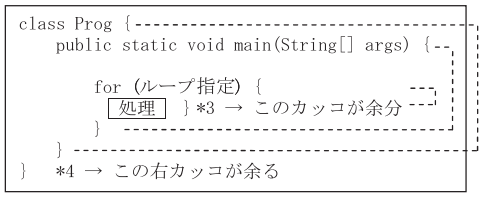

図4.2.6　右カッコが多いケース

右カッコが多いというのは、左カッコを入れ忘れるのと同義である。その場合左カッコを入れ忘れる位置によって、前記以外にもさまざまなエラーメッセージが出る。よく見かけるエラーは「<identifier> がありません」「';' がありません」「型の開始が不正です」などである。これらのエラーはコンパイルエラーとしては頻出のものであるが、エラー原因や場所の特定がしにくいエラーであり、初心者がデバッグに苦労する典型的なケースとなっている。

以上見てきたようにカッコの指定ミスには、いくつかのパターンがある。カッコを正しく組み合わせることは、コンパイルエラー、論理エラーを出さないための重要な要件である。ただ初心者にとっては、カッコを正しく組み合わせ、全体の構造や階層関係がわかるよう記述することが苦手なようである。図4.2.7は実際に学生が書いた例である。インデントを使わずにコーディングしているため、どのカッコの組み合わせが不正であるのか、非常に読み取りにくい。おそらくプログラムを作成している本人自身がわからなくなっている。正しい形は図4.2.8である。インデントを使ってカッコの多重度を見やすくする指導の重要性がここに表れている。

```
class Prog
{
public static void main  ･･･
{
        文１;
{
        文２;
        文３;
}
}   →このカッコが１つ多い
}
}
                    （誤った例）
```

図4.2.7　カッコの組み合わせの例（誤り）

```
class Prog
{
    public static void main  ･･･
    {
        文１;
        {
            文２;
            文３;
        }
    }
}
```

図4.2.8　カッコの組み合わせの例（正しい）

(3) Java アプレットにおける表記

筆者の授業では、Javaアプレットの題材を一部取り入れている。アプレットを利用することで、比較的手軽にウィンドウ型アプリケーションを作成することができる。Javaのプログラムをウェブ画面上で動作させることで、学生はJavaが活用されている領域の広さを、より実感できるのではないかと考える。Javaアプレットを実行するには、ウェブブラウザを使う方法とアプレットビューアを使う方法の２つがある。前者はHTMLファイルの

中にJavaアプレットを埋め込む形態であり、ブラウザを起動したときにアプレットが動作する。実用的かつ実際的なスタイルといえる。一方のアプレットビューアは、簡易ブラウザ上で実行されるもので、実行時にHTMLファイルを必要としない。主にプログラムのデバッグ目的に用いられる。実用性は低いが、アプレットビューアは実行結果をすぐに確認できるという利点がある。この手軽さは初級者向けプログラミングには非常に有効であると考え、筆者の授業ではアプレットビューアによる方法を採用した。

①アクセス修飾子の指定

Javaアプレットのソースプログラムは、図4.2.9のような書き出しになる。import文で2つのパッケージを指定したあと、クラスの指定にAppletクラスの継承と、アクセス修飾子publicの指定を加える。またmainメソッドの代わりにグラフィック描画用のpaintメソッドを指定する。

Javaアプリケーションではクラスの指定でアクセス修飾子を省略することができたが、アプレットのソースコードにはpublicの指定が必須になる。指定を忘れるとコンパイルは通るが、実行時エラーとなる。またpaintメソッドにpublicの指定をつけないとコンパイル自体ができない。現実問題としては、最初に一度正しい形で作成しておけば以降はそのソースコードを流用することが多いので、特に問題になることではない。ただJavaアプリケーションとの相違点という意味で、指導上注意が必要であろう。

```
import java.applet.*;
import java.awt.*;

public class Applet_prog1 extends Applet {
        public void paint(Graphics g) {
```

図4.2.9　Javaアプレットの指定

②アプレットコードのクラス名指定ミス

アプレットビューアで実行する場合はJavaソースコードの中にappletタグを入れる必要がある。このときクラスファイルの名前を間違えるとエラーが発生する。その際、誤って空白文字を混入してしまうと、以下のように見つけにくいエラーが発生する。図4.2.10はファイル名が「Prog_01.java」であるのに対して、applet codeの名前を「Prog_01␣」としてしまったケースである（末尾に空白文字が混入）。この場合、コンパイルは問題なく終了するが、実行時にコマンドプロンプトに図4.2.4のようなエラーメッセージが出る。注意深く見るとクラス名の末尾に空白文字があることは確認できるが、一見正しく読めてしまうため、きわめて発見しにくいエラーとなる。指導上留意すべき点であろう。

```
<applet code="Prog_01 " width="800" height="600">
</applet>
```

```
c:¥java>appletviewer Prog_01.java
load: クラス Prog_01 が見つかりません。
java.lang.ClassNotFoundException: Prog_01
```

図4.2.10　アプレットのクラス指定ミス

4.3　アルゴリズムに関わること

コンパイルエラーではないが、プログラムが論理的に正しくないため実行時に不正な結果となるケースがある。コンパイルエラーにならずに実行ができてしまうため、学生にとってはより困難なデバッグ作業となることが多い。実行時エラーにはさまざまなケースがあるが、よく見られる事例を挙げてみる。

(1) セミコロンの指定位置に関すること

通常、Javaのセンテンスはセミコロンで終了するため、文法上必要な箇所にセミコロンを入れ忘れるとコンパイルエラーになる。反対に、文法上問題がなくても論理的に正しくない場所にセミコロンを入れると、空文として処理されるため実行結果が不正となる。図4.3.1の2例はif文およびfor文の条件指定の直

後にセミコロンを付けてしまったケースである。いずれもセミコロンによって文が終了してしまうため、処理結果が不正となる。

```
if (tensu>=60) ;    // → 不要なセミコロン
      System.out.print("合格です");
```

```
for ( n=0 ; n<10 ; n++ ) ;{  // → 不要なセミコロン
      処理A ;
}
```

図4.3.1　不要なセミコロンの例

if文の例では実行したときに「tensu」の値に関係なく「合格です」の文字が表示されてしまう。もし「tensu」の値が偶然60以上ならば、結果的には正しい実行結果が得られてしまうため、バグの発見は困難になる。for文の例では、「処理Ａ」が10回繰り返されるべきところを、1回しか実行されないという結果になる。ループ処理が正しく動作しない場合、初期値や繰り返し条件に目がいってしまうため、結果としてエラー原因が発見できないことが多い。「行の終わりにはセミコロンを付ける」と機械的に理解している学生がよく間違えるミスである。単純なミスの割にはデバッグに苦労する事例といえる。

(2) 制御構造とブロック化

if文やfor文では、制御構造の中に複数のセンテンスが入る場合、中カッコを用いてブロックを構成しなければならない。単文の場合は中カッコを省略できるので、その書き方に慣れてしまうとブロック化が必要なときに中カッコを付け忘れることになる。この場合コンパイルエラーにならずに論理エラーが発生する。図4.3.2はaがbより大きいときに、aとbの値を交換する処理である。

```
//  ケースＡ（正しい）
if ( a>b ) {
    w=a;
    a=b;
    b=w;
}
```

```
//  ケースＢ（誤り）
if ( a>b )
    w=a;          (1)
    a=b;          (2)
    b=w;          (3)
```

図4.3.2　ブロック化の例

ケースＡは正しくブロック化されているのに対して、ケースＢは中カッコがないため、if文の制御は(1)のw=a;の文にしか作用しないことになる。2つのケースはプログラムの形が似ているため、初心者にはソースコードレベルで両者の違いを理解することが難しい。このような場合、筆者は流れ図を用いて説明することが有効であると考えている。図4.3.3は両者をフローチャートで表現したものである。制御の流れが線で示されるため、ソースコードに比べ処理対象の範囲がより明確になっている。

フローチャートはアルゴリズムを図式化するものであるため、Javaのようなオブジェクト指向型プログラムの表記には、不向きであると言われている。ただこのような基本的アルゴリズムを表現する場合には、フローチャートによる図式表現も有効になると考える。

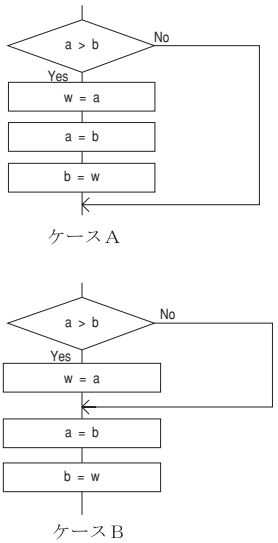

図4.3.3　フローチャートによる比較

なお補足するとJavaではケースＢの場合、変数wを初期化していないと（3）の文でコンパイルエラーが発生する。メッセージは「変数 w は初期化されていない可能性があります」という表示になる。このような一時的変数は、初期化しない使い方が普通であるため、コンパイル時にミスに気づくことができる。これに対してC言語ではこのようなコンパイルエラーは出ない。その結果、実行時に変数bに不正な値が設定されてしまうことになり、より注意が必要である。

(3) 変数の初期化に関連するエラー

前述のケースと関連するが、Javaでは初期化あるいは値が設定されていない変数など、既知でない変数を参照するとコンパイルエラーが発生する。たとえば図4.3.4の「初期化なし」のケースの場合、printメソッドの位置でコンパイルエラーが出る。

```
int kazu;          //  初期化なし
System.out.print("値＝"+kazu+"です");

  →  コンパイルエラーとなる
```

```
int kazu=100;   //  初期化あり
System.out.print("値＝"+kazu+"です");
```

図4.3.4　変数の初期化

これは未確定のままの変数を不用意に参照しないようにするための警告である。このような警告が出ることでJavaでは実行時エラーを未然に防ぐことができる。ただし変数への値設定をif文やfor文などと組み合わせて行う場合、論理的に正しくてもコンパイルエラーが出るケースがあるので注意が必要である。

```
//  ********** ケースＡ **********
int kazu;

if ( 条件１ )
    kazu=100;
System.out.print("値＝"+kazu+"です");
```

```
//  ********** ケースＢ **********
int kazu;
for ( 一定回数の繰り返し ) {
        if ( 条件２ )
            kazu=100;
        else
            kazu=200;
}
System.out.print("値＝"+kazu+"です");
```

図4.3.5　任意の条件下での変数の値設定

たとえば図4.3.5のケースであるが、ケースＡ、ケースＢのいずれの場合もprintメソッドでコンパイルエラーが発生する。ケースＡはif文の条件によって、kazuに値が設定されない可能性があることから、コンパイルエラーとなる。ケースＢではfor文の中にif else型のif文があり、ループ処理の中を少なくとも1回以上実行すれば、kazuに値が設定されるのは明らかである。しかしfor文の繰り返し条件によってはループ処理の部分をスキップしてしまう可能性もあることから、コンパイルエラーが出てしまう。for文の中を確実に実行することがわかっている場合、プログラマの感覚としては（自ら値を設定しているので）kazuの値を初期化する必要性を感じない。ただし、それでもコンパイルエラーが出てしまうのである。この点指導上の注意が必要である。

4.4　プログラミングの前提となるデータ表現の知識に関すること

プログラムの作成とは、単にアルゴリズムを組み立てていくだけの作業ではない。正しいアルゴリズムを作成するためには、そのベースとしてハードウェアやソフトウェアのしくみ、情報科学の基礎知識などが必要とされる。初級プログラミングのレベルにおいては、コンピュータで処理される情報がどのように表現されているかを知ることがまず必要にな

る。

コンピュータで表現される情報には、数値、文字、図形、色がある。数値表現についていえば、記数法（2進数、8進数、16進数）や数の表現（整数、浮動小数点、補数など）が基礎知識として求められる。プログラムの中で数値を表現する場合、使用できる変数にはさまざまな種類がある。たとえばJavaで扱える整数型にはbyte、short、int、longがある。また実数型にはfloatやdoubleがある。自分が扱いたい数値を適切な型で表現するためには、それぞれのデータ型を正しく理解していなければならない。基数変換を行うプログラムを作るのであれば、２進数、８進数、16進数の計算知識は必須となる。また文字や文字列を扱うプログラムを作るのであれば、文字データの表現方法を理解していなければならない。そこではASCIIコードやUnicodeといった文字コードの種類、文字コード体系、１バイトコード／２バイトコードの違いなどの知識が必要とされる。

さらにJavaアプレットを用いて図形や色を表現するプログラムを作るのであれば、グラフィックスに関する知識が必要になる。画像を構成するピクセルの概念、色数とビット数の関係、RGB値による色表現法などは必須の知識となる。

例として、色表現法の基本知識がプログラミングと関わる事例を１つ挙げる。図4.4.1はJavaアプレットを使って10個の四角形を表示するプログラムである。for文を用いて、異なる色の10個の四角形を表示するアルゴリズムになっている。ループ処理の中で、RGB値を小さな範囲で増減させることにより、色合いが少しずつ変化する。結果として10個のグラデーション模様ができあがる。図4.4.1の例は、色の初期値が「黒」であり（red=green=blue=0）、１回のループでredの要素を15ずつ増加させている。greenとblueの値は変化させていないので、結果的に黒から赤へと変化する四角形が10個表示されることになる。このときredの増分値を変更することによって、色の変化率を変えることができる。授業ではredの増分値を次のように変化させながら、学生に結果を確認させている。

① red+=10;
② red+=20;
③ red+=30;

上記①のケースでは、redの変化量が+10と小さく10回のループでは最終値が90になる。見た目の色合いは赤というより、赤味がかかった黒に近い。②のケースではredの変化量が①の２倍であり最終値は180となる。全体的にやや暗めではあるが、黒から赤へ変化していく様子が見て取れる。③のケースではredの増分が+30になるため、より明るい赤へと変化していく。しかしこのプログラムは実行の途中で異常終了する。理由はRGBの上限値が255を超えてしまうからである。

このプログラムの実習は、ループの概念、RGB値の意味、実行時エラーの対処法を学ぶことを目的としている。初心者にとってはプ

```
Color iro;
int red=0,green=0,blue=0;                    //  色の初期値は黒（R=G=B=0）
int x,y,haba=50,takasa=20;

for ( int n=1 ; n<=10 ; n++) {               //  図形を 10 個描く
    iro=new Color(red,green,blue);
    g.setColor(iro);
    g.fillRect(x,y,haba,takasa);             //  四角形の描画
    y+=30;                                   //  次の位置へ移動

    red +=15;           //  赤の増分→この値を変更する
    green+=0;           //  緑の増分（変更なし）
    blue +=0;           //  青の増分（変更なし）
}
```

図4.4.1　四角形の描画

ログラムのコンパイルエラーよりも、論理エラーをデバッグする方がはるかに難しいが、このケースは論理エラーを学習させるよい事例になっている。また同時に、図形描画や色表現の知識を学ぶための適切な教材であるといえる。

なお付け加えておくと、多くの学生は、当初この実行時エラーにほとんど気がつかない。理由の一つは、エラーメッセージがアプレットビューア画面のうしろに隠れてしまうからである。さらには不完全ながら実行結果が画面に表示されるため、一見「正常に動作している」ように見えてしまうことも、エラーに気づかない原因となっている。実際には、コマンドプロンプト画面に、図4.4.2のようなエラーメッセージが表示されている。注意深く見れば、赤の色が値をオーバーしたことは読み取れる。しかし、学生のウィンドウにはアプレットビューアの結果が前面に表示され、コマンドプロンプトは背面に隠れている。異常終了はredの値が240から270に変わったところで発生するため、結果的には 9 個の四角形は表示される。実行結果が何も表示されないのであれば、エラーに気づくことは容易であるが、このケースの場合は、エラー自体を見過ごしてしまうのである。このような点も指導上、留意する点であるといえる。

```
java.lang.IllegalArgumentException: Color
parameter outside of expected range: Red
```

図4.4.2　実行時のエラー画面

4.5　Java固有の表記法に関すること

学習に適した題材が多くあるというのはJavaプログラミング教育を行うことの利点の一つである。ただ初心者にとっては、「多機能で範囲が広い」というJavaの特徴が、逆に障壁になってしまう面も見受けられる。

(1) 予約語や標準入力時の煩雑さ

例として変数の値をコマンドプロンプトに出力する処理を考えてみる（図4.5.1）。ケースＡは代入を使って変数に値をセットするプログラムであり、ケースＢではキーボードから入力した値を変数にセットしている。

```
変数の値は 15 です
― Press any key to exit ･･･
```

ケースＡ（プログラム中で値を代入）

```
整数を入力してください―>15
入力値は 15 です
― Press any key to exit ･･･
```

ケースＢ（キーボードから値を入力）

図4.5.1　標準出力の例

いずれも処理内容は非常にシンプルなものであるが、ソースリストの見た目はかなり異なるものとなる。図4.5.2はケースＡのソースリスト、図4.5.3はケースＢのソースリストである。筆者の授業では、キーボード入力と標準出力とを組み合わせるプログラムを利用することが多く、ケースＢはその基本形となっている。両方のプログラムに共通する予約語や識別名に、class、public、static、void、String、args、int、System、outがある。いずれも、第1回目の授業で紹介する例題にすぐに登場するものである。このうちclass、String 、intについてはある程度説明を行うことになるが、それ以外の語句についてはほとんど触れない。早い段階の授業で理解させることが、まず不可能だからである。必然的に「今は意味を理解できなくてもよいから、とりあえずこのまま記述しなさい」という指導になる。ケースＢの場合は、ソースコードの分量がさらに増えて10行以上になる。わずか２行の出力結果を得るだけであるが、実行画面からは想像しにくい処理や記述が多く含まれることになる（図4.5.4）。

```
class Print_01 {
    public static void main(String[] args){
        int kazu;
        kazu=15;                                    //  変数kazuに値をセット
        System.out.print("変数の値は"+kazu+"です¥n");
    }
}
```

図4.5.2　ケースＡのソースプログラム

```
import java.io.*;
class Input_01 {
    public static void main(String[] args) throws Exception {
        int kazu;
        BufferedReader inp=new BufferedReader
            (new InputStreamReader(System.in));
        String keybd;
        System.out.print("整数を入力してください-->");
        keybd=inp.readLine();
        kazu=Integer.parseInt(keybd);           //  変数kazuに値をセット
        System.out.print("入力値は"+kazu+"です¥n");
    }
}
```

図4.5.3　ケースＢのソースプログラム

キーボードからの入力処理を加えるために、ケースＢではこれだけの記述が必要になっている。先のキーワードと同様であるが、この段階では説明のしようがないので「理解できなくてもよいからとりあえずこのまま記述しなさい」となる。このように説明できない箇所を一部ブラックボックスにしたままプログラミング作業が進んでいくことになる。どのような言語であれ、学習の早い時期に完成形のソースコードを真似て書くことはよくある。コピー＆ペーストを使えば、労力自体はたいした問題ではない。ただ個々のキーワードの意味が理解できていないため、コンパイルエラーが出た場合の対処は難しくなる。その結果、デバッグ効率も悪くなる。Javaの多機能さがプログラミング学習への障壁になってしまう例である。このようにブラックボックスが他の言語に比べて多いことが、初心者にとってJavaを難しいと感じる理由になっているように、筆者は感じている。

・ioパッケージのインポート	import java.io.*;
・mainメソッドの例外処理	throws Exception
・入力バッファの設定	BufferedReader inp=new BufferedReader (new InputStreamReader(System.in));
・入力文字列を格納する変数	String keybd;

図4.5.4　ケースＢで追加された記述

(2) 識別子の命名規則や指定方法に関すること

Javaのソースコードは大文字と小文字を区別する。習慣として、クラス名は先頭を大文字で書き、変数名／配列名／メソッド名は小文字で書くのが一般的である。また2語以上が組み合わされる長い名称に対しては、単語の1文字目を大文字に変える方法もよく見られる。筆者の授業では、クラス名が長くなる場合、単語の切れ目を_（アンダースコア）で区切る方法を採用した。すなわち「ReidaiPrintInt01」と書くのではなく「Reidai_print_int01」と書くのである。その理由は、「Windows環境では大文字／小文字を区別しないため、両者が混在しているとフ

ァイル名の確認や修正がやりにくいこと」「入力時に大文字／小文字の入れ間違いを軽減できること」「単純に見た目の問題で、間隔を空けた方が読みやすいこと」である。Javaの標準的な表記方法とは異なることになるが、入力ミスを軽減でき初心者には一定の効果があったと考える。なお区切り文字についていうと、C言語ではファイル名の区切り文字にハイフンを用いることができる。一方、Javaで使用できるのはアンダースコアのみである。本論のテーマとは直接関係がないが、将来的にソースコード流用の可能性などを考えると、Cのプログラミングにおいても極力アンダースコアを使用しておくことが望ましいと考える。

変数名の付け方についてさらに考察を続けたい。プログラム中で名称がそれほど重要でない変数（たとえばローカル変数など）に、簡易な名称を付けることがよくある。ループ用のカウンタであればint counterと書くよりもint cntあるいはもっと単純にint iのような書き方がよく用いられる。筆者自身の場合も、必要最小限の文字数で命名することが多いように思う。学生の場合はいろいろなケースが見られる。例題に示された変数名を忠実に真似る学生（このケースが最も多い）、どの変数もみな１文字で表現しようとする学生、とりあえず思いつくままに名前を決めていく学生、などさまざまである。変数名について指導上留意すべき点として、以下の事例を紹介する。

前述のグラデーション模様のプログラム（図4.4.1）では、色の三要素RGBを表す変数をint red,green,blueのように定義している。このとき、変数名を省略して書くタイプの学生は、ほとんどがint r,g,bのような書き方をする。発想自体は悪くないのであるが、この中の「g」がアプレットのGraphicsクラスの引数と重なってしまう。その結果、図4.5.5のようなコンパイルエラーが出ることになる。

```
g は paint(java.awt.Graphics) で定義されています。
        int r,g,b;
```

図4.5.5　変数名に関するコンパイルエラー

Javaアプレットではメソッドをpublic void paint（Graphics g）のように指定するが、現実のコーディングではコピー＆ペーストによって記述することがほとんどであり、この行は学生から見ると一種のブラックボックスになっている。Graphicsクラスのオブジェクトを「g」以外の名称にすれば、もちろんエラーにはならないが、学生はそれに気づくことはできない（そもそもアプレットプログラムで「g」以外のオブジェクト名を用いることはあまり一般的でない）。必然的にint r,g,b の変数名を変えることになる。int r,green,b ではいかにもバランスが悪いので、結局int rr,gg,bb くらいに落ち着くことが多い。

これは一例にすぎないが、クラスやオブジェクトの概念を正しく理解していないとコンパイルエラーが解読できない、という状況に学生はたびたび遭遇することになる。ソースコードの一部をブラックボックスにしたまま授業を展開せざるを得ないという、Javaプログラミングの特性といえるであろう。

5．おわりに

本論では、筆者の授業を通してJavaプログラミング教育における指導上の留意点を中心に考察してきた。繰り返しになるがJavaにはプログラミング学習に適した題材が多く含まれている。それは単にアルゴリズムやプログラミングテクニックを習得することだけに留まらず、情報技術全般に関わる知識の習得に役立つ可能性を持っている。

今回取り上げたテーマは「初級プログラミング」の範囲である。このレベルではJavaの大きな特徴であるオブジェクト指向の本質には、ほとんど触れることができない。そのた

め本論でも述べてきたような、さまざまな「ブラックボックス」が授業の中に存在してしまう。インスタンス、継承、カプセル化、GUI、インタフェースといった概念は中級レベルの指導範囲となるであろう。一例として、「変数」を例に取れば、初級レベルでは変数の一般的な概念を述べるに留まっており、基本データ型と参照型の詳しい内容までは言及していない。そのため、実際に学生からは次のような質問が出る。変数を定義するとき、intやdoubleを小文字で指定するのに対し、StringやColorは何故大文字で指定するのか。また同じ変数なのに、エディタ上では何故intやdoubleだけ色付き文字になるのか（Cpadには基本データ型の予約語を色付きで表示する機能がある）。

これらは初級から中級へとプログラミング学習が進んでいく段階で、いずれ理解できる項目であろう。しかし初級レベルで本質的な答えを示すことは難しい。中には、疑問が解消されないまま授業が終了することを不本意に思う学生もいるかもしれない。しかしこのような課題点は残るにせよ、Javaが持っている広範な機能や特徴はプログラミング教育の有効性という点で非常に大きなものがある。指導の際「何をどこまで教えるか」そして「何は教えないか」を常に意識しておくことが、真に重要であろうと考える。

注

1）Cpadは稀杜（kito）氏作成のソフトウェア

参考文献

1）『Javaプログラミング』　広内哲夫著　創成社
2）『Javaプログラミング徹底入門　基礎編』内田智史著　電波新聞社
3）Java2 Platform Standard Edition 5.0（サン・マイクロシステムズ）
http://java.sun.com/j2se/1.5.0/ja/docs/ja/api/index.html