第0章　Javaの基礎知識

この章では Java の基礎知識をダイジェストで紹介する。

0.1　Javaとは

1995 年、Sun Microsystems 社から公表された、比較的新しい言語である。文法は、C あるいは C++ と似ているが、**互換性はない**。C++と同様、**オブジェクト指向**言語であるが、C++に比べてシンプルな仕様になっている。なお、**JavaScript** (ECMAScript) とは文法は似ている（JavaScript が Java に文法を似せている）が、それ以外の関係はなく全く別の言語であるので注意する必要がある。

0.2　Javaの特徴

Java は、誕生当時は Web ページにアニメーションとインタラクティブ性をもたらすための仕組みとして、世に広まった。アプレットと呼ばれる Java のプログラムはネットワークを通じて別のコンピュータに移動して実行されることになる。このような使い方をするためには**安全性**と**可搬性**という特徴が重要になる。

安全性　これは、簡単にいえばアプレットを使って他人のコンピュータに悪戯をすることができない、ということである。もし、ホームページに任意のプログラムを埋め込んでブラウザ上で実行させることができれば、ハードディスク中のデータを消去してしまうなどのイタズラが簡単に行なえる。

安全性を保障するためには、まずプログラムにファイル操作などをさせない、などの制限を課する必要があるが、C のような言語では、ポインタ（アドレス）操作や無制限な型変換などの仕組みを通じて、いくらでも抜け道を作ることができる。Java はこのような抜け道がないよう設計されている。

可搬性　Web ページに埋め込まれるということは、さまざまな機種のコンピュータで実行される可能性があるということである。つまり、Java のアプレットに機種依存性があってはいけない。**コンパイラ**を用いる実行方式ではプログラムが機械語に翻訳されるため、機種依存性は避けられない。一方、**インタプリタ**を用いる方式では、各機種毎にインタプリタを実装するだけで良いが、効率が犠牲になる。このため、Java では **中間言語方式**という方法をとる。

Java のプログラムは Java コンパイラによって **JVM** という仮想 CPU のコードに翻訳される。この仮想コードを各 CPU 上の JVM エミュレータ（一種のインタプリタ）が解釈・実行する。

このように当初、Java はアプレットを作成するための言語として広まった。現在では、インタラクティブな Web ページを作成するためのブラウザ側の仕組みとしては、Adobe Flash などが主流となって、Java アプレットは比較的マイナーな存在になっている。一方で Java の上記のような性質は、他の

分野のアプリケーションでも役に立つため、現在はむしろアプレット以外のアプリケーション（例えば WWW **サーバ側**で動作するサーブレット（Servlet）などのプログラム）を作成するために、広く用いられるようになってきている。

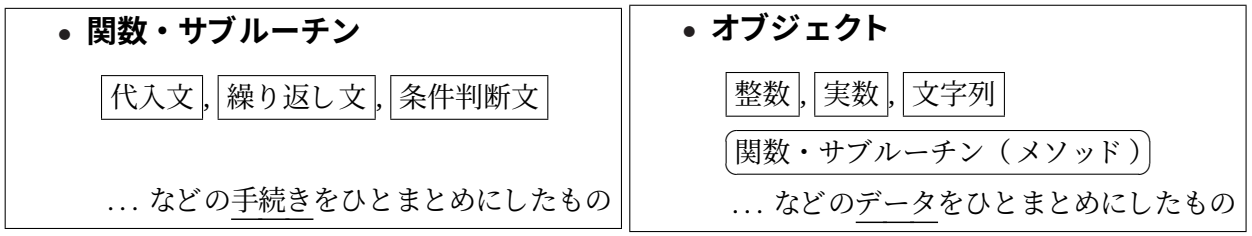
WWW サーバー側プログラム用のプログラミング言語としては、Perl, PHP, Python, Ruby なども有名だが、これらは**動的型付け**を採用している。つまり、実行時まで型エラーは検出しない。Java はこれらと違い**静的型付け**を採用している。つまり、実行前（コンパイル時）に型エラーを検出する。一般に静的型付けは大規模で信頼性が必要とされるシステムの記述に適している。

0.3 オブジェクト指向プログラミング

Java は**オブジェクト指向型**プログラミング（Object-Oriented Programming, OOP）言語である。**手続き型**言語・**関数型**言語・**論理型**言語・オブジェクト指向型言語などと、プログラミング言語を分類することがあるが、このような言語の分類は、主にプログラミング言語が備える部品化の仕組みに基づいている。

オブジェクト指向型言語に限らず、プログラムの部品を**設計**することは、単に**利用**することよりも格段に難しい。まずは、自分で独自のプログラム部品を設計するよりも、オブジェクト指向という仕組みのおかげで豊富に用意された Java の部品群を利用することを学ぶことが必要であろう。この節では、オブジェクト指向型言語が用意する部品を**利用する**ために必要な用語を紹介する。

オブジェクト指向（object-oriented）とは簡単に言えば、従来の手続きを中心としたプログラム部品（サブルーチン、関数）の利用に加えて、データを中心とした部品（**オブジェクト**）の利用を支援することである。関数（サブルーチン）はいくつかの手続きをまとめて一つの部品としたものだが、オブジェクトは、いくつかのデータ（関数 — **メソッド**（method）と呼ばれる—も含む）をまとめて一つの部品としたものである。



実際には、プログラム部品として提供されるのは、オブジェクトそのものではなく、オブジェクトの雛型とでもいうべき **クラス**（class）である。クラスは、そこから生成されるオブジェクトが（具体的なデータ（つまり、1 とか 3.14）ではなく）どのような名前と型の構成要素を持つか、のみを指定したものである。クラスを具体化（instantiate — つまり、`x` という名前の `int` 型の構成要素は 1 で、`y` という名前の `float` 型の要素は、3.14 などと定めること）したものがオブジェクトである。このとき、このオブジェクトはもとのクラスの **インスタンス**（instance, 具体例）である、という。

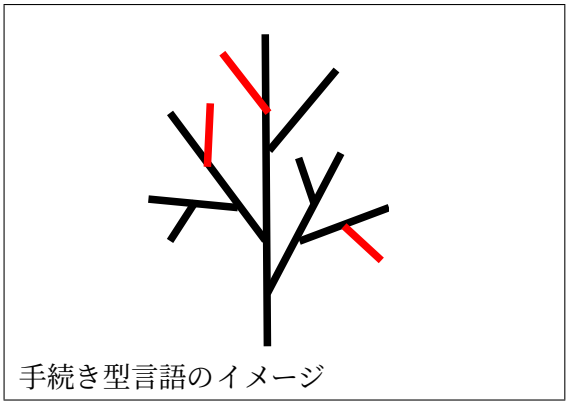
オブジェクトを構成している個々の構成要素を **フィールド**（field）あるいはインスタンス変数（instance variable）、メンバ（member）、という。ただし、関数型の要素は**メソッド**（method）と呼ぶのが普通である。オブジェクトのメソッドを起動することを、擬人的にオブジェクトに**メッセージ**（message）を送る、と表現することがある。

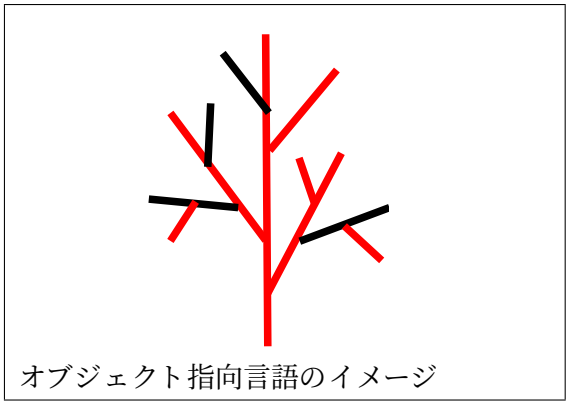
> 正確に言えば、メソッドについてはインスタンスごとにコードを定義するのではなく、クラスごとにコードを定義する（ようになっているオブジェクト指向言語が多い）。オブジェ

クトは各フィールドのデータの他に、どのクラスに属しているか、という情報を持っていて、それによって適切なメソッドのコードが起動される。

複数のオブジェクトがフィールドに内部状態を保持し、互いにメッセージを交換して、その内部状態を変更していく、というのがオブジェクト指向のプログラムの実行のイメージである。

従来型言語では、部品の再利用方法は、既存の部品を関数・サブルーチンとして呼び出すだけだったが、オブジェクト指向型言語では、それに加えて既存の部品（つまりクラス）を少しだけ書き換える（**継承する**, inherit）、という形の再利用の方法が可能になる。手続き型言語ではプログラムの “幹” の部分を変えて “枝” の部分だけを再利用することができたが、オブジェクト指向型言語では、“枝” の部分を変えて “幹” の部分を再利用することもできるのである。


手続き型言語のイメージ


オブジェクト指向言語のイメージ

最近のソフトウェアではユーザーインタフェースの部分（“枝” の部分）が重要であることが多いので、オブジェクト指向という考え方が特に必要となってきている。オブジェクト指向言語は GUI 部品（ボタンやテキストフィールドなど）のような特定の用途の多種のデータ型が必要とされるプログラミングに適している。

0.4 Java のクラスの定義

新しいプログラミング言語を学習するときの慣習により、最初に、画面に “Hello World!” と表示するだけのプログラムを作成する。

通常の Java アプリケーションの Hello World プログラムは次のような形になる。

例題 0.4.1 *Hello World* プログラム

ファイル Hello0.java

```
public class Hello0 {
  public static void main(String args[]) {
    System.out.printf("Hello World!%n");
  }
}
```

この `Hello0.java` の意味を簡単に説明する。

`public class` `Hello0` は `Hello0` という**クラス**を作ることを宣言している。（Java では、どんな簡単なプログラムでもクラスにしなければならないことになっているので、とりあえずこの形のまま使

えば良い。）Java では public なクラス名（この場合 Hello0）とファイル名（この場合 Hello0.java）の .java を除いた部分は**同じでなければならない**[1]。この例の場合はどちらも Hello0 でなければならない。この後のブレース（{）と対応する閉ブレース（}）の間がクラスの定義である。ここに変数（フィールド）や関数（メソッド）の宣言や定義を書く。

Java アプリケーションの場合も C 言語と同じように、main という名前のメソッド（関数）から実行が開始されるという約束になっている。main メソッドの型は C 言語の main 関数の型（int main(int argc, char** argv)）とは異なる void main(String args[]) という型になっている。public や static というキーワード（修飾子）については後述する。とりあえず、この形（**public static** void main(String args[])）の形のまま使えば良い。

なお、String は Java の文字列の型である。String は char の配列ではない。文字列リテラル（定数）は、C 言語と同様二重引用符（" 〜 "）に囲んで表す。

System.out.printf は C 言語の printf に相当するメソッドで文字列を書式指定に従って画面に出力する。つまりこのプログラムは、単に “Hello World!” という文字列を出力するプログラムである。%d, %c, %x, %s などの書式指定は C 言語の printf と同じように使用することができる。一方、%n は Java の書式指定に特有の書き方でシステムに依存する改行コード（Unix では ¥x0A, Windows では、¥x0D¥x0A）を表す。

また、Java 言語では **+演算子**で文字列を連接できる。文字列に数値を+で連接すると、数値が文字列に変換されて、文字列として連接される。%d などの代わりに、+演算子を使って数値などを出力することも多い。

0.5 HelloWorld サーブレット

例題 0.5.1 *Hello World* サーブレット

ファイル HelloServlet.java

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloServlet extends HttpServlet {
  @Override
  public void doGet(HttpServletRequest request, HttpServletResponse response)
              throws IOException {
    response.setContentType("text/html; charset=Windows-31J");
    PrintWriter out = response.getWriter();
    out.println("<html><head></head><body>");
    out.println("Hello World!");
    out.println("</body></html>");
    out.close();
  }
}
```

最初の数行の import 文は、java.io.PrintWriter、javax.servlet.http.HttpServletResponse などいくつかのクラスを使用することを宣言している。

[1] public でないクラス名に対しては、この規則は強制されないが、従っておく方が何かと便利である。

詳細: パッケージは OS のディレクトリやフォルダがファイルを階層的に整理するのと 同じように、クラスを階層的に管理する仕組みである。HttpServlet クラスの正式名称は、パッケージの名前を含めた javax.servlet.http.HttpServlet なのであるが、これを単に HttpServlet という名前で参照できるようにするのに

```
import javax.servlet.http.HttpServlet;
```

という import 文を使う。javax.servlet.http というパッケージに属するクラスすべてをパッケージ名なしで参照できるようにするには、

```
import javax.servlet.http.*;
```

という import 文を使う。

自作のクラスを他のクラスから利用する場合は適切なパッケージに配置するべきである。(自作のクラスをパッケージの中に入れるために package 文というものを使う。) アプレットやサーブレットの場合は、他のクラスから利用するわけではないので、パッケージなしでも良いだろう。(正確に言うと package 文がない時は、そのファイルで定義されるクラスは無名パッケージというパッケージに属することになる。)

Java の既成のクラスを利用するためには、そのクラスが属するパッケージを調べて、それに応じた import 文を挿入する必要がある。(もしくは、クラスをパッケージ名を含めたフルネームで参照する。)

次の public class HelloServlet extends HttpServlet は、HttpServlet という**クラス**を**継承**して (つまり、ほんの少し書き換えて)、新しいクラス HelloServlet を作ることを宣言している。(この時、HelloServlet クラスは HttpServlet クラスのサブクラス、逆に HttpServlet クラスは HelloServlet クラスのスーパークラスと言う。)

HttpServlet クラスは、サーブレットを作成する時の基本となるクラスで、サーブレットとして振舞うための基本的なメソッドが定義されている。すべてのサーブレットはこのクラスを継承して定義する。このため、必要な部分だけを再定義すれば済む。

行の最初の public はこのクラスの定義を外部に公開することを示している。逆に、公開しない場合は、private というキーワードを使う。

HelloServlet クラスは HttpServlet クラスの doGet という名前のメソッドを上書き (**オーバーライド**) している。クラスを継承する時は元のクラス (スーパークラス) のメソッドを上書きすることもできるし、新しいメソッドやフィールドを加えることもできる。doGet クラスの定義の前の行の @Override は JDK5.0 から導入された**オーバーライド アノテーション**というもので、スーパークラスのメソッドをオーバーライドすることを明示的に示すものである。これにより、スペリングミスなどによるつまらない (しかし発見しにくい) バグを減らすことができる。

参考: クラス名に使える文字の種類 Java では、クラス名に次の文字が使える (変数名、メソッド名なども同じ。) このうち数字は先頭に用いることはできない。

アンダースコア (“_”), ドル記号 (“$”), アルファベット (“A”〜“Z”, “a”〜“z”), 数字 (“0”〜 “9”), かな・漢字など (Unicode 表 0xc0 以上の文字)

Java は C 言語と同じようにアルファベットの大文字と小文字は、**区別する**。その他にクラス名は大文字から始める、などのいくつかの決まりとまでは言えない習慣がある。`public` や `void`, `for`, `if` のように Java にとって特別な意味がある単語（**キーワード**）はクラス名などには使えない。

ドル記号とかな・漢字を用いることができるところが C や C++との違いである。

0.6 メソッド 呼び出し

Java ではオブジェクトのメソッドを呼び出すために、

オブジェクト**.メソッド名(引数**$_1$**, ... , 引数**$_n$**)**

という形を用いる。また、フィールド（インスタンス変数）をアクセスするときは、

オブジェクト**.フィールド名**

という書き方を用いる。前述したようにオブジェクトはいくつかのデータをまとめて一つの部品として扱えるようにした物であり、`.`（ドット）演算子は、オブジェクトの中から構成要素を取り出す演算子である。つまり、`out.println( ... )` は、`out` という `PrintWriter` クラスのオブジェクトから `println` というメソッドを取り出して引数を渡す式である。（Java のメソッドは必ずクラスの中で定義されている。そのため、同じオブジェクトのメソッドを呼出すなど特別な場合をのぞき、Java のメソッド呼出しには、このドットを使った記法が必要である。メソッドのドキュメントにはこの部分は明示されないので注意が必要である。）

> **参考:** `.`（ドット）演算子の前に書く値も、メソッド名の後の括弧の間に `,`（コンマ）区切りで書く値も、どちらもメソッドに渡されるデータという意味では違いはないが、上述のようにイメージが異なる。`.` 演算子の前にあるのは “主語” で、括弧の間にある通常の引数は “目的語” のようなイメージである。
>
> メソッドはクラスの中に定義されているので、同じ名前のメソッドが複数のクラスで定義されていて、同じ名前のメソッドでもクラスが異なれば実装が異なることがある。`.` 演算子の前のオブジェクトが、どのメソッドの実装を呼び出すかを決定する。

0.7 変数の宣言

変数の宣言は C と同様、

型名 変数名**;**

の形式で行なう。型名は `int`, `double` などのプリミティブ型か、クラス名である。ただし、C と違って、使用する前に宣言すれば必ずしも関数定義の最初に宣言する必要はない。変数への代入も C と同様**= 演算子**を使う。

0.8 フィールドの宣言

フィールド（インスタンス変数）の宣言は、クラス定義の中に、メソッドの定義と同じレベルに（メソッドの定義の外に）並べて書く。フィールドはそのクラス中のすべてのメソッドから参照することができる。（ある意味で C 言語の大域変数と似ている。）

メソッドの中で**自分自身のフィールドやメソッド (スーパークラスで定義されているものも含む) を参照する時**は、ピリオドを使った記法は必要ない。

フィールドはオブジェクトが存在している間は値を保持している。これに対して、メソッドの中で宣言された変数の寿命はそのメソッドの呼出しの間だけである。2 度め以降の呼び出しでも以前の値は保持していない。

Java のコメント　Java のコメントには C と同じ形式の **"/*"** と **"*/"** の間、という形の他にも、上の例のように **"//"** から行末まで、という形式も使える。（C++と同じ。最近の C の仕様（C99）でも //〜形式のコメントが使えるようになっている。）

0.9 クラスフィールドとクラスメソッド

クラスフィールドは**クラスに属するオブジェクトから共通にアクセスされる変数**であり、クラスメソッドは**フィールド（インスタンス変数）にアクセスせず、クラスフィールドだけにアクセスするメソッド**である。どちらも、クラスによって決まるので、.（ドット）演算子の左にクラス名を書くことによってアクセスできる。以前に登場した System.out も System（正確に言うと java.lang.System）というクラスの out という名前のクラスフィールドである。

クラスメソッド・クラスフィールドのことを、それぞれスタティックメソッド・スタティックフィールドと呼ぶこともある。これは、クラスフィールドやクラスメソッドを定義する時に **static** という修飾子をつけるためである。API 仕様のドキュメントにも static と付記される。例えば、Math クラスのドキュメントの中では、

```
static double cos(double a)
```

のように説明されている。これは使用するときには、**Math.cos(0.1)** のようにクラス名. メソッド名の形に書かなければいけないということを示している。

> **参考:** Java 5.0 以降では static import という仕組みを利用することで、クラスフィールド・メソッドの前のクラス名を省略することができるようになった。例えば、プログラムの先頭に、
>
> ```
> import static java.lang.Math.cos;  // cos 関数だけの場合、
> // または
> import static java.lang.Math.*;    // Math クラスのすべてのクラスフィールド
> ```
>
> と書くと、単に cos(0.1) のように書くことができる。

0.10　インスタンスの生成

一般に、あるクラスのインスタンスを生成するには、**new** という演算子を使う。new の次に**コンストラクタ**（constructor）という、クラスと同じ名前のメソッドを呼び出す式を書く。コンストラクタに必要な引数は各クラスにより異なるので API ドキュメントを調べる必要がある。また、ひとつのクラスが引数の型が異なる複数のコンストラクタを持つ場合もある。

File クラスの場合、コンストラクタ（のなかの一つ）はファイルのパスを表す String 型の引数をとる。例えば、new File("/home/foo/bar.txt")のように書く。

0.11　配列の宣言

```
int[] xs = {100, 137, 175, 175, 137, 100};
```

は、C 言語では

```
int xs[] = {100, 137, 175, 175, 137, 100};
```

と書くべきところだが、Java ではどちらの書き方（[] の位置に注意）も可能である。[] は型表現の一部であるということを強調するため、Java では前者の書き方をすることが望ましい。

また、Java では、配列オブジェクトの **length** というフィールド (?) によって配列の大きさ（要素数）を知ることができる。これも C 言語と異なる点である。

0.12　文字列（String）に関する演算子とメソッド

Java では、**+演算子**を用いて String 型と String 型のオブジェクトを連接する（あるいは、String 型と int 型のオブジェクトを String 型に変換したものを連接する）ことができる。

例:

```
System.out.println("2+2 は" + (2+2));
System.out.println("2+3 は" + (2+3) + "です。");
```

一方、JDK 5.0 からは C 言語のような書式指定を行う printf や sprintf メソッドに相当するメソッドも使用できる。上の println の場合、printf というメソッドを使って、次のように書くこともできる。

```
System.out.printf("2+2 は%d%n",2+2);
System.out.printf("2+3 は%d です。%n", 2+3);
```

また、**String.format** は書式指定を行なって（出力せずに）String 型のオブジェクトを生成するクラスメソッド（java.lang.String クラスのクラスメソッド）である。

printf や format のように可変個の引数を持つメソッドは API のドキュメントでは、

```
static String format(String format, Object... args)
```

のように... を使って表される。

Integer.parseInt は文字列から整数に変換するためのメソッド（java.lang.Integer クラスのクラスメソッド）である。

```
static int parseInt(String s);
```

0.13　総称クラスの使用

総称クラス（generic class）は、型パラメータを持つクラスのことで、JDK5.0 から導入された。代表的な総称クラスの例として ArrayList, HashMap, LinkedList などがあげられる。型パラメータは**<と>の間に**書かれる。

ArrayList は**サイズの変更が可能な配列**である。ArrayList の型パラメータは要素の型を表す。（総称クラスはこのようにコレクション（データの集まり）の型に使われることが多い。）例えば、String 型を要素とする ArrayList はArrayList<String>となり、次のように使用する。

```
ArrayList<String> arr1 = new ArrayList<String>();    // 空の ArrayList 作成
arr1.add("aaa");  arr1.add("bbb");  arr1.add("ccc"); // データ追加
String s = arr1.get(1);                              // データ取出し
```

add メソッドでデータを追加し、get メソッドでデータを取り出すことができる。

int, double のようなプリミティブ型は総称クラスの型パラメータになることができないという制限があるので注意が必要である。このときは Integer, Double などの対応する**ラッパークラス**と呼ばれるクラスを利用する。

Java の主なプリミティブ型とラッパークラスとの対応を以下に挙げる。

プリミティブ型	ラッパークラス
int	Integer
char	Character
double	Double
boolean	Boolean

（ここに挙げている以外のプリミティブ型に対応するラッパークラスは単にプリミティブ型の先頭の文字を大文字にすれば良い。）

ラッパークラスとプリミティブ型の変換はほとんどの場合、自動的に行われる（オートボクシング）ので、int の代りに Integer と書く以外は通常のクラス型をパラメータとするときと変わらない。例えば次のように書くことができる。

```
ArrayList<Integer> arr2 = new ArrayList<Integer>();  // 空の ArrayList 作成
arr2.add(123);  arr2.add(456);  arr2.add(789);       // データ追加
int i = arr2.get(1);                                 // データ取出し
```

ArrayList<String>に int 型の要素を add したり、ArrayList<Integer>から String 型の要素を get したりするのは、当然型エラー（コンパイル時のエラー）になる。

```
ArrayList<String> arr1 = new ArrayList<String> ();
arr1.add(333);          // 型エラー

ArrayList<Integer> arr2 = new ArrayList<Integer> ();
...
String t = arr2.get(2)  // 型エラー
```

このような型エラーをコンパイル時にちゃんと発見したい、というのが、総称クラスの導入のそもそもの動機である。

API ドキュメントの中では、型パラメータは E のような仮のクラス名が使われ、

java.util.ArrayList<E>クラス:

public boolean add(E e)

リストの最後に、指定された要素（e）を追加する。

public E get(int index)

リスト内の指定された位置（index）にある要素を返す。

のように書かれる。

例題 0.13.1 *ArrayList* クラス

ファイルから読み込んだデータの保存に ArrayList を使用する例である。init メソッドでファイルから読み込んだデータを保存し、doGet メソッドでそれを利用している。なお、配列型も総称クラスの型パラメータとして問題なく使用することができる。この例の場合、ファイルの行数が前もってわからないので、配列ではなく ArrayList を使用している。

また、この例では doGet メソッドの中で、拡張 **for** 文（**for-each** 文）を使用している。

ファイル ArrayListTest.java

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.ArrayList;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
public class ArrayListTest extends HttpServlet {
```

```
  private ArrayList<String[]> questions;

  @Override
  public void init(ServletConfig config) throws ServletException {
    questions = new ArrayList<String[]>();
    try {
      File f = new File(config.getServletContext().getRealPath("/WEB-INF/quiz.txt"));
      BufferedReader in = new BufferedReader(
                              new InputStreamReader(
                                 new FileInputStream(f), "Windows-31J"));
      String line="";
      while((line=in.readLine())!=null) {
        line = line.trim();
        if(line.equals(""))
                continue;
        questions.add(line.split("¥¥s+"));
      }
      in.close();
    } catch (IOException e) {
    }

  }
```

```
  @Override
  public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
    response.setContentType("text/html; charset=Windows-31J");
    PrintWriter out = response.getWriter();
    out.println("<html><head></head><body>");
    out.println("<table border='1'>");
    out.println("<tr><th>問</th><th>1</th><th>2</th><th>3</th><th>答</th></tr>");
    for (String[] line : questions) {
      out.printf("<tr><td>%s</td><td>%s</td><td>%s</td><td>%s</td><td>%s</td></tr>%n",
                 line[0], line[1], line[2], line[3], line[4]);
    }
    out.println("</table>");
    out.println("</body></html>");
    out.close();
  }
```

```
}
```

例題 0.13.2 *HashMap* クラス

HashMap は**連想配列**と呼ばれるデータ構造である。通常の配列と異なり、int 型だけではなく、任意の型（String 型など）をキー（添字）として、値を格納・検索することができる。HashMap の型パラメータは 2 つあり、1 つめがキーの型、2 つめが値の型である。下の例では、HashMap<String, Integer>、つまりキーが String 型で値が Integer 型の連想配列を用いている。値の格納には put メソッド、検索には get メソッドを用いる。

java.util.HashMap<K,V>クラス:

public V put(K key, V value)

指定された値（value）と指定されたキー（key）をこのマップに関連付ける

public V get(Object key)

指定されたキー（key）がマップされている値を返す。

Object（java.lang.Object）クラスは Java のすべてのクラスのスーパークラスとなる、クラス階層のルートクラスである。

ファイル HashMapTest.java

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.HashMap;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
public class HashMapTest extends HttpServlet {
```

```
  HashMap<String, Integer> colors;

  @Override
  public void init() throws ServletException {
    colors = new HashMap<String, Integer>();
    colors.put("鴇", 0xf7acbc); colors.put("赤", 0xed1941);
    colors.put("朱", 0xf26522); colors.put("桃", 0xf58f98);
    ...  // 省略
  }
```

```
  @Override
  public void doPost(HttpServletRequest request, HttpServletResponse response)
           throws ServletException, IOException {
    request.setCharacterEncoding("Windows-31J");
    String cn = request.getParameter("colorName");
    Integer cv = colors.get(cn);

    response.setContentType("text/html; charset=Windows-31J");
    PrintWriter out = response.getWriter();
    out.println("<html><head></head><body>");

    if(cv!=null) {
      out.printf("%s 色は<span style='color:#%06x'>こんな色</span>です。", cn, cv);
    } else {
      out.printf("%s 色は見つかりません。", cn);
    }
    out.println("</body></html>");
    out.close();
  }
```

```
}
```

例題 0.13.3 *LinkedList* クラス

LinkedList は ArrayList と同じように要素を付け足していくことができるコレクションの型だが、先頭からも末尾からも要素を追加したり削除したりできる。

ファイル LinkedListTest.java

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.LinkedList;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
public class LinkedListTest extends HttpServlet {
```

```
  private int i=1;
```

```
  @Override
  public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
    response.setContentType("text/html; charset=Windows-31J");
    try {
      // デバッグ用
      i = Integer.parseInt(request.getQueryString());
    } catch (Exception e) {}

    PrintWriter out = response.getWriter();
    out.println("<html><head></head><body>");
    LinkedList<Integer> xs = new LinkedList<Integer>();
    int j=i;
    while(j>0) {
      xs.addFirst(j%10);
      j/=10;
    }
    out.printf("あなたは ");
    for(int k : xs) {
      out.printf("<img src='Images/%d.png' alt='%d'>", k, k);
    }
    out.printf("番目の来訪者です。%n");
    out.printf("</body></html>");
    out.close();    // close を忘れない
    i++;
  }
}
```

キーワード :

Java、C、C++、オブジェクト指向、アプレット、中間言語方式、サーブレット、オブジェクト、クラス、インスタンス、フィールド（インスタンス変数）、メソッド、class、import、継承、extends、オーバーライド、クラスフィールド（クラス変数）、クラスメソッド、new 演算子、 コンストラクタ、配列、length メソッド、+演算子、String.format メソッド、Integer.parseInt メソッド、総称クラス、ArrayList クラス、HashMap クラス、LinkedList クラス