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 社から公表されたプログラミング言語である。（Sun Microsystems 社は 2010 年に Oracle Corporation に吸収合併された。）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 (applet) 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ー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ーに悪戯をすることができない、ということである。もし、Web ページに任意のプログラムを埋め込んでブラウザー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、ポインター（アドレス）演算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 はポインター演算を明示的に提供せず、型変換をきちんとチェックするなど、このような抜け道がないよう設計されている。このような設計は安全性だけではなく、プログラムのバグを未然に防ぐためにも有用である。

> しかし、アプレットでファイル操作やネットワーク操作などがまったくできないというのでは困る場合もある。アプレットの作成者の署名を付加し、そこで作成者が明確なアプレットにファイル操作などを許す**署名つき（signed）アプレット**という仕組みが用意されている。

> さらに、2014 年 1 月の Java 7 update 51 から、ファイル操作やネットワーク操作の有無にかかわらず、Java アプレットに署名が必須となった。
>
> 基本的には、Java の処理系や基本ライブラリーが適切に設計されていれば、アプレットは安全に実行できるはずなのだが、Java の脆弱性（不具合）をついた不正アプレットが多く現れ対処しきれなくなったため、と考えられる。
>
> 注の注: さらに、2017 年にリリースが予定されている Java 9 から、Java アプレットをブラウザーで実行するための Java ブラウザープラグインが deprecated （非推奨）とされる見込みである。

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ー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。コンパイラー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、インタプリターを用いる方式では、各機種毎にインタプリター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 コンパイラーによって JVM (Java Virtual Machine, Java 仮想機械) 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 エミュレーター（一種のインタプリター）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。しかし、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ー側の仕組みとしては、JavaScript などが主流となって、Java アプレットは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 **サーバー側**で動作してウェブページなどを動的に生成するサーブレット（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。

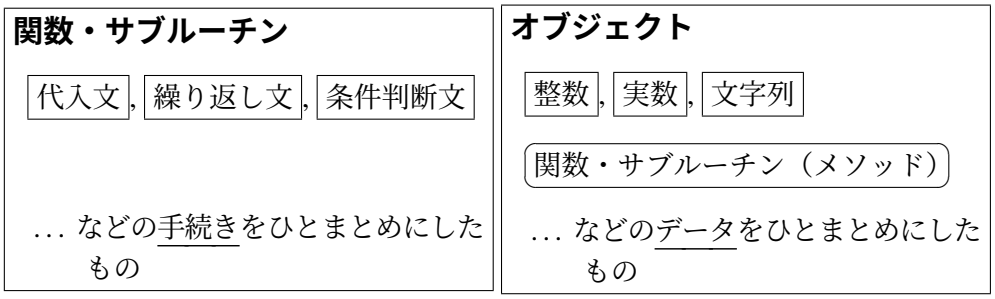
WWW サーバー側プログラム用のプログラミング言語としては、Perl, PHP, Python, Ruby なども有名だが、これらは動的型付けを採用している。つまり、実行時まで型エラーは検出しない。Java はこれらと違い静的型付けを採用している。つまり、実行前（コンパイル時）に型エラーを検出する。一般に静的型付けは大規模で信頼性が必要とされるシステムの記述に適している。

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パラダイム（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。

オブジェクト指向言語に限った話ではないが、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、まず、オブジェクト指向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。



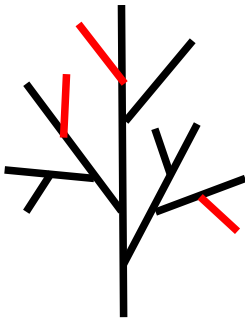
実際には、プログラム部品として提供されるのは、オブジェクトそのものではなく、オブジェクトの雛型とでもいうべき クラス （class）である。クラスは、そこから生成されるオブジェクトが（具体的なデータ（つまり、1 とか 3.14）ではなく）どのような名前と型の構成要素を持つか、のみを指定したものである。クラスを具体化（instantiate — つまり、`x` という名前の `int` 型の構成要素は 1 で、`y` という名前の `float` 型の要素は、3.14 などと定めること）したものがオブジェクトである。このとき、このオブジェクトはもとのクラスの インスタンス （instance, 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

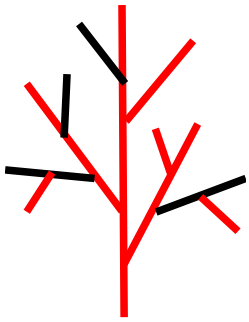
> 正確に言えば、メソッドについてはインスタンスごとにコードを定義するのではなく、クラスごとにコードを定義する（ようになっているオブジェクト指向言語が多い）。ただし、メソッドから参照されるフィールドはインスタンス毎に異なるものである。オブジェクトは各フィールドのデータの他に、どのクラスに属しているか、という情報を持っていて、それによって適切なメソッドのコードが起動される。

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、オブジェクト指向言語では、それに加えて既存の部品（つまりクラス）に少しだけ機能を追加したり、一部を置き換えたりする（継承、インヘリタンス, inheritance）、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。



（赤色が再利用できる部分）
手続き型言語のイメージ



（赤色が再利用できる部分）
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 (Graphical User Interface) 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

```
1 public class Hello0 {
2   public static void main(String args[]) {
3     System.out.printf("Hello␣World!%n");
4   }
5 }
```

`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。（クラスなど、オブジェクト指向の概念の詳しい説明は、後述する。ただし、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。この後の開ブレース（{）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。

> **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（“_”）, アルファベット（“A”〜“Z”, “a”〜“z”）, 数字（“0”〜 “9”）, ドル記号（“$”）, 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。

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。C 言語と違い、String は char の配列ではない。しかし、文字列リテラル（String 型の定数）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変換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 などの変換指定は 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 の変換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。

[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。

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

```
1  import java.io.IOException;
2  import java.io.PrintWriter;
3
4  import javax.servlet.ServletException;
5  import javax.servlet.annotation.WebServlet;
6  import javax.servlet.http.HttpServlet;
7  import javax.servlet.http.HttpServletRequest;
8  import javax.servlet.http.HttpServletResponse;
9
10 @WebServlet("/HelloServlet")
11 public class HelloServlet extends HttpServlet {
12   @Override
13   protected void doGet(HttpServletRequest request,
14                        HttpServletResponse response)
15                   throws ServletException, IOException {
16     response.setContentType("text/html;␣charset=UTF-8");
17     PrintWriter out = response.getWriter();
18     out.println("<html><head></head><body>");
19     out.println("Hello␣World!");
20     out.println("</body></html>");
21     out.close();
22   }
23 }
```

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。

> **詳細:**パッケージ（package）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。ただし、java.lang という基本的なクラスを集めたパッケージは import しなくてもクラス名だけで使用できる。例えば String クラスは java.lang パッケージに属する。

@WebServlet("/HelloServlet") は Servlet の URI のパスを指定するためのアノテーションである。クラスファイルに/HelloServlet というパスの情報が書き込まれ、コンテナーがその情報を読み出して、そのパスに Servlet を配備する。

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。

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。

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、**"//"** 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（クラス変数）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、通常のフィールド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ときに 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、Color クラスのドキュメントの中では、

```
static Color BLACK
```

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。これはそれぞれ使用するときには、Color.BLACK, Math.cos(0.1) のようにクラス名.メソッド名（またはフィールド名）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・クラスメソッド
> ```
>
> 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 コンストラクター（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。コンストラクター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ーを持つ場合もある。

File クラスの場合、コンストラクター（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

配列の宣言の

```
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　演算子

Java の持つ演算子は、C とほとんど同じである。

算術演算子

整数 / 整数

という演算では、整数としての割算（小数点以下は切捨て）としての結果が得られる。

整数 % 整数

では、余りを求める。

```
System.out.printf("%d␣/␣%d␣=␣%d\n", 5, 3, 5/3);
System.out.printf("%d␣%␣%d␣=␣%d\n", 8, 5, 8%5);
```

等価演算子　==は両辺の値が等しければ真を、等しくなければ偽を返す演算子である。==と逆に等しくないかどうかを判定する演算子は!=である。

後置増分演算子・前置増分演算子

a++	a の値を一つだけ増やす	（式全体の値は、増加前の変数の値）
a--	a の値を一つだけ減らす	（式全体の値は、減少前の変数の値）
++a	a の値を一つだけ増やす	（式全体の値は、増加後の変数の値）
--a	a の値を一つだけ減らす	（式全体の値は、減少後の変数の値）

0.13 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 型などのオブジェクトを 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。");
System.out.println(2 + " の二乗は " + (2 * 2) + " 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);
System.out.printf("%d の二乗は %d です。", 2, 2 * 2);
```

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。

変換指定のまとめ System.out.printf や String.format の第 1 引数のなかで、%から始まる部分は変換指定と言い、第 2 引数以降の値に順に置き換えられる。整数（10 進数）は %d、浮動小数点数は%f、文字列は%s を使う。

```
System.out.printf("半径␣%d␣の円の周長は␣%f\n", r, 2 * r * 3.14);
```

高度な変換指定 以下のような変換指定は必要に応じて調べれば良い。

説明	例	出力
桁数揃え	System.out.printf("[%3d]", 1)	[1]
桁数揃え（先頭を 0）	System.out.printf("[%03d]", 1)	[001]
小数点以下の桁数指定	System.out.printf("[%.3f]", 1.0/3)	[0.333]
16 進数で表示 (小文字)	System.out.printf("[%x]", 127)	[7f]
16 進数で表示 (大文字)	System.out.printf("[%X]", 127)	[7F]

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、

```
public 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。

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

0.14　総称クラスの使用

総称クラス（generic class）は、型パラメーター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, ArrayDeque などがあげられる。型パラメーターは<と>の間に書かれる。

ArrayList はサイズの変更が可能な配列のようなものである。（通常の配列と異なり、各要素の定数時間のアクセスはできない。）ArrayList の型パラメーターは要素の型を表す。（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。

```
// コンストラクターは空の ArrayList 作成
ArrayList<String> arr1 = new ArrayList<String>();
// データ追加
arr1.add("aaa");  arr1.add("bb");  arr1.add("cccc");
// データ取出し
String s = arr1.get(1);
```

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。ArrayList の無引数のコンストラクターは空の ArrayList を生成する。本来は総称クラスのコンストラクターは型パラメーターが必要だが、Java 8 からは文脈から推論できる場合、次のように省略できるようになった.

```
ArrayList<String> arr1 = new ArrayList<>();
```

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ー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、型パラメーターとして 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ー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。

```
// コンストラクターは空の ArrayList 作成
ArrayList<Integer> arr2 = new ArrayList<>();
// データ追加
arr2.add(1);  arr2.add(234);  arr2.add(56789);
// データ取出し
int i = arr2.get(1);
```

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<> ();
arr1.add(333);          // 型エラー

ArrayList<Integer> arr2 = new ArrayList<> ();
...
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、型パラメーターは E のような仮のクラス名が使われ、

> java.util.ArrayList<E>クラス:
>
> public ArrayList()
>
> 空のリストを作成します。
>
> public boolean add(E e)
>
> リストの最後に、指定された要素（e）を追加します。
>
> public E get(int index)
>
> リスト内の指定された位置（index）にある要素を返します。

のように書かれる。

例題 0.14.1 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ー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。

for (型 変数名 : 式) 文 [1]

このかたちの **for** 文は JDK5.0 で導入されたものである。この場合、式は直感的には何かの集まりを表すデータ型（配列など — 正確には配列またはインタフェース Iterable を実装するクラス）でなければならない。コロン（:）の前で宣言された変数に、この列の要素が順に代入され、文の実行が繰り返される。

ファイル ArrayListTest.java

```
1 import java.io.BufferedReader;
2 import java.io.File;
3 import java.io.FileInputStream;
4 import java.io.IOException;
```

```
 5 import java.io.InputStreamReader;
 6 import java.io.PrintWriter;
 7 import java.util.ArrayList;
 8
 9 import javax.servlet.ServletConfig;
10 import javax.servlet.ServletException;
11 import javax.servlet.annotation.WebServlet;
12 import javax.servlet.http.HttpServlet;
13 import javax.servlet.http.HttpServletRequest;
14 import javax.servlet.http.HttpServletResponse;
15
16 @WebServlet("/ArrayListTest")
17 public class ArrayListTest extends HttpServlet {
18   private ArrayList<String[]> questions;
19
20   @Override
21   public void init(ServletConfig config) throws ServletException {
22     questions = new ArrayList<>();
23     try {
24       File f = new File(config.getServletContext()
25                         .getRealPath("/WEB-INF/quiz.txt"));
26       BufferedReader in = new BufferedReader(new InputStreamReader(
27                           new FileInputStream(f), "UTF-8"));
28       String line="";
29       while ((line=in.readLine())!=null) {
30         line = line.trim();
31         if (line.equals(""))
32           continue;
33         questions.add(line.split("\\s+"));
34       }
35       in.close();
36     } catch (IOException e) {
37     }
38   }
39
40   @Override
41   protected void doGet(HttpServletRequest request,
42                        HttpServletResponse response)
43                 throws ServletException, IOException {
44     response.setContentType("text/html;␣charset=UTF-8");
45     PrintWriter out = response.getWriter();
46     out.println("<html><head></head><body>");
47     out.println("<table␣border='1'>");
48     out.println("<tr><th>問</th><th>1</th><th>2</th><th>3</th><th>答</th></tr
49     for (String[] line : questions) {
50       out.printf("<tr>");
51       for (String item : line) {
52         out.printf("<td>%s</td>", item);
53       }
```

```
54        out.printf("</tr>%n");
55      }
56      out.println("</table>");
57      out.println("</body></html>");
58      out.close();
59    }
60  }
```

例題 0.14.2 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 の型パラメーターは 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 HashMap()
空の HashMap を作成します。
  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ます。
  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します。

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

```
1  import java.io.IOException;
2  import java.io.PrintWriter;
3  import java.util.HashMap;
4
5  import javax.servlet.ServletException;
6  import javax.servlet.annotation.WebServlet;
7  import javax.servlet.http.HttpServlet;
8  import javax.servlet.http.HttpServletRequest;
9  import javax.servlet.http.HttpServletResponse;
10
11 @WebServlet("/HashMapTest")
12 public class HashMapTest extends HttpServlet {
13   HashMap<String, Integer> colors;
14
15   @Override
16   public void init() throws ServletException {
17     colors = new HashMap<>();
```

```
18      colors.put("鴇", 0xf7acbc); colors.put("赤", 0xed1941);
19      colors.put("朱", 0xf26522); colors.put("桃", 0xf58f98);
20      //途中省略
21      colors.put("桔梗", 0x5654a2); colors.put("薔薇", 0xe9546b);
22    }
23
24    @Override
25    protected void doPost(HttpServletRequest request,
26                          HttpServletResponse response)
27                    throws ServletException, IOException {
28      request.setCharacterEncoding("UTF-8");
29      String cn = request.getParameter("colorName");
30      Integer cv = colors.get(cn);
31
32      response.setContentType("text/html; charset=UTF-8");
33      PrintWriter out = response.getWriter();
34      out.println("<html><head></head><body>");
35
36      if(cv!=null) {
37        out.printf("%s色は<span style='color:#%06x;'>こんな色</span>です。",
38                    cn, cv);
39      } else {
40        out.printf("%s色は見つかりません。", cn);
41      }
42      out.println("</body></html>");
43      out.close();
44    }
45  }
```

例題 0.14.3 ArrayDeque クラス

ArrayDeque 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 ArrayDequeTest.java

```
1   import java.io.IOException;
2   import java.io.PrintWriter;
3   import java.util.ArrayDeque;
4
5   import javax.servlet.ServletException;
6   import javax.servlet.annotation.WebServlet;
7   import javax.servlet.http.HttpServlet;
8   import javax.servlet.http.HttpServletRequest;
9   import javax.servlet.http.HttpServletResponse;
10
11  @WebServlet("/ArrayDequeTest")
12  public class ArrayDequeTest extends HttpServlet {
13    private int i=1;
14
```

```
15   @Override
16   protected void doGet(HttpServletRequest request,
17                        HttpServletResponse response)
18                 throws ServletException, IOException {
19     response.setContentType("text/html;␣charset=UTF-8");
20     try {
21       // デバッグ用
22       i = Integer.parseInt(request.getQueryString());
23     } catch (Exception e) {}
24
25     PrintWriter out = response.getWriter();
26     out.println("<html><head></head><body>");
27     ArrayDeque<Integer> xs = new ArrayDeque<>();
28     int j=i;
29     while(j>0) {
30       xs.addFirst(j%10);
31       j/=10;
32     }
33     out.printf("あなたは␣");
34     for(int k : xs) {
35       out.printf("<img␣src='Images/%d.png'␣alt='%d'␣/>", k, k);
36     }
37     out.printf("番目の来訪者です。%n");
38     out.printf("</body></html>");
39     out.close();    // close を忘れない
40     i++;
41   }
42 }
```

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、 コンストラクター、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、ArrayDeque クラス