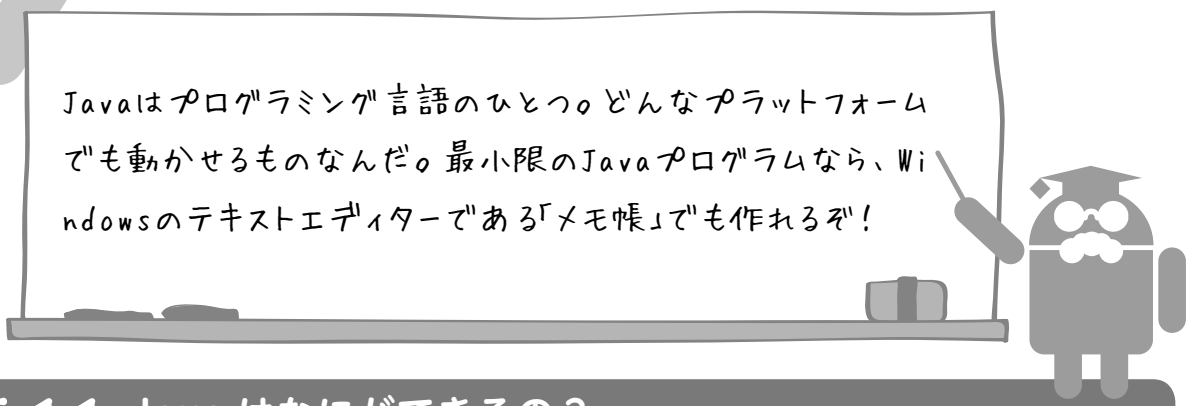
第5章

JAVAって何だ

著：柴田文彦

5-1 アプリ開発に必要なJavaってなに？

著：柴田文彦



5-1-1 Javaはなにができるの？

Java以前の一般的なプログラムは、一定のCPUやOSをターゲットとして開発されていました。オブジェクトコードは、それを実行するマシンのCPUの機械語になっていました。またマシンごとに利用できる機能が異なるため、それ以外の環境で実行させるには、ソースコードの段階で多くの変更を加える必要がありました。

どんなマシンでも動くプログラムを作れる！

プログラミング言語としての「Java」は、1990年代に、今はオラクルに吸収されたサン・マイクロシステムズで開発されたものです。基本的な文法は、当時ポピュラーだったC言語やC++（プラスプラス）のものを受け継いでいます。一種のオブジェクト指向言語ですが、オブジェクト指向については、Objective-C（iPhoneアプリの開発言語）や、その祖先とも言えるSmalltalkの、比較的シンプルな機能を取り入れて設計されたものです。

Javaの開発の動機は、実行環境（プラットフォーム）に依存しないプログラミング環境を実現することにあったと言われています。Javaは、特に実行環境を想定せずに書いたプログラムが、Javaをサポートする様々な実行環境の上で動作することを目標として開発されました。これを「Write Once, Run Anywhere」（一度書いたらどこでも走る）という標語を用意して強調していました（**図1**）。

また、1990年代は、インターネットが急激に普及した時期でもありました。それ以前のアプリケーションプログラムは、フロッピーディスクやCD-ROMといった記録メディアに収納して流通させ、それをパソコンのユーザーが自分のマシンに自らインストールすることで使用可能になるものがほとんどでした。Javaは、インターネットを通してアプリケーションを流通させ、ユーザーは特にインストール作業を意識することなく、ウェブブラウザー上やデスクトップでアプリケーションが利用可能になるような仕組みを備えていました。ただし現在では、様々な理由から、このようなインターネットを通してJavaプログラムを流通させることは敬遠されるようになっています。

ちなみにJavaというネーミングですが、この言葉の本来の意味は、インドネシアのジャワ島（Jawa）のことです。そこから、その島で産出されるジャワ種のコーヒーのことをJavaと呼ぶようになりました。特にアメリカではJavaはコーヒーの別名として一般

的に使われています。そしてプログラミング言語のJavaは、そこから名付けられたとされています。Javaのロゴにコーヒーカップが描かれているのもそのためです。

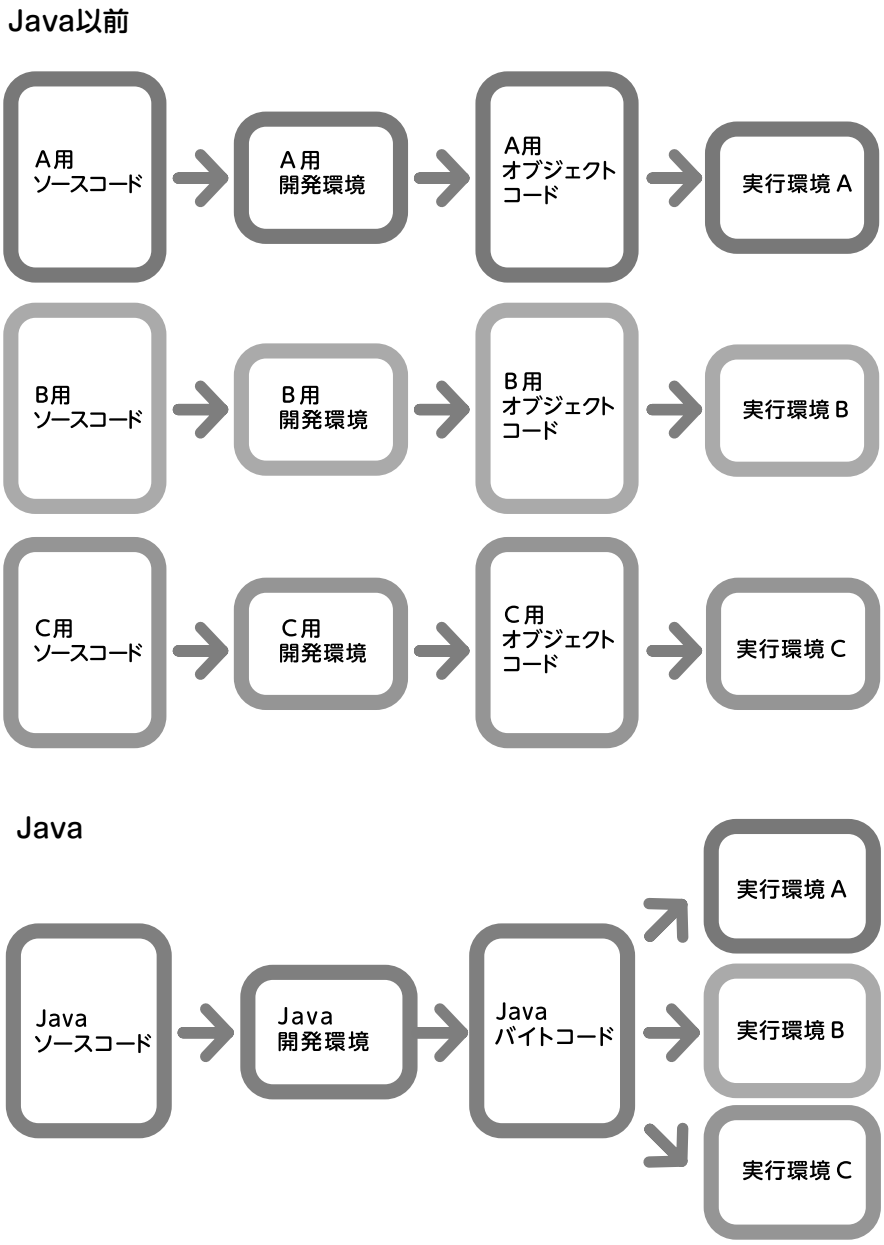


図1：Java以前のプログラミング環境では、実行環境ごとに異なるソースコードを用意し、それぞれ異なる開発環境を使って、専用のプログラムを作ることを基本としていた（上）。Javaでは、1つのソースコードから様々な実行環境で動作するプログラムを作成することができる（下）

Javaプログラムの動作のしくみ

Javaは一種のコンパイラー型言語ということになっています。Javaで書かれたプログラムを実行するには、通常は事前にコンパイルすることが必要です。そのためには、JDKに含まれるJava専用のコンパイラーである「javac」を使います。すでに述べたようにJavaのコンパイラーは、Javaのソースコードを「Javaバイトコード」に変換します。これは一種の中間コードで、そのままではCPUの上で実行することはできません。CPUや機種に依存しないということは、それを直接実行できる環境が存在しないということでもあるからです。

コンパイラー型とは、実行する前にJavaソースコードをまとめてコンパイルして、実行コードに変換してから実行するという意味です。しかし、そのように実行前にコンパイルが必要かどうかということは、言語としての特徴には直接関係ありません。それはその言語に対してどのような処理系が用意されているか、というプログラミング環境の問題です。

ライブラリーとは、アプリとして標準的な機能を実現するためにあらかじめ用意されているプログラム群で、すぐに実行可能な状態でOSの中に組み込まれています

Javaバイトコードは、実際にハードウェアとして存在するパソコンやスマートフォン等の上ではなく、仮想マシンと呼ばれる、一種のソフトウェアプラットフォームの上で実行されます。それが「Java VM（Virtual Machine）」です（**図2上**）。Java VMは、Javaプログラムを実行可能な様々なマシン上で動作しています。それぞれは機種ごとに異なるものですが、Javaのバイトコードを実行する機能についてはすべて共通です。Javaバイトコードは、あらかじめ用意されたJavaのライブラリーを利用しながら、Java VMの上で動作します。このライブラリーは、異なるマシン、異なるOS上でも同様の機能が提供できるように用意されています。

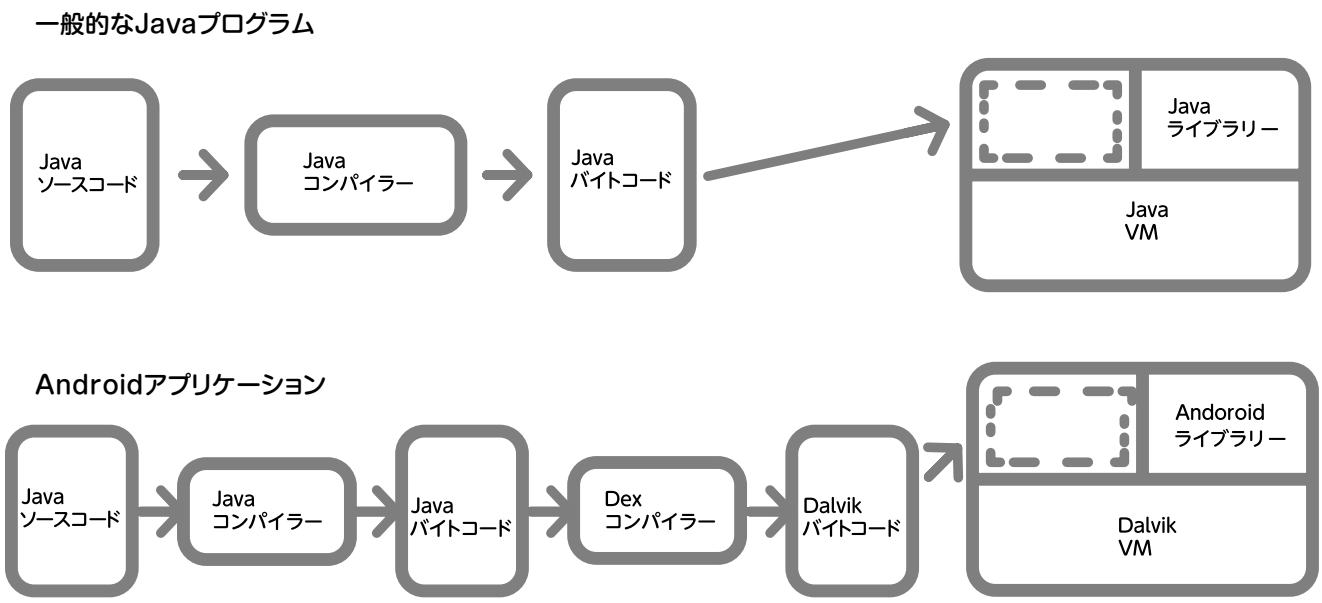


図2: Javaのソースコードは、バイトコードと呼ばれる中間コードにコンパイルされ、Java仮想マシン上で実行される（上）。Androidでは、Javaバイトコードをさらに Dalvikバイトコードにコンパイルし直し、Dalvid仮想マシン上で実行する（下）

Androidのアプリケーションも Javaで記述するからには、Javaコンパイラーを使ってバイトコードに変換します。しかしAndroidアプリの場合、そこで終わりではありません。Androidデバイスは、Java VMによってバイトコードをそのまま実行するわけではないのです。Androidの場合には、Javaバイトコードを、さらにもういちど「Dexコンパイラー」と呼ばれる別のコンパイラーに通して「Dalvik（ダルビック）バイトコード」に変換します。各Androidデバイス上には、一般のJava VMではなく、Dalvikと呼ばれる仮想マシンが装備されています。そしてDalvikバイトコードに変換されたAndroidアプリのプログラムは、Androidライブラリを利用しながら、このDalvik VMの上で動作するのです（**図2下**）。

実際にAndroidアプリを開発する際には、このような内部の変換過程を意識する必要はありません。Javaコンパイラーや Dexコンパイラーを手動で動かしてAndroidアプリを作成することも可能ですが、それには手間も時間もかかり、現実的ではありません。Androidアプリの開発に必要な一連の操作を自動化するために統合開発環境があるのです。いったん適切に設定された開発環境を用意すれば、あとは一般的なJavaのプログラムもAndroidアプリも、ほぼ違いなく扱うことができます。

なお、このようにソースコードからアプリを生成する過程は、単にコンパイルすれば完了というわけではなく、複数の段階にわかれた様々な工程を含んだものとなります。それらを総称してビルド（build）と呼ぶのが普通です。

5-1-2 Javaプログラムを実際に動かしてみよう

Javaプログラムの実行手順

まず、Javaによるプログラミングを学習するために最低限必要となる、Javaのソースコードのコンパイル方法と、コンパイル後のバイトコードの実行方法を示しておきましょう。最も原始的な方法を最初に学びます。

ここでは開発環境は使わずに、JDKに含まれているコマンドを使ってJavaのソースコードをバイトコードに変換し、さらに別のコマンドを使ってバイトコードを実行します(**図3**)。

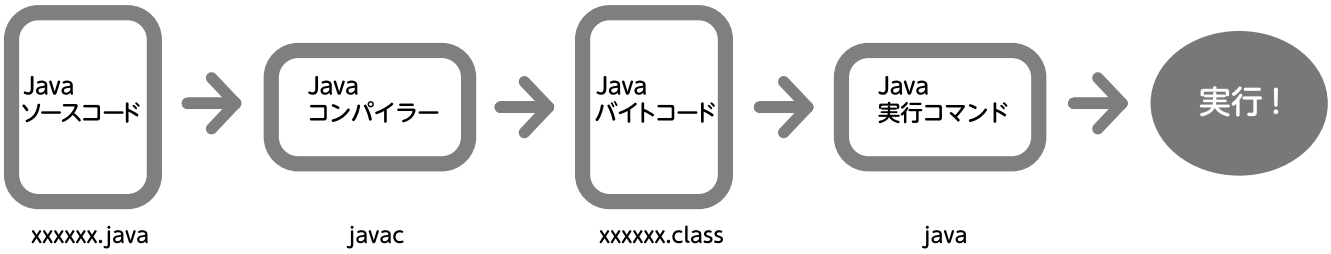


図3:コマンドとして起動するjavacを使ってJavaソースコードをコンパイルし、それによって生成されたJavaバイトコードをjavaコマンドを使って実行する

具体的に言えば、最初にJavaソースコードをバイトコードに変換するには「javac」というコマンドを使います。すでに述べたように、バイトコードそのままでは開発環境のOSの上で実行することはできません。そのためにはJava仮想マシンが必要となります。Javaのバイトコードを仮想マシン上で動作させるためのコマンドがjavaそのものです。

Javaソースコードを用意する

ここでは、とにかくJavaのソースコードを作成し、コンパイルして動作させることを最優先に、それを最小限の労力で実現する方法を考えてみます。そのために、Windowsの標準アプリケーション「メモ帳」を使ってJavaプログラムを書いてみましょう。

まずはメモ帳を起動するところから始めます。Windows 8.1では従来からあるWindowsのアプリを起動するのに、とかく手間がかかりがちになっています。ここでは、一種のショートカットを使ってメモ帳を起動する方法を使います。

まず、デスクトップの左下にある「スタート」ボタンを右クリックして開くメニューから、「ファイル名を指定して実行」を選びます(**図4**)。

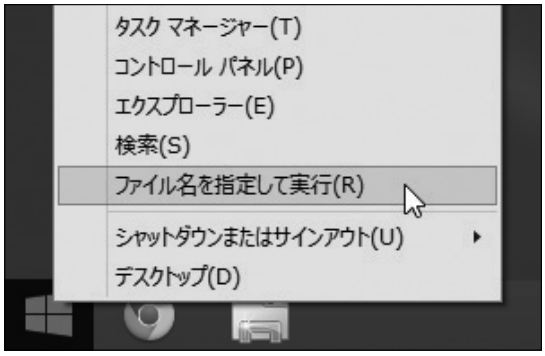


図4:「スタート」ボタンから「メモ帳」を起動するために、右ボタンクリックして「ファイル名を指定して実行」を選ぶ

javaコマンドは、すでに第4章の開発環境のセットアップで、JDKのインストールが正しく完了しているかどうかを確認するために一度起動しているはずです。もう一度確認すると、「スタート」ボタンを右クリックして、表示されるメニューから「コマンドプロンプト」を選び、同名のウィンドウを表示するのでした。このウィンドウの中にコマンドをタイプ入力すれば、そのコマンドで指定したプログラムを起動することができます。

すると「ファイル名を指定して実行」というダイアログが開くので、その中にある「名前:」欄に「notepad」とキーボードから入力し、「OK」ボタンをクリックします（**図5**）。

図5:「ファイル名を指定して実行」ダイアログに「notepad」とタイプ入力して「OK」ボタンをクリック

その結果メモ帳が起動し「無題 - メモ帳」という中身が空白のウィンドウが表示されます。その中に以下のようなプログラムをキーボードから打ち込みます。

```
class Hello {
        public static void main (String[] args) {
                System.out.println("Hello, " + args[0]);
        }
}
```

文字はすべて半角英数字です。単語と単語の間に隙間がある部分は、半角スペースを入れます。ただし、ウィンドウの左端から行の先頭までに隙間がある部分は、「Tab」キーを押してタブを入れます。2行目と4行目には1個、3行目には2個のタブが入っています。その他の行は、ウィンドウの左端からそのまま書き始めます。英字の大文字と小文字は区別します。

この短いソースコードの中にも括弧が3種類出てきていることに注意してください。登場順に中括弧「{}」、普通の括弧「()」、大括弧「[]」です。括弧も、もちろん半角です（**図6**）。

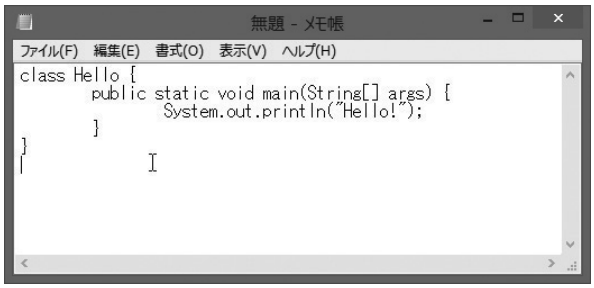


図6:メモ帳にこのようなソースコードを入力する。すべて半角英字で、大文字と小文字も区別する

ソースコードが入力できたら、これをファイルに保存します。初めて保存する際には「ファイル」メニューから「名前を付けて保存...」を選びます（**図7**）。

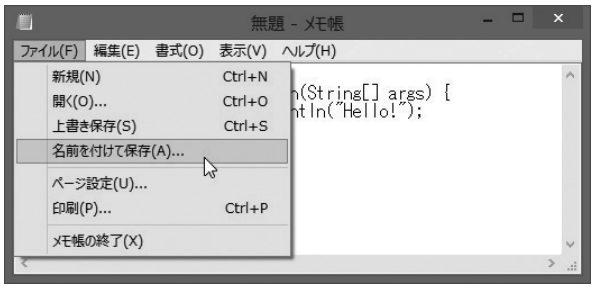


図7:メモ帳のウィンドウの「ファイル」メニューから「名前を付けて保存...」を選ぶ

すると、「名前を付けて保存」ダイアログが表示されるので、場所とファイル名を指定して保存します。ここでは「デスクトップ」に保存することにしました。ファイル名は「Hello.java」とします（**図8**）。拡張子は必ず「.java」にしておきましょう。

図8:「名前を付けて保存」ダイアログの左のコラムで「デスクトップ」をクリックして選択し、「ファイル名」に「Hello.java」とタイプしてから「保存」をクリック

その結果、デスクトップ上に「Hello.java」のアイコンが現れます(**図9**)。この時点ではアイコンは白紙の書類のアイコンになっているはずです。これでJavaのソースコードが準備できました。

図9:デスクトップ上に「Hello.java」としてソースコードが保存された

check!

エディターについて

通常は、「メモ帳」でプログラムを書くプログラマーはいません。後で実際に試してみるように、IDEに含まれるエディターを使うのが普通でしょう。プログラマーによっては、それでは満足できずにIDEとは独立したプログラミング用の高機能なエディターを使う人もいます。プログラミングという作業は、その大部分の時間、ソースコードを作成したり、編集したりすることになります。そのため、プログラマーにとってエディターは非常に重要なツールなのです。

コンパイルして実行する

作成したJavaのソースコードをコンパイルして実行するには、JDKに含まれるツールを使います。そのためには、開発環境のセットアップの際に試したように、「コマンドプロンプト」を使います。その起動方法は、第4章でも示した通り、「スタート」ボタンを右クリックして表示するメニューから選ぶだけです。

コマンドプロンプトを起動したら、作業用のディレクトリを、ソースコードのある場所に移動します。上で作成したソースコードはデスクトップ上に保存したので、デスクトップを作業用のディレクトリにしましょう。コマンドプロンプトに対して「cd Desktop」とタイプして「Enter」キーを押します(**図10**)。すると、コマンドプロンプトが示すカレントディレクトリ(現在のディレクトリ)が、デスクトップに移動したことが分かります。

「cd Desktop」とタイプ入力しただけで、カレントディレクトリがデスクトップに移動したのは、コマンドプロンプトが初期状態でユーザーのホームディレクトリをカレントディレクトリとしているからです。そしてデスクトップは、ユーザーのホームの下にある「Desktop」という名前のディレクトリで表されているのです。このような場所の指定方法は、いわゆる「相対パス」によるものです。どこから別の場所にカレントディレクトリを移動したい場合は、「C:¥」から始まる絶対パスによって指定する方が早い場合もあります。

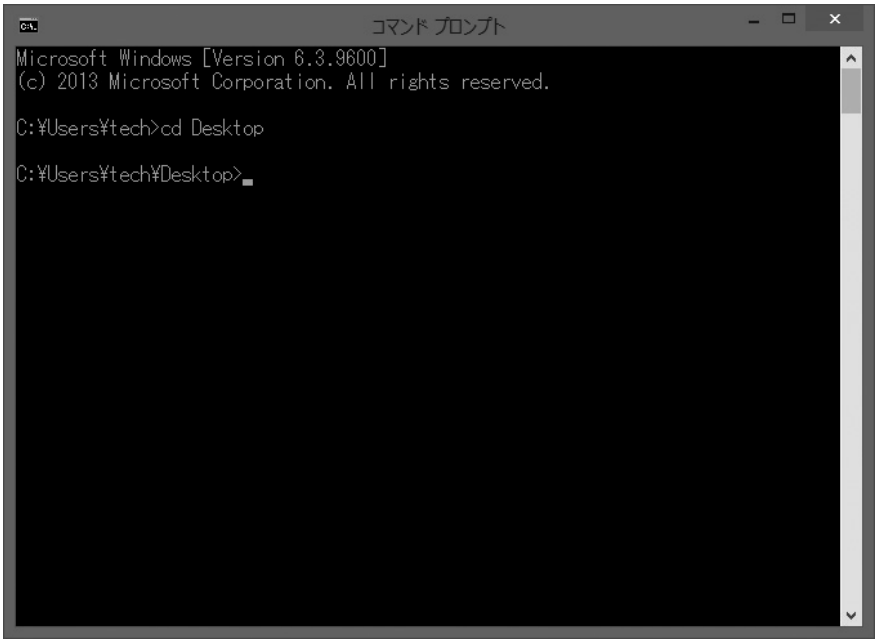


図10:コマンドプロンプトを起動し、「cd Desktop」とタイプして「Enter」キーを押す

カレントディレクトリがデスクトップに移動したら、さっそくその中にあるソースコードをコンパイルしましょう。「javac Hello.java」というコマンドをタイプし「Enter」を押します(**図11**)。

Javaのソースコードなのだから、拡張子は「.java」に決まっているはずだから、いちいち指定する必要はないだろうと思われるかもしれませんが、そうはいきません。ここは「.java」という拡張子を指定するのが仕様となっています。一方、Javaのソースコードを別の拡張子で保存し、それをコンパイルしようとしても、不具合が生じます。これは1つの決まりとして従うしかないでしょう。

図11:コマンドプロンプトに「javac Hello.java」とタイプ入力して、ソースコードをコンパイルする。結果が何も表示されなければエラーはない

すると、何ごともなかったかのように、コマンドプロンプトは次のコマンドを入力するように求めてきます。この場合、「便りのないのは良い便り」が成立します。つまり何も表示しないのは、エラーもなく無事にコンパイルが終了したからです。逆にエラーが

あった場合には、Javaコンパイラーがそのエラーの内容をレポートしてきます。

エラーなくコンパイルが完了したら、続いてそのプログラムを起動してみましょう。そのためには「java Hello」とタイプします（**図12**）。こんどは拡張子を指定する必要はありません。指定してもいけません。

図12: 続いてコマンドプロンプトに「java Hello」とタイプ入力して、コンパイル結果のバイトコードを起動する

その結果は、コマンドプロンプトのウィンドウの中に「Hello!」という文字で表示されます。これでこのプログラムは、意図した通りに完全に動作していることが確認できました。

プログラムを改良する

ここで、少しだけプログラムに手を入れて、もう少しだけ意味のある動作をするように改良してみましょう。3行目の「System.out.println()」の括弧の中を少し変更して、以下のように変更してみましょう。

```
class Hello {
        public static void main(String[] args) {
                System.out.println("Hello, " + args[0]);
        }
}
```

メモ帳のウィンドウ上のプログラムはこのようになるはずです（**図13**）。行の数は変わりません。3行目が少し長くなるだけです。

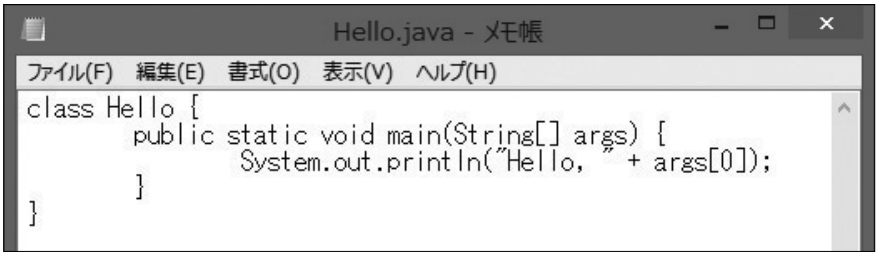


図13: すでに作成したソースコードに手を入れて、3行目のこのように変更する

このプログラムは、javaコマンドを使ってプログラムを実行する際に、コマンドプロンプトのJavaのプログラム名の後に入力した文字列を「Hello, 」に続けて表示するというものです。

変更を加えたソースコードは、当然ながらコンパイルする前に保存する必要があります。このように独立したエディターを使っている場合には、保存し忘れることもあるので注意が必要です。メモ帳の場合、変更を保存するには、「ファイル」メニューか

ら「上書き保存」を選びます（**図14**）。

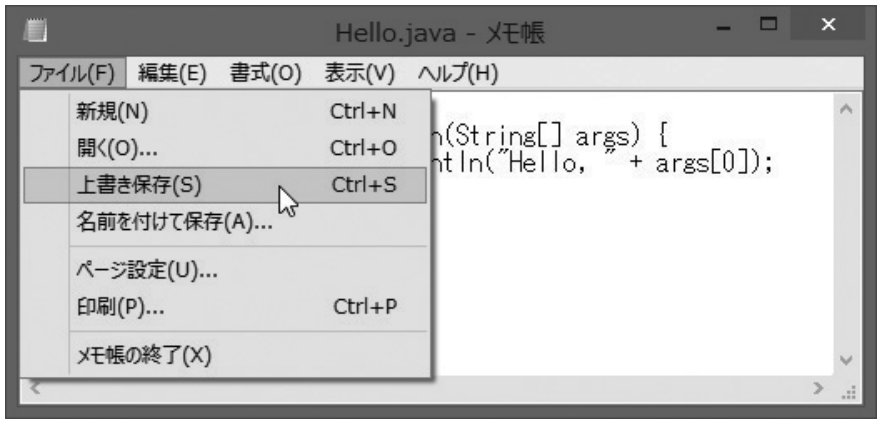


図14:メモ帳の「ファイル」メニューから「上書き保存」を選んで、修正下ソースコードを保存する

その後で、再びソースコードをコンパイルします。また「javac Hello.java」とタイプ入力します。さらに続けて、こんどは「java Hello World」とタイプしてから「Enter」を押してみましょう。その結果、コマンドプロンプトには「Hello, World」と表示されたはずです（**図15**）。

図15:再びコンパイルした後、今回は「java Hello World」とタイプしてもういちどプログラムを起動する

これは、コマンドプロンプトに入力された文字（ここでは「World」）をプログラムで読み取って、オウム返しではありますが、プログラムからの出力として表示しているのです。最初のプログラムよりも、ちょっとだけ高級な動作になりました。

ファイルの拡張子を確認する

今回示したように、メモ帳で作成したJavaソースコードを最初に保存した後には、拡張子の「.java」はアイコンの下に表示されていました。しかし、その後で中身を編集して保存し直すと、アイコンがメモ帳の書類のものに変更され、拡張子が表示されなくなってしまいます（**図16**）。これはWindowsというOSの特性によるもので、拡張子がなくなってしまったわけではなく、あくまで表示されなくなっただけです。

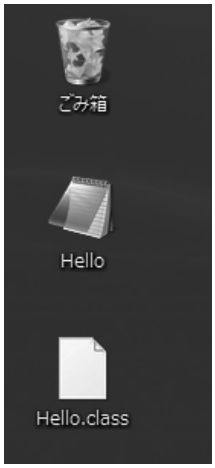


図16:メモ帳で編集結果を再保存すると、「Hello.java」ファイルの拡張子が表示されなくなり、ファイル名の表示は「Hello」となってしまう

今後ソースコードを扱う上でも、ファイル名の拡張子が表示されないのは何かと不便なので、必ず表示するようにWindowsの設定を変更しておきましょう。

まずタスクバーのフォルダー型のアイコンをクリックしてエクスプローラーのウィンドウを開き、「表示」のリボンを選択します（**図17**）。

図17: Explorerの「表示」リボンを開き「ファイル名拡張子」をチェックする

するとその中のオプションの「表示／非表示」のグループに「ファイル名拡張子」というチェックボックスがあるので、それをオンにします。ついでに「隠しファイル」もオンにしておいた方が良いかもしれません。

その後、デスクトップ上のJavaソースコードのファイル名が「Hello.java」と表示されていることを確認してください。

この設定は、もちろんデスクトップ上だけでなく、その他の場所でも有効です。

ここまでのプログラムを解剖する

今回入力して動かしてみたJavaのプログラムは、短いながらも、独立した立派なJavaのプログラムです。しかし、ここまで、その意味については何も説明していませんでした。それでもこのプログラムを動かすことで、「Hello」という文字や、それに続く文字をウィンドウの中に表示することは、目で見て確認できました。そうした機能が、どのようにして実現されているのか、簡単に見ておきましょう。

まず1行目には、

```
class Hello {
```

と書いてあります。

最初の「class」は、ここから「クラス」の定義が始まることを示しています。これはJavaとして定められたキーワードで、他の語で代用することはできません。クラスについては、5-3で解説しています。その次の「Hello」は、これから定義するクラスの名前です。これは任意なので、自分の好きな名前を付けることができます。ただし、途中にスペースを入れたりすることはできません。その次には「{」があります。これは、そこからクラスの定義の中身が始まることを示しています。といっても、この行はそこで終わりで、その後には何もありません。この「{」が示す始まりは、複数行に渡って有効なのです。それはクラス定義に限りません。「{」は複数行に渡る記述の始まりを表すと考えれば良いでしょう。

少し飛んで、5行目には、

```
}
```

があります。

この「}」が、この1行目にある始まり記号「{」に対応しています。つまり、クラスの定義は、1行目以降5行目まで続いていることになります。言い換えれば、その中身は2行目から4行目ということになります。

2行目に戻ると、その部分の記述は、

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

のようになっています。

まずこの行は全体で、「メソッド（method）」の定義の始まりを表しています。メソッドについても5-3で解説していますが、これはプログラムとして何らかの機能を実現する部分に名前をつけて記述するもの、と理解しておいてください。この場合の名前は「main」ですが、その前にも後にも、色々な語が付いています。前の方から見て行きましょう。

「{」と、それに対応する「}」を1行に書き、間にその中身を書いて、全体を1行で済ませることも可能ですが、プログラムが読み難くなるので、通常は複数行に分けて書きます。

4行目にも「}」がありますが、それは1行目の「{」ではなく、2行目の最後にある「{」と対応したものです。つまり「{」と「}」のペアの間に、別の「{」と「}」のペアが含まれています。こうした関係を「ネスト（nest）」していると言います。日本語では「入れ子」になっているとも言います。

最初の「public」は、普通の英語では「公共の」とか「公開された」といった意味でしょう。意味はここでも同じです。このmainメソッドは、このHelloクラスの外部から呼び出してもらうメソッドのため、その存在を公のものにしておく必要があるのです。そのために行の先頭にpublicを付けています。

次の「static」は、普通の英語では「静的な」とか「静止した」といった意味ですね。ここでは少し違った意味を表しています。英語のstaticは、dynamic（動的な）の反意語ですが、この場合のstaticは、動的に作られるのではなく、最初から存在しているという意味になります。最初からとはいつからかと言うと、それはこのHelloクラスが存在したときからです。staticではないメソッドは、これも5-3で説明していますが、クラスからオブジェクトを作成した際にはじめて存在するようになります。この例の場合、Helloクラスからオブジェクトを作成して使うわけではないので、mainメソッドをstaticにしておかないと利用することができないのです。

次の「void」は、英語では「無効な」とか「空の」といった意味です。この場合は、まさかこれから定義するmainメソッドが無効だというわけもないので、どちらかと言うと後者の意味です。しかしもちろんmainメソッドの中身は空ではありません。これは、mainメソッドが返す値が空だと言っているのです。メソッドというものは、どこからか呼び出されて、その機能を実行すると、それを呼び出した側に対して何らかの値を返すことができます。ここではどんな値を返すのかを指定できますが、何も値を返さないという選択肢もあります。その場合に指定するのが、このvoidというわけです。

mainという名前の後の括弧の中には、「String[] args」と記述されています。これは、このmainメソッドを外部から見た際の仕様の一種と考えられます。この部分では、mainメソッドを呼び出す際に与えるパラメーターの仕様を示しています。この意味はString[]というタイプのパラメーターをargsという名前で受け取るという意味です。String[]というのは、文字列を表す「String」の後ろに配列を表す「[]」を付けたもので、それによって文字列の配列を表します。文字列や配列については5-2で説明します。argsは単なる名前なので、実は他の名前でも構いませんが、mainメソッドの場合には、慣習的にこの名前を使うことになっています。

この行にも最後に「{」があるので、このメソッドの定義の中身が次の行から始まっていることが分かります。その定義は、3行目にあり、

```
System.out.println("Hello, " + args[0]);
```

となっています。

この行全体では、別に定義されたメソッドを呼び出して、文字を画面に表示しています。後ろから見ていくと、そのメソッドを呼び出す際に与えるパラメータが括弧の中に入っています。最初の例ではこの部分が「"Hello!"」となっていて、実行するとそのまま「Hello!」という文字が表示されました。Javaでは「""」（ダブルクォーテーション）によって文字列を表します。文字列とは文字の連なったもので、一般的な単語や文を表現できます。

Javaにはdynamicというキーワードはありません。しかし、staticの付いていないものは、基本的にダイナミックであり、動的に作られると考えても良いでしょう。

プログラムを書き替えたものは、「"Hello, " + args[0]」となっていて、文字列の後に「+ args[0]」と、何やら数式が付いています。ここでは詳しく説明しませんが、このうち「args[0]」には、コマンドプロンプトにタイプしたコマンドのうち、「java Hello」の次に書いた単語が入っています。実はそれも文字列です。それらの間にある「+」は、2つの文字列を結合するという機能を持っています。それにより、例えば「java Hello World」とタイプした際に、画面には「Hello, World」と表示されたのです。この行の最初から見て行くと、括弧の前には「System.out.println」と書いてあります。すでに述べたように、この行は別のメソッドを呼び出しているわけですが、この「System.out.println」全体が1つのメソッド名だと思っても構いません。ただし、細かく見て行くと、「.」(ピリオド)で区切られた3つの部分に分かれています。これによって、コンピューターのファイルシステムなどと同様に、メソッドを階層的に表しています。最初の「Sytem」は、このメソッドがJavaというシステムに標準的な機能であることを示しています。2番目の「out」は、その中で、このメソッドが何らかの「出力」を担当するものであることを示します。正確に言えば、この「out」は、そうした機能を持つオブジェクトの名前であって、そのオブジェクトの中の1つのメソッドが「println」なのです。

printlnメソッドは、パラメーターとして与えられた文字列を「標準出力」と呼ばれる場所に出力します。標準出力は、この場合はコマンドプロンプトのウィンドウに表示される文字列のことです。このメソッド名は「print」と「ln」を合成したものです。「print」は「印刷」という意味です。大昔のコンピューターには今のような「画面」というものはなく、結果は一種のプリンターに印刷していました。従って文字列を出力する機能の名前がprintであるのは自然なことだったのです。「ln」は「line」を縮めたもので、それによって1行を出力することを表しています。言い換えれば、この後に改行して、続く文字の出力は次の行から始まります。実際には、パラメーターで与えられた文字列の後に「new line」と呼ばれる改行コードを出力して、標準出力の中で改行を実行します。

check!

セミコロン「;」について

Javaソースコードの行の最後の最後にある「;」(セミコロン)は、プログラムを構成する文の区切りを表すもので、Javaのプログラムでは文の終わりには必ず付ける必要があります。他の行に付いていないのは、行の最後が「{」か「}」となっている部分だけです。「{」では、まだ文が終わっていないので付けません。この場合の「}」は「;」と同様に文の区切りを表すと考えられるので、付ける必要がありません。この「}」の後ろには「;」を付けても害はありませんが、普通は付けません。付けると「}」と「;」の間に空の文があることになります。なお、「;」で区切ることで、見かけの1行に複数の文を記述することもできます。

5-2 Javaプログラミングの基礎（1）

著：柴田文彦

この節からは、少しずつ意味のある動作をするJavaのプログラムを書いていきます。プログラムの内容は、まだ基礎的なものだけですが、Androidアプリを作るにあたって最初の一歩となるので、しっかりと学んでいきましょう！

5-2-1 Eclipseではじめる Java プログラミング

Javaプロジェクトの作成

Eclipse上では、どんなに簡単なプログラムを記述する際にも、まずプロジェクトを作成するところから始めます。基本的な方法は、Androidアプリのプロジェクトを作成する場合と同じですが、選択するプロジェクトのタイプが異なります。

まず、Eclipseの「File」メニューの「New」サブメニューから「Java Project」を選ぶか、ツールバーの「New」ボタンのメニューから「Java Project」を選びます（**図1**）。

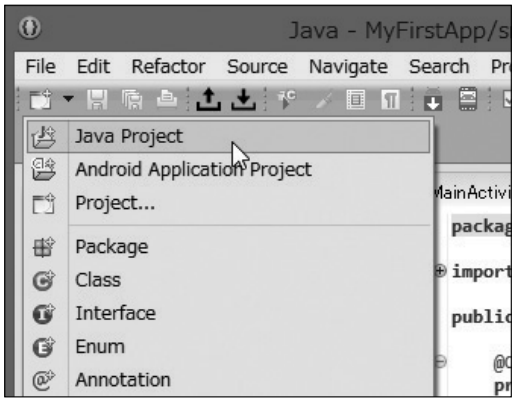


図1：メニューバーの下のツールバーの左端にある「New」ボタン右の下向き三角マークをクリックしてメニューを表示し、「Java Project」を選ぶ

すると「New Java Project」というダイアログが表示されるので、これから作成するプロジェクトの仕様を設定します（**図2**）。といっても、ここでは「Project name:」欄にプロジェクトの名前を入力するだけで良いでしょう。

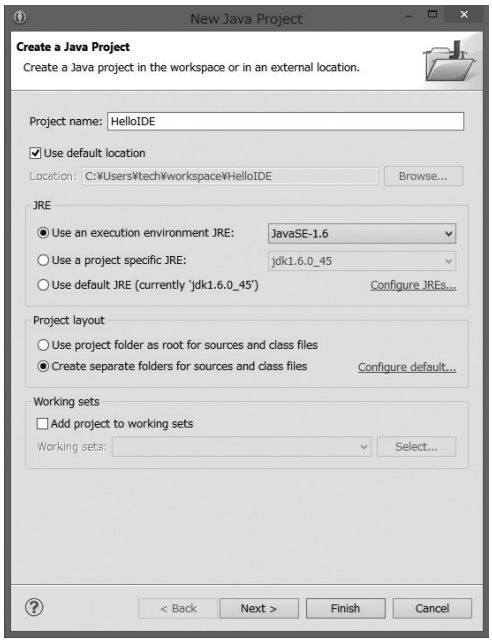


図2:「New Java Project」ダイアログの「Project name:」欄にプロジェクト名(ここでは「HelloIDE」)を入力する

ここでは「HelloIDE」という名前にしましたが、これは好きな名前にして構いません。その他の設定は変更せずに「Finish」をクリックしましょう。ここでは「Next >」ボタンをクリックすることもできます。すると、このプロジェクトに関してさらに細かな設定も可能となりますが、今は不要です(**図3**)。

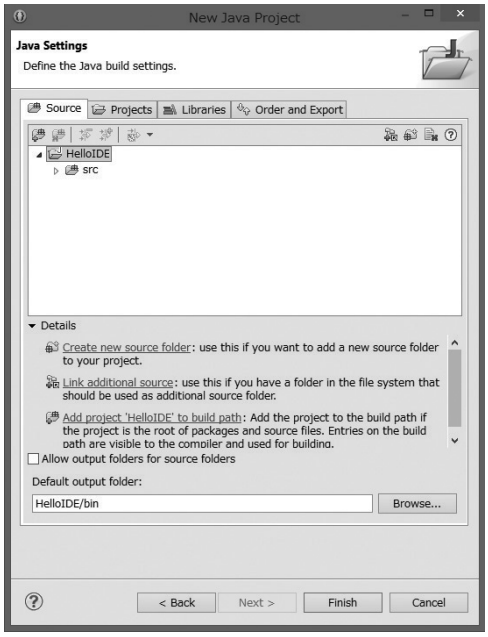


図3:前の「New Java Project」の「Create a Java Project」ダイアログで「Next>」ボタンをクリックすると、さらに細かいオプションを設定するダイアログが開く(今回は不要)

新しいプロジェクトが作成されると、それはEclipseの「Package Explorer」ビューで確認できます(**図4**)。よく見ると、右肩に「J」という文字の付いたフォルダー型の「HelloIDE」が作成されているはずです。これはこのプロジェクトが「Java Project」であることを示しています。

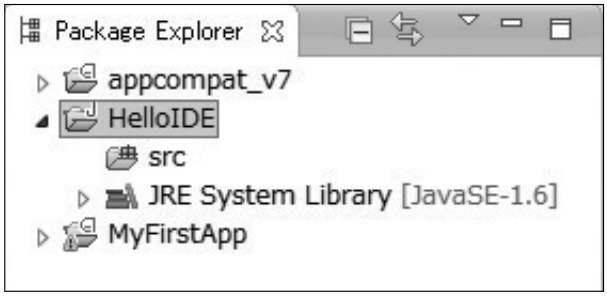


図4：新しいプロジェクト「HelloIDE」が作成され、「Package Explorer」ビューに表示された

Javaソースコードの作成

このプロジェクトの中には、初期状態で「src」と「JRE System Library」という2つのフォルダーが入っています。前者にはJavaソースが入りますが、初期状態では空です。Javaプログラムを書くには、その中にまずソースコードを追加する必要があります。

再びツールバーの「New」ボタンのメニューから、こんどは「Class」を選びましょう（**図5**）。

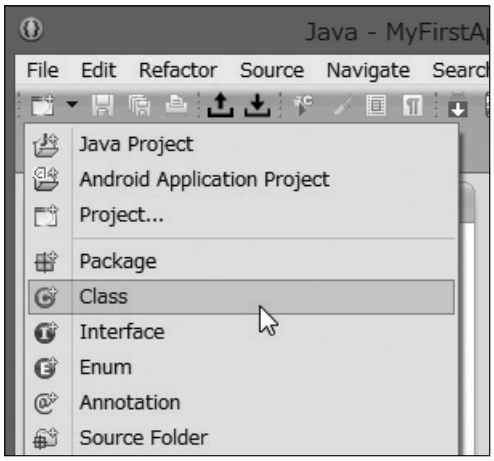


図5：ツールバーの「New」ボタンから表示するメニューで、こんどは「Class」を選ぶ

それによって、こんどは「New Java Class」というダイアログが表示されます（**図6**）。これから作成するJavaのソースコードについての仕様を設定するダイアログです。

図6：「New Java Class」ダイアログの「package:」欄と「Name:」欄に、それぞれパッケージ名とクラス名をタイプ入力する

「src」フォルダーを直接右ボタンクリックして表示されるメニューの「New」サブメニューから「Class」を選んで追加することも可能です。こちらの操作の方が直接的な感じで操作できます。

Javaのソースコードでは、基本的に1つのソースコードで1つのJavaクラスを定義するため、Javaのソースコードを追加するということは、すなわちJavaのクラスを定義することになります。そのため、このダイアログのサブタイトルは「Java Class/Create a new Java class」となっています。

ここには、とりあえず最小限の設定として「Name:」欄に、Javaクラスの名前を入力します。これがそのままJavaソースコードのファイル名にもなります。このダイアログのいちばん上の「Source folder:」には、あらかじめ「HelloIDE/src」が入力されていますが、その下の「Package:」欄は空になっています。この程度のテスト的なプログラムなら、空のままでも構いません。このパッケージ名は、Androidアプリプロジェクトの作成時に指定するパッケージ名と意味は同じです。つまり、逆ドメイン記法の組織名の後ろに、個人の識別子やプロジェクトの名前を書けば良いのです。この例では「jp.techinstitute.ti12345.helloide」としています。

「Finish」ボタンをクリックしてダイアログを閉じると、「src」フォルダーの中にソースコード「Hello.java」が追加されます(**図7**)。このとき、上のダイアログでパッケージ名を指定してあれば、見かけ上そのパッケージ名のフォルダーができて、その中にソースコードが入ります。

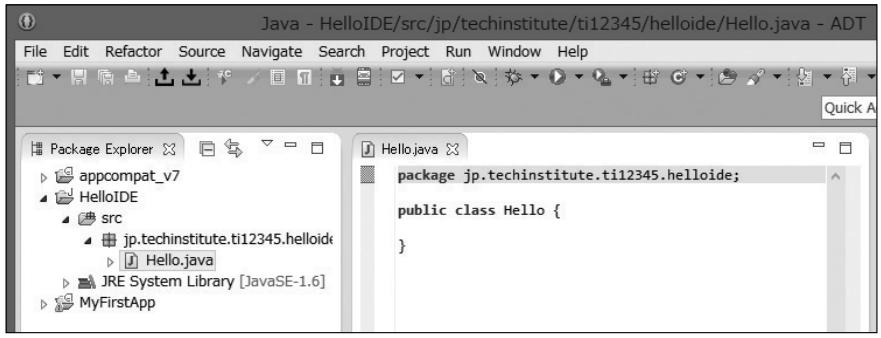


図2: プロジェクトに追加したJavaソースコード「Hello.java」は、プロジェクトの下の「src」フォルダーの中、ダイアログで指定したパッケージの下に入った

作成したソースコードは、自動的にエディターのビューの中の1つのタブとして開くはずです。パッケージ名を指定した場合には、ソースコードの中にもそのパッケージ名が転記されているはずです。

初期状態のソースコードの内容は、

```
public class Hello {


}
```

のように、「Hello」というクラス定義の枠だけが用意された状態です。とりあえず、この枠の中に前の節と同じソースコードを入力してみましょう(**図8**)。println()という1つのメソッドを呼び出すだけのmain()メソッドを定義するものです。

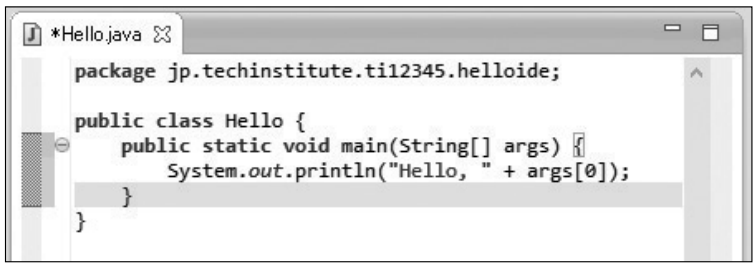


図8: 「Helloクラスの定義の中身を、前の節で書いたソースコードと同じになるように書き加える

Javaプログラムの起動

とりあえず、これをそのまま動かしてみましょう。そのためには、ツールバーの「Run」ボタンをクリックするのが簡単です（**図9**）。

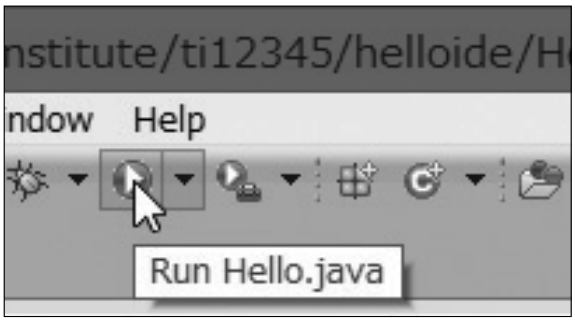


図9: ツールバーにある、緑の丸の中に白い右向き三角形の「Run」ボタンをクリックしてプログラムを起動する

すると、もしまだ編集したソースコードを保存してない場合には「Save and Launch」というダイアログが表示されます（**図10**）。これはプログラムの起動の前に、保存してないソースコードがあるかどうかをチェックし、もしあれば、それを保存するかどうかを確認するためのものです。

図10: プロジェクトに含まれるファイルを編集して保存していない状態で起動しようとすると表示される「Save and Launch」ダイアログ

そのまま「OK」ボタンをクリックすれば、保存して起動されます。複数のファイルを編集している場合には、その前のチェックボックスのオン／オフによって、保存するかどうかを個別に選択することもできます。また「Always save resources before launching」をチェックしておけば、これ以降、未保存のファイルは自動的に（黙って）保存してから起動するようになります。

プログラムの実行結果は「Console」ビューによって確認できます（**図11**）。このビューは、他の状態表示系のビューとスペースを共有しつつ、Eclipseの中央底辺近くに配置されいます。

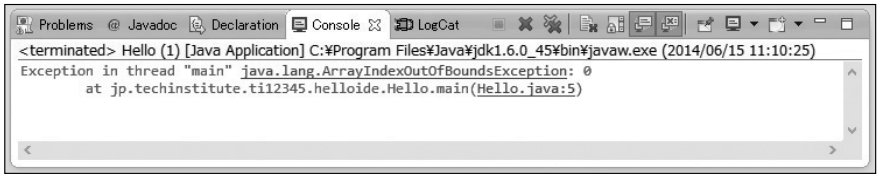


図11: 「Run」ボタンをクリックしただけで起動すると、「Console」ビューにエラーが表示される

独立したエディターを使う場合には、編集したソースコードの保存し忘れ、という心配がありますが、EclipseのようなIDEを使う場合には、それは解消されます。

このコンソール出力を見ると、エラー（例外）が発生して、プログラムが停止してしまったことが分かります。何か間違いがあったのでしょうか。

実はこれはソースコードの間違いという訳ではありません。中身を見てみると、「配列の添字が所定の範囲を超えている」というエラーです。「配列」については、まだ説明していないので、その意味が分からなくても構いません。ここで問題なのは、Javaプログラムがパラメーターを読み込もうとしているのに、そのパラメーターがないということです。

前節で、コマンドプロンプトを使ってプログラムを起動した際には、「java」コマンドと、バイトコードの名前「Hello」の後ろにパラメーターを指定していました。それと同じことをここでも実現する必要があります。しかし、Eclipse上では、プログラムをコマンドによって起動するわけではありません。どうすればよいのでしょうか。

それは、プログラムを起動する際の条件の設定で解決できます。プログラムを起動する際に、「Run」ボタンをクリックするのではなく、メニューを表示して「Run Configurations...」を選びます（**図12**）。

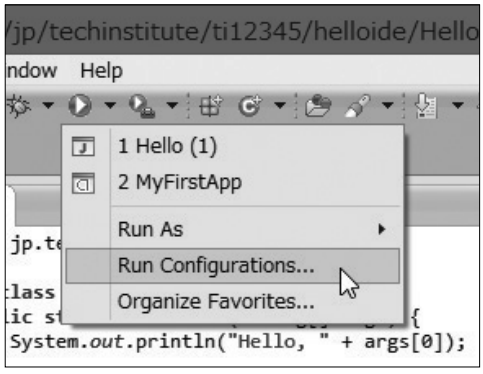


図12: プロジェクトを起動する際に、メニューから「Run Configurations...」を選ぶ

すると「Run Configurations」というダイアログが開くので、ここではその中から「(x)= Arguments」というタブをクリックして選びます（**図13**）

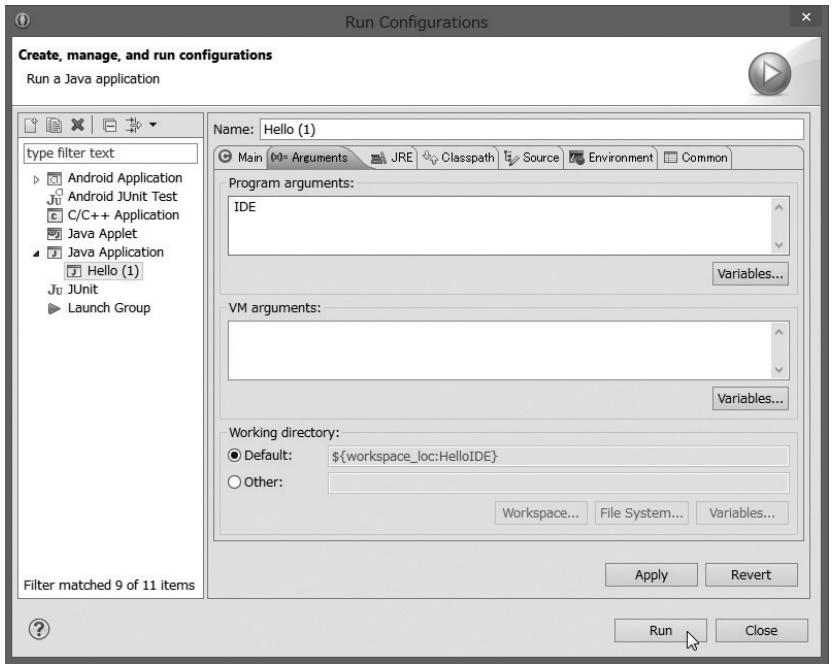


図13: 「Run Configurations」ダイアログでは「(X)= Arguments」タブを選択し、「Program arguments:」欄に、コマンドに与えるパラメータを記入する

これは、Javaプログラムをコマンドとして起動する際に与えるパラメーターを、Eclipseから起動する際にも与えることを可能にするものです。

ここでは「Program arguments:」欄になんでも好きな文字を入力してみましょう。この例では「IDE」としました。間にスペースや改行が入ると、その前後で独立した別のパラメーターとして扱われてしまいます。

このダイアログにある「Run」ボタンをクリックすると、ここで設定したパラメーターを使って、プログラムが起動します。

その結果は、やはりコンソールで確認できます。こんどはエラーは発生せずに、この例の通りなら「Hello, IDE」と表示されたはずです(**図14**)。

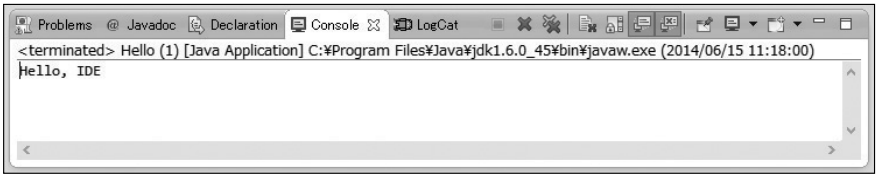


図14: Javaプログラムにパラメーターを与えて起動すると、エラーは発生せず与えたパラメーターが「Hello, 」の後に表示された

argument(アーギュメント)とは「引数」のことで、この場合にはコマンドに与えるパラメーターと同義語と考えてよいでしょう。

5-2-2 変数とデータ

変数って何?

プログラミングの初心者が、まず理解しなければならないのは、変数の扱いです。プログラムで使う変数も、一般的な数学の方程式の中に出てくる変数に似た概念ですが、違いも色々あり、別物と考えた方が無難でしょう。数学では、何か値の分からないものに変数を当てはめておいて、後で方程式を解くことで、その値を求めるために使われます。あるいは、連続的に値が変換する抽象的な概念に変数を割り当てて、最後まで抽象的なものとして扱うために使ったりします。

プログラミング言語の変数は、プログラムの実行中に変化する可能性のある値に名前を付けて記憶しておき、値が変化しても同じ名前で参照できるようにするための機構です。簡単に言えば、変数は何らかのデータを表すために使うものということになります。しかし、Javaをはじめとするオブジェクト指向言語では、後で述べるように変数が表すのは必ずしもデータだけとは限りません。

最初に変数を「宣言」しよう

変数は、いきなり使うことはできません。その前にまず「宣言」します。宣言とは、「今後、こういう種類の、こういう名前の変数を使います」と、あらかじめ申告しておく

ようなものです。変数の宣言は、

変数の型 変数名;

のように書きます。変数の型と変数名の間には空白文字（半角スペースかタブ）が入ります。

変数の型は、その変数の種類を表します。大きく分けると、主に数値を表す基本的データ型と呼ばれるものと、オブジェクトを表すオブジェクト型があります。オブジェクトについては後で説明しますが、基本データ型が単純な単一のデータを表すのに対して、オブジェクトは複雑な構造を持ったものと考えてください。

Javaの基本データ型には、以下のものがあります。

JavaにはC系の言語にあるような符号なしの整数はありません。従って、unsignedというキーワードを整数型の前に付けて、符号なしを表すこともできません。

型	内　容
byte	1バイト整数（-128～127）
short	2バイト整数（-32768～32767）
int	4バイト整数（-2147483648～2147483647）
long	8バイト整数（-9223372036854775808 ～9223372036854775807）
float	4バイト（単精度）浮動小数点数（±1.40239846e-45 ～±3.40282347e+38）
double	8バイト（倍制度）浮動小数点数（±4.94065645841246544e-324 ～±1.79769313486231570e+308）
char	1文字（2バイトのUnicode文字）
boolean	真偽値（true（真）またはfalse（偽））

Javaの変数名の規則は比較的緩く、Unicode文字が使えるので、その気になれば日本語（漢字、ひらがな）の変数名も可能です。ただし通常は、半角英数字、「_」（アンダースコア）、「$」などを組み合わせた単語を変数名とします。変数名の中には「+」や「-」を含むことはできません。後で述べるような演算の記号と区別できなくなるからです。数字で始めることもできません。また変数名の大文字と小文字は区別されます。Javaのプログラミング言語としてあらかじめ定義されている単語、つまりキーワードをそのまま変数名にすることはできません。例えば、型の名前はキーワードなので、そのままでは変数名にできませんが、前後に何か付ければ可能です。

変数名として有効な例と、有効でない例をいくつか示します。

有効なもの：

```
i
i1
my_name
_tempVal
```

有効でないもの：

```
1i（数字で始まっている）
may+day（間に＋が入っている）
your name（間にスペースが入っている）
```

次に実際の変数の宣言の例をいくつか示します。

```
int myInt;
float myFloat;
short myShort;
```

複数の変数名をコンマで区切って、同じ型の変数を複数同時に宣言することもできます。

```
変数の型 変数名1, 変数名2, 変数名3
```

変数の宣言と同時に、その変数の値を初期化することもできます。その際には、「=」を使った代入式によって、初期値を変数に設定します。

```
変数の型 変数名 = 初期値;
```

実際の例としては、以下のような形になります。

```
double myDouble = 3.2834;
```

変数だけでなく「定数」もある

プログラミングでは、変数と良く似たものとして定数も登場します。これも数学や物理学に出てくる定数に近いものがありますが、やはり異なります。プログラミングの定数は、プログラムの動作中に値が変化しない、変化させることができない、特別な変数と考えた方が分かりやすいでしょう。

C系の多くの言語では、変数の宣言、初期化の前に「const」というキーワードを付けることでと、定数を宣言し初期化することができます。Javaでも「const」はキーワードとして予約されていますが、少なくとも定数を宣言するのにconstを使うことはできません。Javaでは、「final」というキーワードを使って定数を定義できます。ただし、finalは、定数の宣言だけでなく、色々な場面で変更を許さないものを宣言する際にも使います。

5-2-3 小学校で習う計算を「演算」してみよう

演算とは何か？

「演算」という言葉は、日常ではあまり使うことはないでしょう。数学では使いますが、比較的狭い範囲でしか使わないかもしれません。コンピューターの世界の演算は、それよりも範囲が広く、単なる「計算」とほとんど同じ意味で使います。もう少し正確に言えば、コンピューターで計算することを演算と呼ぶと言っても良いでしょう。

しかし、コンピューターの演算には、いろいろと種類があります。最も一般的な数値同士、あるいは数値を記憶している変数を使った計算を「算術演算」と呼びます。またコンピューターの数値を構成しているビット単位で細かな処理を実行するものを「ビット演算」と呼びます。さらに、正しい、正しくない、つまり真と偽を組み合わせた論理的な処理のことを「論理演算」と呼びます。

一方、それらの演算に使われる記号のことを「演算子」と呼んでいます。例えば「+」は、算術演算で加算（足し算）を表す演算子というわけです。

ここから、各種の演算と、それに使う演算子を一通り見ていきましょう。

算術演算

算術演算は、数値計算と言い換えてもよい、コンピューター上での演算のことです。算術演算は、数値を表す2つの項を算術演算子でつないで表現します。例えば、以下のような式が算術演算です。

```
a + 3
1 - 4
myInt * i3
```

Javaの算術演算子には、以下のようなものがあります。

算術演算子	意　味
+	加算
-	減算
*	乗算
/	除算
%	剰余算

Javaでも、他の多くのプログラミング言語でも、掛け算は「×」ではなく「*」（半角アスタリスク）、割り算は「÷」ではなく「/」（半角スラッシュ）で表すのが普通です。それは、初期のコンピューターで扱える文字の中に、「×」や「÷」が含まれていなかったことに起因するものです。そこでしかたなく「*」や「/」で代用したというわけです。また「×」は、英文字の「x」（エックス）と紛らわしいので、混同を避けるという目的も

あったかもしれません。

「*」や「/」以外で見慣れないのは「%」（パーセント記号）でしょうか。これは剰余算、つまり割り算の余りを表す演算です。例えば、13 % 4の結果は1になります。13を4で悪と、3余り1になるからです。ちなみに13 % 3の結果も1です。

これらの演算子には優先順位というものがあります。優先順位というのは、（3つ以上の）数値と演算子が交互に連続して書かれているとき（例えば「3 + 2 * 4」など）、その中のどの演算子から先に計算するかということです。つまり、必ずしも書いてある順番に（左から）実行するのではないのです。

上に挙げた演算子の中では、「+」と「-」の優先順位は同じで、他の3つよりも低くなっています。また、「*」と「/」と「%」は同じで、「+」や「-」よりも高くなっています。つまり、上に上げた5つの演算子には2種類の優先順位しかないことになります。簡単に言えば、普通の算数と同じように、掛け算や割り算（剰余算も含む）の優先順位は、足し算や引き算よりも高く、先に実行するということになります。なお、括弧「()」を付けて、演算部分をくくると、その括弧の中は、外よりも先に演算を実行することになります。それによって既定の演算の優先順位を無視することができるのです。

実際にいくつかの演算の例を見てみましょう。

```
int a = 3;
int b = 4;

System.out.println(a + b);
System.out.println(23 % 5);

int c = a * 8 - b;
int d = 28 / (a + b);

System.out.println(c);
System.out.println(d);
```

このプログラムを実行すると、コンソールに出力される数字は、順に7、3、20、4になるはずです。確認してみましょう。

一般的な算術演算子以外の演算子も含めると、Javaの演算子の優先順位はもっとずっと細かく定まっています。ここでは煩雑になるので、それらには触れないでおきます

浮動小数点数とは、少数（0.2とか、3.14とか）を「仮数部」と「指数部」に分けて表現する方式の数のことです。仮数部では桁を無視した数を、指数部では、その桁を示します。3.14の場合、仮数部を314とすると指数部は10^{-2}で、「314×10^{-2}」、あるいは仮数部が31.4なら指数部は10^{-1}で、「31.4×10^{-1}」となります。元の意味は小数点の位置が移動するということで、移動した分は指数部で調整することになります。

インクリメント、デクリメント演算子

算術演算子の一種ですが、ちょっと特殊な使い方をする、変わった演算子として、「インクリメント」、「デクリメント」演算子があります。これらは、変数の値を1だけ増やしたり減らしたりするものです。

通常は整数型の変数に対して使いますが、floatやdoubleなど、浮動小数点型の変数に対しても利用できます。

これらは、以下の2種類だけです。

演算子	意　味
++	インクリメント（1増やす）
--	デクリメント（1減らす）

例えば、変数iの値を1だけ増やすには、通常の加算の演算子を使えば、「i = i +1」のように表現できます。ここにきて、これでは左辺と右辺の値が異なるのに、それを等号「=」で結んでいるのはどういうことか、と疑問に思われるかもしれません。しかしJavaをはじめとするたいていのプログラミング言語では、「=」は、その左右が等しいことを表すのではなく、「代入」を表すのです。つまり、この左側にある変数に、右側にある値を設定するのです。

話が本題からそれましたが、このように通常の算術演算子を使って「i = i +1」と書くところを、インクリメント演算子を使えば「i++」と簡潔に表現することができます。例を見てみましょう。

```
int i = 5;

i = i + 1;
System.out.println(i);

i++;
System.out.println(i);
```

このプログラムを実行すると、コンソールには順に6、7が出力されます。

なお、インクリメント、デクリメント演算子は、上の例のように変数の後に付けることも、前に付けることもできます。単独で書く場合には「i++」も「++i」も同じ効果を発揮しますが、この値を他の変数に代入しようとすると意味が違ってきます。「j = ++i」と書くと、iの値を1増やしてからjに代入するのに対し、「j = i++」と書くと、iの値をjに代入してから、iの値だけを1増やします。実際にプログラムを動かして確認しておきましょう。

```
int i = 3;
int j;

j = ++i;
System.out.println(j);

j = i++;
System.out.println(j);
```

この結果の変数jの値は、いずれも同じ4で、コンソールには4、4が出力されます。

「あるある」か「ないっしょ」だけで考える演算子

コンピューターの中の数字やデータが、すべてビットで表されていることはご存知でしょう。ビットは、1か0か、オンかオフか、あるかないか、などを示すことのできる情報の最小単位です。その最小の情報単位で演算を実行するのがビット演算子です。ビット演算では、2つの整数値、または整数型の変数、またはそれらの組み合わせに対して、それらを構成するビット列の対応するビット同士を演算し、それぞれの演算結果によって構成された値を得ることができます。

まず、Javaのビット演算子には以下のようなものがあります。

演算子	意　味
&	AND（ビット積）
^	XOR（ビット排他的論理和）
|	OR（ビット和）
~	NOT（全ビットの反転）

これらのうち「~」以外は、通常の算術演算子と同様に、演算子の左右の値の間で演算を実行します。「~」は単項演算子と呼ばれるもので、演算の対象となる数値は、この後に続く1つだけです。これは、それに続く数値を構成するビットをすべて反転する（1を0に、0を1にする）機能を持っています。

例えば5という数値は、4ビットの2進数（1と0だけで表した数）で表すと「0101」になります。また、12は同様に「1100」になります。そのため「5 & 12」とすると、対応するビット同士が両方とも1の部分のみ1となり、他は0となるので結果は「0100」で、4になります。「5 | 12」では、対応するビットのどちら一方でも1なら1になり、両方とも0の部分だけが0となるので、結果は「1101」で13になります。実際のプログラムで確かめてみましょう。

```
int a = 5;
int b = 12;

System.out.println(a & b);

System.out.println(a | b);

System.out.println(a ^ b);
```

これらの結果、コンソールには、4、13、9と出力されます。3番目の^（XOR）がどういう演算なのか、考えてみましょう。9を2進数で表すと「1001」となるのがヒントです。

ズラしていく「シフト演算子」

シフト演算子は、整数型の変数や数値を構成するビットを左または右にシフトする、つまり1つずつ左右に動かす機能を持った演算子です。これらの演算子の前の変数、または値を、演算子の後ろの変数または値が示すビット数分だけシフトします。一般的に、右にシフトすると、結果の値は元の値の半分になります。元の値が奇数なら、端数は切り捨てられます。逆に左にシフトすると、結果の値は元の値の2倍になります。

Javaのシフト演算子には、左か右かで、次の2種類があります。

演算子	意　味
<<	左シフト
>>	右シフト

実際の例を見ておきましょう。

```
int a = 13;
int b = 5;

System.out.println(a >> 1);

System.out.println(b << 3);

System.out.println(a << 5);
```

このプログラクを実行すると、コンソールには、6、40, 416と表示れます。

意義なし！ 判決を下すのみの「論理演算子」

論理演算は、論理値、つまり正しいか、正しくないか、真か偽かといった論理を扱う値に対する演算です。

Javaの論理演算子には、次の3種類があります。

演算子	意　味
&&	AND（論理積）
||	OR（論理和）
!	NOT（論理否定）

「&」や「¦」の記号の意味はビット演算子とほぼ同じですが、論理演算では「&&」や「¦¦」のように、2つずつつなげて書くことに注意してください。これらは紛らわしいので、思わぬバグの原因になることも少なくありません。ビット演算の否定（反転）の演算子は「~」だったのに対し、論理演算の否定は「!」です。これも単項演算子です。

ちなみに、論理値を扱う変数は、「boolean型」です。また真という値は「true」、偽という値は「false」で表します。それを使った例をみておきましょう。

```
boolean a = true;
boolean b = false;

System.out.println(a && b);

boolean c = a || b;
System.out.println(c);

System.out.println(!c);
```

このプログラムを実行すると、コンソールには、false、true、falseと表示されます。

どっちが大きいか、小さいかを調べる「比較演算子」

2つの同じ型の値、つまり整数値同士や浮動小数点値同士の大小を比較するのが比較演算で、そのための演算子が比較演算子です。比較演算の結果は、かならず真偽値、つまりboolean型となります。この結果は、boolean型の変数に代入することもできますが、多くの場合は、後で述べるようなif文、while文での条件判断のために使います。

Javaの比較演算子には以下のようなものがあります。

演算子	意　味
<	（左は右より）小さい
<=	（左は右）以下
>	（左は右より）大きい
>=	（左は右）以上
==	（左右が）等しい
!=	（左右は）等しくない

やはりいくつかの例で確認しておきましょう。

```
int a = 24;
int b = 17;

System.out.println(a >= b);

boolean c = a < b;
System.out.println(c);

System.out.println(a != b);
```

この結果、コンソールにはtrue、false、trueと表示されます。

経緯はいいから結論だけおしえてくれる「代入演算子」

代入演算子「=」は、この演算子の左側にある変数に、右側にある数値や、他の変数の値、あるいは演算結果を代入します。また、Javaにも、代入演算子と算術演算子を組み合わせた複合代入演算子というものがあります。それを使えば、ある変数に、その変数の元の値ともう1つの値を演算した結果を代入することができます。これは変数の値を少しずつ変化させるような場合に便利です。

複合代入演算子には以下のようなものがあります。

演算子	意　味
+=	和を代入
-=	差を代入
*=	積を代入
/=	商を代入
%=	剰余を代入
>>=	右シフトした結果を代入
<<=	左シフトした結果を代入
&=	ビット積を代入
^=	ビット排他的論理和を代入
|=	ビット和を代入

これも実際の例で確認しておきましょう。

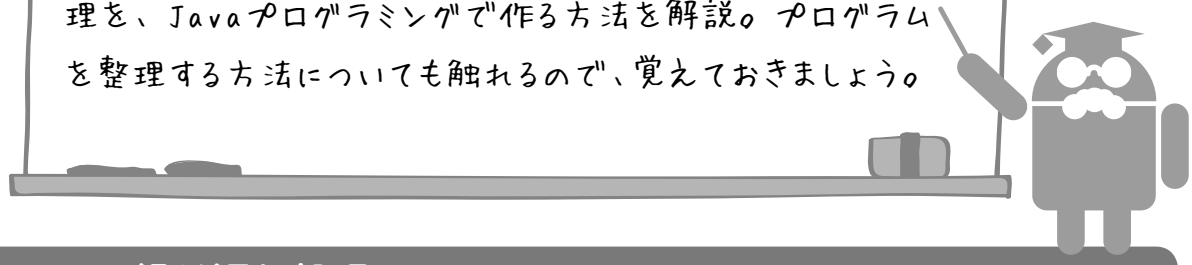
```
int a = 7;
int b = 5;

a += b;
System.out.println(a);

a %= 3;
System.out.println(a);

b &= 2;
System.out.println(b);
```

この結果、コンソールには、12、0、0が表示されます。

5-3 Javaプログラミングの基礎（2）

著：柴田文彦

この節では、コンピューターが得意とする繰り返しなどの処理を、Javaプログラミングで作る方法を解説。プログラムを整理する方法についても触れるので、覚えておきましょう。



5-3-1 繰り返し処理

forによる繰り返し

Javaはもちろん、多くのプログラミング言語では、繰り返し処理を実行するために「for」という構文を用意しています。これには、バリエーションも色々ありますが、C言語のものを継承した基本的なものは、多くの言語がサポートしています。それさえ覚えておけば、繰り返し処理で困ることはないはずです。

forによる繰り返し処理の基本的な形は、以下のような形になっています。

```
for（初期化；継続条件；ループごとの処理）{
文
}
```

これは、forの後の「()」の中に、3つの式を「;」で区切って書き、その後に「{}」で囲って、繰り返し実行する文（複数）を記述するという構造になっています。

「()」の中に最初に書く「初期化」というのは、ループを始める前に、必ず実行する式です。例えば、ループの数をカウントする変数の値の設定などを実行します。「継続条件」というのは、その式を評価した結果が真である間、このループを実行するという条件を表すものです。通常は、ここに何らかの論理演算を書けばよさそうだということが分かるでしょう。「ループごとの処理」というのは、繰り返し実行することになる「{}」で囲まれたブロックの中の文を1回実行した後に、必ず実行する式のことです。通常は、ループの数をカウントする変数の値を増やしたり、減らしたりする処理を書きます。

簡単な例を見てみましょう。

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

これを実行すと0から9までの整数を順にコンソールに出力します。この場合、「初期化」の式は、「int i = 0;」です。これにより整数型の変数iを宣言し、そのまま0に初期化しています。言うまでもなく、これがループの数を数えるカウンターになります。

「継続条件」の式は、「i < 10;」です。この「<」は、左側が右側よりも小さい場合に真と評価される論理演算子でした。つまり、iの値が10より小さいうちは真となるので、その間、ループを回り続けることになります。この演算子が「<」なので、10は含まれず、0から9となるのです。

もし、0から10までループを回したい場合にはどうすれば良いか、考えてみましょう。

3番目の「ループごとの処理」としては、「i++;」と書いてあります。これは変数の値を1だけ増やす、インクリメント演算子でした。これにより、1回ループを回るごとに、カウンターであるiの値は1だけ増えていきます。

もしループを逆向きに回したい場合、つまりカウンターの値を減らしながら繰り返し実行したい場合には、以下のようにします。

```
int sum = 0;
for (int i = 10; i > 0; i--) {
    sum += i;
}
    System.out.println(sum);
```

これを実行すると、カウンターの値は最初が10、そこから1つずつ減っていて1になった時点でループを抜けます。「i > 0;」という継続条件なので、最後の0は含まれません。その結果、ループを始める前に宣言して初期化しておいた変数sumの値は、10から1までの整数の合計、つまり55になります。

一般的なforループの実行の様子をフローチャートに示しておきましょう（**図1**）。

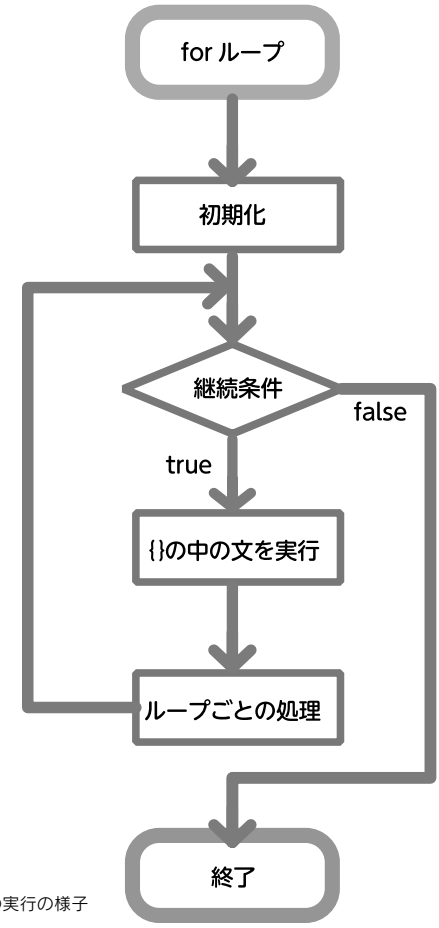


図1:forによるループの実行の様子

これを見てもわかるように、forループの場合、最初から継続条件が満たされていなければ、「{}」の中の文は、一度も実行されません。

whileによる繰り返し

forほどではないかもしれませんが、「while」によるループも一般的に使われます。これは、一定の条件が満たされている間、回数に関係なく繰り返し処理を実行するというものです。一般的には次のような形になっています。

```
while ( 継続条件 ) {
文
}
```

whileループの場合には、まず最初に「継続条件」の式を評価します。これが真なら、「{}」で囲まれたブロックの中の文(複数)を実行します。その後再び「継続条件」の式を評価し、真ならまた「{}」の中を実行するということを繰り返します。もし継続条件が最初から真で、その後変化しなければ、ループは永遠に続くことになります。そうならないためには、繰り返し処理を実行中に継続条件が変化する必要があります。

whileの継続条件は、必ずしもカウンターを使ったものである必要はなく、むしろそうでないところにwhileの価値があるのですが、ここでは単純なカウンターを使った例を示します。

```
int sum = 0;
int counter = 1;
while (counter <= 100) {
    sum += counter++;
}
System.out.println(sum);
```

whileには、forのような初期化やループごとの処理を強制する機構がないので、自分で用意する必要があります。この場合、計算結果を入れる「sum」と、カウンターとして使う「counter」という、いずれもint型の整数をwhileループに入る前に用意しています。継続条件は「counter <= 100」なので、カウンターが100になるまで（100になったときを含む）ループを繰り返します。このループを終わらせるためには、ループの実行の中でカウンターの値を増やしていかなければなりません。そのため「counter++;」としているわけです。このプログラムを実行すると、1から100までの整数の合計値、5050がコンソールに表示されます。

whileループの実行の様子も一般化してフローチャートに示しておきましょう（**図2**）。

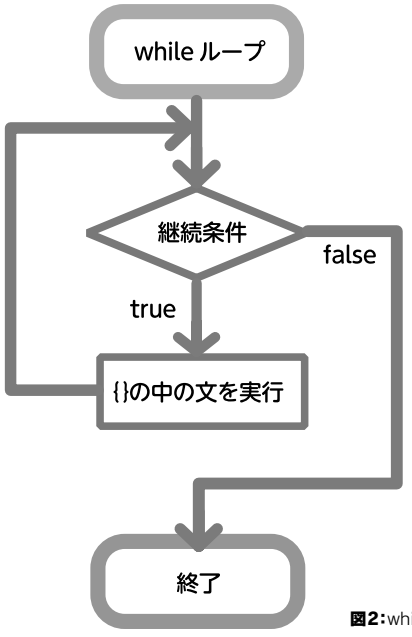


図2: whileによるループの実行の様子

whileの場合も、forと同様、最初から継続条件が満たされていなければ、「{}」の中の文は、一度も実行されません。

do～whileによる繰り返し

forやwhileに比べると、使われる機会は多くありませんが、Javaでは「do～while」というループの表現も可能です。これはwhileと同様に一定の条件が満たされている間はループを繰り返しますが、その条件を評価する場所が違います。最初ではなく最後になります。do～whileは、一般的には次のような形になっています。

```
do {
文
} while ( 継続条件 );
```

この場合、まず「{}」で囲まれたブロックの文（複数）を実行した後で、whileの後の継続条件を評価し、それが真なら再び「{}」の中を実行、偽なら終了するというものです。

簡単な例で試してみましょう。

```
int mul = 1;
int counter = 1;
do {
    mul *= counter++;
} while (counter < 8);
System.out.println(mul);
```

この例では、変数「counter」の値は1から7まで変化することになります。そして変数「mul」には、カウンターの値を順に掛け合わせた数が入ります。その結果は5040となるはずです。

do～whileループの実行の様子もフローチャートに示します（**図3**）。

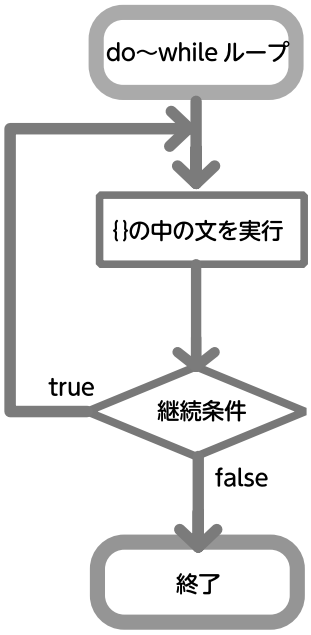


図3:do～whileによるループの実行の様子

forやwhileの場合とは異なり、継続条件の評価は後回しなので、それが満たされていなくても「{}」の中の文は、必ず一度は実行されることに注意しましょう。それこそがdo～whileの存在意義です。

繰り返し処理と配列

「配列」については、以前にちょっとだけ出てきましたが、ここでもう少し詳しく説明しておきましょう。配列とは、同じ型の変数を複数まとめて参照したり、管理するための機構です。変数名も同じものを使いますが、整数の添字（インデックス）を付けて

区別します。添字は変数名の後の「[]」の中に書きます。添字は整数値そのものでも、整数型の変数を指定してもかまいません。

配列も、一般の変数と同じように宣言してから使います。配列の宣言の一般的な形は以下のようになっています。

```
変数の型 [] 配列名 ;
```

例えばint型の配列「numbers」を宣言するには以下のようにします。

```
int[] numbers;
```

配列の中身は最初にまとめて初期化するか、大きさ（要素の数）だけを指定して、中身は空の配列を作っておき、後から1つずつ中身の要素を入れていくこともできます。

最初にまとめて初期化するには、以下のように「{}」を使い、個々の要素を「,」で区切って指定します。

```
変数の型 [] 配列名 = { 要素 1, 要素 2, 要素 3, ....};
```

要素の数に特に制限はありません。実際の例を示すと、以下のような表記が可能です。

```
int[] numbers = {3, 5, 6, 8, 2, 7, 1};
```

このようにnewを使って要素の数を指定して確保した配列の要素の数は、後から変更することはできません。

大きさだけを指定して、中身が空の配列を作るには、「new」というキーワードを使います。

```
配列名 = new 変数の型 [ 要素の数 ];
```

この配列名は、あらかじめ宣言されていなければなりません。配列の宣言と大きさの指定を同時に書くこともできます。

```
変数の型 [] 配列名 = new 変数の型 [ 要素の数 ];
```

例えば、以下のような表記が可能です。

```
double[] dnums = new double[5];
```

この例でも、変数の型を指定する「double」が2回出てきて冗長に感じられるかもしれませんが、このように書く場合には避けられません。もちろん、それらの型は一致していなければなりません。

配列の中身の値を参照するには、配列名の後ろに「[]」を付け、その中に添字（インデックス）として整数値、または整数型の変数を指定します。

配列名 [添字の整数];

実際には、以下のような書くことができます。

```
double dnum = dnums[3];
```

配列は、繰り返し処理と組み合わせて、ループのカウンターを添字として使ってアクセスすると便利です。簡単な例を示しましょう。

```
int[] numbers = {3, 5, 6, 8, 2, 7, 1};
double[] dnums = new double[7];
for (int i = 0; i < 7; i++) {
    dnums[i] = numbers[i] / 2.0;
    System.out.println(dnums[i]);
}
```

このプログラムは、最初に初期化した配列「numbers」から、要素を1つずつ取り出し、それを2で割った値を、サイズを指定しただけの配列「dnums」の対応する位置に格納するというものです。これを実行すると、コンソールには、1.5、2.5、3.0、4.0、1.0、3.5、0.5という数値が順に表示されます。また、実行後の配列dnumsにも同じ値が順に格納されることになります。

なお、ここでは示しませんが、配列は2次元、3次元、あるいはそれ以上の多次元のものを作成して使うこともできます。

ifによる条件付き実行

どんなプログラミング言語にも、条件によって実行する部分を変更する、いわゆる分岐の機能があります。その最も一般的なものは「if」文によるものです。

if文の最も基本的な形は以下のようになっています。

```
if (条件式) {
文
}
```

これは、条件式を評価して、それが成り立つとき、つまりその値が真なら、「{}」で囲まれたブロックの中の文（複数）を実行するというものです。条件式が成立しなければ、条件式の評価以外は何も実行しません。

if文をループの中で使用する簡単な例を示しましょう。

```
for (int i = 0; i < 10; i++) {
    if (i % 2 != 0) {
        System.out.println(i);
    }
}
```

このプログラムは、関数「i」をループカウンターとして、10回ループが回ることは、もはや説明の必要はないでしょう。そのループの中では、if文を実行します。そこで、ループカウンターのiの値を2で割った余りが0でなければ、言い換えればiが2で割り切れないときだけ、それに続く「{}」の中の文を実行します。それは、そのiの値をそのままコンソールに出力するというものです。これにより、0から9の間の奇数だけが表示されることになります。実際にこのプログラムを動かしてみると、1、3、5、7、9が、順に表示されます。

if文の実行の様子をフローチャートに示します（**図4**）。

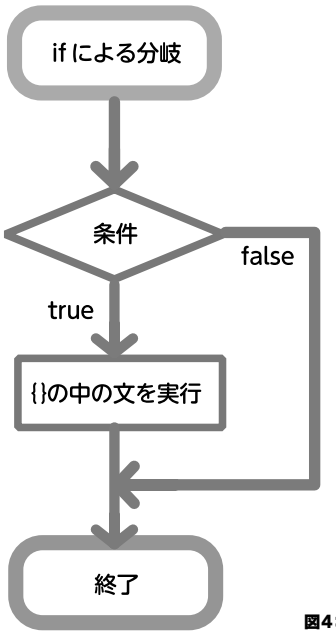


図4:if文による実行の分岐の様子

if～elseによる実行部分の選択

if文には、一種のオプションとして「else」節を付けることができます。左で説明したif文では、条件が成立する場合のみ、「{}」の中の文を実行しましたが、elseを付けることで、逆に条件が正立しない場合のみ実行する文のブロックを付加することができます。

if～elseの一般的な形は以下のようになります。

```
if ( 条件式 ) {
文
} else {
文
}
```

これは、条件式を評価して、それが成り立つとき、つまりその値が真なら、ifの後の「{}」で囲まれたブロックの中の文（複数）を実行し、条件式が成立しなければ、つまりその値が偽なら、elseの後の「{}」で囲まれた文（複数）を実行するというものです。条件式が成立した場合には、もちろんelseの後の「{}」の中の文は実行しません。

やはりif～elseをループの中で使用する簡単な例を見ておきましょう。

```
for (int i = 0; i < 10; i++) {
    if (i % 2 != 0) {
        System.out.println("$"+i);
     } else {
        System.out.println("#"+i);
    }
}
```

このプログラムも、iをループカウンターとして、0から9まで10回ループを回ります。その間、ループカウンターのi値を2で割った余りが0でなければ、その数字の前に「$」

を付けて、そうでなければ、つまりiの値を2で割った余りが0なら、その数字の前に「#」を付けてコンソールに出力します。これにより、0から9の間の奇数には「$」が、偶数には「#」が付いたものが表示されます。実際に動かしてみると、#0、$1、#2、$3、#4、$5、#6、$7、#8、$9が、順に表示されるはずです。

elseの後には、さらにif文付け、さらに条件判断を続けることもできます。ただ、これは特別なものではなく、最初の条件が成立しない場合に、別のif文を実行する、さらにそれを繰り返すものです。一般的には以下のようになるでしょう。

```
if（条件式1）{
文
} else if（条件式2）{
文
} else if（条件式3）{
文
} else if ...
```

一般的なif～else文の実行の様子をフローチャートに示します（**図5**）。

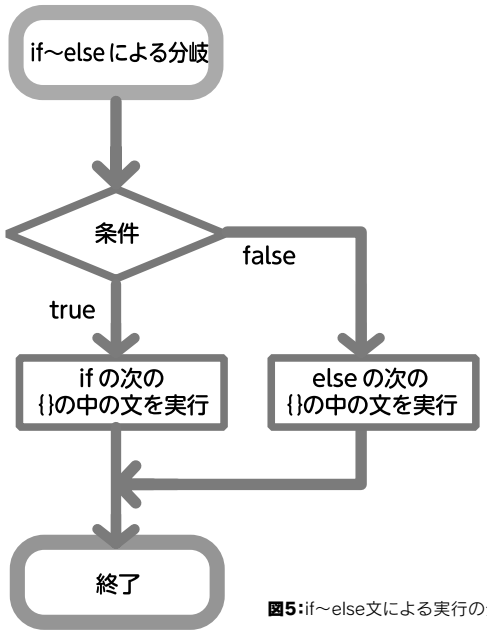


図5: if～else文による実行の分岐の様子

switchによる実行部分の選択

整数の値によって場合分けして、それぞれの場合ごとに異なる処理を実行するためには、「switch～case」文が使えます。この文は、「swith」の後の「()」の中に書いた整数式を評価し、その値（整数値）に応じて、それに続く「case」節のどこを実行するのかを決めます。それぞれのcase節には一種のラベルとして整数値を書いておきます。評価した整数値と、そのラベルの整数値の値が一致したcase節の文（複数）を実行することになります。switch～caseの一般的な形は以下のようになります。

```
switch（整数式）{
    case 整数値1:
        文1
        break;
    case 整数値2:
        文2
        break;
    case 整数値3:
        文3
        break;
    ...

    default:
        文d
        break;
}
```

case節の文は「{}」で囲う必要がありません。「break;」に出会うまで、複数の文を順に実行していきます。caseの代わりに最後にある「default」節は、評価した節数値がcase節のどのラベルの値とも一致しない場合に実行すべき部分を示します。これはオプションで、なくてもかまいません。

実際の例を見てみましょう。

```
int rInt = (int)(Math.random() * 5);
switch (rInt) {
    case 0:
        System.out.println("Low");
        break;
    case 1:
        System.out.println("Mid-Low");
        break;
    case 2:
        System.out.println("Middle");
        break;
    case 3:
        System.out.println("Mid-High");
        break;
    case 4:
        System.out.println("High");
        break;
    default:
        System.out.println("Unknown");
}
```

ここでは、まず0から4の間の乱数を発生させて、それをint型の変数「rInt」に代入しています。Javaで乱数を発生する方法は、何通りかありますが、ここでは、Mathクラスの「random()」メソッドを使っています。このメソッドは、呼ばれるたびに0.0以上1.0未満のdouble型のランダムな数を返します。それを5倍することで、0.0以上5.0未満の数に拡張し、それを「(int)」によってint型に変換しています。この

際、小数点以下は切り捨てられるので、結局rIntには、0、1、2、3、4のいずれかの整数値がランダムに入ります。

switchの後の「()」には、このrIndを直接書いているので、その値によって、それとラベルの数値が一致するcase節の文が実行されます。その値が0なら「Low」、1なら「Mid-Low」、2なら「Middle」、3なら「Mid-High」、4なら「High」と表示するようにしています。念のためにdefault節も用意していますが、この場合rIntの値が0、1、2、3、4以外の値になることはないので、この部分は永遠に実行されません。

case節の文の最後の「break;」は忘れがちです。忘れると、境界を無視して、次のcase節の文まで実行してしまいます。この性質を利用して、複数の値に対して同じ処理を実行させることもできます。例えば、rIntの値が0か1なら「Low」、2なら「Middle」、3か4なら「High」と表示するようにするには、2つのcase節の「break;」を省いて、次のようにすれば良いのです。

```
int rInt = (int)(Math.random() * 5);
switch (rInt) {
    case 0:
    case 1:
        System.out.println("Low");
        break;
    case 2:
        System.out.println("Middle");
        break;
    case 3:
    case 4:
        System.out.println("High");
        break;
    default:
        System.out.println("Unknown");
}
```

case～switch文の実行の様子をフローチャートに示します（**図6**）。これは、すべてのcase節に「break;」があり、default節もある、標準的な場合です。

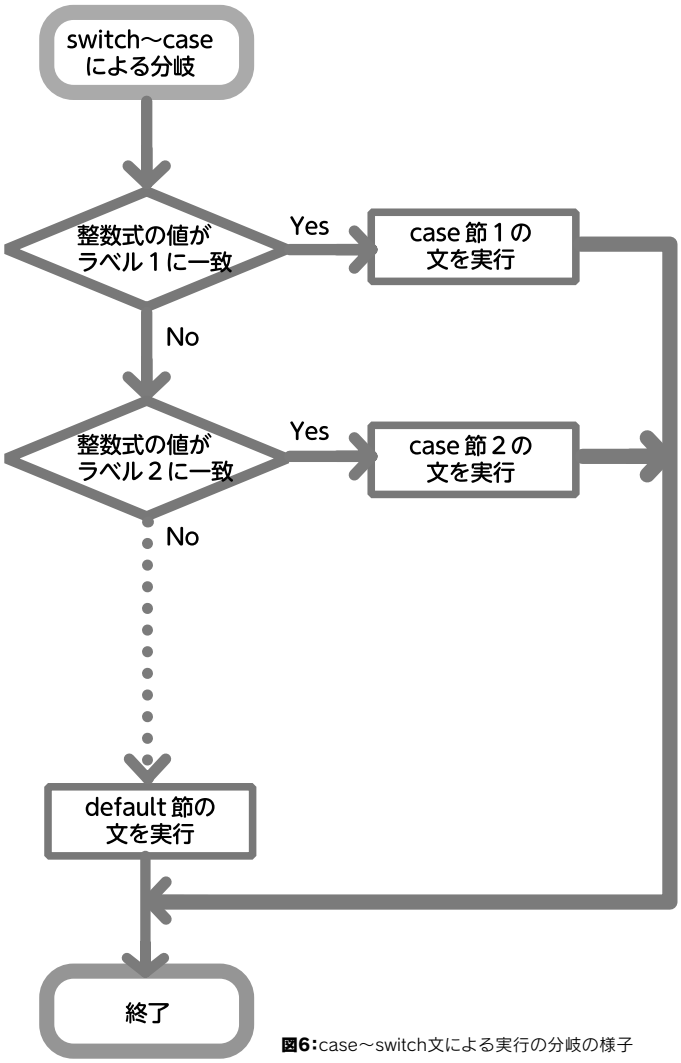


図6:case～switch文による実行の分岐の様子

5-3-3 メソッドの定義と呼び出し

メソッドとは？

Javaの「メソッド」とは、何らかの意味を持つ処理のまとまりに名前を付けておき、他の部分から呼び出して利用できるようにするためのものです。

Javaのメソッドは、かならず何かの「クラス」の中に定義します。書き方によって、そのクラスの中だけで使えるようにしたり、クラスの外部からも呼び出せるようにすることができます。1つのクラスの中には、複数のメソッドを定義することができます。また個々のメソッドごとに、外から見えるかどうかといった性格を別々に設定することができます。

メソッドは、呼び出す際に引数（パラメーター）を渡すことができ、メソッドの処理が終わった際に戻り値を返すことができます。それにより、何らかの値を渡して計算させ、その結果を呼び出し側に戻すことができます。あるいは、画面に図形を表示するような機能を持ったメソッドなら、図形の大きさや色などを引数として指定することができるでしょう。

そうした計算結果の戻り値を返す以外の機能を「副作用」と言います

メソッドの定義

メソッドの定義の一般的な形は以下のようになっています。

```
修飾子 戻り値の型 メソッド名(引数リスト) {
文
}
```

「引数リスト」の部分には、複数の引数を書くことができます。個々の引数の指定は、変数の宣言と同じように、引数の型と引数の名前を空白文字（通常は半角スペース）で区切ってペアにし、それを「,」（カンマ）で区切って並べれば良いのです。

メソッド定義の宣言部分に書く引数のことを「仮引数」と呼びます

文の部分には、そのメソッド内で実行したい文をいくらでも書くことができます。引数を返すメソッドの場合には、必ず「return」文を書く必要があります。それによって戻り値を返します。そのためには「return 戻り値;」のように書きます。「return」文を実行すると、そのメソッドの実行は終了します。

シンプルな例を見てみましょう。

「return」は、必ずしもメソッド本体の最後にあるとは限りません。また1つのメソッドには、1つだけでなく、複数の「return」文を含むことも可能です。

```
static double square(int i) {
    return (double)i*i;
}
```

このメソッド定義では、修飾子にあたるのが「static」です。戻り値の型がdoubleで、引数リストには1つだけ、int型の「i」という引数が挙げられています。メソッド本体の文は1つだけで、それは「return (double)i＊i;」というものです。これは引数として渡されたint型のiの2乗を計算し、それをdouble型として返すという意味です。

メソッドに付けることのできる修飾子は、以前にも簡単に説明しましたが、ここでも使っているstaticか、staticではないか（その場合は何も書かない）というものと、このメソッドに対する他からのアクセスを制御する修飾子の2種類です。メソッドに対して有効なアクセス制御の修飾子を以下に示します

修飾子	意　味
public	すべてのクラスからアクセス可能
protected	このクラスとそのサブクラス、同じパッケージ内のクラスからアクセス可能
なし（デフォルト）	このクラスとそのサブクラスからアクセス可能
private	このクラスからのみアクセス可能

アクセス制御の修飾子とstaticは、同時に指定できます。その際は、例えば以下のように書くことになります。

```
protected static double square(int i) {
    return (double)i＊i;
}
```

この場合、アクセス修飾子を先に書くのが普通で、その方が読みやすくなります。勧められませんが、staticを先に書くこともできます。

メソッドの呼び出し

メソッドを呼び出すには、同じクラス内で定義されたメソッドなら、メソッド名に引数リストを付けて呼び出すことができます。

```
メソッド名(引数リスト);
```

この場合の引数リストは、すでにメソッド定義によって型は決まっているので、値だけを「,」で区切って並べれば良いのです。

もし何らかの値を返すメソッドを呼び出す場合は、そのメソッドの呼び出し自体が、メソッドが返す値を持つことになります。その値は変数を用意して代入したり、他のメソッドへの引数として渡すことができます。

上で定義したメソッドを呼び出す例を見てみましょう。位置関係が分かるようにクラス全体を表示します。

このように、メソッドを呼び出す際に渡す引数のことを「実引数」と呼びます。変数を実引数として渡す場合、仮引数の名前と一致している必要はありません。実引数の値は仮引数にコピーされます

```
public class Hello {
    public static void main (String[] args) {
        System.out.println(square(3));
        double sValue = square(7);
        System.out.println(sValue);
    }

    static double square(int i) {
        return (double)i*i;
    }
}
```

この例では、2度square()メソッドを呼び出しています。最初の呼び出しでは、引数として3を与えています。戻った値は、そのままprintln()メソッドのパラメーターとして使っています。2度目の呼び出しでは、引数として7を与えています。戻った値は、いったんdouble型の変数「sValue」に代入し、その「sValue」をprintln()メソッドに渡して、値を表示しています。これを実行すると、コンソールには9.0と49.0が順に表示されます。

5-3-4 クラスの定義とインスタンス化

クラスとは？

プログラミング用語としての「クラス」（Class）は、日本語になりにくい言葉なので、そのままクラスとしています。しかし、そのままではなお意味が分かりません。一般的なクラスという言葉は、学級とか、階級と訳すことができますが、プログラミングのクラスは、そういうものではありません。いちばん近い訳語は、「種類」です。

オブジェクト指向言語であるJavaでは、プログラミングすることは、すなわちクラスを記述することだと言えます。1つのプログラムは多くのクラスから構成されているのが普通です。それを考えると、クラスはプログラムの大きな区切りを表すものだとも考えられます。実際に、1つのクラスを1つのファイルとして保存するのが普通なので、クラスはプログラムを分割して管理するためのものだと言うこともできます。

メソッドの定義のところで出てきたように、クラスの中で定義するメソッドには、アクセスを制御する修飾子を付けることができました。そのアクセス制御の単位を考えてみると、それはクラスでした。ということは、クラスはアクセスをコントロールする範囲を示すものだと考えることもできます。

Androidアプリのプログラミングでは、他のクラスで定義されたメソッドを呼び出すことも、他のクラスから呼び出されるメソッドを定義することも、独自のクラスを定義することもあります。そうした様々なクラスやメソッドの扱いは、今後この講座が進むにつれ、それぞれ必要に応じて登場し、その都度解説されることになるはずです。ここ

では、最も基本的なクラスの定義と、その利用方法を示すことにします。

クラスの定義

クラスの定義の基本的な形は、以下のようになっています。

```
修飾子 class クラス名 {
クラス本体
}
```

クラス定義の際に付けることのできる「修飾子」はpublicだけで、publicでない場合には何も付けません。

クラス本体は、「フィールド」「コンストラクター」「メソッド」、それぞれの定義から構成されています。

「フィールド」は、クラスの中で使う変数を宣言するもので、大別すると、クラス変数とインスタンス変数に分けられます。前者はスタティック変数とも言うように、staticという修飾子を付けて宣言します。クラス変数は、クラスが存在すれば、いっしょに存在する変数です。インスタンス変数は、後で述べるようにクラスからインスタンスを作成することで、初めて使えるようになるものです。

「コンストラクター」は、一種のメソッドと考えられますが、そのクラスのインスタンスを作成するためにある専用のメソッドです。コンストラクターの名前は、属するクラスの名前と同じになります。言い換えれば、クラス名と同じ名前のメソッドのことを、コンストラクターと呼ぶというわけです。

「メソッド」についてはすでに説明した通りですが、フィールドと同じようにstaticを付ければクラスに属するメソッドとなり、付けなければインスタンスに属するメソッドとなります。

Javaでは、既存のクラスのサブクラスというものを定義することができます。サブクラスは、既存のクラスから自動的にすべての特性を継承します。そのため、サブクラスの定義では、それが継承する既存のクラスと異なる部分だけを記述することになります。サブクラスから見た既存のクラスを「スーパークラス」と呼びます。

既存のクラスのサブクラスを定義する際にはextendsというキーワードを使って、以下のように書きます。

```
修飾子 class クラス名 extends スーパークラス名 {
クラス本体
}
```

また、Javaには「インターフェース」という機構があります。これはクラスに似た面もありますが、実際に動作するメソッドが定義してありません。インターフェースを使うクラスを定義する場合には、そのメソッドを自分で実装することで、外部から呼び出して

もらうことができるようになります。インタフェースを実装する際には、implementsというキーワードを使って、以下のように書きます。

```
修飾子 class クラス名 implements インターフェース名 {
クラス本体
}
```

extendsとimplementsを両方指定して、既存のクラスのサブクラスで、しかもインターフェースを実装するクラスを定義することもできます。

ここではシンプルなクラスを定義してみましょう。「長方形」を表すクラス「Rectangle」です。Eclipseを使っている場合、このクラスを記述する際には新たなクラスファイルをプロジェクトに追加する必要があります。その方法については4-1で述べた通りです。

```
public class Rectangle {
    public double width, height;

    Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double area() {
        return this.width * this.height;
    }
}
```

このクラスには、「width」、「height」というdouble型の2つのフィールドがあります。staticが指定してないので、これはインスタンス変数だということが分かります。コンストラクターは、double型の2つの引数「width」、「height」を取ります。その中身では、受け取った引数を、自分自身のフィールドのインスタンス変数「width」と「height」にコピーします。その際には、インスタンス変数の前に「this.」と付けています。この「this」は、自分自身のオブジェクトを表すキーワードです。これによって、コンストラクターに引数として渡した幅と高さの長方形オブジェクトが作成されます。また、このクラスには1つのインスタンスメソッド「area()」があります。これは、この長方形の面積を求めるものです。長方形の面積は自分自身の幅と高さで決まるので、このメソッドに引数はありません。戻り値はdouble型です。このメソッドの本体は、単に長方形の幅と高さを掛けたものを返すだけです。

インスタンスの作成と利用

クラスが定義できたら、それを利用したプログラムを書いてみましょう。まずクラスのインスタンスを作成し、次にそのインスタンスのメソッドを呼び出して結果を得るというものです。

あるクラスのインスタンスを作成するには、一般的に次のように書くことができます。

クラス名 インスタンス名 = new コンストラクタ名(引数リスト);

通常は、クラス名とコンストラクター名は同じです。クラス名は、インスタンス名で指定する変数の型と考えることもできます。ここでは、配列の時にも出てきたnewというキーワードが重要です。「引数リスト」には、コンストラクターに与える引数を、型の指定はなしで、複数の場合は「,」で区切って並べます。この作法はメソッドを呼び出す場合と同じです。

ここでは、「Rectangle」クラスのインスタンス「rect1」を作成する例を示しましょう。コンストラクターには、引数として3.2と4.8を与えます。つまり幅が3.2、高さが4.8の長方形を作ろうとしています。次のように書けばいいでしょう。

```
Rectangle rect1 = new Rectangle(3.2, 4.8);
```

ただしこれだけでは、何も起こりません。rect1という変数に、Rectangleクラスのインスタンスが代入されただけです。とはいえ、このインスタンスは、幅が3.2、高さが4.8という固有の値を持った、独立した長方形オブジェクトとなっています。また、そのクラスに定義されたインスタンスメソッドを利用することもできます。このようにオブジェクトは、固有のデータと、その処理方法を合わせ持った存在なのです。

インスタンスのメソッドを呼び出して利用するには、以下のように書きます。

インスタンス名.メソッド名(引数リスト);

ここでは、インスタンス名とメソッド名を「.」(ピリオド)で接続するのがポイントです。実際に「rect1」の「area()」メソッドを呼び出してみましょう。

```
System.out.println(rect1.area());
```

この例では、「area()」メソッドが返してきた値、つまりこの長方形オブジェクトの面積を、そのままコンソールに表示しています。結果は、15.36となるはずです。