9.3.3 文字とバイトの関係

先輩、ちょっと混乱してきました。「文字」と「バイト」って何が違うんですか？　文字だって、パソコンの中では０と１なんですよね？

では、文字とバイトの関係について整理しておこう。

朝香さんの指摘どおり、コンピュータの中ではすべての情報は０か１かで表現されます。テキストファイルもバイナリファイルも、突き詰めれば０と１の羅列でしかありません。

しかし、コンピュータが本当に０と１という情報しか扱えないわけではありません。現に私たちはJavaプログラミングで２以上の整数を扱っていますし、文字や画像のような情報も扱えることを知っています。

コンピュータの中では、**さまざまな整数や文字、あるいは色を「一定個数の０と１の並び順で表現する」**ことによって取り扱っています。たとえば、すべての文字にはそのバイト表現（０と１で表現した場合の並び）が決められています。「A」という文字には「01000001」、「7」には「00110111」が割り当てられています。

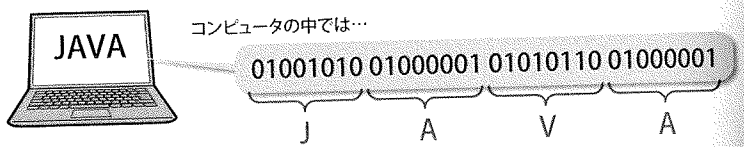

図9-5　文字「A」は、コンピュータ内では「01000001」

よって、「文字Aをファイルに書き込むこと」と「バイト01000001をファイルに書き込むこと」は**実質的に同じこと**なのです。

リスト9-1とリスト9-3は、結局は同じ動作をするんですね。

よってFileOutputStreamを使ってテキストファイルに情報を書き込むことも可能です。しかし、私たち人間にとっては「01000001を書き込む」より「'A'を書き込む」と指定できたほうが便利なため、通常はFileWriterを使うのです。

人間は文字'A'が２進数でどう表現されるかなんて普通は気にしない。とにかく'A'という「文字情報」を書き込めればよいという場合にFileWriterは最適なんだ。

9.3.4 日本語と文字コード体系

文字とバイトが対応していることはわかりました。でも、漢字は種類がたくさんあるから、１バイト（０～255）では表現できないんじゃないですか？

英語圏の人が日常的に使う文字の種類はアルファベットの大文字と小文字、数字、各種記号などを合わせても100種類程度と限られています。しかし、私たち日本人はそれら以外にもひらがな、カタカナ、漢字などたくさんの種類の文字を使います。当然、それらの文字は１バイトでは表現できません。そこで日本では「基本的に２バイトを使って１文字を表現する」方式を採用しました。

ですが困ったことが起きました。「ある文字に、どのようなバイト表現を割り当てるか」というルールが何種類も提唱されて、統一されなかったのです（次ページの図9-6）。これらのルールは**文字コード体系**（character code architecture）または単に文字コードと呼ばれ、JIS、ShiftJIS、EUC、UTF-8などがよく知られています。まったく同じ文字でも、どの文字コード体系を使うかによってバイト列としての表現はまったく異なるものになります。

ってことは、たとえばFileWriterで「あ」っていう文字を書き込むときは、どのルールを使うか指定しなければならないってことですか？

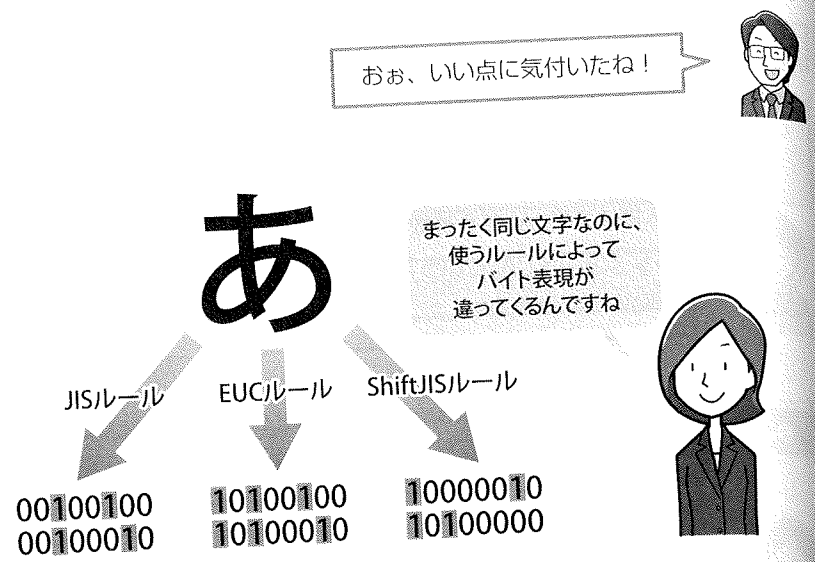

図 9-6　さまざまな文字コード体系による同一文字の表現

FileWriter は引数で与えられた文字をバイト表現に変換してファイルに書き込みます。同様に FileReader もファイルから読んだバイト表現を文字情報に変換して返します。FileReader や FileWriter では、**システム標準の文字コード体系が利用される**ことになっていますが、コンストラクタの引数で明示的に利用する文字コードを指定してファイルを読み書きすることも可能です。また文字列をバイト列に変換する String クラスの getBytes() メソッドを使う場合も、文字コードの指定が可能です。

文字コード体系の別称

一般に JIS、Shift_JIS、EUC と言われる日本語文字コード体系には、別の名前が付いていることがあります（厳密には同じ系統でも内容がわずかに異なるものがあります）。

JIS 系	: ISO-2022-JP
Shift_JIS 系	: SJIS、s-jis、MS932、CP932、Windows-31J
EUC 系	: EUC-JP、eucJP-ms、CP51932

9.4 ファイル操作の落とし穴

9.4.1 恐ろしいファイルの閉じ忘れ

ここまででファイル操作の基本的な解説は終了だよ。けれども本当に怖い落とし穴が 1 つあるから、ここで紹介しよう。絶対に忘れないでほしい。

実はファイル操作を行う場合に**陥りやすい、そして実際に陥ると致命的な不具合につながる落とし穴**があります。それは図 9-1 で紹介した「ファイル操作の基本手順」の STEP3（ファイルを閉じる）を忘れてしまうというものです。

ファイルを開いた後に閉じ忘れると、ほかのプログラムから読み書きできなくなったり、そのファイルを開けなくなったり、という現象が発生することがあります。

close() っていう 1 行を書き忘れちゃうなんて凡ミス、私はしませんよ。湊じゃあるまいし…。

いや、実はそんなに単純なことじゃないんだ。

```
public class Main {
  public static void main(String[] args) throws IOException {
    FileWriter fw = new FileWriter("c:¥¥rpgsave.dat", true);
    fw.write('A');
    fw.flush();   例外  ここで強制終了
    fw.close();
  }
}   実行されない!!
```

図 9-7　close() を書いていても実行されないケース（リスト 9-1）

close() をきちんと記述していても、実行されないことがあります（前ページ図 9-7）。たとえば、リスト 9-1 の場合、8 行目で flush() を呼び出した際に IOException が発生すると main メソッドがその場で終了してしまい、9 行目の close() が実行されません。

必ず close() されることを保証するべし

途中で return したり例外が発生する場合であっても、ファイルを開いたら、必ず閉じなければならない

9.4.2 正しい例外処理

「途中で例外が起きても必ず実行」…そうか、finally ですね！

「一度 try ブロックの処理が行われたら、その後に何があっても（途中で return しても、例外が発生しても）必ず実行される」という特徴が finally ブロックにはあります。これにより、確実にファイルを閉じることができます（図 9-8）。

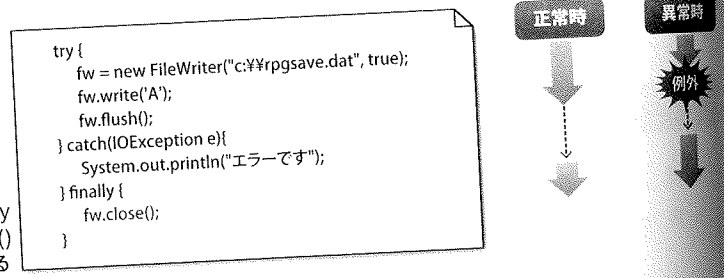

図 9-8　finally を使い、close() の実行を保証する

ファイルを確実に閉じる方法

close() は必ず finally ブロックに書く。

これまでは解説をわかりやすくするために main メソッドに throws IOException を付けて例外処理を省略していました。しかし実際の開発ではリスト 9-4 のような正しい例外処理を必ず行うようにしてください。

リスト 9-4　正しく例外処理を行うプログラム（リスト 9-1 を改訂）

Main.java

```
1  import java.io.*;
2
3  public class Main {
4    public static void main(String[] args) {
5      FileWriter fw = null;
6      try {
7        fw = new FileWriter("rpgsave.dat", true);
8        fw.write('A');
9        fw.flush();
10     } catch(IOException e) {
11       System.out.println("ファイル書き込みエラーです");
12     } finally {
13       if(fw != null) {
14         try {
15           fw.close();
16         } catch(IOException e2) { }
17       }
18     }
19   }
20 }
```

try ブロックの外で宣言し null で初期化しないと、finally ブロック内で close() を呼べない

ファイルを閉じるための finally ブロック

close() が IOException を送出する可能性があるため、再度 try-catch が必要。ただし失敗しても何もできないため catch ブロック内は空にしてある

Java7 における便利な記法

従来の Java6 までは、前ページのリスト 9-4 のような多少、冗長な記述が必要でした。しかし、Java7 では「close() が呼ばれる try-catch 構文」が追加され、以下のようにエレガントに記述することが可能になっています。

```
try(FileWriter fw = new FileWriter("rpgsave.dat");) {
  /* 正常処理 */
} catch(IOException e) {
  /* 例外処理 */
}
```

このように finally や close() を記述しなくてもよいほか、try-catch のネストも不要になります。

ディレクトリの指定方法

FileWriter などのコンストラクタ引数で指定するファイル名は、絶対パス、相対パスのいずれでも記述できます。

```
"c:¥¥data¥¥rpgdata.txt"   ← 絶対パス指定
"rpgdata.txt"             ← 相対パス指定
```

なお、相対パスの場合には Java プログラムを起動した際のカレントフォルダ の中にファイルが作成されます。

9.5 ストリームの概念

9.5.1 ストリームとは？

ありがとうございました！　さっそくセーブ機能を作ってみます！

まぁ焦らずに。確かに API の使い方は理解しただろうけど、「考え方」も理解するとスキルの幅がさらに広がるよ。

JVM の中で動くプログラムがファイルにアクセスする際、FileReader やそのほかのクラスを使えばよいことは十分理解できたでしょう。これらのクラスはすべて、データを**少しずつ読んだり書いたりする**ためのものでした。

みなさんの中には、「ファイルの中身を全部一気に読み込んでくれる次のような命令があればいいのに」と思われる人もいるかもしれません。

```
String data = MyFileReader.readAll("rpgsave.dat");
```

しかし、ここで想像してみてください。もし読み込もうとしているファイルが偶然 10GB もの巨大なファイルだったらどうなってしまうでしょうか？ 変数 data に 10GB 分のデータを読み込もうとしても、おそらく途中でメモリ不足に陥りプログラムは異常終了してしまいます。仮にメモリが足りるとしても、読み込み処理の完了まで長時間待たされてしまうでしょう。

この問題はファイルに限った話ではありません。たとえば、インターネット上のデータをネットワーク経由で読み込むようなプログラムにも、同様の懸念があります。

そもそも JVM が完全に管理下に置いているデータ格納領域はメモリだけです。JVM の外の世界にあるデータ置き場であるファイルやネットワーク上には、「そもそもメモリに入らないほどの巨大なデータ」や「実際に読んでみなければ、ど

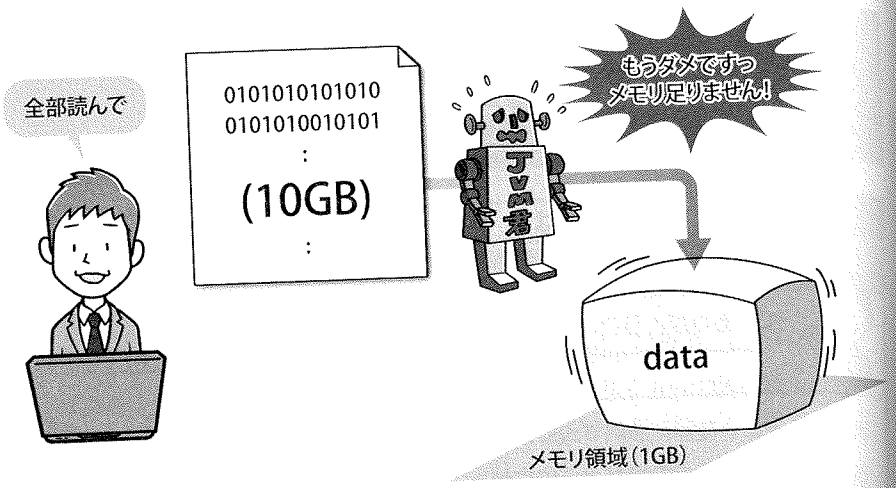

図 9-9　ファイルを一度に読み込むと、OutOfMemoryError が発生

れだけのサイズかわからないデータ」だらけです(図 9-9)。だからこそ、JVMの外部にあるデータはシーケンシャルに少しずつ処理するのが基本的なアプローチなのです。

JVM 外部とのデータのやりとり

JVM の外部にあるデータは、少しずつ読み書きするのが基本。

このアプローチに沿ったデータの読み書きのようすを、プログラミングの世界では図 9-10 のように**ストリーム**(stream)という概念で捉えます。

このイメージに沿って考えれば、FileReader、FileWriter、FileInputStream、FileOutputStream の 4 クラスはそれぞれ図 9-11 のようなイメージで考えることができます。前者 2 つのように文字が流れるストリームは**文字ストリーム**(character stream)、後者 2 つのバイト列が流れるストリームは**バイトストリーム**(byte stream)と呼ばれます。

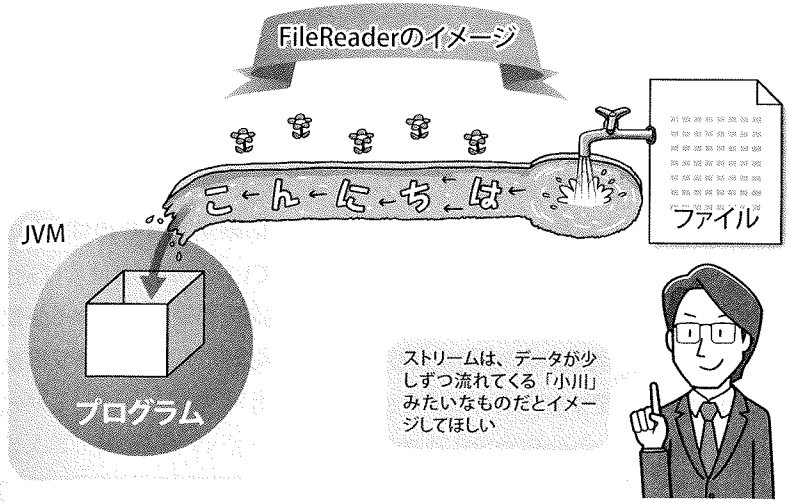

図 9-10　ストリームという「小川」をデータが流れてくる

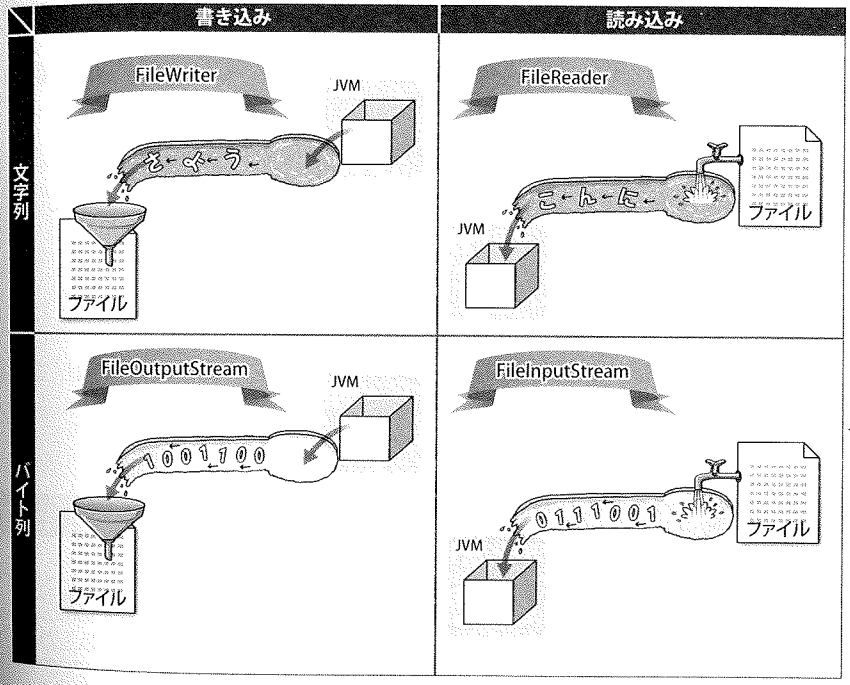

図 9-11　ファイル操作に関連する 4 つのストリームのイメージ

9.5.2 さまざまなストリーム

コンピュータの世界ではストリームの概念が広く用いられてきました。なぜなら**単なるデータの読み書きを小川として抽象的に捉えることで、さまざまな応用や発展が考えられる**からです。

たとえば、「小川の終端は必ずしもファイルだけに限定する必要はない」と先人は考えました。「ネットワークサーバーが繋がっている小川」を使えば通信を、「キーボードが繋がっている小川」を使えばキー入力を、「ディスプレイが繋がっている小川」を使えば画面への文字の表示をまったく同じアプローチで実現できることに気付いたのです。

なるほど！　通信も表示も入力もファイル操作も、「ザックリみれば似たような処理」なんですね！

昔の人は頭いいなぁ…そんなこと、思いつきませんでした。

ネットワークサーバーがつながっているストリームの使い方については、第11章で詳しく紹介しましょう。ここでは、その他の種類のストリームについて紹介していきます。

標準出力、標準エラー出力、標準入力

JVMは起動直後に表9-4のような3つのストリームを自動的に準備しています。

表9-4　JVMの起動と同時に自動的に準備されるストリーム

名称（通称）	標準の接続先	用途
標準出力（stdout）	ディスプレイ	通常の画面表示
標準エラー出力（stderr）	ディスプレイ	エラー情報の画面表示
標準入力（stdin）	キーボード	キーボードからの入力

標準出力（standard out）と**標準エラー出力**（standard error）はどちらもディスプレイにつながるストリームであり、それぞれSystem.outとSystem.errとして準備されています。私たちがずっと使ってきているSystem.out.println()とは、「ディスプレイにつながる小川に文字を流す命令」だったわけですね。

一方の**標準入力**（standard in）は上流にキーボードがつながっているストリームであり、System.inにその実体が準備されます。実際、System.in.read()を実行することで、キーボードからの入力を1バイトずつ読み込むことができます。

文字列やバイト配列を終端に持つストリーム

通常ストリームはJVM外部のデータを少しずつ読み書きする際に利用しますが、JVM内部の変数の読み書きにも応用することが可能です。

たとえば、java.ioパッケージに準備された**StringReaderクラス**は、指定したString型変数から1文字ずつ読み取るための機能を提供します（リスト9-5）。このクラスはFileReaderクラスと似ていますが、終端がファイルではなくコンストラクタで指定された文字列です。なお、StringReaderクラスはFileReaderクラスと同じくjava.io.Readerを継承しています。

リスト9-5　文字列型の変数から1文字ずつ読み込む

Main.java

```
1  import java.io.*;
2
3  public class Main {
4    // throwsで例外処理を省略しています
5    // 実際の開発では正しく例外処理を行ってください
6    public static void main(String[] args) throws IOException {
7      String msg = "第1土曜";
8      Reader sr = new StringReader(msg);
9      char c1 = (char) sr.read();  ← c1には「第」が入る
10     char c2 = (char) sr.read();  ← c2には「1」が入る
11       :
12   }
13 }
```

参考：文字列に1文字ずつ書き込むStringWriterもある

同様に、バイト配列に対して1バイトずつ順次書き込んでいくために、ByteArrayOutputStream クラスが準備されています。書き込みが完了したところで toByteArray() メソッドを呼び出せば、バイト配列が得られます（リスト 9-6）。

リスト 9-6　バイト配列に値を書き込む

Main.java

```
1  import java.io.*;
2
3  public class Main {
4    // throwsで例外処理を省略しています
5    // 実際の開発では正しく処理してください
6    public static void main(String[] args) throws IOException {
7      ByteArrayOutputStream baos = new ByteArrayOutputStream();
8      baos.write(65);
9      baos.write(66);
10     byte[] data = baos.toByteArray();
11     // dataは「65」「66」が入った要素数2のbyte型配列
12       :
13   }
14 }
```

参考：バイト配列から 1 バイトずつ読み込む ByteArrayInputStream もある

APIリファレンスを調べたところ、ByteArrayOutputStreamもFileOutputStreamも、java.io.OutputStream クラスを継承しているんですね。

ストリーム関連クラスは、Reader、Writer、InputStream、OutputStream のいずれかを継承しているんだ。もちろん、それらを継承した自分オリジナルのストリームクラスを作ることも可能だよ。

標準出力の接続先をファイルに変更する

本文では「標準出力は自動的にディスプレイに接続される」と解説しました。しかし、標準入出力が実際に何に接続されるかは Windows などの OS が管理をしており、その標準の接続先がディスプレイであるに過ぎません。この接続先はプログラムを起動する際に接続先を指定し変更することも可能です。

たとえば、以下のように「> ファイル名」を末尾に記述することで標準出力をそのファイルに繋いで Java プログラムの出力結果をファイルに出力することができます。この機能を**リダイレクト**（redirect）と呼びます。

```
>java Main > data.txt
```

また、**パイプ**（pipe）という機能を使えば、あるプログラムの標準出力をリアルタイムに別プログラムの標準入力に流し込むことも可能です。

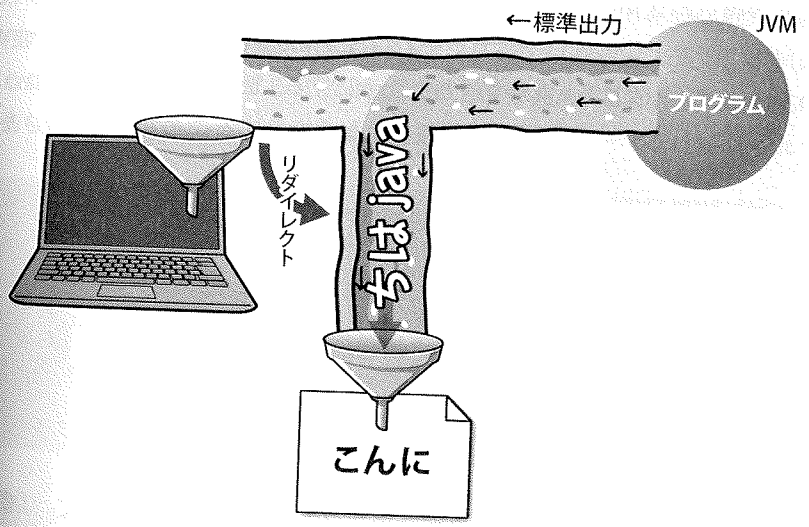

図 9-12　System.out.println() の出力がリダイレクトされるイメージ