1.4 正規表現の活用

1.4.1 パターンマッチング

どうしたんだい。文字列操作で何か困っているのかな？

実は、僕が作っているゲームの「プレイヤー名入力チェック」を書こうとしているんですが、とてもややこしいプログラムになりそうで…。

湊くんが作ろうとしているゲームでは、開始直後に次の条件を満たすプレイヤー名を入力させることにしています。

プレイヤー名の条件

・必ず８文字で、使える文字はＡ～Ｚと０～９だけ。
・最初の文字に数字は使えない。

たとえば、「MINATO01」「ASAKA001」は正しいプレイヤー名です。もしユーザーが入力したプレイヤー名が７文字しかなかったり、ひらがなが含まれていたりした場合、再入力を求めなければなりません。

そのために必要となる「入力チェック判定を行うメソッド」をいざ記述しようとすると、これがかなり大変な作業であることがわかります（リスト 1-5）。

リスト 1-5　正当なプレイヤー名であるかを判定するメソッド

```
1  boolean isValidPlayerName(String name) {
2    if(name.length() != 8) {          ← 文字数が８文字であること
3      return false;
4    }
5    char first = name.charAt(0);
6    if(!(first >= 'A' && first <= 'Z')) {          ← 最初の１文字はＡ～Ｚ
7      return false;
8    }
9    for(int i = 1; i < 8; i++) {
10     char c = name.charAt(i);
11     if(!((c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9'))) {
12       return false;                  ← 以降の文字はＡ～Ｚか０～９
13     }
14   }
15   return true;
16 }
```


確かにかなり大変そう。相手が人間であれば、次の図 1-6 みたいなメモを渡して「この条件でチェックして」って気軽に頼めるのに…。

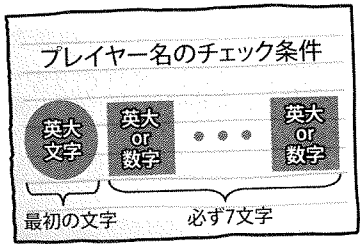

図 1-6　人間にチェックを頼む場合に渡す「プレイヤー名のチェック条件」

実は Java でも、図 1-6 のような形式でチェックを指示する方法があるんだよ。

それでは、実際にリスト 1-5 をよりエレガントな形に書き直してみましょう。String クラスの matches() メソッドを用いることで、複雑だったメソッドがわずか数行になっている点に注目してください（リスト 1-6）。

リスト 1-6　文字列パターンを用いたプレイヤー名のチェック

```
boolean isValidPlayerName(String name) {
  return name.matches("[A-Z][A-Z0-9]{7}");
}
```

文字列パターン

プレイヤー名が条件に一致しているかの判定は、String クラスの matches() メソッドのたった一度の呼び出しだけで完了しています。引数で与えている "[A-Z][A-Z0-9]{7}" という文字列は、**文字列パターン**(string pattern)または単にパターンといわれるもので、図 1-6 のような「文字列の形式的な条件」を**正規表現**(regular expression)という文法に従って記述したものです。

また、matches() メソッドのように、文字列がパターンに従った形式を満たしている(これを「マッチする」といいます)かを照合する処理のことを**パターンマッチング**(pattern matching)といいます。

文字列パターンはちょっと奇妙で呪文みたいですが、これだけであの複雑な条件をすべて指定していると思うとスゴイです。

そうだね。この正規表現はとても便利で、本格的なプログラム開発には欠かせないから、簡単に文法を紹介しておこう。

1.4.2　正規表現の基本文法

正規表現では、いくつかの特殊な記号を使って文字列パターンを指定します。次に、パターンに含まれる文字にどのような意味があるかを解説していきましょう。

①通常の文字：その文字でなければならない

パターン内に記述されたアルファベットや数字、ひらがな、カタカナ、漢字のような一般的な文字は、基本的にそれと同じ文字を表します。

たとえば、"ABC" というパターンは「1 文字目が A、2 文字目が B、3 文字目が C であること」という条件を示しています。以下は "Java" という文字列にマッチしているかどうかを調べる例です。

```
String s = "Java";
s.matches("Java")          // => true
s.matches("JavaJava")      // => false
s.matches("java");         // => false
```

②ピリオド：任意の 1 文字であればよい

パターン中にピリオド記号(.)があった場合、その部分には任意の 1 文字(何でもよいので必ず 1 文字)があればよいという意味です。

たとえば以下の "J.va" というパターンは「1 文字目が J、2 文字目は何でもよい、3 文字目は v、4 文字目は a」という条件を示しています。

```
"Java".matches("J.va")          // => true
```

③アスタリスク：直前の文字の 0 回以上の繰り返し

パターン中にアスタリスク記号(*)が含まれていた場合、その直前の文字の 0 回以上の繰り返しを意味します。

たとえば、"AB*" というパターンは、「1 文字目は A、2 文字目は B、それ以降は B が何文字でも続いてもよい」という形式を示しますので、「AB」や「ABBB」、「ABBBBBBB」などが条件を満たします。

```
"Jaaaaava".matches("Ja*va")    // => true
"あいうxx019".matches(".*")    // => true
```

上記の ".*" という正規表現は「任意の 1 文字を 0 回以上繰り返し」ですので、**「すべての文字列を許す」**という指示になります。これは正規表現でよく利用される慣用句的な表現です。また、このアスタリスク記号は 1.2.2 節で説明した endsWith() や startsWith() の代わりとして文字列の判定に利用できます。その際には次のように記述します。

```
s.matches("Ma.*")      // Maで始まる任意の文字
s.matches(".*ful")     // fulで終わる任意の文字
```

④波カッコ：指定回数の繰り返し

パターン中に波カッコで囲まれた数字が登場した場合、それは直前の文字の指定回数の繰り返しを意味します。たとえば、"HEL {3} O" というパターンは、"HELLLO" というパターンと同じ意味です。

そのほか、表 1-5 のような方法でさまざまな繰り返し回数を指定できます。

表 1-5　正規表現における繰り返し回数の指定方法

パターン記述	意味
{n}	直前の文字の n 回の繰り返し
{n,}	直前の文字の n 回以上の繰り返し
{n,m}	直前の文字の n 回以上 m 回以下の繰り返し
?	直前の文字の 0 回または 1 回の繰り返し
+	直前の文字の 1 回以上の繰り返し

⑤角カッコ：いずれかの文字

パターン中に角カッコ記号（[ ]）で囲まれた部分がある場合、角カッコの中のどれか 1 文字に当てはまることを要求する意味となります。

たとえば、"UR [LIN] " というパターンは、「1 文字目が U、2 文字目が R、3 文字目が L か I か N であること」を意味します。

⑥角カッコ内のハイフン：指定範囲のいずれかの文字

角カッコ中にハイフン記号（－）が含まれる場合、その両端にある文字を含む範囲の任意の 1 文字であることを意味します。

次の例にある "[a-z]" というパターンは a ～ z のいずれかの文字（つまりすべてのアルファベット小文字）とマッチします。例では "url" という文字列と a ～ z のいずれか 3 文字を比較していますので、当然マッチし、結果は true となります。

```
"url".matches("[a-z]{3}")      // => true
```


なお、すべての数字（[0-9]）などパターンで多用されるものは、¥ で始まる次の**文字クラス**としてあらかじめ定義されています。

表 1-6　定義済みの文字クラスの例

パターン記述	意味
¥d	いずれかの数字　([0-9] と同じ)
¥w	英字·数字·アンダーバー　([a-zA-Z_0-9] と同じ)
¥s	空白文字　(スペース、タブ文字、改行文字など)

¥ という記号自体を文字として含めたい場合は ¥¥ を、その他 [や * などの特殊記号を文字として含めたい場合は ¥[や ¥* を使ってください。

⑦ハットとダラー：先頭と末尾

パターン中のハット記号（^）は文字列の先頭を、ダラー記号（$）は文字列の末尾を表します。

たとえば " ^ j.*p$" というパターンは、「先頭文字が j で、最後の文字が p の任意の長さの文字列」を意味します。matches() の場合、これらがなくても同様に動作しますが、後述する split() や replaceAll() で先頭や末尾を明示的に示すために利用します。

今回紹介したのは正規表現のほんの一部でしかない。さらにたくさんの正規表現構文を使いこなすと、驚くほどエレガントなコードを書けるようになるだろう。

1.4.3　正規表現を用いたほかの処理

正規表現パターンで行えることは、文字列照合だけにとどまりません。パターンを使うことで、「文字列の分割」や「置換」もより効率的に記述できます。

split() メソッド：文字列の分割

String クラスの split() メソッドを使うと、1 つの文字列を複数に分割できます。たとえば、次のようなコードを記述すれば、簡単に「カンマかコロンの場所」で

文字列を分割できます。

リスト 1-7　split() メソッドを使った文字列の分割

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     String s = "abc,def:ghi";
4     String[] words = s.split("[,:]");
5     for(String w : words) {
6       System.out.print(w + "->");
7     }
8   }
9 }
```

正規表現パターン

このコードを実行すると、以下の結果が出力されます。

```
abc->def->ghi->
```

replaceAll() メソッド：文字列の置換

String クラスの replaceAll() メソッドを使うと、文字列中でパターンに一致した箇所を別の文字列に置換できます。以下は "beh" の３文字いずれかに当てはまった文字を "X" に置換するコードの例です。

リスト 1-8　replaceAll() メソッドを使った文字列の置換

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     String s = "abc,def:ghi";
4     String w = s.replaceAll("[beh]","X");
5     System.out.println(w);  // => aXc,dXf:gXi
6   }
7 }
```

最初の１つだけを置換する replaceFirst() もある

1.5 文字列の書式整形

1.5.1 桁を揃えた表示

ちょっとミナト。あなたのゲーム画面、表示がガタガタなのどうにかならないの？

だってぇ、しょうがないじゃないか。揃えようとするととても大変なんだよ。

朝香さんが指摘していたのは、次のようなゲームの画面です。

```
■キャラクターの状態■
minato hero  所持金280
asaka witch  所持金32000
sugawara sage  所持金41000
```

行ごとにキャラクターの名前や職業などの文字列長が違うため、表示の桁が揃っていません。でも、本当は以下のように整形された表示にできれば理想的ですね。

```
■キャラクターの状態■    桁の位置が揃っている
minato     hero         所持金    280
asaka      witch        所持金32,000
sugawara   sage         所持金41,000
```

右揃えでカンマ入り

length() メソッドを使い自力で空白を入れて桁を揃えることも不可能ではありませんが、かなり大変です。幸い String クラスには、整形した文字列を組み立

てるための静的メソッド format() が準備されています。

まずは format() メソッドを使った簡単な例から見てみましょう。図 1-7 は、「～日で～わかる～入門」というひな形を準備し、各「～」の部分に数字や文字列を流し込んで文字列を組み立てる例です。

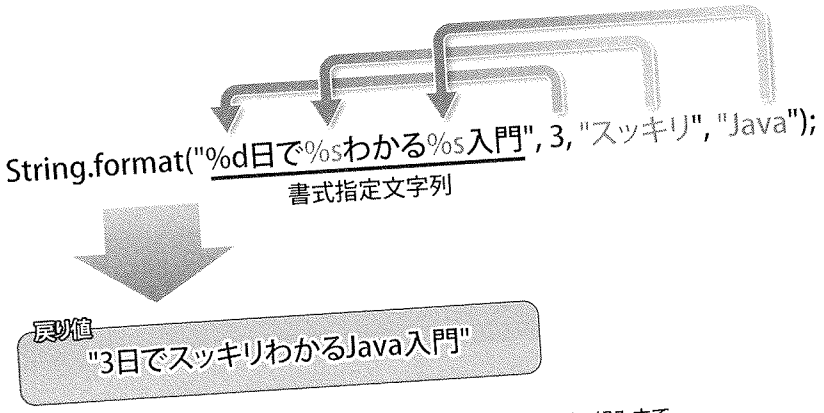

図 1-7　format() による書式を指定した文字列の組み立て

%d とかまた難しそうな文字列が出てきたぞ？　これも正規表現なんですか？

いや、正規表現とはまた違った独自の文法で指定するんだよ。

format() メソッドの第 1 引数に指定するのは、組み立てる文字列のひな形を指定する**書式指定文字列**と呼ばれるもので、専用の記法を用いて記述します。特に % 記号の部分は**プレースホルダ**（place holder）と呼ばれ、第 2 引数以降で指定した具体的な値が順に流し込まれる場所となります。

プレースホルダは、次ページ図 1-8 の文法に従って記述します。たとえば、1.5 節冒頭のゲーム画面の場合、次ページのリスト 1-9 のように記述することで、きれいに整形して画面表示を行うことができるでしょう。

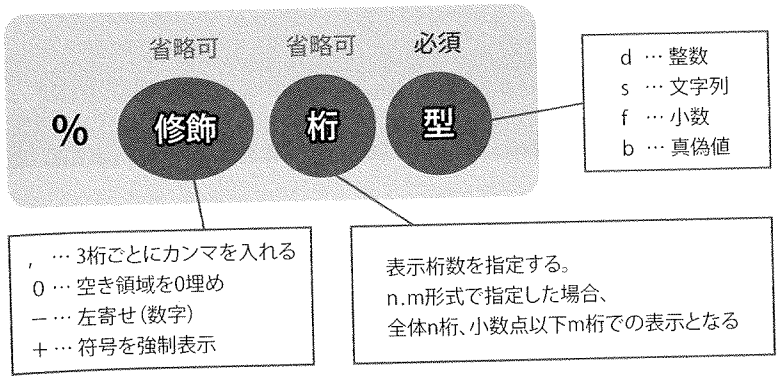

図 1-8　プレースホルダの書式

リスト 1-9　キャラクター状態 1 人分の表示のためのコード

```
1 final String FORMAT = "%8s  %6s  所持金%,5d";
2 String s = String.format
      (FORMAT, hero.getName(), hero.getJob(), hero.getGold());
3 System.out.println(s);
```

ただし、プレースホルダは日本語（全角文字）が混ざると崩れてしまうこともあるから注意して使ってほしい。

なお、String.format() を使って文字列を作ると同時に画面に出力したい場合は、代わりに System.out.printf() メソッドを使うと便利です。引数の指定方法は format() と同じです。

System.out.printf（）の構文

System.out.printf（書式文字列 , パラメータ…）;

```
System.out.printf("製品番号%s-%02d", "SJV", 3);
```

上記のコードを実行すると、以下のように整形されて表示されます。

```
製品番号SJV-03
```

可変長引数

format() や printf() メソッドには、必要に応じて引数をいくつでも渡すことができます。このようなことが可能なのは、これらメソッドが**可変長引数**というしくみを使って宣言されているからです。

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

このように引数リストの型の後ろにピリオドを３つ並べると、その型の引数をいくつでも渡せるようになります。ただし、１つのメソッドの宣言にあたり「...」を使えるのは１回だけです。

可変長引数に渡された値は、配列として取り出すことができます。上記の例では、args 自体は Object 配列として扱われ、args [0] や args [1] として実引数を取得することができます。

なお、「Object…」型は内部的には Object[] として扱われています。そのため、実引数として Object[] や String[] などが渡された場合に、複数の引数が一度に与えられたのか、１つの配列が与えられたのか、JVM は判別することができません。そこで、実引数を１つの配列として渡したい場合には、Object 型にキャストすることになっています。

1.6 この章のまとめ

String クラスが備えるさまざまなメソッド

分類	操作	メソッド名
調査	内容が等しいかを調べる	equals
	ケース（大文字、小文字）を区別せず等しいかを調べる	equalsIgnoreCase
	文字列長を調べる	length
	空文字かを調べる	isEmpty
検索	ある文字列を一部に含むかを調べる	contains
	ある文字列で始まるかを調べる	startsWith
	ある文字列で終わるかを調べる	endsWith
	最初の登場位置を調べる	indexOf
	最後の登場位置を調べる	lastIndexOf
	正規表現を用いて照合する	matches
切り出し	指定位置の１文字を切り出す	charAt
	指定位置から文字列を切り出す	substring
変換	大文字を小文字に変換する	toLowerCase
	小文字を大文字に変換する	toUpperCase
	前後の空白を除去する	trim
	文字列を置換する	replace
	正規表現を用いて置換する	replaceAll
分割	正規表現を用いて分割する	split
整形	書式指定文字列を使い整形する	format

高速な文字列連結

・多数回の文字列連結には、StringBuilder や StringBuffer を用いる。

1.7 練習問題

練習 1-1

1 から 100 までの整数をカンマで連結した以下のような文字列 s を生成するコードを作成してください。

1,2,3,4,5,6,7…98,99,100,

また、完成した文字列 s をカンマで分割し、String 配列 a に格納してください。

練習 1-2

フォルダ名が入っている変数 folder と、ファイル名が入っている変数 file があります。file は必ず「readme.txt」のような形式をしていますが、folder は末尾に ¥ 記号が付いている場合と付いていない場合の両方がありえます。たとえば、「c:¥javadev」や「c:¥user¥」のどちらも folder の値として考えられます。

folder と file を連結して、「c:¥javadev¥readme.txt」のような完全なファイル名としての文字列を完成させるメソッドを作成してください。

練習 1-3

以下の各条件とマッチする正規表現パターンを記述してください。

(1)すべての文字列
(2)最初の 1 文字は A、2 文字目は数字、3 文字目は数字か無し
(3)最初の 1 文字は U、2 ～ 4 文字目は英大文字

1.8 練習問題の解答

練習 1-1 の解答

Main.java

```
public class Main {
  public static void main(String[] args) {
    StringBuilder sb = new StringBuilder();
    for(int i = 0; i < 100; i++) {
      sb.append(i+1).append(",");
    }
    String s = sb.toString();
    String[] a = s.split(",");
  }
}
```

練習 1-2 の解答

```
String concatPath(String folder, String file) {
  if(!folder.endsWith("¥¥")) {
    folder += "¥¥";
  }
  return folder + file;
}
```

練習 1-3 の解答

(1).*　(2)A¥d{1,2}　(3)U[A-Z]{3}