正規表現

正規表現とは、文字列内で文字の組み合わせを照合するために用いられるパターンです。JavaScript では、正規表現はオブジェクトでもあります。これらのパターンは RegExp の exec および test メソッドや、String の match、replace、search、および split メソッドで使用できます。本章では、JavaScript の正規表現について説明します。

正規表現の作成

正規表現は 2 種類の方法で作成できます：

次のように、スラッシュによって囲まれたパターンからなる正規表現リテラルを使用する：

var re = /ab+c/;
正規表現リテラルはスクリプトのロード時にその正規表現をコンパイルします。正規表現が一定のままの場合、この方法を使用するとよいパフォーマンスが得られます。

また次のように、RegExp オブジェクトのコンストラクタ関数を呼び出す方法があります：

var re = new RegExp("ab+c");
コンストラクタ関数を使用すると、実行時にその正規表現をコンパイルします。正規表現パターンが変わることがわかっている場合や、パターンがわからない場合、ユーザーが入力するなど別のソースからパターンを取得する場合は、コンストラクタ関数を使用してください。

正規表現パターンの記述

正規表現パターンは、/abc/ のような単純な文字、または /ab*c/ や /Chapter (¥d+)¥.¥d*/ のような単純な文字と特殊文字との組み合わせからなります。最後の例には記憶装置として用いられる丸括弧があります。パターンのこの丸括弧で囲まれた部分でマッチした箇所は、後で使用できるように記憶されます。詳しくは括弧で囲まれた部分文字列のマッチの使い方を参照してください。

単純なパターンの使い方

単純なパターンとは、直接マッチしている部分を見つけたい文字から構成されたものです。例えば /abc/ というパターンは、実際に 'abc' という文字が一緒にその順で存在しているときだけ、文字列中の文字の組み合わせにマッチします。"Hi, do you know your abc's?" や "The latest airplane designs evolved from slabcraft." といった文字列でのマッチは成功します。どちらの場合でも 'abc' という部分文字列にマッチします。"Grab crab" という文字列では、'abc' という部分文字列が含まれていないためマッチしません。

特殊文字の使い方

1 個以上の b を見つけたり、ホワイトスペースを見つけたりといった直接マッチより高度なマッチの検索では、パターンに特殊文字を使用します。例えば /ab*c/ というパターンでは、1 個の 'a' とその後ろに続く 0 個以上の 'b' (* は直前のアイテムの 0 回以上の出現を意味します)、そしてそのすぐ後ろに続く 'c' で構成される文字の組み合わせにマッチします。"cbbabbbbcdebc," という文字列では、このパターンは 'abbbbc' という部分文字列にマッチします。

以下の表で、正規表現で使用できる特殊文字とその意味を詳しく説明します。

正規表現における特殊文字

文字	意味
¥	以下のルールに基づいてマッチします：

特別な意味を持たない文字の前に付けられたバックスラッシュ文字は、次の文

特別な意味を持たない文字の前に付けられたバックスラッシュ文字は、次の文字が特別なもので、文字通りには評価されないことを表します。
例えば b は文字列中にある小文字の 'b' にマッチします。しかし b の前にバックスラッシュを置いて ¥b とすると、これはどんな文字にもマッチしません。これは単語区切り文字を意味する特殊文字になります。

特別な意味を持つ文字の前に付けられたバックスラッシュ文字は、次の文字が特別なものでなく、文字通りに評価されることを表します。
例えば、パターン /a*/ は特殊文字 '*' の働きによって 0 個以上の a にマッチします。対照的に、パターン /a¥*/ では '*' の特殊性は削除され、'a*' と行った文字列のマッチが可能になります。

バックスラッシュ文字 (¥) 自身も文字列内ではエスケープ文字となるため、RegExp("pattern") 記法を使う際は ¥ 自身のエスケープを忘れないようにしてください。

^ **入力の先頭にマッチします。複数行フラグが true にセットされている場合は、改行文字の直後にもマッチします。**

例えば /^A/ は "an A" の 'A' にはマッチしませんが、"An E" の 'A' にはマッチします。

この文字は、文字集合パターンの先頭にある場合は異なる意味を持ちます。例と詳細については相補文字集合をご覧ください。

$ **入力の末尾にマッチします。複数行フラグが true にセットされている場合は、改行文字の直前にもマッチします。**

例えば /t$/ は "eater" の 't' にはマッチしませんが、"eat" の 't' にはマッチします。

* **直前の文字の 0 回以上の繰り返しにマッチします。**{0,} に相当します。

例えば /bo*/ は "A ghost booooed" の 'boooo' や "A bird warbled" の 'b' にマッチしますが、"A goat grunted" ではマッチしません。

+ **直前の文字の 1 回以上の繰り返しにマッチします。**{1,} に相当します。

例えば /a+/ は "candy" の 'a' や "caaaaaaandy" のすべての a にマッチします。

? **直前の文字の 0 回か 1 回の出現にマッチします。**{0,1} に相当します

.	**直前の文字のゼロ個か一個の出現にマッチします。**{0,1} に相当します。 例えば /e?le?/ は "angel" の 'el' や "angle" の 'le'、あるいは "oslo" の 'l' にマッチします。 *、+、?、{} といった量指定子の直後に使用した場合、その量指定子をデフォルトとは逆の非貪欲 (non-greedy) (最短) マッチにします。デフォルトは欲張り (greedy) (最長) マッチです。 例えば /¥d+/ は非グローバルマッチで "123abc" の "123" にマッチしますが、/¥d+?/ の場合は "1" だけにマッチします。 この特殊文字は、この表の x(?=y) および x(?!y) の項目で説明する先読みアサーションでも使用できます。
.	**改行文字以外のどの 1 文字にもマッチします。** 例えば /.n/ は "nay, an apple is on the tree" の 'an' や 'on' にはマッチしますが、'nay' にはマッチしません。
(x)	**'x' にマッチし、マッチした内容を記憶します。**この括弧はキャプチャリング (格納) 括弧と呼ばれます。 例えば、パターン /(foo) (bar) ¥1 ¥2/ 内の '(foo)' と '(bar)' は、文字列 "foo bar foo bar" の最初の 2 個の単語にマッチし、それを記憶します。パターン内の ¥1 と ¥2 は文字列の最後の 2 個の単語にマッチします。 ¥1, ¥2, ¥n は正規表現のマッチ部分で使用することに注意してください。置換部分で使用する際は $1, $2, $n とする必要があります、例えば 'bar foo'.replace(/(...) (...)/, '$2 $1') というように。$& はマッチした文字列全体を意味します。
(?:x)	**'x' にマッチしますが、マッチした内容は記憶しません。**この括弧は非キャプチャリング (非格納) 括弧と呼ばれ、パターンをグルーピングして、正規表現演算子と一緒に使う際の部分正規表現式を定義することができます。 見本として式 /(?:foo){1,2}/ を見てみましょう。式が /foo{1,2}/ であれば、{1,2} の文字は 'foo' の最後の 'o' にのみ適用されます。非キャプチャリング括弧を使うと、{1,2} は 'foo' という単語全体に適用されます。詳しい情報は、下記の括弧を使うを見てください。
文字	意味
x(?=y)	**'x' に 'y' が続く場合のみ 'x' にマッチします。**この特殊文字は先読みと呼ばれます。

例えば /Jack(?=Sprat)/ は 'Jack' の後に 'Sprat' が続く場合のみ 'Jack' にマッチします。/Jack(?=Sprat|Frost)/ は 'Jack' の後ろに 'Sprat' または 'Frost' が続く場合のみ 'Jack' にマッチします。しかしながら、'Sprat' も 'Frost' もマッチの結果には表れません。

x(?!y) **'x' に 'y' が続かない場合のみ 'x' にマッチします。**これは否定先読みと呼ばれます。

例えば /¥d+(?!¥.)/ は後ろに小数点が続かない数値にマッチします。正規表現 /¥d+(?!¥.)/.exec("3.141") は '141' にマッチしますが '3.141' にはマッチしません。

x|y **'x' または 'y' にマッチします。**

例えば /green|red/ は "green apple" の 'green' や "red apple" の 'red' にマッチします。'x' と 'y' の順番は重要です。例えば a*|b は "b" の中の空文字にマッチしますが、b|a* は同じ文字の "b" にマッチします。

{n} **直前の文字がちょうど n 回出現するものにマッチします。**n には正の整数が入ります。

例えば /a{2}/ は "candy" の 'a' にはマッチしませんが、"caaandy" の最初の 2 個の a にはマッチします。

{n,} 直前の式の少なくとも n回の出現にマッチします。N には正の整数が入ります。

例えば、/a{2,}/ は "aa", "aaaa", "aaaaa" にマッチしますが "a" にはマッチしません。

{n,m} **直前の文字が少なくとも n 回、多くても m 回出現するものにマッチします。**n および m には正の整数が入ります。m を省略した場合は ∞ とみなされます。

例えば /a{1,3}/ は "cndy" ではマッチせず、"candy," の 'a'、"caandy," の 最初の 2 個の a、"caaaaaaandy" の最初の 3 個の a にマッチします。"caaaaaaandy" では元の文字列に a が 4 個以上ありますが、マッチするのは "aaa" であることに注意してください。

[xyz] 文字集合を表します。このパターンでは、**角括弧で囲まれた文字のいずれか 1 個にマッチします。**対象の文字はエスケープシーケンスも含みます。

文字の集合内では、特殊文字 (例えばドット (.) やアスタリスク (*)) は特別な意味を持たないので、それらにエスケープは不要です。以下で例示するように、ハ

イフンを用いて文字の範囲を指定することも可能です。
例えば [abcd] は [a-d] と同じです。これは "brisket" の 'b' や "city" の 'c' にマッチします。/[a-z.]+/ および /[¥w.]+/ はどちらも、"test.i.ng" の全体にマッチします。

[^xyz] 文字集合の否定または補集合です。**角括弧で囲まれた文字ではない文字にマッチします。**ハイフンを用いて文字の範囲を指定することも可能です。文字集合パターンで動作するものすべてがこちらでも機能します。

例えば [^abc] は [^a-c] と同じです。これは "brisket" の 'r' や "chop" の 'h' といった一番最初の該当文字にマッチします。

[¥b] **後退文字（バックスペース、U+0008）にマッチします。**
後退文字自体にマッチさせるには角括弧を使う必要があります。（¥b と混同しないように。）

¥b **単語の区切りにマッチします。**単語の区切りは、単語構成文字の前または後ろに別の単語構成文字がない位置にマッチします。マッチした単語の区切りは、マッチした部分に含まれないことに注意してください。言い換えると、マッチした単語の区切りの長さは 0 です。([¥b] と混同してはいけません。)

例:
/¥bm/ は "moon" の 'm' にマッチします。
/oo¥b/ は "moon" の 'oo' にはマッチしません。これは、'oo' の後ろに単語構成文字である 'n' が続いているためです。
/oon¥b/ は "moon" の 'oon' にマッチします。これは、'oon' が文字列の終端であり単語構成文字が続かないためです。
/¥w¥b¥w/ はどこにもマッチしないでしょう。これは、単語構成文字の後に非単語構成文字および単語構成文字を続けることができないためです。
注記：JavaScript の正規表現エンジンでは「単語 (word)」を構成する文字として特定の文字集合を定義しています。この集合内にない文字は単語区切りと見なされます。この文字集合はかなり限定的なもので、ローマ字の大文字小文字のアルファベット、10 進数字とアンダースコアのみが含まれます。"é" や "ü" といった文字【訳注: そして日本語を構成する文字たちも】、残念ながら単語区切りとして扱われます。

文字	意味
¥B	**単語の区切り以外の文字にマッチします。**これは、前の文字と後ろの文字が同じ種類である位置にマッチします。すなわち両方とも単語であるか、両方とも単語でない場合です。文字列の先頭と終端は、単語ではないと見なされます。

例えば /¥B../ は "noonday" の 'oo' に、/y¥B./ は "possibly yesterday" の 'ye' にマッチします。

¥c*X* **文字列中の制御文字にマッチします。** *X* には A から Z のうち 1 文字が入ります。

例えば /¥cM/ は文字列中の control-M (U+000D) にマッチします。

¥d **数字にマッチします。**[0-9] に相当します。

例えば /¥d/ や /[0-9]/ は "B2 is the suite number" の '2' にマッチします。

¥D **数字以外の文字にマッチします。**[^0-9] に相当します。

例えば /¥D/ や /[^0-9]/ は "B2 is the suite number" の 'B' にマッチします。

¥f **改ページ (U+000C) にマッチします。**

¥n **改行文字 (U+000A) にマッチします。**

¥r **復帰文字 (U+000D) にマッチします。**

¥s **スペース、タブ、改ページ、改行を含む 1 個のホワイトスペース文字にマッチします。**[¥f¥n¥r¥t¥v¥u00a0¥u1680¥u180e¥u2000-¥u200a ¥u2028¥u2029¥u202f¥u205f¥u3000¥ufeff] に相当します。

例えば /¥s¥w*/ は "foo bar" の ' bar' にマッチします。

¥S **ホワイトスペース以外の 1 文字にマッチします。**[^¥f¥n¥r¥t¥v¥u00a0¥u1680¥u2000-¥u200a¥u2028¥u2029¥u202f¥u205f¥u3000¥ufeff] に相当します。

例えば /¥S¥w*/ は "foo bar" の 'foo' にマッチします。

¥t **タブ (U+0009) にマッチします。**

¥v **垂直タブ (U+000B) にマッチします。**

¥w **アンダースコアを含むどの英数字にもマッチします。**[A-Za-z0-9_] に相当します。

例えば /¥w/ は、"apple," の 'a' や "$5.28," の '5' や "3D" の '3' にマッチします。

文字 意味

¥W **前述以外の文字にマッチします。**[^A-Za-z0-9_] に相当します。

例えば /¥W/ や /[^A-Za-z0-9_]/ は、"50%" の '%' にマッチします。

¥*n* *n* に正の整数が入る場合、**正規表現内において *n* 番目の括弧の部分にマッチ**

した最新の部分文字列への後方参照となります（括弧の数は左からカウントします）。

例えば /apple(,)¥sorange¥1/ は "apple, orange, cherry, peach" の 'apple, orange,' にマッチします。

¥0 **NULL 文字 (U+0000) にマッチします。**この後ろに他の数字を続けてはいけません。¥0 の後に（0 から 7 までの）数字が続くと 8 進数のエスケープシーケンスとなるからです。

¥xhh **hh（2 桁の 16 進数）コードからなる文字列にマッチします。**

¥uhhhh **hhhh（4 桁の 16 進数）コードからなる文字列にマッチします。**

¥u{hhhh} **(u フラグがセットされた時のみ) Unicode 値 hhhh (16 進数) からなる文字列にマッチします。**

ユーザー入力を正規表現内の文字列リテラルとして扱うためには上記の記号は全てエスケープする必要がありますが、これは以下の様な置換で簡単に達成することができます：

```
function escapeRegExp(string) {
  return string.replace(/[.*+?^=!:${}()|[¥]¥/¥¥]/g, '¥¥$&'); // $&はマッチした部分文字列全体を意味します
}
```

正規表現の後の g はグローバルサーチを行うオプション/フラグで、全体の文字列を見てすべてのマッチを返します。下のフラグを使った高度な検索に詳しく説明されています。

括弧の使い方

正規表現パターンの一部を括弧で囲むことで、マッチした部分文字列を記憶しておくことができます。いったん記憶されれば、後からその部分文字列を呼び出すことができます。これに関しては括弧で囲まれた部分文字列のマッチの使用で説明しています。

例として /Chapter (¥d+)¥.¥d*/ というパターンを使い、エスケープ文字と特殊文字についても説明した上で、どのようにパターンの一部が記憶されるかを示します。これは 'Chapter ' という文字列に正確にマッチし、それに続く 1 文字以上の数字（¥d はいずれかの数字を、+ は 1 回以上の繰り返しを意味します）、それに続く小数点（それ自体は特殊文字であり、小数点の前の ¥ はパターンが '.' という文字そのものを探すようにす

ることを意味します)、それに続く 0 文字以上の数字 (¥d は数字を、* は 0 回以上の繰り返しを意味します) にマッチします。さらに、最初にマッチした数字の記憶に括弧が使われています。

このパターンは "Open Chapter 4.3, paragraph 6" という文字列で検索され、'4' が記憶されます。このパターンは "Chapter 3 and 4" では見つかりません。この文字列は '3' の後にピリオドがないからです。

マッチした部分を記憶させることなく部分文字列にマッチさせたい場合は、その括弧においてパターンの前に ?: をつけてください。例えば (?:¥d+) は 1 文字以上の数字にマッチしますが、マッチした文字列は記憶しません。

正規表現の使用

正規表現は、RegExp の test および exec メソッド、String の match、replace、search、および split メソッドとともに使用します。これらのメソッドの詳細は JavaScript リファレンス で説明しています。

正規表現を使用するメソッド

メソッド	説明
exec	文字列中で一致するものを検索する RegExp のメソッド。結果情報の配列を返します。
test	文字列中で一致するものがあるかをテストする RegExp のメソッド。true または false を返します。
match	文字列中で一致するものを検索する String のメソッド。結果情報の配列を返します。マッチしない場合は null を返します。
search	文字列中で一致するものがあるかをテストする String のメソッド。マッチした場所のインデックスを返します。検索に失敗した場合は -1 を返します。
replace	文字列中で一致するものを検索し、マッチした部分文字列を別の部分文字列に置換する String のオブジェクト。
split	正規表現または固定文字列を用いて文字列を分割し、部分文字列の配列に入れる String のメソッド。

あるパターンが文字列に存在するかを知りたいときは、test または search メソッドを使

用してください。詳細な情報が知りたいときは（実行時間が長くなりますが）exec または match メソッドを使用してください。exec や match を使用してマッチが成功した場合、これらのメソッドは配列を返し、また結びつけられた正規表現オブジェクトと定義済みオブジェクトである RegExp オブジェクトのプロパティを更新します。マッチが失敗すると、exec メソッドは null（false に変換します）を返します。

次の例では、exec メソッドを使用して文字列を検索します。

```
var myRe = /d(b+)d/g;
var myArray = myRe.exec("cdbbdbsbz");
```

正規表現のプロパティにアクセスする必要がない場合は、次のスクリプトが myArray を作成する別の方法になります:

```
var myArray = /d(b+)d/g.exec('cdbbdbsbz'); // similar to "cdbbdbsbz".match(/d(b+)d/g); however,
    // the latter outputs Array [ "dbbd" ], while
    // /d(b+)d/g.exec('cdbbdbsbz') outputs Array [ "dbbd", "bb" ].
    // See below for further info (CTRL+F "The behavior associated with the".)
```

ある文字列から正規表現を組み立てたい場合は、次のスクリプトのような方法があります:

```
var myRe = new RegExp("d(b+)d", "g");
var myArray = myRe.exec("cdbbdbsbz");
```

これらのスクリプトではマッチが成功すると、配列を返すとともに次表で示されるプロパティを更新します。

正規表現の実行結果

オブジェクト	プロパティまたはインデックス	説明	この例の場合
myArray		マッチした文字列と、すべての記憶された部分文字列です。	['dbbd', 'bb', index: 1, input: 'cdbbdbsbz']
	index	入力文字列でマッチした位置を示す、0 から始まるインデ	1

	index	一致文字列でマッチした位置を示すインデックスです。	1
	input	元の文字列です。	"cdbbdbsbz"
	[0]	最後にマッチした文字列です。	"dbbd"
myRe	lastIndex	次のマッチが始まるインデックスです。(このプロパティは、g オプションを用いる正規表現でのみセットされます。これについてはフラグを用いた高度な検索で説明します。)	5
	source	パターンのテキストです。正規表現の実行時ではなく作成時に更新されます。	"d(b+)d"

この例の 2 つ目の形式で示したように、オブジェクト初期化子を使用して、変数に代入せずに正規表現を使うことができます。しかしながら、この方法では生成される正規表現はすべて、別の正規表現として作成されます。このため、変数に代入しないこの形式を使用する場合は、その正規表現のプロパティに後からアクセスすることができません。例えば、次のようなスクリプトを使用するとしましょう :

```
var myRe = /d(b+)d/g;
var myArray = myRe.exec('cdbbdbsbz');
console.log('The value of lastIndex is ' + myRe.lastIndex);

// "The value of lastIndex is 5"
```

しかし、このスクリプトの場合は次のようになります:

```
var myArray = /d(b+)d/g.exec('cdbbdbsbz');
console.log('The value of lastIndex is ' + /d(b+)d/g.lastIndex);

// "The value of lastIndex is 0"
```

この 2 つの文中の /d(b+)d/g は別の正規表現オブジェクトであり、そのためにそれぞれの lastIndex プロパティの値も異なるのです。オブジェクト初期化子で作成する正規表現のプロパティにアクセスする必要がある場合は、まずそれを変数に代入するようにしてください。

括弧で囲まれた部分文字列のマッチの使用

正規表現パターンに括弧を含めることで、対応するサブマッチが記憶されます。例えば

正規表現パターンに括弧を含めることで、対応するサブマッチが記憶されます。例えば /a(b)c/ は 'abc' という文字列にマッチし、'b' が記憶されます。この括弧で囲まれた部分文字列のマッチは、Array の要素 [1], ..., [n] を使用して呼び出すことができます。

括弧で囲まれた部分文字列は何個でも使用できます。返された配列には、見つかったものすべてが存在します。以下の例では、括弧で囲まれた部分文字列の使用法を説明します。

次のスクリプトは replace() メソッドを使用して文字列中の単語を入れ替えます。テキスト置き換えのために、スクリプトで $1 と $2 を使用して、最初とその次の括弧で囲まれた部分文字列のマッチを示しています。

```
var re = /(¥w+)¥s(¥w+)/;
var str = "John Smith";
var newstr = str.replace(re, "$2, $1");
console.log(newstr);
```

これは "Smith, John" を出力します。

フラグを用いた高度な検索

正規表現には、グローバルな検索や大文字／小文字を区別しない検索を可能にする 4 種類のオプションフラグがあります。これらのフラグは、単独で使用することもまとめて使用することもできます。順番は問いません。フラグは正規表現の一部として含まれます。

正規表現フラグ

フラグ	説明
g	グローバルサーチ。
i	大文字・小文字を区別しない検索。
m	複数行検索。
u	unicode; パターンをユニコードのコードポイントの連続として扱う
y	対象文字列で最後に見つかったマッチの位置から検索を開始する先頭固定 (sticky) 検索を行います。sticky のページをご覧ください。

フラグを正規表現に含めるには、次のようにしてください：

var re = /pattern/flags;
または

var re = new RegExp("pattern", "flags");
フラグは正規表現を作る際になくてはならないものであることに注意してください。後から加えたり取り除いたりすることはできません。

例えば re = /¥w+¥s/g は、1 個以上の文字とそれに続くスペースを探す正規表現を作成します。また、正規表現は文字列全体を通してこの組み合わせを探します。

```
var re = /¥w+¥s/g;
var str = "fee fi fo fum";
var myArray = str.match(re);
console.log(myArray);
```

この例では ["fee ", "fi ", "fo "] が表示されます。また、この例では次の行：

var re = /¥w+¥s/g;
を次の行：

var re = new RegExp("¥¥w+¥¥s", "g");
に置き換えることができます。得られる結果は同じです。

m フラグは、複数行の入力文字列を複数の行として扱うときに用います。m フラグを使用した場合 ^ および $ は、文字列全体の先頭または末尾ではなく、入力文字列中のどの行の先頭または末尾にもマッチします。

.exec() メソッドが使われた時には '**g**' に関連したふるまいは異なります。("class" と "argument" の役割が反対になります: .match() の場合、文字クラス(やデータ型) がメソッドを持ち、正規表現は単なる引数で、.exec() の場合、正規表現がメソッドを持ち、文字は引数です。*str.match(re)* と *re.exec(str)* を比較します) '**g**' フラグが **.exec()** メソッドで使われる時は繰り返して進むためです。

var xArray; while(xArray = re.exec(str)) console.log(xArray);

// produces:
// ["fee ", index: 0, input: "fee fi fo fum"]
// ["fi ", index: 4, input: "fee fi fo fum"]
// ["fo ", index: 7, input: "fee fi fo fum"]
m フラグは複数行の入力文字列が複数行として扱われるように使われます。m が使われた場合、^ と $ は文字列全体の最初と最後の代わりに、入力文字列内のあらゆる行の開始と終了にマッチします。

例

以下では、正規表現の使用法をいくつか例示します。

入力文字列の順序変更

次の例では、正規表現の構造と string.split() および string.replace() の使用法を示します。空白、タブ、1 個のセミコロンで分割された名前（ファーストネームが先頭）からなる、大まかに整形された入力文字列をきれいにフォーマットします。最終的に名前の順序を逆転し（ラストネームが先頭）、リストをソートします。

```
// 名前の文字列は複数の空白やタブを含む。
// また、ファーストネームとラストネームの間に複数の空白があることもある
var names = "Harry Trump ;Fred Barney; Helen Rigby ; Bill Abel ; Chris Hand ";

var output = ["---------- Original String¥n", names + "¥n"];

// 2 種類の正規表現パターンと、格納用の配列を用意する。
// 文字列を分割して配列の要素に収める。

// パターン: ホワイトスペースの 0 回以上の繰り返しのあとにセミコロン、そのあとにホワイトスペースの 0 回以上の繰り返し
var pattern = /¥s*;¥s*/;

// 上記のパターンで文字列を断片に分割し
```

```
// 上記のパターンで文字列を断片に分割し、
// nameList という配列に断片を格納する。
var nameList = names.split(pattern);

// 新たなパターン: 1 個以上の文字、1 個以上のホワイトスペース、1 個以上の文字
// 括弧を用いてパターンの断片を記憶する。
// 記憶した断片は後から参照される。
pattern = /(¥w+)¥s+(¥w+)/;

// 処理された名前を格納する新しい配列。
var bySurnameList = [];

// 名前の配列を表示し、新しい配列にコンマ区切りで名前を
// ラストネーム、ファーストネームの順で格納する。
//
// replace メソッドはパターンにマッチしたものを除去し、
//「2 番目の記憶文字列のあとにコンマとスペース、
// さらにその後に続く 1 番目の記憶文字列」に置き換える。
//
// 変数 $1 および $2 は、パターンにマッチさせた際に
// 記憶しておいた部分文字列を参照する

output.push("---------- After Split by Regular Expression");

var i, len;
for (i = 0, len = nameList.length; i < len; i++){
  output.push(nameList[i]);
  bySurnameList[i] = nameList[i].replace(pattern, "$2, $1");
}

// 新しい配列を表示する。
output.push("---------- Names Reversed");
```

```
for (i = 0, len = bySurnameList.length; i < len; i++){
  output.push(bySurnameList[i]);
}

// ラストネームについてソートし、ソートした配列を表示する。
bySurnameList.sort();
output.push("---------- Sorted");
for (i = 0, len = bySurnameList.length; i < len; i++){
  output.push(bySurnameList[i]);
}

output.push("---------- End");

console.log(output.join("¥n"));
```

特殊文字を用いた入力の確認

次の例では、ユーザーは電話番号を入力します。ユーザーが "Check" ボタンを押すと、スクリプトがその番号の妥当性を確認します。その番号が正当である（正規表現で指定した文字の連続にマッチする）場合、スクリプトはユーザーへの感謝のメッセージを表示し、その番号を承認します。番号が正当でない場合は、その番号が妥当でないことをユーザーに通知します。

正規表現は、0 または 1 個の左括弧 ¥(?、その後に 3 個の数字 ¥d{3}、その後に 0 または 1 個の右括弧 ¥)?、その後、見つかった際に記憶される 1 個のダッシュ、スラッシュ、または小数点 ([-¥/¥.])、その後に 3 個の数字 ¥d{3}、その後、記憶された 1 個のダッシュ、スラッシュ、または小数点のマッチ ¥1、その後に 4 個の数字を探します。

ユーザーが Enter ボタンを押した際に発動する Change イベントにより RegExp.input の値が設定されます。

```
<!DOCTYPE html>
<html>
```

```
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
  <meta http-equiv="Content-Script-Type" content="text/javascript">
  <script type="text/javascript">
   var re = /¥(?¥d{3}¥)?([-¥/¥.])¥d{3}¥1¥d{4}/;
   function testInfo(phoneInput){
    var OK = re.exec(phoneInput.value);
    if (!OK)
     window.alert(RegExp.input + " isn't a phone number with area code!");
    else
     window.alert("Thanks, your phone number is " + OK[0]);
   }
  </script>
 </head>
 <body>
  <p>Enter your phone number (with area code) and then click "Check".
    <br>The expected format is like ###-###-####.</p>
  <form action="#">
   <input id="phone"><button onclick="testInfo(document.getElementById('phone'));">Check</button>
  </form>
 </body>
</html>
```