JavaScript 実習マニュアル

第零版 revision04

大黒学

JavaScript 実習マニュアル・第零版 revision04
著者——大黒学

2011 年 10 月　6 日（木）　第零版発行
2017 年 10 月 27 日（金）　第零版 revision04 発行

目次

第 1 章　JavaScript の基礎

1.1　プログラム

1.1.1　文書と言語

文字を並べることによって何かを記述したものは、「文書」(document) と呼ばれます。

文書を作るためには、記述したいことを意味として持つように、文字を並べていく必要があります。そして、そのためには、文字をどのように並べればどのような意味になるかということを定めた規則が必要になります。そのような規則は、「言語」(language) と呼ばれます。

人間に読んでもらうことを第一の目的とする文書を書く場合は、日本語や中国語やアラビア語のような、「自然言語」(natural language) と呼ばれる言語が使われます。自然言語というのは、人間の社会の中で自然発生的に形成された言語のことです。

言語には、自然言語のほかに、人間が意図的に設計することによって作られた、「人工言語」(artificial language) と呼ばれるものもあります。人間ではなくてコンピュータに読んでもらうことを第一の目的とする文書を書く場合は、通常、自然言語ではなく人工言語が使われます。

1.1.2　プログラムとプログラミング

コンピュータに何らかの動作を実行させるためには、実行してほしいことがどんな動作なのかということを記述した文書をコンピュータに与える必要があります。そのような文書は、「プログラム」(program) と呼ばれます。

プログラムを作成するためには、プログラムを書くという作業だけではなくて、プログラムの構造を設計したり、プログラムの動作をテストしたり、その不具合を修正したりするというような、さまざまな作業が必要になります。そのような、プログラムを作成するために必要となるさまざまな作業の全体は、「プログラミング」(programming) と呼ばれます。

1.1.3　プログラミング言語

プログラムというのも文書の一種ですから、それを書くためには何らかの言語が必要になります。プログラムを書く場合には、プログラムを書くことだけを目的として作られた人工言語を使うのが普通です。そのような、プログラムを書くための専用の言語は、「プログラミング言語」(programming language) と呼ばれます。

プログラミング言語には、たくさんのものがあります。例を挙げると、Fortran、COBOL、Lisp、Pascal、Basic、C、AWK、Smalltalk、ML、Prolog、Perl、PostScript、Tcl、Java、Ruby、…… というように、枚挙にいとまがないほどです。

1.1.4　この文章について

この文章（「JavaScript 実習マニュアル」）は、JavaScript というプログラミング言語を使って、プログラムというものの書き方について説明する、ということを目的とするチュートリアルです。

1.2　ブラウザー

1.2.1　JavaScript とブラウザー

プログラムを書いて、それをコンピュータに実行させるためには、コンピュータだけではなくて、何らかのソフトが必要になります。JavaScript で書かれたプログラムも、それを実行するためには、そのためのソフトが必要です。

JavaScript で書かれたプログラムを実行することのできるソフトのひとつは、私たちがインターネットのウェブページを閲覧するために使っているブラウザーです。

1.2.2　JavaScript コンソール

Google Chrome というブラウザーは、「JavaScript コンソール」(JavaScript Console) と呼ばれる機能を持っています。この機能を使うと、JavaScript のプログラムの断片を入力して、その結果を即座に確かめる、ということができます。

それでは、実際に JavaScript コンソールを使ってみましょう。Google Chrome の画面の右上にある、点が縦に 3 個並んだボタンをクリックして、表示されたメニューで、

その他のツール → デベロッパーツール

を選択してください。そうすると、「デベロッパーツール」(developer tools) と呼ばれるものの画面が開きます。この画面の中に並んでいるタブの中にある「Console」をクリックすると、JavaScript コンソールの画面が開きます。

JavaScript コンソールの画面には、大なり (>) という文字が表示されています。その右側に JavaScript のプログラムの断片を入力して、エンターキーを押すと、その断片が実行されて、その結果が出力されます。たとえば、

```
2*3
```

と入力して、エンターキーを押すと、

```
> 2*3
  6
```

というように、2 と 3 を掛け算した結果が出力されます（このように、JavaScript では、掛け算はアスタリスク (*) によってあらわされます)。

1.2.3 エラー

プログラムの中には、何らかの間違いが含まれていることがあります。そのような、プログラムの中に含まれている間違いは、「エラー」(error) と呼ばれます。

JavaScript コンソールは、入力されたプログラムの断片がエラーを含んでいる場合、そのエラーについてのメッセージを出力します。そのような、エラーについてのメッセージは、「エラーメッセージ」(error message) と呼ばれます。

たとえば、JavaScript コンソールに対して、

```
2*3
```

と入力するつもりだったけれども、間違えて、

```
2#3
```

と入力してしまったとしましょう。そうすると、JavaScript コンソールは、赤い文字で、

```
Uncaught SyntaxError: Invalid or unexpected token
```

というエラーメッセージを出力します。このエラーメッセージは、入力されたものの中に正しくない記述がある、ということを教えてくれています。

1.3 WSH

1.3.1 WSH とは何か

JavaScript で書かれたプログラムを実行することのできるソフトは、ブラウザーだけではありません。たとえば、Windows に標準で組み込まれている WSH(Windows Script Host) というソフトを使っても、JavaScript で書かれたプログラムを実行することが可能です。

WSH は、さまざまなプログラミング言語で書かれたプログラムを実行する機能を持っているソフトです（ただし、デフォルトの状態では、扱うことのできるプログラミング言語は JavaScript と VBScript だけです)。

1.3.2 コマンドプロンプト

Windows には、「コマンドプロンプト」(Command Prompt) という名前のソフトが標準で組み込まれています。これは、「コマンド」(command) と呼ばれるものをキーボードで入力することによってソフトを起動するためのソフトです。

WSH は、アイコンをクリックすることによって起動することも可能ですが、どちらかと言えば、コマンドを入力することによって起動するというのが普通です。

1.3.3 メモ帳

WSH にプログラムを実行させるためには、そのプログラムをファイルに保存する必要があります。

プログラムをファイルに保存したり、すでにファイルに保存されているプログラムを修正したりしたいときは、「テキストエディター」(text editor) と呼ばれるソフトを使います（テキスト

エディターは、単に「エディター」(editor) と呼ばれることもあります)。

Windows には、「メモ帳」(Notepad) という名前のテキストエディターが標準で組み込まれていますので、すでに何らかのテキストエディターを使っているという人以外は、それを使うといいでしょう。

コマンドプロンプトに対して、

```
notepad パス名
```

というコマンドを入力すると、メモ帳が起動します。そして、プログラムを入力したのち、メニューで、

ファイル → 上書き保存

を選択すると、入力したプログラムが、コマンドの中のパス名で指定されたファイルに保存されます。

1.3.4　プログラムの入力

それでは、テキストエディターを使って、次のプログラムを入力して、hello.js というファイルに保存してください。

プログラムの例　hello.js

```
WScript.echo("こんにちは、世界。");
```

このプログラムは、実行すると、「こんにちは、世界。」という言葉を画面に出力する、という動作をします。

JavaScript で書かれたプログラムをファイルに保存する場合、ファイル名の拡張子は、.js にします。

1.3.5　WSH によるプログラムの実行

コマンドプロンプトに対して、

```
cscript パス名
```

というコマンドを入力すると、WSH が起動して、パス名で指定されたファイルの中に保存されているプログラムが実行されます。

それでは、先ほど入力したプログラムを、WSH を使って実行してみましょう。

実行例

```
>cscript hello.js
Microsoft (R) Windows Script Host Version 5.7
Copyright (C) Microsoft Corporation 1996-2001. All rights reserved.

こんにちは、世界。
```

WSH は、このように、まず最初に自分についてのメッセージ（名前、バージョン、著作権者）を出力して、そののちプログラムを実行します。自分についてのメッセージを出力させたくないときは、

```
cscript //nologo パス名
```

というように、コマンドの中に //nologo と書きます。

実行例

```
>cscript //nologo hello.js
こんにちは、世界。
```

1.4　空白と改行と注釈

1.4.1　空白

空白という文字（スペースキーを押したときに入力される文字）は、プログラムの意味に影響を与えません（ただし、半角のみです。全角の空白は影響を与えます)。たとえば、

```
WScript.echo("こんにちは、世界。");
```

というプログラムは、

```
WScript . echo ( "こんにちは、世界。" ) ;
```

と書いたとしても同じ意味になりますし、

```
WScript   .   echo   (   "こんにちは、世界。"   )   ;
```

と書いたとしても同じ意味になります。
ただし、出力される言葉の中に空白を挿入した場合は、その空白も出力されます。たとえば、出力させる言葉として、

```
こ ん に ち は 、 世 界 。
```

というように、半角の空白を含んでいる言葉を書いたとすると、

```
こ ん に ち は 、 世 界 。
```

というように、それらの空白も出力されます。
空白は、プログラムの中のどこにでも挿入することができるというわけではありません。たとえば、名前の途中に空白を挿入する、ということはできません。ですから、echo という名前を、

```
e c h o
```

と書くことはできません。

1.4.2 改行

改行という文字（エンターキーを押したときに入力される文字）も、空白と同じように、プログラムの意味に影響を与えません。たとえば、

```
WScript.echo("こんにちは、世界。");
```

というプログラムは、

```
WScript.echo(
"こんにちは、世界。"
);
```

と書いたとしても同じ意味になります。
改行も、名前の途中などに挿入することができないという点は、空白と同じです。

1.4.3 注釈

プログラムを書いているとき、それを読む人間（プログラムを書いた人自身もその中に含まれます）に伝えたいことを、そのプログラムの一部分として書いておきたい、ということがしばしばあります。プログラムの中に書かれたそのような文字列は、「注釈」(comment) と呼ばれます。
注釈は、プログラムを処理するプログラムが、「ここからここまでは注釈だ」ということを認識することができるように、注釈を書くための文法にしたがって書く必要があります。
JavaScript には、注釈の書き方として二通りのものがあります。
ひとつは、注釈にしたい文字列を // と改行で囲むという書き方です。そうすると、// から改行までが注釈とみなされます。たとえば、JavaScript コンソールに、

```
2+3+4
```

と入力すると、2 と 3 と 4 の足し算が計算されて、9 が出力されますが、

```
2+3//+4
```

と入力すると、+4 という部分が注釈とみなされますので、結果は 5 になります。
もうひとつの注釈の書き方は、注釈にしたい部分を /* と */ で囲むというものです。たとえば、JavaScript コンソールに、

```
2/*+3*/+4
```

と入力すると、+3 という部分が注釈とみなされますので、結果は 6 になります。
/* と */ で囲む文字列は、途中に改行が含まれていてもかまいません。たとえば、

```
/* 私は、
改行を含んでいる
注釈です。 */
```

という記述は、その全体がひとつの注釈とみなされます。

プログラムを作成したり修正したりしているとき、その一部分を一時的に無効にしたい、ということがしばしばあります。そのような場合、無効にしたい部分を削除してしまうと、それを復活させるのに手間がかかりますので、削除するのではなくて、注釈にすることによって無効にするという手段が、しばしば使われます。記述の一部分を注釈にすることによって、それを無効にすることを、その部分を「コメントアウトする」(comment out) と言います。逆に、コメントアウトされている部分を復活させることを、その部分を「コメントインする」(comment in) と言います。

第 2 章　基本的な式

2.1　式の基礎

2.1.1　式と評価と値

JavaScript のプログラムの中には、「式」(expression) と呼ばれるものを書くことができます。前の章で登場した、`2*3` という記述の場合、その中に含まれている `2` と `3` のそれぞれは式で、さらに、その全体もひとつの式です。

式は、何らかの動作をあらわしていると考えることができます。たとえば、`2*3` という式は、2 と 3 を掛け算するという動作をあらわしています。

式が記述している動作を実行することを、その式を「評価する」(evaluate) と言います。たとえば、`2*3` という式の場合は、その式があらわしている、2 と 3 を掛け算するという動作を実行することが、その式を評価するということです。

式を評価すると、その結果として何らかの結果が得られます。式を評価することによって得られた結果は、その式の「値」(value) と呼ばれます。 たとえば、`2*3` という式の場合は、その式を評価することによって得られた、6 という数値が、その式の値になります。

JavaScript コンソールは、式が入力されると、その式を評価して、その値を出力します。

2.1.2　式の分類

式には、次のような種類があります。

- `this`
- リテラル (literal)
- 識別子 (identifier)
- 演算子を含む式 (expression containing operators)
- 関数式 (function expression)
- 配列初期化子 (array initializer)
- オブジェクト初期化子 (object initializer)

たとえば、`2*3` という式の場合、その中に含まれている `2` と `3` のそれぞれはリテラルに分類される式で、その全体は、「演算子を含む式」に分類される式です。

この章では、リテラル、識別子、演算子を含む式、という三つの種類の式について説明したいと思います。それら以外の種類の式については、この章よりもあとの章で説明する予定です。

2.1.3　演算子を含む式

「演算子を含む式」(expression containing operators) というのは、その名前のとおり、その中に「演算子」(operator) と呼ばれるものを含んでいる式のことです。

第 1.2 節で、プログラムの断片として、`2*3` というものを紹介しましたが、これは、演算子を含む式の実例です。

演算子を含む式は、1 個以上の式と、1 個以上の演算子とを組み合わせることによって作られます。たとえば、`2*3` という演算子を含む式は、`2` という式と、`*` という演算子と、`3` という式から構成されています。

演算子は、何らかの処理を意味しています。たとえば、`*` という演算子は、掛け算という処理を意味しています。演算子が意味している処理は、「演算」(operation) と呼ばれます。

2.2 リテラル

2.2.1 リテラルの基礎

特定のデータをあらわしている式は、「リテラル」(literal) と呼ばれます。たとえば、2 という式は、2 という特定の数値のデータをあらわしていますので、リテラルに分類されます。

リテラルを評価すると、それがあらわしているデータが、その値として得られます。たとえば、2 というリテラルを評価すると、それがあらわしている 2 という数値のデータが、その値として得られます。

2.2.2 リテラルの分類

リテラルは、次の 5 種類に分類することができます。

- ヌルリテラル (null literal)
- 真偽値リテラル (Boolean literal)
- 数値リテラル (numeric literal)
- 文字列リテラル (string literal)
- 正規表現リテラル (regular expression literal)

この節で説明するのは、これらの種類のうちで、数値リテラルと文字列リテラルだけです。それら以外の種類のリテラルについては、この章よりもあとの章で説明する予定です。

2.2.3 数値リテラルの基礎

数値 (number) をあらわしているリテラルは、数値リテラル (numerical literal) と呼ばれます。

数値リテラルは、次の 2 種類に分類することができます。

- 10 進数リテラル (decimal literal)
- 16 進整数リテラル (hexadecimal integer literal)

2.2.4 10 進数リテラル

10 進数で数値をあらわしているリテラルは、「10 進数リテラル」(decimal literal) と呼ばれます。

プラスの整数をあらわす 10 進数リテラルは、0 から 9 までの数字を並べることによって作ります。たとえば、`6081` は、6081 というプラスの整数をあらわしている 10 進数リテラルです。

小数をあらわす 10 進数リテラルは、小数点を示すドット (`.`) を書くことによって作ることができます。たとえば、`3.14` は、3.14 という小数をあらわしている 10 進数リテラルです。

ものすごく大きな数値やものすごく小さな数値を記述するのに便利な、「何々掛ける 10 の何乗」という形のリテラルを書くこともできます。$a \times 10^b$ をあらわすリテラルは、a`e`b と書きます。

リテラル	意味
`3e8`	3×10^8
`6.022e23`	6.022×10^{23}
`6.626e-34`	6.626×10^{-34}

2.2.5 16 進整数リテラル

16 進数で整数をあらわしているリテラルは、「16 進整数リテラル」(hexadecimal integer literal) と呼ばれます。

16 進整数リテラルは、16 進数の左側に `0x` を書いたものです。たとえば、`0xff` は、255 という整数をあらわしている 16 進整数リテラルです。

2.2.6 マイナスの数値

数値リテラルの左側にマイナス (`-`) を書いたものは、マイナスの数値をあらわす式になります。たとえば、`-6081` はマイナスの 6081 をあらわしている式で、`-0xff` はマイナスの 255 をあらわしている式です。

ちなみに、マイナスと数値リテラルから構成される式は、リテラルではなくて、演算子を含む式です。

2.2.7　文字列リテラル

文字を 1 列に並べることによって作られたデータは、「文字列」(string) と呼ばれます（「テキスト」(text) と呼ばれることもあります）。

文字列をあらわしているリテラルは、「文字列リテラル」(string literal) と呼ばれます。

文字列リテラルは、二重引用符 (") で文字列を囲むことによって作られます。たとえば、

```
"こんにちは、世界。"
```

というのは、「こんにちは、世界。」という文字列をあらわしている文字列リテラルです。

二重引用符の代わりに単一引用符 (’) で文字列を囲んでもかまいません。たとえば、先ほどの文字列リテラルは、

```
’こんにちは、世界。’
```

と書くこともできます。

2.2.8　エスケープシーケンス

改行のような特殊な文字は、文字列リテラルの中にそのまま書くことができません。そのような特殊な文字は、それをあらわす「エスケープシーケンス」(escape sequence) と呼ばれる、バックスラッシュ(\) で始まる文字列によって記述されます[1]。

エスケープシーケンスには、次のようなものがあります。

`\b`	バックスペース	`\f`	改ページ
`\t`	水平タブ	`\’`	単一引用符
`\v`	垂直タブ	`\"`	二重引用符
`\n`	改行	`\\`	バックスラッシュ
`\r`	キャリッジリターン		

それでは、`\n` を含んでいる文字列リテラルを JavaScript コンソールに入力してみましょう。そうすると、次のように、改行を含んでいる文字列が出力されます。

```
> "hoge\nhoge"
  "hoge
  hoge"
```

2.3　二項演算子

2.3.1　二項演算子の基礎

演算子を含む式がどのような構造になっているかというのは、それが含んでいる演算子によって異なっています。

「二項演算子」(binary operator) と呼ばれるグループに所属している演算子は、

```
式 二項演算子 式
```

という構造の式を作ります。たとえば、第 1.2 節で登場した `2*3` という式は、この構造を持っています。ですから、掛け算を意味する `*` というのは、二項演算子のひとつだということになります。

二項演算子を含む式を評価すると、原則的には、まず演算子の左右の式が評価されて、それらの式の値に対して、二項演算子があらわしている動作が実行されて、その結果が式全体の値になります。

2.3.2　算術演算子

数値に対する計算をあらわしている演算子は、「算術演算子」(arithmetic operator) と呼ばれます。掛け算（乗算）を意味する `*` という演算子は、算術演算子のひとつです。

二項演算子でかつ算術演算子であるような演算子としては、`*` 以外に、次のようなものがあります。

a `+` b　　a と b とを足し算（加算）する。

a `-` b　　a から b を引き算（減算）する。

[1] バックスラッシュは、日本語の環境では円マーク (¥) で表示されることがあります。

a / b　a を b で割り算（除算）する。

a % b　a を b で除算したあまりを求める。

JavaScript コンソールで試してみましょう。

```
> 20+7
  27
> 20-7
  13
> 20/7
  2.857142857142857
> 20%7
  6
```

2.3.3 文字列の連結

+ という演算子は、数値の加算だけではなくて、文字列の連結という意味も持っています。

+ の左右に書かれた式の値のどちらかが文字列だった場合、+ は、数値の加算という動作ではなくて、文字列の連結という動作をします。一方が文字列で他方が数値だった場合は、数値が文字列に変換されたのちにそれらが連結されます。JavaScript コンソールで試してみると、次のようになります。

```
> "吾輩は" + "猫である"
  "吾輩は猫である"
> "文字列" + 999
  "文字列 999"
> 999 + "文字列"
  "999 文字列"
```

2.3.4 優先順位

演算子を含む式は、その中に式を含んでいます。演算子を含む式の中の式は、どんな式でもかまいません。ですから、演算子を含む式の中に演算子を含む式を書くということも可能です。たとえば、

```
2+3*4
```

という式は、演算子を含む式の中に演算子を含む式を書いたものです。でも、この式は、果たして、

```
[2+3]*4
```

という構造なのでしょうか。それとも、

```
2+[3*4]
```

という構造なのでしょうか。

この問題は、個々の演算子が持っている「優先順位」(precedence) と呼ばれるものによって解決されます。

優先順位というのは、演算子が左右の式と結合する強さのことだと考えることができます。優先順位が高い演算子は、それが低い演算子よりも、より強く左右の式と結合します。

* と / と % は、+ と - よりも高い優先順位を持っています。ですから、

```
2+3*4
```

という式は、

```
2+[3*4]
```

という構造だと解釈されます。JavaScript コンソールで試してみると、次のようになります。

```
> 2+3*4
  14
```

2.3.5 結合規則

+ と - は同じ優先順位を持っていて、* と / と % も同じ優先順位を持っています。さて、それでは、

```
10-5-2
```

という式は、

```
10-5 -2
```

という構造なのでしょうか。それとも、

```
10- 5-2
```

という構造なのでしょうか。

この問題は、同一の優先順位を持つ演算子が共有している「結合規則」(associativity) と呼ばれる性質によって解決されます。

結合規則には、「左結合」(left-associativity) と「右結合」(right-associativity) という二つのものがあります。左結合というのは、左右の式と結合する強さが左にあるものほど強くなるという性質で、右結合というのは、それが右にあるものほど強くなるという性質です。

+、 -、 *、 /、 %の結合規則は、すべて左結合です。したがって、

```
10-5-2
```

という式は、

```
10-5 -2
```

という構造だと解釈されます。JavaScript コンソールで試してみると、次のようになります。

```
> 10-5-2
  3
```

2.3.6　グループ化演算子

ところで、2 と 3 とを足し算して、その結果と 4 とを掛け算したい、というときは、どのような式を書けばいいのでしょうか。先ほど説明したように、+ と * とでは、*のほうが優先順位が高くなっていますので、

```
2+3*4
```

では、期待した結果は得られません。

このような場合は、「グループ化演算子」(grouping operator) と呼ばれる、丸括弧 (()) を使います。

丸括弧で囲まれている部分は、演算子の優先順位や結合規則とは無関係に、ひとつの式だと解釈されます。JavaScript コンソールで試してみると、次のようになります。

```
> (2+3)*4
  20
> 10-(5-2)
  7
```

2.4　単項演算子

2.4.1　単項演算子の基礎

「単項演算子」(unary operator) と呼ばれるグループに所属している演算子は、

```
単項演算子 式
```

という構造の式、または、

```
式 単項演算子
```

という構造の式を作ります。式の前に書かれる単項演算子は「前置演算子」(prefix operator) と呼ばれ、式の後ろに書かれる単項演算子は「後置演算子」(postfix operator) と呼ばれます。

単項演算子は、二項演算子よりも高い優先順位を持っています。

2.4.2　符号の反転

-という前置演算子は、数値の符号（プラスかマイナスか）を反転させます。JavaScript コンソールで試してみると、次のようになります。

```
> -(3+5)
  -8
> -(3-5)
  2
```

2.4.3 型

JavaScript のプログラムによって扱われるデータは、「型」(type) と呼ばれるものを持っています。

型というのは、データが所属しているグループのことだと考えることができます。たとえば、数値のデータは「数値型」(number type) という型を持っていて、文字列のデータは「文字列型」(string type) という型を持っています。

typeof という前置演算子は、データの型を英語であらわしている文字列を求めます。JavaScript コンソールで試してみると、次のようになります。

```
> typeof 3
  "number"
> typeof "namako"
  "string"
```

2.5 識別子

2.5.1 変数

JavaScript のプログラムは、しばしば、データを入れることのできる箱を扱います。そのような箱は、「変数」(variable) と呼ばれます。

変数という箱にデータを入れることは、変数にデータを「設定する」(set) と言われたり、変数にデータを「代入する」(assign) と言われたりします。

変数は、それに与えられた名前によって識別されます。変数に与えることのできる名前は、「識別子」(identifier) と呼ばれます。

2.5.2 識別子の作り方

識別子は、次のような規則に従って作ることになっています。

- 識別子を作るために使うことのできる文字は、Unicode 文字、アンダースコア (_)、ドルマーク ($) です。Unicode 文字の中には日本語の文字も含まれていますので、日本語の文字で識別子を作ることも可能ですが、通常は英字と数字が使われます。
- 識別子の先頭の文字として、数字を使うことはできません。
- 予約語 (reserved word) と同じものは識別子としては使えません。「予約語」(reserved word) というのは、用途があらかじめ予約されている単語のことで、var、if、for などがあります。

識別子として使うことのできるものの例としては、次のようなものがあります。

```
a  A  namako  _  $  _a  $a  a8
```

他方、次のようなものを識別子として使うことはできません。

nam@ko	使うことのできない文字を含んでいる。
8a	先頭の文字が数字。
var	同じ予約語が存在する。

2.5.3 変数の宣言

変数を使うためには、その変数を作る必要があります。変数を作ることを、変数を「宣言する」(declare) と言います。

変数を宣言するという動作は、「変数文」(variable statement) と呼ばれるものによって記述されます。

変数文は、変数を宣言するという動作だけではなくて、宣言された変数を初期化する、という動作も記述することができます。「初期化する」(initialize) というのは、宣言された変数にデー

タを設定するということです。宣言された変数に設定されるデータは、「初期値」(initial value) と呼ばれます。

変数文は、基本的には、

```
var 識別子 = 式;
```

と書きます。そうすると、変数が宣言されて、その名前として識別子が与えられます。そして、イコールの右側に書かれた式の値が、その変数に対して初期値として設定されます。たとえば、

```
var a = 7;
```

という変数文を書くことによって、 a という名前で識別される変数が作られて、その変数に 7 という初期値が設定されます。

変数文の中のイコールの前と後ろには、上の例のように 1 個の空白を書くのが普通ですが、それらの空白は絶対に必要というわけではありませんので、

```
var a=7;
```

というように詰めて書いても問題はありません。

2.5.4　識別子の値

識別子は、式として評価することができます。識別子を評価すると、その識別子が与えられている変数に設定されているデータが、その値として得られます。

JavaScript コンソールを使って確かめてみましょう。

まず、 a という名前の変数を作って、それに初期値として 7 を設定してみましょう。

```
> var a = 7;
  undefined
```

変数文は式ではありませんので、それを実行しても値は得られません。ですから、JavaScript コンソールに変数文を入力した場合は、 undefined と出力されます。

次に、 a という識別子を JavaScript コンソールに入力してみましょう。

```
> a
  7
```

そうすると、このように、識別子が評価されて、7 という値が出力されます。

2.6　代入演算子

2.6.1　代入演算子の基礎

変数に設定されているデータは、別のデータを設定することによって、変更することが可能です。

変数にデータを設定したいときは、「代入演算子」(assignment operator) と呼ばれる演算子が使われます。

代入演算子は、すべて二項演算子で、ほとんどすべての演算子よりも低い優先順位を持っています。

2.6.2　単純代入演算子

= という演算子は、「単純代入演算子」(simple assignment operator) と呼ばれます。これは、代入演算子のうちで、もっとも単純な動作をするものです。

= を含む式は、

```
識別子 = 式
```

と書きます。この形の式を評価すると、識別子を名前として持つ変数に、式の値が設定されます。= を含む式の値は、変数に設定されたデータです。JavaScript コンソールで試してみると、次のようになります。

```
> var a = 7;
  undefined
> a = 33
  33
> a
```

```
33
```

代入演算子の前と後ろには、この例のように 1 個の空白を書くのが普通ですが、それらの空白は絶対に必要というわけではありません。

2.6.3 複合代入演算子

`+=`、`-=`、`*=`、`/=`、`%=` などの演算子は、「複合代入演算子」(compound assignment operator) と呼ばれます。

複合代入演算子を含む式は、`=` を含む式と同じように、

識別子 複合代入演算子 式

と書きます。この形の式を評価すると、識別子を名前として持つ変数の内容と式の値に対して演算が実行されて、その結果が変数に設定されます。複合代入演算子を含む式の値は、変数に設定されたデータです。

ところで、「変数の内容と式の値に対して演算が実行されて」と書きましたが、ここで実行される「演算」というのは、いったい何なのでしょうか。

複合代入演算子は、すべて、イコール (`=`) の左側に何らかの二項演算子を書いた形になっています。変数の内容と式の値に対して実行される演算は、イコールの左側に書かれた二項演算子があらわしている演算です。つまり、☆が二項演算子だとすると、

a ☆= b

という式は、

a = a ☆ b

という式と同じ意味になるということです。たとえば、

```
a += 8
```

という式は、

```
a = a + 8
```

という式と同じ意味になります。JavaScript コンソールで試してみると、次のようになります。

```
> var a = 7;
  undefined
> a += 8
  15
> a
  15
```

2.7 インクリメント演算子とデクリメント演算子

2.7.1 インクリメント演算子とデクリメント演算子の基礎

代入演算子を使うことによって、変数の内容を変化させることができるわけですが、変数の内容を変化させる演算子は、代入演算子だけではありません。「インクリメント演算子」(increment operator) と呼ばれる `++` という演算子と、「デクリメント演算子」(decrement operator) と呼ばれる `--` という演算子も、変数の内容を変化させます。

変数に設定されている数値を 1 だけ増加させることを、変数を「インクリメントする」(increment) と言います。そして、変数に設定されている数値を 1 だけ減少させることを、変数を「デクリメントする」(decrement) と言います。

インクリメント演算子は、変数をインクリメントする演算子で、デクリメント演算子は、変数をデクリメントする演算子です。

インクリメント演算子とデクリメント演算子は、どちらも単項演算子で、それぞれ、前置演算子と後置演算子の両方があります。つまり、次のような 4 個の演算子があるということです。

- 前置インクリメント演算子 (prefix increment operator)
- 後置インクリメント演算子 (postfix increment operator)
- 前置デクリメント演算子 (prefix decrement operator)
- 後置デクリメント演算子 (postfix decrement operator)

2.7.2　前置インクリメント演算子

前置インクリメント演算子を含む式は、

++識別子

と書きます。この形の式を評価すると、識別子を名前として持つ変数がインクリメントされます。前置インクリメント演算子を含む式の値は、インクリメントされた後の変数の内容です。JavaScript コンソールで試してみると、次のようになります。

```
> var a = 7;
  undefined
> ++a
  8
> a
  8
```

2.7.3　後置インクリメント演算子

後置インクリメント演算子を含む式は、

識別子++

と書きます。前置インクリメント演算子と後置インクリメント演算子とで違うのは、式の値です。後置インクリメント演算子を含む式の値は、インクリメントされる前の変数の内容です。JavaScript コンソールで試してみると、次のようになります。

```
> var a = 7;
  undefined
> a++
  7
> a
  8
```

2.7.4　前置デクリメント演算子

前置デクリメント演算子を含む式は、

--識別子

と書きます。この形の式を評価すると、識別子を名前として持つ変数がデクリメントされます。前置デクリメント演算子を含む式の値は、デクリメントされた後の変数の内容です。JavaScript コンソールで試してみると、次のようになります。

```
> var a = 7;
  undefined
> --a
  6
> a
  6
```

2.7.5　後置デクリメント演算子

後置デクリメント演算子を含む式は、

識別子--

と書きます。デクリメントされる前の変数の内容が式の値になります。JavaScript コンソールで試してみると、次のようになります。

```
> var a = 7;
  undefined
> a--
  7
> a
  6
```

2.8 関数呼び出し

2.8.1 関数とは何か

JavaScript では、動作をあらわしているデータのことを「関数」(function) と呼びます。たとえば、 `parseInt` という名前の関数は、文字列を整数に変換するという動作をあらわしているデータです。

基本的には、関数は、何らかのデータを受け取って、それらのデータを使って何らかの処理を実行して、その処理の結果を返す、という動作をあらわしています。

関数に対して、それがあらわしている動作を実行させることを、関数を「呼び出す」(call) と言います。

2.8.2 関数名

関数には、名前として識別子を与えることができます。関数に名前として与えられた識別子は、「関数名」(function name) と呼ばれます。

関数名を式として評価すると、その関数名を名前として持つ関数が、その値として得られます。

2.8.3 関数の型

関数は、「オブジェクト型」(object type) という型を持っています。ただし、オブジェクト型を持っているデータは関数だけではありません。

`typeof` は、オブジェクト型を `object` という文字列であらわすのですが、関数だけは特別扱いで、その型を `function` という文字列であらわします。たとえば、 `parseInt` という関数の型を調べてみると、次のようになります。

```
> typeof parseInt
  "function"
```

2.8.4 引数と戻り値

関数が受け取るデータは、「引数」(argument) と呼ばれます（「引数」は「ひきすう」と読みます)。関数は、引数を何個でも受け取ることができます。ただし、受け取った引数のうちの何個を使うかというのは、関数ごとに決まっています。引数をまったく使わない関数もあります。

関数が返すデータは、「戻り値」(return value) と呼ばれます。引数は何個でも受け取ることができますが、関数が返すことができる戻り値は、1 個だけです。

たとえば、 `parseInt` という関数は、引数として受け取った文字列を整数に変換して、その整数を戻り値として返します。

2.8.5 関数呼び出し演算子

関数を呼び出したいときは、「関数呼び出し」(function call) と呼ばれる式を書きます。

関数呼び出しは、演算子を含む式の一種です。関数呼び出しが含んでいるのは、「関数呼び出し演算子」(function call operator) と呼ばれる演算子です。

関数呼び出し演算子は、丸括弧 (`()`) です。グループ化演算子も丸括弧だったわけですが、式の中に丸括弧が出てきた場合、その式の構造によって、その丸括弧が関数呼び出し演算子なのかグループ化演算子なのかということが区別されます。

関数呼び出しは、

式 (式, ···)

と書きます。先頭の式は、値として関数が得られるものでないといけません。

関数呼び出しを評価すると、その先頭に書かれた式を評価することによって得られた関数が呼び出されて、丸括弧の中に書かれた式の値が、その関数に引数として渡されます。そして、呼び出された関数が返した戻り値が、関数呼び出しの値になります。

JavaScript コンソールを使って、 `parseInt` という関数を呼び出してみましょう。

```
> parseInt("801")
  801
```

このように、 `parseInt` は、引数として文字列を受け取って、それを整数に変換した結果を返します。ちなみに、文字列を小数に変換したいときは、 `parseFloat` という関数を使います。

```
> parseInt("3.14")
  3
> parseFloat("3.14")
  3.14
```

2.8.6　メソッド

関数は、「メソッド」(method) と呼ばれることもあります。関数とメソッドは、本質的には同じものですので、単に、呼び方が二通りあるだけです。

「関数」と呼ばれるか「メソッド」と呼ばれるかというのは、それを求めるための式の形に関係があります。単なる識別子によって求められる場合は「関数」と呼ばれることが多いのですが、ドット (.) という二項演算子を含む、

式 . 識別子

という形の式によって求められる場合は、「メソッド」と呼ばれる傾向にあります。

関数呼び出しは、それによって呼び出される関数が「メソッド」と呼ばれる場合、「メソッド呼び出し」(method call) と呼ぶこともできます。

「関数」ではなくて「メソッド」と呼ばれるものの例として、charAt というものがあります。これは、文字列の中から、引数として受け取った番号で指定された文字を取り出して、その文字だけから構成される文字列を返すメソッドです（先頭の文字を 0 番目と数えます)。このメソッドを呼び出すメソッド呼び出しは、

文字列を求める式.charAt(番号を求める式)

と書きます。たとえば、

```
"ABCDEFG".charAt(3)
```

というメソッド呼び出しを評価すると、charAt が呼び出されて、D という文字だけから構成される文字列が返ってきます。JavaScript コンソールで試してみると、次のようになります。

```
> "ABCDEFG".charAt(3)
  "D"
```

charAt のような、文字列を求める式と識別子によって求められるメソッドは、「文字列が持っているメソッド」と呼ばれたり、「文字列のメソッド」と呼ばれたりします。

2.8.7　小数から整数への変換

数値を扱うプログラムでは、しばしば、小数を整数にすることが必要になることがあります。

小数を整数に変換したいときに使われるのは、floor というメソッドです。このメソッドは、

```
Math.floor(式)
```

というメソッド呼び出しを書くことによって呼び出すことができます。

floor は、引数として数値を受け取って、その数値を超えない最大の整数を求めて、その整数を戻り値として返します。JavaScript コンソールで試してみると、次のようになります。

```
> Math.floor(5.3)
  5
> Math.floor(-5.3)
  -6
```

2.9　式文と出力と読み込み

2.9.1　文

プログラムを書くというのは、文法の上でのさまざまな部品を組み立てていくことだと考えることができます。式というのは、そのような部品のひとつです。

JavaScript には、式のほかに、「文」(statement) と呼ばれる文法の上での部品があります。どちらかと言えば、文は、式よりも大きな部品だと考えることができます。つまり、多くの文は、その中に何個かの式を含んでいるということです。ただし、文を含んでいる式というのも、存在しないわけではありません。

文にはさまざまな種類があります。第 2.5.3 項で紹介した変数文も、文の種類のひとつです。

2.9.2 式文

文の種類のひとつとして、「式文」(expression statement) と呼ばれるものがあります。

式文の書き方はとっても簡単で、

```
式;
```

と書くだけです。つまり、式を書いて、その右側にセミコロン (;) を書けば、それが式文になります。

式文は、その中に書かれている式を評価する、という動作をあらわしています。たとえば、

```
a = 8;
```

という式文があらわしている動作は、`a = 8` という式を評価するということです。したがって、この式文を実行すると、`a` という変数に 8 が設定されます。

第 1.3 節で、WSH に実行させるプログラムとして、

```
WScript.echo("こんにちは、世界。");
```

というものを紹介しましたが、このプログラムは、1 個の式文から構成されています。つまり、

```
WScript.echo("こんにちは、世界。")
```

という部分が 1 個の式で、それにセミコロンが加わって式文になっているわけです。ちなみに、この式は、`echo` というメソッドを呼び出すメソッド呼び出しです。

`echo` というのは、引数として受け取ったデータを出力して、さらに 1 個の改行を出力するメソッドです。

2.9.3 文の列

プログラムの中には、文の列を書くことができます。文の列というのは、その名前のとおり、文を並べてできる列のことです。文の列を書くときは、通常、それぞれの文のあいだに改行を挿入して、上から下へ並ぶようにします。

プログラムを実行すると、その中に書かれた文の列は、原則として、書かれている順番のとおりに実行されます。

それでは、実際に、2 個以上の文から構成される文の列を書いて、それがどのように実行されるかを試してみましょう。

プログラムの例 `sequence.js`

```
WScript.echo("菜の花や");
WScript.echo("月は東に");
WScript.echo("日は西に");
```

実行例

```
>cscript //nologo sequence.js
菜の花や
月は東に
日は西に
```

2.9.4 改行のない出力

`echo` は、データを出力したのち、かならず 1 個の改行を出力します。しかし、場合によっては、改行は出力してほしくないということもあります。そんなときは、

```
WScript.stdout.write
```

というメソッドを使えば、改行は出力されません。

プログラムの例 `write.js`

```
WScript.stdout.write("菜の花や");
WScript.stdout.write("月は東に");
WScript.echo("日は西に");
```

実行例

```
>cscript //nologo write.js
菜の花や月は東に日は西に
```

2.9.5　読み込み

キーボードからデータを読み込みたいときは、

```
WScript.stdin.readLine
```

というメソッドを使います。このメソッドは、キーボードから文字列を読み込んで、その文字列を戻り値として返します。引数は不要です。

プログラムの例　readline.js

```
WScript.stdout.write("文字列を入力してください。: ");
var s = WScript.stdin.readLine();
WScript.echo("入力された文字列は" + s + "です。");
```

実行例

```
>cscript //nologo readline.js
文字列を入力してください。: Congratulations!
入力された文字列は Congratulations! です。
```

数値をあらわしている文字列をキーボードから入力した場合も、readLine が返すのは、数値ではなくて、あくまで文字列です。したがって、それを数値として扱いたい場合は、parseInt または parseFloat を使って数値に変換する必要があります。

プログラムの例　sigma.js

```
WScript.stdout.write("整数を入力してください。: ");
var n = parseInt(WScript.stdin.readLine());
WScript.echo("1 から" + n + "までの整数の和は" +
  (n*(n+1)/2) + "です。");
```

実行例

```
>cscript //nologo sigma.js
整数を入力してください。: 100
1 から 100 までの整数の和は 5050 です。
```

第 3 章　選択

3.1　選択の基礎

3.1.1　選択とは何か

文の列を書くことによって、まずこの動作を実行して、次にこの動作を実行して、次にこの動作を実行して……というように、複数の動作を直線的に実行していくという動作を記述することができるわけですが、コンピュータに実行させたい動作は、必ずしも一直線に進んでいくものばかりとは限りません。しばしば、そのときの状況に応じて、いくつかの動作の候補の中からひとつの動作を選んで実行するということも、必要になります。

「いくつかの動作の候補の中からひとつの動作を選んで実行する」という動作は、「選択」(selection) と呼ばれます。

3.1.2　条件

コンピュータは、運を天に任せて動作を選択するわけではありません。選択は、何らかの判断にもとづいて実行されます。

成り立っているか、それとも成り立っていないか、という判断の対象は、「条件」(condition) と呼ばれます。

条件が成り立っていると判断されるとき、その条件は「真」(true) であると言われます。逆に、条件が成り立っていないと判断されるとき、その条件は「偽」(false) であると言われます。

3.1.3　真偽値

真を意味するデータと、偽を意味するデータは、総称して「真偽値」(Boolean) と呼ばれます。

JavaScript では、真偽値は、「真偽値型」(Boolean type) という型を持つデータです。

何の判断もしないで、特定の真偽値を求めたいときは、「真偽値リテラル」(Boolean literal) と呼ばれるリテラルを書きます。

真偽値リテラルには、次の二つがあります。

`true` 真を値とするリテラル

`false` 偽を値とするリテラル

3.2 関係演算子

3.2.1 関係演算子の基礎

二つのデータのあいだに何らかの関係があるという条件が成り立っているかどうかを調べる二項演算子は、「関係演算子」(relational operator) と呼ばれます。

関係演算子の優先順位は、加算や乗算などの演算子よりも低くて、代入演算子よりも高くなっています。

関係演算子を含む式を評価すると、演算子の左右にある式が評価されて、それらの式の値のあいだに関係が成り立っているかどうかという判断が実行されます。そして、関係が成り立っているならば真、成り立っていないならば偽が、式全体の値になります。

3.2.2 大小関係

次の関係演算子を使うことによって、データの大小関係について調べることができます。

a `>` b a は b よりも大きい。

a `<` b a は b よりも小さい。

a `>=` b a は b よりも大きいか、または a と b とは等しい。

a `<=` b a は b よりも小さいか、または a と b とは等しい。

JavaScript コンソールで試してみましょう。

```
> 8 > 5
  true
> 5 > 8
  false
> 5 > 5
  false
> 5 >= 5
  true
```

大小関係があるのは、数値と数値とのあいだだけではありません。文字列と文字列とのあいだにも大小関係があります。

辞書の見出しは、「辞書式順序」(lexicographical order) と呼ばれる順序で並べられています。文字列と文字列とのあいだの大小関係は、辞書式順序で文字列を並べたときに、後ろにあるものは前にあるものよりも大きい、という関係です。

JavaScript コンソールで試してみましょう。

```
> "stay" > "star"
  true
> "star" > "stay"
  false
```

3.2.3 等しいかどうか

二つのデータが等しいかどうかということは、次の関係演算子を使うことによって調べることができます。

a `==` b a と b とは等しい。

a `!=` b a と b とは等しくない。

JavaScript コンソールで試してみましょう。

```
> 5 == 5
  true
> 5 == 8
```

```
    false
> "star" == "star"
    true
> "star" == "stay"
    false
```

3.3　if 文

3.3.1　if 文の基礎

何らかの条件が成り立っているかどうかを調べて、その結果にもとづいて二つの動作のうちのどちらかを実行したい、というときは、「if 文」(if statement) と呼ばれる文を書きます。

if 文は、

```
if (条件式) {
  文の列 1
} else {
  文の列 2
}
```

と書きます。「条件式」のところには、条件をあらわす式、つまり、評価すると値として真偽値が得られる式を書きます。

if 文を実行すると、まず最初に、条件式が評価されます。そして、条件式の値が真だった場合は、文の列 $_1$ が実行されます（その場合、文の列 $_2$ は実行されません)。条件式の値が偽だった場合は、文の列 $_2$ が実行されます（その場合、文の列 $_1$ は実行されません)。

プログラムの例　evenodd.js

```
WScript.stdout.write("整数を入力してください。: ");
var n = parseInt(WScript.stdin.readLine());
WScript.stdout.write(n + "は");
if (n%2 == 0) {
  WScript.stdout.write("偶数");
} else {
  WScript.stdout.write("奇数");
}
WScript.echo("です。");
```

実行例

```
>cscript //nologo evenodd.js
整数を入力してください。: 6
6 は偶数です。
>cscript //nologo evenodd.js
整数を入力してください。: 7
7 は奇数です。
```

3.3.2　インデント

ひとつの if 文は、その全体がひとつの文ですが、その中にはいくつかの文が含まれています。つまり、 if 文は、文の中に文が含まれているという構造を持っているということです。

文の中に文を書く場合は、上のプログラムでもそうしているように、中の文を右にずらして書きます。つまり、先頭に何個かの空白を書くわけです。その理由は、そうすることによって、プログラムを読む人間にとって、文の構造が分かりやすくなるからです。

プログラムの一部分を右にずらして書くことを、その部分を「インデントする」(indent) と言います。

インデントは、してもしなくても、プログラムの実行には何の影響も与えません。しかし、インデントがないと、プログラムの読みやすさが大幅に低下します。ですから、プログラムを書くときは、インデントを忘れないようにしましょう。

3.3.3　else 以降を省略した if 文

二つの動作のうちのどちらかを選択するのではなくて、ひとつの動作を実行するかしないかを選択したい、ということもしばしばあります。そのような場合は、 else 以降を省略した if 文を

書きます。つまり、

```
if (条件式) {
  文の列
}
```

という形の if 文を書くわけです。この形の if 文を実行すると、条件式の値が真の場合は文の列が実行されますが、条件式の値が偽の場合は何も実行されません。

プログラムの例 mtohm.js

```
WScript.stdout.write("時間の長さを分で入力してください。: ");
var m = parseInt(WScript.stdin.readLine());
WScript.stdout.write(m + "分は" + Math.floor(m/60) + "時間");
if (m%60 != 0) {
  WScript.stdout.write(m%60 + "分");
}
WScript.echo("です。");
```

実行例

```
>cscript //nologo mtohm.js
時間の長さを分で入力してください。: 160
160 分は 2 時間 40 分です。
>cscript //nologo mtohm.js
時間の長さを分で入力してください。: 180
180 分は 3 時間です。
```

これは、何分という形式であらわされた時間の長さを、何時間何分という形式に変換するプログラムです。ただし、変換すると何分の部分が 0 分になる場合は、何分の部分を省略します。何分の部分を出力するという動作は、実行するかしないかを選択することになりますので、 else 以降を省略した if 文を使って書かれています。

3.3.4 多肢選択

選択の対象となる動作が 3 個以上あるような選択は、「多肢選択」(multibranch selection) と呼ばれます。多肢選択には、次の二つのタイプがあります。

- ひとつの式の値が何なのかということによって動作を選択するタイプ。
- いくつかの条件のうちのどれが真になるかということによって動作を選択するタイプ。

ひとつの式の値による多肢選択は、「switch 文」(switch statement) と呼ばれる文を使うことによって記述することができます（ switch 文については次の節で説明します)。

いくつかの条件による多肢選択は、 if 文を使うことによって記述することができます。条件と同じ個数の if 文を、 else の後ろにさらに if 文を書くという形で連結していけばいいのです。たとえば、3 個の条件による多肢選択は、

```
if (条件式 1) {
  文の列 1
} else if (条件式 2) {
  文の列 2
} else if (条件式 3) {
  文の列 3
} else {
  文の列 4
}
```

という形の if 文を書くことによって記述することができます。この形の if 文を実行すると、値が真になる条件式が見つかるまで、条件式 1、条件式 2、条件式 3 という順番で条件式が評価されていきます。値が真になる条件式が見つかった場合は、その条件式と同じ番号の文の列が実行されます（その場合、それ以降の条件式は評価されません)。すべての条件式の値が偽だった場合は、文の列 4 が実行されます。

プログラムの例 sign.js

```
WScript.stdout.write("数値を入力してください。: ");
```

```
var a = parseFloat(WScript.stdin.readLine());
WScript.stdout.write(a + "は");
if (a > 0) {
  WScript.stdout.write("プラス");
} else if (a < 0) {
  WScript.stdout.write("マイナス");
} else {
  WScript.stdout.write("ゼロ");
}
WScript.echo("です。");
```

実行例

```
>cscript //nologo sign.js
数値を入力してください。: 5
5 はプラスです。
>cscript //nologo sign.js
数値を入力してください。: -5
-5 はマイナスです。
>cscript //nologo sign.js
数値を入力してください。: 0
0 はゼロです。
```

3.4　switch 文

3.4.1　switch 文の基礎

前の節で説明したように、ひとつの式の値が何なのかということによって動作を選択するタイプの多肢選択は、「switch 文」(switch statement) と呼ばれる文を使うことによって記述することができます。

switch 文は、

```
switch (式) {
  選択肢の列
}
```

と書きます。switch 文を実行すると、switch の右に書かれた式が評価されて、その値が何なのかということによって、選択肢の中からひとつが選択されて、それが実行されます。

switch 文の中には、式の値によって選択されるいくつかの選択肢を書く必要があります。選択肢として書くことができるものとしては、次の 2 種類のものがあります。

- case 節 (case clause)
- default 節 (default clause)

3.4.2　case 節

case 節は、

```
case 式:
  文の列
  break;
```

と書きます。case 節の先頭に書かれる、

```
case 式:
```

という部分は、「case ラベル」(case label) と呼ばれます。

case 節で書かれた選択肢は、switch の右に書かれた式の値と case ラベルの中に書かれた式の値とが一致した場合に選択されて、その中の文の列が実行されます。

プログラムの例　menu.js

```
WScript.echo("1 焼きそば");
WScript.echo("2 お好み焼き");
WScript.echo("3 たこ焼き");
WScript.stdout.write(
```

```
  "注文したいものの番号を入力してください。: ");
var n = parseInt(WScript.stdin.readLine());
switch (n) {
  case 1:
    WScript.echo("あなたは焼きそばを注文しました。");
    break;
  case 2:
    WScript.echo("あなたはお好み焼きを注文しました。");
    break;
  case 3:
    WScript.echo("あなたはたこ焼きを注文しました。");
    break;
}
```

実行例

```
>cscript //nologo menu.js
1 焼きそば
2 お好み焼き
3 たこ焼き
注文したいものの番号を入力してください。: 2
あなたはお好み焼きを注文しました。
```

3.4.3 default 節

default 節は、

```
default:
  文の列
```

と書きます。これを switch 文の最後の選択肢として書いておくと、その選択肢は、case 節で書かれたどの選択肢も選択されなかった場合（つまり、case ラベルの中に書かれたどの式の値も、switch の右に書かれた式の値と一致しなかった場合）に選択されて、その中の文の列が実行されます。

プログラムの例 default.js

```
WScript.echo("1 コーヒー");
WScript.echo("2 紅茶");
WScript.echo("3 ジュース");
WScript.stdout.write(
  "注文したいものの番号を入力してください。: ");
var n = parseInt(WScript.stdin.readLine());
switch (n) {
  case 1:
    WScript.echo("あなたはコーヒーを注文しました。");
    break;
  case 2:
    WScript.echo("あなたは紅茶を注文しました。");
    break;
  case 3:
    WScript.echo("あなたはジュースを注文しました。");
    break;
  default:
    WScript.echo("その番号は無効です。");
}
```

実行例

```
>cscript //nologo default.js
1 コーヒー
2 紅茶
3 ジュース
注文したいものの番号を入力してください。: 3
あなたはジュースを注文しました。
>cscript //nologo default.js
1 コーヒー
2 紅茶
```

```
3 ジュース
注文したいものの番号を入力してください。: 4
その番号は無効です。
```

3.4.4　複数の case ラベル

実は、1 個の case 節に書くことのできる case ラベルは、1 個だけと決まっているわけではなくて、何個でも好きなだけ書くことができます。1 個の case 節の中に複数の case ラベルを書いた場合、その case 節で書かれた選択肢は、 case ラベルの中に書かれた式の値のうちのどれかひとつが、 switch の右に書かれた式の値と一致した場合に選択されます。

プログラムの例　month.js

```
WScript.stdout.write("月を数字で入力してください。: ");
var month = parseInt(WScript.stdin.readLine());
WScript.stdout.write(month + "月");
switch (month) {
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12:
    WScript.echo("は大の月です。");
    break;
  case 2:
  case 4:
  case 6:
  case 9:
  case 11:
    WScript.echo("は小の月です。");
    break;
  default:
    WScript.echo("という月はありません。");
}
```

実行例

```
>cscript //nologo month.js
月を数字で入力してください。: 8
8 月は大の月です。
>cscript //nologo month.js
月を数字で入力してください。: 9
9 月は小の月です。
```

3.5　論理演算子

3.5.1　論理演算子の基礎

処理の対象が真偽値で、処理の結果も真偽値であるような動作をあらわしている演算子は、「論理演算子」(logical operator) と呼ばれます。

JavaScript には、次の三つの論理演算子があります。

a && b　　a かつ b である。

a || b　　a または b である。

! a　　a ではない。

&& と || は、関係演算子よりも低くて代入演算子よりも高い優先順位を持っています。そして、&& は、 || よりも高い優先順位を持っています。

3.5.2　論理積演算子

&& は、「論理積演算子」(logical AND operator) と呼ばれます。これは、二つの条件が両方とも成り立っているかどうかを判断したいとき、つまり、A かつ B という条件が成り立っている

かどうかを判断したいときに使われる論理演算子で、次のような動作をします。

```
true   &&  true     ⟶  true
true   &&  false    ⟶  false
false  &&  true     ⟶  false
false  &&  false    ⟶  false
```

3.5.3 論理和演算子

|| は、「論理和演算子」(logical OR operator) と呼ばれます。これは、二つの条件のうちの少なくとも一つが成り立っているかどうかを判断したいとき、つまり、A または B という条件が成り立っているかどうかを判断したいときに使われる論理演算子で、次のような動作をします。

```
true   ||  true     ⟶  true
true   ||  false    ⟶  true
false  ||  true     ⟶  true
false  ||  false    ⟶  false
```

プログラムの例 leapyear.js

```
WScript.stdout.write("年を西暦で入力してください。: ");
var y = parseInt(WScript.stdin.readLine());
WScript.stdout.write(y + "年はうるう年");
if (y%4 == 0 && y%100 != 0 || y%400 == 0) {
  WScript.echo("です。");
} else {
  WScript.echo("ではありません。");
}
```

実行例

```
>cscript //nologo leapyear.js
年を西暦で入力してください。: 2080
2080 年はうるう年です。
>cscript //nologo leapyear.js
年を西暦で入力してください。: 2100
2100 年はうるう年ではありません。
>cscript //nologo leapyear.js
年を西暦で入力してください。: 2400
2400 年はうるう年です。
```

3.5.4 論理否定演算子

! は、「論理否定演算子」(logical negation operator) と呼ばれます。これは、真偽値を反転させたいとき、つまり、A ではないという条件が成り立っているかどうかを判断したいときに使われる論理演算子で、次のような動作をします。

```
! true     ⟶  false
! false    ⟶  true
```

3.6 条件演算子

3.6.1 条件演算子の基礎

これまで説明してきたように、JavaScript には、if 文と switch 文という、選択を記述するための文があります。しかし、選択を記述する方法は、if 文または switch 文を書くというものだけではありません。1 個の式を書くことによって選択を記述する、ということも可能です。

選択をあらわす式は、「条件演算子」(conditional operator) と呼ばれる、?: という演算子を使うことによって記述されます。

3.6.2 条件演算子を含む式

条件演算子を含む式は、

条件式 ? 式$_1$: 式$_2$

と書きます。「条件式」のところには、評価すると値として真偽値が得られる式を書きます。

条件演算子を含む式を評価すると、まず最初に、条件式が評価されます。そして、条件式の値が真だった場合は、式$_1$が評価されます（その場合、式$_2$は評価されません）。条件式の値が偽だった場合は、式$_2$が評価されます（その場合、式$_1$は評価されません）。そして、評価された式$_1$または式$_2$の値が、条件演算子を含む式の全体の値になります。

JavaScript コンソールで試してみましょう。

```
> true ? 1 : 0
  1
> false ? 1 : 0
  0
```

プログラムの例　hundred.js

```
WScript.stdout.write("数値を入力してください。: ");
var a = parseFloat(WScript.stdin.readLine());
WScript.echo(a + "は 100" +
  ((a >= 100) ? "以上" : "未満") + "です。");
```

実行例

```
>cscript //nologo hundred.js
数値を入力してください。: 200
200 は 100 以上です。
>cscript //nologo hundred.js
数値を入力してください。: 50
50 は 100 未満です。
```

第 4 章　繰り返し

4.1　繰り返しの基礎

4.1.1　繰り返しとは何か

コンピュータに実行させたい動作は、必ずしも、一連の動作をそれぞれ一回ずつ実行していけばそれで達成される、というものばかりとは限りません。しばしば、ほとんど同じ動作を何回も何十回も何百回も実行しなければ意図していることを達成できない、ということがあります。

「同じ動作を何回も実行する」という動作は、「繰り返し」(iteration) と呼ばれます。

この章では、繰り返しというのはどのように記述すればいいのか、ということについて説明します。

4.1.2　繰り返しを記述するための文

繰り返しは、繰り返したい回数と同じ個数の文を書くことによって記述することも可能ですが、そのような書き方だと、繰り返しの回数に比例してプログラムが長くなってしまいます。ですから、多くのプログラミング言語は、繰り返しを簡潔に記述することができるようにする機能を持っています。

JavaScript では、次の 4 種類の文のうちのどれかを使うことによって、繰り返しを簡潔に記述することができます。

- `while` 文 (`while` statement)
- `do-while` 文 (`do-while` statement)
- `for` 文 (`for` statement)
- `for-in` 文 (`for-in` statement)

これらの文のうちで、この章で紹介するのは、`while` 文、`do-while` 文、`for` 文の三つです。`for-in` 文については、第 6.3 節で説明することにしたいと思います。

4.2 while 文

4.2.1 while 文の基礎

while 文は、

```
while（条件式）{
  文の列
}
```

と書きます。「条件式」のところには、評価すると値として真偽値が得られる式を書きます。

while 文は、次のような動作をあらわしています。

(1) 条件式を評価する。その値が偽だった場合、while 文の動作は終了する。

(2) 条件式の値が真だった場合は、文の列を実行する。

(3) (1) に戻って、ふたたび同じ動作を実行する。

4.2.2 無限ループ

繰り返しというのは、何らかの条件が成り立っているあいだだけ実行して、その条件が成り立たなくなったときに終了する、というのが普通です。しかし、永遠に終わらない繰り返しというものを記述することも可能です。永遠に終わらない繰り返しは、「無限ループ」(infinite loop) と呼ばれます。

true という真偽値リテラルは、値が常に真ですから、while 文の条件式としてそれを書くと、その while 文は無限ループになります。

プログラムの例 infinite.js

```
while (true) {
  WScript.echo("Ctrl-C で終了します。");
}
```

このプログラムを実行すると、「Ctrl-C で終了します。」という文字列の出力が繰り返されます。実行したまま放置すると、繰り返しは無限に続きます。このプログラムを終了させたいときは、Ctrl-C、つまりコントロールキーを押しながら C のキーを押す、という操作をします。

4.2.3 自然数の出力

0、1、2、3、4、・・・という数は、「自然数」(natural number) と呼ばれます（0 を除外する場合もあります)。

それでは、すべての自然数を出力するプログラムを、while 文を使って書いてみましょう。

すべての自然数を出力するためには、出力する自然数を入れる変数が必要です。そこで、i という変数を宣言することにします。

あらかじめ初期値として i に 0 を設定しておいて、

(1) i の内容を出力する。

(2) i をインクリメントする。

という動作を繰り返すと、すべての自然数が出力されることになります（ただし、本当にすべての自然数を出力するためには、無限の時間が必要です)。

プログラムの例 natural.js

```
var i = 0;
while (true) {
  WScript.stdout.write(i + " ");
  i++;
}
```

このプログラムも無限ループですので、Ctrl-C で終了させない限り、実行は永遠に終わりません。

4.2.4 有限の回数の繰り返し

while 文の条件式として true という真偽値リテラルを書くと、その繰り返しは無限ループになります。それに対して、while 文の条件式として、繰り返しを開始する時点では値が真だけれ

ども、繰り返しを続行していくと、いつかは値が偽になる、という式を書くと、その繰り返しは有限の回数で終了することになります。

たとえば、すべての自然数を出力する先ほどのプログラムを改造して、while 文の条件式を true から、

```
i <= 100
```

という式に書き換えたとすると、i という変数の内容が、0、1、2、3、4、・・・、100 と増加していくあいだは条件式の値が真ですが、101 になったところで条件式の値が偽になりますので、自然数は 100 までしか出力されないということになります。

プログラムの例 finite.js

```
WScript.stdout.write("自然数を入力してください。: ");
var n = parseInt(WScript.stdin.readLine());
var i = 0;
while (i <= n) {
  WScript.stdout.write(i + " ");
  i++;
}
WScript.echo();
```

実行例

```
>cscript //nologo finite.js
自然数を入力してください。: 20
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

ちなみに、このプログラムの末尾に書かれている、

```
WScript.echo();
```

というのは、改行を出力するという動作をあらわしている文です。

4.3 do-while 文

4.3.1 do-while 文の基礎

do-while 文は、

```
do {
  文の列
} while (条件式);
```

と書きます。「条件式」のところには、評価すると値として真偽値が得られる式を書きます。

do-while 文は、次のような動作をあらわしています。

(1) 文の列を実行する。
(2) 条件式を評価する。その値が偽だった場合、do-while 文の動作は終了する。
(3) 条件式の値が真だった場合は、(1) に戻って、ふたたび同じ動作を実行する。

4.3.2 while 文と do-while 文の相違点

while 文と do-while 文は、どちらも、条件式の値が真であるあいだだけ文の列の実行を繰り返す、という動作をあらわしています。では、この 2 種類の文は、どこに相違点があるのでしょうか。

while 文と do-while 文の相違点は、「最初に何をするか」というところにあります。while 文の場合、最初の動作は「条件式の評価」です。それに対して、do-while 文の場合、最初の動作は「文の列の実行」です。

その結果として、while 文と do-while 文とでは、条件式の値が最初から偽だった場合に異なる動作をすることになります。たとえば、

```
var i = 200;
while (i <= 100) {
  WScript.stdout.write(i + " ");
  i++;
```

```
}
```

というプログラムを実行したとしましょう。すると、

```
i <= 100
```

という条件式の値は、最初から偽ですので、このプログラムは、何も出力しないで終了します。それに対して、

```
var i = 200;
do {
  WScript.stdout.write(i + " ");
  i++;
} while (i <= 100);
```

というプログラムは、条件式の値が最初から偽であるにもかかわらず、条件式の評価よりも先に文の列が実行されますので、200 を出力してから終了します。

4.3.3 どんなときに do-while 文を使えばいいか

それでは、do-while 文というのは、いったいどんなときに使えばいいのでしょうか。

while 文よりも do-while 文を使うほうがいいときというのは、繰り返しの対象となる文の列を少なくとも 1 回は実行する必要がある場合です。たとえば、利用者からのデータの読み込みを、正しいデータが入力されるまで繰り返したい、というような繰り返しは、while 文よりも do-while 文を使うほうが、すっきりと記述することができます。

プログラムの例 dowhile.js

```
var answer;
do {
  WScript.stdout.write("yes と入力してください。: ");
  answer = WScript.stdin.readLine();
} while (answer != "yes");
WScript.echo("ありがとうございました。");
```

実行例

```
cscript //nologo dowhile.js
yes と入力してください。: no
yes と入力してください。: no
yes と入力してください。: yes
ありがとうございました。
```

ちなみに、このプログラムの 1 行目に書かれている、

```
var answer;
```

という文は、answer という変数を宣言する変数文です。変数文は、このように、識別子の右側のイコールと式を省略してもかまいません。イコールと式が省略された場合、変数は、「未定義値」(undefined) と呼ばれるデータによって初期化されます。

4.4 for 文

4.4.1 for 文の基礎

繰り返しにはさまざまなタイプのものがあるのですが、きわめて頻繁に使われるのは、次のような三つの動作によって制御されるタイプの繰り返しです。

- 変数を宣言して初期化する。
- 変数の内容を調べて、繰り返しを続行するかどうかを判断する。
- 変数の内容を変化させる。

たとえば、次の繰り返しも、この三つの動作によって繰り返しが制御されています。

```
var i = 0;
while (i <= 100) {
  WScript.stdout.write(i + " ");
  i++;
```

```
  }
```

すなわち、var i = 0; という変数文で、変数の宣言と初期化を実行して、i <= 100 という式で、変数の内容を調べて、繰り返しを続行するかどうかを判断して、i++ という式で、変数の内容を変化させているわけです。

for 文は、このようなタイプの繰り返しを記述するために存在する文です。

このようなタイプの繰り返しは、while 文を使って記述することも可能ですが、for 文を使うほうがプログラムが読みやすくなります。なぜなら、このようなタイプの繰り返しを while 文を使って記述した場合、それを制御する三つの動作が分散して書かれることになるのに対して、for 文を使って記述した場合は、それらの動作が一箇所にまとまって書かれることになるからです。

4.4.2　for 文の書き方

for 文は、

```
for (var 識別子 = 式1; 式2; 式3) {
  文の列
}
```

と書きます。

for 文は、次のような動作をあらわしています。

(1) 識別子を名前とする変数を作って、式 1 の値を初期値としてそれに設定する。
(2) 式 2 を評価する。その値が偽だった場合、for 文の動作は終了する。
(3) 式 2 の値が真だった場合は、文の列を実行する。
(4) 式 3 を評価する。
(5) (2) に戻って、ふたたび同じ動作を実行する。

変数文と while 文を組み合わせることによって、for 文があらわしている動作を記述すると、次のようになります。

```
var 識別子 = 式1;
while (式2) {
  文の列
  式3;
}
```

次のプログラムは、与えられた整数のすべての約数を出力するという繰り返しを、for 文を使って記述しています。

プログラムの例　divisor.js

```
WScript.stdout.write("プラスの整数を入力してください。: ");
var n = parseInt(WScript.stdin.readLine());
for (var i = 1; i <= n; i++) {
  if (n%i == 0) {
    WScript.stdout.write(i + " ");
  }
}
WScript.echo();
```

実行例

```
>cscript //nologo divisor.js
プラスの整数を入力してください。: 1155
1 3 5 7 11 15 21 33 35 55 77 105 165 231 385 1155
```

第 5 章　関数

5.1　関数の基礎

5.1.1　関数についての復習

この章では、関数について説明します。

関数とはどのようなものかということについては、すでに第 2.8 節で説明しましたので、まずは、そのときに説明したことを復習しておくことにしましょう。

- 「関数」(function) というのは、動作をあらわしているデータのことである。
- 関数は、「オブジェクト型」(object type) という型を持っている。
- 関数には、名前として識別子を与えることができる。関数に名前として与えられた識別子は、「関数名」(function name) と呼ばれる。
- 基本的には、関数は、何らかのデータを受け取って、それらのデータを使って何らかの処理を実行して、その処理の結果を返す、という動作をあらわしている。
- 関数が受け取るデータは「引数」(argument) と呼ばれ、関数が返すデータは「戻り値」(return value) と呼ばれる。
- 関数は引数を何個でも受け取ることができるが、関数が返すことのできる戻り値は 1 個だけである。
- 関数に対して、それがあらわしている動作を実行させることを、関数を「呼び出す」(call) と言う。
- 関数を呼び出す式は、「関数呼び出し」(function call) と呼ばれる。
- 関数呼び出しは、

  式 (式, ···)

  と書く。先頭の式は、値として関数が得られるものでなければならない。
- 関数呼び出しを評価すると、その先頭に書かれた式を評価することによって得られた関数が呼び出されて、丸括弧の中に書かれた式の値が、その関数に引数として渡される。そして、呼び出された関数が返した戻り値が、関数呼び出しの値になる。
- ドット (.) という二項演算子を含む式によって求められる関数は、「メソッド」(method) と呼ばれることもある。

5.1.2 関数の定義

JavaScript は、さまざまな関数を規定しています。たとえば、`parseInt` は、JavaScript が規定している関数のひとつです。

JavaScript で使うことのできる関数は、JavaScript が規定しているものだけではありません。プログラムを書く人がプログラムの中で独自の関数を作って、それを使う、ということも可能です。

関数を作ることを、関数を「定義する」(define) と言います。

関数は、「関数宣言」(function declaration) と呼ばれるものを書くことによって定義することができます。

5.1.3 プログラムの構造

JavaScript では、「プログラムというのは、文または関数宣言を並べることによってできる列である」と規定されています。ですから、関数宣言は、プログラムの中に何個でも並べて書くことができます。

文は、基本的には、並んでいる順序のとおりに実行されます。ですから、文が並んでいる順序は、プログラムの動作を左右することになります。

それに対して、関数宣言をどのような順序で並べたとしても、プログラムの動作は変わりません。プログラムの下のほうに書かれた関数宣言によって定義された関数を、それよりも上に書かれた文から呼び出す、ということも可能です。

しかし、だからと言って、関数宣言は無秩序な順序で並べてもかまわないというわけではありません。関数宣言を無秩序な順序で並べると、人間にとって理解しにくいプログラムになってしまいます。関数宣言は、できるだけプログラムの構造を反映した順序で並べることをお勧めします。

5.1.4 基本的な関数宣言の書き方

引数を受け取らない関数を定義する関数宣言は、

```
function 識別子 () {
  プログラム
```

```
}
```

と書きます。「識別子」のところには、名前として関数に与えたい識別子を書きます。そして、「プログラム」のところには、関数にしたい動作を書きます。たとえば、

```
function world() {
  WScript.echo("こんにちは、世界。");
}
```

という関数宣言を書くことによって、「こんにちは、世界。」という文字列を出力する関数を作って、 world という名前をそれに与えることができます。

プログラムの例　world.js

```
world();

function world() {
  WScript.echo("こんにちは、世界。");
}
```

実行例

```
>cscript //nologo world.js
こんにちは、世界。
```

5.1.5　関数を定義するという機能は何のためにあるのか

JavaScript は、関数を定義するという機能を持っています。そのような、名前によって呼び出すことのできる動作を定義するという機能は、JavaScript に限らず、ほとんどすべてのプログラミング言語が備えているものです。プログラミング言語は、いったい何のために、このような機能を備えているのでしょうか。

人間にとって、複雑なものを理解するというのは、容易なことではありません。ですから、複雑なものを作るときには、それを人間にとって理解しやすいものにする工夫をすることが、とても大切です。

複雑なものを人間にとって理解しやすいものにする上で重要なことは、少数の部品の組み合わせによって全体を構築するということです。個々の部品は、もしもそれ自体が複雑なものである場合は、それもまた、少数の部品の組み合わせによって構築される必要があります。

つまり、単純な部品を組み合わせることによって少し複雑な部品を作って、少し複雑な部品を組み合わせることによってさらに複雑な部品を作って、……というように、部品を階層的に組み合わせることによって構築されたものは、それがどれだけ複雑なものであっても人間にとって理解しやすい、ということです。

プログラムを書く場合も、もしもそれが複雑なものである場合は、それを人間にとって理解しやすいものにする工夫が必要になります。部品を階層的に組み合わせて構築することによって、人間にとって理解しやすいものを作ることができるという原理は、プログラムの場合にも有効です。プログラムを構築するための部品というのは、名前によって呼び出すことのできる動作です。

名前によって呼び出すことのできる動作を定義するという機能がプログラミング言語に備わっているのはいったい何のためなのか、という問題の解答は、人間にとって理解しやすいプログラムを書くことができるようにするため、ということになります。

5.2　スコープ

5.2.1　スコープの基礎

識別子と、その識別子が名前として与えられている対象とのあいだの関係が保たれる、プログラムの上での範囲は、その識別子の「スコープ」(scope) と呼ばれます。

プログラミング言語の多くは、識別子のスコープに関する規則を定めています。JavaScript も例外ではありません。

5.2.2　グローバルなスコープ

プログラムの全域というスコープは、「グローバルなスコープ」(global scope) と呼ばれます。

JavaScript では、関数宣言の外で変数に与えられた識別子は、グローバルなスコープを持つことになります。そのような、それに与えられた識別子がグローバルなスコープを持つ変数は、「グ

ローバル変数」(global variable) と呼ばれます。

関数宣言の内部も、グローバルなスコープの一部分になります。ですから、関数宣言の中でも、グローバル変数に与えられた識別子を書くことによって、その変数の内容を求めたり、その変数にデータを設定したりすることができます。

プログラムの例 global.js

```
var global = "私はグローバル変数です。";
WScript.echo(global);
func();
WScript.echo(global);

function func() {
  WScript.echo(global);
  global = "関数宣言の中で違うデータを設定してみました。";
  WScript.echo(global);
}
```

実行例

```
>cscript //nologo global.js
私はグローバル変数です。
私はグローバル変数です。
関数宣言の中で違うデータを設定してみました。
関数宣言の中で違うデータを設定してみました。
```

5.2.3 ローカルなスコープ

プログラムの全域よりも狭い、限定されたスコープは、「ローカルなスコープ」(local scope) と呼ばれます。

グローバルなスコープを持つ識別子と同一の識別子を、ローカルなスコープを持つ識別子として使っても、問題はありません。ただし、その場合、ローカルなスコープの内部では、グローバルなスコープを持つ識別子を名前として持つものは、隠されて見えなくなります。

JavaScript では、関数宣言の中で変数に与えられた識別子は、その関数宣言の中だけというスコープを持つことになります。そのような、それに与えられた識別子が関数宣言の中だけというローカルなスコープを持つ変数は、「ローカル変数」(local variable) と呼ばれます。

プログラムの例 local.js

```
var a = "私はグローバル変数の a です。";
WScript.echo(a);
func();
WScript.echo(a);

function func() {
  var a = "私は func のローカル変数の a です。";
  WScript.echo(a);
  funcfunc();
  WScript.echo(a);
}

function funcfunc() {
  var a = "私は funcfunc のローカル変数の a です。";
  WScript.echo(a);
}
```

実行例

```
>cscript //nologo local.js
私はグローバル変数の a です。
私は func のローカル変数の a です。
私は funcfunc のローカル変数の a です。
私は func のローカル変数の a です。
私はグローバル変数の a です。
```

5.2.4　ローカルなスコープという規則のメリット

ところで、「関数宣言の中で変数に与えられた識別子は、その関数宣言の中だけというスコープを持つ」という規則には、いったいどのようなメリットがあるのでしょうか。

もしも、「関数宣言の中で変数に与えられた識別子もグローバルなスコープを持つ」という規則が定められていたとするとどうなるか、ということについて考えてみましょう。その場合、変数に識別子を与えるときには、その識別子がすでに別の関数宣言の中で使われていないか、ということに細心の注意を払う必要があります。うっかりと同一の識別子を複数の関数宣言の中で使うと、思わぬ不具合が発生しかねません。

つまり、「関数宣言の中で変数に与えられた識別子は、その関数宣言の中だけというスコープを持つ」という規則は、「関数宣言を書くときに、その関数宣言の外でどのような識別子が使われているかということを、まったく気にする必要がない」というメリットを、プログラムを書く人に与えてくれているのです。

5.3　引数

5.3.1　仮引数

引数を受け取る関数を定義する関数宣言は、

```
function 識別子 (識別子, ··· ) {
  プログラム
}
```

と書きます。つまり、丸括弧の中に、何個かの識別子を書くわけです。丸括弧の中に書かれた識別子は、「仮引数名」(formal argument name) と呼ばれます。

仮引数名は、「仮引数」(formal argument) と呼ばれるものに、名前として与えられます。仮引数というのは、関数が受け取った引数が設定される変数のことです。

仮引数は、ローカル変数です。つまり、仮引数名のスコープは、関数宣言の中だけです。

次のプログラムの中で定義されている stars という関数は、引数として受け取った個数のアスタリスクを出力します。

プログラムの例　stars.js

```
WScript.stdout.write("星の個数を入力してください。: ");
var n = parseInt(WScript.stdin.readLine());
stars(n);

function stars(length) {
  for (var i = 1; i <= length; i++) {
    WScript.stdout.write("*");
  }
  WScript.echo();
}
```

実行例

```
>cscript //nologo stars.js
星の個数を入力してください。: 40
****************************************
```

5.3.2　仮引数名の順序

関数宣言の中で並んでいるそれぞれの仮引数名と、関数呼び出しの中で並んでいるそれぞれの式とは、同じ順序で結び付けられます。たとえば、

```
function namako(a, b, c) {
  プログラム
}
```

という関数宣言で定義された関数を、

```
namako(24, 33, 81)
```

という関数呼び出しで呼び出したとすると、 a に 24 が、 b に 33 が、 c に 81 が設定されることになります。

次のプログラムの中で定義されている rect という関数は、文字列と縦の個数と横の個数を引数として受け取って、その文字列を長方形の形に並べたものを出力します。

プログラムの例 rect.js

```
WScript.stdout.write("文字列を入力してください。: ");
var s = WScript.stdin.readLine();
WScript.stdout.write("横の個数を入力してください。: ");
var w = parseInt(WScript.stdin.readLine());
WScript.stdout.write("縦の個数を入力してください。: ");
var h = parseInt(WScript.stdin.readLine());
rect(s, w, h);

function rect(str, width, height) {
  for (var i = 1; i <= height; i++) {
    for (var j = 1; j <= width; j++) {
      WScript.stdout.write(str);
    }
    WScript.echo();
  }
}
```

実行例

```
>cscript //nologo rect.js
文字列を入力してください。: JavaScript
横の個数を入力してください。: 4
縦の個数を入力してください。: 3
JavaScriptJavaScriptJavaScriptJavaScript
JavaScriptJavaScriptJavaScriptJavaScript
JavaScriptJavaScriptJavaScriptJavaScript
```

5.4 戻り値

5.4.1 return 文

戻り値を返す関数を定義したいときは、「return 文」(return statement) と呼ばれる文を関数宣言の中に書きます。

return 文は、

```
return 式;
```

と書きます。この文は、その中の式を評価して、その値を戻り値として返して、関数を終了させる、という動作をあらわしています。

次のプログラムの中で定義されている square という関数は、1 個の数値を引数として受け取って、その 2 乗を戻り値として返します。

プログラムの例 square.js

```
WScript.stdout.write("数値を入力してください。: ");
var a = parseFloat(WScript.stdin.readLine());
WScript.echo(a + "の 2 乗は" + square(a) + "です。");

function square(a) {
  return a*a;
}
```

実行例

```
>cscript //nologo square.js
数値を入力してください。: 9
9 の 2 乗は 81 です。
```

n が自然数だとするとき、n 自身を除いた n のすべての約数の和が n と等しいならば、n は、「完全数」(perfect number) と呼ばれます。たとえば、6 は、$1+2+3=6$ ですから、完全数です。

次のプログラムの中で定義されている perfect という関数は、1 個の自然数を引数として受け取って、それが完全数ならば真、そうでなければ偽を戻り値として返します。

プログラムの例 perfect.js

```
WScript.stdout.write("自然数を入力してください。: ");
var n = parseInt(WScript.stdin.readLine());
WScript.stdout.write(n + "は完全数");
if (perfect(n)) {
  WScript.echo("です。");
} else {
  WScript.echo("ではありません。");
}

function perfect(n) {
  var m = 0;
  for (var i = 1; i < n; i++) {
    if (n%i == 0) {
      m += i;
    }
  }
  return n == m;
}
```

実行例

```
>cscript //nologo perfect.js
自然数を入力してください。: 28
28 は完全数です。
>cscript //nologo perfect.js
自然数を入力してください。: 30
30 は完全数ではありません。
```

5.4.2 return 文を実行しないで終了した関数の戻り値

JavaScript では、あらゆる関数が戻り値を返します。関数が return 文を実行しないで動作を終了した場合も、その関数は戻り値を返しています。その場合に関数が返す戻り値は、未定義値です。

5.5 関数式

5.5.1 関数式の基礎

これまで説明してきたように、関数を定義したいときは、関数宣言というものを書けばいいわけですが、実は、関数を定義する方法はそれだけではなくて、もうひとつあります。

関数を定義するもうひとつの方法というのは、「関数式」(function expression) と呼ばれるものを書く、という方法です。

関数式というのは式の一種です。関数式を評価すると、関数が定義されて、その関数が式の値になります。

5.5.2 関数式の書き方

関数式は、

```
function(識別子, ··· ) {
  プログラム
}
```

と書きます。function と左丸括弧とのあいだに識別子を書かないというのが、関数宣言との相違点です。中括弧の中に書くプログラムが短い場合は、改行を省略して、

```
function(識別子, ··· ) { プログラム }
```

と書いてもかまいません。

たとえば、次の式を書くことによって、引数を 3 倍する関数を定義することができます。

```
function(a) { return a*3; }
```

5.5.3 変数への関数の設定

関数というのはデータですから、変数に関数を設定することも可能です。
JavaScript コンソールを使って試してみましょう。

```
> var sanbai = function(a) { return a*3; };
  undefined
```

これによって、引数を 3 倍する関数が定義されて、それが sanbai という変数に設定されます。ですから、 sanbai という識別子を評価することによって、その関数を求めることができます。

```
> sanbai(7)
  21
```

5.5.4 即時関数

関数式によって定義された関数は、その場で即座に呼び出すことも可能です。そのような、その場で即座に呼び出される関数は、「即時関数」(immediate function) と呼ばれます。
JavaScript コンソールを使って試してみましょう。

```
> (function(a) { return a*3; })(7)
  21
```

即時関数が使われるのは、ほとんどの場合、ローカルなスコープを作るという目的のためです。

5.6 再帰

5.6.1 再帰とは何か

この節では、「再帰」(recursion) と呼ばれるものについて説明したいと思います。
再帰というのは、全体と同じものが一部分として含まれているという性質のことです。再帰という性質を持っているものは、「再帰的な」(recursive) と形容されます。
ここに、1 台のカメラと 1 台のモニターがあるとします。まず、それらを接続して、カメラで撮影した映像がモニターに映し出されるようにします。そして次に、カメラをモニターの画面に向けます。すると、モニターの画面には、そのモニター自身が映し出されることになります。そして、映し出されたモニターの画面の中には、さらにモニター自身が映し出されています。このときにモニターの画面に映し出されるのは、再帰という性質を持っている映像、つまり再帰的な映像です。
また、先祖と子孫の関係も再帰的です。なぜなら、先祖と子孫との中間にいる人々も、やはり先祖と子孫の関係で結ばれているからです。

5.6.2 基底

再帰という性質を持っているものは、全体と同じものが一部分として含まれているわけですが、その構造は、内部に向かってどこまでも続いている場合もあれば、どこかで終わっている場合もあります。
再帰的な構造がどこかで終わっている場合、その中心には、その内部に再帰的な構造を持っていない何かがあります。そのような、再帰的な構造の中心にあって、その内部に再帰的な構造を持っていないものは、その再帰的な構造の「基底」(basis) と呼ばれます。
先祖と子孫の関係では、親子関係というのが、その再帰的な構造の基底になります。

5.6.3 言葉の再帰的な定義

何かの言葉を定義するために、定義される当の言葉が使われている場合、その定義は、「再帰的な定義」(recursive definition) と呼ばれます。
たとえば、「子孫」という言葉は、次のように定義することができます。

- B が A の子供であるならば、B は A の子孫である。
- C が A の子供であって、かつ B が C の子孫であるならば、B は A の子孫である。

この定義は二つの記述から構成されていますが、二つ目の記述は、「子孫」という言葉を使って「子孫」という言葉を説明しています。ですから、これは再帰的な定義だということになります。
再帰的な定義は、選択が可能な二つ以上の記述から構成されます。そして、それらの選択肢の少なくとも一つは、定義される当の言葉を使わないでそれを説明するものでないといけません。

なぜなら、すべての選択肢が定義される当の言葉を使っているとすると、その定義は循環に陥ってしまうからです。

定義される当の言葉を使わないでそれを説明する選択肢というのは、基底について記述した選択肢のことです。たとえば、「子孫」を再帰的に定義する場合は、親子関係が基底になりますので、「B が A の子供であるならば、B は A の子孫である」という、基底について記述した選択肢を作っておくことによって、定義が循環に陥るのを防ぐことができます。

5.6.4　関数の再帰的な定義

関数は、再帰的に定義することが可能です。関数を再帰的に定義するというのは、定義される当の関数を使って関数を定義するということです。再帰的な構造を持っている概念を取り扱う関数は、再帰的に定義するほうが、再帰的ではない方法で定義するよりもすっきりした記述になります。

言葉を再帰的に定義する場合と同じように、関数を再帰的に定義する場合も、それが循環に陥ることを防ぐために、基底について記述した選択肢を作っておくことが必要になります。

5.6.5　階乗

n が 0 またはプラスの整数だとするとき、n から 1 までの整数をすべて乗算した結果、つまり、

$$n \times (n-1) \times (n-2) \times \cdots \times 1$$

という計算の結果は、n の「階乗」(factorial) と呼ばれて、$n!$ と書きあらわされます。ただし、$0!$ は 1 だと定義します。

たとえば、$5!$ は、

$$5 \times 4 \times 3 \times 2 \times 1$$

という計算をすればいいわけですから、120 ということになります。

階乗という概念は、再帰的な構造を持っています。なぜなら、階乗は、

$$\begin{cases} & 0! = 1 \\ n \geq 1 \text{ ならば} & n! = n \times (n-1)! \end{cases}$$

というように再帰的に定義することができるからです。

階乗を求める関数も、再帰的に定義することができます。次のプログラムは、階乗を求める `factorial` という関数を再帰的に定義しています。

プログラムの例　`factorial.js`

```
WScript.stdout.write("整数を入力してください。: ");
var n = parseInt(WScript.stdin.readLine());
WScript.echo(n + "の階乗は" + factorial(n) + "です。");

function factorial(n) {
  if (n == 0) {
    return 1;
  } else if (n >= 1) {
    return n * factorial(n-1);
  }
}
```

実行例

```
>cscript //nologo factorial.js
整数を入力してください。: 5
5 の階乗は 120 です。
```

5.6.6　フィボナッチ数列

第 0 項と第 1 項が 1 で、第 2 項以降はその直前の 2 項を足し算した結果である、という数列は、「フィボナッチ数列」(Fibonacci sequence) と呼ばれます。フィボナッチ数列の第 0 項から第 12 項までを表にすると、次のようになります。

n	0	1	2	3	4	5	6	7	8	9	10	11	12
第 n 項	1	1	2	3	5	8	13	21	34	55	89	144	233

フィボナッチ数列というのは再帰的な構造を持っている概念ですので、その第 n 項 (F_n) は、

$$\begin{cases} & F_0 = 1 \\ & F_1 = 1 \\ n \geq 2 \text{ ならば} & F_n = F_{n-2} + F_{n-1} \end{cases}$$

というように再帰的に定義することができます。

フィボナッチ数列の第 n 項を求める関数も、再帰的に定義することができます。次のプログラムは、フィボナッチ数列の第 n 項を求める `fibonacci` という関数を再帰的に定義しています。

プログラムの例 `fibonacci.js`

```
WScript.stdout.write("整数を入力してください。: ");
var n = parseInt(WScript.stdin.readLine());
WScript.echo("フィボナッチ数列の第" + n + "項は" +
  fibonacci(n) + "です。");

function fibonacci(n) {
  if (n == 0) {
    return 1;
  } else if (n == 1) {
    return 1;
  } else if (n >= 2) {
    return fibonacci(n-2) + fibonacci(n-1);
  }
}
```

実行例

```
>cscript //nologo fibonacci.js
整数を入力してください。: 7
フィボナッチ数列の第 7 項は 21 です。
```

5.6.7 最大公約数

n がプラスの整数で、m が 0 またはプラスの整数だとするとき、n と m の両方に共通する約数のうちで最大のもののことを、n と m の「最大公約数」(greatest common measure, GCM) と呼びます（m が 0 の場合は、n と m の最大公約数は n だと定義します)。たとえば、54 と 36 の最大公約数は 18 です。

n と m の最大公約数は、「ユークリッドの互除法」(Euclidean algorithm) と呼ばれる次のような再帰的な手順を実行することによって求めることができます。

- m が 0 ならば、n が、n と m の最大公約数である。
- m が 1 以上ならば、n を m で除算したときのあまりを求めて、その結果を r とする。そして、m と r の最大公約数を求めれば、その結果が n と m の最大公約数である。

プログラムの例 `gcm.js`

```
WScript.stdout.write("整数を入力してください。: ");
var n = parseInt(WScript.stdin.readLine());
WScript.stdout.write("整数を入力してください。: ");
var m = parseInt(WScript.stdin.readLine());
WScript.echo(n + "と" + m + "の最大公約数は" +
  gcm(n, m) + "です。");

function gcm(n, m) {
  if (m == 0) {
    return n;
  } else if (m >= 1) {
    return gcm(m, n%m);
  }
}
```

実行例

```
>cscript //nologo gcm.js
```

```
整数を入力してください。: 54
整数を入力してください。: 36
54 と 36 の最大公約数は 18 です。
```

5.7　高階関数

5.7.1　高階関数の基礎

JavaScript では関数というのはデータですので、関数は、引数として関数を受け取ることもできますし、戻り値として関数を返すことも可能です。

引数として関数を受け取る関数や、戻り値として関数を返す関数は、「高階関数」(higher-order function) と呼ばれます。

引数として関数を受け取る関数を呼び出す場合、引数として渡す関数は、名前を持っている必要がありません。ですから、そのような関数は、多くの場合、関数宣言ではなくて関数式によって定義されます。

5.7.2　関数を受け取る関数

JavaScript コンソールを使って、関数を受け取る関数を定義して、それを呼び出してみましょう。

まず、関数式で関数を定義して、それを hundred という変数に設定します。定義するのは、f という仮引数に関数を受け取って、100 を引数にして f を呼び出して、その戻り値を返す、という関数です。

```
> var hundred = function(f) { return f(100); };
  undefined
```

引数を 3 倍して返す関数を引数にして hundred を呼び出すと、次のようになります。

```
> hundred(function(a) { return a*3; })
  300
```

引数を 2 乗して返す関数を引数にして hundred を呼び出すと、次のようになります。

```
> hundred(function(a) { return a*a; })
  10000
```

5.7.3　繰り返しの関数

引数として関数を受け取って、その関数の呼び出しを繰り返す、という関数を定義することによって、繰り返しを簡単に記述することができるようになります。

次のプログラムの中で定義されている times という関数は、引数として整数 n と関数 f を受け取って、0 から $n-1$ までのそれぞれの整数を引数にして f を呼び出します。

プログラムの例　times.js

```
WScript.stdout.write("整数を入力してください。: ");
var n = parseInt(WScript.stdin.readLine());
times(n, function(n) { WScript.stdout.write(n + " "); });
WScript.echo();

function times(n, f) {
  for (var i = 0; i < n; i++) {
    f(i);
  }
}
```

実行例

```
>cscript //nologo times.js
整数を入力してください。: 20
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

5.7.4　関数を返す関数

JavaScript コンソールを使って、関数を返す関数を定義して、それを呼び出してみましょう。

まず、関数式で関数を定義して、それを divide という変数に設定します。定義するのは、引数として数値 a を受け取って関数を返す、という関数です。戻り値として返す関数は、引数として数値 b を受け取って、a を b で割り算した結果を返す、という関数です。

```
> var divide = function(a) { return function(b) { return a/b; }};
  undefined
```

たとえば、100 を引数にして divide を呼び出して、その戻り値を、5 を引数にして呼び出すと、次のようになります。

```
> divide(100)(5)
  20
```

5.7.5 関数を合成する関数

1 個の引数を受け取って戻り値を返す、f と g という 2 個の関数が与えられたとき、1 個の引数を受け取って、それを引数にして g を呼び出して、その戻り値を引数にして f を呼び出して、その戻り値を返す、という動作をする関数を定義することを、f と g を「合成する」(compose) と言います。

次のプログラムの中で定義されている compose という関数は、1 個の引数を受け取って戻り値を返す関数 f と関数 g を引数として受け取って、f と g を合成した関数を返します。

プログラムの例　compose.js

```
WScript.stdout.write("数値を入力してください。: ");
var a = parseFloat(WScript.stdin.readLine());
WScript.echo(a + "を 1000 倍して 333 を加算した結果は" +
  compose(func1, func2)(a) + "です。");

function func1(a) {
  return a+333;
}

function func2(a) {
  return a*1000;
}

function compose(f, g) {
  return function(a) { return f(g(a)); };
}
```

実行例

```
>cscript //nologo compose.js
数値を入力してください。: 77
77 を 1000 倍して 333 を加算した結果は 77333 です。
```

第 6 章　オブジェクト

6.1　オブジェクトの基礎

6.1.1 オブジェクトとは何か

JavaScript では、「オブジェクト」(object) と呼ばれる種類のデータを扱うことができます。

オブジェクトは、0 個以上の変数から構成されているデータだと考えることができます。オブジェクトを構成している変数は、「プロパティー」(property) と呼ばれます。

オブジェクトは、「オブジェクト型」(object type) という型を持っています。

関数というのはオブジェクトの一種です。

6.1.2 オブジェクト初期化子

オブジェクトは、「オブジェクト初期化子」」(object initializer) と呼ばれる式によって生成することができます。

オブジェクト初期化子は、

{ プロパティー設定, ··· }

と書きます。オブジェクト初期化子を評価すると、その中に書かれたそれぞれのプロパティー設定によってプロパティーが作られて、それらのプロパティーから構成されるオブジェクトが生成されます。そして、そのオブジェクトがオブジェクト初期化子の値になります。

プロパティー設定は、

識別子: 式

と書きます[1]。プロパティー設定は、「コロン (:) の左側に書かれた識別子を名前として持つプロパティーを宣言して、そのプロパティーに、コロンの右側に書かれた式の値を初期値として設定する」という動作を意味しています。

JavaScript コンソールを使って試してみましょう。

```
> var a = { namako: 37, umiushi: 61 };
  undefined
```

このように入力することによって、初期値として 37 が設定された namako というプロパティーと、初期値として 61 が設定された umiushi というプロパティーを持つオブジェクトが生成されて、そのオブジェクトが a という変数に設定されます。

typeof を使って、 a に設定されたデータの型を調べてみましょう。

```
> typeof a
  "object"
```

6.1.3　プロパティーを指定する式

プロパティーに設定されているデータを求めたり、そこに別のデータを設定したりするためには、そのプロパティーを指定する式を書く必要があります。

プロパティーを指定する式は、ドット (.) という二項演算子を使って、

式 . 識別子

と書きます。そうすると、ドットの左側に書かれた式の値として得られたオブジェクトが持っている、ドットの右側に書かれた識別子を名前として持つプロパティーが指定されます。

プロパティーを指定する式を評価すると、そのプロパティーに設定されているデータが、その値として得られます。

a という変数が、上で入力した変数文で宣言されているとすると、次のように、プロパティーを指定する式を入力することによって、プロパティーに設定されているデータを求めることができます。

```
> a.namako
  37
> a.umiushi
  61
```

プロパティーに設定されているデータを変更したいときは、代入演算子、インクリメント演算子、デクリメント演算子のいずれかを使います。

```
> a.namako = 88
  88
> a.namako
  88
```

6.1.4　プロパティーの追加

プロパティーは、オブジェクトを生成した後で追加することも可能です。

代入演算子の = を使って、まだ存在していないプロパティーにデータを設定すると、指定した名前を持つプロパティーが作られることになります。

```
> a.hitode = 24
  24
> a.hitode
  24
```

[1] 実は、プロパティー設定のコロン (:) の左側には、識別子だけではなくて、文字列リテラルや数値リテラルを書くこともできるのですが、それについては第 6.2 節で説明します。

6.1.5 プロパティーの削除

プロパティーは、削除することもできます。

プロパティーを削除したいときは、`delete` という単項演算子を使います。演算子を含む式は、

`delete` プロパティーを指定する式

と書きます。この式の値は、プロパティーを削除することができた場合は真、できなかった場合は偽です。

```
> a.namako
  88
> delete a.namako
  true
> a.namako
  undefined
```

6.1.6 オブジェクトと変数との関係

オブジェクトはデータの一種ですが、変数とのあいだの関係は、オブジェクトとそれ以外のデータとで、大きな違いがあります。

オブジェクト以外のデータを変数に設定した場合、そのデータは、変数という箱の中に格納されます。オブジェクト以外のデータが `a` という変数に設定されているとするとき、`a` に設定されているデータを `b` という変数に設定したとすると、`a` に格納されているデータの複製が `b` に格納されます。ですから、そののち別のデータを `a` に設定したとしても、`b` はその影響を受けません。このことを、JavaScript コンソールで確かめてみましょう。

```
> var a = 3;
  undefined
> var b = a;
  undefined
> a = 4
  4
> b
  3
```

オブジェクトを変数に設定した場合、そのオブジェクトは、変数という箱の中には格納されません。その場合に変数に格納されるのは、オブジェクトを指し示す、「参照」(reference) と呼ばれるものです。オブジェクトが `a` という変数に設定されているとするとき、`a` に設定されているオブジェクトを `b` という変数に設定したとすると、`a` に格納されている参照の複製が `b` に格納されます。`a` に格納されている参照と `b` に格納されている参照は、同一のオブジェクトを指し示しています。ですから、そののち、そのオブジェクトのプロパティーに別のデータを設定したとすると、`a` も `b` も、その影響を受けることになります。このことを、JavaScript コンソールで確かめてみましょう。

```
> var a = { x: 3 };
  undefined
> var b = a;
  undefined
> a.x = 4
  4
> b.x
  4
```

6.2 連想配列

6.2.1 連想配列の基礎

「組」(pair) と呼ばれるものが集まってできている集合は、「連想配列」(associative array) と呼ばれます。

組は、「キー」(key) と呼ばれるデータと「値」(value) と呼ばれるデータとを対応させたものです。一つの連想配列の中にあるそれぞれの組は、キーが異なっています。ですから、キーを指定することによって、そのキーに対応している値を求めることができます。また、キーを指定することによって、そのキーに対応している値を変更することも可能です。

JavaScript で「オブジェクト」と呼ばれているものは、実は連想配列です。つまり、オブジェ

クトを構成しているそれぞれのプロパティーは、組だということです。キーに相当するのはプロパティーの名前で、値に相当するのはプロパティーに設定されているデータです。

6.2.2　プロパティー設定

第 6.1 節で、プロパティー設定は、

識別子: 式

と書く、と説明しましたが、実は、プロパティー設定のコロン (:) の左側に書くことができるものは、識別子だけではありません。文字列リテラルや数値リテラルを書くことも可能です。

識別子の代わりに文字列リテラルや数値リテラルを書いた場合は、それらのリテラルの値が、名前としてプロパティーに与えられます。つまり、文字列や数値をキーにすることができるわけです[2]。

JavaScript コンソールに、次のような変数文を入力してみましょう。

```
> var a = { "(^^)v": 100, 3.14: "pi" };
  undefined
```

この変数文によって、(^^)v という文字列がキーで 100 という数値が値という組と、3.14 という数値がキーで pi という文字列が値という組を持つ連想配列が生成されて、その連想配列が a という変数に設定されます。

6.2.3　添字表記

キーを指定して値を求めたり、キーに対応する値を変更したりするためには、その組を指定する記述を書く必要があります。組を指定する記述は、「添字表記」(subscription) と呼ばれます。

添字表記は、

式 $_1$ [式 $_2$]

と書きます。そうすると、式 $_1$ の値として得られた連想配列が持っている、式 $_2$ の値をキーとする組が指定されます。

添字表記の中で使われる角括弧 ([]) は、「添字演算子」(subscript operator) と呼ばれる演算子です。式を角括弧で囲んだものは、「添字」(subscript) と呼ばれます。

添字表記は、式として評価することができます。添字表記を評価すると、それによって指定された組の値が、その式の値として得られます。

a という変数が、上で入力した変数文で宣言されているとすると、次のように、添字表記を入力することによって、組に設定されているデータを求めることができます。

```
> a["(^^)v"]
  100
> a[3.14]
  "pi"
```

組に設定されているデータを変更したいときは、代入演算子、インクリメント演算子、デクリメント演算子のいずれかを使います。

```
> a["(^^)v"] = 120
  120
> a["(^^)v"]
  120
```

6.2.4　組の追加

代入演算子の = を使って、まだ存在していない組にデータを設定すると、そのデータを値として持つ組が作られることになります。

```
> a["(^^;)"] = 30
  30
> a["(^^;)"]
  30
```

6.2.5　組の削除

組を削除したいときは、 delete を使って、

[2] 実は、識別子を書いた場合も、その識別子と同一の文字列がキーになっていると考えることができます。

delete 添字表記

という式を書きます。

```
> a[3.14]
  "pi"
> delete a[3.14]
  true
> a[3.14]
  undefined
```

6.3 for-in 文

6.3.1 for-in 文の基礎

繰り返しを記述するための文としては、第 4 章で、while 文、do-while 文、for 文という三つの文を紹介しましたが、あとひとつ、まだ紹介していない文があります。それは、for-in 文という文です。

for-in 文は、オブジェクト（連想配列）を構成しているすべてのプロパティー（組）に対して何らかの処理を実行したい、というときに使われる文です。

6.3.2 for-in 文の書き方

for-in 文は、

```
for (var 識別子 in 式) {
  文の列
}
```

と書きます。

for-in 文は、次のような動作をあらわしています。

(1) 識別子を名前とする変数を作る。
(2) 式を評価する。その値がオブジェクト（連想配列）ではない場合、for-in 文の動作は終了する。
(3) 式の値がオブジェクトだった場合は、それを構成しているそれぞれのプロパティー名（キー）について、それを変数に設定して文の列を実行する、ということを繰り返す。

次のプログラムは、何個かの組を読み込んで（キーとして end が入力されると、読み込みを終了します）、そののち、読み込んだすべての組を出力します。

プログラムの例 forin.js

```
var object = readObject();
for (var k in object) {
  WScript.echo("キーが" + k + "で値が" + object[k] + "の組");
}

function readObject() {
  var o = {};
  var k;
  var v;
  while ((k = readKey()) != "end") {
    v = readValue();
    o[k] = v;
  }
  return o;
}

function readKey() {
  WScript.stdout.write("キー（end で終了）: ");
  return WScript.stdin.readLine();
}

function readValue() {
  WScript.stdout.write("値: ");
  return WScript.stdin.readLine();
```

```
}
```

実行例

```
>cscript //nologo forin.js
キー（end で終了）: Japan
値: Tokyo
キー（end で終了）: Italy
値: Rome
キー（end で終了）: Ethiopia
値: Addis Ababa
キー（end で終了）: end
キーが Japan で値が Tokyo の組
キーが Italy で値が Rome の組
キーが Ethiopia で値が Addis Ababa の組
```

6.4　メソッド

6.4.1　メソッドについての復習

この節では、メソッドについて説明したいと思います。

メソッドについては、すでに第 2.8.6 項で簡単に説明しましたので、まずは、そこで説明したことを復習しておきましょう。

「メソッド」というのは関数の別の呼び名です。メソッドと関数は本質的には同じものですが、それを求めるための式の形に応じてそれぞれの呼び名を使い分けるという慣習があります。「関数」と呼ばれるのは、それが単なる識別子によって求められる場合で、「メソッド」と呼ばれるのは、それが、ドット (.) という二項演算子を含む、

式 . 識別子

という形の式によって求められる場合です。

6.4.2　メソッドの定義

ところで、メソッドを定義したいときは、いったいどうすればいいのでしょうか。

メソッドを求めるときに使われるドット (.) という二項演算子を含む式は、第 6.1.3 項で説明した、プロパティーを指定する式です。このことは、メソッドというのはプロパティーに設定された関数のことだ、ということを意味しています。ですから、プロパティーに関数を設定すると、その関数はメソッドになります。

JavaScript コンソールで試してみましょう。

```
> var a = { square: function(x) { return x*x; } };
  undefined
> a.square(7)
  49
```

6.4.3　this

メソッドは、自分を持っているオブジェクトを処理することができます。つまり、メソッドは、自分と同じオブジェクトが持っているプロパティーを自由に操作することができるということです。

それでは、メソッドを定義するときに、そのメソッドと同じオブジェクトが持っているプロパティーは、どうすれば指定することができるのでしょうか。

メソッドを定義する関数式の中で、this という式を評価すると、その値として、定義されるメソッドを持っているオブジェクトが得られます。ですから、メソッドを定義する関数式の中に、

```
this.namako
```

という式を書くことによって、そのメソッドと同じオブジェクトが持っている、namako というプロパティーを指定することができます。

JavaScript コンソールで試してみましょう。

```
> var a = { namako: 88 };
  undefined
> a.getNamako = function() { return this.namako; }
```

```
    function () { return this.namako; }
> a.getNamako()
  88
```

6.5 コンストラクタ

6.5.1 コンストラクタの基礎

第 6.1 節では、オブジェクトを生成する方法として、オブジェクト初期化子という式を書く、というものを紹介しました。しかし、オブジェクトを生成する方法はそれだけではありません。

オブジェクトを生成する方法としては、オブジェクト初期化子を使うという方法のほかに、new という演算子と、「コンストラクタ」(constructor) と呼ばれる関数を使う、というものもあります。コンストラクタというのは、オブジェクトの初期化を目的とする関数のことです。

new とコンストラクタを使ってオブジェクトを生成するという方法は、特定の機能を持つオブジェクトを何個も生成する必要がある場合に便利です。

6.5.2 new を含む式

new という演算子を含む式は、

```
new 関数呼び出し
```

と書きます。

new を含む式を評価すると、まずオブジェクトが生成されて、次に関数呼び出しが評価され、それによってコンストラクタが呼び出されます。そして、生成されたオブジェクトが、new を含む式の値として得られます。

JavaScript コンソールで試してみましょう。まず、何もしないコンストラクタを定義します。

```
> var Cons = function() {};
  undefined
```

コンストラクタの名前は、通常はこのように、先頭の文字を大文字にします。ただし、これは文法ではなくて慣習ですので、先頭の文字を小文字にしたとしても、文法的には間違いではありません。

次に、このコンストラクタと new を使って、オブジェクトを生成しましょう。

```
> var a = new Cons();
  undefined
```

生成されたものが本当にオブジェクトかどうかを、typeof を使って確かめてみましょう。

```
> typeof a
  "object"
```

6.5.3 オブジェクトの初期化

new を含む式の中に書かれた関数呼び出しと、普通の関数呼び出しとは、文法的には同じものですが、それによってどのように関数が呼び出されるのかという点に関しては、それらは同じではありません。

new を含む式の中に書かれた関数呼び出しによって呼び出された関数は、あたかも、new によって生成されたオブジェクトが持っているメソッドであるかのように実行されます。

ですから、コンストラクタを定義する関数宣言または関数式の中に、this という式を書くことによって、そのコンストラクタが初期化する対象となるオブジェクトを求めることができます。

JavaScript コンソールで試してみましょう。

```
> var Cons = function(n) { this.namako = n; };
  undefined
> var a = new Cons(77);
  undefined
> a.namako
  77
```

6.5.4　有理数

この節の最初のところでも述べたように、new とコンストラクタを使ってオブジェクトを生成するという方法は、特定の機能を持つオブジェクトを何個も生成する必要がある場合に便利です。

そこで、そのような場合の例として、有理数をあらわすオブジェクトを new とコンストラクタを使って生成するプログラムを書いてみましょう。ちなみに、「有理数」(rational number) というのは、分数を使うことによってあらわすことのできる数値のことです。

次のプログラムは、有理数をあらわすオブジェクトを使って、二つの有理数の和を求めます。

プログラムの例　rational.js

```
var a = readRational();
var b = readRational();
WScript.echo(a + "と" + b + "との和は" + a.add(b) + "です。");

function readRational() {
  WScript.echo("有理数を分数で入力してください。");
  WScript.stdout.write("分子: ");
  var n = parseInt(WScript.stdin.readLine());
  WScript.stdout.write("分母: ");
  var d = parseInt(WScript.stdin.readLine());
  return new Rational(n, d);
}

function Rational(numera, denomi) {
  this.numera = numera;
  this.denomi = denomi;
  this.gcm = function(n, m) {
    if (m == 0) {
      return n;
    } else if (m >= 1) {
      return this.gcm(m, n%m);
    }
  };
  this.reduce = function() {
    var m = this.gcm(this.numera, this.denomi);
    return new Rational(this.numera/m, this.denomi/m);
  }
  this.add = function(r) {
    return (new Rational(
      this.numera*r.denomi + r.numera*this.denomi,
      this.denomi*r.denomi)).reduce();
  };
  this.toString = function() {
    return this.denomi + "分の" + this.numera;
  };
}
```

実行例

```
>cscript //nologo rational.js
有理数を分数で入力してください。
分子: 1
分母: 8
有理数を分数で入力してください。
分子: 3
分母: 8
8 分の 1 と 8 分の 3 との和は 2 分の 1 です。
```

このプログラムの中で使われている有理数のオブジェクトは、toString というメソッドを持っています。これは、オブジェクトを文字列に変換するメソッドです。このように、オブジェクトを文字列に変換するメソッドを、toString という名前で定義しておくと、オブジェクトを文字列に変換する必要が生じたとき、そのメソッドが自動的に呼び出されることになります。

6.6 プロトタイプ

6.6.1 プロトタイプの基礎

JavaScript には、「プロトタイプ」(prototype) と呼ばれるメカニズムがあります。これは、すでに存在している何らかのオブジェクトを原型にして、そこからオブジェクトを生成することができる、というメカニズムです。

この節では、このメカニズムについて説明したいと思います。

6.6.2 プロトタイプオブジェクト

関数というのはオブジェクトの一種ですから、プロパティーを持つことができます。あらゆる関数は、prototype という名前のプロパティーを持っていて、そこにはオブジェクトが設定されています。

JavaScript コンソールで試してみましょう。

```
> var f = function() {};
  undefined
> typeof f.prototype
  "object"
```

関数が持っている prototype プロパティーに設定されているオブジェクトは、「プロトタイプオブジェクト」(prototype object) と呼ばれます。

コンストラクタというのは関数ですから、やはりプロトタイプオブジェクトを持っています。new とコンストラクタを使ってオブジェクトを生成すると、そのオブジェクトは、コンストラクタが持っているプロトタイプオブジェクトを原形にして生成されます。

6.6.3 暗黙の参照

プロトタイプオブジェクトを原形にして生成されたオブジェクトは、そのプロトタイプオブジェクトに対する参照を持っています。オブジェクトが持っている、自分を生成したプロトタイプオブジェクトに対する参照は、「暗黙の参照」(implicit reference) と呼ばれます。

new とコンストラクタを使ってオブジェクトを生成した場合、そのオブジェクトは、自分が持っている暗黙の参照を使うことによって、自分の原型となったプロトタイプオブジェクトのプロパティーからデータを取得することができます。ただし、暗黙の参照を使うことによってプロトタイプオブジェクトのプロパティーにデータを設定することはできません（データを設定しようとすると、そのオブジェクト自身の中に同じ名前のプロパティーが作られることになります）。

JavaScript コンソールで試してみましょう。まず、何もしないコンストラクタを定義します。

```
> var Cons = function() {};
  undefined
```

次に、このコンストラクタのプロトタイプオブジェクトに、hoge というプロパティーを追加します。

```
> Cons.prototype.hoge = 33
  33
```

次に、このコンストラクタと new を使って、オブジェクトを生成します。

```
> var a = new Cons();
  undefined
```

このオブジェクトは、プロトタイプオブジェクトに対する暗黙の参照を持っています。ですから、a.hoge という式を書くことによって、プロトタイプオブジェクトが持っている hoge というプロパティーからデータを取得することができるはずですので、確かめてみましょう。

```
> a.hoge
  33
```

6.6.4 プロトタイプチェーン

オブジェクト A を原形にしてオブジェクト B を生成して、オブジェクト B を原形にしてオブジェクト C を生成した場合、オブジェクト B はオブジェクト A に対する暗黙の参照を持っていて、オブジェクト C はオブジェクト B に対する暗黙の参照を持っています。ですから、オブジェ

クト C は、暗黙の参照をたどることによって、オブジェクト A のプロパティーからデータを取得することができます。

このように、暗黙の参照は、オブジェクトからオブジェクトへ鎖のように連なっています。このようなオブジェクトの連鎖は、「プロトタイプチェーン」(prototype chain) と呼ばれます。

JavaScript コンソールで試してみましょう。まず、Base というコンストラクタを定義して、それが持っているプロトタイプオブジェクトに、 hoge というプロパティーを追加します。

```
> var Base = function() {};
  undefined
> Base.prototype.hoge = 44
  44
```

次に、 Extend というコンストラクタを定義して、その prototype プロパティーに、 Base を使って生成したオブジェクトを設定します。

```
> var Extend = function() {};
  undefined
> Extend.prototype = new Base()
  Base
```

次に、 Extend を使ってオブジェクトを生成します。

```
> var a = new Extend();
  undefined
```

このオブジェクトは、プロトタイプチェーンをたどることによって、hoge というプロパティーからデータを取得することができるはずですので、確かめてみましょう。

```
> a.hoge
  44
```

6.6.5　継承

すでに存在するオブジェクトから機能を引き継ぐことによって新しいオブジェクトを作ることができるという機能は、「継承」(inheritance) と呼ばれます。

よく似た機能を持つ何種類かのオブジェクトを作る場合、それらに共通する機能は、それぞれのオブジェクトごとに書くのではなくて、ひとつにまとめて書くほうが効率的です。継承を使うことによって、共通する機能をひとつのオブジェクトにまとめておいて、個々の特殊なオブジェクトはそのオブジェクトから機能を引き継ぐ、というプログラムの書き方ができます。

JavaScript では、プロトタイプチェーンを利用することによって、継承を実現することができます。

第 7 章　文字列

7.1　文字列の基礎

7.1.1　文字列オブジェクト

JavaScript では、文字列はオブジェクトではありませんが、文字列に対して何らかの処理を実行するためには、多くの場合、「文字列オブジェクト」(string object) と呼ばれるオブジェクトが必要になります。

文字列オブジェクトというのは、その内部に、処理の対象となる文字列と、その文字列を処理するためのさまざまなメソッドを持っているオブジェクトのことです。

文字列オブジェクトは、明示的に生成することも可能ですが、通常は必要に応じて暗黙的に生成されます。

7.1.2　文字列の長さ

文字列を構成している文字の個数は、その文字列の「長さ」(length) と呼ばれます。

文字列オブジェクトが持っている length というプロパティーには、その文字列オブジェクトが持っている文字列の長さが設定されています。

```
> "umiushi".length
  7
```

ちなみに、"umiushi" という式の値は文字列オブジェクトではなくて文字列ですが、この場合は、文字列オブジェクトが暗黙的に生成されて、その文字列オブジェクトの length から長さが取得されます。

7.1.3 文字列オブジェクトのコンストラクタ

文字列オブジェクトを明示的に生成したいときは、new とコンストラクタを使います。

文字列オブジェクトを初期化するコンストラクタは、String という名前です。このコンストラクタに引数として文字列を渡すと、文字列オブジェクトはその文字列で初期化されます。

```
> var s = new String("namako");
  undefined
```

valueOf というメソッドを呼び出すことによって、文字列オブジェクトから、それが持っている文字列を取得することができます。

```
> s.valueOf()
  "namako"
```

String は、fromCharCode というメソッドを持っています。このメソッドは、任意の個数の整数を引数として受け取って、それらの整数を文字コードとする文字から構成される文字列を戻り値として返します。

```
> String.fromCharCode(65, 66, 67, 68, 69, 70, 71)
  "ABCDEFG"
```

7.2 文字列オブジェクトのメソッド

7.2.1 この節について

文字列オブジェクトは、文字列を処理するさまざまなメソッドを持っています。この節では、それらのメソッドのうちの三つを紹介したいと思います。

メソッドの動作を説明するときは、しばしば、そのメソッドを持っているオブジェクトに言及する必要が生じます。このチュートリアルでは、メソッドを持っているオブジェクトを示すために、「自身」という言葉を使うことにしたいと思います。

7.2.2 文字の取り出し

charAt と charCodeAt というメソッドは、引数として整数を受け取って、その整数を番号とする文字を自身から取り出して、その文字を戻り値として返します。番号は、先頭の文字を 0 番目と数えます。

charAt の戻り値は、1 個の文字だけから構成される文字列で、charCodeAt の戻り値は文字コードです。

```
> "ABCDEFG".charAt(3)
  "D"
> "ABCDEFG".charCodeAt(3)
  68
```

次のプログラムは、読み込んだ文字列を構成しているそれぞれの文字のうしろに 1 個の空白を追加することによってできる文字列を出力します。

プログラムの例 space.js

```
WScript.stdout.write("文字列を入力してください。: ");
var s = WScript.stdin.readLine();
WScript.echo(space(s));

function space(s) {
  var r = "";
  for (var i = 0; i < s.length; i++) {
    r += s.charAt(i) + " ";
  }
  return r;
}
```

実行例

```
>cscript //nologo space.js
文字列を入力してください。: mathematics
m a t h e m a t i c s
```

次のプログラムは、読み込んだ文字列から英大文字だけを取り出して並べることによってできる文字列を出力します。

プログラムの例　abbreviate.js

```
WScript.stdout.write("文字列を入力してください。: ");
var s = WScript.stdin.readLine();
WScript.echo(abbreviate(s));

function abbreviate(s) {
  var a = "";
  var c;
  for (var i = 0; i < s.length; i++) {
    c = s.charCodeAt(i);
    if (isUpperCase(c)) {
      a += String.fromCharCode(c);
    }
  }
  return a;
}

function isUpperCase(c) {
   return c >= 65 && c <= 90;
}
```

実行例

```
>cscript //nologo abbreviate.js
文字列を入力してください。: Unidentified Flying Object
UFO
```

7.2.3　部分文字列の取り出し

文字列の一部分になっている文字列は、「部分文字列」(substring) と呼ばれます。

substring というメソッドは、部分文字列を自身から取り出して、それを戻り値として返します。このメソッドには、引数として、取り出す部分文字列の位置を示す 2 個の整数を渡します。1 個目を i、2 個目を j とすると、取り出されるのは、i 番目から $j-1$ 番目までの文字から構成される部分文字列です。2 個目の引数を省略すると、i 番目から最後までの部分文字列が取り出されます。

```
> "ABCDEFG".substring(2, 5)
  "CDE"
> "ABCDEFG".substring(2)
  "CDEFG"
```

次のプログラムは、読み込んだ文字列の中央（文字列の長さが奇数の場合は、中央の文字の左側）にスラッシュ(/) を挿入することによってできる文字列を出力します。

プログラムの例　center.js

```
WScript.stdout.write("文字列を入力してください。: ");
var s = WScript.stdin.readLine();
WScript.echo(center(s));

function center(s) {
  var n = Math.floor(s.length / 2);
  return s.substring(0, n) + "/" + s.substring(n);
}
```

実行例

```
>cscript //nologo center.js
文字列を入力してください。: metaphysics
```

第 8 章　配列

8.1　配列の基礎

8.1.1　配列とは何か

番号によって個々のデータを指定することのできるデータの集合は、「配列」(array) と呼ばれます。

配列を構成しているそれぞれのデータは、その配列の「要素」(element) と呼ばれます。

JavaScript でも、配列を扱うことができます。JavaScript の配列は、オブジェクトの一種です。

8.1.2　配列初期化子

配列は、「配列初期化子」(array initializer) と呼ばれる式を書くことによって生成することができます。

配列初期化子は、

[式, 式, ･･･]

と書きます。配列初期化子を評価すると、その中に書かれた式が評価されて、それぞれの値を要素とする配列が生成されます。そして、生成された配列が、配列初期化子の値になります。

JavaScript コンソールで試してみましょう。

```
> [38, true, "namako"]
  [38, true, "namako"]
```

8.1.3　配列の添字表記

配列初期化子によって配列を生成した場合、その配列を構成しているそれぞれの要素は、配列初期化子の中に書かれた式の順序と同じ順序で一列に並んでいます。そしてそれらの要素には、それらが並んでいる順番のとおりに、0 から始まる番号が与えられています。配列の要素は、その番号によって指定することができます。

配列の要素を求めたり、配列の要素を変更したりしたいときは、連想配列の組を指定する場合と同じように、添字表記を使ってその要素を指定します。

配列の要素を指定する添字表記は、連想配列の組を指定する添字表記と同じように、「添字演算子」(subscript operator) と呼ばれる角括弧 ([]) を使って、

$式_1$ [$式_2$]

と書きます。そうすると、$式_1$ の値として得られた配列が持っている、$式_2$ の値を番号とする要素が指定されます。

JavaScript コンソールで試してみましょう。

```
> var a = [58, 66, 31, 49]
  undefined
> a[2]
  31
> a[2] = "umiushi"
  "umiushi"
> a
  [58, 66, "umiushi", 49]
```

次のプログラムは、0 またはプラスの整数を読み込んで、それを英単語の列に変換した結果を出力します。

プログラムの例　ntow.js

```
WScript.stdout.write("整数を入力してください。: ");
var n = parseInt(WScript.stdin.readLine());
WScript.echo(numberToWords(n));

function numberToWords(n) {
  if (n >= 10) {
```

```
    return numberToWords(Math.floor(n/10)) + " " +
      digitToWord(n%10);
  } else {
    return digitToWord(n);
  }
}

function digitToWord(d) {
   return ["zero", "one", "two", "three", "four", "five",
     "six", "seven", "eight", "nine"][d];
}
```

実行例

```
>cscript //nologo ntow.js
整数を入力してください。: 329507184
three two nine five zero seven one eight four
```

8.1.4 配列の長さ

配列を構成している要素の個数は、その配列の「長さ」(length) または「大きさ」(size) と呼ばれます。

配列が持っている `length` というプロパティーには、その配列の長さが設定されています。

JavaScript コンソールで試してみましょう。

```
> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].length
  10
```

8.1.5 配列のコンストラクタ

配列は、配列初期化子によって生成することができるわけですが、`new` とコンストラクタを使って生成することも可能です。

配列のコンストラクタは、`Array` という名前です。0 またはプラスの整数を引数としてコンストラクタに渡すと、その整数を長さとする、未定義値から構成される配列が生成されます。

JavaScript コンソールで試してみましょう。

```
> var a = new Array(300);
  undefined
> a.length
  300
> a[0]
  undefined
```

8.1.6 エラトステネスのふるい

2 から n までの範囲にあるすべての素数を求めるための手順としては、「エラトステネスのふるい」(sieve of Eratosthenes) と呼ばれるものがよく知られています。

エラトステネスのふるいは、次のような手順です。

(1) 2 から n までのすべての整数を並べた列を作る。
(2) 2 を i とする。
(3) 次の (4) と (5) を、i^2 が n よりも大きくなるまで繰り返す。
(4) i が列の中にあるならば、その倍数のうちで列の中にあるものをすべて取り除く。ただし、i 自身は取り除かない。
(5) i に 1 を加算した整数を i とする。

この手順が終了すると、素数だけが列の中に残ります。

次のプログラムは、2 以上の整数 n を読み込んで、エラトステネスのふるいを使って 2 から n までの範囲にあるすべての素数を求めて、それらを出力します。

プログラムの例 `sieve.js`

```
WScript.stdout.write("2 以上の整数を入力してください。: ");
var n = parseInt(WScript.stdin.readLine());
sieve(n);
```

```
function sieve(n) {
  var sequence = createSequence(n);
  eratosthenes(sequence);
  writeSequence(sequence);
}

function createSequence(n) {
  var sequence = new Array(n+1);
  for (var i = 0; i <= n; i++) {
    sequence[i] = true;
  }
  return sequence;
}

function eratosthenes(sequence) {
  for (var i = 2; i*i < sequence.length; i++) {
    if (sequence[i]) {
      for (var j = i+i; j < sequence.length; j += i) {
        sequence[j] = false;
      }
    }
  }
}

function writeSequence(sequence) {
  for (var i = 2; i < sequence.length; i++) {
    if (sequence[i]) {
      WScript.stdout.write(i + " ");
    }
  }
  WScript.echo();
}
```

実行例

```
>cscript //nologo sieve.js
2 以上の整数を入力してください。: 80
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79
```

8.2 配列のメソッド

8.2.1 この節について

配列は、自身を処理するさまざまなメソッドを持っています。この節では、それらのメソッドのうちの主要なものを紹介したいと思います。

8.2.2 配列の末尾への要素の追加

配列の末尾に要素を追加したいときは、push というメソッドを使います。このメソッドは、引数として受け取ったデータを自身の末尾に追加して、そののちの自身の長さを戻り値として返します。

```
> var a = [87, 64, 53];
  undefined
> a.push(92)
  4
> a
  [87, 64, 53, 92]
```

8.2.3 配列の末尾にある要素の削除

配列の末尾にある要素を削除したいときは、pop というメソッドを使います。このメソッドは、自身の末尾にある要素を自身から削除して、削除した要素を戻り値として返します。

```
> var a = [87, 64, 53, 92];
  undefined
> a.pop()
```

```
  92
> a
  [87, 64, 53]
```

8.2.4　配列の連結

二つの配列を連結したいときは、concat というメソッドを使います。このメソッドは、引数として受け取った配列を自身の末尾に連結することによってできる配列を戻り値として返します。ただし、自身は変化しません。

```
> var a = [0, 1, 2];
  undefined
> var b = [3, 4, 5];
  undefined
> a.concat(b)
  [0, 1, 2, 3, 4, 5]
> a
  [0, 1, 2]
```

8.2.5　配列から文字列への変換

配列を文字列に変換したいときは、join というメソッドを使います。

引数として文字列を渡して join を呼び出すと、戻り値として、自身を構成しているそれぞれの要素をその引数で区切って連結した文字列が得られます。

```
> [61, false, "hitode"].join("///")
  "61///false///hitode"
```

引数を何も渡さないで join を呼び出した場合、それぞれの要素はコンマで区切られます。

```
> [61, false, "hitode"].join()
  "61,false,hitode"
```

配列は、暗黙のうちに文字列に変換されることもあります。その場合、それぞれの要素はコンマで区切られます。

```
> [83, 44, 27] + "hamaguri"
  "83,44,27hamaguri"
```

8.2.6　部分配列の取得

配列の連続した一部分は、「部分配列」(subarray) と呼ばれます。

配列から部分配列を取得したいときは、slice というメソッドを使います。

slice は、引数として 2 個の整数を受け取ります。1 個目を i、2 個目を j とすると、i 番目から $j-1$ 番目までの要素から構成される部分配列が得られます。

```
> [0, 1, 2, 3, 4, 5, 6].slice(2, 5)
  [2, 3, 4]
```

8.2.7　部分配列の置き換え

部分配列を別の配列によって置き換えたいときは、splice というメソッドを使います。

splice は、引数として 2 個以上の整数を受け取ります。1 個目を i、2 個目を n、3 個目以降の引数から構成される配列を a とすると、i 番目から始まる、長さが n の部分配列が、a に置き換わります。戻り値は、置き換わる前の部分配列です。

```
> var a = [0, 1, 2, 3, 4, 5, 6, 7];
  undefined
> a.splice(2, 4, 100, 101, 102)
  [2, 3, 4, 5]
> a
  [0, 1, 100, 101, 102, 6, 7]
```

8.2.8　配列の逆転

reverse というメソッドは、自身を逆の順序に並べ替えて、そののちの自身を戻り値として返します。

```
> var a = [0, 1, 2, 3, 4, 5, 6, 7];
```

```
  undefined
> a.reverse()
  [7, 6, 5, 4, 3, 2, 1, 0]
> a
  [7, 6, 5, 4, 3, 2, 1, 0]
```

8.2.9 配列のソート

配列を構成している要素を指定された順序で並べ替えることを、配列を「ソートする」(sort) と言います。

配列をソートしたいときは、sort というメソッドを使います。

sort は、順序を判定する関数を引数として受け取ります。順序を判定する関数は、2 個の引数を受け取って、1 個目を 2 個目よりも後ろに並べるべきならばプラスの数値、前に並べるべきならばマイナスの数値を返す、という動作をするように定義します。

```
> var a = [5, 3, 6, 1, 7, 0, 2, 4];
  undefined
> a.sort(function(x, y) { return x-y; })
  [0, 1, 2, 3, 4, 5, 6, 7]
> a
  [0, 1, 2, 3, 4, 5, 6, 7]
```

8.2.10 配列の写像

配列を構成しているそれぞれの要素に対して何らかの処理を実行した結果を並べることによってできる配列は、もとの配列の「写像」(map) と呼ばれます。

配列の写像を求めたいときは、map というメソッドを使います。このメソッドに引数として関数を渡すと、それぞれの要素をその関数で処理した結果から構成される配列が得られます。

```
> var a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
  undefined
> a.map(function(n) { return n*n; })
  [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

第 9 章　正規表現

9.1　正規表現の基礎

9.1.1 正規表現とは何か

文字列を処理するときには、しばしば、その中に含まれている特定のパターン（構造）を探し出す必要が生じます。そして、パターンを探し出すという処理を記述するためには、そのパターンそのものを記述する方法が必要になります。

「正規表現」(regular expression) というのは、文字列を使って文字列のパターンを表現するための言語のひとつです。また、この言語を使って作られた文字列のことも、「正規表現」と呼ばれます。

正規表現によってあらわされるパターンと文字列とが一致することを、正規表現（またはそれがあらわしているパターン）と文字列とが「マッチする」(match) と言います（名詞は「マッチング」(matching) です）。

プログラミング言語の多くは、その一部分として正規表現を含んでいます。正規表現は、プログラミング言語ごとの違いがほとんどありませんので、一度覚えてしまえば、別のプログラミング言語のために覚え直す必要はありません。

9.1.2 正規表現の基礎の基礎

1 個の特定の文字というパターンをあらわす正規表現は、たいていの場合、その文字そのものです。たとえば、A という文字は、A という正規表現によってあらわされます。

特定のパターンのうしろに特定のパターンが続いているという構造のことを「連接」(sequence) と呼びます。連接は、正規表現と正規表現とをパターンの順番のとおりに並べることによって表現することができます。たとえば、pi という正規表現は、p という文字のうしろに i という文字が続いているというパターン、つまり pi という文字列をあらわします。

ですから、たいていの場合、特定の文字列というパターンをあらわす正規表現は、それとまったく同じ文字列になります。たとえば、kamome という文字列は、kamome という正規表現によってあらわされます。

9.1.3　正規表現フラグ

正規表現を使って文字列を検索する場合、デフォルトでは、次のような規約のもとで検索が実行されます。

- 文字列を先頭から末尾へ向かって検索して、最初にマッチした時点で検索を終了する。
- 英字の大文字と小文字を区別する。
- 検索の対象が複数行の文字列だとしても、途中に含まれている改行の直前を行末、直後を行頭とみなさない。

これとは異なる規約のもとで検索を実行したい場合は、その規約を記述した文字列を使います。そのような文字列は、「正規表現フラグ」(regular expression flag) と呼ばれます。

正規表現フラグは、次の文字から構成される文字列です（文字を並べる順序に意味はありません）。

g　文字列の全体をグローバル (global) に検索する。

i　大文字と小文字を区別しない (ignore case)。

m　検索の対象が複数行 (multiline) の文字列の場合、途中に含まれている改行の直前を行末、直後を行頭とみなす

たとえば、gi または ig という文字列は、大文字と小文字を区別しないで文字列の全体を検索するという意味を持つ正規表現フラグです。

9.1.4　正規表現オブジェクト

JavaScript では、「正規表現オブジェクト」(regular expression object) と呼ばれるデータを扱うことができます。正規表現オブジェクトはオブジェクトの一種で、正規表現と正規表現フラグをその内部に持っています。

正規表現オブジェクトを生成する方法は二つあります。ひとつは「正規表現リテラル」(regular expression literal) と呼ばれるリテラルを書く方法で、もうひとつは new とコンストラクタを使う方法です。

正規表現リテラルを書くという方法は、プログラムの中に正規表現そのものを書きたいときに使われます。それに対して、new とコンストラクタを使うという方法は、文字列を正規表現オブジェクトに変換したいときに使われます。

9.1.5　正規表現リテラル

正規表現リテラルは、

```
/正規表現/
```

と書きます。つまり、正規表現の前後にスラッシュを書くわけです。正規表現リテラルを評価すると、その中に書かれた正規表現を持つ正規表現オブジェクトが値として得られます。たとえば、

```
/suzume/
```

という式を評価すると、suzume という正規表現を持つ正規表現オブジェクトが作られます。

正規表現フラグを持つ正規表現オブジェクトを、正規表現リテラルを使って生成したいときは、正規表現リテラルの末尾に正規表現フラグを書きます。たとえば、

```
/tsugumi/im
```

という式を評価すると、tsugumi という正規表現と im という正規表現フラグを持つ正規表現オブジェクトが値として得られます。

9.1.6　正規表現オブジェクトのコンストラクタ

正規表現オブジェクトを初期化するコンストラクタは、RegExp という名前です。このコンストラクタに、引数として 1 個の文字列を渡すことによって、その文字列を正規表現として持つ正規表現オブジェクトを生成することができます。たとえば、

```
new RegExp("uguisu")
```

という式を評価すると、uguisu という正規表現を持つ正規表現オブジェクトが値として得られます。

正規表現フラグを持つ正規表現オブジェクトを、new とコンストラクタを使って生成したいときは、コンストラクタに、2 個目の引数として正規表現フラグを渡します。たとえば、

```
new RegExp("sekirei", "gm")
```

という式を評価すると、sekirei という正規表現と gm という正規表現フラグを持つ正規表現オブジェクトが値として得られます。

9.1.7 文字列の検索

正規表現を使って文字列を検索したいときは、文字列オブジェクトが持っている search というメソッドを使います。

search は、引数として正規表現オブジェクトを受け取って、それが持っている正規表現で、先頭から末尾へ向かって自身を検索します。そして、マッチする部分文字列が見つかった場合は、その部分文字列の先頭の位置（先頭を 0 番目として、先頭から数えた番号）を返します。

```
> "012ab56ab9".search(/ab/)
  3
```

正規表現フラグの g が指定されていたとしても、戻り値は、最初にマッチした部分文字列の位置だけです。

```
> "012ab56ab9".search(/ab/g)
  3
```

マッチする部分文字列が見つからなかった場合は、−1 を返します。

```
> "012ab56ab9".search(/ba/)
  -1
```

9.2 メタ文字

9.2.1 メタ文字の基礎

正規表現を構成するそれぞれの文字は、基本的には、自分自身というパターンを意味しています。しかし、特定の文字列ではなくて、いくつかの文字列とマッチするようなパターンを表現するためには、自分自身ではない特別な意味をいくつかの文字に与える必要があります。

正規表現を書くために使われる、自分自身ではなくて何か別のことを意味している文字は、「メタ文字」(metacharacter) と呼ばれます。

どの文字をメタ文字として使うのかというのは、プログラミング言語ごとに多少の違いがありますが、

```
\  .  [  ]  -  ^  $  *  +  ?  {  }  (  )  |
```

というような文字が使われていて、それぞれのメタ文字の意味も、ほぼ統一されています。

9.2.2 文字クラス

文字の集合は、「文字クラス」(character class) と呼ばれます。正規表現は、指定された文字クラスに属する任意の文字というパターンを表現することができます。

9.2.3 すべての文字

すべての文字の集合という文字クラスに属する任意の文字というパターンは、ドット (.) というメタ文字によってあらわされます。たとえば、ma.iko という正規表現は、ma と iko とのあいだに 1 個の任意の文字がある、というパターンをあらわしています。

```
> "mariko".search(/ma.iko/)
  0
> "ma%iko".search(/ma.iko/)
  0
> "machiko".search(/ma.iko/)
  -1
```

9.2.4　文字の列挙

文字を列挙することによって文字クラスを指定したいときは、角括弧 (`[]`) というメタ文字を使います。文字クラスに属する文字を角括弧の中に列挙したものは、その文字クラスに含まれる任意の文字というパターンをあらわす正規表現になります。たとえば、`ma[mkr]iko` という正規表現は、`ma` と `iko` とのあいだに、`m`、`k`、`r` のうちのいずれかが 1 個だけある、というパターンをあらわしています。

```
> "mariko".search(/ma[mkr]iko/)
  0
> "magiko".search(/ma[mkr]iko/)
  -1
```

9.2.5　文字コードの範囲

文字を列挙することによって文字クラスを指定するのではなくて、文字コードの範囲を指定することによって文字クラスを指定する、ということも可能です。

文字コードの範囲で文字クラスを指定したいときは、マイナス (`-`) というメタ文字を使います。角括弧の中に、

文字$_1$ - 文字$_2$

という形のものを書くと、それは、文字$_1$ から文字$_2$ までという文字コードの範囲に含まれるすべての文字を列挙したのと同じ意味になります。たとえば、`[a-z]` という正規表現は、任意の英字の小文字というパターンをあらわします。

```
> "mariko".search(/ma[a-z]iko/)
  0
> "maRiko".search(/ma[a-z]iko/)
  -1
```

9.2.6　文字クラスに属さない文字

文字クラスを指定する方法には、それに属する文字について記述するという方法のほかに、それに属さない文字について記述するという方法もあります。

それに属さない文字を記述することによって文字クラスを指定したいときは、サーカムフレックス (`^`) というメタ文字を使います。左角括弧の直後にサーカムフレックスを書くと、文字クラスは、それに属する文字によって記述されるのではなくて、それに属さない文字によって記述されることになります。たとえば、`[^r]` という正規表現は、`r` という文字を除いたすべての文字から構成される文字クラスに属する任意の文字、というパターンをあらわします。

```
> "mamiko".search(/ma[^r]iko/)
  0
> "mariko".search(/ma[^r]iko/)
  -1
```

サーカムフレックスとマイナスとを組み合わせることも可能です。

```
> "maRiko".search(/ma[^a-z]iko/)
  0
> "mariko".search(/ma[^a-z]iko/)
  -1
```

文字クラスのうちで、しばしば使われるいくつかのものについては、次のような略記法があります。

略記法	もとの正規表現	説明
`\d`	`[0-9]`	数字
`\D`	`[^0-9]`	数字以外
`\w`	`[0-9A-Za-z]`	英数字
`\W`	`[^0-9A-Za-z]`	英数字以外
`\s`	`[ \t\n\r\f]`	ホワイトスペース
`\S`	`[^ \t\n\r\f]`	ホワイトスペース以外

9.2.7 パターンの繰り返し

メタ文字を使うことによって、同じパターンがいくつも繰り返されている、というパターンをあらわす正規表現を作ることも可能です。

0 回以上の繰り返しを表現したいときは、アスタリスク (*) というメタ文字を使います。何らかの正規表現の直後にアスタリスクを書いたものは、その正規表現によってあらわされたパターンが 0 回以上繰り返されたもの、というパターンを意味する正規表現になります。たとえば、m* という正規表現は、長さが 0 以上の m の列、というパターンをあらわします。

```
> "mammmmmmmiko".search(/mam*iko/)
  0
> "maiko".search(/mam*iko/)
  0
```

同じように、[mk]* という正規表現は、m または k から構成される、長さが 0 以上の文字列とマッチします。

```
> "mamkmmkkmkmiko".search(/ma[mk]*iko/)
  0
```

1 回以上の繰り返しとはマッチするけれども、繰り返しの回数が 0 回のときはマッチしないようにしたい、というときは、アスタリスクの代わりにプラス (+) というメタ文字を使います。たとえば、m+ という正規表現は、長さが 1 以上の m の列、というパターンをあらわします。

```
> "mammmmmmmiko".search(/mam+iko/)
  0
> "maiko".search(/mam+iko/)
  -1
```

0 回と 1 回の繰り返しだけとマッチする正規表現を書きたい、というときは、クエスチョンマーク (?) というメタ文字を使います。たとえば、m? という正規表現は、m の 0 回または 1 回の繰り返しというパターンをあらわします。

```
> "maiko".search(/mam?iko/)
  0
> "mamiko".search(/mam?iko/)
  0
> "mammiko".search(/mam?iko/)
  -1
```

繰り返しの回数を整数で指定したいときは、中括弧 ({}) というメタ文字を使います。たとえば、[0-9]{4} という正規表現は、4 桁の数字というパターンをあらわします。

```
> "ma8325iko".search(/ma[0-9]{4}iko/)
  0
> "ma832iko".search(/ma[0-9]{4}iko/)
  -1
> "ma83257iko".search(/ma[0-9]{4}iko/)
  -1
```

繰り返しの対象となる正規表現の範囲は、繰り返しをあらわす記述の直前にあるものだけ、という点に注意してください。つまり、mi+ という正規表現で繰り返しの対象になるのは、mi ではなくて、i だということです。

いくつかの正規表現の列を繰り返しの対象として指定したいときは、丸括弧 (()) というメタ文字で、指定したい正規表現の列を囲みます。たとえば、(mi)+ という正規表現は、mi というパターンを 1 回以上繰り返したもの、というパターンをあらわします。

```
> "mamimimiko".search(/ma(mi)+ko/)
  0
> "mamiiiko".search(/ma(mi)+ko/)
  -1
```

9.2.8 パターンの選択

二つのパターンのうちのどちらか、というパターンの選択を表現したいときは、縦棒 (|) というメタ文字を使います。

縦棒を使って選択を記述したいときは、

正規表現の列 | 正規表現の列

という形の正規表現を書きます。そうすると、縦棒の左右に書かれたそれぞれの正規表現の列があらわしているパターンのどちらか、という意味の正規表現になります。たとえば、a|b という正規表現は、a または b のどちらか、というパターンをあらわします。選択の対象がさらに選択であってもかまいませんので、a|b|c という正規表現を書くことによって、a または b または c のいずれか、というパターンをあらわすことも可能です。

縦棒による選択の範囲は、その直前と直後の正規表現だけではなくて、そのさらに左や右にも及ぶ、という点に注意してください。つまり、ab|cd という正規表現とマッチする文字列は、abd と acd ではなくて、ab と cd だということです。

選択の対象となる正規表現の範囲を狭く限定したいときは、その範囲を丸括弧 (()) で囲みます。たとえば、a(b|c)d という正規表現は、abd と acd という二つの文字列とマッチします。同じように、ma(ri|sa)ko という正規表現は、mariko と masako という二つの文字列とマッチします。

```
> "mariko".search(/ma(ri|sa)ko/)
  0
> "masako".search(/ma(ri|sa)ko/)
  0
> "madoko".search(/ma(ri|sa)ko/)
  -1
```

9.2.9　アンカー

パターンが文字列の中の特定の位置にあるときだけマッチする正規表現を書きたいときは、文字列の中の特定の位置にマッチする正規表現を使います。そのような正規表現は、「アンカー」(anchor) と呼ばれます。

たとえば、サーカムフレックス (^) というメタ文字は、文字列の先頭という位置とマッチするアンカーです。

```
> "xyzccccccc".search(/^xyz/)
  0
> "cccxyzcccc".search(/^xyz/)
  -1
```

ドルマーク ($) というメタ文字は、文字列の末尾という位置とマッチするアンカーです。

```
> "cccccccxyz".search(/xyz$/)
  7
> "cccxyzcccc".search(/xyz$/)
  -1
```

正規表現フラグの m が指定されている場合、サーカムフレックスは文字列の先頭だけではなくて行の先頭にもマッチして、ドルマークは文字列の末尾だけではなくて行の末尾にもマッチします。

```
> "ccc\nxyzcccc".search(/^xyz/)
  -1
> "ccc\nxyzcccc".search(/^xyz/m)
  4
> "cccxyz\ncccc".search(/xyz$/)
  -1
> "cccxyz\ncccc".search(/xyz$/m)
  3
```

9.2.10　エスケープ

ところで、メタ文字自身をあらわす正規表現、つまり特定のメタ文字だけとマッチする正規表現は、どのように書けばいいのでしょうか。たとえば、ドット (.) というメタ文字とマッチする正規表現というのは、いったいどう書くのでしょう。

メタ文字は、自分自身という意味を持っていませんので、メタ文字自身をあらわす正規表現を書くためには、メタ文字が持っている本来の意味を、その文字自身という意味に変更しないといけません。文字が持っている本来の意味を別の意味に変更することを、文字を「エスケープする」(escape) と言います。

文字をエスケープしたいときは、バックスラッシュ(\ ) というメタ文字を使います。バックスラッシュというのは、その直後に書かれた文字をエスケープするという意味を持つメタ文字です。

バックスラッシュでメタ文字をエスケープすると、その文字の意味は、メタ文字としての意味から、その文字自身という意味に変更されます。たとえば、ドット (.) というのはメタ文字ですので、ドット自身ではなくてメタ文字としての意味を持っています。しかし、\. というように、バックスラッシュの右側にドットを書くと、その 2 文字は、ドット自身をあらわす正規表現になります。

```
> "mariko".search(/ma.iko/)
  0
> "mariko".search(/ma\.iko/)
  -1
> "ma.iko".search(/ma\.iko/)
  0
```

ちなみに、バックスラッシュ自身もメタ文字ですから、バックスラッシュ自身をあらわす正規表現は、\\ と書く必要があります。

また、スラッシュ(/) という文字は、メタ文字ではありませんが、正規表現リテラルを作るための特別な文字ですので、正規表現リテラルの中でスラッシュ自身を記述するためには、\/ というように、バックスラッシュを使ってエスケープしないといけません。

9.3 正規表現を扱うメソッド

9.3.1 この節について

文字列オブジェクトは、search 以外にも、正規表現を扱うメソッドを持っています。この節では、そのようなメソッドを紹介したいと思います。

9.3.2 部分文字列の取り出し

正規表現とマッチした部分文字列を文字列から取り出したいときは、match というメソッドを使います。

match は、引数として正規表現オブジェクトを受け取って、それが持っている正規表現で、先頭から末尾へ向かって自身を検索します。そして、マッチした部分文字列から構成される配列を返します。マッチする部分文字列が存在しなかった場合は null を返します。文字列の全体を検索するためには、正規表現フラグの g を指定する必要があります。

```
> "xxx768xxxx5342xxx61xxxx".match(/[0-9]+/g)
  ["768", "5342", "61"]
```

9.3.3 部分文字列の置き換え

正規表現とマッチした部分文字列を別の文字列に置き換えたいときは、replace というメソッドを使います。

replace は、1 個目の引数として正規表現オブジェクト r、2 個目の引数として文字列 s を受け取って、r とマッチした部分文字列を s に置き換えることによってできる文字列を戻り値として返します。文字列の全体を検索するためには、正規表現フラグの g を指定する必要があります。

```
> "xxx768xxxx5342xxx61xxxx".replace(/[0-9]+/g, "000")
  "xxx000xxxx000xxx000xxxx"
```

2 個目の引数として、文字列ではなくて関数 f を渡した場合は、マッチした部分文字列を引数にして f を呼び出して、マッチした部分文字列を f の戻り値で置き換えた文字列を返します。

```
> var f = function(s) { return "[" + s + "]"; };
  undefined
> "xxx768xxxx5342xxx61xxxx".replace(/[0-9]+/g, f)
  "xxx[768]xxxx[5342]xxx[61]xxxx"
```

9.3.4 文字列の分割

区切りとなるパターンで文字列を分割したいときは、split というメソッドを使います。

split は、引数として正規表現オブジェクト r を受け取って、r とマッチした部分文字列を区切り文字列とみなして自身を分割して、それぞれの部分から構成される配列を戻り値として返し

ます。

```
> "ika:tako:ebi:uni".split(/:/)
  ["ika", "tako", "ebi", "uni"]
```

引数として、正規表現ではなくて空文字列を渡すと、split は、自身を個々の文字に分割して、1 文字の文字列から構成される配列を返します。

```
> "isoginchaku".split("")
  ["i", "s", "o", "g", "i", "n", "c", "h", "a", "k", "u"]
```

第 10 章　HTML 文書

10.1　HTML 文書の基礎

10.1.1　HTML

文書の中に特殊な文字列を埋め込むことによって、その文書の構造などを示すために使われる言語は、「マークアップ言語」(markup language) と呼ばれます。

マークアップ言語のひとつとして、「HTML」(Hypertext Markup Language) と呼ばれるものがあります。これは、ウェブページを書くために使われるマークアップ言語です。

HTML を使って書かれた文書は、「HTML 文書」(HTML document) と呼ばれます。HTML 文書が格納されるファイルの拡張子としては、.html または .htm が使われます。

それでは、次の HTML 文書をファイルに保存して（文字コードは UTF-8 にしてください）、そのファイルをブラウザーで開いてみてください。そうすると、ブラウザーの表示領域の中に、「私は HTML 文書です。」と表示されるはずです。

HTML 文書の例　htmldoc.html

```
<!DOCTYPE html>
<html>
  <head><title>HTML 文書</title></head>
  <body>
    <p>私は HTML 文書です。</p>
  </body>
</html>
```

10.1.2　要素

HTML 文書は、「要素」(element) と呼ばれる記述から構成されます。たとえば、

```
<p>私は HTML 文書です。</p>
```

という記述は、段落を作る要素です。

要素は、その先頭と末尾に、「タグ」(tag) と呼ばれる記述を書くことによって作ることができます。先頭に書かれるタグは「開始タグ」(start tag)、末尾に書かれるタグは「終了タグ」(end tag) と呼ばれます。たとえば、段落を作る要素は、 という開始タグと、 という終了タグを書くことによって作られます。

先頭のタグと末尾のタグで囲まれた部分は、要素の「内容」(content) と呼ばれます。たとえば、

```
<p>私は HTML 文書です。</p>
```

という要素の場合は、「私は HTML 文書です。」という部分がその内容です。

要素にはさまざまな種類があります。要素の種類は「要素型」(element type) と呼ばれ、「要素型名」(element type name) という名前によって識別されます。

タグの中には、要素型名を書く必要があります。たとえば、段落を作るためのタグの中には、p という要素型名を書きます。

先ほどの HTML 文書は、次のような要素型の要素から構成されています。

html	HTML 文書の全体を示す要素。
head	HTML 文書についての情報などを書く要素。
title	HTML 文書のタイトルとなる要素。

`body` HTML 文書の本文を記述する要素。

`p` 段落を作る要素。

要素には、開始タグと終了タグのペアによって作られるものだけではなくて、ひとつのタグだけによって作られるものもあります。そのような要素は「空要素」(empty element) と呼ばれます。そして、空要素を作るためのタグは、「空要素タグ」(empty element tag) と呼ばれます。

たとえば、改行は、`br` という要素型名を持つ、`<br>` という空要素タグによって作られます。

10.1.3 文書型宣言

HTML 文書の先頭には、「文書型宣言」(document type declaration) と呼ばれるものを書きます。

文書型宣言の書き方は、HTML のバージョンによって違います。「HTML5」と呼ばれる HTML のバージョンの場合、文書型宣言は、

```
<!DOCTYPE html>
```

と書きます。

10.1.4 属性

要素は、「属性」(attribute) と呼ばれる、名前と文字列の組を持っています。属性の名前は「属性名」(attribute name) と呼ばれ、属性に設定される文字列は「属性値」(attribute value) と呼ばれます。

HTML の要素がどのような名前の属性を持っていて、それがどのような意味を持っているかということは、それぞれの要素型ごとに定められています。

属性に対して属性値を設定したいときは、「属性指定」(attribute specification) と呼ばれる、

```
属性名="属性値"
```

という記述を開始タグの中に書きます。たとえば、

```
<p class="deluxe">私は HTML 文書です。</p>
```

というように、開始タグの中に属性指定を書くことによって、この要素が持っている `class` という属性に対して、`deluxe` という属性値を設定することができます。

10.1.5 HTML 文書と JavaScript

ブラウザーは、JavaScript で書かれたプログラムを実行することができます（ブラウザーによって実行されるプログラムは、「スクリプト」(script) と呼ばれることもあります)。

ブラウザーにプログラムを実行させる方法は二つあります。ひとつは、HTML 文書の中にそのプログラムを書くという方法で、もうひとつは、HTML 文書とは別のファイルにそのプログラムを格納しておいて、それを読み込む記述を HTML 文書の中に書くという方法です。

10.1.6 HTML 文書の中にプログラムを書く方法

HTML 文書の中にプログラムを書きたいときは、`script` という要素型の要素の内容として、そのプログラムを書きます。

HTML 文書の中に書かれたプログラムは、ブラウザーがその HTML 文書を読み込んだ時点で実行されます。

次の HTML 文書をブラウザーで開くと、「こんにちは、世界。」というメッセージが表示されたダイアログボックスが開きます。

HTML 文書の例 `script.html`

```
<!DOCTYPE html>
<html>
  <head><title>script</title></head>
  <body>
    <script>
      alert("こんにちは、世界。");
    </script>
  </body>
</html>
```

この HTML 文書の中のプログラムは、alert というメソッドを使っています。このメソッドは、引数として文字列を受け取って、その文字列を表示するダイアログボックスを開きます。これは、window という名前のオブジェクトが持っているメソッドですので、

```
window.alert("こんにちは、世界。");
```

というのが正式な書き方ですが、window. は省略することができます。

10.1.7　プログラムを読み込む記述を HTML 文書の中に書く方法

HTML 文書とは別のファイルに格納されているプログラムをブラウザーに実行させたいときは、そのプログラムを読み込む記述を HTML 文書の中に書きます。プログラムを読み込む記述は、script 要素を使って書きます。

script 要素は、src という属性を持っています。この属性に対して、プログラムが格納されているファイルの URI を設定しておくと、ブラウザーは、HTML 文書を読み込んだ時点で、そのプログラムを実行します。

次のプログラムと HTML 文書は、先ほどの HTML 文書を二つのファイルに分離したものです。

プログラムの例　world.js

```
alert("こんにちは、世界。");
```

HTML 文書の例　world.html

```
<!DOCTYPE html>
<html>
  <head><title>world</title></head>
  <body>
    <script src="world.js"></script>
  </body>
</html>
```

10.2　イベント

10.2.1　イベント属性

ユーザーによる操作などによって発生する出来事は、「イベント」(event) と呼ばれます。HTML 文書の中に書かれているプログラムは、イベントが発生したときに動作するように書いておくことができます。イベントが発生したときにプログラムによって実行される処理は、「イベント処理」(event processing) と呼ばれます。

イベント処理を実行するプログラムを書く方法のひとつは、「イベント属性」(event attribute) と呼ばれる属性を利用することです。

HTML の要素の多くは、「イベント属性」と呼ばれるいくつかの種類の属性を持っています。イベント属性に対してプログラムを属性値として設定しておくと、そのプログラムは、そのイベント属性の種類に対応するイベントが発生したときに実行されます。

HTML の要素が持っているイベント属性としては、次のようなものがあります。

onmouseover	マウスポインターが重なった。
onmouseout	マウスポインターが出ていった。
onmousedown	マウスのボタンが押された。
onmouseup	マウスのボタンが離された。
onclick	マウスでクリックされた。
onkeydown	キーが押された。
onkeyup	キーが離された。
onkeypress	キーが押されて離された。
onresize	ウィンドウの大きさが変更された。
onabort	画像の読み込みが中断された。
onerror	画像の読み込みの途中でエラーが発生した。

次の HTML 文書をブラウザーで開くと、二つの段落が表示されますので、それらの段落の上

にマウスポインターを重ねてみたり、それらの段落をマウスでクリックしてみてください。

HTML 文書の例 eventat.html

```
<!DOCTYPE html>
<html>
  <head><title>event attribute</title></head>
  <body>
    <p onmouseover="alert(’マウスポインターが重なりました。’)">
      私の上にマウスポインターを重ねてください。
    </p>
    <p onclick="alert(’クリックされました。’)">
      私をクリックしてください。
    </p>
  </body>
</html>
```

この例のように、二重引用符で囲まれた属性値の中に文字列リテラルを書く場合は、二重引用符の代わりに単一引用符（アポストロフィー）を使います。

10.2.2 イベントを発生させた要素のオブジェクト

イベント属性に設定されたプログラムの中で this という式を評価すると、そのイベントを発生させた要素のオブジェクトが、その式の値として得られます。

HTML 文書の例 elemobj.html

```
<!DOCTYPE html>
<html>
  <head><title>element object</title></head>
  <body>
    <p onclick="alert(this)">私をクリックしてください。</p>
  </body>
</html>
```

10.2.3 要素の内容

HTML の要素のオブジェクトは、innerHTML というプロパティーを持っていて、そこには要素の内容が設定されています。このプロパティーに文字列を設定することによって、要素の内容を変更することも可能です。

次の HTML 文書は、自分がクリックされた回数について述べる段落を表示します。

HTML 文書の例 innerhtml.html

```
<!DOCTYPE html>
<html>
  <head><title>innerHTML</title></head>
  <body>
    <p onclick="countup(this)">私は 0 回クリックされました。</p>
    <script>
      var count = 0;
      function countup(ele) {
        count++;
        ele.innerHTML = "私は" + count + "回クリックされました。";
      }
    </script>
  </body>
</html>
```

10.2.4 属性のプロパティー

HTML の要素のオブジェクトは、それが持っている属性と同じ名前のプロパティーを持っています。属性に属性値が設定されている場合、その属性値は、それと同じ名前のプロパティーにも設定されています。また、属性と同じ名前のプロパティーにデータを設定すると、そのデータが属性値として属性に設定されます。

次の HTML 文書によって表示される段落をクリックすると、その段落の要素が持っている id という属性に設定されている属性値（要素の ID）が、ダイアログボックスで表示されます。

HTML 文書の例 attribute.html

```
<!DOCTYPE html>
<html>
  <head><title>attribute</title></head>
  <body>
    <p id="id000" onclick="showId(this)">
      私をクリックしてください。
    </p>
    <script>
      function showId(ele) {
        alert("私の ID は" + ele.id + "です。");
      }
    </script>
  </body>
</html>
```

10.3 フォーム

10.3.1 フォームの基礎

HTML 文書は、人間と対話をするために使われる部品をブラウザーに表示させることができます。

ブラウザーによって表示される、人間との対話に使われる部品は、「フォーム部品」(form control) と呼ばれます。フォーム部品は、「フォーム部品要素」(form control element) と呼ばれる要素によって作られます。

フォーム部品を含んでいるウェブページの一部分または全体は、「フォーム」(form) と呼ばれます。フォームは、 form という要素型の要素によって作られます。

フォーム部品要素は、通常、 form 要素の中に書きます。

10.3.2 input 要素

フォーム部品にはさまざまな種類があるのですが、それらの多くは、 input という要素型の要素を書くことによって作ることができます。

input 要素は、 type という属性を持っています。この属性に対して属性値を設定することによって、作られるフォーム部品の種類が決まります。この属性に設定することのできる属性値の主要なものとしては、次のようなものがあります。

text	1 行の文字列を読み込むフォーム部品。
number	1 個の数値を読み込むフォーム部品。
checkbox	チェックボックス。
radio	ラジオボタン。
button	普通のボタン。
submit	フォームのデータを送信するボタン。

HTML の要素の多くは開始タグと終了タグによって作られるのですが、 input 要素は、1 個のタグだけで作られます。たとえば、1 行の文字列を読み込むフォーム部品を作る input 要素は、

```
<input type="text">
```

と書きます。

input 要素を作るタグのような、それ 1 個だけで要素を作るタグは、「空要素タグ」(empty-element tag) と呼ばれます。

input 要素は、 value という属性を持っています。この属性にデータを設定しておくと、それがフォーム部品の初期値になります。 type 属性の属性値が button または submit の場合、 value 属性に設定された文字列が、ボタンの上に表示されます。

次の HTML 文書は、5 種類のフォーム部品を表示します。

HTML 文書の例 input.html

```
<!DOCTYPE html>
<html>
  <head><title>input</title></head>
```

```
  <body>
    <form>
      <p><input type="text"></p>
      <p><input type="number"></p>
      <p><input type="checkbox">チェックボックス</p>
      <p><input type="radio">ラジオボタン</p>
      <p><input type="button" value="普通のボタン"></p>
    </form>
  </body>
</html>
```

10.3.3 HTML 文書のオブジェクト

window という名前のオブジェクトは、document というプロパティーを持っています。このプロパティーには、HTML 文書のオブジェクトが設定されています。

window.docunent は、window. を省略して、ただ単に document と書いてもかまいません。

10.3.4 フォーム部品のオブジェクト

プログラムは、フォーム部品に対するさまざまな処理を実行することができます。

フォーム部品に対する処理を実行するためには、そのフォーム部品のオブジェクトを求める必要があります。フォーム部品のオブジェクトは、HTML 文書のオブジェクトから求めることができます。

form 要素は、name という属性を持っています。form 要素の name 属性に名前を設定しておくと、それと同じ名前のプロパティーが HTML 要素のオブジェクトの中に作られて、フォームのオブジェクトがそのプロパティーに設定されます。

同じように、フォーム部品の要素も、name という属性を持っています。フォーム部品の要素の name 属性に名前を設定しておくと、それと同じ名前のプロパティーがフォームのオブジェクトの中に作られて、フォーム部品のオブジェクトがそのプロパティーに設定されます。ですから、

```
<form name="form1">
  <input type="text" name="text1">
</form>
```

という要素によって作られたフォームの中のフォーム部品のオブジェクトは、

```
document.form1.text1
```

という式を書くことによって求めることができます。

10.3.5 入力されたデータの取得

文字列を読み込むフォーム部品のオブジェクトは、value というプロパティーを持っています。フォーム部品に入力された文字列は、このプロパティーに設定されます。

次の HTML 文書は、ボタンが押されたときに、入力された文字列をダイアログボックスで表示します。

HTML 文書の例　value.html

```
<!DOCTYPE html>
<html>
  <head><title>value</title></head>
  <body>
    <form name="form1">
      <p><input type="text" name="text1"></p>
      <p><input type="button" value="表示"
           onclick="display()"></p>
    </form>
    <script>
      function display() {
        alert("入力された文字列は" +
          document.form1.text1.value + "です。");
      }
    </script>
  </body>
</html>
```

数値を読み込むフォーム部品のオブジェクトは、valueAsNumber というプロパティーを持っています。フォーム部品に入力された数値は、このプロパティーに設定されます。

次の HTML 文書は、ボタンが押されたときに、入力された数値の 2 乗をダイアログボックスで表示します。

HTML 文書の例 square.html

```
<!DOCTYPE html>
<html>
  <head><title>square</title></head>
  <body>
    <form name="form1">
      <p><input type="number" name="number1" value="0"></p>
      <p><input type="button" value="2 乗を求める"
           onclick="square()"></p>
    </form>
    <script>
      function square() {
        var n = document.form1.number1.valueAsNumber;
        alert(n + "の 2 乗は" + n*n + "です。");
      }
    </script>
  </body>
</html>
```

10.3.6 output 要素

プログラムがデータを処理した結果をフォームの中に表示したいときは、output という要素型の要素が使われます。

output 要素のオブジェクトが持っている value というプロパティーに対してデータを設定すると、そのデータがフォームの中に表示されます。

次の HTML 文書は、ボタンが押されたときに、入力された文字列を output 要素で表示します。

HTML 文書の例 output.html

```
<!DOCTYPE html>
<html>
  <head><title>output</title></head>
  <body>
    <form name="form1">
      <p><input type="text" name="text1"></p>
      <p><input type="button" value="表示"
           onclick="display()"></p>
      <p><output name="output1"></output></p>
    </form>
    <script>
      function display() {
        document.form1.output1.value = document.form1.text1.value;
      }
    </script>
  </body>
</html>
```

10.3.7 フォームのイベント属性

HTML の要素は、フォームに関連する次のようなイベント属性を持っています。

onsubmit	フォームの送信が要求された。
onfocus	入力フォーカスを得た。
onblur	入力フォーカスを失った。
oninput	データが入力された。
onchange	入力の内容などが変更された。

次の HTML 文書は、数値が入力されたときに、入力された二つの数値を乗算した結果を output 要素で表示します。

HTML 文書の例　oninput.html

```
<!DOCTYPE html>
<html>
  <head><title>oninput</title></head>
  <body>
    <form name="form1" oninput="multiply()">
      <p><input type="number" name="number1" value="0">と
         <input type="number" name="number2" value="0">
         とを乗算した結果は
         <output name="output1"></output>です。</p>
    </form>
    <script>
      function multiply() {
        var a = document.form1.number1.valueAsNumber;
        var b = document.form1.number2.valueAsNumber;
        document.form1.output1.value = a * b;
      }
    </script>
  </body>
</html>
```

10.3.8　チェックボックス

checkbox という属性値が type 属性に設定された input 要素は、チェックボックスを作ります。

チェックボックスにチェックが入っているかどうかを調べたいときは、そのチェックボックスのオブジェクトが持っている、checked というプロパティーを使います。このプロパティーには、チェックが入っているかどうかを示す真偽値が設定されています。チェックが入っているときは真、チェックが入っていないときは偽です。

次の HTML 文書は、チェックボックスの checked に設定されている真偽値をダイアログボックスで表示します。

HTML 文書の例　checkbox.html

```
<!DOCTYPE html>
<html>
  <head><title>checkbox</title></head>
  <body>
    <form name="form1">
      <p><input type="checkbox" name="checkbox1">
         チェックボックス</p>
      <p><input type="button" name="button1" value="表示"
                onclick="display()"></p>
    </form>
    <script>
      function display() {
        alert("checked の値は" +
          document.form1.checkbox1.checked + "です。");
      }
    </script>
  </body>
</html>
```

チェックボックスを、最初からチェックされた状態で表示されるようにしたいときは、チェックボックスを作る input 要素の空要素タグの中に、

```
<input type="checkbox" name="checkbox1" checked>
```

というように、checked という属性指定を書きます。

10.3.9　ラジオボタン

radio という属性値が type 属性に設定された input 要素は、ラジオボタンを作ります。

ラジオボタンを作る何個かの input 要素の name 属性に対して、同じ名前を設定すると、それらのラジオボタンはひとつのグループになって、それらの中からひとつを選択することができるようになります。

フォームのオブジェクトが持っているラジオボタンのプロパティーには、個々のラジオボタンのオブジェクトから構成される配列が設定されます。

チェックボックスの場合と同じように、ラジオボタンのオブジェクトが持っている checked というプロパティーには、チェックが入っているならば真、チェックが入っていないならば偽が設定されています。

最初に選択されているラジオボタンを指定したいときは、そのラジオボタンを作る input 要素の空要素タグの中に、 checked という属性指定を書いておきます。

次の HTML 文書は、ラジオボタンで選択された血液型をダイアログボックスで表示します。

HTML 文書の例　blood.html

```
<!DOCTYPE html>
<html>
  <head><title>blood</title></head>
  <body>
    <form name="form1">
      <p>血液型は？</p>
      <p><input type="radio" name="blood" value="A" checked>A 型</p>
      <p><input type="radio" name="blood" value="B">B 型</p>
      <p><input type="radio" name="blood" value="O">O 型</p>
      <p><input type="radio" name="blood" value="AB">AB 型</p>
      <p><input type="button" name="button1" value="表示"
                onclick="display()"></p>
    </form>
    <script>
      function display() {
        var selected;
        var length = document.form1.blood.length;
        var flag = true;
        for (var i = 0; i < length && flag; i++) {
          if (document.form1.blood[i].checked) {
            selected = document.form1.blood[i].value;
            flag = false;
          }
        }
        alert("選択されているのは" + selected + "型です。");
      }
    </script>
  </body>
</html>
```

10.3.10　リストボックス

リストボックスを作りたいときは、 select という要素型の要素と、 option という要素型の要素を使います。 select はリストボックスの全体を作るための要素型で、 option は、リストボックスによって選択される個々の項目を作るための要素型です。

select 要素は、開始タグと終了タグを使って作ります。そして、 option 要素は select 要素の中に書きます。

option 要素も、開始タグと終了タグを使って作ります。その内容として文字列を書くと、それが項目の内容になります。

フォームのオブジェクトが持っているリストボックスのプロパティーには、個々の項目のオブジェクトから構成される配列が設定されます。

項目のオブジェクトから構成される配列は、 selectedIndex というプロパティーを持っています。このプロパティーには、選択されている項目の番号が設定されます。

項目のオブジェクトが持っている text というプロパティーには、その項目の内容が設定されています。

次の HTML 文書は、リストボックスで選択された好きな食べ物をダイアログボックスで表示します。

HTML 文書の例　food.html

```
<!DOCTYPE html>
<html>
  <head><title>food</title></head>
```

```
  <body>
    <form name="form1">
      <p>好きな食べ物は？</p>
      <p>
        <select name="select1">
          <option>お好み焼き</option>
          <option>焼きそば</option>
          <option>オムライス</option>
          <option>カレーライス</option>
          <option>その他</option>
        </select>
      </p>
      <p><input type="button" name="button1" value="表示"
                onclick="display()"></p>
    </form>
    <script>
      function display() {
        var index = document.form1.select1.selectedIndex;
        var text = document.form1.select1[index].text;
        alert("選択されている項目の番号は" + index +
          "で、その項目の内容は" + text + "です。");
      }
    </script>
  </body>
</html>
```

10.4 DOM

10.4.1 DOM の基礎

これまで説明してきたように、ブラウザーにプログラムを実行させることによって、さまざまなことができるわけですが、「DOM」(document object model) と呼ばれるものを使えば、さらにさまざまなことができるようになります。

DOM というのは、プログラムによって HTML 文書などを操作するための API(application programming interface) のことです。API というのは、プログラムが動作する環境とプログラムとのあいだのインターフェースのことで、DOM の場合は、さまざまなメソッドから構成されています。

10.4.2 要素のオブジェクトの取得

DOM には、HTML 文書から要素のオブジェクトを取得するためのいくつかのメソッドがあります。

HTML 文書のオブジェクトが持っている `getElementById` というメソッドは、要素のオブジェクトを取得する DOM のメソッドのひとつです。

`getElementById` を使って要素オブジェクトを取得するためには、あらかじめ ID を要素に与えておく必要があります。要素が持っている `id` という属性に名前を設定しておくと、それがその要素の ID になります。たとえば、

```
<p id="paragraph1">私は段落です。</p>
```

と書くことによって、`paragraph1` という ID を持つ段落を作ることができます。

`getElementById` は、引数として要素の ID を受け取って、その ID を持つ要素のオブジェクトを戻り値として返します。たとえば、`paragraph1` という ID を持つ要素のオブジェクトは、

```
document.getElementById("paragraph1")
```

という式を書くことによって取得することができます。

次の HTML 文書は、ボタンをクリックすることによって、段落として表示された数値をカウントアップさせることができます。

HTML 文書の例　`countup.html`

```
<!DOCTYPE html>
<html>
  <head><title>countup</title></head>
```

```
  <body>
    <p id="number">0</p>
    <form name="form1">
      <p><input type="button" value="カウントアップ"
           onclick="countup()"></p>
    </form>
    <script>
      var number = document.getElementById("number");
      var count = 0;
      function countup() {
        count++;
        number.innerHTML = count;
      }
    </script>
  </body>
</html>
```

10.4.3 DOM によるイベント処理の基礎

第 10.2 節では、イベント処理を実行するプログラムの書き方として、イベント属性を使う方法について説明しました。しかし、イベント処理を実行するプログラムの書き方は、それだけではありません。イベント処理を実行するプログラムは、DOM を使うことによって書くこともできます。

イベント処理を実行するプログラムを DOM を使って書きたいときは、「イベントリスナー」(event listener) と呼ばれるものを使います。イベントリスナーというのは、イベントが発生したときに呼び出されるメソッドを持っているオブジェクトのことです。

イベントリスナーが持っている、イベントが発生したときに呼び出されるメソッドは、「イベントハンドラー」(event handler) と呼ばれます。イベントリスナーを要素のオブジェクトに設定しておくと、その要素でイベントが発生したときに、その要素のオブジェクトに設定されたイベントリスナーが持っているイベントハンドラーが呼び出されます。

イベントリスナーは、要素のオブジェクトだけではなくて、HTML 文書のオブジェクトにも設定することができます。HTML 文書のオブジェクトにイベントリスナーを設定した場合、そのイベントリスナーが持っているイベントハンドラーは、どの要素でイベントが発生した場合も呼び出されることになります。

イベントハンドラーは、「イベントオブジェクト」(event object) と呼ばれるオブジェクトを引数として受け取ります。このオブジェクトは、発生したイベントに関する各種の情報を持っています。

10.4.4 イベントリスナーの設定

HTML 文書のオブジェクトと要素のオブジェクトは、`addEventListener` というメソッドを持っています。このメソッドを使うことによって、HTML 文書のオブジェクトまたは要素のオブジェクトに、イベントリスナーを設定することができます。

`addEventListener` は、3 個の引数を受け取ります。

1 個目の引数は、イベントの種類を指定する、「イベント名」(event name) と呼ばれる文字列です。イベント名には、次のようなものがあります。

`click`	マウスでクリックされた。
`mousedown`	マウスのボタンが押された。
`mouseup`	マウスのボタンが離された。
`mouseover`	マウスポインターが重なった。
`mouseout`	マウスポインターが外へ出た。
`mousemove`	マウスが移動した。
`keydown`	キーボードのキーが押された。
`keyup`	キーボードのキーが離された。

2 個目の引数は、イベントが発生したときに呼び出されるイベントハンドラーです。

3 個目の引数は、イベントを伝達する方向を示す真偽値です。真偽値のそれぞれは、次のような意味に解釈されます。

`true` 最上位の要素からイベントが発生した要素へ。

`false` イベントが発生した要素から最上位の要素へ。

10.4.5 マウスによるイベント

それでは、マウスによるイベントを処理する HTML 文書を、DOM を使って書いてみましょう。

マウスによるイベントが発生したときのマウスポインターの座標は、イベントオブジェクトが持っている、次のプロパティーに設定されます。

`clientX` ブラウザーの表示領域での x 座標。

`clientY` ブラウザーの表示領域での y 座標。

`screenX` デスクトップでの x 座標。

`screenY` デスクトップでの y 座標。

次の HTML 文書は、ブラウザーの表示領域の上でマウスが動くと、そのときのマウスポインターの座標を表示します。

HTML 文書の例 `location.html`

```
<!DOCTYPE html>
<html>
  <head><title>location</title></head>
  <body>
    <p id="screen"></p>
    <p id="client"></p>
    <script>
      var screen = document.getElementById("screen");
      var client = document.getElementById("client");
      document.addEventListener("mousemove", function(event) {
        screen.innerHTML ="screen: (" +
          event.screenX + ", " + event.screenY + ")";
        client.innerHTML = "client: (" +
          event.clientX + ", " + event.clientY + ")";
      }, false);
    </script>
  </body>
</html>
```

10.4.6 キーボードによるイベント

次に、キーボードによるイベントを処理する HTML 文書を、DOM を使って書いてみましょう。

イベントオブジェクトは、`keyCode` というプロパティーを持っています。このプロパティーには、キーボードのイベントを発生させたキーを識別する番号が設定されます。

次の HTML 文書は、キーボードのキーが押されると、そのキーを識別する番号を表示します。

HTML 文書の例 `key.html`

```
<!DOCTYPE html>
<html>
  <head><title>key</title></head>
  <body>
    <p id="key"></p>
    <script>
      var key = document.getElementById("key");
      document.addEventListener("keydown", function(event) {
        key.innerHTML = "keyCode: " + event.keyCode;
      }, false);
    </script>
  </body>
</html>
```

10.4.7 イベントが発生した要素

イベントオブジェクトは、`target` というプロパティーを持っています。このプロパティーには、イベントが発生した要素のオブジェクトが設定されます。

次の HTML 文書は、段落がクリックされると、その段落をダイアログボックスで表示します。

HTML 文書の例 target.html

```
<!DOCTYPE html>
<html>
  <head><title>target</title></head>
  <body>
    <p>私は第一の段落です。</p>
    <p>私は第二の段落です。</p>
    <p>私は第三の段落です。</p>
    <script>
      document.addEventListener("click", function(event) {
        alert(event.target.innerHTML);
      }, false);
    </script>
  </body>
</html>
```

10.5 CSS

10.5.1 CSS の基礎

文書をどのようなスタイルで表示してほしいのかということを記述した文書は、「スタイルシート」(style sheet) と呼ばれます。

ウェブページについても、スタイルシートを書くことによって、どのように表示してほしいのかということをブラウザーに対して細かく指定することができます。

ウェブページのスタイルシートは、通常、「CSS」と呼ばれる言語によって記述されます（CSS は、Cascading Style Sheets の略称です）。

CSS で書かれたスタイルシートは、HTML 文書とは別のファイルに格納しておくこともできますし、HTML 文書の中に埋め込むこともできます。

CSS で書かれたスタイルシートが格納されるファイルの拡張子としては、.css が使われます。

10.5.2 link 要素

HTML 文書とは別のファイルに格納されているスタイルシートを HTML 文書に適用したい場合は、HTML 文書の head 要素の中に link という要素型の空要素を書いて、その要素が持っている rel という属性に stylesheet という属性値を設定して、href という属性に、スタイルシートが格納されているファイルの URI を設定します。

次のスタイルシートは、ページ全体の背景色をオレンジ色にします。

スタイルシートの例 orange.css

```
body { background-color: orange; }
```

次の HTML 文書には、orange.css に格納されているスタイルシートが適用されますので、背景色がオレンジ色のページが表示されます。

HTML 文書の例 link.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>link</title>
    <link rel="stylesheet" href="orange.css">
  </head>
  <body>
    <p>私は段落です。</p>
  </body>
</html>
```

10.5.3 style 要素

HTML 文書に適用するスタイルシートをその HTML 文書の中に埋め込みたい場合は、HTML 文書の head 要素の中に style という要素型の要素を書いて、その要素の中にスタイルシートを書きます。

次の HTML 文書は、段落を青色で表示します。

HTML 文書の例 style.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>style</title>
    <style>
      p { color: blue; }
    </style>
  </head>
  <body>
    <p>私は段落です。</p>
  </body>
</html>
```

10.5.4 ルール

CSS のスタイルシートは、「ルール」(rule) と呼ばれる記述から構成されます。たとえば、

```
p { color: blue; }
```

というスタイルシートは、ひとつのルールから構成されています。

ルールは、セレクターと宣言ブロックから構成されます。「セレクター」(selector) というのは、スタイルを適用する対象の記述のことです。たとえば、セレクターとして要素型名を書くことによって、その要素型を持つすべての要素に対して適用されるスタイルを記述することができます。

「宣言ブロック」(declaration block) は、「宣言」(declaration) と呼ばれるものをいくつか並べて、その全体を中括弧で囲んだものです。

宣言は、プロパティーと値から構成されます。たとえば、

```
color: blue;
```

という宣言は、color というプロパティーと blue という値から構成されています。

「プロパティー」(property) というのは、スタイルの種類をあらわしている名前のことです。たとえば、color というのは文字の色というスタイルをあらわしているプロパティーです。同じように、背景色というスタイルは、background-color というプロパティーによってあらわされます。

「値」(value) というのは、プロパティーで指定された種類のスタイルをどうするのかという具体的な記述のことです。

10.5.5 フォントの大きさ

文字を表示するフォントの大きさというスタイルは、font-size というプロパティーによってあらわされます。

フォントの大きさを指定する値は、10 進数であらわされる長さと、長さの単位から構成されます。たとえば、24px という値を書くことによって、24 ピクセルという大きさを指定することができます。

次の HTML 文書は、80 ピクセルの大きさのフォントを使って段落を表示します。

HTML 文書の例 fontsize.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>font-size</title>
    <style>
      p { font-size: 80px; }
    </style>
  </head>
  <body>
    <p>私は段落です。</p>
  </body>
</html>
```

10.5.6 DOM によるスタイルの設定

CSS のプロパティーに対しては、DOM を使って値を設定することも可能です。

要素のオブジェクトは、style というプロパティーを持っていて、このプロパティーには、CSS のプロパティーに対応するプロパティーを持っているオブジェクトが設定されています。

ただし、JavaScript は、ハイフンを含んでいる文字列を名前として使うことができませんので、DOM のプロパティーの名前と、それに対応する CSS のプロパティーとは、必ずしも一致しているわけではありません。DOM のプロパティーの名前は、CSS のプロパティーを次の規則で変換したものです。

(1) 単語と単語のあいだのハイフンを取り除く。
(2) 2 番目以降の単語の先頭を大文字にする。

たとえば、font-size という CSS のプロパティーには、fontSize という名前の DOM のプロパティーが対応しています。

次の HTML 文書は、ボタンをクリックすることによって、段落を表示するフォントの大きさを変更することができます。

HTML 文書の例 domstyle.html

```
<!DOCTYPE html>
<html>
  <head><title>DOM によるスタイルの設定</title></head>
  <body>
    <form>
      <input type="button" value="拡大" onclick="magnify(1.2)">
      <input type="button" value="縮小" onclick="magnify(0.84)">
    </form>
    <p id="paragraph1">私は段落です。</p>
    <script>
      var style = document.getElementById("paragraph1").style;
      var size = 20;
      style.fontSize = size + "px";

      function magnify(factor) {
        size *= factor;
        style.fontSize = size + "px";
      }
    </script>
  </body>
</html>
```

第 11 章 Canvas

11.1 Canvas の基礎

11.1.1 Canvas とは何か

JavaScript のプログラムは、HTML 文書によって作られた長方形の領域の上にグラフィックスを描画することができます。

JavaScript のプログラムがその上にグラフィックスを描画することができる長方形の領域は、「Canvas」と呼ばれます。

11.1.2 canvas 要素

Canvas は、canvas という要素型の要素によって作られます。

Canvas の大きさは、次の属性を使うことによって指定することができます。

width Canvas の横の長さ。単位はピクセル。デフォルト値は 300。
height Canvas の縦の長さ。単位はピクセル。デフォルト値は 150。

たとえば、次のような canvas 要素を書くことによって、canvas1 という ID を持つ、横の長さが 400 ピクセルで縦の長さが 300 ピクセルの Canvas を作ることができます。

```
<canvas id="canvas1" width="400" height="300"></canvas>
```

11.1.3 描画コンテキスト

canvas 要素の要素オブジェクトは、「描画コンテキスト」(rendering context) と呼ばれるオブジェクトを持っています。Canvas の上にグラフィックスを描画するために必要となる各種の操作は、描画コンテキストが持っているメソッドを呼び出すことによって実行することができます。

canvas 要素の要素オブジェクトから描画コンテキストを取得したいときは、getContext というメソッドを呼び出します。このメソッドには、「コンテキスト ID」(context ID) と呼ばれる文字列を引数として渡す必要があります。コンテキスト ID として 2d という文字列を渡すと、getContext は、Canvas の上に 2 次元のグラフィックスを描画するための描画コンテキストを戻り値として返します。

たとえば、次のような文を書くことによって、canvas1 という名前を持つ canvas 要素の要素オブジェクトを取得して、その要素オブジェクトから、Canvas の上に 2 次元のグラフィックスを描画するための描画コンテキストを取得することができます。

```
var context = document.getElementById("canvas1")
                      .getContext("2d");
```

11.1.4 Canvas の座標系

Canvas の上での位置は、Canvas が持っている座標系を使うことによって指定することができます。

Canvas の座標系は、x 軸と y 軸から構成されます。原点は Canvas の左上の隅にあって、x 軸は右向き、y 軸は下向きです。距離の単位はピクセルです。

11.1.5 長方形を描画するメソッド

描画コンテキストは、長方形を描画する次のようなメソッドを持っています。

strokeRect	長方形を線で描画する。
fillRect	長方形の領域を塗りつぶす。
clearRect	長方形の領域に描画されたグラフィックスを消去する。

これらの三つのメソッドは、すべて、長方形の位置と大きさを指定する、次の四つの引数を受け取ります。

1 個目　左上の頂点の x 座標。
2 個目　左上の頂点の y 座標。
3 個目　横の長さ。
4 個目　縦の長さ。

strokeRect によって描画される線の太さは、描画コンテキストが持っている、lineWidth というプロパティーに設定されている数値によって決定されます。太さの単位はピクセルで、デフォルトは 1 です。

次の HTML 文書は、strokeRect、fillRect、clearRect のそれぞれを使ってキャンバスの上に長方形を描画します。

HTML 文書の例　rect.html

```
<!DOCTYPE html>
<html>
  <head><title>rect</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                            .getContext("2d");
      context.lineWidth = 3;
      context.strokeRect(10, 10, 250, 150);
      context.fillRect(60, 50, 250, 150);
      context.clearRect(110, 90, 250, 150);
    </script>
  </body>
</html>
```

11.2 描画状態

11.2.1 描画状態の基礎

描画コンテキストは、グラフィックスの描画に関連する状態が設定されているさまざまなプロパティーを持っています。たとえば、第 11.1 節で説明したように、描画コンテキストが持っている `lineWidth` というプロパティーには、線の太さという状態が設定されています。

描画コンテキストが保持している状態の集合は、「描画状態」(drawing state) と呼ばれます。

11.2.2 色のプロパティー

描画するグラフィックスの色は、次のプロパティーに設定されています。

`strokeStyle` 線の色。デフォルトは黒色。

`fillStyle` 塗りつぶしの色。デフォルトは黒色。

色を記述した文字列をこれらのプロパティーに設定すると、グラフィックスは、その色で描画されることになります。色は、次の 3 種類の方法によって記述されます。

- 色名を使う方法
- 16 進数を使う方法
- 10 進数を使う方法

11.2.3 色名

色を識別するための英語の単語は、「色名」(color keyword) と呼ばれます。たとえば、`yellow` は、黄色を意味する色名です。

HTML 文書の例 `keyword.html`

```
<!DOCTYPE html>
<html>
  <head><title>keyword</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                            .getContext("2d");
      context.lineWidth = 6;
      context.fillStyle = "orange";
      context.fillRect(10, 10, 250, 150);
      context.fillStyle = "yellow";
      context.fillRect(75, 75, 250, 150);
      context.strokeStyle = "orangered";
      context.strokeRect(75, 75, 250, 150);
      context.strokeStyle = "darkorange";
      context.strokeRect(140, 140, 250, 150);
    </script>
  </body>
</html>
```

11.2.4 16 進数による色の記述

16 進数で色を記述したいときは、

`#`*rrggbb*

という形の文字列を書きます。*rrggbb* というところには、光の三原色（赤と緑と青）のそれぞれの色の強さをあらわす 2 桁の 16 進数を並べて書きます。*rr* が赤で、*gg* が緑で、*bb* が青です。たとえば、水色を 16 進数で記述すると、

```
#00ffff
```

という文字列になります。

HTML 文書の例 `hexadecimal.html`

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

```
  <head><title>hexadecimal</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                            .getContext("2d");
      context.lineWidth = 6;
      context.fillStyle = "#00c0ff";
      context.fillRect(10, 10, 250, 150);
      context.fillStyle = "#00ffff";
      context.fillRect(75, 75, 250, 150);
      context.strokeStyle = "#0000ff";
      context.strokeRect(75, 75, 250, 150);
      context.strokeStyle = "#0080c0";
      context.strokeRect(140, 140, 250, 150);
    </script>
  </body>
</html>
```

11.2.5 10 進数による色の記述

10 進数で色を記述したいときは、

```
rgb(赤, 緑, 青)
```

という形で、光の三原色のそれぞれの色の強さをあらわす、0 から 255 までのあいだの 10 進数を並べて書きます。たとえば、赤紫を 10 進数で記述すると、

```
rgb(255, 0, 255)
```

という文字列になります。

HTML 文書の例 `decimal.html`

```
<!DOCTYPE html>
<html>
  <head><title>decimal</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                            .getContext("2d");
      context.lineWidth = 6;
      context.fillStyle = "rgb(128, 192, 0)";
      context.fillRect(10, 10, 250, 150);
      context.fillStyle = "rgb(192, 255, 0)";
      context.fillRect(75, 75, 250, 150);
      context.strokeStyle = "rgb(0, 192, 0)";
      context.strokeRect(75, 75, 250, 150);
      context.strokeStyle = "rgb(0, 128, 0)";
      context.strokeRect(140, 140, 250, 150);
    </script>
  </body>
</html>
```

11.2.6 描画状態の保存と復元

描画コンテキストが持っている現在の描画状態は、「カレント描画状態」(current drawing state) と呼ばれます。

描画コンテキストは、カレント描画状態に変更を加えてグラフィックスを描画したのち、変更を加える以前の状態に戻ることができるように、次のようなメソッドを持っています。

`save` カレント描画状態のバックアップを作って保存します。

`restore` 保存されているバックアップを使ってカレント描画状態を復元します。

ですから、カレント描画状態に変更を加える関数を定義するときは、最初にカレント描画状態のバックアップを保存しておいて、最後にそのバックアップを使ってカレント描画状態を復元するように書いておくと、その関数を呼び出す前と呼び出したあとのカレント描画状態がまったく

同じになりますので、とても使いやすい関数になります。

次の HTML 文書の中で定義されている fillRedRect という関数は、塗りつぶしの色を変更して長方形を描画しますが、その影響は、関数の外部には波及しません。

HTML 文書の例 save.html

```
<!DOCTYPE html>
<html>
  <head><title>save</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                            .getContext("2d");
      context.fillStyle = "navy";
      context.fillRect(10, 10, 250, 150);
      fillRedRect(context, 75, 75, 250, 150);
      context.fillRect(140, 140, 250, 150);

      function fillRedRect(context, x, y, width, height) {
        context.save();
        context.fillStyle = "red";
        context.fillRect(x, y, width, height);
        context.restore();
      }
    </script>
  </body>
</html>
```

11.3 パス

11.3.1 パスの基礎

描画コンテキストは長方形を描画するメソッドを持っていますので、ただ単にそのメソッドを呼び出すだけで、長方形を描画することができるわけですが、長方形以外の形状は、ひとつのメソッドを呼び出すだけでは描画できません。

長方形以外の形状を描画するためには、いくつかの線を組み合わせることによって形状を構築する、ということが必要になります。いくつかの線を組み合わせることによって作られた形状は、「パス」(path) と呼ばれます。

一つのパスは、0 個または 1 個以上のサブパスから構成されます。「サブパス」(subpath) というのは、連続するいくつかの線を組み合わせることによって作られた形状のことです。パスを構築するというのは、1 個以上のサブパスを構築するということです。

サブパスに線を追加する場合には、「カレントポイント」(current point) と呼ばれる点が、その線の起点になります。カレントポイントというのは、その言葉のとおり、「現在の点」という意味です。サブパスに線を追加すると、カレントポイントはその線の終点へ移動します。

11.3.2 カレントパス

描画コンテキストは、1 個のパスを保持することができます。描画コンテキストが保持しているパスは、「カレントパス」(current path) と呼ばれます。

描画コンテキストが持っている次のメソッドは、カレントパスを描画します。

`stroke` カレントパスを線で描画する。

`fill` カレントパスで囲まれた領域を塗りつぶす。

11.3.3 カレントパスの構築

カレントパスは、次のような手順で構築されます。

(1) カレントパスをリセットする。

(2) 空の新しいサブパスを生成する。

(3) サブパスに線を追加する。

サブパスの生成とサブパスへの線の追加は、何回でも繰り返すことができます。また、ひとつのサブパスに対する線の追加も、何回でも繰り返すことができます。

11.3.4 カレントパスのリセット

カレントパスをリセットしたいときは、描画コンテキストが持っている beginPath というメソッドを呼び出します。たとえば、context が描画コンテキストだとするならば、

```
context.beginPath();
```

と書くことによって、カレントパスをリセットすることができます。

11.3.5 サブパスの生成

空の新しいサブパスを生成したいときは、描画コンテキストが持っている moveTo というメソッドを呼び出します。

moveTo は、x 座標と y 座標を引数として受け取って、空の新しいサブパスを生成して、引数として受け取った座標をカレントポイントとして設定します。たとえば、context が描画コンテキストだとするならば、

```
context.moveTo(70, 30);
```

と書くことによって、空の新しいサブパスを生成して、$(70, 30)$ という座標をカレントポイントとして設定することができます。

11.3.6 直線の追加

サブパスに直線を追加したいときは、描画コンテキストが持っている lineTo というメソッドを呼び出します。

lineTo は、終点の x 座標と y 座標を引数として受け取って、カレントポイントと終点とをつなぐ直線をサブパスに追加して、カレントポイントを終点へ移動させます。たとえば、context が描画コンテキストだとするならば、

```
context.lineTo(200, 150);
```

と書くことによって、カレントポイントと $(200, 150)$ とをつなぐ直線をサブパスに追加して、カレントポイントを $(200, 150)$ へ移動させることができます。

HTML 文書の例 path.html

```
<!DOCTYPE html>
<html>
  <head><title>path</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                            .getContext("2d");
      context.lineWidth = 10;
      context.strokeStyle = "crimson";
      context.beginPath();
      context.moveTo(150, 80);
      context.lineTo(50, 150);
      context.lineTo(150, 220);
      context.moveTo(250, 80);
      context.lineTo(350, 150);
      context.lineTo(250, 220);
      context.stroke();
    </script>
  </body>
</html>
```

11.3.7 開いたサブパスと閉じたサブパス

サブパスには、開いたものと閉じたものとがあります。開いたサブパスというのは、始まりの点と終わりの点があるサブパスのことで、閉じたサブパスというのは、始まりの点も終わりの点もなくて、その上をぐるりと一周することのできるサブパスのことです。

線を追加しただけのサブパスは、開いたサブパスです。閉じたサブパスを作るためには、描画コンテキストが持っている closePath というメソッドを呼び出す必要があります。このメソッドは、カレントポイントと、現在のサブパスの最初の点とを直線でつないだのち、サブパスを閉じます。

HTML 文書の例 closepath.html

```
<!DOCTYPE html>
<html>
  <head><title>closePath</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                            .getContext("2d");
      context.lineWidth = 10;
      context.strokeStyle = "indigo";
      context.beginPath();
      context.moveTo(150, 80);
      context.lineTo(50, 150);
      context.lineTo(150, 220);
      context.moveTo(250, 80);
      context.lineTo(350, 150);
      context.lineTo(250, 220);
      context.closePath();
      context.stroke();
    </script>
  </body>
</html>
```

11.3.8 円弧の追加

描画コンテキストは、arc というメソッドを持っています。このメソッドを呼び出すことによって、サブパスに円弧を追加することができます。

arc は、次の 6 個の引数を受け取ります。

1 個目 中心の x 座標。

2 個目 中心の y 座標。

3 個目 半径。

4 個目 開始角度。右方向が 0 で、時計回りで大きくなる。単位はラジアン。

5 個目 終了角度。右方向が 0 で、時計回りで大きくなる。単位はラジアン。

6 個目 回転の方向を意味する真偽値。真は反時計回りで、偽は時計回り。真を指定した場合も、開始角度と終了角度が大きくなる方向は時計回りのまま。

arc は、カレントパスが空の場合はサブパスを生成します。サブパスが存在している場合は、カレントポイントと、円弧が開始する点とをつなぐ直線をサブパスに追加したのち、円弧をサブパスに追加します。そして、どちらの場合も、円弧が終了する点をカレントポイントとして設定します。

HTML 文書の例 arc.html

```
<!DOCTYPE html>
<html>
  <head><title>arc</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                            .getContext("2d");
      context.lineWidth = 10;
      context.strokeStyle = "darkgreen";
      context.beginPath();
      context.arc(100, 150, 60, radian(90), radian(0), false);
      context.arc(300, 150, 60, radian(180), radian(270), true);
      context.stroke();
```

```
      function radian(degree) {
          return Math.PI/180.0*degree;
      }
    </script>
  </body>
</html>
```

11.4 テキスト

11.4.1 テキストを描画するメソッド

描画コンテキストは、テキスト（文字列）を描画する、fillText というメソッドを持っています。

fillText は、次の 3 個の引数を受け取ります。

1 個目　描画するテキスト（文字列）。

2 個目　テキストを描画する位置の x 座標。

3 個目　テキストを描画する位置の y 座標。

テキストを描画する色は、fillStyle に設定されているものが使われます。

HTML 文書の例　filltext.html

```
<!DOCTYPE html>
<html>
  <head><title>fillText</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                            .getContext("2d");
      context.fillStyle = "mediumblue";
      context.fillText("私はテキストです。", 100, 150);
    </script>
  </body>
</html>
```

11.4.2 フォント

描画コンテキストは、font というプロパティーを持っています。フォントをあらわす文字列をこのプロパティーに設定しておくと、テキストを描画するときに、そのフォントが使われます。

フォントをあらわす文字列は、次のものを空白で区切って並べることによって作ることができます。

フォントの太さ	bold を指定すると太字になる。省略すると標準の太さ。
フォントのスタイル	italic を指定するとイタリック体、oblique を指定するとオブリーク体になる。省略すると標準のスタイル。
フォントの大きさ	24px のように、数値と単位でフォントの大きさを指定する。
フォント名	フォントの名前か、または「総称フォントファミリー名」(generic font family name) と呼ばれる次の名前を指定する。 serif　sans-serif　cursive　fantasy　monospace

HTML 文書の例　font.html

```
<!DOCTYPE html>
<html>
  <head><title>font</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                            .getContext("2d");
```

```
      context.fillStyle = "deeppink";
      fillFontText(context, "30px serif", 80);
      fillFontText(context, "30px monospace", 120);
      fillFontText(context, "italic 30px serif", 160);
      fillFontText(context, "bold 30px serif", 200);
      fillFontText(context, "bold italic 30px serif", 240);

      function fillFontText(context, font, y) {
          context.font = font;
          context.fillText(font, 50, y);
      }
    </script>
  </body>
</html>
```

11.4.3 テキストの配置

テキストを描画するために指定した位置と、実際に描画されるテキストとの水平方向の位置関係は、そのテキストの「配置」(alignment) と呼ばれます。

テキストの配置は、描画コンテキストが持っている `textAlign` というプロパティーによって決定されます。

`textAlign` には、テキストの配置をあらわす、次の文字列のうちのいずれかを設定することができます。

`start`	指定された位置をテキストの先頭にする。デフォルトはこの配置。
`end`	指定された位置をテキストの末尾にする。
`left`	左寄せ。
`right`	右寄せ。
`center`	センタリング。

書字方向（文字を並べる方向）が左から右の場合、 `start` は `left` と同じ意味になって、 `end` は `right` と同じ意味になります。

HTML 文書の例 `textalign.html`

```
<!DOCTYPE html>
<html>
  <head><title>textAlign</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                            .getContext("2d");
      context.fillStyle = "brown";
      context.font = "40px serif";
      strokeLine(context, 200, 10, 200, 290);
      fillAlignText(context, "start", 80);
      fillAlignText(context, "end", 120);
      fillAlignText(context, "left", 160);
      fillAlignText(context, "right", 200);
      fillAlignText(context, "center", 240);

      function fillAlignText(context, align, y) {
        context.save();
        context.textAlign = align;
        context.fillText(align, 200, y);
        context.restore();
      }

      function strokeLine(context, x1, y1, x2, y2) {
        context.save();
        context.lineWidth = 2;
        context.strokeStyle = "midnightblue";
        context.beginPath();
        context.moveTo(x1, y1);
```

```
        context.lineTo(x2, y2);
        context.stroke();
        context.restore();
      }
    </script>
  </body>
</html>
```

11.5 座標系の変換

11.5.1 座標系の変換の基礎

描画状態を構成している状態のひとつに、「変換行列」(transformation matrix) と呼ばれる行列があります。この行列に変更を加えることによって、座標系に対して、移動、拡大、回転などの変換を加えることができます。グラフィックスは、変換行列によって変換された座標系を使って Canvas の上に描画されます。

デフォルトの描画状態が変換行列として保持しているのは、変換を何もしない行列です。

11.5.2 座標系の移動

座標系を移動させたいときは、描画コンテキストが持っている `translate` というメソッドを使います。このメソッドは、x 軸方向と y 軸方向の移動量を引数として受け取ります。

HTML 文書の例 `translate.html`

```
<!DOCTYPE html>
<html>
  <head><title>translate</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                            .getContext("2d");
      circle(context, "maroon");
      context.translate(400, 0);
      circle(context, "darkcyan");
      context.translate(0, 300);
      circle(context, "olivedrab");
      context.translate(-400, 0);
      circle(context, "limegreen");

      function circle(context, color) {
        context.save();
        context.fillStyle = color;
        context.beginPath();
        context.arc(0, 0, 100, 0, Math.PI*2, false);
        context.fill();
        context.restore();
      }
    </script>
  </body>
</html>
```

11.5.3 座標系の拡大

座標系を拡大したいときは、描画コンテキストが持っている `scale` というメソッドを使います。このメソッドは、x 軸方向と y 軸方向の拡大率を引数として受け取ります。

HTML 文書の例 `scale.html`

```
<!DOCTYPE html>
<html>
  <head><title>scale</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
```

```
                          .getContext("2d");
      context.font = "30px serif";
      fillScaleText(context, "(1, 1)", 140, 120, 1, 1);
      fillScaleText(context, "(2, 1)", 220, 120, 2, 1);
      fillScaleText(context, "(1, 3)", 140, 220, 1, 3);
      fillScaleText(context, "(-1, 1)", 120, 120, -1, 1);
      fillScaleText(context, "(1, -1)", 140, 50, 1, -1);

      function fillScaleText(context, text, x, y, sx, sy) {
        context.save();
        context.translate(x, y);
        context.scale(sx, sy);
        context.fillText(text, 0, 0);
        context.restore();
      }
    </script>
  </body>
</html>
```

11.5.4　座標系の回転

座標系を回転させたいときは、描画コンテキストが持っている rotate というメソッドを使います。このメソッドは、回転の角度（単位はラジアン）を引数として受け取ります。回転の中心は原点で、回転のプラスの方向は時計回りです。

HTML 文書の例　rotate.html

```
<!DOCTYPE html>
<html>
  <head><title>rotate</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                          .getContext("2d");
      context.lineWidth = 10;
      context.translate(100, 150);
      strokeRotateLine(context, 0, "seagreen");
      strokeRotateLine(context, radian(30), "tomato");
      strokeRotateLine(context, radian(-30), "deepskyblue");

      function strokeRotateLine(context, theta, color) {
        context.save();
        context.strokeStyle = color;
        context.rotate(theta);
        context.beginPath();
        context.moveTo(0, 0);
        context.lineTo(200, 0);
        context.stroke();
        context.restore();
      }

      function radian(degree) {
        return Math.PI/180.0*degree;
      }
    </script>
  </body>
</html>
```

第 12 章　アニメーション

12.1 アニメーションの基礎

12.1.1 アニメーションとは何か

時間の経過とともに変化するグラフィックスは、「アニメーション」(animation) と呼ばれます。この章では、Canvas を使ってアニメーションを作る方法について説明したいと思います。

12.1.2 タイマー

アニメーションを作るためには、一定の時間ごとに何らかの動作を実行する仕組みが必要です。JavaScript では、「タイマー」(timer) と呼ばれる仕組みを使うことによって、一定の時間ごとに何らかの動作を実行することができます。

タイマーを使うためには、それを設定する必要があります。タイマーを設定したいときは、`setInterval` という関数を使います。

`setInterval` は、次の 2 個の引数を受け取ります。

1 個目　関数。

2 個目　1 個目の引数として受け取った関数を実行する間隔。単位はミリ秒。

次の HTML 文書は、段落として表示された数値を 100 ミリ秒ごとにカウントアップさせます。

HTML 文書の例　`timer.html`

```
<!DOCTYPE html>
<html>
  <head><title>timer</title></head>
  <body>
    <p id="number">0</p>
    <script>
      var count = 0;
      var number = document.getElementById("number");
      setInterval(function() {
        count++;
        number.innerHTML = count;
      }, 100);
    </script>
  </body>
</html>
```

12.1.3 タイマーの設定の解除

タイマーの設定は、解除することも可能です。ただし、タイマーの設定を解除するためには、そのタイマーを識別する ID が必要になります。

`setInterval` は、自分が設定したタイマーを識別するための ID を戻り値として返します。ですから、その戻り値を変数に設定しておけば、それを使ってその設定を解除することができます。

タイマーの設定を解除したいときは、`clearInterval` というメソッドを呼び出します。このメソッドは、タイマーを識別する ID を引数として受け取って、その ID によって識別されるタイマーの設定を解除します。

次の HTML 文書は、ボタンをクリックすることによって、数字の増加を開始させたり停止させたりすることができます。

HTML 文書の例　`clear.html`

```
<!DOCTYPE html>
<html>
  <head><title>clear</title></head>
  <body>
    <p id="number">0</p>
    <form>
      <input type="button" value="開始" onclick="start()">
      <input type="button" value="停止" onclick="stop()">
    </form>
    <script>
      var count = 0;
      var id = -1;
      var number = document.getElementById("number");
```

```
      function start() {
        if (id == -1) {
          number.innerHTML = 0;
          id = setInterval(function() {
            count++;
            number.innerHTML = count;
          }, 100);
          count = 0;
        }
      }

      function stop() {
        if (id != -1) {
          clearInterval(id);
          id = -1;
        }
      }
    </script>
  </body>
</html>
```

12.1.4　定期的な描画

グラフィックスが移動するアニメーションは、タイマーを使って、位置を移動させながらグラフィックスを定期的に描画することによって作ることができます。

次の HTML 文書は、位置を右へ移動させながら、タイマーを使って正方形を定期的に描画します。

HTML 文書の例　afterimage.html

```
<!DOCTYPE html>
<html>
  <head><title>afterimage</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                            .getContext("2d");
      context.fillStyle = "deeppink";
      var x = 0;
      setInterval(function() {
        x++;
        context.fillRect(x, 140, 20, 20);
      }, 10);
    </script>
  </body>
</html>
```

12.1.5　残像の消去

アニメーションを作るためには、多くの場合、ただ単にグラフィックスを定期的に描画するだけではなくて、残像となったグラフィックスを消去することも必要になります。

残像となったグラフィックスは、clearRect を使うことによって消去することができます。

次の HTML 文書は、先ほどの HTML 文書と同じように、位置を右へ移動させながら、タイマーを使って正方形を定期的に描画します。ただし、先ほどとは違って、描画する前に、残像となった正方形を消去します。

HTML 文書の例　animation.html

```
<!DOCTYPE html>
<html>
  <head><title>animation</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
```

```
                          .getContext("2d");
      context.fillStyle = "limegreen";
      var x = 0;
      setInterval(function() {
        x++;
        context.clearRect(0, 0, 400, 300);
        context.fillRect(x, 140, 20, 20);
      }, 10);
    </script>
  </body>
</html>
```

12.2 座標系の変換によるアニメーション

12.2.1 座標系の移動によるアニメーション

第 11.5 節で紹介した座標系の変換を利用して、アニメーションを作ってみましょう。

まず最初は、座標系の移動によるアニメーションです。グラフィックスが移動するアニメーションは、グラフィックスを描画する位置を変化させることによって作ることも可能ですが、複雑なグラフィックスの場合は、座標系の移動を使うほうが、簡単にアニメーションを作ることができます。

次の HTML 文書は、十字の形のグラフィックスが移動するアニメーションを、座標系の移動を使って作っています。

HTML 文書の例 anitran.html

```
<!DOCTYPE html>
<html>
  <head><title>animation by translation</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                          .getContext("2d");
      context.lineWidth = 6;
      context.strokeStyle = "dodgerblue";
      var x = 0;
      var y = 0;
      setInterval(function() {
        x++;
        y += 0.75;
        context.clearRect(0, 0, 400, 300);
        cross(context, x, y);
      }, 10);

      function cross(context, x, y) {
        context.save();
        context.translate(x, y);
        context.beginPath();
        context.moveTo(-20, 0);
        context.lineTo(20, 0);
        context.moveTo(0, -20);
        context.lineTo(0, 20);
        context.stroke();
        context.restore();
      }
    </script>
  </body>
</html>
```

12.2.2 座標系の拡大によるアニメーション

次は、座標系の拡大によるアニメーションです。

次の HTML 文書は、テキストが拡大するアニメーションを、座標系の拡大を使って作っています。

HTML 文書の例　anisca.html

```
<!DOCTYPE html>
<html>
  <head><title>animation by scaling</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                            .getContext("2d");
      context.fillStyle = "crimson";
      context.font = "10px serif";
      var s = 1;
      setInterval(function() {
        s *= 1.01;
        context.clearRect(0, 0, 400, 300);
        fillScaleText(context, "scale", 10, 280, s);
      }, 10);

      function fillScaleText(context, text, x, y, s) {
        context.save();
        context.translate(x, y);
        context.scale(s, s);
        context.fillText(text, 0, 0);
        context.restore();
      }
    </script>
  </body>
</html>
```

12.2.3　座標系の回転によるアニメーション

次は、座標系の回転によるアニメーションです。

次の HTML 文書は、直線が回転するアニメーションを、座標系の回転を使って作っています。

HTML 文書の例　anirota.html

```
<!DOCTYPE html>
<html>
  <head><title>animation by rotation</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                            .getContext("2d");
      context.lineWidth = 10;
      context.strokeStyle = "olivedrab";
      var theta = 0;
      setInterval(function() {
        theta += 0.01;
        context.clearRect(0, 0, 400, 300);
        strokeRoteteLine(context, 200, 150, theta);
      }, 10);

      function strokeRoteteLine(context, x, y, theta) {
        context.save();
        context.translate(x, y);
        context.rotate(theta);
        context.beginPath();
        context.moveTo(0, 20);
        context.lineTo(0, -140);
        context.stroke();
        context.restore();
      }
    </script>
  </body>
</html>
```

12.3 インタラクティブなアニメーション

12.3.1 マウスポインタの座標

この節では、インタラクティブなアニメーション、つまり、ユーザーが操作することのできるアニメーションを作ってみたいと思います。

マウスの操作によるインタラクティブなアニメーションを作る場合、Canvas の座標系でのマウスポインタの座標が必要になります。ところが、イベントオブジェクトが持っているマウスポインタの座標は、ブラウザーの表示領域の座標系でのものと、デスクトップの座標系でのものだけで、Canvas の座標系でのものは持っていません。

そこで登場するのが、`getBoundingClientRect` というメソッドです。これは、要素のオブジェクトが持っているメソッドで、戻り値として、要素を取り囲む長方形の位置と大きさを、ブラウザーの表示領域の座標系であらわしているオブジェクトを返します。このオブジェクトは、次のプロパティーを持っています。

`left`	長方形の左端の x 座標。
`right`	長方形の右端の x 座標。
`top`	長方形の上端の y 座標。
`bottom`	長方形の下端の y 座標。

`getBoundingClientRect` を呼び出すことによって、ブラウザーの表示領域の座標系での `canvas` 要素の座標が分かりますので、それを使うことによって、Canvas の座標系でのマウスポインターの座標を求めることができます。

次の HTML 文書は、Canvas をマウスでクリックしていくことによって、折れ線を描画することができます。

HTML 文書の例 `bounding.html`

```
<!DOCTYPE html>
<html>
  <head><title>bounding</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var canvas = document.getElementById("canvas1")
      var context = canvas.getContext("2d");
      var rect = canvas.getBoundingClientRect();
      var lastX = 0;
      var lastY = 0;
      document.addEventListener("click", function(event) {
        line(context, event);
      }, false);
      context.lineWidth = 3;
      context.strokeStyle = "darkgreen";

      function line(context, event) {
        var x = event.clientX - rect.left;
        var y = event.clientY - rect.top;
        context.beginPath();
        context.moveTo(lastX, lastY);
        context.lineTo(x, y);
        context.stroke();
        lastX = x;
        lastY = y;
      }
    </script>
  </body>
</html>
```

12.3.2 波紋のようなもの

それでは、実際にインタラクティブなアニメーションを作ってみましょう。

次の HTML 文書は、マウスで Canvas をクリックすると、その位置を中心にして波紋のようなものが広がっていきます。

HTML 文書の例　ripple.html

```
<!DOCTYPE html>
<html>
  <head><title>ripple</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var canvas = document.getElementById("canvas1")
      var context = canvas.getContext("2d");
      var rect = canvas.getBoundingClientRect();
      document.addEventListener("click", function(event) {
        ripples.push({
          x: event.clientX - rect.left,
          y: event.clientY - rect.top,
          r: 0
        });
      }, false);
      context.lineWidth = 4;
      context.strokeStyle = "lightblue";
      var ripples = [];
      setInterval(function() {
        context.clearRect(0, 0, 400, 300);
        drawRipples();
      }, 20);

      function drawRipples() {
        for (var i = 0; i < ripples.length; i++) {
          circle(context, ripples[i].x, ripples[i].y,
            ripples[i].r);
          ripples[i].r++;
        }
      }

      function circle(context, x, y, r) {
        context.beginPath();
        context.arc(x, y, r, 0, Math.PI*2, false);
        context.stroke();
      }
    </script>
  </body>
</html>
```

12.3.3　キーによる方向転換

次の HTML 文書は、矢印のキーを押すことによって、正方形が移動する方向を変更することができます。

HTML 文書の例　direction.html

```
<!DOCTYPE html>
<html>
  <head><title>direction</title></head>
  <body>
    <canvas id="canvas1" width="400" height="300"></canvas>
    <script>
      var context = document.getElementById("canvas1")
                            .getContext("2d");
      document.addEventListener("keydown", function(event) {
        if (event.keyCode >= 37 && event.keyCode <= 40) {
          direction = event.keyCode - 36;
        }
      }, false);
      context.fillStyle = "orangered";
      var direction = 3; // right
      var x = 0;
      var y = 140;
      setInterval(function() {
```

```
        context.clearRect(0, 0, 400, 300);
        context.fillRect(x, y, 20, 20);
        move();
      }, 10);

      function move() {
        switch (direction) {
          case 1: // left
            x--;
            break;
          case 2: // up
            y--;
            break;
          case 3: // right
            x++;
            break;
          case 4: // down
            y++;
            break;
        }
      }
    </script>
  </body>
</html>
```

第 13 章　ゲーム

13.1　スロットマシン

13.1.1　スロットマシンとは何か

この章では、HTML と JavaScript を使って作られたゲームを紹介したいと思います。

まず最初に紹介するのは、スロットマシンです。

「スロットマシン」(slot machine) という言葉は、もともとはコインを投入することによって作動するギャンブル機械の総称でしたが、現在では、「リールマシン」と呼ばれるギャンブル機械を指して使われることが多いようです。

「リールマシン」(reel machine) というのは、絵柄が描かれた複数個の円筒を回転させて、その回転が停止したときの絵柄の並び方によって配当を決定する、というギャンブル機械のことです。絵柄が描かれたそれぞれの円筒は、「リール」(reel) と呼ばれます。リールマシンを発明したのは、アメリカの Charles Fey(1862–1944) という人です。

13.1.2　スロットマシンの HTML 文書

次の HTML 文書は、スロットマシンを HTML と JavaScript で書いたものです。

HTML 文書の例　slot.html

```
<!DOCTYPE html>
<html>
  <head><title>スロットマシン</title></head>
  <body>
    <canvas id="canvas1" width="400" height="160"></canvas>
    <form>
      <input type="button" value="開始" onclick="start()">
      <input type="button" value="停止" onclick="stop(0)">
      <input type="button" value="停止" onclick="stop(1)">
      <input type="button" value="停止" onclick="stop(2)">
    </form>
    <p id="message"></p>
    <script>
      var message = document.getElementById("message");
      var context = document.getElementById("canvas1")
                              .getContext("2d");
      context.fillStyle = "springgreen";
```

```
  context.font = "200px serif";
  context.textAlign = "center";
  var status = 0;
  var timerid;
  var reel = [new Reel(0), new Reel(1), new Reel(2)];
  drawReel();

  function Reel(reelid) {
    this.id = reelid;
    this.n = 1;
    this.moving = false;
    this.increment = function() {
      if (this.moving) {
        if (this.n <= 8) {
          this.n++;
        } else {
          this.n = 1;
        }
      }
    };
    this.start = function() {
      this.moving = true;
    };
    this.stop = function() {
      this.moving = false;
    };
  }

  function start() {
    message.innerHTML = "";
    status = 3;
    reel[0].start();
    reel[1].start();
    reel[2].start();
    timerid = setInterval(drawReel, 100);
  }

  function stop(reelid) {
    reel[reelid].stop();
    status--;
    if (status == 0) {
      determine();
    }
  }

  function determine() {
    clearInterval(timerid);
    var n0 = reel[0].n;
    var n1 = reel[1].n;
    var n2 = reel[2].n;
    if (n0 == n1 && n1 == n2) {
      message.innerHTML = "おめでとうございます。";
    } else if (n0 == n1 || n1 == n2 || n2 ==  n0) {
      message.innerHTML = "なかなかやりますね。";
    } else {
      message.innerHTML = "残念でした。";
    }
  }

  function drawReel() {
    context.clearRect(0, 0, 400, 300);
    for (var i = 0; i <= 2; i++) {
      reel[i].increment();
      context.fillText(reel[i].n, i*120+80, 150);
    }
  }
</script>
```

```
  </body>
</html>
```

本来のスロットマシンは、リールが回転すると絵柄が変化するわけですが、この HTML 文書で変化するのは、絵柄ではなくて 1 から 9 までの数字です。

「開始」のボタンをクリックすると、すべてのリールの回転が開始します。リールの回転は、それぞれのリールに対応する「停止」のボタンをクリックすることによって停止させることができます。すべてのリールの回転が停止した場合は、数字がそろっているかどうかに応じたメッセージが表示されます。

13.2 15 パズル

13.2.1 15 パズルとは何か

この節では、「15 パズル」と呼ばれるパズルの HTML 文書を紹介したいと思います。

15 パズルは、「スライディングブロックパズル」と呼ばれるパズルの一種です。「スライディングブロックパズル」(sliding block puzzle) というのは、盤面の上で駒をスライドさせていって、特定の駒の配置を作ることを目的とするパズルのことです。

「15 パズル」(15 puzzle) は、4×4 の升目を持つ盤面と、1 から 15 までの数字が書かれた 15 個の駒を使うスライディングブロックパズルです。駒は、初期状態では、

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

という配置になっています。そして、1 箇所だけ空いている升目を利用することによって駒をスライドさせて、目的の配置を作ります。目的となる配置は一つではなく、たとえば、

1	2	3	4
12	13	14	5
11		15	6
10	9	8	7

という渦巻き型などがあります。

13.2.2 15 パズルの HTML 文書

次の HTML 文書は、15 パズルを HTML と JavaScript で書いたものです。

HTML 文書の例 15puzzle.html

```
<!DOCTYPE html>
<html>
  <head><title>15 パズル</title></head>
  <body>
    <canvas id="canvas1" width="400" height="400"></canvas>
    <script>
      var size = 100;  // 升目の大きさ
      var margin = 10; // 升目のマージン
      var canvas = document.getElementById("canvas1")
      var context = canvas.getContext("2d");
      context.font = "70px serif";
      context.textAlign = "center";
      var rect = canvas.getBoundingClientRect();
      var puzzle = new Puzzle();
      puzzle.draw();
      document.addEventListener("click", function (event) {
        puzzle.clicked(event);
      }, false);

      function Puzzle() {
        this.state = [
```

```
          [1, 2, 3, 4],
          [5, 6, 7, 8],
          [9, 10, 11, 12],
          [13, 14, 15, -1]
        ];
        this.clicked = function(event) {
          var i = Math.floor((event.clientY - rect.top) / size);
          var j = Math.floor((event.clientX - rect.left) / size);
          if (i >= 0 && i <= 3 && j >= 0 && j <= 3) {
            this.move(i, j);
          }
        };
        this.move = function(i, j) {
          if (j >= 1 && this.empty(i, j-1)) {        // 左
            this.exchange(i, j, i, j-1);
          } else if (j <= 2 && this.empty(i, j+1)) { // 右
            this.exchange(i, j, i, j+1);
          } else if (i >= 1 && this.empty(i-1, j)) { // 上
            this.exchange(i, j, i-1, j);
          } else if (i <= 2 && this.empty(i+1, j)) { // 下
            this.exchange(i, j, i+1, j);
          }
        };
        this.exchange = function(i1, j1, i2, j2) {
          this.state[i2][j2] = this.state[i1][j1];
          this.state[i1][j1] = -1;
          this.drawPiece(i1, j1);
          this.drawPiece(i2, j2);
        };
        this.draw = function() {
          for (var i = 0; i <= 3; i++) {
            for (var j = 0; j <= 3; j++) {
              this.drawPiece(i, j);
            }
          }
        };
        this.drawPiece = function(i, j) {
          var n = this.state[i][j];
          if (n == -1) {
            context.clearRect(j*size, i*size, size, size);
          } else {
            context.fillStyle = "green";
            context.fillRect(j * size + margin, i * size + margin,
              size - margin, size - margin);
            context.fillStyle = "white";
            context.fillText(n,
              j*size + size * 0.55, i*size + size * 0.78);
          }
        };
        this.empty = function(i, j) {
          return this.state[i][j] == -1;
        };
      }
    </script>
  </body>
</html>
```

空いている升目の隣にある駒をクリックすると、その駒が、空いている升目にスライドします。

13.3　モグラ叩き

13.3.1　モグラ叩きとは何か

この節では、「モグラ叩き」と呼ばれるゲームの HTML 文書を紹介したいと思います。

「モグラ叩き」(whac-a-mole) は、モグラを退治するゲームです。

モグラは、原っぱの地下に棲んでいますが、しばしば地上に出現します。ただし、原っぱのど

こに出現するかということをあらかじめ予測することはできません。プレイヤーは、地上に出現したモグラを叩くことによって、そのモグラを退治することができます。

13.3.2 乱数

モグラ叩きを作るためには、原っぱの中のランダムな位置にモグラを出現させる必要があります。そのような処理をプログラムに実行させたいときは、通常、「乱数」と呼ばれるものが使われます。

「乱数」(random number) というのは、予測することのできない数値のことです。

JavaScript では、`random` というメソッドを呼び出すことによって、1 個の乱数を求めることができます。このメソッドは、

```
Math.random()
```

というメソッド呼び出しを書くことによって呼び出すことができます。このメソッドの戻り値は、0 と 1 のあいだの乱数です。

`floor` を使うことによって、整数の乱数を求めることもできます。たとえば、

```
Math.floor(Math.random() * 6) + 1
```

という式を書くことによって、1 から 6 までの整数の乱数を求めることができます。ですから、この式を使うことによって、サイコロを作ることができます。

13.3.3 モグラ叩きの HTML 文書

次の HTML 文書は、モグラ叩きを HTML と JavaScript で書いたものです。

HTML 文書の例 `mole.html`

```
<!DOCTYPE html>
<html>
  <head><title>モグラ叩き</title></head>
  <body>
    <p id="scoreboard">0/0</p>
    <canvas id="canvas1" width="200" height="200"></canvas>
    <form>
      <input type="button" value="開始" onclick="start()">
    </form>
    <script>
      var size = 10;        // 原っぱの大きさ
      var areasize = 20;    // 升目の大きさ
      var margin = 1;       // 升目のマージン
      var interval = 1000; // 出現間隔（ミリ秒）
      var score = 0;        // 退治したモグラの数
      var moles = 0;        // 出現したモグラの数
      var rect;             // Canvas の位置と大きさ
      var scoreboard = document.getElementById("scoreboard");
      var canvas = document.getElementById("canvas1");
      var context = canvas.getContext("2d");
      var rect = canvas.getBoundingClientRect();
      var field = new Field(); // 原っぱ

      function Field() {
        this.position = { i: 0, j: 0 }; // モグラの位置
        this.mole = false;  // モグラが出現中かどうか
        this.table = new Array(size);
        this.click = function(event) {
          var i = this.position.i;
          var j = this.position.j;
          var mouse = this.mousePosition(
            event.clientX, event.clientY);
          if (this.mole && i == mouse.i && j == mouse.j) {
            scoreboard.innerHTML = ++score + "/" + moles;
            this.mole = false;
            this.table[i][j].drawRect();
          }
        };
        this.emerge = function() {
```

```
          this.table[this.position.i][this.position.j].drawRect();
          this.position = this.randomPosition();
          this.table[this.position.i][this.position.j].drawMole();
          this.mole = true;
          scoreboard.innerHTML = score + "/" + ++moles;
        };
        this.randomPosition = function() {
          return {
            i: Math.floor(Math.random() * size),
            j: Math.floor(Math.random() * size)
          };
        }
        this.createTable = function() {
          for (var i = 0; i < size; i++) {
            this.table[i] = new Array(size);
            for (var j = 0; j < size; j++) {
              this.table[i][j] = new Area(i, j);
            }
          }
        };
        this.mousePosition = function(x, y) {
          return {
            i: Math.floor((y - rect.top) / areasize),
            j: Math.floor((x - rect.left) / areasize)
          };
        };
        this.isInsideField = function(i, j) {
          return i >= 0 && i < size && j >= 0 && j < size;
        };
        this.createTable();
      }

      function Area(i, j) {
        this.i = i; // 縦方向の位置
        this.j = j; // 横方向の位置
        this.drawRect = function() {
          context.fillStyle = "lime";
          context.fillRect(
            this.j * areasize + margin,
            this.i * areasize + margin,
            areasize - margin * 2, areasize - margin * 2);
        };
        this.drawMole = function() {
          context.fillStyle = "maroon";
          context.beginPath();
          context.arc(
            this.j * areasize + areasize * 0.5,
            this.i * areasize + areasize * 0.5,
            areasize * 0.3, 0, Math.PI*2);
          context.fill();
        };
        this.drawRect();
      }

      function start() {
        setInterval(function() { field.emerge(); }, interval);
        document.addEventListener("click", function(event) {
          field.click(event);
        }, false);
      }
    </script>
  </body>
</html>
```

「開始」のボタンをクリックすると、ゲームが開始されます。モグラ（小豆色の円で表示されます）は、マウスでクリックすることによって退治することができます（モグラが出現した升目

の中ならば、どこをクリックしてもかまいません)。表示されている数字は、退治したモグラの数と、出現したモグラの数です。

13.4 テニスのようなゲーム

13.4.1 当たり判定とは何か

この節では、ラケットを動かすことによってボールを打ち返すことのできる、テニスのようなゲームをプレイすることのできる HTML 文書を紹介したいと思います。

画面上に表示された複数の物体が互いに衝突したときに何らかの処理が実行されるようなゲームを作るためには、「当たり判定」と呼ばれる技術が必要になります。「当たり判定」(collision detection) というのは、ゲームの中に登場する二つの物体が、空間的に接触しているかそれとも離れているかということを判定することです。

13.4.2 線分の当たり判定

まずは、1 次元空間での当たり判定について考えてみましょう。

1 次元空間では、すべての物体は線分という形状を持つことになります。線分は、位置と長さであらわされます。

引数として二つの線分の位置（左端）と長さを受け取って、それらの当たり判定をするメソッドは、次のように定義することができます。

```
this.overlap = function(a, alen, b, blen) {
  if (a < b) {
    return b <= a + alen;
  } else {
    return a <= b + blen;
  }
};
```

13.4.3 長方形の当たり判定

次に、2 次元空間での当たり判定について考えてみましょう。

2 次元空間の物体はさまざまな形状を持っているわけですが、ここでは説明を簡潔にするために、水平と垂直の辺を持つ長方形だけについて考えることにします。

長方形のオブジェクトを生成するコンストラクタは、次のように定義することができます。

```
function Rectangle(x, y, width, height) {
  this.x = x;            // 左上の頂点の x 座標
  this.y = y;            // 左上の頂点の y 座標
  this.width = width;    // 横の長さ
  this.height = height; // 縦の長さ
    ...
```

二つの長方形の当たり判定は、x 軸方向と y 軸方向のそれぞれについて 1 次元の当たり判定をして、それらの論理積を求めることによって実現することができます。つまり、次のようなメソッドを定義すればいいわけです。

```
this.collision = function(rect) {
  return this.overlap(this.x, this.width,
           rect.x, rect.width) &&
         this.overlap(this.y, this.height,
           rect.y, rect.height);
};
```

13.4.4 テニスのようなゲームの HTML 文書

次の HTML 文書は、三つの辺を壁で囲まれたコートの中でラケットを動かしてボールを打ち返す、というゲームを HTML と JavaScript で書いたものです。

HTML 文書の例 tennis.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>テニスのようなゲーム</title>
```

```
    <style>
      p { font-size: 20px; }
      canvas { background-color: green; }
    </style>
  </head>
  <body>
    <p id="scoreboard">0</p>
    <canvas id="canvas1" width="300" height="400"></canvas>
    <form>
      <input type="button" value="開始" onclick="start()">
    </form>
    <p id="message"></p>
    <script>
      var scoreboard = document.getElementById("scoreboard");
      var message = document.getElementById("message");
      var canvas = document.getElementById("canvas1");
      var context = canvas.getContext("2d");
      context.fillStyle = "springgreen";
      context.font = "20px serif";
      context.textAlign = "center";
      var canvasrect = canvas.getBoundingClientRect();
      var width = canvas.width;
      var height = canvas.height;
      var courtrect = new Rectangle(0, 0, width, height);
      var ball = new Ball(14, 14, 3);
      var racket = new Racket(width/2, height-40, 40, 10);
      var score = 0;
      var timerid = -1;
      document.addEventListener("mousemove", function(event) {
        racket.mousemoved(event);
      }, false);

      function Ball(width, height, step) {
        this.rect = new Rectangle(0, 0, width, height);
        this.step = step;      // 描画ごとの移動距離
        this.r = width / 2;    // 半径
        this.direction = "se"; // 移動方向
        this.reset = function() {
          this.rect.x = 0;
          this.rect.y = 0;
          this.direction = "sw";
        };
        this.move = function(x, y) {
            this.clear();
            this.rect.x += this.step * x;
            this.rect.y += this.step * y;
            context.fillStyle = "yellow";
            this.draw();
        };
        this.moveDirection = function() {
          switch (this.direction) {
            case "se": // 右下
              this.move(1, 1);
              break;
            case "sw": // 左下
              this.move(-1, 1);
              break;
            case "nw": // 左上
              this.move(-1, -1);
              break;
            case "ne": // 右上
              this.move(1, -1);
              break;
          }
        };
        this.turn = function() {
          switch (this.rect.outside(courtrect)) {
```

```
      case "west":
        if (this.direction == "sw") {
          this.direction = "se";
        } else if (this.direction == "nw") {
          this.direction = "ne";
        }
        break;
      case "east":
        if (this.direction == "se") {
          this.direction = "sw";
        } else if (this.direction == "ne") {
          this.direction = "nw";
        }
        break;
      case "north":
        if (this.direction == "ne") {
          this.direction = "se";
        } else if (this.direction == "nw") {
          this.direction = "sw";
        }
        break;
    }
  };
  this.hit = function() {
    if (this.south() &&
        this.rect.collision(racket.rect)) {
      scoreboard.innerHTML = ++score;
      if (this.direction == "se") {
        this.direction = "ne";
      } else if (this.direction == "sw") {
        this.direction = "nw";
      }
    }
  };
  this.south = function() {
    return this.direction == "se" || this.direction == "sw";
  };
  this.beyondSouth = function() {
    return this.rect.outside(courtrect) == "south";
  };
  this.clear = function() {
    context.clearRect(this.rect.x, this.rect.y,
                      this.rect.width, this.rect.height);
  }
  this.draw = function() {
    context.beginPath();
    context.arc(this.rect.x + this.r,
                this.rect.y + this.r,
                this.r, 0, Math.PI*2, false);
    context.fill();
  }
}

function Racket(x, y, width, height) {
  this.rect = new Rectangle(x, y, width, height);
  this.mousemoved = function(event) {
    this.clear();
    this.rect.x = event.clientX - canvasrect.left -
      this.rect.width / 2;
    this.draw();
  };
  this.clear = function() {
    context.clearRect(this.rect.x, this.rect.y,
      this.rect.width, this.rect.height);
  };
  this.draw = function() {
    context.fillStyle = "silver";
```

```
        context.fillRect(this.rect.x, this.rect.y,
          this.rect.width, this.rect.height);
      };
    }

    function Rectangle(x, y, width, height) {
      this.x = x;           // 左上の頂点の x 座標
      this.y = y;           // 左上の頂点の y 座標
      this.width = width;   // 横の長さ
      this.height = height; // 縦の長さ
      this.outside = function(rect) {
        if (this.x < rect.x) {
          return "west";
        } else if (this.x + this.width >
                rect.x + rect.width) {
          return "east";
        } else if (this.y < rect.y) {
          return "north";
        } else if (this.y + this.height >
                rect.y + rect.height) {
          return "south";
        } else {
          return "inside";
        }
      };
      this.collision = function(rect) {
        return this.overlap(this.x, this.width,
                  rect.x, rect.width) &&
               this.overlap(this.y, this.height,
                  rect.y, rect.height);
      };
      this.overlap = function(a, alen, b, blen) {
        if (a < b) {
          return b <= a + alen;
        } else {
          return a <= b + blen;
        }
      };
    }

    function gameOver() {
      ball.clear();
      message.innerHTML = "ゲームオーバー";
      clearInterval(timerid);
      timerid = -1;
    }

    function start() {
      if (timerid == -1) {
        message.innerHTML = "";
        scoreboard.innerHTML = "0";
        ball.reset();
        score = 0;
        timerid = setInterval(function() {
          if (ball.beyondSouth()) {
            gameOver();
          } else {
            ball.turn();
            ball.hit();
            ball.moveDirection();
          }
        }, 10);
      }
    }
  </script>
 </body>
</html>
```

「開始」のボタンをクリックすると、ゲームが開始されます。ラケットは、マウスを使って左右に移動させることができます。ラケットでボールを 1 回打ち返すごとに、得点に 1 点が加算されます。ボールがコートの下の辺まで到達すると、ゲームオーバーになります。

参考文献

[DOM2,2000] Arnaud Le Hors, Philippe Le Hegaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion and Steve Byrne (eds.), “Document Object Model (DOM) Level 2 Core Specification, Version 1.0”, World Wide Web Consortium, 2000.

[HTML5,2010] Ian Hickson ed., “HTML5: A vocabulary and associated APIs for HTML and XHTML”, World Wide Web Consortium, 2010.

[HTMLCanvas2DContext,2010] Ian Hickson ed., “HTML Canvas 2D Context”, World Wide Web Consortium, 2010.

[ECMAScript,2009] “ECMAScript Language Specification, Fifth Edition”, Ecma International, 2009.

[佐藤,2007] 佐藤信正、『JavaScript 完全マスター・再入門編』、メディア・テック出版、2007、ISBN 978-4-89627-344-1。

[羽田野,2007] 羽田野太巳、『標準 DOM スクリプティング : JavaScript+DOM による Web アプリデザインの基礎』、ソフトバンククリエイティブ、2007、ISBN 4-7973-3638-2。

[羽田野,2010] 羽田野太巳、白石俊平、古籏一浩、太田昌吾、『Google API Expert が解説する HTML5 ガイドブック』、インプレスジャパン、2010、ISBN 978-4-8443-2927-5。

[古籏,2008] 古籏一浩、『DOM Scripting : 高機能な Web ページ構築のために』、Web 標準テキストシリーズ、1、技術評論社、2008、ISBN 978-4-7741-3326-3。

[柳井,2010] 柳井政和、『マンガでわかる JavaScript』、秀和システム、2010、ISBN 978-4-7980-2783-8。

[山田,2010] 山田祥寛、『JavaScript 本格入門 : モダンスタイルによる基礎から Ajax・jQuery まで』、技術評論社、2010、ISBN 978-4-7741-4466-5。

索引