

ソフトウェア開発

－ 授業配布版 －

平野 照比古

目次

第 1 回	ガイダンス	1
1.1	受講に関する注意	1
1.2	参考図書	1
1.3	シラバス	2
1.4	JavaScript の実行環境	2
1.5	strict モードについて	3
1.6	第 1 回目復習課題	3
第 2 回	JavaScript のデータとプログラミングの基礎	5
2.1	データの型	5
2.1.1	プリミティブデータ型	5
2.1.2	Number 型	5
2.1.3	String 型	6
2.1.4	Bool 型	7
2.1.5	undefined	7
2.1.6	null	7
2.2	変数	7
2.3	配列	8
2.3.1	配列の宣言と初期化	8
2.3.2	配列のメソッド	9
2.4	演算子	10
2.4.1	代入、四則演算	10
2.4.2	論理演算子と比較演算子	11
2.5	制御構造	12
2.6	組み込みオブジェクト	12
2.6.1	組み込み関数	12
2.6.2	Math オブジェクト	13
2.6.3	Date オブジェクト	14
2.7	第 2 回目復習課題	16
第 3 回	関数	17
3.1	今回の内容	17
3.2	関数の定義方法と呼び出し	17

3.2.1	簡単な関数の例	17
3.2.2	仮引数への代入	19
3.2.3	可変引数をとる関数	20
3.2.4	変数のスコープ	20
3.2.5	スコープチェイン	23
3.3	JavaScript における関数の特徴	23
3.3.1	関数もデータ	24
3.3.2	コールバック関数	24
3.3.3	自己実行関数	25
3.4	クロージャ	26
3.4.1	クロージャの例 – 変数を隠す	26
3.4.2	クロージャの例 – 無名関数をその場で実行する	27
第 4 回	オブジェクト	31
4.1	配列とオブジェクト	31
4.2	オブジェクトリテラルと JSON	33
4.3	クラス	34
4.3.1	クラスの宣言とインスタンスの生成	34
4.3.2	クラスメソッド	36
4.3.3	静的メソッド	38
4.3.4	継承	38
4.3.5	<code>instanceof</code> 演算子	39
4.3.6	同じクラスのインスタンスに通し番号をつける	39
第 5 回	オブジェクトのプロトタイプと継承	41
5.1	オブジェクト属性	41
5.1.1	<code>class</code> 属性	41
5.1.2	<code>extensible</code> 属性	42
5.1.3	<code>prototype</code> 属性	44
5.1.4	<code>prototype</code> の使用例	44
5.2	継承	47
5.3	エラーオブジェクトについて	48
5.3.1	エラー処理の例	48
5.3.2	エラーからの復帰	51
第 6 回	正規表現	53
6.1	正規表現オブジェクトの記述方法	53
6.2	文字列のパターンマッチングメソッド	57

第 7 回	DOM の利用	61
7.1	HTML 文書の構成	61
7.2	CSS の利用	63
7.3	DOM とは	66
7.4	DOM のメソッドとプロパティ	66
7.4.1	DOM のメソッド	66
7.4.2	DOM の要素のプロパティ	70
第 8 回	イベント	71
8.1	イベント概説	71
8.2	イベント処理の方法	71
8.3	DOM Level 2 のイベント処理モデル	72
8.4	HTML における代表的なイベント	72
8.4.1	ドキュメントの <code>onload</code> イベント	72
8.4.2	マウスイベント	73
8.5	イベント処理の例	74
第 9 回	PHP の超入門	83
9.1	Web サーバーの基礎知識	83
9.1.1	サーバーの重要な設定項目	83
9.1.2	CGI	83
9.1.3	Web サーバーのインストール	83
9.2	PHP とは	84
9.3	PHP プログラムの書き方	84
9.4	データの型	84
9.5	変数	85
9.5.1	可変変数	86
9.5.2	定義済み変数	86
9.6	文字列	87
9.6.1	文字列の定義方法	87
9.6.2	文字列を取り扱う関数	88
9.7	式と文	89
9.8	配列	90
9.8.1	配列の基礎	90
9.8.2	配列に関する制御構造	92
9.8.3	配列に関する関数	93
9.9	関数	95
9.9.1	PHP の関数の特徴	95
9.9.2	関数の定義の例	96
9.9.3	可変関数	98

第 10 回 サーバーとのデータのやり取り	99
10.1 サーバーとのデータ交換の基本	99
10.2 スパースグローバルの補足	101
10.3 Web Storage	101
10.4 Ajax	105
第 11 回 jQuery	111
11.1 jQuery とは	111
11.2 jQuery の基本	111
11.2.1 jQuery() 関数と jQuery オブジェクト	111
11.2.2 jQuery オブジェクトのメソッド	112
11.3 ドキュメントの構造の変更	113
11.4 イベントハンドラーの取り扱い	113
11.5 Ajax の処理	114
11.6 jQuery のサンプル	114
第 12 回 JavaScript ライブラリーの配布	117
12.1 jQuery のコードを読む	117
12.2 JavaScript ファイルの短縮化	118
12.2.1 JavaScript ファイルの短縮化について	118
12.2.2 短縮化の例	120
12.3 Web サイトの効率化	123
12.3.1 Contents Deliverly Network(CDN)	123
12.3.2 CSS Sprite	124
第 13 回 XML ファイルの処理	125
13.1 XML ファイルとは	125
13.1.1 W3C における XML の解説	125
13.1.2 XML ファイルの例	126
13.2 Google Maps における Polyline の表示	128
13.3 ブラウザでの処理	128
13.4 PHP による処理	133
第 14 回 システム開発のためのヒント	135
14.1 コードの構造	135
14.1.1 良いコードとは	135
14.1.2 プログラムとデータの分離	135
14.2 コードの管理	136
14.2.1 複数でシステム開発をするときの問題点	136
14.2.2 バージョン管理の概念	137
14.2.3 バージョン管理ソフトの概要	137

14.2.4 git の使い方	137
---------------------------	-----

第1回 ガイダンス

1.1 受講に関する注意

この授業に関する注意は次の通りである。

- JavaScript を通常の計算機言語として利用するための解説を行う。
- 進度が今までのプログラミングの授業より早いので復習をよくすること。
- 演習は原則行わない。出された課題は自宅で行うこと。また、レポートの提出を必ずすること。
- パソコンを授業に持参する必要がある場合はその旨、前回の授業で指示する。
- 11 年度以前の学生に対しては「アルゴリズムとデータ構造」の読み替え科目となっている。
- 復習用の課題を必ず行うこと。次回の授業の開始時に確認の小テストを行うことがある。
- 最終的な成績は試験を行う。
- 配布資料等は <http://www.hilano.org/hilano-lab> で公開する予定

1.2 参考図書

1. E. Brown, 初めての JavaScript 第 3 版, ES2015 以降の最新ウェブ開発, オライリージャパン
2. D. Crockford, JavaScript: The Good Parts 「良いパーツ」によるベストプラクティス, オライリージャパン
3. David Flanagan, JavaScript 第 6 版, オライリージャパン
4. David Flanagan, JavaScript クイックリファレンス第 6 版, オライリージャパン
5. Nicholas C. Zakas, ハイパフォーマンス JavaScript, オライリージャパン
6. Nicholas C. Zakas, メンテナブル JavaScript 読みやすく保守しやすい JavaScript コードの作法, オライリージャパン

1.3 シラバス

この授業のシラバスは次のように予定している。

授業回数	内容
第1回	授業のガイダンスとブラウザの開発者モードについて JavaScript の実行環境の確認
第2回	JavaScript が扱うデータ データの型と演算子に関する注意など
第3回	関数の定義方法と変数のスコープ
第4回	オブジェクトの定義方法
第5回	オブジェクト属性と継承 プロトタイプによる継承など
第6回	正規表現と文字列の処理
第7回	DOM の利用 HTML 文書の例、CSS と DOM の基礎
第8回	イベント処理 イベントモデルとイベント処理の例
第9回	PHP 超入門 PHP に関する簡単なプログラムの例
第10回	サーバーとのデータの交換 (1) PHP 入門の続きとサーバーとのデータ交換の基礎
第11回	jQuery DOM の処理を簡単にするライブラリーの紹介
第12回	JavaScript ライブラリーの配布 jQuery のコードの短縮化、Web サイトの効率化
第13回	XML ファイルの処理
第14回	システム開発のヒント
第15回	最終試験と解説

1.4 JavaScript の実行環境

JavaScript は HTML 文書内で使われる場合が多いが、通常の計算機言語として取り扱うこともできる。この授業では JavaScript の計算機言語としての特徴について述べた後、Web ブラウザ内での処理について解説を行う。しかし、JavaScript を単独で実行する環境はほとんどなく¹、このテキストではブラウザ内で実行する例だけを取り上げる。

最近のブラウザは JavaScript の実行環境を提供するだけでなく、開発環境も提供している。Google Chrome のデベロッパーツール、Firefox の Web 開発、Opera の開発者用ツール、Internet

¹ コマンドラインから JavaScript を実行する環境としては Node.js があるが、新規にインストールする必要がある。

Explorer の開発者ツールなどがそうである。これらのツールは「F12」または「Control+Shift+I」のショートカットキーで表示、非表示ができる²。表示されるタブには次のようなものがある³。

- **Elements** HTML の構造 (DOM の構造) を表示する。対応する要素の部分が反転する。
- **Console** エラーや警告が表示される。また、JavaScript のプログラムが対話形式で実行できる。
- **Sources** HTML 文書のソースが表示される。JavaScript のソースの場合にはプログラムの実行を中断するブレイクポイントの設定が可能である。
- **Network** ファイルの取得などの順序やかかった時間などが表示される。
- **Timeline** ブラウザの処理に関する情報が表示される。
- **Profiles** JavaScript の関数で使われた実行時間などが記録できる。

1.5 strict モードについて

ECMAScript の改訂版 ES2015⁴ では **strict** モードという厳密な解釈をするモードが導入された。このモードでは従来見つけにくい単純なバグがエラーとなる。主な違いは次のとおりである。

	非 strict モード	strict モード
変数の宣言	必要ではない	必要
書き込み不可なプロパティへの代入	エラーが発生しない	エラーが発生
関数の arguments オブジェクトの値の変更	可能	不可能
関数の arguments.caller	参照可能	エラーが発生
関数の arguments.callee	参照可能	エラーが発生
8 進リテラル (0 で始まる数)	使用可能	エラーが発生

プログラムを **strict** モードにするためには先頭に `"use strict;"` を記述する。

1.6 第 1 回目復習課題

課題 1.1 各自が使用しているブラウザにおいて JavaScript の文が直接実行できるコンソールが開けること、コンソールで「1+2」と入力すると、計算結果が表示されることを確認すること。

課題 1.2 次のコードを拡張子が `html` のファイルで作成する。それをブラウザで開き、デベロッパーツールのコンソールから `foo();` と入力したときと、その後、コンソールで `i` と入力したときの結果を確認すること。同様のことを 9 行目の `i` の宣言を省いて行うこと

また、**strict** モードに変更し (7 行目の後に `"use strict;"` を追加する)、9 行目の変数の宣言の行を取り除くとエラーが発生することを確認すること。

²Opera は「F12」に、Internet Explorer は「Control + Shift+I」に対応していないようである。

³この例は Chrome のものの一部である。

⁴ES6 とも呼ばれる。今後、EcmaScript のバージョンは規格が制定された年号を用いることとなった。

```
1 <!DOCTYPE HTML>
2 <html>
3 <head>
4 <meta charset="UTF-8"/>
5 <title>初めての JavaScript</title>
6 <script type="text/ecmascript">
7 //<![CDATA[
8 function foo(){
9   let i;
10  for(i=1;i<10;i++) {
11    console.log(`${i} ${i*i}`);
12  }
13 }
14 alert("デベロッパーツールからコンソールを開いてコンソールから foo(); と入力してください。");
15 //]]>
16 </script>
17 </head>
18 </html>
```

コードの簡単な説明

- 1行目 HTML5 における HTML 文書の宣言
- 4行目 このファイルの文字コード (エンコーディング) を UTF-8 に指定
- 6行目 スクリプトの開始の要素。プログラミング言語が ECMAScript であることを宣言
- 7行目 `//`は行末までの部分をコメントにする JavaScript の記法。残りの部分はこれ以降 12 行目までは通常の文字として解釈することを指定 (CDATA セクションの開始)。7行目と 13 行目を消去したらどうなるのか確認することまたその理由も考えること⁵。
- 8行目は関数 `foo()` の宣言。13 行目までがこの関数の定義範囲
- 9行目は変数 `i` の宣言 (従来の JavaScript では `var` を用いて宣言していたが、この授業では ES2015 で新しく導入された `let` だけを用いる。)
- 10行目 C 言語などでおなじみの繰り返しの指定
- 11行目 引数内の式をコンソールに出力。ここでは ES2015 で導入された文字列の中に式を埋め込むことができるテンプレートリテラル形式 (バッククォート```の上にある文字で挟む) で記述している。
- 14行目はメッセージボックスにコンソールを開くことを指示。

開発者ツールを開き、コンソールから `foo()`; を入力する。

⁵最近のブラウザではエラーが起きないかもしれない。

第2回 JavaScript のデータとプログラミングの基礎

2.1 データの型

JavaScript の方には大きく分けてプリミティブ型と非プリミティブ型の2種類がある。

2.1.1 プリミティブデータ型

表 2.1は JavaScript におけるプリミティブデータ型の一覧である。

表 2.1: プリミティブデータ型

型	説明
Number	64 ビット浮動小数点形式だけ
String	文字列型、文字のデータ型はない。ダブルクォート (") かシングルクォート (') で囲む。ES2015 では文字列内に式の結果を埋め込むことができるテンプレートリテラル形式 (バッククォート `ではさむ ¹) が追加された。
Boolean	true か false の値のみ
undefined	変数の値が定義されていないことを示す
null	null という値しか取ることができない特別なオブジェクト

変数や値の型を知りたいときは `typeof` 演算子を使う。

2.1.2 Number 型

数を表現する方法 (数値リテラル) としては次のものがある。

- **整数リテラル** 10 進整数は通常通りの形式である。16 進数を表す場合は先頭に `0x` か `0X` をつける。先頭が `0` でそのあとに `x` または `X` が来ない場合は 8 進数と解釈される場合がある。`strict` モードではこの形式はエラーとなる。
2 進数は `0b` で、8 進数は `0o` で始まるリテラルが定義されている。

¹キーボードの```の上にある文字

- **浮動小数点リテラル** 整数部、そのあとに必要なならば小数点、小数部そのあとに指数部がある形式である。

特別な Number Number 型には次のような特別な Number が定義されている。

- **Infinity** 無限大を表す読み出し可能な変数である。オーバーフローした場合や $1/0$ など演算の結果としてこの値が設定される。
- **NaN** Not a Number の略である。計算ができなかった場合表す読み出し可能な変数である。文字列を数値に変換できない場合や $0/0$ などの結果としてこの値が設定される。

2.1.3 String 型

文字列は" または ' ではさむ。1 文字だけの文字の型はない。ES2015 では文字列の中に式の値を埋め込むことができるテンプレートリテラルが新たに導入された。テンプレートリテラルはバッククォート (```) ではさむ。埋め込む式は `${2+3}` のように記述する。

```
'2+3=${2+3}' => "2+3=5"
```

文字列に関する演算子としては、2 つの文字列をつなげる接続演算子がある。JavaScript では + を用いる。詳しくは 2.4 を参照のこと。

文字列に関する情報 (プロパティ) や操作 (メソッド) として次のものがある²。

課題 2.1 次の実行結果を確かめなさい。

1. `"0123456789".indexOf("1");`
2. `"0123456789".indexOf("a");`
3. `"0123456789".indexOf("1",2);`
4. `"0,1,2,3".split(",");`
5. `"0,1,2,3".split(", ",2);`
6. `"0123".split("");`
7. `"0123456789".substring(3);`
8. `"0123456789".substring(-3);`
9. `"0123456789".slice(-3);`
10. `"0123456789".substring(3,5);`
11. `"0123456789".slice(3,5);`
12. `"0123456789".slice(3,-3);`

²この表にある正規表現については後日解説をする

表 2.2: 文字列のメソッドとプロパティ

メンバー	説明
<code>length</code>	文字列の長さ
<code>indexOf(needle, start)</code>	与えられた文字列内に needle が初めて現れる位置を返す。 start の引数があれば、その位置以降から調べる。見つからない場合は <code>-1</code> を返す。
<code>lastIndexOf(needle, start)</code>	与えられた文字列内に needle が一番最後に現れる位置を返す。 start の引数があれば、指定された位置以前から調べる。見つからない場合は <code>-1</code> を返す。
<code>split(separator, limit)</code>	separator の文字列 (または正規表現) で与えられた文字列を分割し、配列で返す。 separator の部分は返されない。2 番目の引数はオプションで、分割する最大数を返す。
<code>substring(start, end)</code>	与えられた文字列の start から end の前の位置までの部分文字列を返す。
<code>slice(start, end)</code>	与えられた文字列の start から end の前の位置までの部分文字列を返す。値が負のときは文字列末尾から数えた位置を表す。

2.1.4 Bool 型

`true` と `false` の 2 つの値をとる。この 2 つは予約語である。論理式の結果としてこれらの値が設定されたり、論理値が必要なところでこれらの値に設定される。詳しくは 2.4 を参照のこと。

2.1.5 undefined

値が存在しないことを示す読み出し可能な変数である。変数が宣言されたのに値が設定されていないときの初期値や、戻り値が定義されていない関数の戻り値はこの値になる。

2.1.6 null

`typeof null` の値が `"object"` であることを示すように、オブジェクトが存在しないことを示す特別なオブジェクト値（であると同時にオブジェクトでもある）である。

2.2 変数

JavaScript の変数の特徴は次の通りである。

- 変数名はアルファベットまたは `_` (アンダースコア) で始まる英数字または `_` が続く文字列

- 大文字と小文字は区別される。
- 変数の宣言は `let` で行う。`var` でもできるが、問題があるのでこの授業では使用しない。
- 宣言時に初期化ができる。ES2015 で導入された `const` による変数の宣言では初期化時の値の変更ができない。
- 非 `strict` モードのときは変数は宣言をしなくても使用できる。初期化していない変数を演算の対象として使用するとエラーが起こる。詳しくは後述する。
- 変数に保存するデータの型には制限がない。途中で変更することもできる。

2.3 配列

2.3.1 配列の宣言と初期化

配列を使うためには、変数を配列で初期化する必要がある。変数の宣言と同時に行ってもよい。

```
let a = []; //空の配列の定義
let b = [1,2,3]; //要素が3つある配列
```

`a` は空の配列で、`b` は `b[0]=1, b[1]=2, b[2]=3` となる。次のことに注意する必要がある。

- 配列の各要素のデータの型は同じでなくてもよい。
- 実行時に配列の大きさを自由に変えられる。
- 配列の要素に配列を置くことができる。

```
let a=[1,[2,3,4],"a"];
```

- 配列の変数の代入は参照である。

```
>let a = [0,1,2,3,4];
undefined
>let b = a; //別の変数への代入
undefined
>b;
(5) [0, 1, 2, 3, 4] // a と同じ内容が表示
>b[3]= b[3]*10; // 4番目の要素を10倍
30
>b;
(5) [0, 1, 2, 30, 4]
>a;
(5) [0, 1, 2, 30, 4] // a も b と同じ要素の値を持つ
```

配列の代入が参照なので片方の要素を変化させると、両方とも変わることが分かる。

- 配列の要素の一部をまとめて別の変数に代入する分割代入が ES2015 で導入された。分割代入では代入演算子=の左辺を配列の形で記述する。

```
[a,,b,...c] = [0,1,2,3,4,5,6,7];
```

この代入の結果は次のとおりである。

```
a ==> 0
```

```
b ==> 2
```

```
c ==> [3, 4, 5, 6, 7]
```

- 変数 **a** には 0 番目の、**b** には 3 番目の、**c** には 4 番目以降の配列の要素が代入される。
- **a** と **b** の間に, , があるので 2 番目の要素は代入されない。
- 4 番目以降の要素はまとめて変数 **c** に代入されている。

この機能は関数の戻り値が配列であるとき、戻り値の必要なところだけ利用するために便利な機能である。

なお、ここで使用している... は展開演算子と呼ばれ、配列に対して要素を並べたものを表す。

課題 2.2 次の実行結果を確かめなさい。

1. `[,[,a]] = [1,[2,3,4],5]; console.log(a);`
2. `a=10;b=20;[b,a]=[a,b];console.log(a);console.log(b)`

2.3.2 配列のメソッド

表 2.3は配列のメソッドやプロパティをまとめたものである。

配列のメソッドとしてはこのほかに、引数に処理をする関数を与えるものがある。それらについては関数のところで解説をする。

課題 2.3 次の実行結果を確かめなさい³。

1. `[1,2,[],3].length;`
2. `let a=[1,2,3]; console.log(a.pop()); console.log(a.length);a;`
3. `let a=[1,2,3]; a.push(4,5); console.log(a.length);a;`
4. `let a=[1,2,3]; a.shift(4,5); console.log(a.length);a;`
5. `let a=[1,2,3]; a.join(" ");`
6. `let a=[1,2,3,4,5]; console.log(a.slice(1,2)); console.log(a.length);a;`
7. `let a=[1,2,3,4,5]; console.log(a.splice(1,2)); console.log(a.length);a;`
8. `let a=[1,2,3,4,5]; console.log(a.indexOf(3)); console.log(a.indexOf(3,3));`
9. `let a=[3,1,2,3,4,5]; console.log(a.lastIndexOf(3)); console.log(a.lastIndexOf(3,2));`

³2以降をコンソールで連続して行う場合には 3以降にある `let` はつけないこと。

表 2.3: 配列のメソッドとプロパティ

メンバー	説明
<code>length</code>	配列の要素の数
<code>join(separator)</code>	配列を文字列に変換する。 <code>separator</code> はオプションの引数で、省略された場合はカンマ, である。
<code>pop()</code>	配列の最後の要素を削除し、その値を返す。配列をスタックとして利用できる。
<code>push(i1,i2,...)</code>	引数で渡された要素を配列の最後に付け加える。配列をスタックやキューとして利用できる。
<code>shift()</code>	配列の最初の要素を削除し、その値を返す。配列をキューとして利用できる。
<code>unshift()</code>	配列の最初の要素を削除し、その値を返す。配列をキューとして利用できる。
<code>slice(start,end)</code>	<code>start</code> から <code>end</code> の前の位置にある要素を取り出した配列を返す。元の配列は変化しない。
<code>splice(start,No,i1,i2,...)</code>	<code>start</code> の位置から <code>No</code> の要素を取り除き、その位置に <code>i1,i1,...</code> 以下の要素を付け加える。元の配列を変更する。
<code>indexOf(value,start)</code>	配列の要素で <code>value</code> に等しい (<code>===</code>) ものを探し、その位置を返す。見つからない場合は <code>-1</code> を返す。オプションの引数 <code>start</code> はそのインデックス以降から探す。
<code>lastIndexOf(value,start)</code>	配列の要素で <code>value</code> に等しいものを探し、その位置を返す。見つからない場合は <code>-1</code> を返す。オプションの引数 <code>start</code> はそのインデックス以前から探す。

2.4 演算子

2.4.1 代入、四則演算

数に対しては C 言語と同様の演算子を使用できる。ただし、次のことに注意すること。

- `+` 演算子は文字列の接続にも使用できる。`+` 演算子は左右のオペランドが `Number` のときだけ、和をとる。一方が文字列の場合は数を文字列に直して、文字列として接続を行う。
- そのほかの演算子 (`-*/`) については文字列を数に変換してから数として計算する。
- 文字列全体が数にならない場合には変換の結果が `NaN` になる。

```
2+3 => 5
```

```
"2"+3 => "23"
```

```
"2"-0 +3 => 5
"2"*3    => 6
"2"*"3"  => 6
"f" *2    => NaN
"0xf"*2   => 30
"0o10"*2  => 16
```

- 整数を整数で割った場合、割り切れなければ小数となる。小数部分を切り捨てたいときは `Math.floor()` を用いる。

```
1/3 => 0.3333333333333333
Math.floor(1/3) => 0
```

2.4.2 論理演算子と比較演算子

論理演算子

Boolean 型に対して使用できる演算子は次の 3 つである。

- `!` 論理否定
- `&&` 論理積
- `||` 論理和

論理演算子を Boolean 型でない値を与えると次の値は `false` とみなされる。

- 空文字列 `""`
- `null`
- `undefined`
- 数字の `0`
- 数値の `NaN`
- Boolean の `false`

論理和や論理積では左のオペランドの結果により、式の値が決まる場合は右のオペランドの評価は行われない。たとえば、論理和の場合、左の値が `true` であれば右のオペランドの評価が行われな

い。次の例では変数 `a` の値は `1` のままである。

```
let a=1; true || (a=3);
```

比較演算子

比較演算子は比較の結果、Boolean の値を返す演算である。C 言語と同様の比較演算子を使用できる。`>`、`>=`、`<`、`<=`などがある。等しいことを比較するためには`==`(等価比較演算子)のほかに型変換を伴わない等価比較演算子`===`がある。等価比較演算子`==`は必要に応じて型変換を行う。特別な事情がない限り、`==`や`!=`は使用しないこと。

```
0 == "0" => true
0 === "0" => false
```

同様に非等価比較演算子`!=`にも型変換を伴わない非等価演算子`!==`がある。

`NaN === NaN`の結果は`false`である。値が`NaN`であるかを調べるには関数`isNaN()`を用いる。

2.5 制御構造

JavaScript ではC言語などと同様に`if`文、`for`文、`while`文、`switch`文がある。使い方は同様なのでここでは説明をしない。また、Java やC++のように`for`文の初期化のところに現れる変数を`let`を用いて宣言することもできる。この宣言された変数に関するJavaScript 特有の注意は次回解説をする。

2.6 組み込みオブジェクト

2.6.1 組み込み関数

JavaScript に組み込まれている代表的な関数としては次のものがある。

- `parseInt(string,radix)` `string`(文字列) と `radix`(基数、省略したときは10) をとり、先頭から見て正しい整数表現のところまで整数に変換する。
- `parseFloat(string)` `string`(文字列) をとり、先頭から見て正しい浮動小数点表現のところまで浮動小数に変換する。
- `isNaN(N)` `N` が数であれば `true`、そうでなければ `false` を返す。
- `isFinite(N)` `N` が数値または数値に変換できる値でかつ `Infinity` または `-Infinity` でないときに `true`、そうでないとき、`false` を返す。
- `encodeURIComponent(string)` `string` を URL エンコードする。Google の検索で日本語を含むものを対象とすると、アドレスバーに%で始まる文字列が見えるが、これが URL エンコードした結果である。
- `decodeURIComponent(string)` `encodeURIComponent(string)` の逆の操作をする。
- `encodeURI(string)` `string` を URI エンコーディングする。この関数はプロトコル部分などはエンコードしない。
- `decodeURI(string)` `encodeURI(string)` の逆の操作をする。

2.6.2 Math オブジェクト

Math オブジェクトには数学的な定数の定義 (円周率など) や三角関数などの関数が定義されている。表 2.4 はそれらのプロパティや関数をまとめたものである。いくつか注意しておく。

表 2.4: JavaScript における Math オブジェクトの種類

名称	種類	説明
Math.E	定数	自然対数の底 (2.71828182...)
Math.LN10	定数	$\log_e 10$
Math.LN2	定数	$\log_e 2$
Math.LOG2E	定数	$\log_2 e$
Math.LOG10E	定数	$\log_{10} e$
Math.PI	定数	円周率 ($\pi = 3.141592...$)
Math.SQRT1_2	定数	$\frac{1}{2}$ の平方根 $\sqrt{\frac{1}{2}}$
Math.SQRT2	定数	2 の平方根 $\sqrt{2}$
Math.abs(x)	関数	x の絶対値
Math.acos(x)	関数	逆余弦関数 $\arccos x$
Math.asin(x)	関数	逆正弦関数 $\arcsin x$
Math.atan(x)	関数	逆正接関数 $\arctan x$
Math.atan2(y, x)	関数	$\arctan \frac{y}{x}$ を計算。 x が 0 のときでも正しく動く
Math.ceil(x)	関数	x 以上の整数で最小な値を返す。
Math.cos(x)	関数	余弦関数 $\cos x$
Math.exp(x)	関数	指数関数 e^x
Math.floor(x)	関数	x の値を超えない整数
Math.log(x)	関数	自然対数 $\log x$
Math.max([x1, x2, ..., xN])	関数	与えられた引数のうち最大値を返す
Math.min([x1, x2, ..., xN])	関数	与えられた引数のうち最小値を返す
Math.pow(x, y)	関数	指数関数 x^y
Math.random()	関数	0 と 1 の間の擬似乱数を返す
Math.round(x)	関数	x の値を四捨五入する
Math.sin(x)	関数	正弦関数 $\sin x$
Math.sqrt(x)	関数	平方根を求める。 \sqrt{x}
Math.tan(x)	関数	正接関数 $\tan x$

- 配列の要素の最大値や最小値を求めるために Math.max や Math.min を直接使用できない。展開演算子... を用いると計算ができる。次の実行例を参照のこと。

```
>a = [3,5,23,1,4];
```

```
(5) [3, 5, 23, 1, 4]
>Math.max(a);
NaN
>Math.max(...a);
23
```

- EcmaScript 2016 ではべき乗の演算子**が導入されているので `Math.pow()` は使わなくてもよい。

2.6.3 Date オブジェクト

日付や時間に関するデータを扱うコンストラクタ関数である。引数の数や型によりいくつかの異なる日付と日時のオブジェクトが作成できる。

- 引数なしの場合は現在の日付と日時を用いて作成

```
>new Date()
Thu Oct 01 2015 13:17:46 GMT+0900 (東京 (標準時))
```

- 引数が数値の場合は世界標準時 (UTC) において 1970 年 1 月 1 日 0 時から経過時間をミリ秒単位で作成
- 引数が文字列の場合は日時を示す文字列に基づいて作成 (次の例を参照)
- 引数の数が 2 つ以上 7 つの場合は年、月、日、時、分、秒、ミリ秒の順が指定されたとして作成。ない引数は 0 と解釈される。
また、月の値は 0 が 1 月を表す。

```
>new Date(2015,10,2,10,10,20,500);
Mon Nov 02 2015 10:10:20 GMT+0900 (東京 (標準時))
```

上記の例では月の値 10 となっているので 11 月 (Nov) に設定されている。

引数の値が範囲を外れる (日に対して 0 を指定するなど) 場合でも、正しく作成される。たとえば日の値が 0 のときは指定した月の 1 日の 1 日前、つまり前の月の最終日と解釈される。

表 2.5: JavaScript における Date オブジェクトのメソッド

名称	説明
<code>getTime()</code>	日付オブジェクトのタイムスタンプを得る
<code>setTime(time)</code>	日付オブジェクトのタイムスタンプを <code>time</code> に設定する
<code>getFullYear()</code>	西暦の年の値を得る。 <code>getYear()</code> もあるが、2000 年問題の影響があるので使わないこと
<code>setFullYear(year,month,date)</code>	<code>year</code> 年 <code>month</code> + 1 月
<code>getMonth()</code>	月の値を得る。0 が 1 月を表すことに注意
<code>setMonth(month,date)</code>	<code>month</code> +1 月 <code>date</code> 日に設定する。
<code>getDate()</code>	日の値を得る。
<code>setDate(date)</code>	<code>date</code> 日に設定する。
<code>getHours()</code>	時間の設定を行う
<code>setHours(hour,min,sec,ms)</code>	<code>hour</code> 時 <code>min</code> 分 <code>sec</code> 秒 <code>ms</code> ミリ秒に設定する
<code>getMinutes()</code>	分の値を得る
<code>setMinutes(min,sec,ms)</code>	<code>min</code> 分 <code>sec</code> 秒 <code>ms</code> ミリ秒に設定する
<code>getSeconds()</code>	秒の値を得る
<code>setMinutes(sec,ms)</code>	<code>sec</code> 秒 <code>ms</code> ミリ秒に設定する
<code>getMilliseconds()</code>	ミリ秒の値を得る
<code>setMilliseconds(ms)</code>	<code>ms</code> ミリ秒に設定する
<code>getTimezoneOffset()</code>	ローカルタイムと UTC の時差を分単位で返す
<code>getDay()</code>	曜日 (0 が日曜日) を得る

課題 2.4 次の日時を求める式を答えよ。

1. 与えられた日時から 1 週間後の日時
2. 与えられた日時の翌月の 1 日
3. 与えられた日時の前の月の最終日
4. 与えられた日時の月の第 1 月曜日

2.7 第2回目復習課題

課題 2.5 次の式の評価結果を求めなさい。

式	結果	理由
<code>4+"5"</code>		
<code>4-"5"</code>		
<code>4+"ff"</code>		
<code>4+"0xff"</code>		
<code>4+parseInt("ff")</code>		
<code>4+parseInt("0xff")</code>		
<code>4+parseInt("ff",16)</code>		
<code>4+"1e1"</code>		
<code>4+parseInt("1e1")</code>		
<code>4+parseFloat("1e1")</code>		
<code>"4"*"5"</code>		
<code>"4"/"5"</code>		
<code>[] .length</code>		
<code>[[]] .length</code>		
<code>0 == "0"</code>		
<code>0 == []</code>		
<code>"0" == []</code>		
<code>![]</code>		
<code>false == []</code>		
<code>false == undefined</code>		
<code>[] == []</code>		
<code>typeof []</code>		
<code>null == undefined</code>		
<code>a=[], b=a, a==b;</code>		

第3回 関数

3.1 今回の内容

JavaScript における関数はいろいろな目的のために利用される。今回の授業では次のことについて説明する。

- 関数の定義方法と使用法
- 関数への引数の渡し方
- 関数の戻り値
- JavaScript 内での変数のスコープ
- 関数も単なるオブジェクト

さらに次の事柄についても説明する。

- 無名関数
- コールバック関数
- 自己実行可能関数
- 関数内で定義された関数
- 関数を返す関数
- クロージャ

3.2 関数の定義方法と呼び出し

3.2.1 簡単な関数の例

次の例は `sum()` という関数を定義している。

```
function sum(a,b) {  
  let c = a + b;  
  return c;  // return a + b;   でもよい。  
}
```

関数の定義は次の部分から成り立っている。

- **function** キーワード
戻り値の型を記す必要はない。
- 関数の名前
function の後にある識別名が関数の名前になる。この場合は **sum** が関数の名前になる。
- 引数のリスト
関数名の後に () 内にカンマで区切られた引数を記述する。この場合は変数 **a** と **b** が与えられている。引数はなくてもよい。
- 関数の本体であるコードブロック
{ } で囲まれた部分に関数の内容を記述する。
- **return** キーワード
関数の戻り値をこの後に記述する。戻り値がない場合には戻り値として **undefined** が返される。

次の部分はこの関数の実行例である。

```
>sum(1,2)
3
>sum(1)
NaN
>sum(1,2,3)
3
```

- 引数に 1 と 2 を与えれば期待通りの結果が得られる。
- 引数に 1 だけを与えた場合、エラーが起こらず、**NaN** となる。これは、不足している引数 (この場合には **b**) には **undefined** が渡されるためである。**1+undefined** の結果は **NaN** になる。
- 引数を多く渡してもエラーが発生しない。無視されるだけである。

これらのことから JavaScript の関数はオブジェクト指向で使われるポリモーフィズムをサポートしていない。さらに、次の問題で見るように同じ関数を定義してもエラーにならない。後の関数の定義が優先される。

課題 3.1 次に定義する関数 **sum(a,b,c)** について **sum(1,2,3)** と **sum(1,2)** を実行した結果を確認し、どちらの関数が実行されているか確認しなさい。

```
function sum(a, b){
  let c = a+b;
  return c;
}
```

```
function sum(a, b, c){  
  let d = a+b+c;  
  return d;  
}
```

3.2.2 仮引数への代入

仮引数に値を代入してもエラーとはならない。仮引数で渡された値がプリミティブなときとそうでないときとでは呼び出し元における変数の値が異なる。

実行例 3.1 次の例は呼び出した関数の中で仮引数の値を変化させたときの例である。

```
1 function func1(a){  
2   a = a*2;  
3   return 0;  
4 }  
5 function func2(a){  
6   a[0] *=2;  
7   return 0;  
8 }
```

- `func1()` では仮引数 `a` の値を 2 倍している。これを次のように実行すると、呼び出し元の変数の値には変化がないことがわかる。つまり、プリミティブデータ型を関数の引数で渡すと値そのものが渡される (値渡し)。

```
>a = 4;  
4  
>func1(a);  
0  
>a;  
4
```

- `func2()` の仮引数は配列が想定してある。この先頭の値だけ 2 倍される関数である。これに配列を渡すと、戻ってきたとき配列の先頭の値が変化している。つまり、プリミティブ型以外では仮引数の渡し方が参照渡しであることがわかる。

```
>a = [1,2,3];  
[1, 2, 3]  
>func2(a);  
0  
>a;  
[2, 2, 3]
```

3.2.3 可変引数をとる関数

前節で定義した引数の和を計算する関数において、引数の数を固定しないようにするためには仮引数に展開演算子をつける¹。

実行例 3.2 次で定義された関数は任意個数の引数を取る関数の例である。

```
function sumN(...args) {  
    let S = 0;  
    for(let i=0;i<args.length;i++) {  
        S += args[i];  
    }  
    return S;  
}
```

- 1行目で関数の仮引数 `args` に展開演算子 `...` をつけている。
- `args` は配列となるのでその大きさは `args.length` でわかる。
- これを用いて総和のプログラムが2行目から5行目に記述されている。

実行例は次のとおりである。

```
>sumN(1,2,3,4);  
10  
>sumN(1,2,3,4,5);  
15
```

課題 3.2 実行例 3.2で定義した関数 `sumN` について次の問いに答えよ。

1. `sumN()` の結果を調べよ。
2. 変数 `a` が数値からなる配列のとき、`sumN()` を用いて `a` の要素の総和を求めるにはどうすればよいか答えよ。

3.2.4 変数のスコープ

変数のスコープとはある場所で使われている変数がどこから参照できるかという概念である。JavaScript では次の特徴がある。

1. 非 `strict` モードでは変数は宣言しなくても使用できる。
2. 関数内で `let` により明示的に定義された変数はブロック内で有効となる。² ブロックとは `{ }` で囲まれた部分である。ただし、宣言よりも前では使用できない。

¹これまでの JavaScript では、関数の引数を表す配列のような性質を持つ `arguments` オブジェクトが用意されていたが、ES2015 では非推奨となった。

²変数を `var` で宣言すると関数の途中で宣言しても、関数の先頭で宣言したと同じ効果を持つ。これを変数の巻き上げという。

3. 関数の外で宣言された変数や宣言されないで使された変数はすべてグローバルとなる。

実行例 3.3 変数のスコープを次の例で確かめる。

```
1 let S = "global";
2 function func1(){
3     console.log(S);
4     return 0;
5 }
6 function func2(){
7     console.log(S);
8     let S = "local";
9     console.log(S);
10    return 0;
11 }
12 function func3(){
13     let S = "local";
14     func1();
15     return 0;
16 }
17 function func4(){
18     let S = "local in func4";
19     func5 = function(){
20         console.log(S);
21         return 0;
22     };
23     console.log(S);
24     return 0;
25 }
```

このリストは次のようになっている。

- `func1()` から `func4()` までの関数が定義されている。
- 1 行目ではグローバル変数 `S` が宣言されていて `"global"` という文字列の値に初期化されている。
- 2 行目から 5 行目で `func1()` が定義されている。
 - 3 行目で `S` の値をコンソールに出力している。
 - この関数内で変数 `S` は宣言されていないので 1 行目で定義したグローバル変数が参照される。

```
>func1();  
global  
0
```

- 6行目から11行目で `func2()` が定義されている。
 - 7行目と9行目で `S` の値をコンソールに2回出力している。
 - この関数内で変数 `S` は8行目で `let` を用いてローカル変数として宣言されている。
 - `let` で宣言された変数はブロック内で宣言される前には使うことができないので実行時にエラーが発生する。

```
>func2();  
VM362:2 Uncaught ReferenceError: S is not defined  
    at func2 (<anonymous>:2:17)  
    at <anonymous>:1:1
```

- 12行目から16行目で `func3()` が定義されている。
 - 13行目でローカル変数 `S` を定義して、初期値を `"local"` としている。
 - 14行目で初めに定義した関数 `func1()` を呼び出している。
 - `func1()` の実行の際は、もともとの関数が定義された時の変数 `S` (1行目) が参照される。

```
>func3();  
global  
0
```

- 17行目から25行目で `func4()` が定義されている。
 - 18行目でローカル変数 `S` の値を設定している。
 - 23行目の出力は18行目の値となる。
- ```
>func4();
local in func4
0
```
- 19行目では関数オブジェクトを変数 `func5` に代入している。これにより `func5()` という関数が定義される。
  - `let` による変数の宣言がないので変数 `func5` はグローバル変数となる (3参照)。
  - `func5()` 内では変数 `S` の内容を出力が定義されている。
  - `func4()` を実行した後では `func5()` が実行できる。
  - 関数 `func5()` が定義された段階での変数 `S` はこの関数が `func4()` の中で定義されているので、18行目の変数 `S` が参照される。この状況についてはクローージャの項でより詳しく説明する。

```
func5();
local in func4
0
```

**課題 3.3** 次のプログラムは何を計算するか答えよ。

```
function f(n) {
 if(n<=1) return 1;
 return n*f(n-1);
}
```

### 3.2.5 スコープチェーン

関数の中で関数を定義すると、その内側の関数内で `let` で宣言された変数のほかに、一つ上の関数で利用できる (スコープにある) 変数が利用できる。これがスコープチェーンである。例を挙げる。

```
let G1, G2;
function func1(a) {
 let b, c;
 function func2() [
 let G2, c;
 ...
]
}
```

- 関数 `func1()` ではグローバル変数である `G1` と `G2`、仮引数の `a` とローカル変数 `b` と `c` が利用できる。
- 関数 `func2()` ではグローバル変数である `G1`、`func1()` の仮引数の `a` と `func1()` のローカル変数 `b`、`func2()` のローカル変数 `G2` と `c` が利用できる。
- `func1()` で定義された `c` は `func2()` では同名の `c` が宣言されているために隠されてしまう。

このように内側で定義された関数は自分自身の中で定義されたローカル変数があるかを探し、見つからない場合には一つ上のレベルでの変数を探す。これがスコープチェーンである。JavaScript の関数のスコープは関数が定義されたときのスコープチェーンが適用される (実行例 3.3 の 14 行目参照)。これをレキシカルスコープと呼ぶ。レキシカルスコープは静的スコープとも呼ばれる。これに対して実行時にスコープが決まるものは動的スコープと呼ばれる。

## 3.3 JavaScript における関数の特徴

JavaScript 関数ではほかの言語では見られない関数の取り扱い方法がある。

### 3.3.1 関数もデータ

関数もデータ型のひとつなので、関数の定義を変数に代入することができる。課題 3.3 の 19 行目以降で関数オブジェクトを変数に代入している。代入はいつでもできるので、実行時に関数の定義を変えることも可能である。また、関数もオブジェクトなので、ある関数の戻り値として関数自体を返すことも可能である。

### 3.3.2 コールバック関数

課題 3.3 の 19 行目以降の関数オブジェクトは関数の引数として直接渡すこともできる。このとき、`function` の後には関数名がない。このような関数は無名関数と呼ばれる。HTML 文書などでは、いろいろなイベント (マウスがクリックされた、一定の時間が経過した) が発生したときに、その処理を行う関数を登録する必要がある。このように関数に引数として渡される関数のことをコールバック関数という。

**実行例 3.4** 次の例は、一定の経過時間後にある関数を呼び出す `window` オブジェクトの `setTimeout()` メソッドの使用例である。

```
1 let T = new Date();
2 window.setTimeout(
3 function callMe(){
4 let NT = new Date();
5 if(NT.getTime()-T.getTime()<10000) {
6 console.log(Math.floor((NT.getTime()-T.getTime())/1000));
7 window.setTimeout(callMe,1000);
8 }
9 },1000);
```

- 1 行目では実行開始時の時間を変数 `T` に格納している。
- このメソッドは一定時間経過後に呼び出される関数と、実行される経過時間 (単位はミリ秒) を引数に取る。
- 実行する関数は 3 行目から 9 行目で定義されている。
- この関数内で一定の条件のときはこの関数を呼び出す。この関数に名前は `callMe` である (3 行目)。<sup>3</sup>
- 4 行目で呼び出されたときの時間を求め、経過時間が 10000 ミリ秒以下であれば (5 行目)、経過時間を秒単位で表示する (6 行目)。
- さらに、自分自身を 1 秒後に呼び出す (7 行目)。

---

<sup>3</sup>`function` オブジェクトには引数を表す配列のようなオブジェクト `arguments` があり、そのプロパティ `callee` を使うのが従来の方法である。ES2015 では `arguments` が非推奨となったため、コールバック関数に名前を付けている。



なお、関数の宣言を `function foo()\{...\}` でする代わりに、`let foo = function()...\{...\}` と無名関数を利用して定義してもよい。

`setTimeout()` メソッドのコールバック関数に引数の値を渡したいときは、`setTimeout()` メソッドの 3 番目以降の引数に記述する。

**課題 3.4** 次のリストは上記の例のコールバック関数を引数付にしたものである。

```
let T = new Date();
window.setTimeout(
 function callMe(L){
 let NT = new Date();
 if(NT.getTime()-T.getTime()<10000) {
 console.log(Math.floor((NT.getTime()-T.getTime())/1000));
 L +=1000;
 window.setTimeout(callMe,L,L);
 }
 },1000,1000);
```

どのような動作をするか確認しなさい。

### 3.3.3 自己実行関数

関数を定義してその場で直ちに実行することができる。たとえば次のコードを考える。これは課題 1.2 の関数の中身である。

```
let i;
for(i=1;i<10;i++) {
 console.log(`${i} ${i*i}`);
}
```

このプログラムを実行すると 1 から 9 までの値とその 2 乗の値がコンソールに出力される。実行後に、コンソールに `i` と入力すれば 10 が出力される。

一方、課題 1.2 では関数が定義されただけで何も出力されないで、`foo()` と入力して関数を実行する必要がある。この場合、変数 `i` は関数内のローカル変数なので関数実行後、変数 `i` の値は参照できない (`undefined` が出力される)。

前者の場合は変数 `i`、後者の場合は関数名 `foo` がグローバル変数が残ってしまっている。これを避けるためには定義した関数をその場で実行できる機能があればよい。これを実行するためには次のように記述する。

```
(function(){
 let i;
 for(i=1;i<10;i++) {
 console.log(`${i} ${i*i}`);
 })();
```

この様に関数の定義を全体で `()` で囲み、そのあとに関数の呼び出しを示すための `()` を付ける。

この技法は、初期化の段階で1回しか実行しない事柄を記述し、かつグローバルな空間を汚さない(余計な変数などを残さない)手段として用いられる。

なお、ES2015 で導入された `let` で変数 `i` を `for` 内で宣言すると、その変数は `for` 文内でしか存在しない。

```
for(let i=1;i<10;i++) {
 console.log(`${i} ${i*i}`);
}
```

### 3.4 クロージャ

関数内部で宣言された変数は、その外側から参照することができない。つまり、その関数は関数内のローカル変数を閉じ込めている。しかし、関数内部で定義された関数を外部に持ち出す(グローバルな関数にする)と、持ち出された関数のスコープチェーン内に定義された親の関数のスコープを引き継いでいることから、親の関数のローカル変数の参照が可能となる。

このように関数とその依存する環境(変数や呼び出せる関数などのリスト)を合わせたものをその関数のクロージャと呼ぶ。

#### 3.4.1 クロージャの例 – 変数を隠す

ここで上げる例は実用に乏しいと思われるかもしれないが、この後に出てくるオブジェクトの項ではより実用的なものと理解できるであろう。

**実行例 3.5** 次の関数を考える。

```
1 function f1() {
2 let n=0;
3 return function() {
4 return n++;
5 };
6 }
```

- ローカル変数 `n` を定義し、初期値を `0` としている (2 行目)。
- 戻り値はローカル変数 `n` の値を返す関数である (3 行目から 5 行目)。
- この戻り値の関数はローカル変数の値を `1` 増加させている (5 行目)。

これを次のように実行する。

- 変数 `ff` に関数 `f1()` で返される関数オブジェクトを代入する。

```
>ff= f1();
() {
 return n++;
}
```

これにより変数 **ff** は関数オブジェクトになる。

- **f1()** 内のローカル変数は参照できない。

```
>n;
VM351:2 Uncaught ReferenceError: n is not defined(...)
```

- 戻り値の関数は変数 **ff** を用いて参照できる。

```
>ff();
0
>ff();
1
>ff();
2
```

何回か実行すると戻り値が順に増加している。つまり、ローカルには変数 **n** が存在している。

- もう一度関数 **f1()** を実行すると新しい関数を得られる。

```
>ff2=f1();
() {
 return n++;
}
>ff();
2
>ff2();
0
```

### 3.4.2 クロージャの例 – 無名関数をその場で実行する

例 3.5で定義された関数 **f1** を一度だけ実行して、それがこれ以上実行されないようにするためにはこの関数を無名関数としてその場で実行すればよい。

**実行例 3.6** 次のリストは、使い捨ての関数の例である。このような方法を用いるとオブジェクトに共通の変数を持つことができる。

```
let foo = (function () {
 let n=0;
 return function() {
```

```

 return n++;
};
})();

```

- 無名関数を定義した部分を () でくくり、引数リストをその後の () に記述する。ここでは、引数がないので中はない。
- 戻された関数オブジェクトを変数 `foo` に代入する。
- 前と同じように実行できる。

```

>foo();
0
>foo();
1
>foo();
2

```

**実行例 3.7** 次の例は関数内のローカル変数の値を単純に返すと、不都合が起こる例である<sup>4</sup>

```

1 function f2() {
2 let a = [];
3 let i;
4 for(i=0; i<3; i++) {
5 a[i] = function() {
6 return i;
7 };
8 }
9 return a;
10 }

```

- 変数 `a` を空の配列で初期化する (2 行目)。
- この配列に配列の添え字の値を返す関数を定義する (4 行目から 8 行目)
- 配列全体を戻り値として返す (9 行目)。

これを実行すると次のようになる<sup>5</sup>。

```

>funcs=f2()
[function a.(anonymous function)(), function a.(anonymous function)(),
 function a.(anonymous function)()]
>funcs[1]()
3

```

<sup>4</sup>この例は Stoyan Stefanov(水野貴明、渋川よしき訳) オブジェクト指向 JavaScript、アスキーメディアワークス、2012 年の 103 ページ以降から取りました。

<sup>5</sup>ここでの出力例は Chrome によるものである。ブラウザによっては出力が異なる。

これは、`f2()` を実行した後では、変数 `i` の値が 3 となっており、それぞれの関数が実行されるときにはこの値が参照されるためである。

**実行例 3.8** この不具合は、`for` の制御変数 `i` を `for` 文の初期化のところで宣言すれば発生しない<sup>6</sup>。

```
function f2() {
 let a = [];
 for(let i=0; i<3; i++) {
 a[i] = function() {
 return i;
 };
 }
 return a;
}
```

これを実行すると、期待した結果が得られる。

```
>funcs2 = f3();
[function anonymous(), function anonymous(), function anonymous()]
>funcs2[0]();
0
>funcs2[1]();
1
```

この形で `let` による変数の宣言を `var` に変えると変数の巻き上げにより正しく動作しない。

正しく動作させるためには値を関数の引数に (値渡しで) 渡すことで、スコープチェーンを切ればよい。

```
1 function f3() {
2 let a = [];
3 let i;
4 for(i=0; i<3; i++) {
5 a[i] = (function(x){
6 return function() {
7 return x;
8 }
9 })(i);
10 };
11 return a;
12 }
```

- 5 行目から 9 行目で、その場で実行される、引数を取る無名関数を用意している。

---

<sup>6</sup>ES2015 では `for` 文の初期化内で `let` で宣言された変数のスコープ規則が別に定められている。

- この戻り値は、5 行目の無名関数の引数の値を返す無名関数 (6 行目から 8 行目) である。
- 仮引数には、実行されたときの `i` のコピーが渡されるので、その後、もとの変数の値が変わっても呼び出されたときの値が仮引数の `x` に保持される。

**課題 3.5** 実行例 3.7において、`for` 文の代わりに `while` 文に書き換えたときに正しく動作するようにせよ。ただし、直前のように即時実行関数を用いなくて実現すること。

## 第4回 オブジェクト

### 4.1 配列とオブジェクト

配列はいくつかのデータをまとめて一つの変数に格納している。各データを利用するためには `foo[1]` のように数による添え字を使う。これに対し、オブジェクトでは添え字に任意の文字列を使うことができる。

**実行例 4.1** 次の例はあるオブジェクトを定義してその各データにアクセスする方法を示している。

```
let person = {
 name : "foo",
 birthday : {
 year : 2001,
 month : 4,
 day : 1
 },
 "hometown" : "神奈川",
}
```

- オブジェクトは全体を `{ }` で囲む。
- 各要素はキーと値の組で表される。両者の間は `:` で区切る。
- キーは任意の文字列でよい。
- キーの文字列が変数名の約束事に合わないときはキー全体を `" "` で囲う必要がある。
- 値は JavaScript で取り扱えるデータなあらば何でもよい。上の例ではキー `birthday` の値がまたオブジェクトとなっている。値が関数であるキーはそのオブジェクトのメソッドと呼ばれる。
- 各要素の値を取り出す方法は2通りある。

— `.` 演算子を用いてオブジェクトのキーをそのあとに書く。

```
>person.name;
"foo"
```

— 配列と同様に `[]` 内にキーを文字列として指定する。

```
>person["name"];
"foo"
```

- オブジェクトの中にあるキーをすべて網羅するようなループを書く場合や変数名として利用できないキーを参照する場合には後者の方法が利用される。
- キーの値が再びオブジェクトであれば、前と同様の方法で値を取り出せる。

```
>person.birthday;
Object {year: 2001, month: 4, day: 1}
>person.birthday.year;
2001
>person.birthday["year"];
2001
```

この例のように取り出し方は混在してもよい。

- キーの値は代入して変更できる。

```
>person.hometown;
"神奈川"
>person.hometown="北海道";
"北海道"
>person.hometown;
"北海道"
```

- 存在しないキーを指定すると値として `undefined` が返る。

```
>person.mother;
undefined
```

- 存在しないキーに値を代入すると、キーが自動で生成される。

```
>person.mother = "aaa";
"aaa"
>person.mother;
"aaa"
```

- オブジェクトのキーをすべて渡るループは `for-in` で実現できる。
  - `for( v in obj )` の形で使用する。変数 `v` はループ内でキーの値が代入される変数、`obj` はキーが走査されるオブジェクトである。<sup>1</sup>
  - キーの値は `obj[v]` で得られる。

---

<sup>1</sup>1ページの [1162 ページ] にはオブジェクトの独自プロパティだけを見たい場合には `hasOwnProperty()` を使うことが推奨されている。



```
>for(i in person) { console.log('i ${person[i]}');};
name foo
birthday [object Object]
hometown 北海道
mother aaa
undefined
```

最後の `undefined` は `for` ループの戻り値である。

オブジェクトを`{}`の形式で表したものをオブジェクトリテラルとよぶ。

**課題 4.1** `window` オブジェクトにはどのようなプロパティがあるか調べよ。2 つ以上のブラウザで実行し、比較すること。

## 4.2 オブジェクトリテラルと JSON

JSON(JavaScript Object Notation) はデータ交換のための軽量なフォーマットである。形式は JavaScript のオブジェクトリテラルの記述法と全く同じである。

- 正しく書かれた JSON フォーマットの文字列をブラウザとサーバーの間でデータ交換の手段として利用できる。
- JavaScript 内で、JSON フォーマットの文字列を JavaScript のオブジェクトに変換できる。
- JavaScript 内のオブジェクトを JSON 形式の文字列に変換できる。

JavaScript のオブジェクトと JSON フォーマットの文字列の相互変換の手段を提供するのが **JSON** オブジェクトである。

**実行例 4.2** 次の例は 2 つの同じ形式からなるオブジェクトを通常の配列に入れたものを定義している。

```
var persons = [{
 name : "foo",
 birthday :{ year : 2001, month : 4, day : 1},
 "hometown" : "神奈川",
},
{
 name : "Foo",
 birthday :{ year : 2010, month : 5, day : 5},
 "hometown" : "北海道",
}];
```

次の例はこのオブジェクトを **JSON** に処理させたものである。

```
>s = JSON.stringify(persons);
"[{"name":"foo","birthday":{"year":2001,"month":4,"day":1},
"hometown":"神奈川県"},
{"name":"Foo","birthday":{"year":2010,"month":5,"day":5},
"hometown":"北海道"}]"
>s2 = JSON.stringify(persons,["name","hometown"]);
"[{"name":"foo","hometown":"神奈川県"},{"name":"Foo","hometown":"北海道"}]"
>o = JSON.parse(s2);
[Object, Object]
>o[0];
Object {name: "foo", hometown: "神奈川県"}
```

- JavaScript のオブジェクトを文字列に変更する方法は `JSON.stringify()` を用いる。このまま見ると"がおかしいように見えるが表示の関係でそうになっているだけである。なお、結果は途中で改行を入れているが実際は一つの文字列となっている。
- `JSON.stringify()` の二つ目の引数として対象のオブジェクトのキーの配列を与えることができる。このときは、指定されたキーのみが文字列に変換される。
- ここでは、`"name"` と `"hometown"` が指定されているので `"birthday"` のデータは変換されていない。
- JSON データを JavaScript のオブジェクトに変換するための方法は `JSON.parse()` を用いる。
- ここではオブジェクトの配列に変換されたことがわかる。
- 各配列の要素が正しく変換されていることがわかる。

課題 4.2 実行例 4.2において、

```
s3 = JSON.stringify(persons,["year"]);
```

としたときの結果はどうなるか調べなさい。

## 4.3 クラス

### 4.3.1 クラスの宣言とインスタンスの生成

前節で同じキーを持つオブジェクトを複数作成したい場合に同じようなコードを繰り返すのは問題である。また、キーの追加をするときの修正の手間がかかりすぎてプログラムのメンテナンスが面倒になる。

そこで、オブジェクトのひな形を作成し、それをもとにオブジェクトを構成することが考えられる。このオブジェクトのひな形は通常クラスと呼ばれる。ES2015 ではクラスの宣言ができる。クラスから作成されたオブジェクトはこのクラスのインスタンスと呼ばれる。

前節の例をクラスを用いて書き直すと次のようになる。

```
1 class Person{
2 constructor(name, year, month, day, hometown="神奈川県"){
3 this.name = "foo";
4 this.birthday = {
5 year : year,
6 month : month,
7 day : day
8 };
9 this["hometown"] = hometown;
10 }
```

- 1行目がクラスの宣言である。キーワード **class** の後にクラス名 **Person** を記述する。通常、クラス名は大文字で始まる。関数の時と同様にクラスの宣言を変数に代入することもできる。
- その後の{と}内にクラスの記述を行う。
- クラスの定義には初期化を行う **constructor()** が必須である。ここでは5つの引数を取るコンストラクタが定義されている。5番目の引数はデフォルトの値が指定されている。
- キーワード **this** はクラスから作成されたインスタンスを指す。
- 3行目から9行目で、そのインスタンスのオブジェクトのキーはプロパティと呼ばれる。プロパティの名前は前のオブジェクトと同じである。
- 9行目の定義はコンストラクタの最後の引数はデフォルトの値が与えられている。
- クラス内のコードは **strict** モードで実行される。

```
>p = new Person("foo",1900,4,1);
Person {name: "foo", birthday: {…}, hometown: "神奈川県"}
```

クラスの定義からインスタンスを作成するためにはキーワード **new** をつけてクラスを呼び出す。上の例ではコンストラクタが引数を必要としているので次のようになる。

```
>p = new Person("foo",2001,8,18);
Person {name: "foo", birthday: {…}, hometown: "神奈川県"}
>p.birthday
{year: 2001, month: 8, day: 18}
>p.hometown;
"神奈川県"
```

コンストラクタの最後の引数にはデフォルト値が指定されているので、この実行例のように値が指定されていない場合にはデフォルト値に設定されていることが分かる。

**課題 4.3** 実行例 4.1と同様に、**class** を用いて作成されたインスタンスについてプロパティの値の変更、プロパティの追加ができるか確認しなさい。

### 4.3.2 クラスメソッド

クラス内では **constructor** のほかにメソッドと呼ばれる関数で定義されたプロパティがある。また、メソッドにはアクセッサプロパティと呼ばれるオブジェクトに値を渡すセッター、値を得るゲッターの2種類を指定することもできる。ES2015 ではゲッターやセッターにはキーワード **get** と **set** を用いる。

**実行例 4.3** 次のリストは **Person** に文字列に変換する **toString()** と、実行時における年令を返すゲッター **age** を追加したものである。

```
1 class Person{
2 constructor(name, year, month, day, hometown = "神奈川"){
3 this.name = "foo";
4 this.birthday = {
5 year : year,
6 month : month,
7 day : day
8 };
9 this["hometown"] = hometown;
10 }
11 toString() {
12 return `${this.name}さんは‘+
13 `${this.birthday.year}年${this.birthday.month}月${this.birthday.day}日に’ +
14 `${this.hometown}で生まれました。’;
15 }
16 get age() {
17 let today = new Date();
18 let age = today.getFullYear() - this.birthday.year;
19 if(today.getTime() <
20 new Date(today.getFullYear(),
21 this.birthday.month-1,
22 this.birthday.day).getTime()) age--;
23 return age;
24 }
25 }
```

11 行目から 15 行目でメソッド **toString()** が定義されている。

一般に **toString()** メソッドはすべてのオブジェクトに定義されていて、文字列が必要な時に呼び出される。ここでは「~ さんは~ 年~ 月~ 日に~ で生まれました。」という表示を返す。

実行例は次のとおりである。文字列が必要なときに呼び出されることがわかる。

```
>p.toString();//toString() の明示的呼び出し
"2001 年 4 月 1 日}"
```

```
>`${p}` //暗黙の toString() 呼び出し
"2001 年 4 月 1 日}"
```

16 行目から 24 行目でゲッター `age` が定義されている。

- 16 行目でキーワード `get` をつけて、ゲッター `age()` を宣言している。メソッドは関数なので `()` をつける必要がある。ただし、ゲッターに仮引数をつけることはできない。
- 17 行目でアクセス時の時間を変数 `today` に保存している。
- 18 行目でアクセス時の年から誕生日の年を引いている。
- 19 行目から 22 行目で、今年の誕生日が過ぎているかどうかをチェックしている。アクセス時の年と誕生日の月を日をもとに日付を作成し、その時間 (`getTime()` を利用) を比較することでチェックができる。
- 誕生日前ならば 18 行目で求めた値を 1 減少させる。

実行例は次のとおりである。

```
>p.age;
16
>p.age = 50;
50
>p.age;
16
```

- 定義の方には `()` があるが、つけるとエラーになる。記述の方法はプロパティと同じようになる。
- 代入の式はエラーがなく実行できるが、ゲッターの機能は変わらない。

```
p.age(10);
VM87:1 Uncaught TypeError: p.age is not a function
 at <anonymous>:1:3
```

**課題 4.4** 指定された日付における年令を求めるメソッドを作成しなさい。求める年齢は次の条件を満たすこと。

- 引数がない場合には `age` と同じように実行時
- 年しかない場合にはその年の 1 月 1 日現在
- 年と月しかない場合にはその年月の 1 日現在

**課題 4.5** 実行例 4.3 のリストにあるプロトタイプメソッド `age()` の前にある `get` を省略して実行する (通常の方法) とどうなるか報告せよ。

### 4.3.3 静的メソッド

クラスに対して、インスタンスを作成しないで使用できる静的メソッドを定義できる。キーワード `static` を付けたものが静的メソッドとなる。インスタンス化されたオブジェクトからは使用できない。

`Math` オブジェクトにある各関数が `Math` オブジェクトの静的メソッドの例になっている。

### 4.3.4 継承

すでにあるクラスをもとに機能の追加や変更を加えた新しいクラスを作ることができる。新しいクラスはもとなるクラスを継承しているという。もとのクラスは新しいクラスのスーパークラス、新しいクラスはもとなるクラスのサブクラスという。JavaScript では複数のクラスから同時に継承する多重継承はサポートされていない。

**実行例 4.4** 次の例は実行例 4.3のクラス `Person` を継承して新しいクラス `Student` を作成するものである。

```
1 class Student extends Person {
2 constructor(name, id, year, month, day, hometown) {
3 super(name, year, month, day, hometown);
4 this.id = id;
5 }
6 }
```

- クラスを継承するためにはクラス名の後に、キーワード `extends` を付けて継承するクラス名を書く (1行目)。
- `Student` クラスのコンストラクタには `Person` クラスで必要なパラメタのほかに `id` が付け加えている (2行目)。
- 親クラス (スーパークラス) のコンストラクタを呼び出すために `super()` を実行する (3行目)
- 4行目で追加のプロパティの設定を行っている。

実行例は次のとおりである。

```
>s = new Student("foo",1523999,2001,4,1);//Person のデフォルト引数値が設定されている
Student {name: "foo", birthday: {...}, hometown: "神奈川", id: 1523999}
> ‘${s}’ //toString() も Person で定義されたものが使われる
"foo さんは 2001 年 4 月 1 日に}神奈川で生まれました。 "
>s2 = new Student("foo",1523999,2001,4,1,"厚木");//デフォルト以外の設定もできる
Student {name: "foo", birthday: {...}, hometown: "厚木", id: 1523999}
>‘${s2}’;
"foo さんは 2001 年 4 月 1 日に}厚木で生まれました。 "
```

**課題 4.6** `Student` クラスにプロパティ `id` も表記するような `toString()` メソッドを定義すると、`Person` の `toString()` が上書きされることを確認しなさい。

### 4.3.5 instanceof 演算子

`instanceof` 演算子はオブジェクトを生成したコンストラクタ関数が指定されたものかを判定できる。

**実行例 4.5** 次の結果は Chrome で、実行例 4.3の例である。

```
>p instanceof Person
true
>p instanceof Object;
true
>p instanceof Date;
false
```

`Person` オブジェクトが `Object` を継承しているために `o instanceof Object` が `true` となっている。

### 4.3.6 同じクラスのインスタンスに通し番号をつける

実行例 4.4のプロパティ `id` を重複がなくかつ連続な値に設定しようとするクラスにおいて管理する変数が必要となる。オブジェクト指向言語ではクラスのインスタンスで共通に持つ変数をクラス変数という仕組みがあるが、JavaScript には直接的には存在しない。これを実現するためにはクラスをクロージャの中で定義して、その中に変数を置けばよい。

**課題 4.7** 次の例は即時実行関数内に `id` を管理する変数を用意して、その関数がクラス式を返すようにしたものである。

```
1 const Student = (function(){
2 let id = 10000;
3 return class extends Person {
4 constructor(name, year, month, day, hometown) {
5 super(name, year, month, day, hometown);
6 this.id = id++;
7 }
8 }
9 })();
```

- 即時実行関数の戻り値はクラス式であり、それを変数 `Student` に代入している。同じ名前のクラスが再定義されないようにするため変数を `const` で宣言している。

- 2行目で次に設定する `id` を保存する変数を初期化している。
- 関数の戻り値としてクラス式を返す。クラスの定義は以前のものとほとんど同じである。
- 異なるのはインスタンスに設定した後で `id` を管理する変数を増加させているところ (6行目の右辺)。

実行例は次のとおりである。

```
>s = new Student("foo",2001,4,1); //初めのインスタンスの id は 10000
{name: "foo", birthday: {...}, hometown: "神奈川", id: 10000}
>'${s}';
"fooさんは2001年4月1日に}神奈川で生まれました。"
>s2 = new Student("foo",2001,4,1,"厚木"); //次のインスタンスの id は 10001
{name: "foo", birthday: {...}, hometown: "厚木", id: 10001}
>'${s2}';
"fooさんは2001年4月1日に}厚木で生まれました。"
```



## 第5回 オブジェクトのプロトタイプと継承

### 5.1 オブジェクト属性

JavaScript の関数オブジェクトには `prototype`、`class` と `extensible` という 3 つの属性がある。`prototype` 属性はオブジェクトの継承に関係するので、最後に説明をする。

#### 5.1.1 class 属性

オブジェクトの `class` 属性はオブジェクトの型情報を表す文字列である。最新の ECMAScript 5 でもこの属性を設定する方法はない。直接値を取得する方法もない。クラス属性は間接的にしか得られない。組み込みのコンストラクタで生成されたオブジェクトではそのクラス名が間接的に得られるが、独自のコンストラクタ関数では、`"Object"` しか得られない。

**実行例 5.1** 実行例 4.1 の例で確認する。

`Object` から継承した `toString()` を直接呼び出すと次のような結果になる。

```
>p = new Person();
Person {name: "foo", birthday: Object, hometown: "神奈川"}
>p.toString();
"[object Object]"
>p + "";
"[object Object]"
```

ここで、`Person()` コンストラクタには `toString()` メソッドが定義されていないので、もともとの `Object` で定義されている `toString()` が呼び出されている。`Array()` オブジェクトには `toString()` が定義 (オーバーライド) されているので次のような形にしないとこのような結果が得られない。

```
>Object.prototype.toString.call([]);
"[object Array]"
```

これは、`Object` の `prototype` 属性に定義されている `toString` を引数のメソッドとして利用することを意味する。`prototype` の意味は後で説明をする。同じように他のオブジェクトで実行すると次のようになる (Opera で実行)。

```
>Object.prototype.toString.call(null);
"[object Null]"
>Object.prototype.toString.call(undefined);
```

```
"[object Undefined]"
>Object.prototype.toString.call(NaN);
"[object Number]"
>Object.prototype.toString.call(window);
"[object global]"
>window+" "
"[object Window]"
```

なお、FireFox や Internet Explorer では最後の例は次のようになる。

```
>Object.prototype.toString.call(window);
"[object Window]"
>window+" "
"[object Window]"
```

### 5.1.2 extensible 属性

`extensible` はオブジェクトに対してプロパティの追加ができるかどうかを指定する。ECMAScript 5 ではこの属性の取得や設定ができる関数が用意されている。

この属性の取得は `Object.isExtensible()` に調べたいオブジェクトを引数にして渡す。オブジェクトのプロパティを拡張できなくするためには `Object.preventExtension()` に引数として設定したいオブジェクトを渡す。

**実行例 5.2** 実行例 4.1の例で確認する。

```
>p = new Person();
Person {name: "foo", birthday: Object, hometown: "神奈川"}
>p.mother = "aaa"
"aaa"
>Object.preventExtensions(p);
Person {name: "foo", birthday: Object, hometown: "神奈川", mother: "aaa"}
>p.grandmother = "AAA";
"AAA"
>p.grandmother;
undefined
>p.mother = "bbb";
"bbb"
>p.mother;
"bbb"
>delete p.mother;
true
>p.mother
undefined
```

- `Object.preventExtensions(p)` を実行する前では存在しない属性の追加ができています (`p.mother`)。
- 設定後は、新しい属性が定義できていない (`p.grandmother`)。
- `Object.preventExtensions(p)` では属性の削除までは禁止できない (`delete p.mother` が成功している)。

属性の削除まで禁止したい場合には `Object.seal()` を用いる。一度この関数を実行されたオブジェクトは解除できない。また、すでにその状態になっているかどうかは、`Object.isSealed()` を用いる。書き込み可の属性の値は変えることができる。

**実行例 5.3** 実行例 4.1の例で確認する。

```
>p = new Person();
Person {name: "foo", birthday: Object, hometown: "神奈川県"}
>Object.isSealed(p);
false
>Object.seal(p);
Person {name: "foo", birthday: Object, hometown: "神奈川県"}
>Object.isSealed(p);
true
>p.mother = "aaa";
"aaa"
>p.mother;
undefined
>p.hometown
"神奈川県"
>p.hometown = "Japan"
"Japan"
>p.hometown
"Japan"
```

- `Object.seal()` を実行した後、存在しない属性 `p.mother` は定義されていないことがわかる。
- 既存の属性の値は変更可能である。

最も強く、オブジェクトを拘束するためには、`Object.freeze()` を用いる。また、この状態を確認するためには `Object.isFrozen()` を用いる。

**実行例 5.4** 実行例 5.3に引き続いて `Object.freeze()` を実行した結果である

```
>Object.isFrozen(p);
false
>Object.freeze(p);
Person {name: "foo", birthday: Object, hometown: "Japan"}
```

```
>p.hometown = "Tokyo"
"Tokyo"
>p.hometown;
"Japan"
```

`p.hometown` の値が設定できていないことがわかる。このような状態で属性値を変えたい場合には、属性にたいするセッターメソッドを定義することになる。

なお、`Object.seal()` や `Object.freeze()` の影響は、渡されたオブジェクト自身の属性にしか影響を及ぼさない。継承元のオブジェクトには影響を及ぼさない。

### 5.1.3 prototype 属性

オブジェクトの `prototype` 属性の値は、同じコンストラクタ関数で生成された間で共通のものになっている。オブジェクトリテラルで生成されたオブジェクトは `Object.prototype` で参照できる。`new` を用いて生成されたオブジェクトはそのコンストラクタ関数の `prototype` を参照する。このコンストラクタ関数の `prototype` もオブジェクトであるから、その `prototype` も存在する。この一連の `prototype` オブジェクトをプロトタイプチェーンとよぶ。

ECMAScript 5 で定義されている `Object.create()` メソッドは引数で与えられたオブジェクトの `prototype` を `prototype` に持つオブジェクトを生成する。このメソッドに対して2番目の引数を与えて、新たに生成されたオブジェクトのプロパティを指定できる。

### 5.1.4 prototype の使用例

**実行例 5.5** 次の例は4.1の属性を簡略し、いくつかのメソッドを追加したものである。

```
1 function Person(){
2 this.name = "foo";
3 this.year = 2001;
4 this.month = 4;
5 this.day = 1;
6 this.toString = function(){
7 return "私の名前は"+this.name+"です";
8 }
9 this.showBirthday = function() {
10 return this.name+"の誕生日は"+
11 this.year+"年"+this.month+"月"+this.day+"日です";
12 }
13 }
```

- 6行目から8行目ではオブジェクトを文字列に変換する必要ができたときに呼び出される `toString()` メソッドが定義されている。

- 9行目から12行目では誕生日をわかりやすい形で表示する `showBirthday()` メソッドを定義している。

実行例は次のようになる。

```
>p=new Person();
Person {name: "foo", year: 2001, month: 4, day: 1}
>p +"";
"私の名前はfoo です"
>p.showBirthday();
"foo の誕生日は2001 年 4 月 1 日です"
```

このコードをみるとコンストラクタ関数が呼ばれるごとに、2つのメソッドのコードが繰り返し使用されていることがわかる。

このように同じコンストラクタ関数から生成されるオブジェクトに対して共通するプロパティやメソッドはコンストラクタ関数の `prototype` に移動した方が効率が良い。

**実行例 5.6** 次のコードは、実行例 5.5のメソッドを `prototype` に移動し、さらにプロパティとして書き直したものである。

```
1 function Person2(name, y, m, d){
2 this.name = name;
3 this.year = y,
4 this.month = m,
5 this.day = d
6 }
7 Person2.prototype = {
8 toString : function(){
9 return "私の名前は"+this.name+"です";
10 },
11 get age(){
12 var Now = new Date();
13 var Age = Now.getFullYear() - this.year;
14 if((Now.getMonth()+1) < this.month) {
15 Age--;
16 } else {
17 if((Now.getMonth()+1) == this.month &&
18 Now.getDate() < this.day) Age--;
19 }
20 return Age;
21 },
22 get birthday() {
23 return this.year+"年"+this.month+"月"+this.day+"日";
```

```

24 }
25 }

```

- `prototype` はオブジェクトなのでオブジェクトリテラルの形式で定義する。
- 8行目から10行目では `toString()` メソッドを定義している。
- 11行目から21行目では、`age` プロパティを定義している。`function` キーワードの代わりに `get` を用いることで、ゲッターとして定義していることになる。

この関数は実行時におけるオブジェクトの満年齢を得ることができる。

- 12行目で実行時の日時を求めている。
- 13行目で実行時の年から、誕生日の年の差を求めている。
- 14行目では実行時の月と誕生日の月を比較し、誕生日の月を過ぎていなければ、年齢を1だけ減らす。

`Date.getMonth()` メソッドは0(1月)から11(12月)の値を返すことに注意する。

- 17行目では実行時の月と誕生日の月が同じ時に、両者の日を比較して、誕生日前か判定している。

- 22行目から24行目では `birthday` プロパティをゲッターとして定義している。

実行結果は次のとおりである。

```

>p2 = new Person2("me",1995,4,1);
Person2 {name: "me", year: 1995, month: 4, day: 1}
>p2.birthday;
"1995 年 4 月 1 日"
>p2.age;
20
>p.age=30;
30
>p2.age;
20
>Person2.prototype.constructor
Object() { [native code] }

```

- `p.age` にはセッターが定義されていないので見かけ上の代入を行っても無効であることがわかる。
- `Person2.prototype.constructor` の結果が `Person2` になっていないのは、7行目で `Person2.prototype` に代入しているオブジェクトに `constructor` プロパティがないからである。通常はこれには問題があるので代入するオブジェクト内に次のような記述を追加するのがよい。

```

constructor : Person2,

```

別の方法としては、`Person2.prototype.constructor = Person2;` の行を追加する方法もある。

なお、`get` キーワードの代わりに `set` キーワードを用いるとセッターを定義できる。

**課題 5.1** 実行例 5.6において、`age` プロパティがセッターとして使われたときには注意を促すメッセージを表示するようにしなさい。

## 5.2 継承

オブジェクトの継承とは既に存在するオブジェクト (クラス) に対して機能の追加や修正を行って新しいオブジェクト (クラス) を構成することである。JavaScript ではクラスをサポートしていないので厳密な意味での継承はできないが、`prototype` を用いることでメソッドの継承が可能となっている。

**実行例 5.7** 次のリストは実行例 5.6の `Person2` を継承して、学籍番号を追加のプロパティとする `Student` オブジェクトのコンストラクタ関数である。

```
1 function Student(n, id, y, m, d){
2 this.name = n;
3 this.year = y;
4 this.month = m;
5 this.day = d;
6 this.id = id;
7 }
8 Student.prototype = new Person2();
9 Student.prototype.constructor = Student;
```

- `name` などのプロパティはオブジェクトごとに違う値をとるので `Person2.prototype` 内には置くことができない。したがって、それぞれを `this` のプロパティに格納する (2行目から6行目)。
- `Person2` の `prototype` を利用するために、`Student.prototype` にオブジェクトを新規に作成して代入する (8行目)。これにより、この後で `Person2` のプロパティが変更されても、`Student` オブジェクトには影響がない。
- `Student.prototype.constructor` を `Student` に戻しておく (9行目)

実行結果は次のとおりである。

```
>s = new Student("me",1323300,1995,4,1)
Student {name: "me", year: 1995, month: 4, day: 1, id: 1323300}
>s.age;
20
```

表 5.1: エラーオブジェクトのプロパティ

| プロパティ                | 説明                                         |
|----------------------|--------------------------------------------|
| <code>message</code> | エラーに関する詳細なメッセージ。コンストラクタで渡された文字列か、デフォルトの文字列 |
| <code>name</code>    | エラーの名前。エラーを作成したコンストラクタ名になる                 |

```
>s.name
"me"
>s+"";
"私の名前は me です"
>s.constructor;
Student(n, id, y, m, d){
 this.name = n;
 this.year = y;
 this.month = m;
 this.day = d;
 this.id = id;
}
```

`Person2` で定義されたメソッドが利用できていることがわかる。

**課題 5.2** 実行例 5.7 の `Student` に所属学部を表示するメソッドを追加しなさい。所属学部のプロパティは追加しないで学籍番号から求めること。

## 5.3 エラーオブジェクトについて

エラーオブジェクトとはエラーが発生したことを知らせるオブジェクトである。通常は計算の継続ができなくなったときにエラーオブジェクトをシステムに送る操作が必要である。これを通常、エラーを投げる (`throw` する) という。エラーオブジェクトには表 5.1 のようなプロパティがある。

### 5.3.1 エラー処理の例

エラー処理の例をいくつか挙げる。

**実行例 5.8** 次のリストは、実行例 5.6 において、コンストラクタに与えられた引数をチェックして不正な値の場合にはエラーを投げるように書き直したものである。

なお、このリストでは `Person2` の `age` の内容が以前のものと同一なので ... で省略している。



```
1 function Person(name, y, m, d){
2 if(name === "") throw new Error("名前がありません");
3 this.name = name;
4 this.year = y;
5 if(m<1 || m>12) throw new Error("月が不正です");
6 var date = new Date(y,m,0);
7 if(d<1 || d>date.getDate()) throw new Error("日が不正です");
8 this.month = m,
9 this.day = d
10 }
11 Person.prototype = {
12 toString : function(){
13 return "私の名前は"+this.name+"です";
14 },
15 get age(){
16 ... // 内容は省略
17 },
18 get birthday() {
19 return this.year+"年"+this.month+"月"+this.day+"日";
20 }
21 }
```

- 2行目で、**name** が空文字であればエラーを発生させている。
- 5行目では月の値の範囲をチェックしている。
- 6行目では、与えられた年と月からその月の最終の日を求めている。**Date.getMonth()** の戻り値が 0(1月) から 11(12月) になっているので、**new Date(y,m,0)** により翌月の1日の1日前、つまり、問題としている月の最終日が設定できる<sup>1</sup>。
- 7行目で与えられた範囲に日が含まれていなければエラーを発生させている。

これをいくつかのデータで実行した結果は次のようになる。

- 通常の日時ならば問題なく、オブジェクトが構成される。

```
>p = new Person("foo",1995,4,1);
Person {name: "foo", year: 1995, month: 4, day: 1}
```
- 1996年はうるう年なので2月29日が存在する。したがって、エラーは起こらず正しくオブジェクトが作成できる。

```
>p = new Person("foo",1996,2,29);
Person {name: "foo", year: 1996, month: 2, day: 29}
```

---

<sup>1</sup>課題 2.4 の 3 番目の問題の解答である。

- 1995 年はうるう年ではないので 2 月 29 日がない。したがって、エラーが起きる。

```
>p = new Person("foo",1995,2,29);
Uncaught Error: 日が不正です (…)
```

- 不正な月や日では当然、エラーが起こる。

```
>p = new Person("foo",1995,13,29);
Uncaught Error: 月が不正です (…)
>p = new Person("foo",1995,12,0);
Uncaught Error: 日が不正です (…)
```

**課題 5.3** 実行例 5.8においてエラーチェックが完全ではないことを指摘し、それを改良しなさい。

このリストの欠点はこのオブジェクトを継承するオブジェクトに対して、継承先のオブジェクトに対してエラーチェックの部分を再び書く必要がある。これを改良したのが次のリストである。

**実行例 5.9** 次のリストはエラーチェックの部分を関数化して、継承先でも同じようなチェックができるようにしたものである。

```
1 function Person2(name, y, m, d){
2 this.checkDate(y, m, d);
3 this.checkName(name);
4 this.name = name;
5 this.year = y;
6 this.month = m,
7 this.day = d
8 }
9 Person2.prototype = {
10 toString : function(){
11 return "私の名前は"+this.name+"です";
12 },
13 checkDate : function(y, m, d) {
14 var date = new Date(y,m,0);
15 if(m<1 || m>12) throw new Error("月が不正です");
16 if(d<1 || d>date.getDate()) throw new Error("日が不正です");
17 },
18 checkName : function(name) {
19 if(name === "") throw new Error("名前がありません");
20 },
21 get age(){
22 ... // 内容は省略
23 },
```

```
24 get birthday() {
25 return this.year+"年"+this.month+"月"+this.day+"日";
26 }
27 }
```

このリストではエラーチェックをする関数を作成して、それを `Person2.prototype` のなかに置いている。

- 2行目と3行目でエラーチェックをする関数を呼び出している。この関数はエラーが起きたときはエラーを投げるので、コンストラクタ関数の制御から離れることとなる。
- それぞれのエラーチェックをする関数は13行目から17行目と18行目から20行目に記述されている。
- 実行例5.7における `Person2` 継承した `Student` オブジェクトは次のようになる。

```
1 function Student(n, id, y, m, d){
2 this.checkDate(y, m, d);
3 this.checkName(name);
4 this.name = n;
5 this.year = y;
6 this.month = m;
7 this.day = d;
8 this.id = id;
9 }
10 Student.prototype = new Person2();
11 Student.prototype.constructor = Student;
```

これを実行すると `Person2` と同じように動作することがわかる。

```
>s=new Student("foo",1223300,1995,12,1);
Student {name: "foo", year: 1995, month: 12, day: 1, id: 1223300}
>s=new Student("foo",1223300,1995,4,0);
Uncaught Error: 日が不正です (...)
>s=new Student("foo",1223300,1995,13,1);
Uncaught Error: 月が不正です (...)
```

### 5.3.2 エラーからの復帰

前節の例ではエラーが発生するとそこでプログラムの実行が止まってしまう。エラーが発生したときに、投げられた (`throw` された) エラーを捕まえる (`catch` する) ことが必要である。このためには `try{...}catch{...}` 構文を使用する。

**実行例 5.10** 次のリストは `try{...}catch{...}` 構文を用いてオブジェクトが正しくできるまで繰り返す。なお、このリストはブラウザで実行することを想定している。

```
1 function test() {
2 var y, m, d;
3 for(;;) {
4 try {
5 y = Number(prompt("生まれた年を西暦で入力してください"));
6 m = Number(prompt("生まれた月を入力してください"));
7 d = Number(prompt("生まれた日を入力してください"));
8 return new Person2("foo", y, m, d);
9 } catch(e) {
10 console.log(e.name+": "+e.message);
11 }
12 }
13 }
```

- テストを繰り返す関数 `test()` が定義されている。
- 3行目では無限ループが定義されている。正しいパラメータが与えられたときに8行目で作成されたオブジェクトを戻り値にして関数の実行が終了する。
- `try{}` 内にはエラーが発生するかもしれないコードを中に含める。
  - － ここでは年、月、日の入力を `prompt` 用いてダイアログボックスを表示させ、そこに入力させている。
  - － 戻り値は文字列なので、`Number` で数に直している。
- 与えられた入力が正しくなければエラーが投げられ、`catch(e)` の中に制御が移る。
- `catch(e)` における `e` には発生したエラーオブジェクトが渡されるので、コンソールにその情報を出力する (10行目)。

`try{...}catch{}` 構文については次のようなこともある。

- `finally{}` を付けることもできます。`try{...}catch{}` 内の部分は `try` や `catch` の部分が実行された後必ず呼び出される<sup>2</sup>。
- `try{...}catch{...}` 構文は入れ子にできる。投げられたエラーに一番近い `catch` にエラーがつかまる。

---

<sup>2</sup>この例では正常時には `return` が実行されるので呼び出されません。

## 第6回 正規表現

正規表現とは、一定の規則にあてはまる文字列のパターンを記述する文字列のことである。正規表現とも呼ばれる。JavaScript では `RegExp` クラスが正規表現を表す。

### 6.1 正規表現オブジェクトの記述方法

正規表現のオブジェクトは `RegExp()` コンストラクタで生成できるが、正規表現リテラルを使って記述することもできる。正規表現リテラルは文字列をスラッシュ(/) で囲んで記述する。

次の正規表現はどちらも `s` で始まる文字列を表す。これを `s` で始まる文字列にマッチするという。

```
var pattern = new RegExp("^s");
var pattern = /^s/;
```

**リテラル文字** 正規表現では次の文字は特別な意味を持つ。これらはメタ文字と呼ばれる。

`^ $ . * + ? = ! | \ / ( ) [ ] { }`

これらの文字をそのまま文字として使いたい場合はその前にバックスラッシュ(\) を付ける。英数字の前にバックスラッシュを付けると別な意味になる場合がある (表 6.1)。

表 6.1: 正規表現のリテラル文字

| 文字                  | 意味                                                                                     |
|---------------------|----------------------------------------------------------------------------------------|
| 英数字                 | 通常の文字                                                                                  |
| <code>\0</code>     | NULL 文字 ( <code>\u0000</code> )                                                        |
| <code>\t</code>     | タブ ( <code>\u0009</code> )                                                             |
| <code>\n</code>     | 改行 ( <code>\u000A</code> )                                                             |
| <code>\v</code>     | 垂直タブ ( <code>\u000B</code> )                                                           |
| <code>\f</code>     | 改ページ ( <code>\u000C</code> )                                                           |
| <code>\r</code>     | 復帰 ( <code>\u000D</code> )                                                             |
| <code>\xnn</code>   | 16 進数 <code>nn</code> で指定された ASCII 文字 ( <code>\x0A</code> は <code>\n</code> と同じ)       |
| <code>\uxxxx</code> | 16 進数 <code>xxxx</code> で指定された Unicode 文字 ( <code>\u000D</code> は <code>\r</code> と同じ) |
| <code>\cX</code>    | 制御文字 ( <code>\cC</code> は <code>\u0003</code> と同じ)                                     |

**文字クラス** 個々のリテラル文字を `[]` で囲むことでそれぞれの文字の一つにマッチする。このほかにもよく使われる文字クラスには特別なエスケープシーケンスがある (表 6.2)。

表 6.2: 文字クラス

| 文字                  | 意味                                                    |
|---------------------|-------------------------------------------------------|
| <code>[...]</code>  | <code>[]</code> 内の任意の 1 文字                            |
| <code>[^...]</code> | <code>[]</code> 内以外の任意の 1 文字                          |
| <code>.</code>      | 改行 (Unicode の行末文字) 以外の任意の 1 文字 <code>[^\n]</code> と同じ |
| <code>\w</code>     | 任意の単語文字。 <code>[A-Za-z0-9]</code> と同じ                 |
| <code>\W</code>     | 任意の単語文字以外の文字。 <code>[^A-Za-z0-9]</code> と同じ           |
| <code>\s</code>     | 任意の Unicode 空白文字                                      |
| <code>\S</code>     | 任意の Unicode 空白文字以外の文字                                 |
| <code>\d</code>     | 任意の数字。 <code>[0-9]</code> と同じ                         |
| <code>\D</code>     | 任意の数字以外の文字。 <code>[^0-9]</code> と同じ                   |
| <code>\\b</code>    | リテラルバックスペース                                           |

たとえば、`\d\d` は 2 桁の 10 進数にマッチする。先頭に 0 が来てもよい。先頭に 0 が来る場合を除くのであれば `[1-9]\d` となる。一般には、同じパターンの繰り返しが必要になることが多い。

**繰り返し** 正規表現の文字の繰り返しを指定できる (表 6.3)。

表 6.3: 繰り返しの指定

| 文字                 | 意味                                                 |
|--------------------|----------------------------------------------------|
| <code>{m,n}</code> | 直前の項目の <code>m</code> 回から <code>n</code> 回までの繰り返し  |
| <code>{m,}</code>  | 直前の項目の <code>m</code> 回以上の繰り返し                     |
| <code>{m}</code>   | 直前の項目の <code>m</code> 回の繰り返し                       |
| <code>?</code>     | 直前の項目の 0 回 (なし) か 1 回の繰り返し。 <code>{0,1}</code> と同じ |
| <code>+</code>     | 直前の項目の 1 回以上の繰り返し。 <code>{1,}</code> と同じ           |
| <code>*</code>     | 直前の項目の 0 回以上の繰り返し。 <code>{0,}</code> と同じ           |

- 4 桁の 10 進数のパターンは `\d\d\d\d` で表されるが、繰り返しを使うと `\d{4}` で表される。
- 1 桁以上の 10 進数は `\d+` で表される。先頭が 0 でないようにすると、`[1-9]\d*` と表される。

**非貪欲な繰り返し** 通常、正規表現において繰り返しはできるだけ長く一致するように繰り返しが行われる。これを貪欲な繰り返しという。たとえば、`"aaaaab"` という文字列に対し、正規表現 `/a+/` がマッチする部分は `aaaaa` の長さ 5 の文字列になる (もっともここではこのパターンが含まれることしかわからない)。

これに対し、できるだけ短い文字列のマッチで済ませる繰り返しを、非貪欲な繰り返しという。これを指定するには繰り返し指定の後に`?`を付ける。つまり、`??`、`+`、`*?`、`{1,5}?` などのように記述する。

したがって、`"aaaaab"`という文字列に対し、正規表現 `/a+?/` がマッチする部分は `a` の長さ 1 の文字列になる。しかし、正規表現 `/a+?b/` がマッチする部分は全体の`"aaaaab"`になる。これはマッチが開始する位置が、この文字列の先頭から始まるためである。詳しくは実行例 6.5を参照のこと。

**選択、グループ化、参照** 正規表現にはいくつかのパターンの選択や、表現のグループ化ができる。また、グループ化したところに一致した文字列を後で参照する (前方参照) ことができる (表 6.4)<sup>1</sup>。

表 6.4: 選択、グループ化、参照の指定

| 文字                   | 意味                                                                                                                     |
|----------------------|------------------------------------------------------------------------------------------------------------------------|
| <code> </code>       | この記号の左右のどちらかを選択する                                                                                                      |
| <code>(...)</code>   | 正規表現のグループ化をする。これにより、 <code>*</code> 、 <code>+</code> 、 <code> </code> などの対象がグループ化されたものになる。また、グループに一致した文字列を記憶して後で参照できる。 |
| <code>(?:...)</code> | グループ化しか行わない。一致した文字列を記憶しない。                                                                                             |
| <code>\n</code>      | グループ番号 <code>n</code> で指定された部分表現に一致する。グループ番号は左から数えた (の数である。ただし、 <code>(?</code> は数えない。                                 |

JavaScript では文字列は`"`ではさむものか`'`ではさむことになっている。グループ番号を使うと文字列のパターンのチェックができる。

**実行例 6.1** 選択、グループ化の例を挙げる。

- `(J|j)ava(S|s)cript` は `JavaScript`, `Javascript`, `javaScript`, `javascript` の 4 つにマッチする。
- `[+-]?\d+` は符号つき (なくてもよい)10 進数とマッチする。`[+-]?`により、符号がなくてもよいことに注意すること。

**一致位置の指定** 正規表現には文字列が現れる位置を指定することができる (表 6.5)。

最後の 2 つは `Java` にはマッチさせたいが `JavaScript` にはマッチさせたくないときなどに使用できる。

**フラグ** 正規表現にはいくつかのフラグを指定できる (表 6.6)。フラグを書く位置は正規表現リテラルを表す `/.../` の後に書く。

<sup>1</sup>元来の正規表現では前方参照ができない。この機能は正則言語のより広い範囲の言語であることを示している

表 6.5: 一致位置の指定

| 文字                 | 意味                                                                        |
|--------------------|---------------------------------------------------------------------------|
| <code>^</code>     | 文字列の先頭                                                                    |
| <code>\$</code>    | 文字列の最後                                                                    |
| <code>\b</code>    | 単語境界。 <code>\w</code> と <code>\W</code> の間の位置。[ <code>\b</code> ] との違いに注意 |
| <code>\B</code>    | 単語境界以外                                                                    |
| <code>(?=p)</code> | 後に続く文字列が <code>p</code> に一致することが必要。                                       |
| <code>(?!p)</code> | 後に続く文字列が <code>p</code> に一致しないことが必要。                                      |

表 6.6: フラグの指定

| 文字             | 意味                                                                               |
|----------------|----------------------------------------------------------------------------------|
| <code>i</code> | 大文字と小文字を区別しない                                                                    |
| <code>g</code> | グローバル検索をする。初めに一致したものだけでなくすべてを検索する。                                               |
| <code>m</code> | 複数行モードにする。 <code>^</code> は文字列の先頭だけでなく、行の先頭に。 <code>\$</code> は文字列の末尾と行の末尾に一致する。 |

`g` のフラグはパターンマッチした部分文字列を置き換えるメソッド内でしか意味を持たない。たとえば、`/javascript/ig` である。これは `JaVaScRipt` などにマッチする。

**課題 6.1** 次の文字列にマッチする正規表現を作れ。

1. C 言語の変数名の命名規則に合う文字列
2. 符号付小数。符号はなくてもよい。整数の場合は小数点はなくてもよい。また、小数点はあっても小数部はなくてもよい。整数部分には数字が少なくとも一つはあること。たとえば `-1.` にはマッチするが、`-.0` には整数部分がないのでマッチしない。`.` のエスケープを忘れないようにすること。
3. 前問の正規表現を拡張して、指数部が付いた浮動小数にマッチするものを作れ。指数部は `E` または `e` で始まり、符号付き (なくてもよい) 整数とする。
4. 24 時間生の時刻の表し方。時、分、秒はすべて 2 桁とし、それらの区切りは `:` とする。たとえば午後 1 時 10 分 6 秒は `13:10:06` である。また、`13:10:66` は秒数が 60 以上になっているのでマッチしてはいけない。



## 6.2 文字列のパターンマッチングメソッド

表 6.7は正規表現による文字列のパターンマッチングが使用できる **String** オブジェクトのメソッドの一覧である。

表 6.7: 文字列で正規表現が使えるメソッド

| 文字                     | 引数          | 意味                                                                                                                |
|------------------------|-------------|-------------------------------------------------------------------------------------------------------------------|
| <code>match()</code>   | 正規表現        | 引数の正規表現にマッチした部分を文字列の配列で返す。見つからない場合は <code>null</code> 変える。 <code>g</code> フラグがない場合の補足は下の例を参照。                     |
| <code>replace()</code> | 正規表現、置換テキスト | <code>g</code> フラグがあれば一致した部分すべてを、ない場合は、はじめのところだけ置換した文字列を返す。置換文字列の中でグループ化した部分文字列を <code>\$1,\$2,...</code> で参照できる。 |
| <code>search()</code>  | 正規表現        | 正規表現に一致した位置を返す。見つからない場合は <code>-1</code> を返す。 <code>g</code> フラグは無視される。                                           |
| <code>split()</code>   | 正規表現、分割最大数  | 正規表現のある位置で文字列を分割する。2 番目の引数はオプション                                                                                  |

これらをまとめた実行例を示す。

**実行例 6.2** 4桁の数字にマッチする正規表現オブジェクトを作成する。

```
var ex = /\d\d\d\d/;
undefined
```

この正規表現に対し、それぞれのメソッドを適用させる。

```
>"20144567".search(ex);
0
>"20144567".match(ex);
["2014"]
>"20144567".replace(ex,"AA");
"AA4567"
```

- 検索対象の文字列はすべて数字からなるのでこの正規表現にマッチする位置は 0 である。
- マッチした文字列は先頭から 4 文字となる。
- "AA"で置き換えると先頭の 4 文字が置き換えられる。

正規表現に `g` フラグをつけて同じようなことをする。

```
>var exg = /\d\d\d\d/g;
undefined
>"20144567".search(exg);
0
>"20144567".match(exg);
["2014", "4567"]
>"20144567".replace(exg,"AA");
"AAAA"
```

g フラグがあるので `match()` や `replace()` が複数回実行されていることがわかる。

**実行例 6.3** 次の例はマッチした文字列を置換する。

```
>"aaa bbb".replace(/(\w*)\s*(\w*)/, "$2,$1");
"bbb,aaa"
```

英数字からなる 2 つの文字列 (`(\w*)`) の順序を入れ替えて、その間に、`,` を挿入する。`$1` は文字列 `aaa` に、`$2` は文字列 `bbb` にマッチしている。

**実行例 6.4** 次の例は文中の `Java` を `JavaScript` に変える。ここでは `JavaScript` を `JavaJavaScript` にしないために `(?!p)` を用いる。

```
>"Java と JavaScript は全く違う言語です.".replace(/Java(?!Script)/,"JavaScript");
"JavaScript と JavaScript は全く違う言語です。"
```

**実行例 6.5** 貪欲さと非貪欲さの確認を行う。

```
>"aaaaab".match(/a+/);
["aaaaa"]
>"aaaaab".match(/a+?/);
["a"]
```

上の例は貪欲なので `a` の繰り返しの部分を最大限の位置でマッチしている。下の例は非貪欲なので最小限の長さの部分にしかマッチしていない。

```
>"aaaaab".match(/a+?b/);
["aaaaab"]
```

この例では、`ab` にマッチしていないことに注意すること。初めに先頭の `a` にマッチしたので `b` が来るところまでマッチする。

**実行例 6.6** 次の例は前方参照を行っている。

```
>"abccdbcc".search(/((.)\2).*\1/);
-1
```

- (.) の部分は左かっこが 2 番目にあるので、\2 で参照できる。したがって、(.)\2 は同じ文字が 2 つ続いていることを意味する。この部分全体が再び ( ) でくくられているので、その部分は \1 で参照できる。
- したがってこの正規表現は、同じ文字の繰り返しが 2 回現れる文字列にマッチする。
- 文字列 "abccbcc" には同じ文字が連続して現れるのが末尾の "cc" しかないなので、この文字列にはマッチしない (戻り値が -1 である)。

```
>"abccbcc".search(/((.)\2).*\1/);
2
>"abccbcc".match(/((.)\2).*\1/);
["ccbcc", "cc", "c"]
```

- この文字列では cc という同じ文字を繰り返した部分が 2 か所あるのでマッチする。はじめの cc の位置が先頭から 2 番目なので戻り値が 2 となっている。
- match() を行くと、cc ではさまれた部分文字列が戻り値の配列の先頭に、以下、\1 と \2 にマッチした部分文字列が配列に入っている。

```
>"abccbcbckkccaaMMaa".match(/((.)\2).*\1/);
["ccbcbckkcc", "cc", "c"]
```

- この例では cc が部分文字列に 3 か所現れている。
- 貪欲なマッチのため、マッチした部分は 1 番目と 3 番目の cc にはさまれた部分である。この文字列には aa ではさまれた部分文字列も存在するが、g フラグが付いていないのではじめにマッチしたものだけが戻ってこない。
- 配列の残りの成分には \1 と \2 が入っている。

```
>"abccbcbckkccaaMMaa".match(/((.)\2).*\1/g);
["ccbcbckkcc", "aaMMaa"]
```

g フラグを付けるとマッチした部分文字列の配列が戻ってくるが、\1 などの情報は得られない。

```
"abccbcbckkccaaMMaa".match(/((.)\2).*?\1/g);
["ccbcc", "aaMMaa"]
```

この例は非貪欲でグローバルなマッチである。非貪欲にすると一番目と 2 番目の cc にはさまれた部分と aa ではさまれた部分がそれぞれマッチする。

次の例は前方参照を利用してデータなどに含まれる文字列の引用符が対応している (シングルクオート同士、ダブルクオート同士) かを確認している。

```
>'\'abcd"df"\'\''.search(/^(['"]).*\1$/);
0
>'\'abcd"df"\'\''.match(/^(['"]).*\1$/);
["\'abcd"df\'\'", "']
```

- \\' は文字列データとして与えられたことを仮定している。
- 文字クラス ['] が () で囲まれているのでここで一致した文字が \1 で参照できる。
- したがって、文字列の先頭と最後に同じ引用符が来る場合に文字は見つかる。
- match() の戻り値は配列で、正規表現に g フラグがないときは、初めがマッチした文字列、以下順に \1, \2... にマッチした文字列が入っている。

**実行例 6.7** 文字列を指定した文字列で分割する split() の分割する文字列にも正規表現が使える。

```
>" 1, 2 , 3 , 4".split(/\s*,\s*/);
[" 1", "2", "3", "4"]
```

0 個以上の空白、,、0 個以上の空白の部分で分割している。1 の前にある空白が除去できていない。

```
>" 1, 2 , 3 , 4".split(/\W+/);
["", "1", "2", "3", "4"]
```

非単語文字の 1 個以上の並びで分割している。先頭の空白で分割されているので、分割された最初の文字列は空文字列 "" となっている。

```
>" 1, 2 , 3 , 4".replace(/\s/, "").split(/\W+/);
["1", "2", "3", "4"]
```

先頭の分割文字列が空文字になるのを防ぐために、初めに空白文字を空文字に置き換えて (取り除いて) いる。その文字列に対し非単語文字列で分割しているので空文字が分割結果に表れない。

この様に文字列に対してメソッドを順に続けて記述することができる。

**課題 6.2** 次の実行結果がどうなるか答えよ。

1. "aaaabaaabb".match(/.\*b/);
2. "aaaabaaabb".match(/.\*b/g);
3. "aaaabaaabb".match(/.\*?b/);
4. "aaaabaaabb".match(/.\*?b/g);
5. "abccbckkkccaaMMaacc".match(/((.)\2).\*\1/);
6. "abccbckkkccaaMMaacc".match(/((.)\2).\*\1/g);
7. "abccbckkkccaaMMaacc".match(/((.)\2).\*?\1/);
8. "abccbckkkccaaMMaacc".match(/((.)\2).\*?\1/g);
9. "abccbckkkccaaMMaa".match(/((.)\2).\*\1/);
10. "abccbckkkccaaMMaa".match(/((.)\2).\*\1/g);
11. "abccbckkkccaaMMccaa".match(/((.)\2).\*\1/g);
12. "abccbckkkccaaMMccaa".match(/((.)\2).\*?\1/g);

## 第7回 DOMの利用

この節からは HTML 文書の取り扱いを始める。この授業では 2014 年 10 月 28 日に W3C の Recommendation となった HTML5 を用いる<sup>1</sup>。

### 7.1 HTML 文書の構成

**実行例 7.1** 次のリストは Google Maps を利用して地図を表示するものである<sup>2</sup>。

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
5 <title>初めての GoogleMaps</title>
6 <script type="text/javascript"
7 src="http://maps.google.com/maps/api/js?sensor=false"></script>
8 <script type="text/javascript">
9 window.onload = function() {
10 var latlng = new google.maps.LatLng(35.486210,139.341443);
11 var myOptions = {
12 zoom: 10,
13 center: latlng,
14 mapTypeId: google.maps.MapTypeId.ROADMAP
15 };
16 var mapCanvas = document.getElementById("map_canvas")
17 var map = new google.maps.Map(mapCanvas, myOptions);
18 }
19 </script>
20 <link rel="stylesheet" type="text/css" href="map.css" />
21 </head>
22 <body>
23 <div id="map_canvas" ></div>
```

---

<sup>1</sup><http://www.w3.org/TR/2014/REC-html5-20141028/>

<sup>2</sup>Google Maps API3 の公開当初は API キーは存在しなかったが、その後 API キーが必要となった。利用当初の状況から API キーがなくても動作するが、API キーを取得することを勧める。取得の方法は付録に示す。

```
24 </body>
```

```
25 </html>
```

ここではこの HTML 文書の構成について解説をする。

- 1 行目は HTML 文書の DOCTYPE 宣言である。この形は HTML5 におけるもので、以前のものと比べて記述が簡単になっている。
- 2 行目はこの HTML 文書のルート要素と呼ばれるものである。最後の 25 行目の `</html>` までは有効となる。すべての要素はこの範囲になければならない。
- 3 行目から始まる `<head>` はブラウザに表示されない、いろいろな HTML 文書の情報を表す。
  - － 4 行目はこの文書の形式や文字集合を記述している。ここでは内容は `text/html` の形式、つまり、テキストで書かれた `html` の形式で書かれていることを表す。<sup>3</sup>
  - － 5 行目の `<title>` はブラウザのタブに表示される文字列を指定している。
  - － 6 行目から 7 行目は Google Maps のライブラリーを読み込むためのものである。このように JavaScript のプログラムは外部ファイルとすることができる。
  - － 8 行目から 19 行目は HTML 文書内に書かれた JavaScript である。詳しい解説は後の授業で行う。
  - － 20 行目は HTML 文書の見栄えなどを規定する CSS ファイルを外部から読み込むことをしている。
- HTML 文書で実際にブラウザ内で表示される情報は `<body>` 要素内に現れる。
- このリストでは Google Maps を表示するための `<div>` 要素が一つあるだけである。このとき、`<div>` は `<body>` の子要素であるといい、`<body>` は `<div>` の親要素という。
- 各要素名または要素の終了を示すタグ (`<...>`) の間に文字列がある場合、その部分はテキストノードと呼ばれるノードが作成されている。

各要素は `<` との中に現れる。初めに現れる文字列が要素名であり、そのあとに属性と属性値がいくつか並ぶ。

- 属性とその属性値は `=` で結ばれる。
- 属性値は `"` ではさまれた文字列として記述する。
- 6 行目の `<script>` 要素では属性 `type` と `src` が設定されている。
- 23 行目の `<div>` 要素では属性 `id` に属性値 `map_canvas` を設定している。なお、この要素は CSS によっても属性が定義されている。

次のリストは 20 行目で参照している CSS ファイルの内容である。

---

<sup>3</sup>このような方法でファイルのデータ形式を表すことを MIME(Multipurpose Internet Mail Extension) タイプと呼ぶ。元来、テキストデータしか扱えない電子メールに様々なフォーマットのデータを扱えるようにする規格である。

```

1 #map_canvas{
2 width:500px;
3 height:500px;
4 float:left;
5 margin:5px 10px 5px 10px;
6 }

```

- CSS の各構成要素は HTML 文書の要素を選択するセクタ (ここでは`#map_canvas`) とそれに対する属性値の並び (`[属性]:[属性値];`) からなる。
- `#` で始まるセクタはそのあとの文字列を`<id>`の属性値に持つ要素に適用される。
- したがって、この規則は 23 行目の`<div>`要素に適用される。
- その内容は Google Maps が表示される画面の大きさ (`width` と `height`)、配置の位置 (`float`) と要素の外に配置される空白 (`margin`) を指定している。

## 7.2 CSS の利用

カスケーディングスタイルシート (CSS) は HTML 文書の要素の表示方法を指定するものである。CSS は JavaScript から制御できる。

文書のある要素に適用されるスタイルルールは、複数の異なるルールを結合 (カスケード) したものである。スタイルを適用するためには要素を選択するセクタで選ぶ。

表 7.1 は CSS3 におけるセクタを記述したものである<sup>4</sup>。

表 7.1: CSS3 のセクタ

| セクタ                         | 解説                                                                                           |
|-----------------------------|----------------------------------------------------------------------------------------------|
| <code>*</code>              | 任意の要素                                                                                        |
| <code>E</code>              | タイプが <code>E</code> の要素                                                                      |
| <code>E[foo]</code>         | タイプが <code>E</code> で属性 <code>"foo"</code> を持つ要素                                             |
| <code>E[foo="bar"]</code>   | タイプが <code>E</code> で属性 <code>"foo"</code> の属性値が <code>"bar"</code> である要素                    |
| <code>E[foo~="bar"]</code>  | タイプが <code>E</code> で属性 <code>"foo"</code> の属性値がスペースで区切られたリストでその一つが <code>"bar"</code> である要素 |
| <code>E[foo^="bar"]</code>  | タイプが <code>E</code> で属性 <code>"foo"</code> の属性値が <code>"bar"</code> で始まる要素                   |
| <code>E[foo\$="bar"]</code> | タイプが <code>E</code> で属性 <code>"foo"</code> の属性値が <code>"bar"</code> で終わる要素                   |
| <code>E[foo*="bar"]</code>  | タイプが <code>E</code> で属性 <code>"foo"</code> の属性値が <code>"bar"</code> を含む要素                    |
| <code>E[foo ="en"]</code>   | タイプが <code>E</code> で属性 <code>"foo"</code> の属性値がハイフンで区切られたリストでその一つが <code>"en"</code> で始まる要素 |

次ページへ続く

<sup>4</sup><http://www.w3.org/TR/selectors/>より引用。

表 7.1: CSS3 のセレクタ (続き)

| セレクタ                              | 解説                                                                         |
|-----------------------------------|----------------------------------------------------------------------------|
| <b>E:root</b>                     | <b>document</b> のルート要素                                                     |
| <b>E:nth-child(n)</b>             | 親から見て <b>n</b> 番目の要素                                                       |
| <b>E:nth-last-child(n)</b>        | 親から見て最後から数えて <b>n</b> 番目の要素                                                |
| <b>E:nth-of-type(n)</b>           | そのタイプの <b>n</b> 番目の要素                                                      |
| <b>E:nth-last-of-type(n)</b>      | そのタイプの最後から <b>n</b> 番目の要素                                                  |
| <b>E:first-child</b>              | 親から見て一番初めの子要素                                                              |
| <b>E:last-child</b>               | 親から見て一番最後の子要素                                                              |
| <b>E:first-of-type</b>            | 親から見て初めてのタイプである要素                                                          |
| <b>E:last-of-type</b>             | 親から見て最後のタイプである要素                                                           |
| <b>E:only-child</b>               | 親から見てただ一つしかない子要素                                                           |
| <b>E:only-of-type</b>             | 親から見てただ一つしかないタイプの要素                                                        |
| <b>E:empty</b>                    | テキストノードを含めて子要素がない要素                                                        |
| <b>E:link, E:visited</b>          | まだ訪れたことがない ( <b>:link</b> ) か訪れたことがある ( <b>visited</b> ) ハイパーリンクのアンカーである要素 |
| <b>E:active, E:hover, E:focus</b> | ユーザーに操作されている状態中の要素                                                         |
| <b>E:target</b>                   | 参照 URI のターゲットである要素                                                         |
| <b>E:enabled, E:disabled</b>      | 使用可能 ( <b>:enable</b> ) か使用不可のユーザーインターフェイスの要素                              |
| <b>E:checked</b>                  | チェックされているユーザーインターフェイスの要素                                                   |
| <b>E::first-line</b>              | 要素のフォーマットされたはじめの行                                                          |
| <b>E::first-letter</b>            | 要素のフォーマットされたはじめの行                                                          |
| <b>E::before</b>                  | 要素の前に生成されたコンテンツ                                                            |
| <b>E::after</b>                   | 要素の後に生成されたコンテンツ                                                            |
| <b>E.warning</b>                  | 属性 <b>class</b> が "warning" である要素                                          |
| <b>E#myid</b>                     | 属性 <b>id</b> の属性値が "myid" である要素                                            |
| <b>E:not(s)</b>                   | 単純なセレクタ <b>s</b> にマッチしない要素                                                 |
| <b>E F</b>                        | 要素 <b>E</b> の子孫である要素 <b>F</b>                                              |
| <b>E &gt; F</b>                   | 要素 <b>E</b> の子である要素 <b>F</b>                                               |
| <b>E F+</b>                       | 要素 <b>E</b> の直後にある要素 <b>F</b>                                              |
| <b>E ~ F</b>                      | 要素 <b>E</b> の直前にある要素 <b>F</b>                                              |

いくつか注意する点を挙げる。

- 属性 **id** の属性値の前に **#** をつけることでその要素が選ばれる。
- 属性 **class** の属性値の前に **.** をつけることでその要素が選ばれる。
- **nth-child(n)** には単純な式を書くことができる。詳しくは実行例 7.1 を参照のこと。このセ



レクタは複数書いてもよい。

- $E F$  と  $E > F$  の違いを理解しておくこと。たとえば `div div` というセレクタは途中で別の要素が挟まれていてもよい。また、`<div>` 要素が 3 つある場合にはどのような 2 つの組み合わせも対象となる。

**課題 7.1** 次の HTML 文書において `nth-child` の ( ) 内に次の式を入れた時どうなるか報告しなさい。ここで `<ol>` は箇条書きの開始を示す要素であり、`<li>` は箇条書きの各項目を示す要素である。

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
5 <title>nth-child のチェック</title>
6 <style type="text/css" >
7 li:nth-child(n){
8 background:yellow;
9 }
10 </style>
11 </head>
12 <body>
13
14 1 番目
15 2 番目
16 3 番目
17 4 番目
18 5 番目
19 6 番目
20
21 </body>
22 </html>
```

1.  $n$  (ここでのリストの設定)
2.  $2n$
3.  $n+3$
4.  $-n+2$

**課題 7.2** 前問のリストに対し、背景色が次のようになるように CSS を設定しなさい。

1. 偶数番目が黄色、基数番目がオレンジ色
2. 1 番目、4 番目、... のように 3 で割ったとき、1 余る位置が明るいグレー

3. 4 番目以下がピンク
4. 下から 2 番目以下が緑色

## 7.3 DOM とは

Document Object Model(DOM) は HTML 文書などの要素をノードとしたツリー構造で管理する方法である。DOM のメソッドやプロパティを使うことで各要素にアクセスしたり、属性値やツリーの構造を変化させることができる。DOM の構造は開発者ツールなどで見ることができる。

- Opera と Google Chrome では開発者ツールの Elements タブで確認できる。ここで要素上で右クリックして Edit as HTML を選択するとテキストとして編集できる。
- FireFox では開発ツールの「インスペクタ」タブで DOM ツリーが確認できる。要素上で右クリックから「HTML として編集」とするとテキストとして編集できる。
- IE では開発者ツールを開き、左にある「HTML」タブで同様のことができる。

## 7.4 DOM のメソッドとプロパティ

DOM では DOM ツリーを操作するためにメソッドやプロパティが規定されている。これらの手段を用いて DOM をサポートする文書にアクセスができる。

メソッドとはそのオブジェクトに対する操作である。次のような手段を提供している。

- 条件に合う要素または要素のリストを得る。
- 要素の属性を参照、変更ができる。
- 要素を新規に作成する。
- ある要素に子要素を追加したり、取り除いたりする。

プロパティはそのオブジェクトが持つ性質であり、それらの値を参照できる。ほとんどのプロパティは書き直しできない。

なお、ここでのメソッドやプロパティは DOM 文書で使用可能なものである。したがって、HTML 文書も DOM をサポートするブラウザであれば同様の方法で部分的に書き直すことが可能である。

### 7.4.1 DOM のメソッド

表 7.2 は DOM のメソッドのリストである。`document` だけに適用できるものと、すべての要素に適用できるものがある。なお、結果が要素のリストであるものについては配列と同様に [ ] によりそれぞれの要素を指定できる。

なお、表中の名前空間 (Namespace) とは、指定した要素が定義されている規格を指定するものである。一つの文書内で複数の規格を使用する場合、作成する要素がどこで定義されているのかを

指定する。これにより、異なる規格で同じ要素名が定義されていてもそれらを区別することが可能となる。HTML 文書では <http://www.w3.org/1999/xhtml> を指定する<sup>5</sup>。

表 7.2: DOM のメソッド

| メソッド名                                     | 対象要素                  | 説明                                                                                                              |
|-------------------------------------------|-----------------------|-----------------------------------------------------------------------------------------------------------------|
| <code>getElementById(id)</code>           | <code>document</code> | 属性 <code>id</code> の値が <code>id</code> の要素を得る。                                                                  |
| <code>getElementsByTagName(Name)</code>   | 対象要素                  | 要素名が <code>Name</code> の要素のリストを得る。                                                                              |
| <code>getElementsByClassName(Name)</code> | 対象要素                  | 属性 <code>class</code> の値が <code>Name</code> の要素のリストを得る。                                                         |
| <code>getElementsByName(Name)</code>      | <code>document</code> | 属性 <code>name</code> が <code>Name</code> である要素のリストを得る。                                                          |
| <code>querySelector(selectors)</code>     | 対象要素                  | <code>selectors</code> で指定された CSS のセレクタに該当する一番初めの要素を得る。                                                         |
| <code>querySelectorAll(selectors)</code>  | 対象要素                  | <code>selectors</code> で指定された CSS のセレクタに該当する要素のリストを得る。                                                          |
| <code>getAttribute(Attrib)</code>         | 対象要素                  | 対象要素の属性 <code>Attrib</code> の値を読み出す。得られる値はすべて文字列である。                                                            |
| <code>setAttribute(Attrib,Val)</code>     | 対象要素                  | 対象要素の属性 <code>Attrib</code> の値を <code>Val</code> にする。数を渡しても文字列に変換される。                                           |
| <code>hasAttribute(Attrib)</code>         | 対象要素                  | 対象要素に属性 <code>Attrib</code> がある場合は <code>true</code> を、ない場合は <code>false</code> を返す。                            |
| <code>removeAttribute(Attrib)</code>      | 対象要素                  | 対象要素の属性 <code>Attrib</code> を削除する。                                                                              |
| <code>createElement(Name)</code>          | <code>document</code> | <code>Name</code> で指定した要素を作成する。                                                                                 |
| <code>createElementNS(NS,Name)</code>     | <code>document</code> | 名前空間 <code>NS</code> にある要素 <code>Name</code> を作成する。                                                             |
| <code>createTextNode(text)</code>         | <code>document</code> | <code>text</code> を持つテキストノードを作成する。                                                                              |
| <code>cloneNode(bool)</code>              | 対象要素                  | <code>bool</code> が <code>true</code> のときは対象要素の子要素すべてを、 <code>false</code> のときは対象要素だけの複製を作る。                    |
| <code>appendChild(Elm)</code>             | 対象要素                  | <code>Elm</code> を対象要素の最後の子要素として付け加える。 <code>Elm</code> がすでに対称要素の子要素のときは最後の位置に移動する。                             |
| <code>insertBefore(newElm, PElm)</code>   | 対象要素                  | 対象要素の子要素 <code>PElm</code> の前に <code>newElm</code> を子要素として付け加える。 <code>Elm</code> がすでに対称要素の子要素のときは指定された位置に移動する。 |
| <code>removeChild(Elm)</code>             | 対象要素                  | 対象要素の子要素 <code>Elm</code> を取り除く。                                                                                |
| <code>replaceChild(NewElm, OldElm)</code> | 対象要素                  | 対象要素の子要素 <code>OldElm</code> を <code>NewElm</code> で置き換える。                                                      |

**実行例 7.2** 次の例は 1 月から 12 月までを選択できるプルダウンメニューを作成するものである。この様なプルダウンメニューをテキストエディタで入力すると次のようになる。

<sup>5</sup><http://dev.w3.org/html5/spec-LC/namespaces.html>

```
<!DOCTYPE html>
<head>
<meta charset="UTF-8"/>
<title>プルダウンメニュー</title>
</head>
<body>
 <form id="menu">
 <select>
 <option value="1">1 月</option>
 <option value="2">2 月</option>
 <option value="3">3 月</option>
 <option value="4">4 月</option>
 <option value="5">5 月</option>
 <option value="6">6 月</option>
 <option value="7">7 月</option>
 <option value="8">8 月</option>
 <option value="9">9 月</option>
 <option value="10">10 月</option>
 <option value="11">11 月</option>
 <option value="12">12 月</option>
 </select>
 </form>
</body>
</html>
```

- ユーザからの入力を受け付ける要素は通常、`<form>`要素内に記述する。
- プルダウンメニュー の要素名は`<select>`で、選択する内容は`<option>`要素である。
- `<option>`要素の属性 `value` の値が選択した値として利用できる。
- `<option>`要素内の文字列 (テキストノード) がプルダウンメニューに表示されるものになる。
- `<select>`は`<form>`の子要素であり、各`<option>`は`<select>`の子要素となっている。

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
5 <title>プルダウンメニューの作成</title>
6 <script type="text/ecmascript">
7 //<![CDATA[
```

```
8 window.onload = function(){
9 var i, Option;
10 var Select = document.createElement("select");
11 document.getElementById("menu").appendChild(Select);
12 for(i=1;i<=12;i++) {
13 Option = document.createElement("option");
14 Option.setAttribute("value",i);
15 Select.appendChild(Option);
16 Option.appendChild(document.createTextNode(i+"月"));
17 }
18 }
19 //]]>
20 </script>
21 </head>
22 <body>
23 <form id="menu"></form>
24 </body>
25 </html>
```

- 8 行目の `window.onload` はファイルのロードが終わった後に発生するイベントの処理をする関数を表す。イベントの詳細については後で解説する。
- 10 行目では `<select>` 要素を作成している。
- 11 行目では 10 行目で作成した `<select>` 要素を `getElementById("menu")` で得た `<form>` 要素の子要素に設定している。
- 12 行目から始まる `for` ループで 12 個の `<option>` 要素を作成し、`<select>` 要素の子要素としている。
  - － 13 行目で `<option>` 要素を新規に作成し、14 行目で、その要素の属性 `value` に値を設定している。
  - － 15 行目ではその `<option>` 要素を `<select>` 要素の子要素としている。
  - － さらに、15 行目では表示する文字列をもつテキストノードを作成し、16 行目でそれを `<option>` 要素の子要素としている。

**課題 7.3** 実行例 7.2 に対して次のことを行いなさい。

1. ブラウザの「ページのソースを表示」を選択して、ソースコードがどのようなになっているか確認する。
2. DOM ツリーが実行例 7.2 のリストと同じようにできていることを確認する。
3. `<select>` 要素を構成する部分を関数化して、月だけでなく日 (1 日から 31 日) が選べるプルダウンメニューも作成できるようにしなさい。関数の引数を工夫すること。

### 7.4.2 DOM の要素のプロパティ

表 7.3は DOM の要素に対するプロパティである。

表 7.3: DOM 要素に対するプロパティ

プロパティ名	説明
<code>firstChild</code>	指定された要素の先頭にある子要素
<code>lastChild</code>	指定された要素の最後にある子要素
<code>nextSibling</code>	指定された子要素の次の要素
<code>previousSibling</code>	現在の子要素の前にある要素
<code>parentNode</code>	現在の要素の親要素
<code>hasChildNodes</code>	その要素が子要素を持つ場合は <code>true</code> 持たない場合は <code>false</code> である。
<code>nodeName</code>	その要素の要素名前
<code>nodeType</code>	要素の種類 (1 は普通の要素、3 はテキストノード)
<code>nodeValue</code>	(テキスト) ノードの値
<code>childNodes</code>	子要素の配列
<code>children</code>	子要素のうち通常の要素だけからなる要素の配列 (DOM4 で定義)
<code>firstElementChild</code>	指定された要素の先頭にある通常の要素である子要素 (DOM4 で定義)
<code>lastElementChild</code>	指定された要素の最後にある通常の要素である子要素 (DOM4 で定義)
<code>nextElementSibling</code>	指定された子要素の次の通常の要素 (DOM4 で定義)
<code>previousElementSibling</code>	現在の子要素の前にある通常の要素 (DOM4 で定義)

なお、DOM4<sup>6</sup>とは2015年11月現在W3Cが定める次のDOMの規格のProposed Recommendation(勧告案)である。DOMの規格は今までにLevel 1からLevel 3までがRecommendation(勧告)となっている。

- これらのプロパティのうち、`nodeValue`を除いてはすべて、読み取り専用である。
- ある要素に子要素がない場合にはその要素の`firstChild`や`lastChild`は`null`となる。
- ある要素に子要素がある場合、その`firstChild.previousSibling`や`lastChild.nextSibling`も`null`となる。`firstElementChild`などでも同様である。

プロパティに関する例はイベント処理のところで示す。

**課題 7.4** 67ページから始まるリストと68ページから始まるリストにおける`<select>`要素の子要素の数を調べよ。異なる場合にはその理由を述べよ。

`document.getElementsByTagName("select")[0].childNodes.length`を使うとよい。また、`document.getElementsByTagName("select")[0].childNodes[0].nodeType`も調べるとよいかもしれない。また、`childNodes`の代わりに`children`を用いたらどうなるかも調べるとよい。

<sup>6</sup><http://www.w3.org/TR/domcore/>

## 第8回 イベント

### 8.1 イベント概説

イベントとはプログラムに対して働きかける動作を意味する。Windows で動くプログラムにはマウスボタンが押された (クリック) などのユーザからの要求、一定時間経ったことをシステムから通知される (タイマーイベント) などありとあらゆる行為がイベントという概念で処理される。プログラムが開始されるということ自体イベントである。このイベントはプログラムの初期化をするために利用される。

イベントの発生する順序はあらかじめ決まっていないので、それぞれのイベントを処理するプログラムは独立している必要がある。このようにイベントの発生を順次処理していくプログラムのモデルをイベントドリブンなプログラムという。

### 8.2 イベント処理の方法

イベントは各要素内で発生するので、イベントを処理する関数を適当な要素に登録する。この関数をイベントハンドラーという。イベントハンドラーの登録には表 8.1にある属性に関数を登録する方法と、メソッドを用いて要素に登録する方法がある。最近の傾向としては、HTML 文書には表示するものだけを記述し、イベントハンドラーなどの JavaScript のプログラムに関することは書かないということがある。この授業ではメソッドを通じて登録する方法だけを紹介する。

要素にイベントハンドラーに登録するメソッドは次の `document` オブジェクトのメソッドを使う。

```
addEventListener(string type ,function listener ,[boolean useCapture])
```

引数の意味は次のとおりである。

- `type` はイベントの種類を示す文字列であり、表 8.1におけるイベントの属性名から `on` を取り除いたもの
- `listener` はイベントを処理する関数 (イベントハンドラー)
- `useCapture` はイベントの処理順序 (詳しくは後述)

要素からイベントハンドラーを削除する `document` オブジェクトのメソッドは次のとおりである。

```
removeEventListener(string type ,function listner ,[boolean useCapture])
```

イベントハンドラーが削除されるためには登録したときと同じ条件を指定する必要がある。

通常、イベントハンドラーの関数は通常、イベントオブジェクトを引数に取るように定義する。

## 8.3 DOM Level 2 のイベント処理モデル

DOM Level 2 のイベント処理モデルでは、あるオブジェクトの上でイベントが発生すると次の順序で各オブジェクトにイベントの発生が伝えられる。

1. 発生したオブジェクトを含む最上位のオブジェクトにイベントの発生が伝えられる。このオブジェクトにイベント処理関数が定義されていなければなにも起きない。
2. 以下順に DOM ツリーに沿ってイベントが発生したオブジェクトの途中にあるオブジェクトにイベントの発生が伝えられる (イベントキャプチャリング)。
3. イベントが発生したオブジェクトまでイベントの発生が伝えられる。
4. その後、DOM ツリーに沿ってこのオブジェクトを含む最上位のオブジェクトまで再びイベントの発生が伝えられる (イベントバブリング)。

DOM Level 2 のモデルではイベントが発生したオブジェクトは渡されたイベントオブジェクトの **target** プロパティで、イベントを処理している関数が登録されたオブジェクトは **currentTarget** で参照できる。また、イベントが発生したオブジェクトにイベントハンドラーが登録されていなくてもその親やその上の祖先にイベントハンドラーが登録されていると、イベントが発生する。

**addEventListener()** の 3 番目の引数を **false** にするとイベント処理はイベントバブリングの段階で、**true** にするとイベント処理はイベントキャプチャリングの段階で呼び出される。このイベントの伝播を途中で中断させるメソッドが **stopPropagation()** である。

通常、ブラウザの画面で右クリックをすると、ページのコンテキストメニューが表示される。このようなデフォルトの操作を行わせないようにする方法が **preventDefault()** である。

## 8.4 HTML における代表的なイベント

### 8.4.1 ドキュメントの onload イベント

Web ブラウザは HTML 文書を次のような順序で解釈し、実行する<sup>1</sup>。

1. 初めに **document** オブジェクトを作成し、Web ページの解釈を行う。
2. HTML 要素やテキストコンテンツを解釈しながら、要素やテキスト (ノード) を追加する。この時点では、**document.readyState** プロパティは **"loading"** という値を持つ。
3. **<script>** 要素が来ると、このこの要素を **document** に追加し、スクリプトを実行する。したがって、この時点で存在しない要素に対してアクセスすることはできない。つまり、**<script>** 要素より後に書かれている要素のはアクセスできない。したがって、この段階では後で利用するための関数の定義やイベントハンドラーの登録だけをするのがふつうである。
4. ドキュメントが完全に解釈できたら、**document.readyState** プロパティは **"interactive"** という値に変わる。

---

<sup>1</sup>この説明は少し簡略化している。詳しくは [3](1ページ) の 13.3 節を参照のこと



5. ブラウザはドキュメントオブジェクトに対し `DOMContentLoaded` イベントを発生させる。この時点では画像などの追加コンテンツの読み込みは終了していない可能性がある。
6. それらのことが終了すると、`document.readyState` プロパティの値が `"complete"` となり、`window` オブジェクトに対して `load` イベントを発生させる。

### 8.4.2 マウスイベント

表 8.1 は HTML 文書で発生するマウスイベントを記述したものである。

表 8.1: マウスイベントの例

イベントの発生条件	イベントの属性名
ボタンがクリックされた	<code>onclick</code>
ボタンが押された	<code>onmousedown</code>
マウスカーソルが移動した	<code>onmousemove</code>
マウスボタンが離された	<code>onmouseup</code>
マウスカーソルが範囲に入った	<code>onmouseover</code>
マウスカーソルが範囲から出た	<code>onmouseout</code>

表 8.2はマウスイベントのプロパティを表したものである。

表 8.2: マウスイベントのプロパティ

プロパティ	型	意味
<code>target</code>	<code>EventTarget</code>	イベントが発生したオブジェクト
<code>currentTarget</code>	<code>EventTarget</code>	イベントハンドラーが登録されているオブジェクト
<code>screenX</code>	<code>long</code>	マウスポインタのディスプレイ画面における $x$ 座標
<code>screenY</code>	<code>long</code>	マウスポインタのディスプレイ画面における $y$ 座標
<code>clientX</code>	<code>long</code>	マウスポインタのクライアント領域における相対的な $x$ 座標 (スクロールしている場合には <code>window.pageXOffset</code> を加える)
<code>clientY</code>	<code>long</code>	マウスポインタのクライアント領域における相対的な $y$ 座標 (スクロールしている場合には <code>window.pageYOffset</code> を加える)
<code>pageX</code>	<code>long</code>	マウスポインタの <code>document</code> 領域における相対的な $x$ 座標
<code>pageY</code>	<code>long</code>	マウスポインタの <code>document</code> 領域における相対的な $y$ 座標
<code>ctrlKey</code>	<code>boolean</code>	<code>cntrl</code> キーが押されているか
<code>shiftKey</code>	<code>boolean</code>	<code>shift</code> キーが押されているか

次ページへ続く

表 8.2: マウスイベントのプロパティ(続き)

プロパティ	型	意味
altKey	boolean	alt キーが押されているか
metaKey	boolean	meta キーが押されているか
button	unsigned short	マウスボタンの種類、0 は左ボタン、1 は中ボタン、2 は右ボタンを表す。
eventPhase	unsigned short	イベント伝播の現在の段階を表す。次の値がある。 Event.CAPTURING_PHASE(1)、Event.AT_TARGET(2)、 Event.BUBBLING_PHASE(3)

## 8.5 イベント処理の例

**実行例 8.1** 次の例は、form 要素内でユーザの入力を取り扱う代表的な input 要素の処理方法を示したものである。

このページでは大きさが指定された div 要素が 3 つ並び、その横にプルダウンメニュー、ラジオボタン、テキスト入力エリアと設定ボタンが並んでいる。

```
1 <!DOCTYPE html>
2 <head>
3 <meta charset="UTF-8"/>
4 <title>イベント処理</title>
5 <link rel="stylesheet" type="text/css" href="event.css"/>
6 <script type="text/ecmascript" src="event.js"></script>
7 </head>
8 <body>
9 <div class="block" id="Squares">
10 <div></div><div></div><div></div>
11 </div>
12 <form>
13 <select id="select">
14 <option value="red">赤</option>
15 <option value="yellow">黄色</option>
16 <option value="blue">青</option>
17 </select>
18 <div id="radio">
19 <div><input type="radio" value="red" name="color">赤</div>
20 <div><input type="radio" value="yellow" name="color">黄</div>
21 <div><input type="radio" value="blue" name="color">青</div>
22 </div>
```

```

23 <div>
24 色名<input type="text" size="10" id="colorName"></input>
25 <input type="button" value="設定" id="Set"></input>
26 </div>
27 <div id="position">
28 <div>クリック位置</div>
29 <div><div>clientX</div><input type="text" size="3" class="click">
30 <div>clientY</div><input type="text" size="3" class="click"></div>
31 <div><div>pageX</div><input type="text" size="3" class="click">
32 <div>pageY</div><input type="text" size="3" class="click"></div>
33 <div><div>pageXOffset</div><input type="text" size="3" class="click">
34 <div>pageYOffset</div><input type="text" size="3" class="click"></div>
35 <div>from Boundary</div>
36 <div><div> Left</div><input type="text" size="3" class="click">
37 <div>Top</div><input type="text" size="3" class="click"></div>
38 </div>
39 </form>
40 </body>
41 </html>

```

このリストではCSS ファイル、JavaScript ファイルが外部ファイルとなっている (5 行目と 6 行目)。

このページでは左にある 3 つの領域でマウスをクリックすると、クリックした位置の情報が右側下方の 8 つのテキストボックスに表示される。

- 上の 6 つはイベントオブジェクトのプロパティをそのまま表示している (表 7.3 参照)。
- 最下部の値はそれぞれの領域から見た相対位置を表している。

プルダウンメニュー、ラジオボタンで選択された色、テキストボックスでは最後にクリックされた領域の背景色を変えることができる。プルダウンメニュー、ラジオボタンでは値に変化があったときに設定が行われる。テキストボックスでは CSS3 で定義された色の表示形式が使用できる。設定するためには隣の「設定」ボタンを押す必要がある。

次のリストは上の HTML ファイルから読み込まれる CSS ファイルである。

```

1 .block {
2 display : inline-block;
3 vertical-align : middle;
4 }
5 .block > div {
6 width : 200px;
7 height : 200px;
8 display : inline-block;
9 }

```

```
10 #radio {
11 width:80px;
12 }
13 .click {
14 text-align:right;
15 }
16 form {
17 display : inline-block;
18 vertical-align : middle;
19 }
20 input, select{
21 font-size:25px;
22 }
23 #position > div {
24 text-align:center;
25 font-size: 20px;
26 }
27 #position > div > div {
28 display:inline-block;
29 width: 90px;
30 text-align: right;
31 font-size: 18px;
32 }
```

このスタイルシートは各要素の大きさや配置について指定している。

- 左側の3つの領域は **id** が **block** である要素の直下の **div** 要素であることから5行目のセレクトが適用される。
  - － これらの3つの部分は大きさが 200px の正方形となっている (6行目と7行目)。
  - － 横に並べるようにするため、**display** を **inline-block** に指定している。
  - － 背景色 (**background**) はプログラムから設定される。
- これらの領域全体を囲む **div** の垂直方向の配置の位置が3行目で定められている。
- 右上方のプルダウンメニューのフォントの大きさは20行目からのセレクトで定義されているが、文書のロード時に 30px に変更している。
- ラジオボタンの要素の幅は10行目からのセレクトで定められている。
- クリックしたときの位置情報を示す部分の CSS は23行目以下で定められている。
  - － テキストボックスがない行は23行目から26行目が適用されている。

- テキストボックスがある行は、テキストボックスの位置をそろえるために、その前の文字列を `div` 要素の中に入れ、幅 (29 行目)、テキストの配置位置 (30 行目) とフォントの大きさ (31 行目) を指定している。

```
1 window.onload = function() {
2 var Squares = document.getElementById("Squares");
3 var Select = document.getElementById("select");
4 var ColorName = document.getElementById("colorName");
5 var Radio = document.getElementById("radio");
6 var Set = document.getElementById("Set");
7 var Inputs = document.getElementsByClassName("click");
8 var lastClicked = Squares.children[1];
```

ここでは `id` 属性を持つ要素すべてを得ている。また、8 行目で、最後にクリックされた正方形のオブジェクトを記憶する変数を初期化している。

```
9 Squares.children[0].style.background = "red";
10 Squares.children[1].style.background = "yellow";
11 Squares.children[2].style.background = "blue";
12 Select.style.fontSize = "30px";
```

- `id` が `Squares` である `div` 要素の下には 3 つの `div` 要素がある。これらの要素の背景色を指定するために、`children` プロパティを用いて参照している。。
- スタイルを変更するためには `style` プロパティの後に属性を付ける。
- 9 行目から 11 行目では左の 3 つの領域の背景色 (`background`) を設定している。
- フォントの大きさを指定する CSS 属性は `font-size` であるが、これは `-` を含んでいるのでそのままでは JavaScript の属性にならない。このような属性は `-` を省き、次の単語を大文字で始めるという規約がある。したがって、この場合には `fontSize` となる。

次のリストは正方形の領域がクリックされたときのイベントハンドラーを定義している部分である。イベントハンドラーは 3 つの正方形を含んだ `div` 要素につけている。

```
13 Squares.addEventListener("click",function(E){
14 Inputs[0].value = E.clientX;
15 Inputs[1].value = E.clientY;
16 Inputs[2].value = E.pageX;
17 Inputs[3].value = E.pageY;
18 Inputs[4].value = window.pageXOffset;
19 Inputs[5].value = window.pageYOffset;
20 var R = E.target.getBoundingClientRect();
21 Inputs[6].value = E.pageX-R.left;
```

```

22 Inputs[7].value = E.pageY-R.top;
23 ColorName.value = E.target.style.background;
24 lastClicked = E.target;
25 },false);

```

- 表示するためのテキストボックスのリストは7行目で得ている。
- 14行目から19行目でそれぞれのテキストボックスに該当する値を入れている。
- 21行目と22行目では該当する領域からの相対位置を求めている。そのためにはそれぞれの領域がページの先頭からどれだけ離れているかを知る必要がある。その情報を得るメソッドが `getBoundingClientRect()` である。

このメソッドは次のようなプロパティを持つ `ClientRect` オブジェクトを返す。

プロパティ	解説
<code>top</code>	領域の上端の Y 座標
<code>bottom</code>	領域の下端の Y 座標
<code>left</code>	領域の左端の X 座標
<code>right</code>	領域の右端の X 座標
<code>width</code>	領域の幅
<code>height</code>	領域の高さ

- これらの値とイベントが起きた位置の情報から領域内での位置が計算できる。
- 24行目ではクリックした正方形のオブジェクトを更新している。

ここでは3つの入力方法に対して、イベントハンドラーを定義している。

```

26 Select.addEventListener("change", function(){
27 lastClicked.style.background = Select.value; },false);
28 Radio.addEventListener("click", function(E){
29 alert(E.target.tagName);
30 if(E.target.tagName === "DIV") {
31 E.target.firstChild.checked = true;
32 }
33 console.log("----" + Radio.value);
34 lastClicked.style.background = Radio.querySelector("input:checked").value;
35 },false);
36 Set.addEventListener("click", function(){
37 lastClicked.style.background = ColorName.value; }, false);
38 }

```

- 26行目から27行目ではプルダウンメニューに `change` イベントのハンドラーを定義している。直前にクリックされた部分の背景色を変えている。

- 28 行目から 34 行目はラジオボタンのあるところがクリックされたときのイベントハンドラーを設定している。ラジオボタンは **name** 属性が同じ値のものが一つのものとして扱われる (どこか一つだけオンになる)。
  - － 29 行目ではクリックされた場所の要素名を表示している (通常は必要ない)。HTML の要素名は小文字で書かれていても大文字に変換されることが確認できる。
  - － HTML のリストの 28 行目にある **div** 要素にクリックのイベントハンドラーを登録する (28 行目から 29 行目) ので、クリックされた場所がラジオボタンの上でなくても、この範囲内であればイベントは発生する。
  - － クリックされた場所がラジオボタンの上でなければ (ラジオボタンの親要素の **div** 要素が **E.target** となる) のでその **firstChild** が値を変えるラジオボタンになる (31 行目)。
  - － ラジオボタンの集まりには **value** プロパティがない。チェックされている場所を探すために **querySelector("input:checked")** を用いて、チェックされている要素を探す (33 行目の右辺)。
  - － 色の設定はテキストボックスの値を代入する (36 行目)。

**課題 8.1** 次のようなオブジェクトをもとにプルダウンメニューを作成する JavaScript の関数を作成しなさい。

関数に渡すオブジェクト:{"red": "赤", "orange": "橙", "yellow": "黄色", "green": "緑"}  
作成される DOM

```
<select>
 <option value="red">赤</option>
 <option value="orange">橙</option>
 <option value="yellow">黄色</option>
 <option value="green">緑</option>
</select>
```

また、18 行目から 22 行目にあるようなラジオボタンを作成する関数も作成しなさい。

**実行例 8.2** 次の例は 3 つのプルダウンメニューが順に年、月、日を選択できるものである。年や月が変化 (**change** イベントが発生) すると日のプルダウンメニューの日付のメニューがその年月にある日までに変わるようになっている。

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
5 <title>日付</title>
6 <script type="text/ecmascript">
7 //<![CDATA[
8 window.onload = function(){
```

```
9 function makeSelectNumber(from, to, prefix, suffix, id, parent){
10 var i, option;
11 var Select = makeElm("select", {"id":id}, parent);
12 for(i=from; i<=to; i++) {
13 option = makeElm("option",{ "value":i}, Select);
14 makeTextNode(prefix+i+suffix,option);
15 }
16 return Select;
17 };
```

このプログラムは8行目から始まる `window.onload` のイベントハンドラーの中にある。

9行目から17行目では指定された範囲の値を選択できるプルダウンメニューを作成する関数 `makeSelectNumber()` を定義している。

- 引数は順に、下限値 (**from**)、上限値 (**to**)、数字の前後に付ける文字列 (**prefix** と **suffix**)、プルダウンメニュー の **id** 属性の属性値 (**id**) と親要素 (**parent**) である。
- 11行目で **select** 要素を作成している。作成のために `makeElm` 関数 (18行目から26行目で定義) を呼び出している。
- 12行目から15行目で **option** 要素を順に作成している。
- 14行目でプルダウンメニューに表示される文字列をテキストノードを作成する関数 `makeTextNode()` (28行目から30行目) を呼び出している。

```
18 function makeElm(name, attribs, parent) {
19 var elm = document.createElement(name);
20 var attrib;
21 for(attrib in attribs) {
22 elm.setAttribute(attrib,attribs[attrib]);
23 };
24 if(parent) parent.appendChild(elm);
25 return elm;
26 }
```

18行目から26行目で与えられた要素を作成する関数 `makeElm()` を定義している。

- 引数は順に、要素名 (**name**)、属性値のリスト (**attribs**) と親要素の指定である。
- 指定された要素を作成し (19行目) て、与えられた属性とその属性値がメンバーになっているオブジェクトの値を順に選んで属性値を設定する (21行目から23行目)。
- 親要素が指定されている場合には、作成した要素を子要素として付け加える (24行目)。
- 作成した要素を戻り値にする (25行目)。



```
27 function makeTextNode(text,parent) {
28 parent.appendChild(document.createTextNode(text));
29 };
```

関数 `makeTextNode()` は指定された文字列を基にテキストノードを作成し、指定された親要素の子要素に設定している。

```
30 var Form = document.getElementById("menu");
31 var Year = makeSelectNumber(2000,2020,"","年","year", Form);
32 var Month = makeSelectNumber(1,12,"","月","month", Form);
33 var Days = [];
34 for(d=28; d<=31; d++) {
35 Days[d] = makeSelectNumber(1,d,"","日","day");
36 }
```

56 行目の `form` 要素内にプルダウンメニューを作成する。プルダウンメニューで表示される日付は実行当日になるように設定される。

- 30 行目では `form` 要素を得ている。
- 31 行目と 32 行目ではそれぞれ年、月のプルダウンメニューを作成して `form` 要素の子要素にしている。
- 28 日から 31 日まであるプルダウンメニューを 4 つ作成して配列に格納している (34 行目から 36 行目)。ここでの関数呼び出しは最後の親要素が省略されているので、別の要素の子要素としては登録されない。

```
37 var d, today = new Date();
38 Year.value = today.getFullYear();
39 Month.value = today.getMonth()+1;
40 d = new Date(Year.value, Month.value,0).getDate();
41 Form.appendChild(Days[d]);
42 Form.children[2].value = today.getDate();
```

ここでは、実行時の日付がプルダウンメニューの初期値として表示されるようにしている。

- `Date()` オブジェクトを引数なしでよぶと、実行時の時間が得られる (37 行目)。<sup>2</sup>
- 38 行目で年を得ている。`Date` オブジェクトの `getFullYear()` メソッドは西暦の下 2 桁しか返さないので使わないこと。
- 39 行目で月を得ている。`getMonth()` メソッドの戻り値は 0(1 月) から 11(12 月) である。
- 41 行目と 42 行目では、得られた年と月をそれぞれのプルダウンメニューに設定している。月の値を 1 増やしていることに注意すること。

---

<sup>2</sup>この時間は 1970 年 1 月 1 日午前 0 時 (GMT) この時刻を UNIX エポックという。からの経過時間であり、単位はミリ秒である。

- **Date()** オブジェクトを年、月、日 (さらに、時、分、秒もオプションの引数として与えられる) を与えて、インスタンスを作成できる。実行当日の翌月の 0 日を指定することで翌月の 1 日の 1 日前の日付に設定できる。その日付から **getDate()** メソッドで現在の月の最後の日 が得られる<sup>3</sup>(40 行目)。

なお、**Date** オブジェクトの **Day()** は曜日を得るメソッドで 0(日曜日) から 6(土曜日) の値が 返る。

- 41 行目で、この日数を持つプルダウンメニューを子要素として追加し、42 行目で、日の選 択する値を設定している。

```
43 Form.addEventListener("change", function(){
44 var d2 = Form.children[2].value
45 d = new Date(Year.value, Month.value, 0).getDate();
46 if(d != Form.children[2].children.length) {
47 Form.replaceChild(Days[d],Form.children[2]);
48 Form.children[2].value = Math.min(Form.children[2].length, d2);
49 }
50 },false);
51 }
52 //]]>
53 </script>
```

ここではプルダウンメニューの値が変化したイベント (**change**) ハンドラーを登録している。

- イベントハンドラーは **form** 要素につけている。
- 44 行目で、現在表示されている日を保存している。
- 45 行目で、現在、プルダウンメニューで設定されている年月の日数を求めている。
- この日数と現在表示されている日数のプルダウンメニューの日数が異なる場合には (46 行目) 子ノードを入れ替える (47 行目)。さらに、初期値を現在の値と月の最終日の小さいほうに設 定する (48 行目)。

```
54 </head>
55 <body>
56 <form id="menu"></form>
57 </body>
58 </html>
```

ここでは HTML 文書内で表示される要素を記述している。ここでは **body** 要素内に **form** 要素があ るだけである。

---

<sup>3</sup>ネットの情報を参考にした。**Date** オブジェクトのコンストラクタは範囲外の日時を指定しても正しい日時に解釈して くれる。

## 第9回 PHPの超入門

### 9.1 Web サーバーの基礎知識

#### 9.1.1 サーバーの重要な設定項目

Web サーバーは hypertext transfer protocol(HTTP) を利用して、クライアントプログラム (Web ブラウザなど) からの要求を処理する。Apache はこのサービスを提供するものとして有名である。Apache の設定ファイル `httpd.conf` の中で設定される重要な項目は次のとおりである。

- ポート番号 : TCP/IP のサービスを識別するための番号で、次の 3 種類に分けられる<sup>1</sup>。

種類	範囲	内容
System Ports(Well Known Ports)	1 番 ~ 1023 番	assigned by IANA
User Ports, (Registered Ports)	1024 番 ~ 49151 番	assigned by IANA
Dynamic Ports(Private or Ephemeral Ports)	49152 番 ~ 65535 番	never assigned

IANA(Internet Assigned Numbers Authority) は DNS Root, IP アドレスやインターネットプロトコルのポート番号などを管理している団体である。

- ドキュメントルート : Apache では Web サーバーに直接要求できるファイルはこのフォルダ (ディレクトリ) の下にあるものだけである。

なお、HTTP の System ポート番号は 80 番となっている<sup>2</sup>。

#### 9.1.2 CGI

Common Gateway Interface(CGI) とは Web コンテンツを動的に生成する標準の方法で、Web サーバー上では Web サーバーと Web コンテンツを生成するためのインターフェイスを与える。この授業では CGI のプログラムは PHP(PHP: Hypertext Preprocessor) を使用する。

#### 9.1.3 Web サーバーのインストール

Web サーバー (Apache) や PHP をインストールし、HTTP のサーバー上で PHP と連携させるためにはいくつかの設定をする必要がある。これらのインストールを一括で行うことができるパッケージも存在する。

<sup>1</sup><https://tools.ietf.org/html/rfc6335#section-6>

<sup>2</sup>ポート番号の一覧は次のところにある。

<https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml>

**XAMPP** XAMPP は Apache、MySQL、Perl と PHP を一括してインストールするパッケージである。インストール時にはいくつかのパッケージをインストールしない選択も可能である。XAMPP をインストールしたときに注意する点は次のとおりである。

- インストール時に Apache をサービスとして実行するとパソコンを立ち上げた時に Apache を起動させる手間が省ける。
- 標準の文字コードは **UTF-8** である。
- PHP の設定でタイムゾーンがヨーロッパ大陸になっている。タイムスタンプを利用するプログラムがうまく動かない場合があるので注意すること<sup>3</sup>。

## 9.2 PHP とは

日本 PHP ユーザー会のホームページ<sup>4</sup>には次のように書かれている。

PHP は、オープンソースの汎用スクリプト言語です。特に、サーバサイドで動作する Web アプリケーションの開発に適しています。言語構造は簡単で理解しやすく、C 言語の基本構文に多くを拠っています。手続き型のプログラミングに加え、(完全ではありませんが) オブジェクト指向のプログラミングも行うことができます。

なお、PHP は Web アプリケーションのためではなく、通常のプログラミング言語としても使用できる。通常のプログラミング言語として使用するためには、コマンドプロンプトから PHP が実行できるように、環境変数 PATH に `php.exe` があるフォルダを追記しておくといよい。

また、コマンドプロンプトから `php -S localhost:8080` を実行すると起動したフォルダをドキュメントルートとするポート番号が 8080 のサーバーが立ち上がる。

なお、このテキストの PHP の仕様に関する説明は日本 PHP ユーザー会から多く引用している。

## 9.3 PHP プログラムの書き方

通常のサーバーの設定では PHP のプログラムの拡張子は `php` である。Web アプリケーションではクライアントから拡張子が `php` のファイルが要求されると、サーバーが PHP のプログラムを処理して、その出力がクライアント側に送られる。

PHP で処理すべき部分は `<?php` と `?>` の間に記述する。それ以外の部分はそのまま出力される。このブロックは複数あってもかまわない。

## 9.4 データの型

PHP は 8 種類の基本型をサポートする。

---

<sup>3</sup>`date_default_timezone_get()` 関数で調べることができる。

<sup>4</sup><http://www.php.gr.jp>

- 4 種類のスカラー型
  - 論理値 (boolean)  
TRUE と FALSE の値をとる。小文字で書いてもよい。
  - 整数 (integer)  
整数の割り算はない。つまり  $1/2$  は 0 ではなく 0.5 となる。なお、php 7 では整数の割り算の結果を与える `intdiv()` 関数が導入された。たとえば、`intdiv(1,2)` の結果は 0 となる。
  - 浮動小数点数 (float, double も同じ)
  - 文字列 (string)  
JavaScript と同様に文字列の型はあるが、文字の型はない。
- 2 種類の複合型
  - 配列 (array)  
`array()` を用いて作成する。引数にはカンマで区切られた `key=>value` の形で定義する。`key=>` の部分はなくてもよい。このときは `key` として `0,1,2,...` が順に与えられる。
  - オブジェクト (object)
- 2 種類の特別な型:
  - リソース (resource)  
オープンされたファイル、データベースへの接続、イメージなど特殊なハンドル
  - ノル (NULL)  
ある変数が値を持たないことを示す。

## 9.5 変数

PHP の変数の特徴は次のとおりである。

- 変数は特に宣言しない。
- 変数名は `$` で始まる。
- 変数に値を代入すればその値はすべてコピーされる。
- コピーではなく参照にしたい場合には変数の前に `&` を付ける。

**実行例 9.1** 次の例は単純な値を代入してある変数の参照をチェックしている。

```
1 <?php
2 $a = 1;
3 $b = $a;
4 print "1:\$a=$a, \$b=$b\n"; // 1:$a=1, $b=1
```

```
5 $a = 2;
6 print "2:\$a=$a, \$b=$b\n"; // 2:$a=2, $b=1
7 $b = &$a;
8 print "3:\$a=$a, \$b=$b\n"; // 3:$a=2, $b=2
9 $a = 10;
10 print "4:\$a=$a, \$b=$b\n"; // 4:$a=10, $b=10
11 ?>
```

- 3行目で変数 `$a` の値を変数 `$b` の代入している。4行目で見るように両者の値は同じである。
- 5行目で `$a` の値を変更しても、変数 `$b` の値は変化しない(6行目)。
- 7行目では変数 `$b` に変数の参照を代入している。この場合には変数 `$a` の値を変更すると(9行目) 変数 `$b` の値も変更される(10行目)。

### 9.5.1 可変変数

ある変数の値が文字列のとき、その前に`$`をつけると、その文字列が変数名として使える。

**実行例 9.2** 次のリストは可変変数の使用例である。

```
1 <?php
2 $a = 1;
3 $b = 2;
4
5 $c = "a";
6 print "\$ $c = " . $$c . "\n"; // $c = 1
7
8 $c = "b";
9 print "\$ $c = " . $$c . "\n"; // $c = 2
10 ?>
```

- 変数`$a` と `$b` にそれぞれ1と2を代入している(1行目と2行目)。
- 変数`$c` に文字列" a"を代入し、`$$c` を出力させると`$a` の値が表示される(5行目と6行目)。
- 同様に、変数`$c` に文字列" b"を代入して、`$$c` を出力させると`$b` の値が表示される(8行目と9行目)。

### 9.5.2 定義済み変数

PHP にはいくつかの定義済みの変数が存在する。そのうち、スーパーグローバルと呼ばれる変数はグローバルスコープを持つ定義済みの変数である(表 9.1)。

表 9.1: スーパーグローバル

変数名	変数の内容
<code>\$GLOBALS</code>	グローバルスコープで使用可能なすべての変数への参照
<code>\$_SERVER</code>	サーバー情報および実行時の環境情報
<code>\$_GET</code>	HTTP 通信における GET によるパラメータ
<code>\$_POST</code>	HTTP 通信における POST によるパラメータ
<code>\$_FILES</code>	HTTP 通信でファイルアップロードに関する情報
<code>\$_COOKIE</code>	HTTP 通信におけるクッキーの情報
<code>\$_SESSION</code>	HTTP 通信におけるセッション中に保持される情報
<code>\$_REQUEST</code>	HTTP 通信におけるパラメータ (GET や POST の区別をしない)
<code>\$_ENV</code>	環境変数のリスト
<code>\$argc</code>	コマンドプロンプトから実行したときに与えられた引数の数
<code>\$argv</code>	コマンドプロンプトから実行したときに与えられた引数のリスト

これらの変数のうち、**\$ENV** `$argc` と `$argv` 以外は CGI で使用する。

`$argv` は PHP をコマンドプロンプトから実行したときの (空白で区切られた) パラメータを保持する文字列の配列である。`$argv[0]` には実行している PHP ファイル名が渡される。`$argc` は `count($argv)` と同じ値である。

## 9.6 文字列

### 9.6.1 文字列の定義方法

PHP では文字列を定義する方法が 3 通りある。

- シングルクォート (') ではさむ。  
中に書かれた文字がそのまま定義される。
- ダブルクォート (") ではさむ。  
改行などの制御文字が有効になる。また、変数はその値で置き換えられる。これを変数が展開されるという。
- ヒアドキュメント形式  
複数行にわたる文字列を定義できる。<<<の後に識別子を置く。文字列の最後は行の先頭に初めの識別子を置き、そのあとに文の終了を表す; を置く。平ドキュメントの終わりを示す識別子は行の先頭から書く必要がある。識別子の前に空白などを入れてはいけない。

**実行例 9.3** 次の例はいろいろな文字列の動作を確認するものである。

```

1 <?php
2 print 'string1 \' :abcd\n';
3 print "string2 \" :abcd\n"; // string1 ' :abcd\nstring2 " :abcd
4
5 $a = 1;
6 print 'string3 \' :$a bcd\n';
7 print "string4 \" :$a bcd\n"; //string3 ' :$a bcd\nstring4 " :1 bcd
8 print "string5 \" :{$a}bcd\n"; //string5 " :1bcd
9
10 $b = array(1,2,3);
11 print "string6:$b\n"; //string6:Array
12 print "string7:$b[1]aa\n"; //string7:2aa
13 print "string7:$b[$a]aa\n"; //string7:2aa
14
15 print <<<_EOL_
16 string8:
17 aa
18 \$a = $a
19 _EOL_; //string8:
20 // aa
21 // $a = 1

```

- シングルクォートの文字列では改行を意味する `\n` がそのまま出力されているのに対し、ダブルクォートの文字列では改行に変換されている。
- シングルクォートの文字列では変数名がそのまま出力されているのに対し、ダブルクォートの文字列では変数の値に変換されている。この場合、変数として解釈されるのは変数名の区切りとなる文字 (空白など) である。そのため7行目では変数の後に空白を入れている。
- 空白を入れたくない場合などは変数名全体を `{ }` で囲む (8行目)。
- 変数が配列の場合には添え字を個々に指定しなくてはならない。その指定には別の変数が使用できる。
- ヒアドキュメントでは途中の改行もそのまま出力される。また、変数は展開される。

### 9.6.2 文字列を取り扱う関数

PHP では文字列を取り扱う関数が用意されている。これらの関数は文字列のメソッドとなっていないことに注意する必要がある。Unicode 文字列を取り扱う必要がある場合には `mb_` で始まる関数群を使用する必要がある。具体的な関数の例は PHP のサイトで確認すること。



## 9.7 式と文

文の最後には必ず;`;`が必要である。その他はほとんど C 言語と同じ構文が使える。演算子「`+`」は JavaScript と異なり、通常の数演算となる。文字列に対して「`+`」を用いると数に直されて計算される。文字列の接続には「`.`」を用いる。

**実行例 9.4** 次の例は文字列の演算子の確認である。

```
1 <?php
2 print 1 + "2" . "\n"; // 3
3 print "1" + "2" . "\n"; // 3
4 print "1" . "2" . "\n"; // 12
5 print 1 . 2 . "\n"; // 12
6
7 print "1a" + "2" . "\n"; // 3
8 print "a1" + "2" . "\n"; // 2
9 print "0x1a" + "2" . "\n"; // 28
```

- `+`演算子は常に数としての加法として扱われ、`.`は文字列の接続として扱われる。
- 文字列が数として不正な文字を含むとその直前まで解釈した値を返す (7 行目と 8 行目)。
- 16 進リテラルも正しく解釈される (9 行目)。

比較演算子についても C 言語と同様のもののほかに、データ型も込めて等しいことをチェックする `===` 演算子と `!==` 演算子がある。

表 9.2<sup>5</sup> は変数の状態を調べる関数が引数の値によりどのようなになるか、また、論理値が必要なところでどのようなになるかを表している。

- `gettype()` 変数の型を調べる
- `isempty()` 変数が空であることを調べる
- `is_null()` 変数の値が `NULL` であるかを調べる
- `isset()` 変数の値がセットされていて、その値が `NULL` でないことを調べる
- `if(変数)` としたときに `TRUE` となるかどうか

いろいろな値を論理値として評価した結果が JavaScript と異なる場合があるので注意が必要である。

表 9.3 は比較演算子 `==` の結果をまとめたものである。この演算子ではいろいろなものを数に変換してから判定をする。通常のプログラムでは厳密な比較をするのがよい。

**課題 9.1** PHP と JavaScript で `if(変数)` と `==` の結果が異なるものを指摘しなさい。

PHP では分岐、繰り返しなどの制御構造は C 言語と同じように `if` 文、`for` 文、`while` 文と `switch` 文などがある。

<sup>5</sup> この表と次の表は PHP ユーザー会の HP から転用した。

表 9.2: PHP 関数による `$x` の比較

式	<code>gettype()</code>	<code>empty()</code>	<code>is_null()</code>	<code>isset()</code>	boolean : <code>if(\$x)</code>
<code>\$x = "";</code>	string	TRUE	FALSE	TRUE	FALSE
<code>\$x = null;</code>	NULL	TRUE	TRUE	FALSE	FALSE
<code>var \$x;</code>	NULL	TRUE	TRUE	FALSE	FALSE
<code>\$x</code> が未定義	NULL	TRUE	TRUE	FALSE	FALSE
<code>\$x = array();</code>	array	TRUE	FALSE	TRUE	FALSE
<code>\$x = false;</code>	boolean	TRUE	FALSE	TRUE	FALSE
<code>\$x = true;</code>	boolean	FALSE	FALSE	TRUE	TRUE
<code>\$x = 1;</code>	integer	FALSE	FALSE	TRUE	TRUE
<code>\$x = 42;</code>	integer	FALSE	FALSE	TRUE	TRUE
<code>\$x = 0;</code>	integer	TRUE	FALSE	TRUE	FALSE
<code>\$x = -1;</code>	integer	FALSE	FALSE	TRUE	TRUE
<code>\$x = "1";</code>	string	FALSE	FALSE	TRUE	TRUE
<code>\$x = "0";</code>	string	TRUE	FALSE	TRUE	FALSE
<code>\$x = "-1";</code>	string	FALSE	FALSE	TRUE	TRUE
<code>\$x = "php";</code>	string	FALSE	FALSE	TRUE	TRUE
<code>\$x = "true";</code>	string	FALSE	FALSE	TRUE	TRUE
<code>\$x = "false";</code>	string	FALSE	FALSE	TRUE	TRUE

## 9.8 配列

### 9.8.1 配列の基礎

PHP の配列は `array()` 関数で作成する。配列には C 言語などのような数字を添え字とするものとキーと値の組み合わせであるものがある。

**実行例 9.5** 次のリストは配列の取り扱いに関する簡単な例である。

```

1 $a = array(0,10,20,30);
2 $b = $a;
3 $a[0] = 5;
4 print_r($a);
5 print_r($b);

```

- 1 行目で 4 つの値を持つ配列を定義
- 2 行目で別の変数に値を代入
- 3 行目で初めの配列の先頭の値を変更

表 9.3: == による緩やかな比較

	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	" "
TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
1	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE
-1	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
"1"	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE
array()	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
"php"	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
" "	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE

- 4行目と5行目で配列の内容を出力するために `print_r()` 関数を用いている。

出力結果は次のとおりである。

```
Array
(
 [0] => 5
 [1] => 10
 [2] => 20
 [3] => 30
)
Array
(
 [0] => 0
 [1] => 10
 [2] => 20
 [3] => 30
)
```

`$b=$a` で代入された変数 `$b` は `$a` とは別の領域が割り当てられていることがわかる。

**課題 9.2** 実行例 9.1 で `print_r()` 関数の代わりに `var_dump()` 関数を用いて出力がどのように変わるか確認しなさい。

キーと値を組み合わせた配列については次の項で解説をする。

### 9.8.2 配列に関する制御構造

JavaScript のオブジェクトのプロパティを列挙するのに利用した `for(... in ...)` に相当する `foreach` がある。この構文は形をとる。

`foreach(「配列名」 as [キー =>] 値)`

「キー」と「値」のところは単純な変数を置くことができる。「キー」の部分は省略可能である。

**実行例 9.6** 次の例は `foreach` の使用例である。

```
1 $a = array("red","yellow","blue");
2 foreach($a as $val) {
3 print "$val\n";
4 }
5 foreach($a as $key=>$val) {
6 print "$key:$val\n";
7 }
```

1 行目で単純な配列を定義して、2 行目から 4 行目と 5 行目から 7 行目でこの配列の要素をすべて表示させている。この部分の出力は次のとおりである。

```
red
yellow
blue
0:red
1:yellow
2:blue
```

単純な配列のときは「キー」の値は 0,1,2,... を順にとる (出力 4 行目から 6 行目)。ここでは `$a[$key]=$val` の関係が成立していることに注意すること。

```
1 $a = array("red"=>"赤","yellow"=>"黄","blue"=>"青");
2 foreach($a as $val) {
3 print "$val\n";
4 }
5 foreach($a as $key=>$val) {
6 print "$key:$val\n";
7 }
```

1 行目では JavaScript におけるオブジェクトリテラルの形式に似た連想配列を定義して、2 行目から 4 行目と 5 行目から 7 行目で以前と同じ形で配列の内容を表示している。この出力は次のようになる。

```
赤
黄
青
```

```
red:赤
yellow:黄
blue:青
```

**実行例 9.7** 次の例は PHP の配列に関する特別な状況を説明するためのものである。

```
1 $b = array();
2 $b[3] = "3rd";
3 $b[0] = "0";
4 print count($b)."\n";
5 foreach($b as $key=>$val) {
6 print "$key:$val\n";
7 }
8 for($i=0;$i<count($b);$i++) {
9 if(isset($b[$i])) {
10 print "$i:$b[$i]\n";
11 }
12 }
```

この出力は次のようになる。

```
2 //5 行目の出力
3:3rd // 6 行目の出力 1 回目
0:0 // 6 行目の出力 2 回目
0:0 //1 0 行目の$i=0 のときの値
```

- 1 行目で配列を初期化し、その後、3 番目と先頭の要素に値を代入している。
- 関数 `count()` は配列の大きさ (正確には配列にある要素の数) を求めるものである。2 つの値しか定義していないので 2 となる。
- `foreach` で配列の要素を渡るときは定義した順に値が設定される。`count($b)` の値が 2 であったことから `$b[3]` の出力は現れない。

### 9.8.3 配列に関する関数

PHP には配列の処理に関するいろいろな関数 (もどき) がある。

`list()` `list()` は配列の個々の要素をいくつかの変数にまとめて代入できる。これは関数ではなく、言語構造である。

```
list($first,$second, $third) = array(1,2,3,4,5);
```

この結果は、変数 `$first`、`$second` と `$third` にそれぞれ 1,2,3 が代入される。途中に変数を書かなければ、その分は飛ばされる。

```
list($first, , $third) = array(1,2,3,4,5);
```

この場合は変数`$first` と `$third` だけ代入が行われる。

**array\_pop() と array\_push()** `array_pop()` はもとの配列から最後の要素を取り除いて配列の大きさを一つ少なくし、取り除いた要素を戻り値とする。`array_push()` は与えられた引数を順に配列の最後に付け加えて配列の大きさを増加させる。

```
$a = array(1,2,3,4);
print array_pop($a); // 4 が出力される
array_push($a,10,20); // $a は array(1,2,3,10,20) となる
```

配列の先頭に対して要素を取り出したり (`array_shift()`)、追加する (`array_unshift()`) することもできる。

**配列のキーと値の存在の判定** `in_array($needle, $array)` は与えられた配列 (`$array`) 内に指定した値 (`$needle`) が存在するかを調べる関数である。値の型まで比較するためには省略可能な 3 番目の引数を `TRUE` にすればよい。存在すれば `TRUE` が返る。

`array_key_exists($needle, $array)` は与えられた配列 (`$array`) 内に指定したキー (`$needle`) が存在するかを調べる関数である。存在すれば `TRUE` が返る。

**並べ替え** PHP には配列を並べ替える関数がいくつか用意されている。`sort()` は各要素を昇順に並べかえる。降順に並べ替えるには `rsort()` 関数を用いる。このほか、連想配列のキーで並べ替える `krsort()` と `krsort()` 関数、ユーザー指定の比較関数を用いて並べ替える `usort()` 関数などがある。詳しくは PHP ユーザー会のマニュアルを参考にする。

**配列からランダムな値を取り出す** `shuffle()` は与えられた配列の要素をランダムに並べ替える関数である。配列をそのままにしておきたい場合には `array_rand()` 関数を用いる。この関数は取り出すキーの値を返す。

**配列の切り出しと追加** 配列の要素の一部を取り出して別の配列にしたり (`array_slice()`)、配列の一部を別の要素に置き換えたり (`array_splice()`) できる。

`array_slice($array, $start, $length)` は与えられた配列 `$array` の `$start` の位置から長さ `$length` の部分を配列として返す関数である。

- `$length` が与えられないときは、配列の最後まで切り取られる。
- 元の配列は変化しない。
- `$start` の値が負のときは、配列の最後から数えた位置となる。

```
$a = array(0,1,2,3,4,5);
$res = array_slice($a,3); // $res は array(3,4,5), $a は変化しない
$res = array_slice($a,-2); // $res は array(4,5)
$res = array_slice($a,-3,2); // $res は array(3,4)
```

`array_splice($array, $start, $length, $replace)` は配列 `$array` の `$start` の位置から長さ `$length` の部分を切り取り戻り値とする。切り取られた部分には配列 `$replace` の要素が入る。なお、引数 `$start` については `array_slice()` と同じ扱いとなる。

```
$a = array(0,1,2,3,4,5);
$r = array_splice($a,3,1,array("a","b")); // $r は array(3)
 // $a は array(0,1,2,"a","b", 4, 5)
```

課題 9.3 `array_splice()` 関数を用いて、`array_pop()`、`array_push()`、`array_shift()`、`array_unshift()` 関数を実現しなさい。

## 9.9 関数

### 9.9.1 PHP の関数の特徴

PHP の関数の特徴は次のとおりである。

- 関数はキーワード **function** に引き続いて `()` 内に、仮引数のリストを書く。そのあとに `{}` 内にプログラム本体を書く。
- 引数は値渡しである。参照渡しをするときは仮引数の前に `&` を付ける。
- 関数のオーバーロードはサポートされない。
- 関数の宣言を取り消せない。
- 仮引数の後に値を書くことができる。この値は引数がなかった場合のデフォルトの値となる。デフォルトの値を与えた仮引数の後にデフォルトの値がない仮引数を置くことはできない。
- 関数は使用される前に定義する必要はない。
- 関数内で関数を定義できる。関数内で定義された関数はグローバルスコープに存在する。ただし、外側の関数が実行されないと定義はされない。
- 関数内の変数はすべてローカルである。
- 関数の外で定義された変数は参照できない。関数外で宣言された変数を参照するためにはスーパーグローバル `$GLOBAL` を利用する。
- 関数の戻り値は **return** 文の後の式の値である。複数の値を関数の戻り値にしたいときは配列にして返せばよい。

なお、PHP 7 では関数の仮引数や戻り値に対して型を宣言できるようになった。使用できる型には `array`、`bool`、`float`、`string` などがある。詳しくは PHP のサイトで確認すること。

PHP には外部ファイルを読み込むための関数 `include()` と一度だけ指定されたファイルを読み込む `require_once()` がある。このファイルの中で定義された変数は、読み込むテキストを直接記述したときと同じである。

### 9.9.2 関数の定義の例

実行例 9.8 次の例はユーザー定義関数の使用例である。

```
1 <?php
2 function example($a, $as, &$b, $f=false) {
3 print "\$a = $a\n";
4 print_r($as);
5 print "\$b = $b\n";
6 if(!isset($x)) $x = "defined in function";
7 print "\$x = $x\n";
8 if($f) {
9 print "\$GLOBALS['x'] = ". $GLOBALS['x']."\n";
10 }
11 $a = $a*2;
12 $as[0] += 10;
13 $b = $b*2;
14 return array($a,$as);
15 }
16
17 $a = 10;
18 $as = array(1,2);
19 $b = 15;
20 $x = "\$x is defined at top level";
21 example($a, $as, $b, true);
22 print "\$a = $a\n";
23 print_r($as);
24 print "\$b = $b\n";
25 list($resa) = example($a, $as, $b);
26 print "\$resa = $resa\n";
27 ?>
```

- 1 行目の関数では 3 番目の引数が参照渡し、4 番目の引数にデフォルト値が設定されている。
- 6 行目の関数 `isset()` は与えられた変数に値が設定されているかを調べる関数である。
- ここでは `$x` は仮引数ではないのでローカルな変数となり、`isset($x)` は `false` となる。  
`!isset($x)` は `true` となるので変数 `$x` には "defined in function" が代入される。
- 9 行目ではデフォルトの引数のチェックのための部分である。
- 変数に値を代入して、呼び出し元の変数が変わるかのチェックをする (11 行目から 13 行目)。
- 14 行目は最初の二つの仮引数を配列にして戻り値としている。



この関数の動作をチェックするのが残りの部分である。ここでは、関数に渡す引数の値を設定している。20 行目で呼び出した関数内での出力結果は次のようになる。

```
$a = 10
Array
(
 [0] => 1
 [1] => 2
)
$b = 15
$x = defined in function
$GLOBALS['x'] = $x is defined at top level
```

- 仮引数の値は正しく渡されている。
- 6 行目は判定が **true** になるのでここで新たに値が設定され、それが 7 行目で出力される。
- デフォルトの仮引数に対して **true** が渡されているので、9 行目が実行される。スーパーグローバル **\$GLOBAL** にはグローバルスコープ内の変数が格納されている。ここでは 19 行目に現れる変数の値が表示される。

21 行目から 23 行目の出力は次のようになる。

```
$a = 10
Array
(
 [0] => 1
 [1] => 2
)
$b = 30
```

- 初めの 2 つの引数は配列であっても書き直されていない。11 行目と 12 行目の設定は戻り値にしか反映されない。
- 参照渡しの変数 **\$b** は書き直されている。
- 24 行目の関数呼び出しはデフォルトの引数がないので、引数 **\$f** の値が **false** に設定され、9 行目は実行されない。
- **list()** は配列である右辺の値のうち先頭から順に指定された変数に代入するので関数内の 11 行目で計算された値を変数 **\$resa** に代入することになる。

この部分の結果は次のとおりである。

```
$a = 10
Array
(
 [0] => 1
 [1] => 2
)
$b = 30
$x = defined in function
$resa = 20
```

### 9.9.3 可変関数

文字列が代入された変数を用いて、その文字列を変数名にできる可変変数と同様に、文字列が代入された変数の後に `()` をつけると、その文字列の関数を呼び出すことができる。

**実行例 9.9** 次のリストは可変変数の利用例である。

```
1 <?php
2 function add($a, $b) { return $a+$b;}
3 function sub($a, $b) { return $a-$b;}
4
5 $a = 5;
6 $b = 2;
7 $f = "add";
8 print "$f($a,$b) = " . $f($a,$b) . "\n";// add(5,2) = 7
9
10 $f = "sub";
11 print "$f($a,$b) = " . $f($a,$b) . "\n";// sub(5,2) = 3
12 ?>
```

- 2行目と3行目で2つの関数 `add()` と `sub()` を定義している。
- 7行目で変数 `$f` に文字列 `"add"` を代入して、8行目で可変関数として呼び出すと、関数 `add` が呼び出されていることがわかる。
- 同様に、変数 `$f` に文字列 `"sub"` を代入して、可変関数として呼び出すと、関数 `sub` が呼び出されていることがわかる。

## 第10回 サーバーとのデータのやり取り

### 10.1 サーバーとのデータ交換の基本

Web ページにおいてサーバーにデータを送る方法には POST と PUT の 2 通りの方法がある。

**実行例 10.1** 次のリストは実行例 8.1 のリストに `form` のデータを POST で送るようにしたものである。JavaScript の部分だけ異なるのでそこだけのリストになっている。

`windows.onload =function()` 内に次のコードを追加する。

```
var Form = document.getElementsByTagName("form")[0];
Form.setAttribute("method","POST");
Form.setAttribute("action","09sendData.php");
```

HTML の要素に対しては次のことを行う。

- `<select>` 要素の属性に `name="select"` を追加する。
- `id` が `"colorName"` であるテキストボックスに `name="colorName"` を追加する。
- 「設定」 ボタンの要素の後に次の要素を追加する。

```
<input type="submit" value="送信" id="Send"></input>
```

このページでは「送信」 ボタンを押すと `<form>` の `action` 属性で指定されたプログラムが呼び出される。ここでは Web ページと同じ場所にある `09sendData.php` が呼び出される。このファイルのリストは次の通りである。

```
1 <?php
2 print <<<_EOL_
3 <!DOCTYPE html>
4 <head>
5 <meta charset="UTF-8"/>
6 <title>サーバーに送られたデータ</title>
7 </head>
8 <body>
9 <table>
10 _EOL_;
11 foreach($_POST as $key=>$value) {
```

```
12 print "<tr><td>$key</td><td>$value</td></tr>\n";
13 }
14 print <<<_EOL_
15 </table>
16 </body>
17 </html>
18 _EOL_;
19 ?>
```

- 2 行目から 10 行目の間はヒアドキュメント形式で HTML 文書の初めの部分を出力させている。
- `method="POST"` で呼び出されたときには `form` 要素内の `name` 属性が指定されたものの値がスーパーグローバル `$_POST` 内の連想配列としてアクセスができる。
- 11 行目から 13 行目でそれらの値を `table` 要素内の要素として出力している。

出力された結果は次のようになる。

```
<!DOCTYPE html>
<head>
<meta charset="UTF-8"/>
<title>サーバーに送られたデータ</title>
</head>
<body>
<table><tr><td>select</td><td>yellow</td></tr>
<tr><td>color</td><td>green</td></tr>
<tr><td>colorName</td><td>gray</td></tr>
</table>
</body>
</html>
```

なお、`method="PUT"` で呼び出した場合にはスーパーグローバル `$_GET` を用いる。また、スーパーグローバル `$_REQUEST` は `method="POST"` でも `method="PUT"` で呼び出された場合の `$_POST` や `$_GET` の代わりに使用できる。

`type="submit"` の `input` 要素は、ボタンが押されたときに直ちに、`action` 属性で指定された処理が呼び出される。通常は、サーバーにデータを送る前に最低限のエラーチェックを行い、エラーがない場合にだけサーバーと通信するのが良い。

**課題 10.1** PUT と GET を用いた HTML ファイルで実行後の URL がどうなっているか確認しなさい。さらに、実行例 10.1 の PHP のプログラムを直接 URL に記述して呼び出したらどうなるか。また、`Form.setAttribute("method","POST");` の部分を `Form.setAttribute("method","PUT");` に変更して PUT による通信の場合について調べよ。

## 10.2 スパースグローバルの補足

スパースグローバルのうち、これまでに説明してこなかったものについて解説をする。

**\$\_SERVER** この変数はサーバーにアクセスしたときのクライアントの情報などを提供する。具体的な内容はクライアントごとに異なる。

**課題 10.2** `foreach` 構文を用いて **\$\_SERVER** の内容を表示し、どのようなものが送られている確認せよ。

**\$\_COOKIE** COOKIE とは Web サーバー側からクライアント側に一時的にデータを保存させる仕組みである。サーバーと通信するたびに、これらのデータがクライアント側からサーバー側に送られる。これにより、すでに訪問したことがあるサイトに対して情報を開始時に補填する機能などを実現できる。

**\$\_SESSION** セッションとはある作業の一連の流れを指す。たとえば会員制のサイトではログイン後でなければ会員専用のページを見ることができない。情報のページに直接行くことができないような仕組みが必要である。

HTTP 通信はセッションレスな通信である（各ページが独立して存在し、ページ間のデータを直接渡せない）。セッションを確立するために、クライアント側から情報を送り、それに基づいてサーバー側が状況を判断するなどの操作を意識的にする必要がある。

初期のころはページ間のデータを受け渡すために表示しない `<input>` 要素の中に受け渡すデータを入れていた。これはセキュリティ上問題が生ずるので、COOKIE による認証が行われるようになった。

PHP ではセッションを開始するための関数 `session_start()` とセッションを終了させるための関数 `session_destroy()` が用意されている。セッションを通じて保存しておきたい情報はこの連想配列に保存する。セッションの管理はサーバーが管理することとなる。なお、この機能は COOKIE の機能を利用して実現されている。

## 10.3 Web Storage

HTML5 から独立した規格である Web Storage には `localStorage` と `sessionStorage` の2種類のオブジェクトがある。これらのオブジェクトは文字列をキーに、文字列の値を持つ Storage オブジェクトである。同一の出身（プロトコルやポート番号も含む）のすべてのドキュメントが同じ `localStorage` と `sessionStorage` を共有する。`sessionStorage` のデータは意識的に消さない限り存在する。これに対し、`sessionStorage` はウインドウやブラウザが閉じられると消滅する。

これにより、セッション間の情報の移動や以前の訪れたことのあるページの情報の保存を可能にしている。

**実行例 10.2** 次の例は `localStorage` を用いて、ページのアクセス時間を保存し、他のページに移動してもそのデータが利用できることを示すものである。

ページのアクセス時間を記録するページのリストは次のとおりである。

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
5 <title>WebStorage --- localStorage</title>
6 <script type="text/javascript">
7 //
8 var Storage = window.localStorage;
9 //var Storage = window.sessionStorage;
10 window.onload = function() {
11 var AccessList, Message = document.getElementById("message");
12 var D = new Date();
13 if(Storage["access"]) {
14 AccessList = JSON.parse(Storage["access"]);
15 } else {
16 AccessList = [];
17 appendMessage(Message, "初めてのアクセスです");
18 }
19 AccessList.unshift(D.getTime());
20 Storage["access"] = JSON.stringify(AccessList);
21 appendMessage(Message, "今までのアクセス時間です");
22 AccessList.forEach(function(D, i, A) {
23 appendMessage(Message, new Date(D));
24 });
25 }
26 function appendMessage(P, Mess) {
27 var div = document.createElement("div");
28 P.appendChild(div);
29 div.appendChild(document.createTextNode(Mess));
30 }
31 //]]
32 </script>
33 </head>
34 <body>
35 <form action="10next.html">
36 <input type="submit" value="次のページ"></input>
37 </form>
38 <div id="message"/>
39 </body>
40 </html></pre></div>
```

- 8行目で記述を簡単にするためと後で `localStorage` を `sessionStorage` に簡単に修正できるように変数 `Storage` を導入している。
- 11行目で結果を表示するための `<div>` 要素の得ている。
- 12行目ではアクセスされた時間を求めている。
- `WebStorage` に `access` の要素が存在するかを調べ、存在する場合には変数 `accessList` に配列のデータとして復元する (13行目から14行目)。これは、今までのアクセス時間を新しい順に並べた配列である。
- 存在しない場合には、変数 `accessList` を初期化し (16行目)、初回のアクセスである旨の表示を行う (17行目)。テキストを表示する関数は26行目から30行目に定義してある。
- 新しい順に並べてあるアクセス時間の先頭に今回のアクセス時間 (1970年1月1日0時 (GMT) からのミリ秒単位の時間) を挿入し (19行目の `unshift()` は配列の先頭に引数のデータを挿入する配列のメソッドである)、`WebStorage` に JSON 形式で保存する (20行目)。
- 以下のデータがアクセス時間のリストであるメッセージを表示した (21行目) 後、アクセス時間をすべて表示している (22行目から24行目)。
  - － 配列の `forEach` メソッドは、配列の個々の要素に対して引数で与えられた関数を実行する。
  - － このメソッドの戻り値はない。
  - － 引数で渡される関数の引数は3つあり、処理する配列の現在の値 (`D`)、配列の位置 (`i`) と処理する配列 (`A`) である。したがって、`A[i]` と `D` は同じ値である。
  - － 処理結果を配列に戻したければ `A[i]` に値を代入すればよい。
- 26行目から30行目が Web ページ上にデータを表示する関数を定義している部分である。
  - － 引数は、表示するための親要素 (`P`) と表示する文字列 (`Mess`) である。
  - － `<div>` 要素を作成し (27行目)、与えられた親要素の子要素にする (28行目)。
  - － さらに作成した子要素に与えられた文字列を表示するテキスト要素を子要素に付け加える (29行目)

前のページで「次のページ」ボタンを押したとき、移動先のページのリストは次のとおりである。

```

1 <script type="text/javascript">
2 //
3 var Storage = window.localStorage;
4 //var Storage = window.sessionStorage;
5 window.onload = function() {
6 var AccessList, Message = document.getElementById("message");
7 appendMessage(Message, "新しいページです");
</pre>
</div>
```

```
8 if(Storage["access"]) {
9 AccessList = JSON.parse(Storage["access"]);
10 } else {
11 AccessList = [];
12 appendMessage(Message, "初めてのアクセスです");
13 }
14 appendMessage(Message, "今までのアクセス時間です");
15 AccessList.every(function(D, index, A) {
16 appendMessage(Message, new Date(D-0));
17 return index < 3;
18 });
19 }
20 function appendMessage(P, Mess) {
21 var div = document.createElement("div");
22 P.appendChild(div);
23 div.appendChild(document.createTextNode(Mess));
24 }
25 //]]
26 </script>
27 </head>
28 <body>
29 <div id="message"/>
30 </body>
31 </html>
```

このリストは呼び出す前のページのものほとんど変わらないが、アクセス時間の表示が今回のものを含めて 4 つだけ表示されるようになっている (15 行目から 18 行目)。

- 前のリストの 22 行目で `AccessList.forEach` となっている部分が `AccessList.every` となっている。
- この配列のメソッドは引き渡された関数をすべての要素を順に適用し、戻り値の論理積を返す。
- どこかのところで `false` となれば全体の論理積は `false` となるので実行はそこで打ち切られる。
- 論理和を求める `some` という配列のメソッドもある。

**課題 10.3** このリストで次のことを行いなさい。

1. デベロッパーツールから「Application」タブ (Chrome の場合) を開き、Local Storage の内容を確認する。



2. このページのタブを閉じ、いったんブラウザを終了してから再度、このページを開いたときに、以前のデータがそのまま表示されることを確認する。
3. `localStorage` の代わりに `sessionStorage` に変更したとき (リストの 3 行目をコメントにし、4 行目のコメントを外す) に、1 と 2 がどのように変わるか調べる。
4. `AccessList.every` となっている部分を `AccessList.some` でも正しく実行できるようにする。

すでに、いくつかのサイトではこの機能を用いており、その開発者ツールで見ることが可能である。これらのデータの形式は文字列である。構造化されたデータは JSON 形式で保存するのがよいであろう。

**課題 10.4** いくつかのサイトにおいて Storage が利用されているか、どのようなデータが保存されているか調べよ。

## 10.4 Ajax

Ajax とは Asynchronous Javascript+XML の略で、非同期 (Asynchronous) で Web ページとサーバーでデータの交換を行い、クライアント側で得られたデータをもとにその Web ページを書き直す手法である<sup>1</sup>。Google Maps がこの技術を利用したことで一気に認知度が高まった。検索サイトでは検索する用語の一部を入力していると検索用語の候補が出てくる。これも Ajax を使用している (と考えられる)。

Ajax の機能は `XMLHttpRequest` という機能を用いて実現されている。

**実行例 10.3** 次の例は実行例 8.2 で日付が変わったときに、その日の記念日をメニューの下部に示すものである。記念日のデータは <http://ja.wikipedia.org/wiki/日本の記念日一覧> の表示画面からコピーして作成した。

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
5 <title>今日は何の日</title>
6 <script type="text/ecmascript">
7 //<![CDATA[
8 window.onload = function(){
9 function makeSelectNumber(from, to, prefix, suffix, id, parent){
10 var i, option;
11 var Select = makeElm("select", {"id":id}, parent);
12 for(i=from; i<=to; i++) {
13 option = makeElm("option",{ "value":i}, Select);

```

---

<sup>1</sup><http://adaptivepath.org/ideas/ajax-new-approach-web-applications/>

```
14 makeTextNode(prefix+i+suffix,option);
15 }
16 return Select;
17 };
18 function makeElm(name, attribs, parent) {
19 var elm = document.createElement(name);
20 var attrib;
21 for(attrib in attribs) {
22 elm.setAttribute(attrib,attribs[attrib]);
23 };
24 if(parent) parent.appendChild(elm);
25 return elm;
26 }
27 function makeTextNode(text,parent) {
28 parent.appendChild(document.createTextNode(text));
29 };
30 var i, today = new Date();
31 var y = today.getFullYear();
32 var m = today.getMonth();
33 var d;
34 var Form = document.getElementById("menu");
35 var Year = makeSelectNumber(2000,2020,"","年","year", Form);
36 var Month = makeSelectNumber(1,12,"","月","month", Form);
37 var Days = [];
38 for(i=28; i<=31; i++) {
39 Days[i] = makeSelectNumber(1,i,"","日","day");
40 }
41 Year.value = y;
42 Month.value = m+1;
43 d = new Date(y, m+1,0).getDate();
44 Form.appendChild(Days[d]);
45 Form.children[2].value = today.getDate();
46 var changePullldown = function(){
47 var d2 = Form.children[2].value
48 d = new Date(Year.value, Month.value, 0).getDate();
49 if(d != Form.children[2].children.length) {
50 Form.replaceChild(Days[d],Form.children[2]);
51 d2 = Math.min(Form.children[2].length, d2);
52 Form.children[2].value = d2;
53 }
```

```
54 var xmlHttpObj = null;
55 if(window.XMLHttpRequest) {
56 xmlHttpObj = new XMLHttpRequest();
57 } else if(window.ActiveXObject) {
58 xmlHttpObj = new ActiveXObject("Msxml2.XMLHTTP");//IE6
59 } else {
60 try {
61 xmlHttpObj = new ActiveXObject("Microsoft.XMLHTTP");//IE5
62 } catch(e) {
63 }
64 }
65 if(xmlHttpObj) {
66 xmlHttpObj.onreadystatechange = function(){
67 if(xmlHttpObj.readyState == 4 && xmlHttpObj.status == 200) {
68 document.getElementById("details").firstChild.nodeValue =
69 xmlHttpObj.responseText;
70 }
71 }
72 xmlHttpObj.open("GET",
73 "./aniversary.php?month=" + Month.value+ "&day="+d2,true);
74 xmlHttpObj.send(null);
75 }
76 }
77 Form.addEventListener("change", changePulldown,false);
78 changePulldown();
79 }
80 //]]>
81 </script>
82 </head>
83 <body>
84 <form id="menu"></form>
85 <p id="details"> </p>
86 </body>
87 </html>
```

- 以前のものとは46行目以降が異なっている。イベントハンドラーを関数として定義している。
- 47行目から52行目は以前と同じプルダウンメニューの処理である。
- 53行目から63行目は裏でサーバーと通信するためのXMLHttpRequestオブジェクトを作成している。

表 10.1: XMLHttpRequest の通信の状態<sup>3</sup>

値	状態	詳細
0	UNSENT	<code>open()</code> がまだ呼び出されていない。
1	OPENED	<code>send()</code> がまだ呼び出されていない。
2	HEADERS_RECEIVED	<code>send()</code> が呼び出され、ヘッダーとステータスが通った。
3	LOADING	ダウンロード中； <code>responseText</code> は断片的なデータを保持している。
4	DONE	一連の動作が完了した。

- 古いバージョンの IE は別のオブジェクトで通信をするので、ブラウザが `XMLHttpRequest` メソッドを持つか確認し (54 行目)、持っている場合はそのオブジェクトを新規作成する (55 行目)。
  - 56 行目から 62 行目は古い IE のためのコードである。
  - このようにブラウザの機能の違いで処理を変えることをクロスブラウザ対策という。通常はブラウザがその機能を持つかどうかで判断する。
- `XMLHttpRequest` が生成できたら (65 行目)、このオブジェクトが生成する `onreadystatechange` イベントのイベントハンドラーを登録する (66 行目から 71 行目)。
  - `XMLHttpRequest` の `readyState` は通信の状態を表す。4 は通信終了を意味する。これらの値については表 10.1 を参照のこと。
  - 通信が終了しても正しくデータが得られたかを調べる必要がある。200 は正しくデータが得られたことを意味する<sup>2</sup>。
  - 得られたデータは `responseText` で得られる。この場合、得られたデータは文字列となる。このほかに `responXML` で XML データが得られる。
- 72 行目から 73 行目が通信の開始する。ここでは、GET で行うので、URL の後に必要なデータを付ける。
- GET では送るデータ本体がないので、通信の終了をのため `null` を送信する。POST のときはここでデータ本体を送る。
- プルダウンメニューが変化したときのイベントハンドラーを登録し (77 行目)、最後に現在の日付データをサーバーに要求する。、
- 得られたデータは 85 行目の `p` 要素の中に入れる (68 行目から 69 行目)。この要素の `firstChild` を指定しているので 85 行目には `<p>` と `</p>` の間に空白を設けて、テキストノードが存在するようにしている。

**課題 10.5** 表 10.1 の状態は `XMLHttpRequest` オブジェクトのプロパティである。

たとえば、`XMLHttpRequest.DONE` で利用できる。残りのものについてもこの方法で利用できることを確認しなさい。

<sup>2</sup>Http 通信の終了コードについては <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html> を参照のこと

次のリストは Ajax で呼び出される PHP のプログラムである。

```

1 <?php
2 mb_internal_encoding("UTF8");
3 //print mb_internal_encoding();
4 $m = isset($_GET["month"])?$_GET["month"]:$argv[1];
5 $d = isset($_GET["day"])?$_GET["day"]:$argv[2];
6 $data = file("aniversary.txt",FILE_IGNORE_NEW_LINES);
7 for($i=0;$i<count($data);$i++) {
8 $data[$i] = mb_convert_encoding($data[$i],"UTF8");
9 $mm = mb_split("\[",$data[$i]);
10 if(count($mm) >1) {
11 if(mb_convert_encoding($m."月","UTF8") === $mm[0]) break;
12 }
13 }
14 for($i++;$i<count($data);$i++) {
15 $data[$i] = mb_convert_encoding($data[$i],"UTF8");
16 $dd = mb_split("\s-\s",$data[$i]);
17 if(($d."日") === $dd[0]) break;
18 }
19 // print mb_convert_encoding($dd[1],"SJIS");
20 print $dd[1];
21 ?>

```

- 2行目で内部で処理をするエンコーディングをUTF8にしている。関数、`mb_internal_encoding` 関数を引数なしで呼び出すと現在採用されているエンコーディングを得ることができる。
- 4行目と5行目では月 (`$m`) と日 (`$d`) の値をそれぞれの変数に設定している。
  - － ここではコマンドプロンプトからもデバッグできるように、スーパーグローバル`$_GET` 内に値があれば (`isset()`) が `true` になれば、その値を、そうでなければコマンドからの引数を設定している。
  - － スーパーグローバル`$argv` はの先頭は呼び出したファイル名であり、その後に引数が順に入る<sup>4</sup>。
- 6行目の `file` 関数は指定されたファイルを行末文字で区切って配列として返す関数である。この引数には URL も指定できる。
  - － この関数は2番目の引数をとることができる。次の定数を組み合わせて使う。

<code>FILE_USE_INCLUDE_PATH</code>	<code>include_path</code> のファイルを探す
<code>FILE_IGNORE_NEW_LINES</code>	配列の各要素の最後に改行文字を追加しない
<code>FILE_SKIP_EMPTY_LINES</code>	空行を読み飛ばす

<sup>4</sup>C 言語の `main` 関数は通常、`int main(int argc, char* argv[])` と宣言される。`argc` は `argv` の配列の大きさを表し、渡された引数のリストが `argv[]` に入っている。このとき、`argv[0]` は実行したときのファイル名が入る。

- `file_get_contents()` はファイルの内容を一つの文字列として読み込む。Web ページの解析にはこちらの関数を使うとよい。

- 読み込むファイルの一部を次に記す。

1 月 [編集]

1 日 - 鉄腕アトムの日

2 日 - 月ロケットの日

[中略]

31 日 - 生命保険の日、愛妻家の日

2 月 [編集]

1 日 - テレビ放送記念日、ニオイの日

2 日 - 頭痛の日

[以下略]

- 月の部分の後には [がある。
- 日の情報は `␣-␣` で区切られている (`␣` は空白を表す)。
- すべての日の情報が入っている。

- 7 行目から 13 行目までは指定された月の行を見つける。

- 8 行目で念のためコードを `UTF8` に変更している。
- 関数 `mb_split()` 関数は第 1 引数に指定された文字列パターンで第 2 引数で指定された文字列を分割して配列として返す関数である。
- 分割を指定する文字列には正規表現が使えるので、文字 `[` で分割するために、`"\[` としている (9 行目)。
- 指定された文字列があれば配列の大きさが 1 より大きくなる。その行に対して求める月と一致しているか判定し、等しければループを抜ける (11 行目)。

- 14 行目から 18 行目までは指定された月での指定された日の情報を探している。日を決定する方法も月と同じである。文字列の分割は `"\s-\s"` となっている<sup>5</sup>。

- 20 行目で得られた情報をストリームに出力している。

**課題 10.6** 上の HTML のリストの 85 行目の `<p>` と `</p>` の空白を取り除いたら正しく動かなくなることを確認せよ。

なお、構造化されたデータとしては XML 形式でもよいが、より軽量な JSON で与えることも可能である。このときは、`JSON.parse()` で JavaScript のオブジェクトに直せばよい。

---

<sup>5</sup>これは `"\s"` ではうまく行かなかったためである。

## 第11回 jQuery

### 11.1 jQuery とは

jQuery の開発元<sup>1</sup> には次のように書かれている。

jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.

jQuery は JavaScript のライブラリーであり、これまでに解説してきた JavaScript の使い方を簡単にすることを目的としている。2016 年 11 月 19 日現在、jQuery は ver. 3.1.1 が最新のものとなっている。ver. 3.0 は 2016 年 6 月 9 日にリリースされた。それまで提供されていた ver. 1(最終は 1.12.4) と ver. 2(最終は 2.2.4) は重大な訂正 (critical support patch) しか行わないと記されている。

また、配布されているライブラリーはコメントなどがそのまま入っているものと、コメントを取り除き、変数名などを短いものに置き換えて、ファイルサイズを小さくしたものの 2 種類がある。

### 11.2 jQuery の基本

#### 11.2.1 jQuery() 関数と jQuery オブジェクト

jQuery ライブラリーは `jQuery()` というグローバル関数を一つだけ定義する。この関数の短縮形として、`$` というグローバル関数も定義する。`jQuery()` 関数はコンストラクタではない。`jQuery` 関数は引数の与え方により動作が異なるが、戻り値として `jQuery` オブジェクトと呼ばれる DOM の要素を拡張した集まりを返す。このオブジェクトには多数のメソッドが定義されていて、これらの要素群を操作できる。

`jQuery()` 関数の呼び出し方は引数の型により返される要素群が異なる。

- 引数が (拡張された)CSS セレクタ形式の場合(機能 1)

CSS セレクタにマッチした要素群を返す。省略可能な 2 番目の引数として要素や `jQuery` オブジェクトを指定した場合には、その要素の子要素からマッチしたものを返す。この形はすでに解説した DOM のメソッド `querySelectorAll()` と似ている。

---

<sup>1</sup><https://jquery.com> 参照日:2016/11/19

- 引数が要素や Document オブジェクトなどの場合(**機能 2**)  
与えられた要素を jQuery オブジェクトに変換する。
- 引数として HTML テキストを渡す場合(**機能 3**)  
テキストで表される要素を作成し、この要素を表す jQuery オブジェクトを返す。createElement() メソッドに相当する。  
省略可能な 2 番目の引数は属性を定義するものであり、オブジェクトリテラルの形式で与える。
- 引数として関数を渡す場合(**機能 4**)  
引数の関数はドキュメントがロードされ、DOM が操作で可能になった時に実行される。

### 11.2.2 jQuery オブジェクトのメソッド

jQuery オブジェクトに対する多くの処理は HTML の属性や CSS スタイルの値を設定したり、読み出したりすることである。

- これらのメソッドに対してセッターとゲッターを同じメソッドを使う。メソッドに値を与えるとセッターになり、ないとゲッターになる。
- セッターとして使った場合は戻り値が jQuery オブジェクトとなるので、メソッドチェーンが使える。
- ゲッターとして使った場合は要素群の最初の要素だけ問い合わせる。

**HTML 属性の取得と設定** attr() メソッドは HTML 属性用の jQuery のゲッター/セッターである。

- 属性名だけを引数に与えるとゲッターとなる。
- 属性名と値の 2 つを与えるとセッターになる。
- 引数にオブジェクトリテラルを与えると複数の属性を一時に設定できる。
- 属性を取り除く removeAttr() もある。

**CSS 属性の取得と設定** css() メソッドは CSS のスタイルを設定する。

- CSS スタイル名は元来の CSS スタイル名（ハイフン付）でも JavaScript のキャメル形式でも問い合わせ、設定が可能である。
- 戻り値は単位を含めて文字列で返される。

**HTML フォーム要素の値の取得と設定** val() は HTML フォーム要素の value 属性の値の設定や取得ができる。これにより、<select>要素の選択された値を得ることなどができる。



**要素のコンテンツの取得と設定** `text()` と `html()` メソッドはそれぞれ要素のコンテンツを通常のテキストまたは HTML 形式で返す。引数がある場合には、既存のコンテンツを置き換える。

## 11.3 ドキュメントの構造の変更

表 11.1は挿入や置換を行う基本的なメソッドをまとめたものである。これらのメソッドには対になるメソッドがある。

表 11.1: ドキュメントの構造の変更するメソッド

<code>\$(T).method(C)</code>	<code>\$(C).method(T)</code>	機能
<code>append</code>	<code>appendTo</code>	要素 T の最後の子要素として C を付け加える
<code>prepend</code>	<code>prependTo</code>	要素 T の初めの子要素として C を付け加える
<code>before</code>	<code>insertBefore</code>	要素 T の直前の要素として C を付け加える
<code>after</code>	<code>insertAfter</code>	要素 T の直後の要素として C を付け加える
<code>replaceWith</code>	<code>replaceAll</code>	要素 T と C を置き換える

このほかに、要素をコピーする `clone()`、要素の子要素をすべて消す `empty()` と選択された要素 (とその子要素すべて) を削除する `remove()` もある。

## 11.4 イベントハンドラーの取り扱い

jQuery のイベントハンドラーの登録はイベントの種類ごとにメソッドが定義されている。指定された jQuery オブジェクトが複数の場合にはそれぞれに対してイベントハンドラーが登録される。次のようなイベントハンドラー 登録メソッドがある。

<code>blur()</code>	<code>error()</code>	<code>keypress()</code>	<code>mouseup()</code>	<code>mouseover()</code>	<code>select()</code>
<code>change()</code>	<code>focus()</code>	<code>keyup()</code>	<code>mouseenter()</code>	<code>mouseup()</code>	<code>submit()</code>
<code>click()</code>	<code>focusin()</code>	<code>load()</code>	<code>mouseleave()</code>	<code>resize()</code>	<code>unload()</code>
<code>dblclick()</code>	<code>keydown()</code>	<code>mousedown()</code>	<code>mousemove()</code>	<code>scroll()</code>	

このほかに、特殊なメソッドとして `hover()` がある。これは `mouseenter` イベントと `mouseleave` イベントに対するハンドラー を同時に登録できる。また、`toggle()` はクリックイベントに複数のイベントハンドラー を登録し、イベントが発生するごとに順番に呼び出す。

イベントハンドラー の登録解除には `unbind()` メソッドがある。このメソッドの呼び出しはいろいろな方法があるが、`removeEventListener()` と同様な形式として 1 番目の引数にイベントタイプ (文字列で与える)、2 番目の引数に登録した関数を与えるものがある。この場合に、登録したイベントハンドラー には名前が必要となる。

イベントに関してはこのほかにも便利な事項が多くある。

## 11.5 Ajax の処理

jQuery では Ajax の処理に関するいろいろな方法を提供している。ここではもっとも簡単な処理を提供する `jQuery.ajax()` 関数を紹介する。

この関数は引数にオブジェクトリテラルをとる。このオブジェクトリテラルの属性名として代表的なものを表 11.2 に挙げる。

表 11.2: `jQuery.ajax()` 関数で利用できる属性 (一部)

属性名	説明
<code>type</code>	通信の種類。通常は"POST"または"GET"を指定
<code>url</code>	サーバーのアドレス
<code>data</code>	"GET"のときは URL の後に続けるデータ。"POST"のときは <code>null</code>
<code>dataType</code>	戻り値のデータの型を指定する。"text"、"html"、"script"、"json"、"xml"などがある
<code>success</code>	通信が正常終了したときに呼び出されるコールバック関数
<code>error</code>	通信が成功しなかったときに呼び出されるコールバック関数
<code>timeout</code>	タイムアウト時間をミリ秒単位で指定

## 11.6 jQuery のサンプル

**実行例 11.1** 次の例は実行例 10.3 を jQuery を用いて書き直したものである。

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
5 <title>今日は何の日 (jQuery 版)</title>
6
7 <script type="text/ecmascript" src="jquery-3.1.1.min.js"></script>
8 <script type="text/ecmascript">
9 //<![CDATA[
10 $(window).ready(function(){
11 function makeSelectNumber(from, to, prefix, suffix, id, parent){
12 var i, option;
13 var Select = $("<select/>", {"id":id});
14 if(parent) parent.append(Select);//$ (parent).append(Select);
15 for(i=from; i<=to; i++) {
16 option = $("<option/>",{ "value":i, "text":prefix+i+suffix});

```

```
17 Select.append(option);
18 // $("<option/>",{ "value":i,"text":prefix+i+suffix}).appendTo(Select);
19 }
20 return Select;
21 };
22 var i, today = new Date();
23 var y = today.getFullYear();
24 var m = today.getMonth();
25 var Form = $("#menu");
26 var Year = makeSelectNumber(2000,2020,"","年","year", Form);
27 var Month = makeSelectNumber(1,12,"","月","month", Form);
28 var Days = [];
29 for(i=28; i<=31; i++) {
30 Days[i] = makeSelectNumber(1,i,"","日","day");
31 }
32 Year.val(y); //$("#year").val(y);
33 Month.val(m+1); //$("#month").val(m+1);
34 var d = new Date(y, m+1,0).getDate();
35 Form.append(Days[d]);
36 $("#day").val(today.getDate());
37 var changePullldown = function(){
38 var d2 = $("#day").val();
39 d = new Date(Year.val(), Month.val(), 0).getDate();
40 if(d != $("#day option").length) {
41 $("#day").replaceWith(Days[d]);
42 $("#day").val(Math.min($("#day option").length, d2));
43 }
44 jQuery.ajax({
45 type:"GET",
46 url: "./aniversary.php",
47 data: "month=" + Month.val()+ "&day="+d2,
48 dataType: "text",
49 success : function(Data){
50 $("#details").text(Data);
51 },
52 error:function(){alert("error");}
53 });
54 };
55 Form.change(changePullldown);
56 changePullldown();
```

```
57 });
58 //]]>
59 </script>
60 </head>
61 <body>
62 <form id="menu"></form>
63 <p id="details"> </p>
64 </body>
65 </html>
```

- 10 行目の `$(window).ready()` はドキュメントが解釈されたときに引数の関数が呼び出されるイベントである。ここでの要素の参照は機能 2 を用いている。
- 11 行目から 21 行目は連続した番号を値に持つプルダウンメニューを作成する関数である。仕様は以前と同じである。
  - － 13 行目では `<select>` 要素を作成し、同時に属性も設定している (機能 3)。
  - － 15 行目から 19 行目で `<option>` 要素を定義し、`<select>` 要素の子要素にしている。なお、15 行目と 16 行目は `appendTo()` メソッドを用いると 18 行目のコメントアウトしであるように記述することができる。
- 22 行目から 31 行目も依然とほとんど同じである。25 行目では機能 3 を用いて要素を参照している。
- 32 行目、33 行目と 36 行目はプルダウンメニューの初期値を本日に設定している。
- 37 行目から 54 行目はプルダウンメニューが変化したときに呼び出される関数を定義している。
  - － jQuery のメソッドを用いていることをのぞけば 38 行目から 43 行目までは以前と同じである。40 行目のセレクトは `id` が `day(#day)` の下にある `<option>` 要素を選択し、その数を `length` で調べている。
  - － 44 行目から 53 行目は Ajax の処理を定義している。ブラウザによる違いに対する対処や、XMLHttpRequest の処理部分が大幅に減っていて見やすいコードになっていることがわかる。

## 第12回 JavaScript ライブラリーの配布

### 12.1 jQuery のコードを読む

次のリストは jQuery-3.1.1.js の冒頭の部分である<sup>1</sup>。

```

1 /*!
2 * jQuery JavaScript Library v3.1.1
3 * https://jquery.com/
4 *
5 * Includes Sizzle.js
6 * https://sizzlejs.com/
7 *
8 * Copyright jQuery Foundation and other contributors
9 * Released under the MIT license
10 * https://jquery.org/license
11 *
12 * Date: 2016-09-22T22:30Z
13 */
14 (function(global, factory) {
15 "use strict";
16 if (typeof module === "object" && typeof module.exports === "object") {
17 // For CommonJS and CommonJS-like environments where a proper 'window'
18 // is present, execute the factory and get jQuery.
19 // For environments that do not have a 'window' with a 'document'
20 // (such as Node.js), expose a factory as module.exports.
21 // This accentuates the need for the creation of a real 'window'.
22 // e.g. var jQuery = require("jquery")(window);
23 // See ticket #14549 for more info.
24 module.exports = global.document ?
25 factory(global, true) :
26 function(w) {
27 if (!w.document) {
28 throw new Error("jQuery requires a window with a document");
29 }
30 return factory(w);
31 };
32 } else {
33 factory(global);
34 }
35 // Pass this if window is not defined yet
36 })(typeof window !== "undefined" ? window : this, function(window, noGlobal) {
37 // Edge <= 12 - 13+, Firefox <=18 - 45+, IE 10 - 11, Safari 5.1 - 9+, iOS 6 - 9.1
38 // throw exceptions when non-strict code (e.g., ASP.NET 4.5) accesses strict mode
39 // arguments.callee.caller (trac-13335). But as of jQuery 3.0 (2016), strict mode should be common

```

<sup>1</sup>元来のソースでのインデントはタブでつけているがここでは空白 2 文字に変更してある。また、一部の空行も省略した。

```
40 // enough that all such attempts are guarded in a try block.
41 "use strict";
```

このリストでは詳細なコメントが付いていて比較的読みやすい。コードから次のことがわかる。

- このバージョンでは **strict** モードで動作する (15 行目と 41 行目)
- クロスブラウザ対策が行われている (37 行目から 40 行目のコメント)。
- 36 行目で定義されている関数の仮引数に **window** が使われている。

次のリストはこの部分の最小化した部分のリストである。元来は改行が入っていないが、対応をわかりやすくするために改行を入れてある。

```
1 /*! jQuery v3.1.1 | (c) jQuery Foundation | jquery.org/license */
2 !function(a,b){"use strict";"object"==typeof module&&
3 "object"==typeof module.exports?module.exports=
4 a.document?b(a,!0):
5 function(a){if(!a.document)throw
6 new Error("jQuery requires a window with a document");return b(a)}:b(a)}
7 ("undefined"!=typeof window?window:this,
8 function(a,b){"use strict";
```

短縮化では次のことを行っていることがわかる。

- 各関数内で変数名は 1 文字から始めている。
- 関数の仮引数も **a** から付け直している。
- JavaScript の固有の関数は当然のことながら変換されていない。
- このライブラリーは一つの関数を定義して、その場で実行している。14 行目の **function()** の前に **()** が付いている。
- 短縮化されたコードではこの部分が **!function()** となっている。関数オブジェクトを演算の対象とすることでその場で実行する。
- そのほかにもキーワード **true** の代わりに **!0** としている。
- **if(){}else{}** 構文は **?:** に置き換えている。

## 12.2 JavaScript ファイルの短縮化

### 12.2.1 JavaScript ファイルの短縮化について

JavaScript ファイルの短縮化を行う方法はいくつかあるが、ここでは、Google が提供する Closure Compiler を紹介する<sup>2</sup>。

このサービスは次のように説明されている。

<sup>2</sup><https://developers.google.com/closure/compiler/> 2016 年 11 月 23 日参照

The Closure Compiler is a tool for making JavaScript download and run faster. Instead of compiling from a source language to machine code, it compiles from JavaScript to better JavaScript. It parses your JavaScript, analyzes it, removes dead code and rewrites and minimizes what's left. It also checks syntax, variable references, and types, and warns about common JavaScript pitfalls.

これによると、元来の JavaScript のコードをより良い JavaScript のコードに変換し、簡単な警告を表示するようである。

使い方については次のように書かれている。

You can use the Closure Compiler as:

- An open source Java application that you can run from the command line.
- A simple web application.
- A RESTful API.

To get started with the compiler, see "How do I start" below.

ここでは 2 番目にある Web アプリケーションで行う。

このサイトは <http://closure-compiler.appspot.com/home> である (図 12.1)。左側のテキス

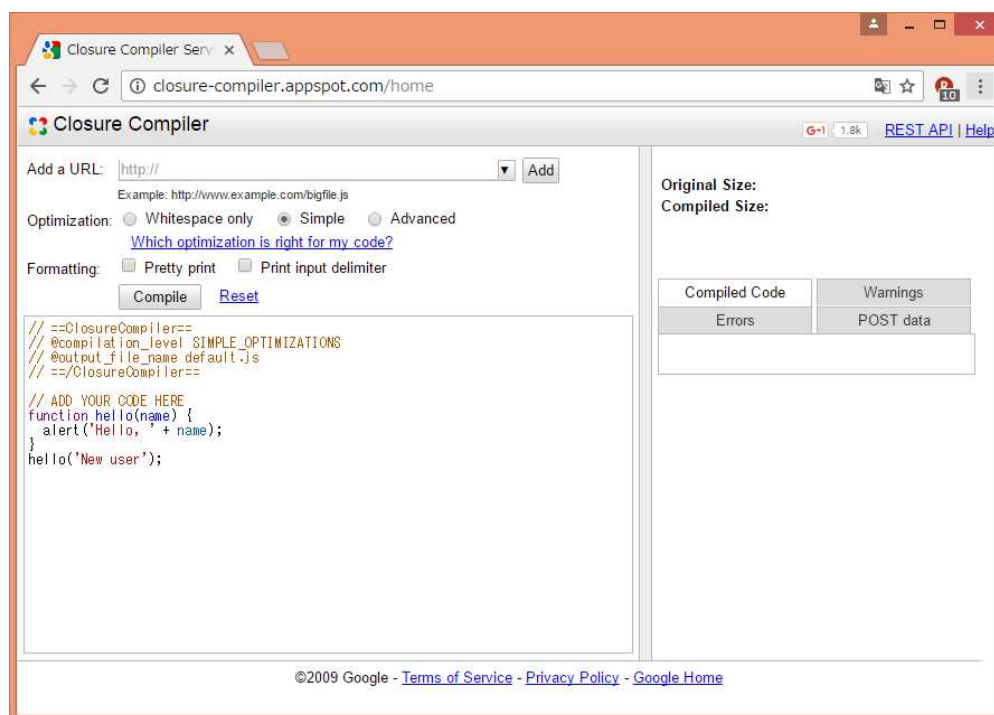


図 12.1: Closure Compiler のホームページ

トボックスにコードを張り付けて、「Compile」のボタンを押せばよい。

### 12.2.2 短縮化の例

ここでは実行例 8.1にある `event.js` で短縮化の効果を見ることにする。

図 12.2はその結果である。

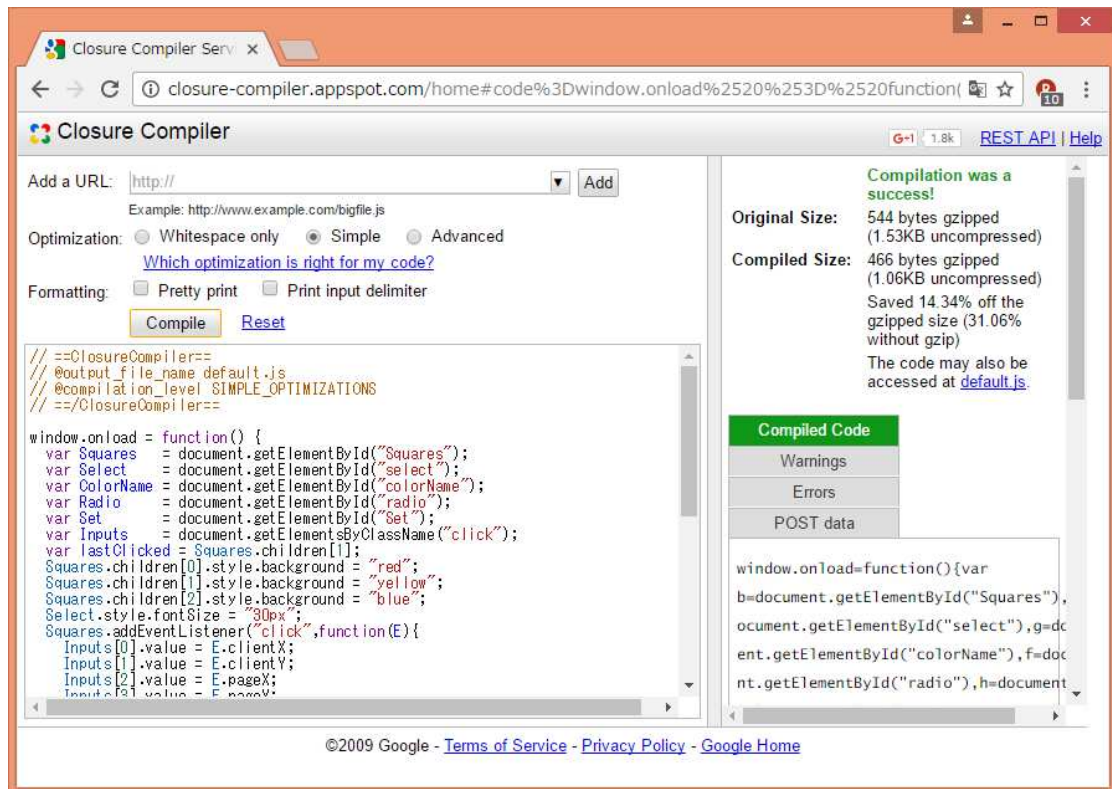


図 12.2: Closure Compiler の結果

画面の右のほうで、元のファイルの大きさが 1.53KB であったのに対し、短縮化の結果が 1.06KB となったことがわかる。

短縮化の効果を見るために、対応するコードのところに改行を入れたものが次のリストである。

```

1 window.onload=function(){var b=document.getElementById("Squares"),
2 e=document.getElementById("select"),
3 g=document.getElementById("colorName"),
4 f=document.getElementById("radio"),
5 h=document.getElementById("Set"),
6 c=document.getElementsByClassName("click"),
7 d=b.children[1];
8 b.children[0].style.background="red";
9 b.children[1].style.background="yellow";

```



```

10 b.children[2].style.background="blue";
11 e.style.fontSize="30px";
12 b.addEventListener("click",function(a){
13 c[0].value=a.clientX;
14 c[1].value=a.clientY;
15 c[2].value=a.pageX;
16 c[3].value=a.pageY;
17 c[4].value=window.pageXOffset;
18 c[5].value=window.pageYOffset;
19 var b=a.target.getBoundingClientRect();
20 c[6].value=a.pageX-b.left;
21 c[7].value=a.pageY-b.top;
22 g.value=a.target.style.background;d=a.target,!1);
23 e.addEventListener("change",function(){d.style.background=e.value,!1);
24 f.addEventListener("click",function(a){
25 alert(a.target.tagName);
26 "DIV"===a.target.tagName&&(a.target.firstChild.checked=!0);
27 console.log("----"+f.value);
28 d.style.background=f.querySelector("input:checked").value,!1);
29 h.addEventListener("click",function(){d.style.background=g.value,!1)});

```

ここで行われている短縮化は次のとおりである。

- 空白の除去
- 変数宣言をまとめる。
- 変数名の単純化
- いくつかの定数を短いものに変える。  
false は!1 に変えている (5 文字から 2 文字)。
- if 文の簡略化

短縮化後の 29 行目は if 文であったものが論理式の&&で置き換えられている。

この一方で document.getElementById などはシステムで定義されているのでそのまま短縮化、共通化できない。この部分も短くするためにはソースコードに工夫が必要である。

しかし、document.getElementById をラップする関数を定義してプログラムコードを短くしてうまくいかない。

```

function getElm(N){
 return document.getElementById(N);
}
var Squares = getElm("Squares");

```

```
var Select = getElm("select");
...

```

図 12.3がコンパイルの結果である。

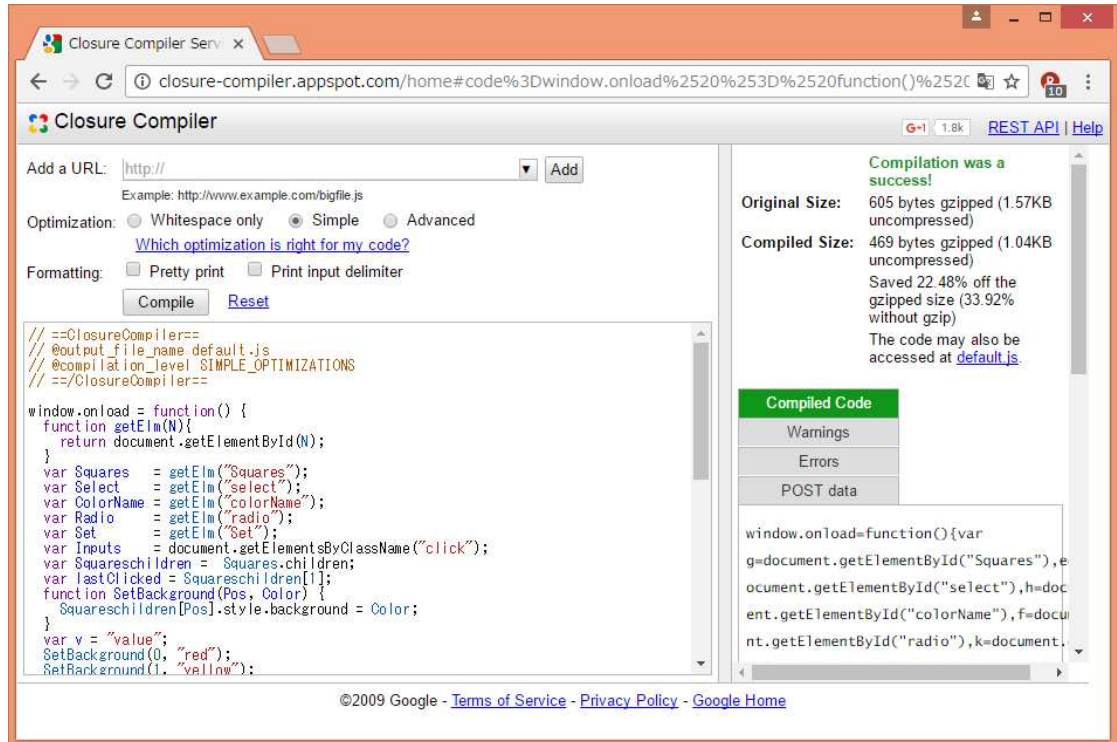


図 12.3: Closure Compiler の結果 (2)

コンパイルの結果を見ると、ラップした関数は消えていて、もとの `document.getElementById` に戻っている。

これを避けるためには関数の引数に `document` と `document.getElementById` を渡すことで解決できる。このとき、引数で渡された `document.getElementById` の実行時の `this` が `document` にならないので、`call` を用いて `this` を `document` にする必要がある。また、関数は 1 度実行する必要がある。

次のリストはそうように書き直したものである。

```
var Squares, Select, ColorName, Radio, Set;
(function(document,getElementById){
 Squares = getElementById.call(document,"Squares");
 Select = getElementById.call(document,"select");
 ColorName = getElementById.call(document,"colorName");
 Radio = getElementById.call(document,"radio");
}
```

```
Set = getElementById.call(document,"Set");
 })(document,document.getElementById);
```

図 12.4がその結果である。コンパイル後の結果が 1KB とわずかに小さくなっていることがわかる。

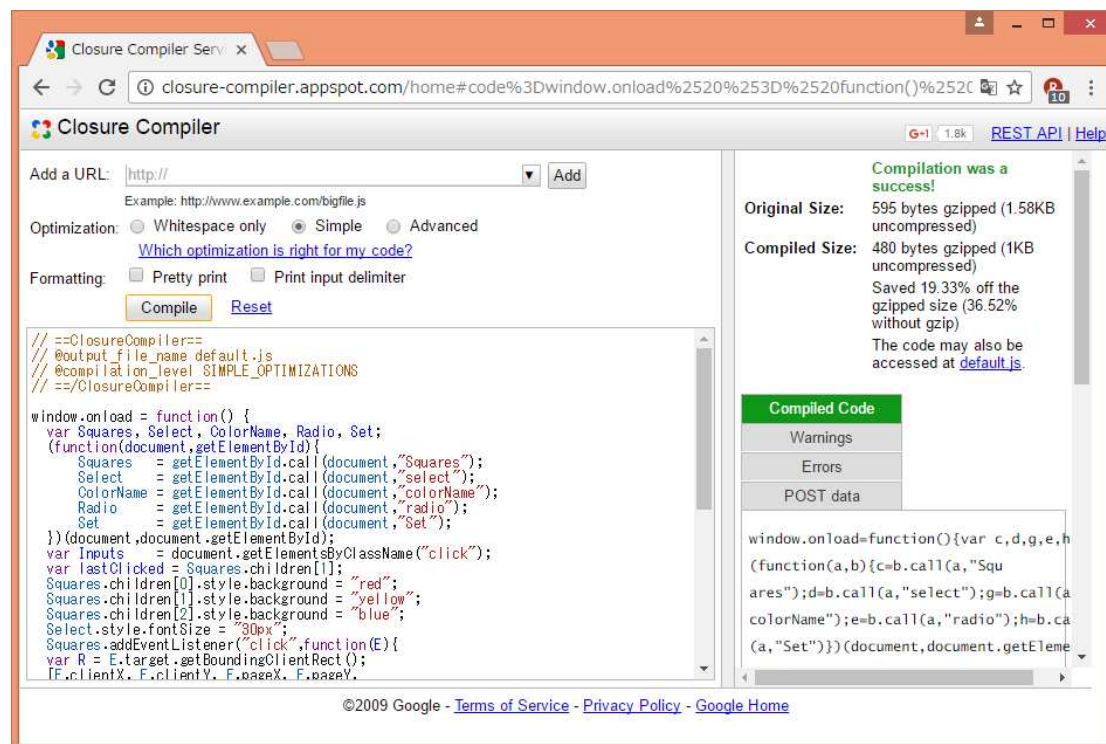


図 12.4: Closure Compiler の結果 (3)

この部分の短縮化のコードは次のようになっている。

```
var c,d,g,e,h;
(function(a,b){c=b.call(a,"Squares");d=b.call(a,"select");
g=b.call(a,"colorName");e=b.call(a,"radio");
h=b.call(a,"Set")})(document,document.getElementById)
```

このようにシステムで定義されたメソッドやプロパティを関数の引数として渡すと短縮化の効率上がる。また、これによりソースコードの理解が難しくなる。

## 12.3 Web サイトの効率化

### 12.3.1 Contents Delivery Network(CDN)

jQuery の短縮化はファイルサイズを減らすことで Web サイトの負荷も減らしている。さらに、このようなファイルはいろいろなサイトで使用されているのでクライアント側でもキャッシュしてお

けばダウンロードの回数を減らせば、回線の負荷が減る。このためには、ライブラリーを置いておくサイトを用意し、そこのファイルを参照すればよい。このような目的のサイトを Contents Deliverly Network(CDN) と呼ぶ。jQuery の場合は <https://code.jquery.com/jquery-3.1.1.min.js> が短縮化されたファイルの CDN の一つである。

### 12.3.2 CSS Sprite

Yahoo Japan のトップページには小さな画像がたくさんある。ブラウザは表示しようとするページに画像があれば、そのページのサイトに画像を要求しに行く。したがって、表示する画像が多いとその回数分だけ、サーバーと通信が行われる。このことはサーバーに負荷がかかる。これを避けるためにブラウザは最近ダウンロードしたファイルを保存して、同じものが要求されたときには、サーバーに要求しないで、保存してあるファイルを使用する (キャッシュの利用)。

この機能を利用してサーバーは、小さな画像が複数含まれる単一の大きな画像を用意し、その一部だけをページに表示するページを用意する。この方法で画像を表示するためには CSS の background 機能を利用する。これを CSS Sprite と呼ぶ。

Yahoo Japan のソースコードで **background-image** をキーワードに検索すれば CSS Sprite で使用されている画像が見つかるであろう。

なお、Yahoo Japan などのアクセスが多いサイトではサーバーとの通信の回数を減らす目的で CSS ファイルや JavaScript のファイルは外部ファイルにしていらないので合わせて確認してほしい。

## 第13回 XML ファイルの処理

### 13.1 XML ファイルとは

#### 13.1.1 W3C における XML の解説

World Wide Web Consortium (W3C) によれば XML(eXtensible Markup Language) は次のように説明されている<sup>1</sup>。

The Extensible Markup Language (XML) is a simple text-based format for representing structured information: documents, data, configuration, books, transactions, invoices, and much more. It was derived from an older standard format called SGML (ISO 8879), in order to be more suitable for Web use.

XML は構造化されたデータを表現する単純なテキストベースのフォーマットである。構造化された文書とは文書、データ、構成、本、送付状やそのほかもろもろのものである。

さらに続けて次のように記されている。

If you are already familiar with HTML, you can see that XML is very similar. However, the syntax rules of XML are strict: XML tools will not process files that contain errors, but instead will give you error messages so that you fix them. This means that almost all XML documents can be processed reliably by computer software.

ここにも書いてあるように XML の形式は HTML の形式に似ているが、XML の文法上に規則は厳格である。

さらに続けて HTML との違いが述べられている。

- すべての要素は閉じられているか、空の要素としてマークされる。
- 空の要素は通常のように閉じられている (`<happiness></happiness>`) か、特別な短い形式 (`<happiness />`) である。
- HTML では属性値はある条件下のもと (空白や特殊文字を含む場合) 以外ではで囲む必要はないが、その規則は覚えておくのは難しい。XML においては属性値は常にで囲む必要がある (`<happiness type="joy">`)。
- HTML では組み込まれている要素名 (とその属性) の集まりがあるが、XML ではそのようなものは存在しない (例外は `xml` で始まるものがある)。

---

<sup>1</sup><https://www.w3.org/standards/xml/core>

- HTML ではいくつかの組み込まれた文字名 (エンティティ) があるが、XML には次の5つのものしかない。

&lt;(<), &gt;(>), &amp;(&), &quot;(&quot;), &apos;(&apos;)

XML では独自にエンティティを定義できる。

### 13.1.2 XML ファイルの例

ここではGPSのログデータを保存する形式の一つとして普及しているGPX形式を取り上げる。  
<http://www.topografix.com/gpx.asp> で規格を見ることができる。

#### GPX ファイルの構造

GPX ファイルの構造は次のようになっている。

ルート要素は `gpx` で、その下の子要素としては次のようなものがある。

- `wpt`(ウェイポイント)  
ある地点の情報
- `rte`(ルート)  
子要素として地点を表す `rtept` を持つ
- `trk`(トラック)  
順序付けられた地点のリスト。子要素として `trkseg` を持つ
- `trkseg` は順序付けられた地点 (`trkpt`) のリストを持つ

これらの要素はすべてある必要はない。

`wpt` と `trkpt` のおもな構成要素は次の通りである。

名称	型	説明
lat	属性	地点の緯度
lon	属性	地点の経度
ele	要素	標高
time	要素	地点での時刻。世界標準時

#### GPX ファイルの例

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <gpx version="1.1"
3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 xmlns="http://www.topografix.com/GPX/1/1"
5 xsi:schemaLocation="http://www.topografix.com/GPX/1/1
6 http://www.topografix.com/GPX/1/1/gpx.xsd"
7 <trk>
8 <name>20160324</name>

```

```

9 <number>1</number>
10 <trkseg>
11 <trkpt lat="35.485802" lon="139.340909">
12 <ele>70.794189</ele>
13 <time>2016-03-24T08:33:23Z</time>
14 </trkpt>
15 <trkpt lat="35.485799" lon="139.340913">
16 <ele>70.794189</ele>
17 <time>2016-03-24T08:33:28Z</time>
18 </trkpt>
19 <trkpt lat="35.485793" lon="139.340899">
20 <ele>70.794189</ele>
21 <time>2016-03-24T08:33:33Z</time>
22 </trkpt>
23 ... 略 ...
24 </trkseg>
25 </trk>
26 </gpx>

```

- 2 行目から 7 行目が `gpx` のルート要素
- 8 行目に `trk`
- 9 行目と 10 行目にはこのルートの名前、通し番号などが存在
- さらに `trkseg` が一つだけ存在
- `trkseg` 内には `trkpt` が存在

初めの位置の情報は次の通り

- 位置は北緯 35.4858026°(`lat`)、東経 139.340909°(`lon`)
- 標高 70.794189m(`ele`)
- 時刻 2016-03-24T08:33:28(2016 年 3 月 24 日 8:33:28)

## 地球の形

地球の形は測量法や測量法施行令で定められている。測量法施行令第三条では地球の長半径と扁平率が定められている。

- 一 長半径 六百三十七万八千百三十七メートル
- 二 扁平率 二百九十八・二五七二二二一〇一分の一

これから地球上の 3 次元の位置を求めるには次のようにする。

元来は緯度と経度に対して扁平率を考慮して空間の位置を求める必要がある。しかし、GPX ファイルのデータでは 2 点間の距離が小さいのと、扁平率が小さく、高さも地球の半径  $R = 6378137\text{m}$

に対して小さいので無視して計算することにする。経度  $\lambda$ 、緯度  $\phi$  の地点の空間位置は次の通りである。

$$x = R \cos \phi \cos \lambda$$

$$y = R \cos \phi \sin \lambda$$

$$z = R \sin \phi$$

## 13.2 Google Maps における Polyline の表示

**Polyline** は指定した地点を折れ線をつなぎ、**Polygon** は指定した地点を折れ線をつないだ多角形を表示する。これらのオブジェクトは `new google.maps.Polyline(<option>)` または `new google.maps.Polygon(<option>)` で作成する。オプションで指定する代表的なものは次の通りである。

プロパティ	説明
<b>map</b>	表示する地図
<b>path</b>	LatLng を要素とする (MVC) 配列
<b>strokeColor</b>	(縁取りの) 線の色
<b>strokeOpacity</b>	(縁取りの) 線の色の不透明度
<b>strokeWeight</b>	(縁取りの) 線の幅
<b>fillColor</b>	塗りつぶしの色 (Polygon のみ)
<b>fillOpacity</b>	塗りつぶしの色の不透明度 (Polygon のみ)

## 13.3 ブラウザでの処理

通常、ブラウザのプログラムではローカルに配置されたフォルダを参照できない。これはセキュリティのためである。この例外がファイルのアップロードである。ユーザの責任において指定したファイルをサーバーにアップできるようになる。

最近の JavaScript では **FileReader** オブジェクトを用いることで、サーバーに転送する代わりにそのファイルをブラウザ内で処理できる。

与えられた GPX ファイル内の道のりを Google Maps 上に表示する例を考える。

次のリストは HTML の部分である。

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Check Trace</title>
5 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
6 <link rel="stylesheet" type="text/css" href="map.css"/>
7 <script src="http://maps.google.com/maps/api/js?"
8 type="text/ecmascript" charset="UTF-8"></script>
9 <script src="map-new2.js" type="text/ecmascript"></script>
10 </head>

```



```

11 <body>
12 <div id="map_canvas" style="width:800px; height:800px"></div>
13 <div id="form">
14 <div>
15 <form id="params">
16 <div><input type="file" id="file" value="ログデータファイルの選択"/></div>
17 </form>
18 </div>
19 <div id="info">
20 </div>
21 </div>
22 </body>
23 </html>

```

- 6 行目はこのページの外部スタイルシートを読み込むことを指定している。
- 7 行目から 8 行目で Google Maps API のファイルを読み込む。ここでは API キーが表示されていないので、コンソール画面に警告が現れる。
- 9 行目は GPX ファイルのデータを処理する JavaScript の読み込みを指定している。
- 12 行目にある Google Maps の表示域である。
- 13 行目から 17 行目にはファイルのアップロードの要素が定義されている。
- 19 行目は表示された GPX ファイルに関する情報を示すものである。

次のリストはこの HTML で読み込まれるスタイルシートである。

```

1 #map_canvas{
2 display: inline-block;
3 }
4 #form{
5 display: inline-block;
6 vertical-align:top;
7 }
8 table tr td {
9 text-align: right;
10 }
11 table > tr:nth-child(1) td:nth-child(2) {
12 text-align:center;
13 }

```

主に、地図とフォームを横並びにし (2 行目と 5 行目の `display:inline-block`)、表の数字を右寄せに (9 行目)、文字のところを中央ぞろえにしている (12 行目)。

次のリストは外部の JavaScript ファイルである。

```

1 var PARAMS = {w:800, h:800};
2 window.onload = function (){
3 var param = location.href.split("?");
4 if(param.length>1) {
5 param = param[1].split("&");
6 param.forEach(function(v) {

```

```

7 vv = v.split("=");
8 if(vv.length>1) PARAMS[vv[0]] = vv[1];
9 });
10 }
```

2 行目から 53 行目で読み込み終了後に実行される関数を定義している。

- 3 行目から 10 行目で URL の後ろにパラメータ (?) があるかをチェックしている。
- あった場合には、パラメータは&で区切られているのでそこで分解し (5 行目)、それぞれに対して (6 行目の `forEach`)=で名前と値に分割して、パラメータ (変数 `PARAM`) に代入している。

```

11 var Clrs = [[255,0,0],[63,63,63],[0,255,0],
12 [0,0,255],[127,127,0],[127,0,127],[0,127,127]];
13 var Colors = Clrs.map(function(C) { return "rgb("+C.join(",")+")";});
14 var C2 = Clrs.map(function(C) {
15 return "rgb("+ C.map(function(C1){
16 return Math.floor((C1+255*2)/3,1)).join(",")+")";});
```

ここでは、GPX のログ内に複数の<trkseg>があった場合に、それら色分けするための色を指定している。道のりの色は後で見るように下が透けるように指定して (88 行目) いるので、地図上と情報用の背景色が同じように見えるように色を調整している。

```

17 var GMap = google.maps;
18 var canvas = document.getElementById("map_canvas");
19 canvas.setAttribute("style","width:"+PARAMS.w+"px; height:"+PARAMS.h+"px;");
20 var map = new GMap.Map(canvas,
21 {center:new GMap.LatLng(35.486210,139.341443),
22 mapTypeId: GMap.MapTypeId.ROADMAP,
23 scaleControl:true,
24 scaleControlOptions:GMap.ControlPosition.BOTTOM_LEFT,
25 zoom:10
26 });
```

ここでは初期の地図の表示を行っている。

```

27 var Points=[];
28 var table = document.getElementById("info");
29 document.getElementById("file").onchange =function(E) {
30 var R = 6378137;
31 var reader = new FileReader();
32 reader.onload = function(){
33 var log = new window.DOMParser().parseFromString(reader.result,"text/xml");
34 var trks = log.getElementsByTagName("trk");
35 Array.prototype.forEach.call(trks,function(trk,i){
36 var trksegs = trk.getElementsByTagName("trkpt");
37 Points[i] = Array.prototype.map.call(trksegs,function(V) {
38 var lat = V.getAttribute("lat");
39 var lon = V.getAttribute("lon");
40 var latRad = lat * Math.PI/180;
41 var lonRad = lon * Math.PI/180;
42 var latCos = Math.cos(latRad);
43 return [lat, lon,
44 new Date(V.getElementsByTagName("time")[0].firstChild.nodeValue).getTime()].concat(
```

```

45 [R*latCos*Math.cos(lonRad),R*latCos*Math.sin(lonRad),R*Math.sin(latRad)]];
46 });
47 console.log(i);
48 });
49 ShowPath(8, 0.5, 0);
50 }
51 reader.onerror= function(){alert("Error")};
52 reader.readAsText(E.target.files[0]);
53 };

```

ここでは、アップロードするファイルが変化したときのイベント処理を行っている。

- 31 行目で `FileReader` オブジェクトを作成している。
- 32 行目から 50 行目で、このオブジェクトが利用可能になった時のイベント処理関数を登録している。実際の起動は 52 行目で発生する。
- 33 行目で読み込まれたテキストを `"text/xml"` で処理をする。処理をするオブジェクトが `window.DOMParser()` の `parseFromString()` メソッドである。
- この処理結果が DOM の構造になるので、今までの DOM の処理が提要できることになる。
- 34 行目で `trk` 要素に分解している。
- それぞれの `trk` 要素に対して処理を行うために、`Array` オブジェクトのメソッド `forEach` を用いている。
- 34 行目で得られたものは `HTML コレクション` なので、このメソッドが直接適用できない。そこで `call` メソッドを利用することで解決している。
- 36 行目で各 `trk` 要素内の `trkpt` 要素を求め、そこから各 `trkpt` 要素の緯度、経度、時間を求めて、空間の位置を計算し (38 行目から 42 行目)、結果を配列で返している (`map` メソッドを利用)。空間の位置を計算しているのは道のりの長さを求めるためであるが、今回のプログラムではそれは利用していない。
- 49 行目でログを表示する関数を呼び出している。
- 51 行目は読み込みが失敗したときの関数を登録している。
- 52 行目では指定されたファイルのデータをテキストとして読み込んでいる。

次の部分は、ログの情報を表示する部分である。

```

54 function makeTR(data, tbl, color){
55 var tr = document.createElement("tr");
56 tbl.appendChild(tr);
57 if(color>=0) tr.style.background = C2[color%C2.length];
58 data.forEach(function(val) {
59 var td = document.createElement("td");
60 tr.appendChild(td);
61 td.appendChild(document.createTextNode(val));
62 });
63 }

```

- この関数の引数は、表に表示するデータ、表示するオブジェクト、(14 行目で定義した) 背景色の指定の 3 つである。
- 55 行目で作成した `tr` 要素の中に、与えられたデータを表示 (58 行目から 62 行目) する。

次のリストは道のりを表示する部分である。

```

64 function ShowPath(w, o, s) {
65 var tbl = document.createElement("table");
66 table.appendChild(tbl);
67 var latMin=180, latMax=-180, lonMin=180, lonMax=-180;
68 makeTR(["番号", "日付", "開始時間", "終了時間", "地点数"], tbl, -1);
69 Points.forEach(function(Ps, i) {
70 var Routes = Ps.map(function(P){
71 latMax = Math.max(latMax, P[0]);
72 latMin = Math.min(latMin, P[0]);
73 lonMax = Math.max(lonMax, P[1]);
74 lonMin = Math.min(lonMin, P[1]);
75 return new GMap.LatLng(P[0],P[1]);
76 });
77 var d1 = new Date(Ps[0][2]);
78 var d2 = new Date(Ps[Ps.length-1][2]);
79 makeTR([i+1,
80 d1.toLocaleDateString(),
81 d1.toLocaleTimeString(),
82 d2.toLocaleTimeString(),
83 Routes.length], tbl, i);
84 new GMap.Polyline({
85 path: Ps.map(function(P){return new GMap.LatLng(P[0],P[1]);}),
86 strokeColor: Colors[(i+s)%Colors.length],
87 strokeWeight:w,
88 strokeOpacity:o,
89 map: map});
90 }
91);
92 var latLngB = new GMap.LatLngBounds(new GMap.LatLng(latMax, lonMax));
93 latLngB.extend(new GMap.LatLng(latMin, lonMin));
94 map.fitBounds(latLngB);
95 }
96 }
```

- この関数の引数は道のりの幅、不透明度である。
- 69 行目から 76 行目で道のりを構成する地点を Google Maps の地点オブジェクトに変え、かつ、道のりの緯度、経度の最大値と最小値を求めている。
- 79 行目から 83 行目で道のりの情報を表示する。
- 84 行目から 90 行目で与えられたデータをもとに、道のりを表示している。道のりのデータはオブジェクトリテラルの形で与えられている。
- 92 行目から 94 行目では表示された道のりをすべて含む最大のズームレベルを設定 (`fitBounds` メソッド) している。

## 13.4 PHP による処理

XML ファイルの処理は PHP でも可能である。次の例は GPX アイルから道のりに必要な JSON オブジェクトを作成する例である。

```

1 <?php
2 function getPos($lat, $lon) {
3 $R = 6378137;
4 $latRad = $lat * M_PI/180;
5 $lonRad = $lon * M_PI/180;
6 $latCos = cos($latRad);
7 return array(
8 $R*$latCos*cos($lonRad), $R*$latCos*sin($lonRad), $R*sin($latRad));
9 }
```

与えられた緯度経度から空間の位置を求める関数である。戻り値は空間の位置を配列として返す。

次のリストは与えられたファイルから GPX ファイルを処理する関数である。ファイル名は初めの引数で与えられる。

```

10 function SetDatafromFile($fn,$mode) {
11 $data = new DOMDocument();
12 $data->load($fn);
13 $trks = $data->getElementsByTagName("trk");
14 $len = $trks->length;
15 $trackdata = array();
16 $lengthdata = array();
17 $cnt=0;
```

- XML ファイルを読み込むために `DOMDocument` のインスタンスを作成 (11 行目) している。
- `load` メソッドにファイル名を指定することで、DOM オブジェクトに変換される (12 行目)。
- PHP ではピリオド `.` は文字列の接続演算子なのでメソッドは `->` を用いる。
- `getElementsByTagName` で `trk` を取得 (13 行目) する。
- 14 行目でその数を変数 `$len` に格納している。
- トラックごとの情報をしまうための変数を初期化 (15 行目と 16 行目) している。

```

18 for($i=0;$i<$len;$i++){
19 $trk = $trks->item($i);
20 $trksegs = $trk->getElementsByTagName("trkpt");
21 $len2 = $trksegs->length;
22 if($len2 <10) continue;
23 $trkseg = $trksegs->item(0);
24 $latmin = $latMax = $lat = $trkseg->getAttribute("lat");
25 $lonmin = $lonMax = $lon = $trkseg->getAttribute("lon");
26 $newtrk = array("[$lat,$lon]");
27 $ppos = getPos($lat, $lon);
28 $length = 0;
```

各 `trk` に対して距離などを求める。

- `trk` の一つを変数`$trk`に格納している。呼び出しに注意 (19 行目) すること。
- `$trk`に含まれる `trkpt` のリストを得る (20 行目)。
- 極端に短いログは省く (22 行目)。
- ログ内の地点の範囲を示す変数を初期化 (24、25 行目) している。
- 地点の情報をしまう配列を初期化 (26 行目) している。
- 初めの位置の空間座標を変数`$ppos`に格納 (27 行目) している。
- 積算距離の変数を初期化 (28 行目) する。

```

29 for($j = 1; $j < $len2; $j++) {
30 $trkseg = $trksegs->item($j);
31 $lat = $trkseg->getAttribute("lat");
32 $lon = $trkseg->getAttribute("lon");
33 $latMax = max($lat-0,$latMax);
34 $latmin = min($lat-0,$latmin);
35 $lonMax = max($lon-0,$lonMax);
36 $lonmin = min($lon-0,$lonmin);
37 array_push($newtrk,"[$lat,$lon]");
38 $cpos = getPos($lat, $lon);
39 $xd = $ppos[0]-$cpos[0];
40 $yd = $ppos[1]-$cpos[1];
41 $zd = $ppos[2]-$cpos[2];
42 $length += sqrt($xd*$xd+$yd*$yd+$zd*$zd);
43 $ppos = $cpos;
44 }

```

- 各 `trkpt` から緯度と経度を取り出し (31、32 行目)、今までの範囲外ならば範囲を更新 (33 行目から 36 行目) している。
- 緯度と経度を配列に追加 (37 行目) し、空間の位置を計算 (38 行目) する。
- ひとつ前の点との差を求め (39 行目から 41 行目)、距離を計算 (42 行目) する。
- ひとつ前の点を更新 (43 行目) する。

```

45 if($lonMax - $lonmin > 0.001 || $latMax - $latmin > 0.001) {
46 $cnt++;
47 array_push($trackdata,
48 '{"range":["$latMax,$latmin,$lonMax,$lonmin"],"route":["' .
49 implode("",$newtrk).'"],"length":' . ($length/1000).'}');
50 }
51 }
52 return $trackdata;
53 }
54 ?>

```

- 移動がほとんどないトラックは登録しない (45 行目)。
- 登録するトラックを JSON 形式で作成 (48 行目から 49 行目) する。
- 関数 `implode` は配列の要素を与えられた文字列を挟んで出力する。

## 第14回 システム開発のためのヒント

### 14.1 コードの構造

#### 14.1.1 良いコードとは

良いコードとはどのようなものであろうか。それは個人によっても違うであろう。しかしながら、多くの公開されているコードを眺めているといくつかの共通点が見つかる。

- プログラムの構造がわかりやすくなるように字下げ (インデント) を付ける。
- 変数名や関数名は実態を表すようにする。  
いくつかの単語をつなげて変数名や関数名にすることが多くなる。これらが読みやすいように途中に出てくる単語の先頭は大文字にすることが最近の傾向である (キャメル形式と呼ばれる)。
- 不必要に長いプログラム単位を作らない。  
ある程度まとまった作業をする部分は関数にすると見通しの良いプログラムとなる。
- プログラム自体が説明書になっている。  
コメントを多用して必要な説明をすべて記述する。
- プログラムとデータの分離  
ある処理をするときの分岐の条件をプログラム内に直接記述すると、分岐の条件が変わったときにはプログラムをコンパイルしなおす必要が生ずる。これを避けるためには外部からデータを読み込んで、それに基づいた処理を行うようにすることができると考えてみる。

#### 14.1.2 プログラムとデータの分離

**実行例 14.1** 期末試験の結果、90 点以上ならば S、80 点以上ならば A、70 点以上ならば B、60 点以上ならば C、それ以外は E の成績をつけることにした。期末試験の点を与えて、評価の S などを返す関数 `seiseki` を作成しなさい。

一番思いつくコードは次のようなものであろう (JavaScript で記述)。

```
function seiseki(val) {
 if(val >= 90) return "S";
 if(val >= 80) return "A";
 if(val >= 70) return "B";
```

```
 if(val >= 60) return "C";
 return "E";
}
```

これに対し、数値と戻り値をそれぞれ配列の値として用意するとつぎのようなコードとなるであろう。

```
function seiseki(val) {
 var Score = [90,80,70,60,0];
 var Results= ["S","A","B","C","E"];
 var i;
 for(i=0;i<Grade.length;$i++) {
 if(val>=Grade[i]) return Hyouka[i];
 }
 return "Error";
}
```

二つの配列をまとめて一つの配列にするほうがコードの管理上わかりやすくなるであろう。

```
var Results =[["S",90],["A",80],["B",70],["C",60],["E",0]];
```

または、次のような連想配列にしてもよい。

```
var Results ={S:90,A:80,B:70,C:60,E:0};
function seiseki(val) {
 for(v in Results) {
 if(val >=Results[v]) return v;
 }
}
```

JavaScript では `for(.. in ..)` で連想配列の要素をすべて渡ることができ、かつわたる順序が定義された順なのでこのコードが可能となる。

このように配列をうまく利用するとロジックの部分とパラメータの部分の分離が可能となる。

なお、JavaScript では JavaScript 内から外部ファイルを自動で読み込むことができないので上のような変数 **Results** を宣言する必要がある。この変数部分を別のプログラムから作成すればもう少し管理が楽になるであろう。

## 14.2 コードの管理

### 14.2.1 複数でシステム開発をするときの問題点

システムを開発していると次のような問題が発生する。

- 複数の人間で開発している場合、開発者の間で最新のコードの共有



- ファイルを間違えて削除したり、修正がうまくいかなかったときに過去のコードに戻す
- 安定しているコードに新規の機能を付け加えてテストをする場合、安定したコードに影響を与えないようにする

このようなコードの変更履歴の管理するソフトウェアをバージョン管理ソフトという。バージョン管理ソフトウェアの対象となるファイルはテキストベースのものを主としている。

### 14.2.2 バージョン管理の概念

バージョン管理ソフトは各時点におけるファイルの状態を管理するデータベースである。このデータベースは一般にリポジトリと呼ばれる。

開発者は通常次の手順でシステムの開発を行う。

- リポジトリからファイルの最新版を入手
- ローカルな環境でファイルを変更。
- ファイルをリポジトリに登録

バージョン管理ソフトでは同じファイルを複数の開発者が変更した場合、競合が発生していないかをチェックする機能が備わっている。この機能がない場合には同じファイルの変更は同時に一人しか変更ができないようにファイルをロックする。バージョン管理システムでは一連の開発の流れがある。この開発の流れをブランチと呼ぶ。あるブランチに対して新規の機能を付け加えたいときなどには元のブランチには手を付けずに別のブランチを作成して、そこで開発をする。機能が安定すれば元のブランチに統合 (merge) する。

### 14.2.3 バージョン管理ソフトの概要

CVS(Concurrent Versions System) やこの改良版である Subversion はバージョン管理するためにサーバーが必要となる。開発者はこのサーバーにアクセスしてファイルの更新などを行う。

これに対し、git は各開発者のローカルな環境にサーバー上のリポジトリの複製を持つ。これにより、ネットワーク環境がない状態でもバージョン管理が行える。git ではネットワーク上に GitHub と呼ばれるサーバーを用意しており、ここにリポジトリを作成することで開発者間のデータの共有が可能となっている。データを公開すれば無料で利用できるほか、有料のサービスやサーバー自体を個別に持つサービスも提供している。

現在では有名なオープンソースのプロジェクトが ‘GitHub’ を利用して開発を行っている。

### 14.2.4 git の使い方

git の詳しい使い方については次のサイトを参照すること。

<https://git-scm.com/book/ja/v2/>

ここでは簡単な使い方を解説する。

## git の特徴

git のバージョン管理システムとして次のような点が挙げられている。

- 分散型のバージョンシステムなので、ローカルでブランチの作成が可能  
集中型のバージョン管理システムではブランチの作成が一部の人しかできない。
- 今までのファイルの変更履歴などが見える。
- GitHub 上ではファイルの更新などの通知を受けることができる。
- 開発者に対して変更などの要求が可能 (pull request)

## git のインストール

git はローカルにリポジトリを持つのでそれを処理する環境をインストールする必要がある。ここでは git for windows を紹介する。このソフトは次のところからダウンロードできる。

<https://git-for-windows.github.io/>

デフォルトの設定で十分である。

「Git Bash」、「GitCMD」と「Git GUI」の 3 つがインストールされる。Unix 風のコマンドプロンプトが起動する Git Bash がおすすめである。

## 初期設定

ローカルでいくつかの準備が必要となる。

**ユーザーネームの登録** GitHub で使用するリポジトリのユーザ名を登録は次のコマンドで行う。

```
git --config --global user.name "Foo"
```

ここでは Foo がユーザ名となる。

**メールアドレスの登録** GitHub からの通知を受けるメールアドレスは次のコマンドで行う。

```
git --config --global user.email "Foo@example.com"
```

**SSH Key の登録** GitHub との接続には SSH による通信で行う。この通信は公開鍵暗号方式で行うので、キーを生成する必要がある。キーの生成は次のコマンドで行う。

```
ssh-keygen -t rsa -C "Foo@example.com"
```

-C のオプションはコメントである。

この後でキーを保存するフォルダが聞かれるが、デフォルトでかまわない。その後、passphrase の入力が求められるので、何かの文字列を入力する。マニュアルによれば passphrase はパスワードのようなもので、10 から 20 文字の長さで大文字、小文字、数字と記号が混在することが推奨されている。再度、同じものの入力が求められ、一致すれば通信のためのキーが生成される。

指定したフォルダの .ssh の下に id\_rsa(秘密鍵) と id\_rsa.pub(公開鍵) の 2 つのファイルが作成される。

### GitHub のアカウントの取得

ローカルだけで開発をするのであればアカウントは必要ないが、データを交換するのであればアカウントを取ることを勧める。有料で使用するものでなければこのためのアカウントは新たにとることを勧める。

作成した公開鍵の内容をアカウントで設定する必要がある。

アカウントを取ったら GitHub 上でテストのプロジェクトを中身が空で作成する。

### リポジトリの作成と運用

リポジトリの初期のブランチ名は **master** となっている。

コマンド	パラメータ	説明
<b>git init</b>		プロジェクトの初期化
<b>git add</b>	ファイルリスト	プロジェクトへのファイルの登録
<b>git commit</b>	<b>-m</b> コメント	リポジトリへの変更の登録
<b>git remote add</b>	短縮名 リポジトリ	リポジトリの短縮名の登録
<b>git push</b>	<b>-u</b> 短縮名 ブランチ名	ブランチへの変更の登録
<b>git clone</b>	リポジトリ名	リモートのリポジトリのコピーを作成
<b>git pull</b>	リポジトリ名 ブランチ名	リモートのリポジトリのコピーを作成

通常は次の手順で新しいプロジェクトを構成する。

1. **git init** で新たにリポジトリを作成する。または、**git clone** で GitHub からリポジトリのコピーを取得する。
2. ファイルを編集する。
3. 編集したファイルを **git add** でステージする。
4. **git commit** でその時点での変更を登録する。**-m** オプションで簡単なコメントをつけることを忘れないこと。このオプションをつけないと **vim** というテキストエディタが立ち上がり、コメントの編集を行うこととなる。
5. **git push** リモート名 ブランチ名 で変更をリモートのリポジトリに反映される。これが拒否された場合には誰かが **push** しているのでその変更を **pull** してから調整して再度 **push** する。

ファイルの状態については次のことに注意する必要がある。

- リポジトリ内のファイルは追跡されている (tracked) ものと追跡されていない (untracked) に分けられる。
- ファイルの状態として変更されていない (unmodified)、変更されている (modified) とステージされている (staged) の 3 つの状態がある。
- modified から staged にするコマンドが **git add** である。このコマンドを新規にリポジトリに追加するという意味にとらえないことが必要である。