

今日では JavaScript はサーバー側のアプリケーションやデスクトップアプリケーションの作成にも使用されている。それでも JavaScript は HTML に埋め込んでブラウザ上で実行され、インタラクティブな Web ページを記述する用途が主流である。以下では HTML 中の JavaScript の使用に絞って説明する。

HTML 文書の全体はだいたい次のような構造を持っている。

```
1 <!Doctype HTML>
2 <html>
3 <head>
4 <meta http-equiv="Content-Type" content="text/html;
  charset=utf-8">
5 <title>...</title>
6   :
7 </head>
8 <body>
9   :
10 </body>
11 </html>
```

HTML に埋め込むとき JavaScript は `<script type="text/javascript">` ~ `</script>` というタグの間に書く。

```
<script type="text/javascript">
  // ここに JavaScript のプログラムを書く。
</script>
```

あるいは、`src` という属性を使って、

```
<script type="text/javascript" src="nantoka.js">
</script>
```

で、別ファイル (`nantoka.js`) の JavaScript ソースを読み込める。なお `type="text/javascript"` は省略可能である。

この例でわかるように JavaScript ソースファイルの拡張子は通常 `.js` である。

この `script` タグは HTML 文書の `<head>` ~ `</head>` の間、`<body>` ~ `</body>` の間のどちらにも書くことができる。

2015 年に制定された ECMAScript 6 からモジュールという仕組みが導入された。モジュールを記述しているソースを読み込む場合は

```
<script type="module" src="kantoka.js"></script>
```

のように `type="module"` とする。また、モジュールのソースファイルには `.mjs` という拡張子を使うこともある。

1.3 JavaScript の開発環境

JavaScript のプログラムは通常のテキストエディターで編集できるが、統合開発環境 (IDE) を使うと、シンタックスハイライトなどの支援があり、デバッガーも統合されているため、プログラムを迅速に開発することができる。

統合開発環境としては、Eclipse Foundation の Eclipse, JetBrains 社の IntelliJ IDEA, Microsoft 社の Visual Studio, Visual Studio Code などがある。Eclipse や IntelliJ IDEA は Java の開発環境としても有名である。なお Visual Studio Code はそれ自体も JavaScript (TypeScript を含む) を使って記述されている。

またブラウザ上で実行できるオンライン開発環境もいくつか見つけることができる。JSFiddle (<https://jsfiddle.net/>), CodePen (<https://codepen.io/>), JS Bin (<https://jsbin.com/>), Plunker (<https://plnkr.co/>) などは JavaScript に特化したオンライン IDE である。一方 Ideone (<https://www.ideone.com/>), Rextester (<https://rextester.com/>), JDoodle (<https://www.jdoodle.com/>), Paiza.IO (<https://paiza.io/ja>) などは JavaScript 以外にも多くのプログラミング言語を選択して、実行することができる。

1.4 JavaScript の基本文法

JavaScript の基本的な文法は C 言語あるいは Java 言語に似ているため、これらの言語を既に知っている人は、ほとんどの部分は見慣れているだろう。ただし、型に関する宣言は必要ないので、異なるキーワードを用いる部分がある。

定数

JavaScript の定数は、39 や 3.14159 などの数値、true, false などの真偽値などがある。文字列定数は、0 個以上の文字を二重引用符「"」または一重引用符「'」で囲む。(例えば、「"Hello, world!"」 「'Takamatsu, Kagawa'」などが文字列の定数である。)

関数呼出し

関数は、前もって定義されているプログラムの部品である。関数名のあとに丸括弧の対「(~)」を書き、その中に関数に渡すデータを表す式(引数・ひきすう)をコンマ「,」で区切って書くと関数呼出しの式になり、前もって用意されたコードを利用することができる。

例えば、[alert](#) は警告ダイアログを開いてメッセージを表示する関数である。次のように書くと、「Hello, world!」と表示するダイアログを開く。

ファイル alert1.js

```
1 alert("Hello, world!");
```

式と文

式のあとにセミコロン「;」をつけると文になる。文を単純に並べると、上(左)から順次実行される。

ファイル alert2.js

```
1 alert("Hello");
2 alert("world!");
```

このプログラムはまず、“Hello”というダイアログを開いてから、“World!”というダイアログを開く。

console.log メソッド

コンソール画面（後述）にメッセージを出力するためには _____ というメソッドを使う。引数の数は決まっておらず、与えられた引数を順に出力する。

ファイル console.js

```
1 console.log("hello", 10, 24);
```

オブジェクトとドット演算子

JavaScript のオブジェクトは、いくつかの構成要素からなる複合的なデータで、名前に関連付けられたデータ（プロパティ）や関数（メソッド）の集まりである。ドット「.」は、オブジェクトからその構成要素（プロパティやメソッド）を取り出すための演算子である。上の console.log も正確に表現すると、console という名前のオブジェクトの log という名前に関連付けられている関数（メソッド）である。

メソッドは、いわばオブジェクトに“属している”関数である。一般的にはオブジェクトの種類が異なれば、同じ名前のメソッドでも異なる動作をする。つまり xxx.do_something(1, "abc") と yyy.do_something(1, "abc") のように、同じ名前のメソッドであっても、オブジェクト xxx と yyy の種類が異なれば、異なる動作をする。

コメント

コメントは、プログラム中に挿入する人間に対するメッセージ・覚え書きで、プログラムの実行には影響を及ぼさない。

JavaScript のコメントは「`_____`」から「`_____`」までの間と、「`_____`」からその行の終わりまでである。前者のコメントは複数行にまたがることができる。

問 1.4.1 「`"`」が含まれる文字列、「`'`」が含まれる文字列、「`'`」と「`"`」の両方が含まれる文字列を JavaScript で表す方法を調べて実装せよ。

1.5 JavaScript プログラムのデバッグ

上述の console.log 関数は、コンソールに出力する関数である。「コンソール」の表示の仕方はブラウザによって異なるが、Chrome の場合は、「`:`」 - 「その他のツール」 - 「デベロッパー ツール」、Firefox の場合は、「`≡`」 - 「その他のツール」 - 「Web 開発ツール」、Edge の場合は、「`...`」 - 「その他のツール」 - 「開発者ツール」、で表示されるようである。

一方 alert 関数は、警告ダイアログを開いて、それをユーザーが閉じるまで、次の処理に進まない。

JavaScript はプログラムの実行前にエラーを見つけてくれないので、気軽に試せる反面、デバッグがしにくい、という短所がある。問題があったときでも、実行が止まってしまって画面に何も表示されず、そのままでは手がかりが何も得られないことが多い。そういう場合はブラウザの「コンソール」を表示しておけば、`console.log` 関数の出力のほかに、実行時に表示されるエラーメッセージを見ることができる。

また `alert` や `console.log` 関数呼出しをプログラム中に適宜挿入しておいて実行すると、どこまで実行されたか、どこで実行が止まっているか、などを確認することができる。

1.6 変数と代入

変数

変数は値をいれることができる入れ物に名前をつけたものである。JavaScript には型チェックはないので、変数の宣言に型の情報は必要なく、`var` というキーワードで変数を宣言する。このとき次のように「`=`」の右側に初期値を指定することができる。

```
var i = 0;
```

変数の宣言の最後は、セミコロン「`;`」をつける。ただし、JavaScript には自動的にセミコロンを挿入する、というありがたい（お節介な）仕様があるので、省略される（忘れる）場合もある。

次のように、複数の変数をコンマ「`,`」で区切って宣言することも可能である。

```
var i = 0, j = 1, k;
```

2015 年の ECMAScript 6 (ES6) からはさらに `let`, `const` という 2 つのキーワードが追加された。このうち、`let` はブロックスコープの局所変数を宣言する。つまり、`let` というキーワードで宣言された変数は、その周りのブレース（`{` ~ `}`）の間でのみ有効である。

ファイル `let.js`

```
1 var x = 99;
2 {
3   let x = 1;
4   console.log("x = ", x);
5 }
6 console.log("x = ", x);
```

このプログラムは、

```
x = 1
x = 99
```

とコンソールに出力する。

一方、`const` はブロックスコープの代入不可の変数を宣言する。つまり、代入して値を変えようとする、実行時にエラーになる。

ファイル `const.js`

```
1 const x = 2;
2 console.log("x = ", x);
3 x = 99;
4 console.log("x = ", x);
```

このプログラムは、`x` の値を変更しようとする、エラーになるので、「`x = 2`」とコンソールに出力した時点で止まってしまい、「`x = 99`」は表示されない。

変数名の規則

JavaScript の変数名の中に使える文字は、大文字と小文字のアルファベット (`a~z, A~Z`)、数字 (`0~9`)、アンダースコア「`_`」、ドル記号「`$`」である。このうち、数字は先頭の文字としては使えない。ドル記号「`$`」は、プログラムが自動的に生成する変数名などに使うことを意図されているようで、あまり人が書くプログラムには使われない。

他のユニコードの文字、例えば、ギリシャ文字（「`Σ`」, 「`π`」など）、かなや漢字なども変数名に使うことができるが、あまり使われていない。残念ながら絵文字は変数名には使えないようである。

アルファベットの大文字と小文字は 異なる ので `abc` と `ABC` は異なる変数である。

算術演算子

四則演算の「`+`」、「`-`」、「`*`」、「`/`」、「`%`」などの演算子は C 言語や Java とほぼ同じである。それぞれ、足し算、引き算、掛け算、割り算、割り算の余り、を計算する。ただし C や Java と異なり、「`/`」は整数同士の割り算でも結果は小数になりうる。

ファイル `arithmetic.js`

```
1 console.log(1 / 2); // 0.5 になる。
```

また、「`+`」演算子は文字列の接続にも使用できる。

ファイル `arithmetic.js`

```
2 console.log("2 * 3 is " + (2 * 3) + ".");
```

これは「`2 * 3 is 6.`」と出力する。上の例で示すように、普通の数学の式と同様、計算の順番を変えるために丸括弧「`()`」を用いる。（波括弧「`{ }`」や角括弧「`[]`」はこの目的には使わない。）しかし、掛け算「`*`」は足し算「`+`」より優先なので上の `2 * 3` の周りの丸括弧は省略可能である。

また「**」は累乗の計算をする演算子である（ECMAScript 7 (2016) より取り入れられた。）。この「**」は右結合の演算子である。つまり、右から先に計算する。

ファイル arithmetic.js

```
3 console.log("2 ** 2 ** 3 is " + (2 ** 2 ** 3) + ".");
```

これは $2^{2^3} = 256$ なので、「2 ** 2 ** 3 is 256.」と出力する。（ $(2^2)^3 = 64$ ではない。）また、 $2 ** 2 ** 3$ の周りの丸括弧も省略可能である。

代入

変数やオブジェクトのプロパティの値を変更することを代入という。JavaScript では代入には「=」演算子を用いる。「変数 = 式」（あるいは `obj.prop = 式`）という式で「=」の左辺の変数やプロパティの値が右辺の式を計算した結果になる。

一方、変数の値は、変数の名前を書くだけで取り出すことができる。

ファイル varSample1.js

```
1 var x = 2, y = 3;
2 x = x + y;
3 y = x - y;
4 x = x - y;
5 console.log("x = ", x, ", y = ", y);
```

このプログラムは、「x = 3, y = 2」と出力する。

複合代入演算子

また、「+=」、「-=」のような演算子は算術演算と代入を同時に行う演算子で、例えば `x += y` は `x = x + y` を表す。上のプログラム例は次のように書き直すことができる。

ファイル varSample2.js

```
1 var x = 2, y = 3;
2 x += y;
3 y *= -1;
4 y += x;
5 x -= y;
6 console.log("x = ", x, ", y = ", y);
```

インクリメントとデクリメント

JavaScript では「++」、「--」のような演算子もよく使われる。「++」は変数に1だけ加算（ ）する演算子、「--」は変数から1だけ減算（ ）する演算子である。`x++` は `x = x + 1`、`y--` は `y = y - 1` と（ほとんど）同じ意味である。

問 1.6.1 $1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8 \times 9$ の値を `alert` で表示する JavaScript プログラムを作成せよ。

問 1.6.2 変数名として漢字やギリシャ文字が使えることを確認する JavaScript プログラムを作成せよ。

1.7 DOM の操作

Document Object Model ([DOM](#)) は HTML ページのような文書の構造をオブジェクトとして表現したものである。HTML ページから読み込まれた JavaScript のプログラムは定義済みのグローバル変数を通して DOM を操作することによって、HTML ページの内容を読み込んだり、書き換えたりすることができる。特に `document` や `window` は、役に立つ多くのプロパティとメソッドを持つ。まず、`document` は現在のページを表すオブジェクトであり、`window` は現在のページを含むウインドウ（あるいはタブ）を表すオブジェクトである。

その他のグローバル変数については、MDN の Window インターフェイスに関するページに説明されている。

DOM の使用例

次の例:

ファイル `hello.html`

```
2 <p id="para">Hello!</p>
3 <script>
4   const para = document.getElementById("para");
5   para.style.color = "blue";
6   para.innerHTML += " Good Night!";
7 </script>
```

で使われている `document` のメソッド `getElementById(id)` は、HTML 要素を `id` 属性で指定して取得する。この例の場合は、「`<p id="para">Hello!</p>`」という要素（に対応するオブジェクト）を得ることになる。ここで、`p` は段落 (paragraph) を表す HTML のタグである。この要素の `style` プロパティは HTML の `style` 属性に対応する。また、`innerHTML` プロパティは、要素の子孫を文字列としてアクセスする。色 (`color`) を `blue` に変更し、`innerHTML` プロパティに `" Good Night!"` という文字列を後ろに追加する結果、この HTML 要素は「`<p id="para" style="color: blue;">Hello! Good Night!</p>`」と書いてあったのと同じ結果になる。

この例で示すように、代入演算子「`=`」はオブジェクトのプロパティを変更する場合にも使用できる。

以下によく使われる DOM のプロパティとメソッドの説明とその使用例をあげる。

`window` のプロパティ

location プロパティ

現在の文書の場所 (URL) を表す。

history プロパティ

現在のページが読み込まれているタブで過去に訪れたページの情報を表す。

navigator プロパティ

スクリプトを実行しているアプリケーション (ブラウザ) に関する情報を表す。

document のメソッド

createElement メソッド

`document.createElement(tagName)` の形で、`tagName` で示される HTML 要素を生成する。

createTextNode メソッド

`document.createTextNode(str)` の形で、新しい Text ノードを生成する。「<」 「&」 などの HTML で特殊な意味を持つ文字をエスケープするために利用できる。

要素 (Element) のメソッド

append メソッド

`element.append(something)` の形で、`element` の最後の子のあとに `something` を挿入する。`something` は、Element (正確には Node) か文字列である。文字列の場合は、Text ノードとして挿入される。

prepend メソッド

`element.prepend(something)` の形で、`element` の最初の子のまえに `something` を挿入する。

appendChild メソッド

`element.appendChild(child)` の形で、`element` の最後の子のあとに `child` を挿入する。`child` は Element である。

insertBefore メソッド

`element.insertBefore(child, sibling)` の形で、`element` の子の `sibling` の直前に `child` を挿入する。

insertAdjacentHTML メソッド

`element.insertAdjacentHTML(position, str)` は、`str` を HTML として構文解析し、その結果のノードを `element` の `position` で示される位置に挿入する。`position` は次のうちのどれかである ('beforebegin', 'afterbegin', 'beforeend', 'afterend')。

ここで、append メソッドと appendChild メソッドは似ているが、基本的には append が新しいメソッドで便利なので、これを使えば良い。ただし、Internet Explorer など古いブラウザは append メソッドに対応していないなど、互換性にやや難がある。

ファイル domSample.html

```
7 <p id="para"></p>
8 <script>
9   const para = document.getElementById("para");
10  const newElem = document.createElement("a");
11  newElem.href = "https://ja.wikipedia.org";
12  newElem.innerHTML = "Wikipedia";
13  para.appendChild(newElem);
```

```

14     const newText = document.createTextNode (
15         "<a href='https://github.com'>GitHub</a>");
16     para.append(newText);
17     para.insertAdjacentHTML('beforeend',
18         "<a href='https://google.com'>Google</a>");
19     para.prepend(document.createElement("br"));
20     const code = document.createElement("code");
21     code.innerHTML = "1 + 2 + 3";
22     para.appendChild(code);
23     const newSpan = document.createElement("span");
24     newSpan.style.color = "green";
25     newSpan.innerHTML = "WWW";
26     para.insertBefore(newSpan, newText);
27 </script>

```

この例は結局、次のような HTML を書いたのと同じ結果になる。(読みやすいように改行は追加している。)

```

1 <p id="para">
2   <br>
3   <a href="https://ja.wikipedia.org">Wikipedia</a>
4   <span style="color: green;">WWW</span>
5   &lt;a href='https://github.com'&gt;GitHub&lt;/a&gt;
6   <a href="https://google.com">Google</a>
7   <code>1 + 2 + 3</code>
8 </p>

```

ここで、br は改行 (line break)、a はハイパーリンク (anchor)、code はコンピュータコードの断片を表す HTML のタグである。また span は特に何かを表すものではないが、スタイルを指定するために使用されるタグである。

その他のメソッドについては、MDN の DOM に関するページの Document, Element, Node, Text などのリンク、HTML DOM API のページの HTMLElement, HTMLInputElement などのリンクから調べられる。

一方、場合によっては、実際にページに表示することなしに HTML や SVG のソースコードの構文解析が必要なことがあるかもしれない。その場合は DOMParser API を利用することができる。

CSS プロパティ

CSS は Cascading Style Sheet のことで、HTML 文書などの見栄えを表現するための言語である。HTML 要素の style プロパティから CSS プロパティを操作することができる。以下に、このプリントで使用する可能性のある style のプロパティを挙げる。なお、これらの例でわかるように、font-family や white-space のような、ハイフン「-」を含む CSS のプロパティは JavaScript のプロパティ名としては、ハイフン「-」を除いて、その直後の文字を大文字にして fontFamily, whiteColor のようにする。

color プロパティ

文字などの色を指定する。

backgroundColor プロパティ

CSS の background-color プロパティに対応する。要素の背景色を指定する。

fontFamily プロパティ

CSS の font-family プロパティに対応する。フォントファミリー名を指定する。

fontSize プロパティ

CSS の font-size プロパティに対応する。フォントのサイズを指定する。

display プロパティ

要素の表示方法を指定する。block, inline, noneなどを指定できる。特に none にすると、非表示となり、レイアウトにも影響を与えなくなる。

visibility プロパティ

要素の表示 (visible)/非表示 (hidden) を指定する。特に hidden にすると、表示はされないがレイアウトには表示されているのと同様に影響を与える。

borderStyle プロパティ

CSS の border-style プロパティに対応する。境界線の線のスタイルを指定する。solid, dotted, dashed, ...などを指定できる。

listStyleType プロパティ

CSS の list-style-type プロパティに対応する。箇条書きのマークを設定する。disc, circle, square, decimal, lower-roman, upper-roman, lower-latin, upper-latin, ...などを指定できる。

whiteSpace プロパティ

CSS の white-space プロパティに対応する。要素内の空白をどう扱うかを指定する。normal, nowrap (行を折り返さない), pre (連続する空白はそのまま残す), ...などを指定できる。

これらのプロパティは null あるいは空文字列 "" を代入することで、リセットする (つまり何も設定されていない状態に戻す) ことができる。

以下にこれらのプロパティの使用例を紹介する。

ファイル cssSample.html

```
7 <p><span id="span1">abc</span>
8   <span id="span2">lmn</span>
9   <span id="span3">xyz</span></p>
10 <script>
11   const span1 = document.getElementById("span1");
12   span1.style.color = "green";
13   span1.style.backgroundColor = "orange";
14   const span2 = document.getElementById("span2");
15   span2.style.visibility = "hidden";
16   const span3 = document.getElementById("span3");
17   span3.style.color = "blue";
18   span3.style.backgroundColor = "pink";
19   span3.style.borderStyle = "dashed";
20 </script>
```

これは次のように表示される。



その他の CSS プロパティについては、MDN の CSS リファレンスのページなどから調べられる。

CSS セレクターを用いるメソッド

HTML 要素の取得には、他に `document.querySelector(selectors)` や `document.querySelectorAll(selectors)` というメソッドが便利である。ここで `selectors` は CSS セレクターを表す文字列である。CSS セレクターは HTML 中の要素を指定するための書式である。前者は、CSS セレクターに _____ を返し、後者は CSS セレクターに _____ (厳密に言うと、`NodeList` というデータ型) を返す。

次のプログラム例:

ファイル `querySelector.html`

```
7 <ol>
8   <li>トマト</li>
9   <li>キュウリ</li>
10  <li>かぼちゃ</li>
11 </ol>
12 <ul>
13   <li>ハマチ</li>
14   <li>マグロ</li>
15   <li>カンパチ</li>
16   <li>サーモン</li>
17 </ul>
18 <script>
19   const elms = document.querySelectorAll("ul > li");
20   for (let i = 0; i < elms.length; i++) {
21     const elm = elms[i];
22     if (i % 2 == 1) {
23       elm.style.listStyleType = "square";
24     } else {
25       elm.style.listStyleType = "circle";
26     }
27   }
28 </script>
```

では、`"ul > li"` というセレクターが `ul` の直下の `li` 要素を示すので、`ul` (`ol` ではなく) の直下の `li` 要素だけが、`list-style-type` を変更されている。(配列や `for` 文に関しては、後で説明する。)

ここで `ol` は番号付き (ordered) のリスト (list)、`ul` は番号なし (unordered) のリスト、`li` はリストの項目 (item) を表すための HTML のタグである。

よく使われると思われる CSS セレクターには以下のようなものがある。

`a`, `span`, `li` など
指定された型の要素
`.classname`
クラス `classname` を持つ要素
`#idname`
`id` 属性が `idname` の要素
`A B`

A で選択される要素の子孫で B で選択される要素

A > B

A で選択される要素の直接の子で B で選択される要素

A + B

A で選択される要素の直後にある B で選択される要素

詳しくは MDN の CSS セレクターのページから調べることができる。

また、querySelector メソッド、querySelectorAll メソッドは、document だけではなく、要素にも用意されていて、その要素の子孫で条件を満たす要素を返すことができる。

ファイル querySelector2.html

```
7 <ol id='veges'>
8   <li>トマト</li>
9   <li class='foo'>キュウリ</li>
10  <li>かぼちゃ<span class='foo'>!?!</span></li>
11 </ol>
12 <ul id='fish'>
13   <li>ハマチ</li>
14   <li class='foo'>マグロ</li>
15   <li>カンパチ</li>
16   <li class='foo'>サーモン</li>
17 </ul>
18 <script>
19   const veges = document.getElementById("veges");
20   for (const elm of veges.querySelectorAll(".foo")) {
21     elm.style.color = "green";
22   }
23 </script>
```

この例では id='veges' の要素の子孫で foo クラスを持つ要素だけが緑色になる。

問 1.7.1 次の要件を満たす JavaScript プログラムを作成せよ。

1. HTML 要素 para1 のフォントを 'Times New Roman' に変更する
2. HTML 要素 para2 のフォントサイズを 24pt に変更する
3. HTML 要素 para3 のボーダーの色を blue に変更する

1.8 関数

関数は、繰り返し使用するプログラムの一部の命令列を部品として、再利用できるようにしたものである。

関数の定義

関数の定義の書き方も C 言語と良く似ているが、JavaScript では戻り値の型を書く必要がないので、C 言語で関数の戻り値の型を書く部分に、キーワード `return` を用いるところだけが異なる。また、仮引数の型を宣言する必要もない。戻り値を表す `return` 文の書き方も C 言語と同じである。

一般的に、関数の定義は次のような形をしている。

```
function 関数名(変数1, ..., 変数m) { 文1 ... 文n }
```

この関数を実行する（呼出す）には `関数名(式1, ..., 式m)` という形で使う。このとき、式₁ ~ 式_m の値が変数₁ ~ 変数_m にそれぞれ代入され、文₁ ~ 文_n が順に実行される。この変数₁ ~ 変数_m のことを _____ (parameter) と呼ぶ。

return 文

分類	一般形	補足説明
return 文	return 式 ;	値を返す場合
	return ;	値を返さない場合

return 文は関数の呼出し元に値を返す。つまり、プログラムの実行が関数の呼出し元に戻り、return 文の式の値が、関数呼出し式の値になる。

関数本体の最後の「}」にたどり着いた時も、プログラムの実行は関数の呼出し元に戻る。

例えば、次の例は3乗を計算する関数 cube である。

ファイル cube.js

```
1 // JavaScript
2 function cube(n) {
3   return n * n * n;
4 }
```

ファイル cube.c

```
3 /* (参考) C 言語 */
4 double cube(double n) {
5   return n * n * n;
6 }
```

この関数を使用する例は次のようになる。

```
console.log("4 の 3 乗は ", cube(4)); // 4 の 3 乗は 64
```

匿名関数

JavaScript では無名の関数 (anonymous function) を式として使うことができる。次のような形を用いる。

```
function (変数1, ..., 変数m) { 文1 ... 文n }
```

つまり、**function** というキーワードと括弧の間に関数名が入らない。このような匿名関数は変数やオブジェクトのプロパティに代入したり、関数呼出しの引数として渡したりすることができる。

アロー関数

ES6 から匿名関数をさらに短く書けるアロー関数という構文が用意された。(厳密に言うと、単なる構文の違いではなく、this というキーワードの指すオブジェクトが異なる、など意味の違いもある。)

分類	一般形	補足説明
アロー関数	(変数 ₁ , ..., 変数 _m) => { 文 ₁ ... 文 _n }	

value プロパティ

HTML の input タグに入力された値を JavaScript から取り出すときは value というプロパティを使用する。また parseInt は文字列を整数に変換するメソッドである。そして <input> の値が変更されたときのイベントは _____ である。

ファイル factorial.html

```
9 <input type="number" id="num" value="1">
10 の階乗は <span id="result">1</span>
11 <script>
12   const num = document.getElementById("num");
13   num.addEventListener("change", () => {
14     const n = parseInt(num.value);
15     let i, r = 1;
16     for (i = 1; i <= n; i++) {
17       r *= i;
18     }
19     document.getElementById("result").innerHTML = r;
20   });
21 </script>
```

なお、input の type 属性には number の他に text, color (色), date (日付)などを指定することができる。また value というプロパティは入力された値を文字列として取り出すが、valueAsNumber あるいは valueAsDate とすると、それぞれ数値、日付として解釈したものを取り出す。この input のほかに select タグと option タグの組み合わせで複数の選択肢を提供する入力要素も作成できる。

DOMContentLoaded イベント、load イベント

これまでの例のように JavaScript から getElementById メソッドなどを使って要素を取得する場合、HTML 中でその要素が定義されたあとに script タグを入れる必要があった。だから </body> の直前に script タグを入れることが多い。

しかし、script タグが HTML のどこにあっても、要素が正常に取得できることが望ましい。この場合、次の例のように document の _____ というイベントにリスナーをセットして、ページの DOM の読み込みが終わったときに実行されるようにしておく。(window の _____ という、画像やスタイルシートなどすべてが読み込まれたときに発生するイベントを利用するときもある。)

ファイル script.html

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8"/>
5   <script src="script.js"></script>
6 </head>
7 <body>
8   <button id="btn">click!</button>
```

```
9 <p id="para">Hello!</p>
10 </body>
11 </html>
```

ファイル script.js

```
1 document.addEventListener("DOMContentLoaded", () => {
2   const btn = document.getElementById("btn");
3   const para = document.getElementById("para");
4   btn.addEventListener("click",
5     () => para.innerHTML += "!");
6 });
```

要素を確実に取得できるようにするためには、ほかに script タグに defer="true" という属性を入れるという方法もある。

```
<script src="script.js" defer="true"></script>
```

この場合、スクリプトの読み込みはこの時点で開始されるが、スクリプトの実行は、ページの読み込みが完了した時点で始まる。

以降のプログラム例では、必要な場合、スクリプトは

```
document.addEventListener("DOMContentLoaded", () => {
  });
```

に囲まれているか、defer="true" で読み込まれているか、必要な要素

のあとで読み込まれている、と仮定する。

requestAnimationFrame メソッド

アニメーションに用いられる関数で、画面が再描画される前に呼び出されるコールバック関数を登録する。コールバック関数はミリ秒単位の現在時刻を引数として受け取る。

ファイル requestAnimationFrame.js

```
2   const p = document.getElementById("para");
3
4   function step(ts) {
5     // 10 秒で一周
6     const th = Math.trunc(ts * 36 / 1000) % 360;
7     p.style.color = "oklch(90% 80% " + th + ")";
8     p.innerHTML = th + "°";
9     requestAnimationFrame(step); // 間接的に再帰する
10  }
11
12  requestAnimationFrame(step);
```

このプログラムは、再描画のたびに HTML 要素の色と表示する文字列を変更してアニメーションを実現している。Math.trunc は、小数を受け取って、それから小数点以下を取り除いた整数を返すメソッドである。

似たような目的で用いられる関数として、一定の間隔で実行されるコールバック関数を登録する [setInterval](#) メソッドや、一定時間経過後に呼び出されるコールバック関数を登録する [setTimeout](#) メソッドがある。

テンプレートリテラル

テンプレートリテラル (template literal) は、途中に式を挿入することができる文字列定数である。ES6 から導入された。途中で改行して複数行にまたがることもできる。文字列をバッククォート「`」で囲む。

ファイル templateLiteral.js

```
3   const neko = `  
4     ^ ^  
5   /(\`A\`)ノ  
6     |  
7   `;  
8   p.style.fontFamily = "'MS ゴシック',monospace";  
9   p.style.whiteSpace = "pre";  
10  p.innerHTML = neko;
```

バッククォート自体を含めるにはバックスラッシュ「\」に続けてバッククォートを書く（「\`」）。

テンプレートリテラルの中には「`\${式}`」という形を含むことができる。この式の値が文字列に変換されて、この場所に埋込まれる。先ほどの requestAnimationFrame に対するプログラム例

(requestAnimationFrame.js) の一部は、次のように書くこともできる。

ファイル requestAnimationFrame2.js

```
7   p.style.color = `oklch(90% 80% ${th})`;  
8   p.innerHTML   = `${th}°`;
```

問 1.9.1 テキストフィールド（例: ）に入力すると、入力した文字列を別の要素の最後（または最初）に追加する JavaScript プログラムを作成せよ。

問 1.9.2 数値用のテキストフィールド（例: ）に入力すると、その 101 倍を計算して、別の要素の最後（または最初）に追加する JavaScript プログラムを作成せよ。

問 1.9.3 選択式メニュー `<select>~</select>`（例: ）で選択すると、別の要素の font-family を選択されたものに変更する JavaScript プログラムを作成せよ。

問 1.9.4 色入力用の要素 `<input type="color" >`（例: ）で、別の要素 para の文字色を変更する JavaScript プログラムを作成せよ。

1.10 制御構造

JavaScript の制御構造はやはり C 言語や Java とほとんど同じカタチが用意されている。繰返しの for 文については JavaScript 独自のカタチも追加されている。

if 文

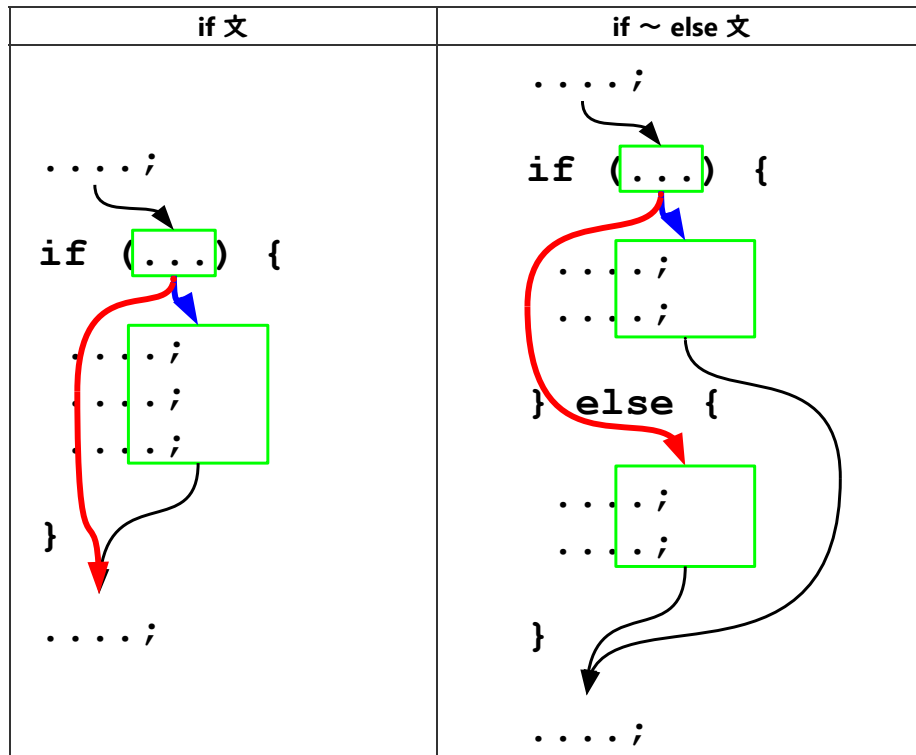
条件判断には次のようなカタチを用いる。

```
if ( 式1 ) 文1
```

式₁を評価して、その値が真であれば、 を実行する。式の値が偽であるときは、**何もしない**

```
if ( 式1 ) 文1 else 文2
```

式₁を評価して、その値が真であれば、 を実行する。式の値が偽であるときは、 を実行する。



等価演算子

「==」は両辺の値が等しければ真を、等しければ偽を返す演算子である。
 「==」と逆に等しくないかどうかを判定する演算子は「 」である。

関係演算子

以下の4つがある。

<	左辺が右辺よりも小さいとき真
>	左辺が右辺よりも大きいとき真
<=	左辺が右辺よりも小さいか等しいとき真
>=	左辺が右辺よりも大きいか等しいとき真

「=<」とか「=>」という演算子はないので注意する。（「=>」はアロー関数に使用される。）

ブロック（複合文）

文の並びを波括弧 (ブレース — 「`{`」 と 「`}`」 —) で囲んだものを複合文またはブロックという。(文の前や間にいくつかの宣言があってもよい。) 複合文は構文上単一の文と見なされる。 複合文中の文は上 (左) から順に一つずつ実行される。

通常、if 文の制御する文 (後述の while 文、for 文などでも同様) は、たとえ一つの文でも (間違いを避けるため) 波括弧で囲んでブロックにする。

△ 望ましくないスタイル	◎ 望ましいスタイル
<pre>if (n1 > n2) max = n1; else max = n2;</pre>	<pre>if (n1 > n2) { max = n1; } else { max = n2; }</pre>

論理演算子

論理演算子は以下のような演算子である。左右非対称である — つまり左オペランドを評価して値が決まれば、_____ は評価しない — ことに注意する。(これを短絡評価という。)

演算子	呼び方	説明
<code>&&</code>	論理 AND 演算子 かつ	左オペランドを評価して、偽であれば、偽を返す。真であれば、右オペランドを評価してその値を返す。
<code> </code>	論理 OR 演算子 または	左オペランドを評価して、真であればその値を返す。偽であれば、右オペランドを評価してその値を返す。

次の例は if 文を使って、現在時刻にあわせて表示するメッセージを変える。

ファイル `if.js`

```
2  const now = new Date(); // 現在時刻を取得する
3  const h   = now.getHours(); // 「時」の部分を取り出す
4  let msg;
5  if (h <= 10) {
6    msg = "おはようございます";
7  } else if (h <= 18) {
8    msg = "こんにちは";
9  } else {
10   msg = "こんばんは";
11 }
12 const para = document.getElementById("para");
13 para.innerHTML = msg;
```

条件演算子

三項演算子とも呼ばれる。 `式1 ? 式2 : 式3` の値は 式₁ をまず評価して結果が真であれば 式₂ を評価した値となる。式₁ が偽であれば、式₃ を評価した値となる。上の if.js は次のように書き換えることもできる。

ファイル `conditional.js`

```
2  const now = new Date(); // 現在時刻を取得する
3  const h   = now.getHours(); // 「時」の部分を取り出す
```

```

4 | const msg = h <= 10 ? "おはようございます"
5 |             : h <= 18 ? "こんにちは"
6 |             : "こんばんは";
7 | const para = document.getElementById("para");
8 | para.innerHTML = msg;

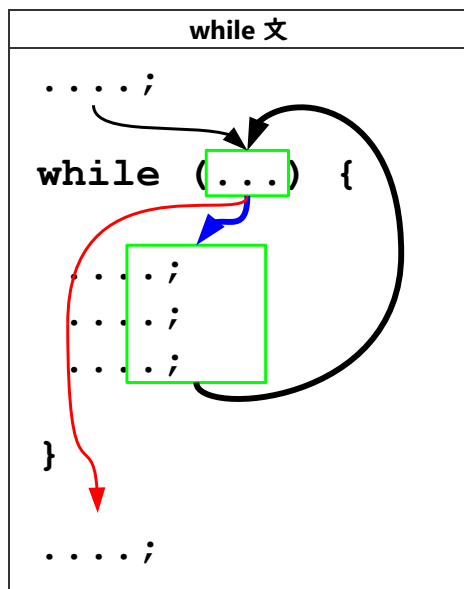
```

while 文

```
while ( 式1 ) 文1
```

式₁ が真である限り、___ (ループ本体) の実行を繰り返す。

注: ループ本体が一度も実行されないことがある。



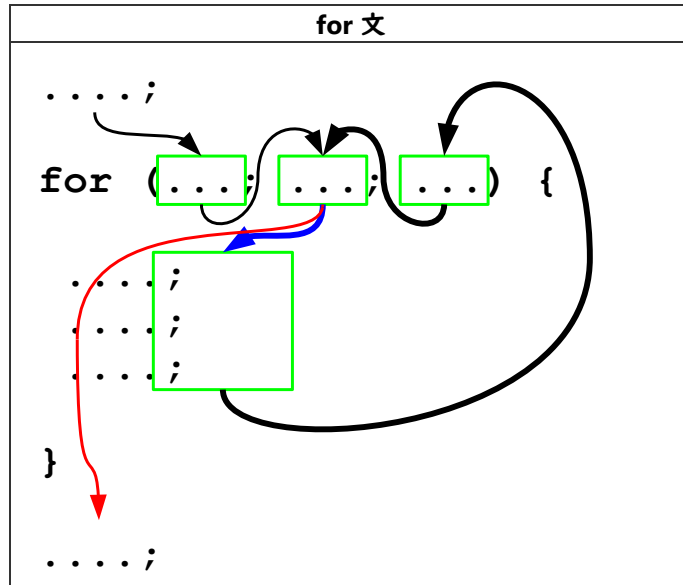
for 文

```
for ( 式1; 式2; 式3 ) 文1
```

ループに入る前にまず ___ を実行する。

___ が真である間、___ (ループ本体) と ___ を繰り返し実行する。

詳細: 式₁ ~ 式₃は省略可能である。式₂を省略したときは、true と書くのと同じ意味になる。



次は、for 文を使って、文字の色を徐々に変える例である。

ファイル for.js

```

2   const para = document.getElementById("para");
3
4   for (let i = 0; i < 24; i++) {
5       const s = document.createElement("span");
6       s.style.color = `hsl(${i * 15}, 100%, 50%)`;
7       s.innerHTML = "■"; // 直接絵文字を書いても OK
8       para.appendChild(s);
9   }

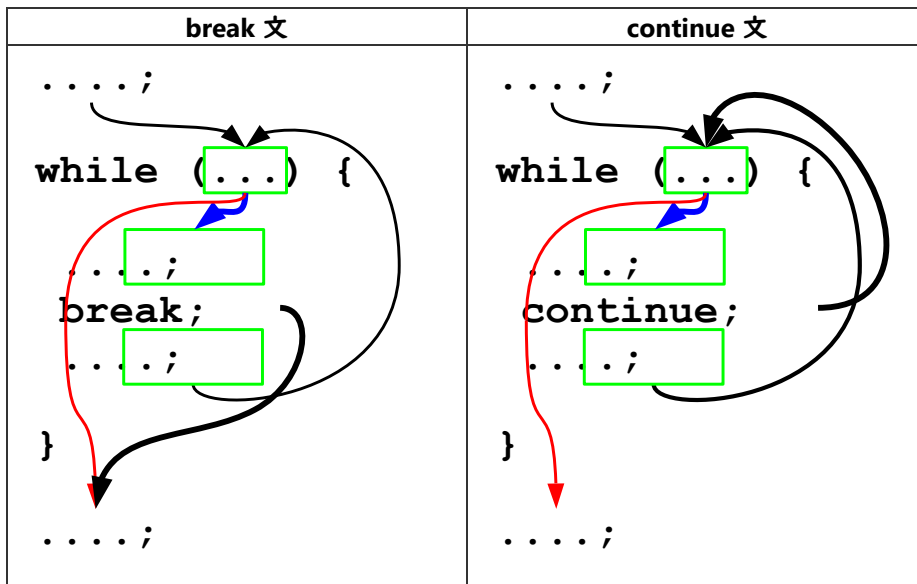
```

break 文

(もっとも内側の) 繰返し文 (do ~ while 文, while 文, for 文) を _____。
 (外側の繰返し文を一気に抜け出すことはできない。)

continue 文

(もっとも内側の) 繰返し文のはじめ (do ~ while 文、while 文の場合は条件式、for 文の場合は第 3 式) にもどる。



try ~ catch 文

例外処理をするために使われる try ~ catch 文は

```
try ブロック1 catch (変数) ブロック2
```

という形で用いる。

この形は、まずブロック₁を実行する。この中でエラーが起こらなければ、そのまま次へ進む。ブロック₁の中で**例外**（エラーと考えて良い）と呼ばれる状況が起こったとき、catch の後ろのブロック₂を実行する。“変数”に例外の情報を持つオブジェクトが初期値として渡されてブロックが実行される。

また、最後に `finally ブロック3` という形（finally節）がつく場合もある。その場合、finallyブロックは例外が起こったか否か、例外がキャッチされたか否か、にかかわらず、必ず実行される。

throw 文

プログラムにより、例外を発生させるには throw 文を用いる。 `throw 式;`

問 1.10.1 ボタンをクリックすると、別の要素 para を隠す・表示する JavaScript プログラムを作成せよ。

問 1.10.2 ボタンをクリックすると、別の要素 para に表示したカウンター「あなたは???回クリックしました。」が増加し、キリ番（10の倍数）になると、特別なメッセージを表示する（あるいは色を変える）JavaScript プログラムを作成せよ。

問 1.10.3 日付入力用の要素 `<input type="date" >`（例: `2025/07/29`）で日付を入力して、別の要素 para に表示するが、土・日ならばそれぞれ青・赤に色を変える JavaScript プログラムを作成せよ。（参考: MDN の Date, MDN の `<input type="date">`）

問 1.10.4 CSS の `hsl()` または `oklch()` を使って、2次元の色の、色相 (H) と彩度 (S) を変えて輝度 (L) を固定するグラデーションを作成する JavaScript プログラムを作成せよ。

参考: `hsl(...)`, `oklch(...)` については [hsl\(\) \(MDN\)](#)、[oklch\(\) \(MDN\)](#)、[色選択ツール \(MDN\)](#)、[説明図](#)、[OKLCH Color Picker & Converter](#)などを参考にせよ。

1.11 配列

配列の宣言

配列は (同じ種類の) データを集めて、番号 (、そえじ) でアクセスできるようにしたものである。次の例で示すように、各要素をコンマ「,」で区切り、周りを角括弧 (「[」 ~ 「]」) で囲む

```
var colors = ["red", "green", "yellow", "cyan"];
```

分類	一般形	補足説明
配列アクセス	式 [式]	a[1], b[2][3] など

配列の各要素は `配列名[式]` でアクセスする。「[」と「]」の式を添字という。JavaScript の配列の有効な添字は から始まる。つまり、上の例では `colors[0]` は "red"、`colors[1]` は "green" である。

for ~ of 文

```
for ( 変数1 of 式1 ) 文1
```

式₁ は配列 (などの物の集まりを表すデータ) で、変数₁ を集まりの各要素に割り当てて、 の実行を繰り返す。

ファイル `forOf.js`

```
2   const para = document.getElementById("para");
3
4   const colors = ["pink", "gold", "olive", "aqua"];
5   for (const c of colors) {
6       const s = document.createElement("span");
7       s.style.color = c;
8       s.innerHTML = c;
9       para.appendChild(s);
10      para.append(', ');
11  }
```

この例では、変数 `c` が "pink", "gold", "olive", "aqua" の 4 つの値になって、繰り返し部分を繰り返す。

配列のプロパティ・メソッド

配列のプロパティ `length` は配列の要素数を表す。C 言語や Java の配列と違って、JavaScript の配列は自由に伸び縮みすることができる。配列に対して次のようなメソッドが用意されている。

push メソッド

`arr.push(e1, e2, ...)` の形で配列 `arr` の末尾に要素 `e1, e2, ...` を追加する。

pop メソッド

`arr.pop()` の形で配列 `arr` の末尾の要素を取り除き、その要素を返す。

unshift メソッド

`arr.unshift(e1, e2, ...)` の形で配列 `arr` の先頭に要素 `e1, e2, ...` を追加する。

shift メソッド

`arr.shift()` の形で配列 `arr` の最初の要素を取り除き、その要素を返す。

sort メソッド

`arr.sort(e)` の形で配列 `arr` を昇順にソートする。

join メソッド

`arr.join(str)` の形で配列 `arr` の要素を連結した文字列を返す。区切り文字は `str` である。

indexOf メソッド

`arr.indexOf(e)` の形で配列 `arr` の中で `e` と同じ要素が最初に現れる添字を返す。現れない場合は `-1` を返す。

at メソッド

`arr.at(n)` の形で配列 `arr` の添字 `n` の位置の要素を返す。特に `n` が負の値のときは末尾からカウントする

ファイル `array.js`

```
15 addColorSample(p, colors);
16 colors.push("pink"); // 末尾に追加する
17 const c1 = colors.shift(); // 先頭を削除する
18 const c2 = colors.pop(); // 末尾を削除する
19 colors.unshift("fuchsia"); // 先頭に追加する
20 colors.push(c1);
21 colors.unshift(c2);
22 addColorSample(p, colors);
23 colors.sort(); // 辞書順にソートする
24 addColorSample(p, colors);
25 p.append(`${ " ", colors.join(", "), " }`);
26 p.insertAdjacentHTML("beforeend", "<br>");
27 p.append(`pink は ${colors.indexOf("pink")} 番目`);
28 p.insertAdjacentHTML("beforeend", "<br>");
29 p.append(colors.at(-2));
```

ちなみに `addColorSample` は次のように定義した関数である。

ファイル `array.js`

```
1 function addColorSample(elem, cs) {
2   for (const c of cs) {
3     const s = document.createElement("span");
4     s.style.color = c;
5     s.innerHTML = c;
6     elem.appendChild(s);
7     elem.append(", ");
8   }
}
```

```

9 |     elem.insertAdjacentHTML("beforeend", "<br>");
10 | }
11 |

```

多次元配列

配列の配列は多次元配列とも呼ばれる。例えば次のように二次元配列を定義することができる。

ファイル array2d.js

```

2 |     const board = [ ["blue", "lightgray", "red"],
3 |                     ["black", "yellow", "red"],
4 |                     ["orange", "ivory", "green"] ];
5 |
6 |     const ta = document.getElementById("ta");
7 |     for (let i = 0; i < board.length; i++) {
8 |         const row = document.createElement("tr");
9 |         ta.append(row);
10 |         for (let j = 0; j < board[i].length; j++) {
11 |             const c = board[i][j];
12 |             row.insertAdjacentHTML("beforeend",
13 |                 `<td style="color: ${c};">■</td>`);
14 |         }
15 |     }

```

配列の配列の各要素には角括弧を `配列名[式1][式2]` のようにつなげて書くことでアクセスすることができる。この場合、`board[0][0]` は "blue"、`board[1][2]` は "red" である。（上の例は for ~ of 文を使うことで角括弧を使わずに書くこともできる。）

配列に対する高階関数

配列に対して、便利なので高階関数（関数を引数に取る関数）が、いくつか用意されている。

なかでも `map`、`filter`、`flatMap`、`reduce`、`forEach` などがよく使われる。

`xs.map(f)` は、配列 `xs` の各要素に関数 `f` を適用した結果を要素とする新しい配列である。`xs.filter(p)` は、配列 `xs` の各要素のうち、関数 `p` を適用した結果が真となる要素だけからなる新しい配列である。`xs.flatMap(f)` は、配列 `xs` の各要素に関数 `f` を適用した結果を接続した新しい配列である。例えば、`[1, 2, 5, 8].flatMap(v => v % 2 == 0 ? [v - 1, v + 1] : [])` は `[[], [1, 3], [], [7, 9]]` である。（ちなみに、`[1, 2, 5, 8].map(v => v % 2 == 0 ? [v - 1, v + 1] : [])` は `[[], [1, 3], [], [7, 9]]` である。）`nodes.forEach(f)` は、`xs.map(f)` と似ているが、戻り値を持たない。例えば、`querySelectorAll` メソッドの戻り値は `map` メソッドは持っていないが `forEach` メソッドは持っている。次にこれらの関数を使ったプログラム例を示す。

ファイル higherOrderFunction.js

```

1 | function addColoredText(elm, c, txt) {
2 |     const s = document.createElement("span");

```

```

3   s.innerHTML = txt;
4   s.style.color = c;
5   elm.appendChild(s);
6 }
7
8 document.addEventListener("DOMContentLoaded", () => {
9   const para = document.getElementById("para");
10
11   const xs = [0, -10, 24, 182, 37, 652, -76, 93];
12   const ys = xs.filter(x => 0 <= x && x <= 360);
13   const zs = ys.map(x => `hsl(${x} 100% 50%)`);
14   para.append(ys);
15   para.appendChild(document.createElement("br"));
16   zs.map(c => addColoredText(para, c, "@"));
17   para.appendChild(document.createElement("br"));
18   const ws = [128, 255].flatMap(r =>
19     [85, 170, 255].flatMap(g =>
20     [64, 128, 192, 255].map(b =>
21     `rgb(${r} ${g} ${b})`));
22   para.append(ws);
23   para.appendChild(document.createElement("br"));
24   ws.map(c => addColoredText(para, c, "#"));
25 });

```

ここで `ws` は `r` が、128 と 255、`g` が 85 と 170 と 255、`b` が 64 と 128 と 192 と 255 のいずれかを取る、すべての `r, g, b` の組み合わせを列挙したリストになる。

分割代入

ES6 から、代入の左辺に要素が変数になっている配列を書くことができるようになった。これを 分割代入 (destructuring assignment) という。対応する位置にある値が、変数に代入される。

ファイル `destructuring.js`

```

4   const cs = ['red', 'yellow', 'green', 'blue'];
5   let [x, y, ...rest] = cs; // x が 'red'、
6   // y が 'yellow'、rest が ['green', 'blue']
7   [y, x] = [x, y];        // x と y の入れ替え

```

ここで `...` は rest parameters という構文で、上の例では、`x, y` に割り当てられなかった、残りの配列の要素が `rest` という配列にまとめられる。

スプレッド構文

逆に `...` を配列の初期値が並ぶ中などで使う形を スプレッド構文 (spread syntax) と呼ぶ。`...` の後に式を書いて、配列などの中に別の配列などを展開することができる。

ファイル `spreadSyntax.js`


```






















2   const fruits = ["apple", "banana", "orange"];
3   const veges = ["tomato", "lettuce"];
4   const foods = [...veges, "tofu", ...fruits];


```

この例では、foods の値は ["tomato", "lettuce", "tofu", "apple", "banana", "orange"] になっている。

問 1.11.1 配列 `const fontNames = ["Times New Roman", "Courier New", "Cooper", "Consolas", "Calibri", "Cambria", "Chiller"];` から選択式メニュー (`<select>~ </select>`) の中身を生成する JavaScript プログラムを作成せよ。

問 1.11.2  の絵文字を 0.5 秒ごとに順に表示してアニメーションする JavaScript プログラムを作成せよ。

注意: JavaScript は文字列の内部表現に UTF-16 を使用しており、 などの文字は 2 文字扱いになる。このため " という文字列から `charAt` メソッドで取り出すというやり方はうまく行かない。配列 `["", "", "", "", "", "", "", "", "", ""]` を使うなどのやり方にする。


問 1.11.3  画像ファイルを複数用意し、12 fps 程度で画像を入れ替えて、アニメーションを生成する JavaScript プログラムを作成せよ。


問 1.11.4 (`array2d.html` の) `array2d.js` を `for ~ of` 文を用いて書き換えよ。

問 1.11.5 すべての `li` 要素の最初と最後に何かの絵文字を挿入する JavaScript プログラムを作成せよ。(`querySelectorAll` の戻り値は厳密には配列ではないが、 `forEach` メソッドは利用できる。)

問 1.11.6 すべての `ul` の直下の `li` 要素の最初と最後に何かの絵文字を挿入する JavaScript プログラムを作成せよ。

問 1.11.7 すべての `h2` の直後の `p` 要素の文字色を赤色にする JavaScript プログラムを作成せよ。

問 1.11.8  オセロゲーム JavaScript プログラム / ドット絵エディター JavaScript プログラムを作成せよ。

問 1.11.9  `<textarea>~</textarea>` に対するコンマ区切りのユーザー入力から配列をつくり、それからさらにグラフやオセロの盤面を作成する JavaScript プログラムを作成せよ。

1.12 オブジェクト

オブジェクトリテラル

オブジェクトリテラル (object literal) はオブジェクトの表現の一つで、プロパティ名とその値をコロン「:」で区切って対にしたものを、コンマ「,」区切りにして並べ波括弧「{」~「}」で囲んだものである。

```
var pt = { x: 9, y: 11, z: -2 };
```

これで `pt.x` が 9, `pt.y` が 11, `pt.z` が -2 になる。

オブジェクトリテラルを左辺とする分割代入も可能である。ただし、ブロックを書いてもよい箇所（つまり、ほとんどの箇所）で使うときは、ブロックの開始と混同しないように、適宜代入式を丸かっこで囲む必要がある。

```
{ x: tx, y: ty } = pt); // tx が 9, ty が 11 になる
```

this

関数定義の中の `this` は基本的には“メソッドを所有しているオブジェクト自身”を指す。つまり、`obj.foo(a1, a2, ...)` という形で `foo` が呼び出されたとき、`foo` の定義の中の `this` は `obj` を指している。

ファイル `this.js`

```
4  const obj = {
5    a: "abc",
6    foo: function() {
7      return this.a;
8    }
9  };
10 const q = obj.foo(); // "abc"
```

アロー関数 `(... => ...)` の中での `this` は、アロー関数が定義されている場所の周囲の `this` と同じである。

また `bind` というメソッドは、与えられた関数の `this` の値を固定した新しい関数を返す。

ファイル `this.js`

```
12 const obj2 = { a: "xyz" };
13 const foo2 = obj.foo.bind(obj2);
14 const q2 = foo2(); // "xyz"
```

上で述べた場合以外での `this` の使用は、わかりにくい可能性があるので注意する必要がある。

問 1.12.1 次のような形式のオブジェクトの配列から、

```
[
  { zip: "100-0014",
    city: "東京都千代田区",
    address: "永田町二丁目3-1",
    family: "日本",
    given: "花子" },
  { zip: "760-0017",
    city: "香川県高松市",
    address: "番町四丁目1-10",
    family: "讃岐",
    given: "太郎" },
  :
]
```

]

下の例のようなテーブル (<table>~ </table>) の中身を生成する JavaScript プログラムを作成せよ。

郵便番号	市町村	住所	氏名
100-0014	東京都千代田区	永田町二丁目3-1	日本 花子
760-0017	香川県高松市	番町四丁目1-10	讃岐 太郎
			⋮

問 1.12.2 ⚠ 配列の `map` メソッドを使って、上の問のようなオブジェクトの配列からテーブル (<table>~</table>) の中身を生成する JavaScript プログラムを作成せよ。

問 1.12.3 ⚠ さらに、配列の `filter` メソッドを使って、特定の条件を満たす行 (例えば、郵便番号が7から始まる) を取り除いてからテーブルを生成する JavaScript プログラムを作成せよ。

1.13 その他のデータ型

正規表現 (regular expression)

正規表現は接続・選択・反復の3つの基本演算で文字列の集合を表す手法である。詳しくは MDN の正規表現に関するページに説明されている。

JavaScript の正規表現リテラルは _____ 「/」で囲まれた部分である。後ろの「/」のあとにいくつかのフラグ (flag) をつけることがある。例えば、「g」は、最初のマッチだけではなく、最後まで検索することを示す。また、「i」は大文字・小文字を区別しないことを示す。

文字列の `replace` メソッドは、文字列中の正規表現にマッチした部分を、置換文字列に置き換えた新しい文字列を返す。置換文字列の中の、`$1`, `$2`, ... などは、正規表現のなかの1番目、2番目、... の丸括弧内にマッチした部分に置き換わる。

ファイル `regex.js`

```
2   const p   = document.getElementById("para");
3   const pat = /Name: +([A-Za-z]+) +([A-Za-z]+)/g;
4   const out = 'Name: <span style="font-variant: small-caps;">$2</span>, $1';
5   const source = `
6   Name: Masaya Otsuki
7   Age: 22
8   City: Takamatsu
9
10  Name: Yasuhiro Komoto
11  Age: 36
12  City: Kyoto
13
14  Name: Maya Usui
15  Age: 40
16  City: Matsue
17  `;
```

```
18 |   const result = source.replace(pat, out);
19 |   p.style.whiteSpace = 'pre';
20 |   p.innerHTML = result;
```

このプログラムは「Name:」のあとの「名 姓」を「姓, 名」に置き換え、姓の部分を SMALL CAPS にする。つまり、「Masaya Otsuki」を「OTSUKI, Masaya」に置き換える。

正規表現が文字列にマッチするかどうかだけを知りたいときは、正規表現の test メソッドを用いる。

```
const regex = /[0-9]+/;
console.log(regex.test("shibuya109")); // true
console.log(regex.test("Hayashi"));    // false
```

文字列 (String) のメソッド

文字列 (String) に関する便利なメソッドをいくつか紹介する。詳しくは MDN の String に関するページに説明されている。

charAt メソッド

文字列の n 文字目を取り出す。例えば "abcd".charAt(2) は になる。

substring メソッド

`str.substring(m, n)` は文字列 `str` の m 文字めから $n-1$ 文字めまでの部分文字列を取り出す。例えば "Takamatsu".substring(2, 5) は になる。

match メソッド

文字列のなかで正規表現にマッチする部分を取り出す。例えば "akb48 boeing787".match(/[0-9]+/g) は になる。

split メソッド

文字列を指定した区切り文字列で分割する。例えば "03-1234-5678".split("-") は になる。

また split メソッドの逆に、文字列の配列から区切り文字列を指定して、一つの文字列を生成するには、配列の join メソッドを用いる。例えば ["pen", "pineapple", "apple", "pen"].join("-") は になる。

数学関係 (Math) のメソッド

Math.~ という名前で数学関係のメソッドや定数がまとめられている。やはり MDN の Math にまとめられている。

Math.random メソッド

Math.random() は 0 以上 1 未満の疑似乱数を返す。

Math.PI

円周率を返す。

Math.sin メソッド

Math.sin(x) は x (ラジアン) のサイン (正弦) を返す。

Math.cos メソッド

Math.cos(x) は x (ラジアン) のコサイン (余弦) を返す。

などなど。

問 1.13.1 正規表現を使って input から入力された文字列が JavaScript の識別子（あるいは他の言語の識別子）であるかをチェックする JavaScript プログラムを作成せよ。

1.14 ジェネレーター

ES6 よりジェネレーター (generator) 関数という機構が導入された。ジェネレーター関数はコルーチンの一種 (stackless coroutine) を提供する。コルーチンは、関数とは異なり、二度目以降の呼出しのとき、直前の呼出しの続きから実行を開始する。

ジェネレーター関数は `function*` というキーワードで定義される。次の例はフィボナッチ数列を生成するジェネレーター関数である。

ファイル generator.js

```
1 function* gfib(n) {
2   let a = 1, b = 1;
3   while (a < n) {
4     yield a;
5     let tmp = b;
6     b += a;
7     a = tmp;
8   }
9 }
```

ジェネレーター関数の内部では `yield` というキーワードを使って値を生成する。

ジェネレーター関数を呼び出すと、すぐに関数内部のコードが実行されるのではなく、一旦、ジェネレーター (generator) オブジェクトが作られて返される。このジェネレーターオブジェクトの `next` メソッドを呼び出すと、ジェネレーター関数内部のコードが実行され、`yield` された値を `value` プロパティーとして持つオブジェクトを返す。さらに `next` メソッドを呼び出すと yield 式の続きから実行が再開され、やはり、次の `yield` された値を `value` プロパティーとして持つオブジェクトを返す。

ファイル generator.js

```
14 const gen = gfib(100);
15 para.append(gen.next().value); // 1 を出力する
16 para.append(", ");
17 para.append(gen.next().value); // 1 を出力する
18 para.append(", ");
19 para.append(gen.next().value); // 2 を出力する
20 para.append(", ");
21 para.append(gen.next().value); // 3 を出力する
```

ジェネレーターオブジェクトは `for ~ of` 文の中でも使うことができる。

`for ~ of` 文はジェネレーターオブジェクト（より一般的には `iterable` オブジェクト）の `next` メソッドによって返されるオブジェクトの `value` プロパティー

一を変数に代入してループする。ジェネレーター関数の中のコードの実行が return するか、ジェネレーター関数の本体を抜けるとループを終了する。

ファイル generator.js

```
24 for (const v of gfib(10)) {
25   para.append(v); // 1, 1, 2, 3, 5, 8 を出力する。
26   para.append(", ");
27 }
```

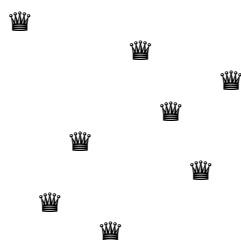
次のコードは素数列 (_____) を生成するジェネレーターである。

ファイル primes.js

```
1 function* from(n) {
2   while (true) {
3     yield n;
4     n++;
5   }
6 }
7
8 function* sieve(n, gen) {
9   for (const m of gen) {
10    if (m % n !== 0) yield m;
11  }
12 }
13
14 function* primes() {
15   let gen = from(2);
16   while (true) {
17     const next = gen.next().value;
18     yield next;
19     gen = sieve(next, gen);
20   }
21 }
```

このジェネレーターは「エラトステネスのふるい」というアルゴリズムを実装している。from(n) は単に $n, n+1, n+2, \dots$ という、1 ずつ増加する数列を生成するジェネレーターである。sieve(n, gen) は gen というジェネレーターが生成する数列から n の倍数を除いた数を生成するジェネレーターになる。

最後にパズルを解くためにジェネレーターを使っている例を挙げる。次の例はエイト・クイーンという有名なパズルの解を計算する。8×8 のチェス盤の上に 8 つのクイーンを、どの駒も他の駒に取られないように配置する。このプログラムでは解を配列で表している。例えば [1, 5, 8, 6, 3, 7, 2, 4] は第1行の第1列、第2行の第5列、第3行の第8列、... という解で次のような配置を表す。



ファイル queens.js

```
1 function safe(arr, k) {
2   const len = arr.length
```

```

3   for (let i = 0; i < len; i++) {
4       const v = arr[i];
5       if (v == k
6           || v + len - i == k
7           || v - len + i == k) {
8           return false;
9       }
10  }
11  return true;
12 }
13
14 function* queens(n) {
15     if (n == 0) {
16         yield [];
17     } else {
18         const sofar = queens(n - 1);
19         for (const ans of sofar) {
20             for (let k = 0; k < 8; k++) {
21                 if (safe(ans, k)) {
22                     yield [...ans, k];
23                 }
24             }
25         }
26     }
27 }

```

`safe(arr, n)` は `arr.length` 行目まで与えられている配置 `arr` に新しく `(arr.length + 1, n)` のマスに新しくクイーンを置いて安全かどうか、を表す関数である。

`queen(n)` は `n - 1` 行目までの安全な配置を `queen(n - 1)` で再帰的に求めて、`n` 行目の安全な配置を生成する。

問 1.14.1 等比数列（例えば $\{1, 2, 4, 8, 16, 32, \dots\}$ ）を生成する無限のジェネレーター関数を定義せよ。

問 1.14.2 引数 k を受取り、

$$\begin{cases} a_1 = k \\ a_n = \frac{a_{n-1}}{2} & n \geq 2 \text{ かつ } a_{n-1} \text{ が偶数のとき、} \\ a_n = 3a_{n-1} + 1 & n \geq 2 \text{ かつ } a_{n-1} \text{ が奇数のとき、} \end{cases}$$

で定義される数列 $\{a_n\}$ を生成する無限のジェネレーター関数を定義せよ。

1.15 モジュール

2015 年に制定された ECMAScript 6 からモジュールという仕組みが追加された。一般的にモジュールとは、まとまりのある機能を持った部品のことであり、他のプログラミング言語にはよくある、このような仕組みを JavaScript は長い間公式には持っていなかった。これにより、大きなプログラムをモジュールという単位に分割し、必要に応じて読み込むことができるようになった。また、重複して同じコードを読み込むことが防がれるようになった。

モジュールを HTML から読み込むときは `script` タグに `type="module"` と指定する。JavaScript でモジュールの機能を利用するためには `export` 文と `import` 文を使用する。

export 文

関数やクラスの定義や変数の宣言の前に `export` というキーワードをつける。その関数や変数は、モジュールの外部に公開される。

```
export const foo = 41;
export function bar(n) {
  return n * n;
}
```

または、モジュールファイルの最後に、`export` 文を書いて、いくつかの関数や変数を一度に公開することもできる。

```
const foo = 41;

function bar(n) {
  return n * n;
}

function baz(m) {
  return m * m * m;
}

export { foo, bar, baz };
```

import 文

他のモジュールで公開されている関数などを利用する場合は `import` 文でインポートする必要がある。例として `./modules/foobar.js` という場所にあるファイルで定義されているモジュールをインポートするとする。次のような `import` 文を使うと `foo`, `baz` が利用できる。

```
import { foo, baz } from './modules/foobar.js';
```

また次のような `import` 文は `./modules/baz.js` が `export` している関数を `BAZ` という接頭辞付きで呼ぶことができるようにする。例えば `baz.js` が `qux()` という関数を `export` していれば、`BAZ.qux()` のように呼び出すことができる。

```
import * as BAZ from './modules/baz.js';
```

このようにインポートされた関数や変数は、同じファイルで定義された関数や変数と同様に使用できる。

モジュールに関しては他に `default` というキーワードを使うデフォルトエクスポートなどがあるが、ここでは説明を割愛する。必要に応じて MDN の JavaScript モジュールに関するページなどから調べられる。