

# Table of Contents

この書籍について	1.1
読み始める前の事前準備	1.2
本書の目的	1.3
文章の間違いに気づいたら	1.4
第1部: 基礎文法	1.5
JavaScriptとは	1.5.1
コメント	1.5.2
変数と宣言	1.5.3
値の評価と表示	1.5.4
データ型とリテラル	1.5.5
演算子	1.5.6
暗黙的な型変換	1.5.7
関数と宣言	1.5.8
文と式	1.5.9
条件分岐	1.5.10
ループと反復処理	1.5.11
オブジェクト	1.5.12
プロトタイプオブジェクト	1.5.13
配列	1.5.14
文字列	1.5.15
ラッパーオブジェクト	1.5.16
関数とスコープ	1.5.17
関数とthis	1.5.18
クラス	1.5.19
例外処理	1.5.20
非同期処理	1.5.21
Map/Set	1.5.22
JSON	1.5.23
Date	1.5.24
Math	1.5.25
ECMAScript	1.5.26
第1部: おわりに	1.5.27
第2部: 応用編 (ユースケース)	1.6
アプリケーション開発の準備	1.6.1
モジュール	1.6.2
Ajaxで通信	1.6.3
エントリーポイント	1.6.3.1

---

HTTP通信	1.6.3.2
データを表示する	1.6.3.3
Promiseを活用する	1.6.3.4
Node.jsでCLIアプリ	1.6.4
Node.jsでHello World	1.6.4.1
コマンドライン引数を処理する	1.6.4.2
ファイルを読み込む	1.6.4.3
MarkdownをHTMLに変換する	1.6.4.4
ユニットテストを記述する	1.6.4.5
Todoアプリ	1.6.5
エントリポイント	1.6.5.1
アプリの構成要素	1.6.5.2
フォームとイベント	1.6.5.3
イベントとモデル	1.6.5.4
Todoの更新と削除を実装する	1.6.5.5
Todoアプリのリファクタリング	1.6.5.6

---

## [WIP] JavaScriptの入門書

ECMAScript 2018時代のJavaScript入門書

[Tweet](#) [Watch](#) [Star](#)

これからJavaScriptを始める人がES2015以降をベースにして学べる本(予定)  
プログラミングをやったことがあるが、今のJavaScriptがよくわからないという人が、今のJavaScriptアプリケーションを読み書きできるようになるもの。

### 更新情報を購読

この本は現在実装中であるため、予告なしに変更される可能性があります。  
この本の更新情報を受け取りたい方はメールアドレスを登録することで通知を受け取れます。

メールアドレス

[登録](#)



No Image

## 読み始める前の事前準備

この書籍には、すでにプログラミングしたことがある人向けの記述も含まれています。そのため、この書籍でプログラミング言語を初めて学ぶ場合には、まずコードに慣れる必要があります。

ブラウザがあればJavaScriptのコードは実行できます。

まずは、ブラウザをインストールしてください。ブラウザにはMicrosoft Edge、Firefox、Google Chrome、Safariなどがありますが、この書籍ではFirefoxを使用していきます。すでにブラウザをインストール済みの場合も、最新のバージョンを使っているかを確認してください。

ブラウザの準備ができたら、次のURLをブラウザでひらいてください。

- <https://jsprimer.net/intro/preparation/>

次のJavaScriptのサンプルコードを書いて、ブラウザでJavaScriptを実行してみましょう。

### サンプルコード

```
function hello(name) {
  console.log("こんにちは" + name + "さん");
}

hello("名前");
```

### エディタ

先ほどのサンプルコードを、次のエディタに書き写して"実行"ボタンを押して、JavaScriptを実行みてください。

"と"で囲まれた名前という文字列は、自分の名前に書き換えてみてください。

```
// この行を消して、先ほどのコードを書き写して"実行"を押してみよう
```

実行した結果として、次のようなメッセージが表示されているなら実行成功です。

こんにちは名前さん

"終了"を押してエディタを終了できます。

"実行"に失敗した場合は、次の注意事項を読んでもう一度挑戦してみてください。

### 注意事項

"実行"したときに、syntaxErrorというようなエラーが表示されたなら、コードのどこかを書き間違えています。次の点に気をつけて成功するまで試してみて下さい。

- コードは日本語入力をオフにして入力します
  - "と"で囲まれた部分以外は、すべて半角の英数字で入力してください
  - "(ダブルクオート)、; (セミコロン)、括弧、スペースなどの記号も半角になっているかを確認してください
- 1行入力ごとにEnterキーを押すことで改行できます

- 大文字と小文字は区別されます
  - このサンプルコードでは小文字の英数字しか利用していません
- コードの意味はまだ分からなくても問題ありません
  - 正しくコードを入力して、実行できることが重要です

*SyntaxError: illegal character*や*SyntaxError: Invalid or unexpected token*のようなエラーメッセージがでている場合は、どこかに全角の記号や全角の英数字が混ざっています。

一度すべてのコードを消してから、もう一度入力して見ましょう。

## コードを実行できない

どうしても実行できない場合は、まだこの書籍を読むには少し早いかもしれません。また、書籍は文字を中心とした解説になるため、初めてのプログラミングではイメージが難しいこともあります。そのため、もっと情報量の多い動画をなど見て、まずはコードを実行できるようにすることから始めるのがよいかかもしれません。

"JavaScript 入門 動画" といったキーワードで検索し、映像で学んで見てからでも遅くはありません。

また、もっとも効率的なのは、身近な知り合いに聞くことです。一人で学ぼうとすると、思わぬところで詰まってしまい進めなくなる場合があります。特に新しいことは最初の出発点が一番むずかしいです。そのため、他の人に手伝ってもらうことで意外と簡単に進めるようになります。

## ようこそJavaScriptへ

サンプルコードが実行できたなら、おそらくこの書籍は読めるはずです。

この書籍は、JavaScriptというプログラミング言語の基本的な文法から、実際のウェブアプリケーションで使われるようなパターンについてを学んでいく書籍です。JavaScriptという言語は、常に変化している言語としても知られています。その意味は、古いコードが動かなくなるのではなく、新しいやり方が常に増えていくという意味での変化です。

この書籍では、そのような変化していくJavaScriptの学び方についても学んでいきます。

# 本書の目的

本書の目的と読者対象を簡単にまとめたものです。

この文章は、まだ不完全な状態であるため、次のIssueを参照してください。

See [はじめに/本書の目的 · Issue #103 · asciidwango/js-primer](#)

- 本書がやらないこと
  - すべての文法や機能を網羅するのが目的ではない
  - ECMAScriptの仕様を厳密に学ぶことは目的ではない
  - DOMについて学ぶのが目的ではない
  - 他の言語と比較するのが目的ではない
  - Node.jsの使い方をマスターするのが目的ではない
  - JavaScriptのリファレンスを目指すものではない
    - 詳しくはMDNを参照すればよい
  - JavaScriptのライブラリを書くのが目的ではない
  - JavaScriptのライブラリの使い方を学ぶのが目的ではない
  - これを読んだから何か作れるというゴールがある訳ではない
- 本書の目的
  - 文法とともに実際にどのようなケースで使われてるのかを知ること
  - 必要なものを必要なだけ学びJavaScriptを読み書きできるようになることが目的
  - JavaScriptは変化を取り入れている言語であるため、JavaScriptの変化に対して対応できる基礎をつけていく
    - 過去にGood Partsと呼ばれていたものが良くないものとなっていることがある
  - 何か問題があるときに、その解決方法を自分で調べることができるようにすること
- 本書の読者対象
  - 完全なプログラミング初心者は対象ではない
  - JavaScript以外の言語をやったことがある人
  - JavaScriptを触ったことがある人
  - 古いJavaScriptを知っているが、今のJavaScriptはよくわからない人

## 文章の間違いに気づいたら

全くバグがないプログラムはないと同様に、全く間違いのない技術書は存在しません。この書籍ではできるだけ間違い（特に技術的な間違い）を減らせるように努力していますが、どうしても誤字脱字や技術的な間違い、コード例の間違いなどを見落としている場合があります。

そのため「この書籍には間違いが存在する」と思って読んでいくことを推奨しています。もし、読んでいて間違いを見つけたなら、ぜひ報告してください。

また、意味や意図が読み取れないというわかりにくいため、よくわからないという疑問をもつことがあります。そのような疑問もぜひ報告してください。

もし、その疑問が実際には間違いではなく勘違いであっても、回答をもらうことで自分の理解を修正できます。そのため、疑問を問い合わせても損することはないはずです。

この書籍はGitHub上で公開されているため、GitHubリポジトリのIssueとしてあなたの疑問を報告できます。

- 書籍や内容に対する質問 => [こちらから質問できます](#)
- 内容のエラーや問題の報告 => [こちらからバグ報告できます](#)
- 内容をもっと詳細に解説する提案 => [こちらから提案できます](#)
- 新しいトピックなどの提案 => [こちらから提案できます](#)
- その他のIssue => [その他のIssueはこちらから](#)

GitHubのアカウントを持っていない方は、次のフォームから報告できます。

<https://goo.gl/forms/lOx4ckFyb0fB9cBM2>

## 問題を修正する

また、この書籍はGitHub上で文章やサンプルのソースコードがすべて公開されています。

- <https://github.com/asciidwango/js-primer>

そのため、あなたは問題を報告するだけではなく修正することもできます。GitHubでソースコードが公開されているため、修正内容をPull Requestすることで問題を修正できます。

詳しいPull Requestの送り方はCONTRIBUTING.mdにかかれているので参考にしてください。

- <https://github.com/asciidwango/js-primer/blob/master/CONTRIBUTING.md>

誤字を1文字修正するものから技術的な間違いを修正するものまで、どのような修正であっても感謝されます。問題を見つけたら、ぜひ修正することにも挑戦してみてください。

## 参考

- 専門書には間違いもある：柴田 芳樹 (Yoshiki Shibata) : So-netブログ
- 技術書の間違いに気付いたら：柴田 芳樹 (Yoshiki Shibata) : So-netブログ

## 第一部: 基本文法

JavaScriptの基本文法について解説します。

## JavaScriptとは

JavaScriptはウェブブラウザ、Node.jsを始め、今はIoT（Internet of Things）デバイスなど幅広い環境で動作する言語となっています。

しかし、すべての環境で全く同じコードが動くわけではありません。その理由のひとつとして、JavaScriptという仕様があるわけではなく、JavaScriptという実装があるだけだからです。

JavaScriptという言語は、Ecma Internationalによって標準化されたECMAScriptという仕様の実装になります。そのため、JavaScriptが使える環境でも利用できる機能が異なります。

逆に、ECMAScriptという仕様で定義されている機能は、基本的にどの実行環境でも同じ動作をします。

□ JavaScriptとECMAScriptについて明確な区別を付けずに記述しているという点について書く

この章では、そのような実行環境に依存しないJavaScriptの機能について学んでいきます。

## JavaScriptってどのような言語？

JavaScriptは、C、Java、Self、Schemeなどの言語の影響を受けています。

また、JavaScriptはクラスベースではなくプロトタイプベースの言語です。ECMAScript 2015から class 構文が追加されました。このクラスもプロトタイプベースの上で実現されています。

大部分がオブジェクトであり、そのオブジェクト同士のコミュニケーションによって成り立っています。オブジェクトには、ECMAScriptの仕様として定められたビルトインオブジェクトと、実行環境が定義したオブジェクトとユーザの定義したオブジェクトが存在します。

実行環境が定義しているオブジェクトとしては、ブラウザがもつDOM（Document Object Model）APIやNode.jsがもつコアAPIなどがあります。

第一部の基本文法ではECMAScriptの定義する構文やビルトインオブジェクトを扱います。第二部のユースケースではブラウザが定義するDOM APIやNode.jsのコアAPIを扱い小さなアプリケーションを作成します。

JavaScriptの言語的な特徴を簡単に紹介すると、JavaScriptは大文字小文字を区別します。

たとえば、次のように name という変数を大文字と小文字で書いた場合に、それぞれは別々の名前の変数として認識されます。

```
// `name` という名前の変数を宣言
const name = "azu";
// `NAME` という名前の変数を宣言
const NAME = "azu";
```

また、クラスは大文字で開始しなければならないといった命名規則が意味をもつケースはありません。そのため、あくまで別々の名前として認識されるというだけになっています。

また、JavaScriptには特別な意味をもつキーワード（または予約語）が存在しています。キーワードと同じ名前の変数や関数は宣言できません。先ほどの、変数を宣言する const もキーワードのひとつとなっているため、const という変数名は宣言できません。

JavaScriptは、文（Statement）ごとに処理していき、文はセミコロン（;）によって区切られます。特殊なルールにもとづき、セミコロンがない文も行末に自動でセミコロンが挿入されるという仕組みも持っています。<sup>1</sup>しかし、暗黙的なものへ頼ると意図しない挙動が発生するため、セミコロンは常に書くようにします。

また、スペース、タブ文字などは空白文字（ホワイトスペース）と呼ばれます。空白文字をいくつ文の中に置いても挙動に違いはありません。

JavaScriptの実行コンテキストとして"Script"と"Module"があります。この2つの実行コンテキストの違いは意識しなくとも問題ありません。

"Module"はJavaScriptをモジュールとして実行するために、ECMAScript 2015で導入されたものです。"Module"の実行コンテキストでは古く安全でない構文や機能は一部禁止されているものがあります。

最後に、JavaScriptにはstrict modeという実行モードが存在しています。名前のとおり厳格な実行モードで、古く安全でない構文や機能が一部禁止されています。"Module"の実行コンテキストでは、このstrict modeがデフォルトとなっています。

"use strict"という文字列をファイルまたは関数の先頭に書くことで、そのスコープにあるコードはstrict modeで実行されます。

```
"use strict";
// このコードはstrict modeで実行される
```

strict modeでは、`eval` や `with` といったレガシーな機能や構文を禁止します。また、あきらかな問題を含んだコードに対しては早期的に例外を投げることで、開発者が間違いに気づきやすくしてくれます。たとえば、次のような `var` などのキーワードを含まずに変数を宣言しようとした場合に、strict modeでは例外が発生します。（strict modeでない場合は、例外が発生せずにグローバル変数が作られていました。）

```
"use strict";
mistypedVariable = 42; // => ReferenceError
```

このように、strict modeでは開発者が安全にコードを書けるように、JavaScriptの落とし穴を一部ふさいでくれます。そのため、常にstrict modeで実行できるコードを書くことがより安全なコードにつながります。

本書では、明示的に「strict modeではない」ことを宣言した場合を除き、すべてstrict modeとして実行できるコードを扱います。

<sup>1</sup>. Automatic Semicolon Insertionと呼ばれる仕組みです。 ↪

## コメント

コメントはプログラムとして評価されないため、ソースコードの説明を書くために利用されています。

JavaScriptには大きく分けて2種類のコメントがあります。

### 一行コメント

一行コメントは名前のとおり、一行づつコメントを書く際に利用します。 `//` 以降がコメントとして扱われるため、プログラムとして評価されません。

```
// 一行コメント
// この部分は評価されない
```

### 複数行コメント

複数行コメントは、複数行のコメントやJSDocのようなツールのためのアノテーションに利用します。 `/*` と `*/` で囲まれた範囲がコメントとして扱われるため、プログラムとして評価されません。

```
/* 複数行コメント
   囲まれている範囲が評価されない
*/
```

複数行コメントをネストすることはできないため、次のようなケースは `SyntaxError` となります。

```
/* ネストされた /* 複数行コメント */ SyntaxError */
```

### [ES2015] HTML-likeコメント

ES2015から後方互換性のための仕様としてHTML-likeコメントが追加されています。これはブラウザの実装に合わせた後方互換性のための仕様として定義されています。

基本的に既存のウェブサイトが壊れるような変更をECMAScript仕様には入れることができないためです。

HTML-likeコメントは名前のとおり、HTMLのコメントと同じ表記です。

```
<!-- この行はコメントと認識される
console.log("この行はJavaScript");
--> この行もコメントと認識される
```

ここでは、`<!--` と `-->` がそれぞれ一行コメントとして認識されます。

JavaScriptをサポートしていないブラウザでは、`<script>` タグを正しく認識できいために書かれたコードが表示されていました。それを避けるために `<script>` の中をコメントアウトし、表示はされないが実行されるという回避策が取られていました。今は `<script>` タグをサポートしていないブラウザはないため、この回避策は不要です。

```
<script language="javascript">
<!--
  document.bgColor = "brown";
-->
</script>
```

一方、`<script>` タグ内、つまりJavaScript内にHTMLコメントが書かれているサイトは残っているため、後方互換性のための仕様として追加されています。

歴史的経緯は別として、ECMAScriptではこのように後方互換性が慎重に取り扱われます。ECMAScriptは一度入った仕様が使えなくなることは殆どないため、基本文法で覚えたことが使えなくなることはありません。一方、新しく入った仕様でよりよい機能が増え、それを学び続けることには変わりありません。

# 変数と宣言

プログラミング言語には、数値などのデータに名前を付けたり繰り返し利用するために、データを保持するための変数があります。

JavaScriptでは、「これは変数です」という宣言をするキーワードとして、`var`、`let`、`const`があります。

この章では、変数の宣言方法を見ていきます。

## `var`

`var` キーワードを使い変数宣言ができます。

たとえば、次のコードでは、`bookTitle` という変数を宣言しています。この場合、`bookTitle` は値が代入されていないため、デフォルト値として `undefined` で初期化されます。（`undefined` は値が代入されてないことを表す値です）

```
var bookTitle;
```

この `bookTitle` という変数には、`=` 演算子を使うことで値を代入できます。

```
var bookTitle;
bookTitle = "JavaScriptの本";
```

変数宣言と同時に初期値を代入することもできます。次の例では、`bookTitle` という変数を宣言し、同時に "JavaScriptの本" で初期化しています。

```
var bookTitle = "JavaScriptの本";
```

また、変数宣言は`,`で区切ることにより、同時に複数の変数を定義できます。次のコードでは、`bookTitle` と `bookCategory` の変数を順番に定義しています。

```
var bookTitle = "JavaScriptの本",
bookCategory = "プログラミング";
```

これは次のように書いた場合と同じ意味になります。

```
var bookTitle = "JavaScriptの本";
var bookCategory = "プログラミング";
```

## 変数名に利用できる文字種

JavaScriptでは変数名として使える識別子には次のルールがあります。

- 先頭の1文字目は`$`、`_`、アルファベット、`\uxxxx` 形式のUnicodeエスケープシーケンスのどれか
  - アルファベットは"A"から"Z"（大文字）と"a"から"z"（小文字）
- 2文字目以降は、上記に加えて、数字、一部Unicode文字、U+200C、U+200Dのどれか

このルールの例外として、予約語として定義されているキーワードは変数名には利用できません。[JavaScript variable name validator](#)でどのような変数が利用可能かをチェックできます。

```
var $; // OK: $から開始できる
```

```
var _title; // OK: _から開始できる
var jquery; // OK: アルファベット
var es2015; // OK: 数字は先頭以外なら利用できる
var valid日本語; // OK: 先頭以外なら一部Unicode文字も利用可能
```

次のような変数は上記のルールに反するため、構文エラー（`SyntaxError`）となります。

```
var 2nd; // NG: 数字から始まっている
var var; // NG: `var`は予約語であるため利用できない
```

先ほど紹介したように変数の宣言に利用できるキーワードとして、`var` 以外にも `let` と `const` があります。どちらも変数宣言の一種で、利用できる変数名のルールは同じとなります。

それでは、次は `let` と `const` について見ていきます。

## [ES2015] let

`let` キーワードを使い、現在のスコープに対して変数宣言できます。

`let` の使い方は `var` とほとんど同じです。次のコードでは、`bookTitle` という変数を宣言し、同時に "JavaScriptの本" で初期化しています。

```
let bookTitle = "JavaScriptの本";
```

`let` と `var` は、スコープの扱いと同じ変数名の再定義の扱いが異なります。スコープについて「[関数とスコープ](#)」の章で扱うため、現時点では「よりよい `var`」ということだけ覚えておくとよいです。

`let` と `const` は同一スコープ内で同じ変数名を再定義できません。次のように同じ変数名で再定義しようとする構文エラー（`SyntaxError`）になります。これにより、間違えて変数を二重に定義してしまうというミスを防ぐことができます。

```
let x; // "x"を定義する
let x; // 同じ"x"を定義するとSyntaxErrorとなる
```

一方、`var` は同一スコープ内で同じ変数名を再定義できます。これは意図せずに同じ変数名を定義し値を上書きしてしまう問題があるため、`var` を避ける理由の 1 つとなります。

```
var x; // "x"を定義する
var x; // 同じ変数名"x"を定義できる
```

## [ES2015] const

最後に `const` ですが、`let` に対してさらに制約をつけた変数宣言という位置づけになります。基本的な使い方は `let` と同じですが、`const` は再代入できない変数を定義するキーワードです。

`var` や `let` では、変数宣言と代入を別々に行うことができました。

```
// varやletで宣言した変数には代入できる
let bookTitle; // `undefined`で初期化される
bookTitle = "JavaScriptの本"; // 値を代入している
```

しかし、`const` での宣言と代入を別々に行うコードは構文エラー（`SyntaxError`）となります。

```
const bookTitle; // SyntaxError: missing = in const declaration  
bookTitle = "JavaScriptの本";
```

`const` は必ず宣言時に値を指定しなければなりません。

```
const bookTitle = "JavaScriptの本";
```

そして、一度 `const` で宣言された変数には再代入できなくなります。そのため、次のコードでは `bookTitle` を上書きしようとして `TypeError` となります。

```
const bookTitle = "JavaScriptの本";  
bookTitle = "上書き"; // TypeError: invalid assignment to const `bookTitle`
```

一般的に変数への再代入は「変数の値は最初に定義した値と常に同じである」という参照透過性と呼ばれるルールを壊すため、バグを発生させやすい要因として知られています。

変数を再代入をしたいケースとしてループ中に値の変化させたい場合などがあります。しかし、多くのケースで代替できる表現があるため必ずしも `var` や `let` を使わなくても実現できます。`const` を使うことでバグに気づきやすくなるため、`const` を積極的に利用していくことを推奨しています。

## まとめ

JavaScriptにおける変数宣言として `var`、`let`、`const` があることについて学びました。

`var` はもっとも基礎的な変数宣言方法です。`let` と `const` は `var` の問題を改善するためにES2015で導入されました。

`var` は殆どのケースで `let` や `const` に置き換えが可能です。`const` は再代入できない変数を定義するキーワードです。再代入を禁止することで、ミスから発生するバグを減らすことが期待できます。

そのため `const` で変数を定義できないかを検討してから、できない場合は `let` を使うことを推奨しています。

## 値の評価と表示

変数宣言を使うことで値に名前をつける方法を学びました。次はその値をどのように評価するかについてです。

値の評価とは、次のような入力を評価して結果を返すことを示しています。

- `1 + 1` という式を評価したら `2` という結果を返す
- `bookTitle` という変数を評価したら、変数に代入されている値を返す

この値の評価方法を見ていくため、実行環境でJavaScriptを実行する方法を見ていきます。

## ブラウザでJavaScriptを実行する

まずはブラウザ上でJavaScriptのコードを実行してみましょう。この書籍ではブラウザとしてFirefoxを利用します。次のURLからFirefoxをダウンロードし、インストールしてください。

- Firefox: <https://www.mozilla.org/ja/firefox/>

ブラウザでJavaScriptを実行する方法としては大きく分けて2つあります。1つ目はブラウザの開発者ツールのコンソール上でJavaScriptコードを評価する方法です。2つ目はHTMLファイルを作成しJavaScriptコードを読み込む方法です。

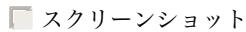
### ブラウザの開発者ツールのコンソール上でJavaScriptコードを評価する方法

ブラウザやNode.jsなど多くの実行環境には、コードを評価してその結果を表示するREPL（read–eval–print loop）と呼ばれる開発者向けの機能があります。Firefoxでは開発者ツールのWebコンソールと呼ばれる機能にREPL機能が含まれています。

Firefoxの開発者ツールは次のいずれかの方法で開きます。

- Firefox メニュー（メニューバーがある場合やmacOSでは、ツールメニュー）のWeb開発サブメニューで"Webコンソール"を選択する
- キーボードショートカットCtrl+Shift+K（macOSではCommand+Option+K）を押下する

詳細は"Webコンソールを開く"を参照してください。



"コンソール"を選択すると、コマンドライン（二重山かっこ»から始まる欄）に任意のコードを入力し評価できます。このコマンドラインがブラウザにおけるREPL機能です。

REPLに`1`という値を入力すると、その評価結果である`1`が次の行に表示されます。

```
>> 1
1
```

`1 + 1`という式を入力すると、その評価結果である`2`が次の行に表示されます。

```
>> 1 + 1
2
```

次に`const`キーワードを使って`bookTitle`という変数を宣言してみると、`undefined`という結果が次の行に表示されます。変数宣言は変数名と値を関連づけるだけであるため、変数宣言自体は何も値を返さないという意味で`undefined`が結果になります。REPLではそのまま次の入力ができるため、`bookTitle`という入力をすると、先ほ

ど変数に入れた "JavaScriptの本" という結果が次の行に表示されます。

```
> const bookTitle = "JavaScriptの本";
undefined
> bookTitle
"JavaScriptの本"
```

このようにコマンドラインのREPL機能では、JavaScriptのコードを1行ごとに実行できます。Shift + Enterで改行して複数行の入力もできます。好きな単位でJavaScriptのコードを評価できるため、コードの動きを簡単に試したい場合などに利用できます。

注意点としては、REPLではそのREPLを終了するまで `const` キーワードなどで宣言した変数が残り続けます。たとえば、`const` での変数宣言は同じ変数名を二度定義できないというルールでした。そのため1行づつ実行しても同じ変数名の定義を行うと構文エラー（`SyntaxError`）となります。

```
> const bookTitle = "JavaScriptの本";
undefined
> const bookTitle = "JavaScriptの本";
SyntaxError: redeclaration of const bookTitle
```

ブラウザでは、ページをリロードするとREPLの実行状態もリセットできます。`redeclaration`（再定義）に関するエラーメッセージが出た際にはページをリロードしてみてください。

## HTMLファイルを作成しJavaScriptコードを読み込む方法

REPLはあくまで開発者向けの機能であるため、ウェブサイトではJavaScriptはHTMLからスクリプトとして読み込み実行します。ここでは、HTMLとJavaScriptファイルを使ったJavaScriptコードの実行方法を見ていきます。

HTMLファイルとJavaScriptファイルの2種類を使い、JavaScriptのコードを実行する準備をしていきます。ファイルを作成するためAtomやVisual Studio CodeなどのJavaScriptなどに対応したエディタを用意しておくとスムーズです。エディタはどんなものでも問題ありませんが、必ずUTF-8の文字コードでファイルを保存してください。

ファイルを作成するディレクトリはどんな場所でも問題ありませんが、ここでは `example` という名前のディレクトリにファイルを作成していきます。

まずはJavaScriptファイルとして `index.js` ファイルを `example/index.js` というパスに作成します。`index.js` の中に次のようなコードを書いておきます。

```
1;
```

次にHTMLファイルとして `index.html` ファイルを `example/index.html` というパスに作成します。このHTMLファイルから先ほど作成した `index.js` ファイルを読み込み実行します。`index.html` の中には次のようなコードを書いておきます。

`index.html`

```
<html>
<head>
  <meta charset="UTF-8">
  <title>Example</title>
  <script src="./index.js"></script>
</head>
<body></body>
</html>
```

重要なのは `<script src=".index.js"></script>` という記述です。これは同じディレクトリにある `index.js` という名前のJavaScriptファイルをスクリプトとして読み込むという意味になります。

最後にブラウザで作成した `index.html` を開きます。HTMLファイルを開くには、ブラウザにHTMLファイルをドロップアンドドロップまたはファイルメニューから"ファイルを開く"でHTMLファイルを選択します。HTMLファイルを開いた際に、ブラウザのアドレスバーには `file:///` から始まるローカルファイルのファイルパスが表示されます。

先ほどと同じ手順で"Web コンソール"を開いてみると、コンソールには何も表示されていないはずです。REPLでは自動で評価結果のコンソール表示まで行いますが、JavaScriptコードとして読み込んだ場合は勝手に評価結果を表示されることはありません。あくまで自動表示はREPLの機能です。そのため多くの実行環境ではコンソール表示するためのAPI（機能）が存在しています。

## Console API

JavaScriptの多くの実行環境では、Console APIがコンソール表示を行うAPIとなっています。`console.log(引数)` の引数にコンソール表示したい値を入れることで、評価結果がコンソールに表示されます。

先ほどの `index.js` の中身を次のように書きかえます。そしてページをリロードする `1` という値を評価した結果がWebコンソールに表示されます。

```
console.log(1); // => 1
```

次のように引数に式を書いた場合は先に引数（（と）の間に書かれたもの）の式を評価してから、その結果をコンソールに表示します。そのため、`1 + 1` の評価結果として `2` がコンソールに表示されます。

```
console.log(1 + 1); // => 2
```

同じように引数に変数を渡すこともできます。この場合もまず先に引数である変数を評価してから、その結果をコンソールに表示します。

```
const total = 42 + 42;
console.log(total); // => 84
```

Console APIは原始的なプリントデバッグとして利用できます。「この値は何だろう」と思ったらコンソールに表示すると解決する問題は多いです。またJavaScriptの開発環境は高機能化が進んでいるため、Console API以外にもさまざまな機能がありますがここでは詳細は省きます。

この書籍では、コード内で評価結果を表示するためにConsole APIを利用していきます。

すでに何度も登場していますが、コード内のコメントで `// => 評価結果` と書いている場合があります。このコメントは、その左辺にある値を評価した結果またはConsole APIで表示した結果を掲載しています。

```
// 式の評価結果の例（コンソールには表示されない）
1; // => 1
// 変数の評価結果の例（コンソールには表示されない）
const total = 42 + 42;
// totalの評価結果は84
total; // => 84
// Console APIでコンソールに表示する例
console.log("JavaScript"); // => "JavaScript"
```

## ウェブ版の書籍でコードを実行する

ウェブ版の書籍では実行できるサンプルコードには実行というボタンが配置されています。このボタンでは実行するたびに毎回新しい環境を作成して実行するため、REPLで発生する変数の再定義といった問題はおきません。

一方で、REPLと同じように`_1`というコードを実行すると`_1`という評価結果を得られます。またConsole APIにも対応しています。サンプルコードを改変して実行するなどよりコードへの理解を深めるために利用できます。

```
console.log("Console APIで表示");
// 値を評価した場合は最後の結果が表示される
42; // => 42
```

## コードの評価とエラー

JavaScriptのコードを実行したときにエラーメッセージが表示されて意図したように動かなかった場合もあるはずです。プログラムを書くときに一度もエラーを出さずに書き終えることは殆どありません。特に新しいプログラミング言語を学ぶ際にはトライアンドエラー（試行錯誤）することはとても重要です。

エラーメッセージがWebコンソールに表示された際にはあわてずにそのエラーメッセージを読むことで多くの問題は解決できます。またエラーには多く分けて構文エラーと実行時エラーの2種類があります。ここではエラーメッセージの簡単な読み方を知り、そのエラーを修正する足がかりを見ていきます。

### 構文エラー

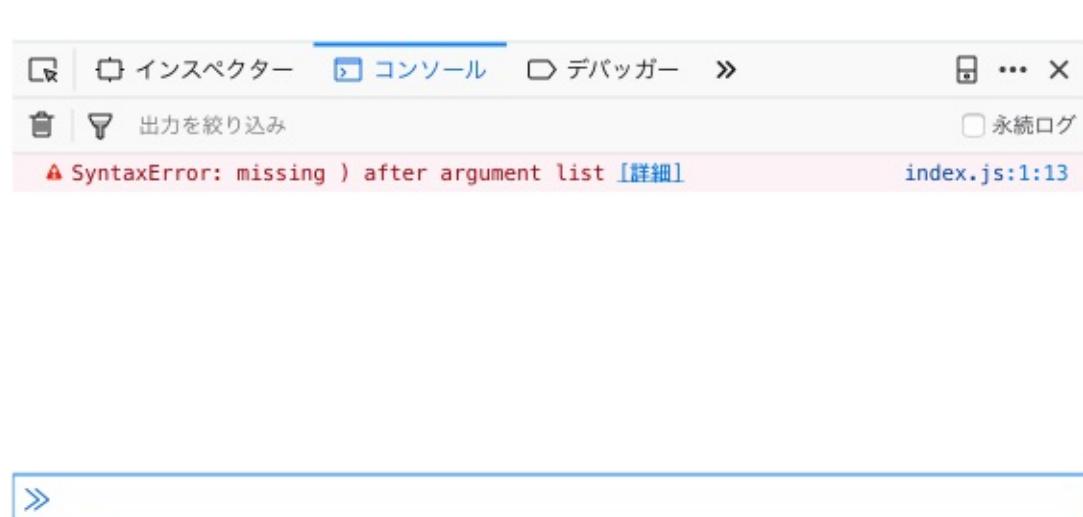
構文エラーは書かれたコードの文法が間違っている場合に発生するエラーです。

JavaScriptエンジンは、コードをパース（解釈）してから、プログラムとして実行できる形に変換して実行します。コードをパースする際に文法の問題が見つかると、その時点で構文エラーが発生するためプログラムとして実行できません。

次のコードでは、関数呼び出しに`)`をつけ忘れているため構文エラーが発生します。

index.js

```
console.log(1; // => SyntaxError: missing ) after argument list
```



Firefoxでこのコードを実行すると次のようなエラーメッセージがコンソールへ表示されます。

```
SyntaxError: missing ) after argument list[詳細] index.js:1:13
```

エラーメッセージはブラウザによって多少の違いはありますが、基本的には同じ形式のメッセージになります。このエラーメッセージをパーツごとに見てみると次のようにになります。

```
SyntaxError: missing ) after argument list[詳細] index.js:1:13
^^^^^^^^^ ^^^^^^^^^^ ^^^^^^ ^^^^^^
|          | 行番号:列番号
エラーの種類      エラー内容の説明           ファイル名
```

メッセージ	意味
SyntaxError: missing ) after argument list	エラーの種類は SyntaxError で、関数呼び出しの ) が足りないこと
index.js:1:13	例外が index.js の1行目13列目で発生したこと

Firefoxは[詳細]というリンクがエラーメッセージによっては表示されます。この[詳細]リンクはエラーメッセージに関するMDNの解説ページへのリンクとなっています。この例のエラーメッセージでは次の解説ページへリンクされています。

- [https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Errors/Missing\\_parenthesis\\_after\\_argument\\_list](https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Errors/Missing_parenthesis_after_argument_list)

このエラーメッセージや解説ページから、関数呼び出しの ) が足りないため構文エラーとなっていることがわかります。そのため、次のように足りない ) を追加することでエラーを修正できます。

```
console.log(1);
```

構文エラーによっては少しエラーメッセージから意味が読み取りにくいものもあります。

次のコードでは、 const を cosnt とタイプミスしているため構文エラーが発生しています。

index.js

```
cosnt a = 1;
```

SyntaxError: unexpected token: identifier[詳細] index.js:1:6

メッセージ	意味
SyntaxError: unexpected token: identifier	エラーの種類は SyntaxError で、予期しないものが識別子（変数名）に指定されている
index.js:1:6	例外が index.js の1行目6列目で発生したこと

プログラムをパースする際に index.js:1:6 で予期しない（構文として解釈できない）識別子が見つかったため、構文エラーが発生したという意味になります。1行目6列目（行は1から、列は0からカウントする）である a という文字列がおかしいということになります。しかし、実際には cosnt というタイプミスがこの構文エラーの原因です。

なぜこのようなエラーメッセージになるかというと、 cosnt （ const のタイプミス）はキーワードではないため、ただの変数名として解釈されます。そのため、このコードは次のように解釈されそのような文法は認められないということで構文エラーとなっています。

```
変数名 変数名
```

このようにエラーメッセージとエラーの原因は必ずしも一致しません。しかし、構文エラーの原因はコードの書き間違いであることが殆どです。そのため、エラーが発生した位置やその周辺を注意深く見ることで、エラーの原因を特定できます。

## 実行時エラー

実行時エラーはプログラムを実行している最中に発生するエラーです。実行時（ランタイム）におけるエラーであるためランタイムエラーと呼ばれることもあります。APIに渡す値の問題から起きる `TypeError` や存在しない変数を参照しようとして起きる `ReferenceError` などさまざまな種類があります。

実行時エラーが発生した場合は、そのコードは構文としては正しい（構文エラーではない）ですが、別のことが原因でエラーが発生しています。

次のコードでは `a` という存在しない変数を参照したため `ReferenceError` になっています。

`index.js`

```
console.log(a); // => ReferenceError: a is not defined
```

ReferenceError: a is not defined[詳細] index.js:1:1

メッセージ	意味
<code>ReferenceError: a is not defined</code>	エラーの種類は <code>ReferenceError</code> で、 <code>a</code> という未定義の識別子を参照した
<code>index.js:1:1</code>	例外が <code>index.js</code> の1行目1列目で発生したこと

`a` という変数や関数が存在するかは実行してみないとわかりません。多くの実行環境がConsole APIのようなビルトインの機能を提供しているように、`a` という名前の機能を提供する実行環境も考えられます。そのため、実行して `a` という識別子を参照したときに初めて存在するか否かが判明して、存在しない場合は `ReferenceError` となります。

このように、実行時エラーは該当する箇所を実行するまでエラーになるかがわからない場合も多いのです。そのため、どこまではちゃんと実行できたか順番に追っていくような、エラーの原因を特定する作業が必要になっている場合があります。このようなエラーの原因を特定し、修正する作業のことをデバッグと呼びます。

実行時エラーは構文エラーに比べてエラーの種類も多く、その原因もプログラムの数だけあります。エラーの原因を見つけることが大変な場合も多いですが、JavaScriptはとても良く使われている言語なので、ウェブ上には類似するエラーを報告している人も多いです。そのため、エラーメッセージで検索をしてみると、類似するエラーの原因と解消方法が見つかるケースもあります。

## まとめ

ブラウザ上でJavaScriptを実行する方法として開発者ツールを使う方法とHTMLからJavaScriptファイルを読み込む方法を紹介しました。「第1部 基本文法」で紹介するサンプルコードは基本的にこれらの方法で実行できます。サンプルコードを自分なりに改変して実行するなどするとより理解が深くなるため、サンプルコードの動作を自分自身で確認してみてください。

コードを実行してエラーが発生した場合にはエラーメッセージや位置情報などが表示されます。これらのエラー情報を使ってデバッグすることでエラーの原因を取り除けるはずです。

JavaScriptにおいては多くのエラーはすでに類似するケースがウェブ上に報告されています。構文エラーや実行時エラーの典型的なものはMDNの[JavaScript エラーリファレンス](#)にまとめられています。また[Google](#)、[GitHub](#)、[Stack Overflow](#)などでエラーメッセージを検索することで、エラーの原因を見つけられることもあります。

エラーがWebコンソールに表示されているならば、そのエラーは修正できます。エラーを過度に怖がる必要はありません。エラーメッセージなどのヒントを使ってエラーを修正していくようにしましょう。



# データ型とリテラル

## データ型

JavaScriptは動的型付け言語に分類される言語であるため、静的型付け言語のような変数の型はありません。

しかし、文字列、数値、真偽値といった値の型は存在します。これらの値の型のことをデータ型とよびます。

データ型を大きく分けると、プリミティブ型とオブジェクトの2つに分類されます。

プリミティブ型（基本型）は名前のとおり、文字列や数値などの基本的な値の型のことです。プリミティブ型の値は、一度作成したらその値自体を変更することはできないという immutable の特性を持ちます。

一方、プリミティブ型ではないものをオブジェクト（複合型）と呼び、オブジェクトは複数のプリミティブ型の値またはオブジェクトからなる集合です。オブジェクトは、一度作成した後もその値自体を変更できるため mutable の特性を持ちます。また、オブジェクトは値そのものではなく、値への参照を使い操作されるため参照型のデータともいえます。

データ型を細かく見ていくと、6つのプリミティブ型とオブジェクトからなります。

- プリミティブ型（基本型）
  - 真偽値（Boolean）: `true` または `false` のデータ型
  - 数値（Number）: `42` や `3.14159` などの数値のデータ型
  - 文字列（String）: `"JavaScript"` などの文字列のデータ型
  - `undefined`: 値が未定義であることを意味するデータ型
  - `null`: 値が存在しない null 値を意味するデータ型
  - シンボル（Symbol）: ES2015から追加された一意で不变な値のデータ型
- オブジェクト（複合型）
  - プリミティブ型以外のデータ
  - オブジェクト、配列、関数、正規表現、`Date`など

プリミティブ型でないものは、オブジェクトであるということを覚えていれば問題ありません。

`typeof` 演算子を使うことで、次のようにデータ型を調べることができます。

```
typeof true; // => "boolean"
typeof 42; // => "number"
typeof "JavaScript"; // => "string"
typeof Symbol("シンボル"); // => "symbol"
typeof undefined; // => "undefined"
typeof null; // => "object"
typeof [ "配列" ]; // => "object"
typeof { "key": "value" }; // => "object"
typeof function() {}; // => "function"
```

残念ながら `typeof null; // => "object"` となるのは歴史的経緯のある仕様バグ<sup>1</sup>です。他のプリミティブ型の値については、`typeof` 演算子でそれぞれのデータ型を調べることができます。

オブジェクトと一言にいっても JavaScriptではすべてがオブジェクトであると言われるほど、多くの種類が存在します。`typeof` 演算子ではすべてのオブジェクトの種類を判定することはできません。

つまり、`typeof` 演算子は、プリミティブ型またはオブジェクトかを判別するもので、オブジェクトの詳細なデータ型については別の方法を判定するようになっています。

詳しい判定方法については各オブジェクトの章を参照してください。

## リテラル

プリミティブ型の値やオブジェクトはリテラルを使うことでプログラムに表現できます。

TODO: リテラルとはプログラム上で数値や文字列など、直接記述した内容がそのデータ型の値を書ける記法を定義したものです。たとえば、" "と" "で囲んだ文字列と扱えるため、繰り返し扱うデータ型は簡単に書けるようになっています。リテラル表現がない場合は、その値を作る関数に引数を渡して作成する形になります。そのような冗長な表現を避ける方法として主要な値にはリテラルが用意されています。

次の3つのプリミティブ型はそれぞれリテラル表現を持っています。

- 真偽値
- 数値
- 文字列

### 真偽値 (Boolean)

真偽値は `true` と `false` のリテラルがあります。それぞれは `true` と `false` の値を返すリテラルとなります。

```
true; // => true
false; // => false
```

### 数値 (Number)

数値は大きく分けて `42` のような整数リテラルと `3.14159` のような浮動小数点リテラルがあります。

#### 整数リテラル

整数リテラルは次の4種類があります。

- 10進数: 先頭が `0` ではない数値 - `10`
- 2進数: `0b` または `0B` - `0b1`
  - `0b` の後ろには `0` または `1` の数値
- 8進数: `0o` または `0O` - `0o72`
  - `0o` の後ろには `0` から `7` までの数値
- 16進数: `0x` または `0X` - `0x15`
  - `0x` の後ろには `0` から `15` までの数値

JavaScriptでは、0から9の数字のみで書かれた数値は10進数として扱われます。

`0b` から始まる2進数リテラルは、ビットを表現するのによく利用されています。

```
0b1111; // => 15
```

`0o` から始まる8進数リテラルは、ファイルのパーミッションを表現するのによく利用されています。

```
0o777; // => 511
```

`0x` から始まる16進数リテラルは、文字のコードポイントやRGB値の表現などに利用されています。

```
0xFF; // => 255
```

TODO: もっと具体的なイメージの話をしたい。

	表記例	用途
10進数	42	数値
2進数	0b0001	ビット演算など
8進数	0o777	ファイルのパーミッションなど
16進数	0xEEFF	文字コード、RGB値など

## 浮動小数点数リテラル

JavaScriptの浮動小数点数は[IEEE 754](#)を採用しています。浮動小数点数をリテラルと書く場合には次の2種類の表記が利用できます。

- `3.14159` のような `.`（ドット）を含んだ数値
- `2e8` のような `e` または `E` を含んだ数値

`0`から始まる浮動小数点数は、`0`を省略して書くことができます。

```
.123; // => 0.123
```

しかし、JavaScriptでは`.`をオブジェクトにおいて利用する機会が多いため、`0`から始まる場合でも省略せずに書いたほうが意図しない挙動を減らせるでしょう。

Note 変数名が数字から始めることができないのは、数値リテラルと衝突してしまうことが理由としてあげられます。

## 文字列 (String)

文字列リテラル共通のルールとして、同じ記号で囲んだ範囲を文字列として扱います。文字列リテラルとして次の3種類のリテラルがありますが、すべて評価した結果は同じ"文字列"です。

```
"文字列";
'文字列';
`文字列`;
```

## ダブルクオートとシングルクオート

`"`（ダブルクオート）と`'`（シングルクオート）は全く同じ意味となります。PHPやRubyなどとは違い、どちらのリテラルでも評価結果は同じとなります。

文字列リテラルは同じ記号で囲む必要があるため、次のように文字列の中に同じ記号が出現した場合は、`\`という形で`\`を使いエスケープしなければなりません。

```
'8 o\'clock'; // => "8 o'clock"
```

そのため、文字列内部に出現しないリテラル記号を使うことで、エスケープをせずに書くことができます。

```
"8 o'clock"; // => "8 o'clock"
```

ダブルクオートとシングルクオートどちらも改行をそのまま入力することはできません。次のように改行を含んだ文字列は定義できずに構文エラー（`syntaxError`）となります。

```
"複数行の
文字列を
入れたい"; // Syntax Error
```

改行の代わりに改行記号のエスケープシーケンス（`\n`）を使うことで複数行の文字列を書くことができます。

```
"複数行の\n文字列を\n入れたい";
```

複数行の文字列は次のテンプレートリテラルを使うことでもっと直感的に書くことができます。

## [ES2015] テンプレートリテラル

テンプレートリテラルは ``` (バッククオート) で囲んだ範囲を文字列とするリテラルです。テンプレートリテラルでは、複数行の文字列を改行記号なしに書くことができます。

複数行の文字列も ``` で囲めば、そのまま書くことができます。

```
`複数行の
文字列を
入れたい`; // => "複数行の\n文字列を\n入れたい"
```

また、名前のとおりテンプレートのような機能を持っています。テンプレートリテラル内で ``${変数名}`` と書いた場合に、その変数の値を埋め込むことができます。

```
const string = "文字列";
console.log(`これは${string}です`); // => "これは文字列です"
```

テンプレートリテラルも他の文字列リテラルと同様に同じリテラル記号を内包したい場合は、`\`` を使いエスケープする必要があります。

```
`This is \`code\``; // => "This is `code`"
```

## nullリテラル

nullリテラルは `null` 値を返すリテラルです。`null` は「値がない」ということを表現する値です。

次のように、未定義の変数を参照した場合は、参照できないため `ReferenceError` の例外が投げられます。

```
foo; // "ReferenceError: foo is not defined"
```

`foo` は値がないということを表現したい場合は、`null` 値を代入することで、`null` 値をもつ `foo` という変数を定義できます。これにより、`foo` を値がない変数として定義し、参照できるようになります。

```
const foo = null;
console.log(foo); // => null
```

## オブジェクトリテラル

JavaScriptにおいてあらゆるものの中身となるのがオブジェクトです。そのオブジェクトを作成する方法のひとつとしてオブジェクトリテラルがあります。オブジェクトリテラルは `{}` (中括弧) を書くことで、新しいオブジェクトを作成できます。

```
const object = {} // 中身が空のオブジェクトを作成
```

オブジェクトリテラルはオブジェクトの作成と同時に中身を定義できます。オブジェクトのキーと値を `:` で区切ったものを `{}` の中に書くことで作成と初期化が同時に行えます。

次のコードで作成したオブジェクトは `key` というキー名と `value` という値をもつオブジェクトを作成しています。キー名には、文字列またはSymbolを指定し、値にはプリミティブ型の値からオブジェクトまで何でも入れることができます。

```
const object = {
  key: "value"
};
```

このとき、オブジェクトがもつキーのことをプロパティ名と呼びます。この場合、`object` は `key` というプロパティを持っていると言います。

`object` の `key` を参照するには、`.` (ドット) で繋ぎ参照する方法と、`[]` (ブラケット) で参照する方法があります。

```
const object = {
  "key": "value"
};
// ドット記法
console.log(object.key); // => "value"
// ブラケット記法
console.log(object["key"]); // => "value"
```

ドット記法では、プロパティ名が変数名と同じく識別子である必要があります。そのため、次のように識別子として利用できないプロパティ名はドット記法として書くことができません。

```
// プロパティ名は文字列の"123"
var object = {
  "123": "value"
};
// NG: ドット記法では、数値から始まる識別子は利用できない
object.123
// OK: ブラケット記法では、文字列として書くことができる
console.log(object["123"]); // => "value"
```

オブジェクトは多くの機能や仕組みを持つていますが、詳細については第n章で紹介します。そのため、オブジェクトリテラルが出てきたら新しいオブジェクトを作成しているんだなと見てください。

■ TODO: 第n章を埋める

## 配列リテラル

オブジェクトリテラルと並んでよく使われるリテラルとして配列リテラルがあります。配列リテラルは `[` と `]` で値をカンマ区切りで囲み、Arrayオブジェクトを作成します。配列（Arrayオブジェクト）とは、複数の値に順序をつけて格納できるオブジェクトの一種です。

```
const emptyArray = [] // 空の配列を作成
const array = [1, 2, 3]; // 値をもった配列を作成
```

作成した配列の要素を取得するには、配列に対して `array[index]` でアクセスできます。配列のインデックスは 0 から開始する数値となっています。

```
const array = ["index:0", "index:1", "index:2"];
console.log(array[0]); // => "index:0"
console.log(array[1]); // => "index:1"
```

配列についての詳細は第n章で紹介します。

■ TODO: 第n章を埋める

## 正規表現リテラル

JavaScriptは正規表現をリテラルで書くことができます。正規表現リテラルは / と / で正規表現のパターン文字列を囲みます。正規表現のパターン内では、+ や \ (バックスラッシュ) から始まる特殊文字が特別な意味を持ちます。

次のコードでは、数字にマッチする特殊文字である \d を使い、1文字以上の数字にマッチする正規表現をリテラルで表現しています。

```
const numberRegExp = /\d+/; // 1文字以上の数字にマッチする正規表現
// 123が正規表現にマッチするかをテストする
console.log(numberRegExp.test(123)); // => true
```

`RegExp` コンストラクタを使うことで文字列から正規表現オブジェクトを作成することができますが、特殊文字の二重エスケープが必要になり直感的に書くことが難しくなります。

正規表現オブジェクトについて詳しくは、[文字列の章](#)で紹介します。

## まとめ

この章では、データ型とリテラルについて学びました。

- 6つのプリミティブ型とオブジェクトがある
- 真偽値、数値、文字列についてリテラル表現がある
- オブジェクトではオブジェクトと配列リテラルがある

## [コラム] undefinedはリテラルではない

プリミティブ型として紹介した `undefined` はリテラルではありません。`undefined` はただのグローバル変数で、`undefined` という値を持っているだけです。

これを証明するために、`var` を使って `undefined` という名前のローカル変数を宣言してみます。次のようにstrict modeではない環境においては、`undefined` というローカル変数を `var` で定義できます。もちろん `let` や `const` では、同名の変数は再定義できないため `undefined` の再定義はできません。

```
// strict modeではない実行環境
function fn(){
    var undefined = "独自の未定義値"; // undefinedと名前の変数をエラーなく定義できる
    console.log(undefined); // => "独自の未定義値"
}
fn();
```

これに対して `null` はグローバル変数ではなくリテラルであるため、`var` を使っても再定義できません。リテラルは変数名として利用できない予約語であるため、このような違いが生じています。

```
var null; // => SyntaxError
```

このコラムでは、説明のために `undefined` というローカル変数を宣言しましたが、現実ではこのような使い方は非推奨です。無用な混乱を生むだけなので避けるべきです。またstrict modeの場合や、`const`、`let` を利用した場合はそもそも定義できません。

## 参考

- [11.6.2 Reserved Words](#)
- [11.8.3.1 Static Semantics: MV](#)
- [no-undefined - Rules - ESLint - Pluggable JavaScript linter](#)
  - `undefined` の宣言を禁止するESLintルール

1. JavaScriptが最初にNetscapeで実装された際に `typeof null === "object"` となるバグがありました。このバグを修正するとすでにこの挙動に依存しているコードは壊れるため、修正が見送られ現在の挙動が仕様となりました。<http://2ality.com/2013/10/typeof-null.html>を参照 ↪

2. `øo` は数字のゼロと小文字アルファベットの `o` ↪

# 演算子

演算子はよく利用する計算を関数やメソッドではなく、記号として表現したものです。たとえば、足し算を行う `+` も演算子の一種で、演算子には多くの種類があります。

演算子は演算する対象をもちます。この演算子の対象のことを被演算子（オペランド）と呼びます。

次のコードでは、`+` 演算子が値同士を足し算する加算演算を行っています。このとき、`+` 演算子の対象となる `1` と `2` という2つの値がオペランドです。

```
1 + 2;
```

このコードでは `+` 演算子に対して、前後に合計2つのオペランドがありました。このように2つのオペランドを取る演算子を二項演算子と呼びます。

```
// 二項演算子とオペランドの関係
オペランド1 演算子 オペランド2
```

また、1つの演算子に対して1つのオペランドだけをとるものもあります。たとえば、次のような数値をインクリメントする `++` 演算子は、前後どちらか一方にオペランドを置きます。

```
let num = 1;
num++;
// または
++num;
```

このように1つのオペランドを取る演算子を単項演算子と呼びます。単項演算子と二項演算子で同じ記号を使うことがあります、そのために呼び方を変えています。

この章では、演算子ごとにその演算子の処理について学んでいきます。また、演算子の中でも比較演算子は、JavaScriptでも特に挙動が理解にしにくい暗黙的な型変換という問題と密接な関係があります。そのため、演算子をひととおり見た後に、暗黙的な型変換と明示的な型変換について学んでいきます。

## 二項演算子

四則演算など基本的な二項演算子を見ていきます。

### プラス演算子（`+`）

2つの数値を加算する演算子です。

```
console.log(1 + 1); // => 2
```

JavaScriptでは、数値は内部的にIEEE 754方式の浮動小数点数として表現されています。（データ型とリテラルを参照）そのため、整数と浮動小数点数の加算もプラス演算子で行えます。

```
console.log(10 + 0.5); // => 10.5
```

### マイナス演算子（`-`）

2つの数値を減算する演算子です。

```
console.log(1 - 1); // => 0
console.log(10 - 0.5); // => 9.5
```

## 乗算演算子（\*）

2つの数値を乗算する演算子です。

```
console.log(2 * 8); // => 16
console.log(10 * 0.5); // => 5
```

## 除算演算子（/）

2つの数値を除算する演算子です。

```
console.log(8 / 2); // => 4
console.log(10 / 0.5); // => 20
```

## 剰余演算子（%）

2つの数値のあまりを求める演算子です。

```
console.log(8 % 2); // => 0
console.log(9 % 2); // => 1
console.log(10 % 0.5); // => 0
console.log(10 % 4.5); // => 1
```

## [ES2016]べき乗演算子（\*\*）

2つの数値のべき乗を求める演算子です。左オペランドを右オペランドでべき乗した値を返します。

```
// べき乗演算子（ES2016）で2の4乗を計算
console.log(2 ** 4); // => 16
```

べき乗演算子と同じ動作をする `Math.pow` メソッドがあります。

```
console.log(Math.pow(2, 4)); // => 16
```

べき乗演算子はES2016で後から追加された演算子であるため、関数と演算子それぞれ存在しています。他の二項演算子は演算子が先に存在していたため、`Math` には対応するメソッドがありません。

## 単項演算子（算術）

単項演算子は、1つのオペランド受け取り処理する演算子です。

### 単項プラス演算子（+）

単項演算子の`+`はオペランドを数値に変換します。

次のコードでは、数値の `1` を数値へ変換するため、結果は変わらず数値の `1` です。 `+数値` のように数値に対して、単行プラス演算子を付けるケースはほぼ無いでしょう。

```
console.log(+1); // => 1
```

また、単項プラス演算子は、数値以外も数値へと変換します。次のコードでは、数字（文字列）を数値へ変換しています。

```
console.log(+ "1"); // => 1
```

一方、数値に変換できない文字列などは `NaN` という特殊な値へと変換されます。

```
// 数値ではない文字列はNaNという値に変換される
console.log(+ "文字列"); // => NaN
```

`NaN` は"Not-a-Number"の略称で、数値ではないがNumber型の値を表現しています。`NaN` はどの値とも（`NaN`自身に対しても）一致しない特性があり、`Number.isNaN` メソッドを使うことで `NaN` の判定を行えます。

```
// 自分自身とも一致しない
console.log(NaN === NaN); // => false
// Number型である
console.log(typeof NaN); // => "number"
// Number.isNaNでNaNかどうかを判定
console.log(Number.isNaN(NaN)); // => true
```

しかし、単項プラス演算子は文字列から数値への変換に使うべきではありません。なぜなら、`Number` コンストラクタ関数や `parseInt` 関数などの明示的な変換方法が存在するためです。詳しくは[暗黙的な型変換](#)の章で解説します。

## 単項マイナス演算子（`-`）

単項マイナス演算子はマイナスの数値を記述する場合に利用します。

たとえば、マイナスの1という数値を `-1` と書くことができる原因是、単項マイナス演算子を利用しているからです。

```
console.log(-1); // => -1
```

また、単項マイナス演算子はマイナスの数値を反転できます。そのため、"マイナスのマイナスの数値"はプラスの数値となります。

```
console.log(-( -1 )); // => 1
```

単項マイナス演算子も文字列などを数値へ変換します。

```
console.log(- "1"); // => -1
```

また、数値へ変換できない文字列などをオペランドに指定した場合は、`NaN` という特殊な値になります。そのため、単項プラス演算子と同じく、文字列から数値への変換に単行マイナス演算子を使うべきではありません。

```
console.log(- "文字列"); // => NaN
```

## インクリメント演算子（`++`）

インクリメント演算子（`++`）は、オペランドの数値を `+1` する演算子です。オペランドの前後どちらかにインクリメント演算子をおくことで、オペランドに対して値を `+1` した値を返します。

```
let num = 1;
num++;
console.log(num); // => 2
// 次のようにした場合と結果は同じ
// num = num + 1;
```

インクリメント演算子（`++`）は、オペランドの後に置くか前に置くかで、それぞれで評価の順番が異なります。

後置インクリメント演算子（`num++`）は、次のような順で処理が行われます。

1. `num` の評価結果を返す
2. `num` に対して `+1` する

そのため、`num++` が返す値は `+1` する前の値となります。

```
let x = 1;
console.log(x++); // => 1
console.log(x); // => 2
```

一方、前置インクリメント演算子（`++num`）は、次のような順で処理が行われます。

1. `num` に対して `+1` する
2. `num` の評価結果を返す

そのため、`++num` が返す値は `+1` した後の値となります。

```
let x = 1;
console.log(++x); // => 2
console.log(x); // => 2
```

この2つの使い分けが必要となる場面は多くありません。そのため、評価の順番が異なることだけを覚えておけば問題ないといえます。

## デクリメント演算子（`--`）

デクリメント演算子（`--`）は、オペランドの数値を `-1` する演算子です。

```
let num = 1;
num--;
console.log(num); // => 0
// 次のようにした場合と結果は同じ
// num = num - 1;
```

デクリメント演算子は、インクリメント演算子と同様に、オペランドの前後のどちらかに置くことができます。デクリメント演算子も、前後どちらに置くかでの評価の順番が変わります。

```
// 後置デクリメント演算子
let x = 1;
console.log(--x); // => 1
console.log(x); // => 0
// 前置デクリメント演算子
let y = 1;
console.log(--y); // => 0
console.log(y); // => 0
```

## 比較演算子

比較演算子はオペラント同士の値を比較し、真偽値を返す演算子です。

### 厳密等価演算子（`==`）

厳密等価演算子は、左右の2つのオペラントを比較します。同じ型で同じ値である場合に、`true` を返します。

```
console.log(1 === 1); // => true
console.log(1 === "1"); // => false
```

また、オペラントがどちらもオブジェクトである時は、オブジェクトの参照が同じである場合に、`true` を返します。

次のコードでは、空のオブジェクトリテラル(`{}`)同士を比較しています。オブジェクトリテラルは、新しいオブジェクトを作成します。そのため、異なるオブジェクトを参照する変数を`==`で比較すると`false`を返します。

```
// {} は新しいオブジェクトを作成している
const objA = {};
const objB = {};
// 生成されたオブジェクトは異なる参照となる
console.log(objA === objB); // => false
// 同じ参照を比較している場合
console.log(objA === objA); // => true
```

### 厳密不等価演算子（`!=`）

厳密不等価演算子は、左右の2つのオペラントを比較します。異なる型または異なる値である場合に、`true` を返します。

```
console.log(1 !== 1); // => false
console.log(1 !== "1"); // => true
```

`==`を反転した結果を返す演算子となります。

### 等価演算子（`==`）

等価演算子（`==`）は、2つのオペラントを比較します。同じデータ型のオペラントを比較する場合は、厳密等価演算子（`==`）と同じ結果になります。

```
console.log(1 == 1); // => true
console.log("str" == "str"); // => true
console.log("JavaScript" == "ECMAScript"); // => false
// オブジェクトは参照が一致しているならtrueを返す
// {} は新しいオブジェクトを作成している
const objA = {};
const objB = {};
console.log(objA == objB); // => false
console.log(objA == objA); // => true
```

しかし、等価演算子（`==`）はオペラント同士が異なる型の値であった場合に、同じ型となるように暗黙的な型変換してから比較を行います。

そのため、次のような、見た目からは結果を予測できない挙動が多く存在します。

```
// 文字列を数値に変換してから比較
```

```

console.log(1 == "1"); // => true
// "01"を数値にすると`1`となる
console.log(1 == "01"); // => true
// 真偽値を数値に変換してから比較
console.log(0 == false); // => true
// nullの比較はfalseを返す
console.log(0 == null); // => false
// nullとundefinedの比較は常にtrueを返す
console.log(null == undefined); // => true

```

意図しない挙動となることがあるため、暗黙的な型変換が行われる等価演算子（`==`）を使うべきではありません。代わりに、厳密等価演算子（`===`）を使い、異なる型を比較したい場合は明示的に型を合わせるべきです。

例外的に、等価演算子（`==`）が使われるケースとして、`null` と `undefined` の比較があります。

次のように、比較したいオペランドが `null` または `undefined` であることを判定したい場合に、厳密等価演算子（`===`）では2度比較する必要があります。等価演算子（`==`）では `null` と `undefined` の比較結果は `true` となるため、一度の比較でよくなります。

```

const value = undefined; /* または null */
// === では2つの値と比較しないといけない
if (value === null || value === undefined) {
  console.log("valueがnullまたはundefinedである場合の処理");
}
// == では null と比較するだけでよい
if (value == null) {
  console.log("valueがnullまたはundefinedである場合の処理");
}

```

このように等価演算子（`==`）を使う例外的なケースはありますが、基本的に等価演算子（`==`）は暗黙的な型変換を行うためバグを引き起こしやすいです。そのため、仕組みを理解するまでは常に厳密等価演算子（`===`）を利用することを推奨します。

## 不等価演算子（`!=`）

不等価演算子（`!=`）は、2つのオペランドを比較し、等しくないなら `true` を返します。

```

console.log(1 != 1); // => false
console.log("str" != "str"); // => false
console.log("JavaScript" != "ECMAScript"); // => true
console.log(true != true); // => false
// オブジェクトは参照が一致していないならtrueを返す
const objA = {};
const objB = {};
console.log(objA != objB); // => true
console.log(objA != objA); // => false

```

不等価演算子も、等価演算子（`==`）と同様に異なる型のオペランドを比較する際に、暗黙的な型変換を行ってから比較します。

```

console.log(1 != "1"); // => false
console.log(0 != false); // => false
console.log(0 != null); // => true
console.log(null != undefined); // => false

```

そのため、不等価演算子（`!=`）は、利用するべきではありません。代わりに暗黙的な型変換を行わない厳密不等価演算子（`!==`）を利用します。

## 大なり演算子/より大きい ( > )

大なり演算子は、左オペランドが右オペランドより大きいならば、`true` を返します。

```
console.log(42 > 21); // => true
console.log(42 > 42); // => false
```

## 大なりイコール演算子/以上 ( >= )

大なりイコール演算子は、左オペランドが右オペランドより大きいまたは等しいならば、`true` を返します。

```
console.log(42 >= 21); // => true
console.log(42 >= 42); // => true
console.log(42 >= 43); // => false
```

## 小なり演算子/より小さい ( < )

小なり演算子は、左オペランドが右オペランドより小さいならば、`true` を返します。

```
console.log(21 < 42); // => true
console.log(42 < 42); // => false
```

## 小なりイコール演算子/以下 ( <= )

小なりイコール演算子は、左オペランドが右オペランドより小さいまたは等しいならば、`true` を返します。

```
console.log(21 <= 42); // => true
console.log(42 <= 42); // => true
console.log(43 <= 42); // => false
```

## ビット演算子

ビット演算子はオペランドを符号付き32bit整数に変換してから演算します。ビット演算子による演算結果は10進数の数値を返します。

たとえば、`9` という数値は符号付き32bit整数では次のように表現されます。

```
console.log(0b000000000000000000000000000000010001); // => 9
// Number#toStringメソッドを使うことで2進数表記の文字列を取得できる
console.log((9).toString(2)); // => "1001"
```

また、`-9` という数値は、ビッグエンディアンの2の補数形式で表現されるため、次のようにになります。

```
console.log(0b1111111111111111111111110111); // => 4294967287
// ゼロ桁埋め右シフトをしてからNumber#toStringで2進数表記を取得できる
console.log((-9 >>> 0).toString(2)); // => "1111111111111111111111110111"
```

## ビット論理積 ( & )

論理積演算子 (`&`) はビットごとのAND演算した結果を返します。

```
console.log(15 & 9); // => 9
```

```
console.log(0b1111 & 0b1001); // => 0b1001
```

## ビット論理和（|）

論理和演算子（|）はビットごとのOR演算した結果を返します。

```
console.log(15 | 9); // => 15
console.log(0b1111 | 0b1001); // => 0b1111
```

## ビット排他的論理和（^）

排他的論理和演算子（^）はビットごとのXOR演算した結果を返します。

```
console.log(15 ^ 9); // => 6
console.log(0b1111 ^ 0b1001); // => 0b0110
```

## ビット否定（~）

単項演算子の否定演算子（~）はオペランドを反転した値を返します。これは1の補数と知られている値と同じものです。

```
console.log(~15); // => -16
console.log(~0b1111); // => -0b10000
```

否定演算子（~）はビット演算以外でも使われています。

JavaScriptの `String#indexOf(string)` は、文字列中にある `string` の位置を見つけて返すメソッドです。この `indexOf` メソッドは、検索対象が見つからない場合に、`-1` を返します。

```
const string = "森森本森森";
// 見つかった場合はindex値を返す
console.log(string.indexOf("本")); // => 2
// 見つからぬ場合は-1を返す
console.log(string.indexOf("火")); // => -1
```

否定演算子（~）は1の補数を返すため、`~-1` の`0`となります。

```
console.log(~0); // => -1
console.log(~(-1)); // => 0
```

JavaScriptでは`0`もif文では`false`として扱われます。そのため、`~-indexOfの結果`が`0`となることを利用して書くイディオムが一部では使われています。

```
const string = "森森本森森";
if (string.indexOf("火") === -1) {
    // 見つからなかった場合の処理
}
// 否定演算子(`~`)で類似表現
if (~string.indexOf("火")) {
    // 見つからなかった場合の処理
}
```

このイディオムは文字列を検索した結果を真偽値で取得できれば不要となるケースが殆どです。ES2015以降では`String#include`で真偽値を取得できるため、分かりにくいけのイディオムとなりつつあります。

```
const string = "森森森森森";
// `String#include`は"火"があるならtrueを返す
if (!string.includes("火")) {
    // 見つからなかった場合の処理
}
```

左シフト演算子（`<<`）

左シフト演算子は、`number` を `bit` の数だけ左へシフトします。左にあふれたビットは破棄され、`0` のビットを右から詰めます。

```
number << bit;
```

次のコードでは、9を2ビット分だけ左ヘシフトしています。

```
console.log( 9 << 2); // => 36  
console.log(0b1111 << 2); // => 0b111100
```

右シフト演算子（`>>`）

右シフト演算子は、`number` を `bit` の数だけ右へシフトします。右にあふれたビットは破棄され、左端のビットのコピーを左から詰めます。

```
number >> bit;
```

次のコードでは、`-9` を2ビット分だけ右ヘシフトしています。左端のビットのコピーを使うため、常に符号は維持されます。

ゼロ埋め右シフト演算子（>>>）

ゼロ埋め右シフト演算子は、`number` を `bit` の数だけ右ヘシフトするのは右シフト演算子（`>>`）と同じです。右にあふれたビットは破棄され、`0` のビットを左から詰めます。

次のコードでは、`-9` を2ビット分だけゼロ埋め右シフトしています。左端のビットは `0` となるため、常に正の値となります。

代入演算子 ( = )

代入演算子 (=) は変数に対して値を代入します。代入演算子については「[変数と宣言の章](#)」も参照してください。

```
let x = 1;  
x = 42;  
console.log(x); // => 42
```

また、代入演算子は二項演算子と組み合わせて利用できます。`+=`、`-=`、`*=`、`/=`、`%=`、`<=>`、`>>=`、`>>>=`、`&=`、`^=`、`|=`のように、演算した結果を代入できます。

```
let num = 1;
num += 10; // num = num + 10; 同じ
console.log(num); // => 11
```

## [ES2015] 分割代入 (Destructuring assignment)

今までみてきた代入演算子は1つの変数に値を代入するものでした。分割代入を使うことで、配列やオブジェクトの値を複数の変数へ同時に代入できます。分割代入は短縮記法のひとつでES2015から導入された構文です。

分割代入は、代入演算子（`=`）を使うのは同じですが、左辺のオペラントが配列リテラルやオブジェクトリテラルとなります。

次のコードでは、右辺の配列の値を、左辺の配列リテラルの対応するインデックスにかかれた変数名へ代入します。

```
const array = [1, 2];
// aには`array`の0番目の値、bには1番目の値が代入される
const [a, b] = array;
console.log(a); // => 1
console.log(b); // => 2
```

これは、次のように書いたのと同じ結果になります。

```
const array = [1, 2];
const a = array[0];
const b = array[1];
```

同様にオブジェクトも分割代入に対応しています。オブジェクトの場合は、右辺のオブジェクトのプロパティ値を、左辺に対応するプロパティ名へ代入します。

```
const object = {
  "key": "value"
};
// プロパティ名`key`の値を、変数`key`として定義する
const { key } = object;
console.log(key); // => "value"
```

これは、次のように書いたのと同じ結果になります。

```
const object = {
  "key": "value"
};
const key = object.key;
```

## 条件（三項）演算子（`?` と `:`）

条件演算子（`?` と `:`）は三項をとる演算子であるため、三項演算子とも呼ばれます。

条件演算子は `条件式` を評価した結果が `true` ならば、`True`の時処理する式 の評価結果を返します。`条件式` が `false` である場合は、`False`の時処理する式 の評価結果を返します。

```
条件式 ? Trueの時処理する式 : Falseの時処理する式;
```

`if`文との違いは、条件演算子は式として書くことができるため値を返します。次のように、`条件式`の評価結果により `"A"` または `"B"` どちらかを返します。

```
const valueA = true ? "A" : "B";
console.log(valueA); // => "A";
const valueB = false ? "A" : "B";
console.log(valueB); // => "B";
```

条件分岐による値を返せるため、条件によって変数の初期値が違う場合などに使われます。

次の例では、`text` 文字列に `prefix` となる文字列を先頭に付ける関数を書いています。`prefix` の第二引数を省略したり文字列ではないものが指定された場合に、デフォルトの `prefix` を使います。第二引数が省略された場合には、`prefix` に `undefined` が入ります。

条件演算子の評価結果は値を返すので、`const` を使って宣言と同時に代入できます。

```
function addPrefix(text, prefix) {
    // `prefix`が指定されていない場合は"デフォルト:"を付ける
    const pre = typeof prefix === "string" ? prefix : "デフォルト:";
    return pre + text;
}

console.log(addPrefix("文字列")); // => "デフォルト:文字列"
console.log(addPrefix("文字列", "カスタム")); // => "カスタム文字列"
```

`if`文を使った場合は、宣言と代入を分ける必要があるため、`const` を使うことができません。

```
function addPrefix(text, prefix) {
    let pre = "デフォルト:";
    if (typeof prefix === "string") {
        pre = prefix;
    }
    return pre + text;
}

console.log(addPrefix("文字列")); // => "デフォルト:文字列"
console.log(addPrefix("文字列", "カスタム")); // => "カスタム文字列"
```

## 論理演算子

論理演算子は基本的に真偽値を扱う演算子で、AND、OR、NOTを表現できます。

### AND演算子（`&&`）

AND演算子（`&&`）は、左辺の値の評価結果が `true` であるならば、右辺の評価結果を返します。左辺の評価が `true` ではない場合、右辺は評価されません。

このような値が決まった時点でのそれ以上評価しないものを短絡評価（ショートサーキット）と呼びます。

```
const x = true;
const y = false;
// x -> y の順に評価される
console.log(x && y); // => false
// 左辺が falsy であるなら、その時点で false を返す
// x は評価されない
console.log(y && x); // => false
```

AND演算子は、if文と組み合わせて利用することが多い演算子です。次のように、`value` がString型かつ値が "str" である場合という条件をひとつの式として書くことができます。

```
const value = "str";
if (typeof value === "string" && value === "str") {
  console.log(`#${value} is string value`);
}
// if文のネストで書いた場合と結果は同じとなる
if (typeof value === "string") {
  if (value === "str") {
    console.log(`#${value} is string value`);
  }
}
```

このときに、`value` がString型でない場合は、その時点で `false` となります。

短絡評価はif文のネストに比べて短く書くことができます。

しかし、if文が3重4重にネストしているのは不自然なのと同様に、AND演算子やOR演算子が3つ4つ連続する場合は複雑で読みにくいコードです。その場合は抽象化ができるいかを検討するべきサインとなります。

## OR演算子 ( || )

OR演算子 ( `||` ) は、左辺の値の評価結果が `false` であるならば、右辺の評価結果を返します。AND演算子 ( `&&` ) とは逆に、左辺が `true` である場合は、右辺を評価せず `true` を返します。

```
const x = true;
const y = false;
// xがtrueなのでyは評価されない
console.log(x || y); // => true
// yはfalseなのでxを評価した結果を返す
console.log(y || x); // => true
```

OR演算子は、if文と組み合わせて利用することが多い演算子です。次のように、`value` が `0` または `1` の場合にif文の中身が実行されます。

```
const value = 1;
if (value === 0 || value === 1) {
  console.log("valueは0または1です。");
}
```

## NOT演算子 ( ! )

NOT演算子 ( `!` ) は、オペランドの評価結果が `true` であるならば、`false` を返します。

```
console.log(!false); // => true
console.log(!true); // => false
```

NOT演算子は必ず真偽値を返すため、次のように2つNOT演算子を重ねて真偽値へ変換するという使い方も見かけます。

```
const string = "";
// 空文字はfalsyな値
console.log (!!string); // => false
```

このようなケースの多くは、比較演算子を使うなどより明示的な方法で、真偽値を得ることができます。安易に `!!` による変換に頼るよりは別の方法を探してみるのがいいでしょう。

```
const string = "";
// 空文字でないことを判定
console.log(string.length > 0); // => false
```

## グループ演算子 ( ( と ) )

グループ演算子は複数の二項演算子が組み合わさった場合に、演算子の優先順序を明示できる演算子です。

たとえば、次のようにグループ演算子で囲んだ部分が最初に処理されるため、結果も変化します。

```
const a = 1;
const b = 2;
const c = 3;
console.log(a + b * c); // 7
console.log((a + b) * c); // => 9
```

演算子の優先順序はECMAScript仕様で定義されていますが、多様な演算子が出てきた場合に見分けるのは難しいです。グループ演算子はもっとも優先度が高い演算子となります。そのため、グループ演算子を使い優先順序を明示できます。

次のようなグループ演算子を使わずに書いたコードを見てみましょう。`a` が `true` または、`b` かつ `c` が `true` であるときに処理されます。

```
if (a || b && c) {
    // a が true または
    // b かつ c が true
}
```

ひとつの式に複数の種類の演算子が出てくると読みにくくなる傾向があります。このような場合にはグループ演算子を使い、結合順を明示して書くようにしましょう。

```
if (a || (b && c)) {
    // a が true または
    // b かつ c が true
}
```

## 文字列結合演算子 ( + )

数値にでてきたプラス演算子 ( + ) は、文字列の結合に利用できます。

プラス演算子は、文字列を結合した文字列を返します。

```
const value = "文字列" + "結合";
console.log(value); // => "文字列結合"
```

## カンマ演算子 ( , )

カンマ演算子 ( , ) は、カンマ ( , ) で区切った式を左から順に評価し、最後の式の評価結果を返します。

次の例では、式1、式2、式3の順に評価され、式3の評価結果を返します。

式1, 式2, 式3;

これまでに、カンマで区切るという表現は、`var`による変数宣言などでも出てきました。左から順に実行する点ではカンマ演算子の挙動は同じものですが、構文としては似て非なるものです。

```
const a = 1, b = 2, c = a + b;  
console.log(c); // => 3
```

一般にカンマ演算子を利用する機会は殆どないため、「カンマで区切った式は左から順に評価される」ということだけを知つていれば問題ありません。<sup>1</sup>

<sup>1</sup>. カンマ演算子を活用したテクニックとしてindirect callというものがあります。 ↪

# 暗黙的な型変換

この章では、明示的な型変換と暗黙的な型変換について学んでいきます。

[演算子](#)の章にて、等価演算子において暗黙的な型変換による意図しない挙動について紹介しました。

等価演算子（`==`）はオペランド同士が異なる型の値であった場合に、同じ型となるように暗黙的な型変換してから比較を行います。

暗黙的な型変換の例として、数値と真偽値の加算を見てみましょう。多くの言語では、数値と真偽値の加算は型エラーとなり、コンパイルエラーまたは実行時エラーとなります。しかし、JavaScriptでは暗黙的な型変換が行われ、エラーなく処理されます。

この例では、真偽値の `true` が数値の `1` へと暗黙的に変換されてから加算処理が行われます。

```
// エラーとなって欲しい
// しかし、暗黙的な型変換が行われ、数値の加算として計算される
1 + true; // => 2
// 次のように暗黙的に変換されてから計算される
1 + 1; // => 2
```

JavaScriptでは、エラーが発生するのではなく、暗黙的な型変換が行われてしまうケースが多くあります。暗黙的に変換が行われた場合、プログラムは例外を投げずに処理が進むため、バグの発見が難しくなります。

そのため、暗黙的な型変換は避けるべき挙動です。

この章では、次のことを学んでいきます。

- 暗黙的な型変換とはどのようなものなのか
- 暗黙的ではない明示的な型変換の方法
- 明示的な変換だけでは解決しないということ

## 暗黙的な型変換とは

暗黙的な型変換とは次のことをいいます。

- ある処理において、その処理過程において行われる明示的ではない型変換のこと

暗黙的な型変換は、演算子による演算や関数の処理過程で行われます。ここでは、演算子における暗黙的な型変換についてを中心に見ていきます。

### 等価演算子の暗黙的な型変換

もっとも有名な暗黙的な型変換は、先ほども出てきた等価演算子（`==`）です。等価演算子は、オペランド同士が同じ型となるように暗黙的な型変換をしてから、比較します。

次のように等価演算子（`==`）による比較は、驚くような結果を作り出します。

```
// 異なる型である場合に暗黙的な型変換が行われる
1 == "1"; // => true
0 == false; // => true
10 == ["10"]; // => true
```

この他にも等価演算子による予想できない結果は、比較する値と型の組み合わせの数だけあります。そのため、等価演算子の比較結果がどうなるか覚えるのは現実的ではありません。

しかし、等価演算子については暗黙的な型変換を避ける簡単な方法があります。

それは、常に厳密等価演算子（`==`）を使うことです。値を比較する際は、常に厳密等価演算子を使うことで、暗黙型変換をせずに値を比較できます。

```
1 === "1"; // => false  
0 === false; // => false
```

## さまざまな暗黙的な型変換

他の演算子についても、具体的な例を見てみましょう。

次のコードでは、数値の `1` と文字列の `"2"` をプラス演算子で処理しています。プラス演算子（`+`）は、数値の加算と文字列の結合を両方実行できるように多重定義されています。このケースでは、JavaScriptは文字列の結合を優先する仕様となっています。そのため、数値の `1` を文字列の `"1"` へ暗黙的に変換してから、文字列結合します。

```
1 + "2"; // => "12"  
// 演算過程で次のように暗黙的な変換が行われる  
"1" + "2"; // => "12"
```

もうひとつ、数値と文字列での暗黙的な型変換を見てみましょう。次のコードでは、数値の `1` から文字列の `"2"` を減算しています。

JavaScriptには、文字列に対するマイナス演算子（`-`）の定義はありません。そのため、マイナス演算子の対象となりえる数値が優先されます。これにより、文字列の `"2"` を数値の `2` へ暗黙的に変換してから、減算します。

```
1 - "2"; // => -1
// 演算過程で次のように暗黙的な変換が行われる
1 - 2; // => -1
```

2つの値までは、まだ結果の型を予想できます。しかし、3つ以上の値を扱う場合に結果を予測できなくなります。

次のように3つ以上の値を演算する場合に、値の型が混ざっていると、演算する順番によっても結果が異なります。

```
const x = 1, y = "2", z = 3;
console.log(x + y + z); // => "123"
console.log(y + x + z); // => "213"
```

このように、処理の過程でオペランドの型によって、自動的に変換されることを暗黙的な型変換と呼んでいます。

暗黙的な型変換では、結果の値の型はオペランドの型に依存しています。それを避けるには、暗黙的ではない変換—つまり明示的な型変換をする必要があります。

## 明示的な型変換

プリミティブ型への明示的な型変換する方法を見ていきます。

### Any -> 真偽値

JavaScriptでは `Boolean` コンストラクタ関数を使うことで、任意の値を `true` または `false` の真偽値に変換できます。

```
Boolean("string"); // => true
Boolean(1); // => true
Boolean({}); // => true
Boolean(0); // => false
Boolean(""); // => false
Boolean(null); // => false
```

JavaScriptでは、どの値が `true` でどの値が `false` になるかは、次のルールによって決まります。

- falsyな値は `false` になる
- falsyでない値は `true` になる

falsyな値とは次の6種類の値のことを言います。

- `false`
- `undefined`
- `null`
- `0`
- `NaN`
- `""` (空文字)

この変換ルールはif文の条件式の評価と同様です。次のようにif文に対して、真偽値以外の値を渡した時に、真偽値へと暗黙的に変換されから判定されます。

```
// x は undefined
```

```
let x;
if (!x) {
  console.log("falsyな値なら表示", x);
}
```

真偽値については、暗黙的な型変換のルールが少ないため、明示的に変換せずに扱われることも多いです。しかし、より正確な判定をして真偽値を得るには、次のように厳密等価演算子（`==`）を使い比較することです。

```
// x は undefined
let x;
if (x === undefined) {
  console.log("xがundefinedなら表示", x);
}
```

## 数値 -> 文字列

数値から文字列へ明示的に変換する場合は、`String` コンストラクタ関数を使います。

```
String(1); // => "1"
```

`String` コンストラクタは、数値以外にも色々な値を文字列へと変換できます。

```
String("str"); // => "str"
String(true); // => "true"
String(null); // => "null"
String(undefined); // => "undefined"
String(Symbol("シンボル")); // => "Symbol(シンボル)"
// プリミティブ型ではない値の場合
String([1, 2, 3]); // => "1,2,3"
String({ key: "value" }); // => "[object Object]"
String(function() {}); // 実装依存の結果
```

上記の結果からも分かるように `String` コンストラクタ関数での明示的な変換は、万能な方法ではありません。真偽値、数値、文字列、`undefined`、`null`、シンボルのプリミティブ型の値に対して変換は見た目どおりの文字列を得ることができます。

一方、プリミティブ型以外の値に対しては直感的な値を返しません。これは、オブジェクトに対しては別の方法があるためです。配列には `Array#join` メソッド、オブジェクトには `JSON.stringify` メソッドなどより柔軟な文字列化をする方法があります。そのため、`String` コンストラクタ関数での変換は、あくまでプリミティブ型に対してのみに留めるべきです。

## シンボル -> 文字列

プラス演算子を文字列に利用した場合、文字列の結合を優先します。「片方が文字列なら、もう片方のオペランドとは関係なく、結果は文字列となるのでは？」と考えるかもしれません。

```
"文字列" + x; // 文字列となる？
```

しかし、ES2015で追加されたプリミティブ型であるシンボルは暗黙的に型変換できません。文字列結合演算子をシンボルに対して利用すると例外を投げるようになっています。そのため、片方が文字列であるからと言ってプラス演算子の結果は必ず文字列になるとは限らないことが分かります。

```
"文字列と" + Symbol("シンボルの説明"); // => TypeError
```

この問題も `String` コンストラクタ関数を使うことで、シンボルを明示的に文字列化することで解決できます。

```
"文字列と" + String(Symbol("シンボルの説明")); // => "文字列とSymbol(シンボルの説明)"
```

## 文字列 -> 数値

文字列から数値に変換する典型的なケースとしてはユーザー入力から数字を受け取ることがあげられます。ユーザー入力は文字列でしか受け取ることができないため、それを数値に変換してから利用する必要があります。

文字列から数値へ明示的に変換するには `Number` コンストラクタ関数が利用できます。

```
// ユーザー入力を文字列として受け取る
const input = window.prompt("数字を入力してください", "42");
// 文字列を数値に変換する
const number = Number(input);
console.log(typeof number); // => "number";
console.log(number); // 入力された文字列を数値に変換したもの
```

また、文字列から数字を取り出し変換する関数として `Number.parseInt`、`Number.parseFloat` も利用できます。

`Number.parseInt` は文字列から整数を取り出し、`Number.parseFloat` は文字列から浮動小数点数を取り出すことができます。`Number.parseInt(文字列, 基数)` の第二引数には基数指定します。たとえば、文字列をパースして10進数として数値を取り出したい場合は、第二引数に基数として `10` を指定します。

```
// "1"をパースして10進数として取り出す
Number.parseInt("1", 10); // => 1;
// 余計な文字はパース時に無視して取り出す
Number.parseInt("42px", 10); // => 42
Number.parseInt("10.5", 10); // => 10
Number.parseFloat("1"); // => 1
Number.parseFloat("42.5px"); // => 42.5
Number.parseFloat("10.5"); // => 10.5
```

しかし、ユーザーが数字を入力するとは限りません。`Number` コンストラクタ関数、`Number.parseInt`、`Number.parseFloat` は、数字以外の文字列を渡すと `NaN` (Not a Number) を返します。

```
// 数字ではないため、数値へは変換できない
Number("文字列"); // => NaN
// 未定義の値はNaNになる
Number(undefined); // => NaN
```

そのため、任意の値から数値へ変換した場合は、`NaN` になってしまった場合の処理を書く必要があります。変換した結果が `NaN` であるかは `Number.isNaN(x)` メソッドで判定できます。

```
const userInput = "任意の文字列";
const number = Number.parseInt(userInput, 10);
if (!Number.isNaN(number)) {
  console.log("NaNではない値にパースできた", number);
}
```

## [コラム] NaNはNot a NumberだけどNumber型

`NaN` はNot a Numberの略称で、特殊な性質をもつNumber型のデータです。

この `NaN` というデータの性質については[IEEE 754](#)で規定されており、JavaScriptだけの性質ではありません。

`NaN` という値をつくる方法は簡単で、Number型と互換性のない性質のデータをNumber型へ変換した結果は `NaN` となります。たとえば、オブジェクトは数値とは互換性の無いデータです。そのため、オブジェクトを明示的に変換したとしても結果は `NaN` になります。

```
Number({}); // => NaN
```

また、`Nan` は何と演算しても結果は `NaN` になる特殊な値です。次のように、計算の途中で値が `NaN` になると、最終的な結果も `NaN` となります。

```
const x = 10;
const y = x + NaN;
const z = y + 20;
console.log(x); // => 10
console.log(y); // => NaN
console.log(z); // => NaN
```

`Nan` は `Number` 型の一種であるという名前と矛盾したデータに見えます。

```
// NaNはnumber型
typeof NaN; // => "number"
```

`NaN` しか持っていない特殊な性質として、自分自身と一致しないという特徴があります。この特徴を利用することで、ある値が `NaN` であるかを判定することができます。

```
function isNaN(x) {
    // NaNは自分自身と一致しない
    return x !== x;
}
isNaN(1); // => false
isNaN("str"); // => false
isNaN({}); // => false
isNaN([]); // => false
isNaN(NaN); // => true
```

同様の処理を行う方法として `Number.isNaN(x)` メソッドがあります。実際に値が `NaN` かを判定する際には、`Number.isNaN(x)` メソッドを利用するとよいでしょう。

```
Number.isNaN(NaN); // => true
```

`Nan` は暗黙的な型変換の中でもっとも避けたい値となります。理由として、先ほど紹介したように `Nan` は何と演算しても結果が `NaN` となってしまうためです。これにより、計算していた値がどこで `NaN` となったのかが分かりにくく、デバッグが難しくなります。

たとえば、次の `sum` 関数は可変長引数（任意の個数の引数）を受け取り、その合計値を返します。しかし、`sum(x, y, z)` と呼び出した時の結果が `NaN` になってしまいました。これは、引数の中に `undefined`（未定義の値）が含まれているためです。

```
// 任意の個数の数値を受け取り、その合計値を返す関数
function sum(...values) {
    return values.reduce((total, value) => {
        return total + value;
    }, 0);
}
const x = 1, z = 10;
let y; // `y`はundefined
sum(x, y, z); // => NaN
```

そのため、`sum(x, y, z);` は次のように呼ばれていたのと同じ結果になります。`undefined` に数値を加算すると結果は `NaN` となります。

```
sum(1, undefined, 10); // => NaN
// 計算中にNaNとなるため、最終結果もNaNになる
1 + undefined; // => NaN
NaN + 10; // => NaN
```

これは、`sum` 関数において引数を明示的にNumber型へ変換したとしても回避することはできません。つまり、次のように明示的な型変換しても解決できない問題あることが分かります。

```
function sum(...values) {
    return values.reduce((total, value) => {
        // `value`をNumberで明示的に数値へ変換してから加算する
        return total + Number(value);
    }, 0);
}
const x = 1, z = 10;
let y; // `y`はundefined
sum(x, y, z); // => NaN
```

この意図しない `NaN` への変換を避ける方法として、大きく分けて2つの方法があります。

- `sum` 関数側（呼ばれる側）で、Number型の値以外を受け付けなくする
- `sum` 関数を呼び出す側で、Number型の値のみを渡すようにする

つまり、呼び出す側または呼び出される側で対処することですが、どちらも行うことがより安全なコードにつながります。

そのためには、`sum` 関数が数値のみを受け付けるということを明示する必要があります。

明示する方法として `sum` 関数のドキュメント（コメント）として記述したり、引数に数値以外の値がある場合は例外を投げるという処理を追加するといった形です。

JavaScriptではコメントで引数の型を記述する書式として [JSDoc](#) が有名です。また、実行時に値がNumber型であるかをチェックし `throw` 文で例外をなげることで、`sum` 関数の利用者に使い方を明示できます。（`throw` 文については「[例外処理](#)」の章で解説します）

この2つを利用して `sum` 関数の前提条件を詳細に実装したものは次のようになります。

```
/**
 * 数値を合計した値を返します。
 * 一つ以上の数値と共に呼び出す必要があります。
 * @param {...number} values
 * @returns {number}
 */
function sum(...values) {
    return values.reduce((total, value) => {
        // 値がNumber型ではない場合に、例外を投げる
        if (typeof value !== "number") {
            throw new Error(`"${value}"はNumber型ではありません`);
        }
        return total + Number(value);
    }, 0);
}
const x = 1, z = 10;
let y; // `y`はundefined
console.log(x, y, z);
// Number型の値ではない`y`を渡しているため例外が発生する
sum(x, y, z); // => Error
```

このように、`sum` 関数はどのように使うべきかを明示することで、エラーとなった時に呼ばれる側と呼び出し側でどちらに問題があるのかが明確になります。この場合は、`sum` 関数へ `undefined` な値を渡している呼び出し側に問題があります。

JavaScriptは、型エラーに対して暗黙的な型変換を行うなど驚くほど許容しています。そのため、大きなJavaScriptアプリケーションを書く場合は、このような検出しにくいバグを見つけられるように書くことは重要です。

## 明示的な変換でも解決しないこと

先ほどの例からも分かるように、あらゆるケースが明示的な変換で解決できるわけではありません。Number型と互換性がない値を数値にしても、`Nan`となってしまいます。一度、`Nan`になってしまふと`Number.isNaN(x)`で判定して処理を終えるしかありません。

JavaScriptの型変換は基本的に情報が減る方向へしか変換できません。そのため、明示的な変換をする前に、まず変換がそもそも必要なのかを考える必要があります。

### 空文字かどうかを判定する

たとえば、文字列が空文字なのかを判定したい場合を考えみましょう。`""`(空文字)はfalsyな値であるため、明示的に`Boolean`コンストラクタ関数で真偽値へ変換できます。しかし、falsyな値は空文字以外にもあるため、明示的に変換したからといって空文字だけを判定できるわけではありません。

次のコードでは、明示的な型変換を行っていますが、`0`も空文字となってしまい意図しない挙動になっています。

```
// 空文字かどうかを判定
function isEmptyString(string) {
    return !Boolean(string);
}
isEmptyString(""); // => true
// falsyなら値なら、trueを返している
isEmptyString(0); // => true
// undefined渡した場合もtrueとなる
isEmptyString(); // => true
```

殆どのケースにおいて、真偽値を得るには、型変換ではなく別の方法が存在します。

この場合、空文字とは「String型で文字長が0の値」であると定義することで、`isEmptyString`関数はもっと正確に書くことができます。次のように実装することで、値が空文字であるかを正しく判定できるようになりました。

```
// 空文字かどうかを判定
function isEmptyString(string) {
    // String型でlengthが0の値が空文字
    return typeof string === "string" && string.length === 0;
}
isEmptyString(""); // => true
// falsyな値でも正しく判定できる
isEmptyString(0); // => false
isEmptyString(); // => false
```

`Boolean`を使った型変換は、楽をするための型変換であり、正確に真偽値を得るための方法ではありません。そのため、型変換をする前にまず別の方法で解決できないかを考えることも大切です。

# 関数と宣言

関数とは、ある一連の手続き（文の集まり）を1つの処理としてまとめる機能です。関数を利用してすることで、同じ処理を毎回書くのではなく、一度定義した関数を呼び出すことで同じ処理を実行できます。

これまで利用してきたコンソール表示を行うConsole APIも関数です。`console.log`は「受け取った値をコンソールへ出力する」という処理をまとめた関数です。

この章では、関数の定義方法や呼び出し方について見ていきます。

## 関数宣言

JavaScriptでは、関数を定義するために`function`キーワードを使います。`function`から始まる文は関数宣言と呼び、次のように関数を定義できます。

```
// 関数宣言
function 関数名(仮引数1, 仮引数2) {
    // 関数を呼び出された時の処理
    // ...
    return 関数の返り値;
}
// 関数呼び出し
const 関数の結果 = 関数名(引数1, 引数2);
console.log(関数の結果); // => 関数の返り値
```

関数は次の4つの要素から構成されています。

- 関数名 - 利用できる文字列は変数名と同じ
- 仮引数 - 関数の呼び出し時に渡された値が入る変数。複数ある場合は`,`（カンマ）で区切る
- 関数の中身 - `{`と`}`で囲んだ関数の処理を書く場所
- 関数の返り値 - 関数を呼び出したときに、呼び出し元へ返される値

宣言した関数は、`関数名()`と関数名にカッコをつけることで呼び出せます。関数を引数と共に呼ぶ際は、`関数名(引数1, 引数2)`とし、引数が複数ある場合は`,`（カンマ）で区切れます。

関数の中身では`return`文によって、関数の実行結果として任意の値を返せます。

次のコードでは、引数で受け取った値を2倍にして返す`multiple`という関数を定義しています。`multiple`関数は`num`という仮引数が定義されており、`10`という値を引数として渡して関数を呼び出しています。仮引数の`num`には`10`が代入され、その値を2倍にしたものを作成`return`文で返しています。

```
function multiple(num) {
    return num * 2;
}

console.log(multiple(10)); // => 20
```

関数では`return`文が実行されると、関数内ではそれ以降の処理は行われません。また関数が値を返す必要がない場合は、`return`文では返り値を省略できます。`return`文の返り値を省略した場合は、`undefined`という値を返します。

```
function fn() {
    // 何も返り値を指定しない場合は`undefined`を返す
    return;
    // この文は実行されません
```

```

}
console.log(fn()); // => undefined

```

関数が何も値を返す必要がない場合は、`return` 文そのものを省略できます。`return` 文そのものを省略した場合は、`undefined` という値を返します。

```

function fn() {
}

console.log(fn()); // => undefined

```

## 関数の引数

JavaScriptでは、関数の定義した仮引数の個数と実際に呼び出し時の引数の個数が違っても、関数を呼び出せます。そのため、引数の個数があつてないときの挙動を知る必要があります。また、引数が省略されたときに、デフォルトの値を指定するデフォルト引数という構文についても見ていきます。

### 引数が少ないとき

定義した関数の仮引数よりも呼び出し時の引数が少ない場合、余った仮引数には`undefined` という値が代入されます。

次のコードでは、引数として渡した値をそのまま返す`echo` 関数を定義しています。`echo` 関数は仮引数`x`を定義していますが、引数を渡さずに呼び出すと仮引数`x`には`undefined`が入ります。

```

function echo(x) {
  return x;
}

echo(1); // => 1
echo(); // => undefined

```

複数の引数を受け付ける関数でも同様に、余った仮引数には`undefined`が入ります。

次のコードでは、2つの引数を受け取りそれを配列として返す`argumentsToArray` 関数を定義しています。このとき、引数として1つの値しか渡していない場合、残る仮引数には`undefined`が代入されます。

```

function argumentsToArray(x, y) {
  return [x, y];
}

argumentsToArray(1, 2); // => [1, 2]
// 仮引数のxには1、yにはundefinedが入る
argumentsToArray(1); // => [1, undefined]

```

### [ES2015] デフォルト引数

デフォルト引数（デフォルトパラメータ）は、仮引数に対応する引数が渡されていない場合に、デフォルトで代入される値を指定できます。次のように、仮引数に対して`仮引数 = デフォルト値` という構文で、仮引数ごとにデフォルト値を指定できます。

```

function 関数名(仮引数1 = デフォルト値1, 仮引数2 = デフォルト値2) {
}

```

次のコードでは、渡した値をそのまま返す echo 関数を定義しています。さきほどの echo 関数とは異なり、仮引数 `x` に対してデフォルト値を指定しています。そのため、引数を渡さずに echo 関数を呼び出すと、`x` には "デフォルト値" が代入されます。

```
function echo(x = "デフォルト値") {
    return x;
}

echo(); // => 1
echo(); // => "デフォルト値"
```

ES2015でデフォルト引数が導入されるまでは、OR演算子（`||`）を使ったデフォルト値の指定がよく利用されていました。

```
function addPrefix(text, prefix) {
    const pre = prefix || "デフォルト:";
    return pre + text;
}

console.log(addPrefix("文字列")); // => "デフォルト:文字列"
console.log(addPrefix("文字列", "カスタム:")); // => "カスタム:文字列"
```

しかし、OR演算子（`||`）を使ったデフォルト値の指定にはひとつ問題があります。OR演算子（`||`）では、左辺のオペランドが falsy な値の場合、右辺のオペランドを評価します。falsy な値とは、次のような `false` のような値のことです。

- `false`
- `undefined`
- `null`
- `0`
- `NaN`
- `""` (空文字)

そのため、次のように `prefix` に空文字を指定した場合にもデフォルト値が入ります。これは書いた人が意図した挙動なのかがとてもわかりにくく、このような挙動はバグにつながることがあります。

```
function addPrefix(text, prefix) {
    const pre = prefix || "デフォルト:";
    return pre + text;
}

// falsy な値を渡すとデフォルト値が入ってしまう
console.log(addPrefix("文字列")); // => "デフォルト:文字列"
console.log(addPrefix("文字列", "")); // => "デフォルト:文字列"
console.log(addPrefix("文字列", "カスタム:")); // => "カスタム:文字列"
```

デフォルト引数を使って書くことで、このような挙動は減らすことができ安全です。デフォルト引数では、引数が渡されなかった場合のみデフォルト値が入ります。

```
function addPrefix(text, prefix = "デフォルト:") {
    return prefix + text;
}

// falsy な値を渡してもデフォルト値は代入されない
console.log(addPrefix("文字列")); // => "デフォルト:文字列"
console.log(addPrefix("文字列", "")); // => "文字列"
console.log(addPrefix("文字列", "カスタム:")); // => "カスタム:文字列"
```

## 引数が多いとき

関数の仮引数に対して引数の個数が多い場合、溢れた引数は単純に無視されます。

次コードでは、2つの引数を足し算する `add` 関数を定義しています。この `add` 関数には仮引数が2つしかありません。そのため、3つ以上の引数を渡しても3番目以降の引数は単純に無視されます。

```
function add(x, y) {
  return x + y;
}
add(1, 3); // => 4
add(1, 3, 5); // => 4
```

## 可変長引数

関数において引数の数が固定ではなく、任意の個数の引数を受け取りたい場合があります。たとえば、`Math.max(...args)` は引数を何個でも受け取り、受け取った引数の中で最大の数値を返す関数です。このような、固定した数ではなく任意の個数の引数を受け取れることを可変長引数とよびます。

```
// Math.maxは可変長引数を受け取る関数
const max = Math.max(1, 5, 10, 20);
console.log(max); // => 20
```

可変長引数を実現するためには、Rest parametersか関数の中でのみ参照できる `arguments` という特殊な変数を利用します。

### [ES2015] Rest parameters

Rest parametersは、仮引数名の前に `...` をつけた仮引数のことで、残余引数とも呼ばれます。Rest parametersには、関数に渡された値が配列として代入されます。

次のコードでは、`fn` 関数に `...args` という Rest parametersが定義されています。この `fn` 関数を呼び出したときのが、`args` という変数に配列として代入されます。

```
function fn(...args) {
  // argsは引数の値が順番に入った配列
  console.log(args[0]); // => "a"
  console.log(args[1]); // => "b"
  console.log(args[2]); // => "c"
}
fn("a", "b", "c");
```

Rest parametersは、通常の仮引数と組み合わせても定義できます。他の仮引数と組み合わせる際には、必ずRest parametersは末尾の仮引数として定義する必要があります。

次のコードでは、1番目の引数は `arg1` に代入され、残りの引数が `restArgs` に配列として代入されます。

```
function fn(arg1, ...restArgs) {
  console.log(arg1); // => "a"
  console.log(restArgs); // => ["b", "c"]
}
fn("a", "b", "c");
```

## arguments

可変長引数を扱う方法として、`arguments` という関数の中でのみ参照できる特殊な変数があります。`arguments` は関数に渡された引数の値がすべて入ったArray-likeなオブジェクトです。Array-likeなオブジェクトは、配列のようにインデックスで要素へアクセスできます。しかし、`Array` ではないため、実際の配列とは異なり `Array` のメソッドは利用できないという特殊なオブジェクトです。

次のコードでは、`fn` 関数に仮引数が定義されていません。しかし、関数の内部では `arguments` という変数で、実際に渡された引数を配列のように参照できます。

```
function fn() {
  // `arguments`はインデックスを指定して各要素にアクセスできる
  console.log(arguments[0]); // => "a"
  console.log(arguments[1]); // => "b"
  console.log(arguments[2]); // => "c"
}
fn("a", "b", "c");
```

Rest parametersが利用できる環境では、`arguments` 変数を使うべき理由はありません。`arguments` 変数には次のような問題があります。

- Arrow Functionでは利用できない (Arrow Functionについては後述)
- Array-likeオブジェクトであるため、Arrayのメソッドを利用できない
- 関数が可変長引数を受け付けるのかを仮引数だけを見て判断できない (宣言的ではない)

そのため、可変長引数が必要な場合はRest parametersでの実装を推奨します。

## 関数の引数と分割代入

関数の引数においても分割代入 (Destructuring assignment) が利用できます。分割代入はオブジェクトや配列からプロパティを取り出し、変数として定義し直す構文です。

次のコードでは、関数の引数として `user` オブジェクトを渡し、`id` プロパティをコンソールへ出力しています。

```
function printUserId(user) {
  console.log(user.id); // => 42
}
const user = {
  id: 42
};
printUserId(user);
```

関数の引数に分割代入を使うことで、このコードは次のように書けます。次のコードの `printUserId` 関数はオブジェクトを引数として受け取ります。この受け取った `user` オブジェクトの `id` プロパティを変数 `id` として定義しています。

```
// 第1引数のオブジェクトから`id`プロパティを変数`id`として定義する
function printUserId({ id }) {
  console.log(id); // => 42
}
const user = {
  id: 42
};
printUserId(user);
```

代入演算子（`=`）におけるオブジェクトの分割代入では、左辺に定義したい変数を定義し、右辺のオブジェクトから対応するプロパティを代入していました。関数の仮引数が左辺で、関数に渡す引数を右辺と考えるとほぼ同じ構文であることが分かります。

```
const user = {
  id: 42
};
// オブジェクトの分割代入
const { id } = user;
console.log(id); // => 42
// 関数の引数の分割代入
function printUserId({ id }) {
  console.log(id); // => 42
}
printUserId(user);
```

この関数の引数における分割代入は、オブジェクトだけではなく配列についても利用できます。

```
function print([first, second]) {
  console.log(first); // => 1
  console.log(second); // => 2
}
const array = [1, 2];
print(array);
```

## [コラム] 同じ名前の関数宣言は上書きされる

関数宣言で定義した関数は、関数の名前でのみ区別されます。また、同じ名前の関数を複数回宣言した場合には、後ろで宣言された関数によって上書きされます。

```
function fn(){
  return 1;
}
function fn(){
  return 2;
}
console.log(fn()); // => 2
```

仮引数の定義が異なっていても、関数の名前が同じなら上書きされます。そのため、引数の違いで関数を分けたい場合は、別々の名前で定義するか関数の内部で処理を分岐する必要があります。[オーバーロード](#)

## 関数はオブジェクト

JavaScriptでは、関数は関数オブジェクトとも呼ばれ、オブジェクトの一種です。関数はただのオブジェクトとは異なり、関数名に`()`を付けることで、まとめた処理を呼び出すことができます。

一方で、`()`をつけて呼び出されなければ、関数をオブジェクトとして参照できます。そのため、関数は他の値と同じように変数へ代入したり、関数の引数として渡すことが可能です。

次のコードでは、定義した`fn`関数を`myFunc`変数へ代入してから、呼び出しています。

```
function fn() {
  console.log("fnが呼び出されました");
}
// 関数`fn`を`myFunc`変数に代入している
const myFunc = fn;
myFunc();
```

このように関数が値として扱えることを、[ファーストクラスファンクション](#)（第一級関数）と呼びます。

さきほどのコードでは、関数宣言をしてから変数へ代入していましたが、最初から関数を値として定義することも可能です。

関数を値として定義する場合には、関数宣言と同じ `function` キーワードを使った方法と Arrow Functionを使った方法があります。どちらの方法も、関数を式（代入する値）として扱うため関数式と呼びます。

## 関数式

関数式とは、関数を値として変数へ代入している式のことを言います。関数宣言は文でしたが、関数式では関数を文字列などと同じように値として扱っています。

```
// 関数式
const 関数名 = function() {
    // 関数を呼び出した時の処理
    // ...
    return 関数の返り値;
};
```

関数式では `function` キーワードの右辺に書く関数名は省略できます。なぜなら、定義した関数式は変数名で参照できるためです。一方、関数宣言では `function` キーワードの右辺の関数名は省略できません。

```
// 関数式は変数名で参照できるため、"関数名"を省略できる
const 変数名 = function() {
};

// 関数宣言では"関数名"は省略できない
function 関数名() {
```

このように関数式では、名前を持たない関数を変数に代入できます。このような名前を持たない関数を匿名関数（または無名関数）と呼びます。

もちろん関数式でも関数に名前を付けることもできます。しかし、この関数の名前は関数の外からは呼ぶことができません。一方、関数の中からは呼ぶことができるため、再帰的に関数を呼び出す際などに利用されます。

```
// factorialは関数の外から呼び出せる名前
// innerFactは関数の外から呼び出せない名前
const factorial = function innerFact(n) {
    if (n === 0) {
        return 1;
    }
    // innerFactを再帰的に呼び出している
    return n * innerFact(n - 1);
};
console.log(factorial(3)); // => 6
```

## [ES2015] Arrow Function

関数式には `function` キーワードを使った方法以外に、Arrow Functionと呼ばれる書き方があります。名前のとおり矢印のような `=>`（イコールと大なり記号）を使い、匿名関数を定義する構文です。次のように、`function` キーワードを使った関数式とよく似た書き方をします。

```
const 関数名 = () => {
    // 関数を呼び出した時の処理
    // ...
    return 関数の返り値;
};
```

Arrow Functionには書き方のいくつかパターンがありますが、`function` キーワードに比べて短く書けるようになっています。また、Arrow Functionには省略記法があり、次の場合にはさらに短く書けます。

- 関数の仮引数が1つのときは`()`を省略できる
- 関数の処理が1つの式である場合に、ブロックと`return`文を省略できる
  - その式の評価結果を`return`の返り値とする

```
// 仮引数の数と定義
const fnA = () => { /* 仮引数がないとき */ };
const fnB = (x) => { /* 仮引数が1つのみのとき */ };
const fnC = x => { /* 仮引数が1つのみのときは()を省略可能 */ };
const fnD = (x, y) => { /* 仮引数が複数の時 */ };

// 値の返し方
// 次の2つの定義は同じ意味となる
const mulA = x => { return x * x; } // ブロックの中でreturn
const mulB = x => x * x; // 1行のみの場合はreturnとブロックを省略できる
```

Arrow Functionについては次のような特徴があります。

- 名前を付けることができない（常に匿名関数）
- `this` が静的に決定する（詳細は「[関数とスコープ](#)」の章で解説します）
- `function` キーワードに比べて短く書くことができる
- `new` できない（コンストラクタ関数ではない）
- `arguments` をもたない

たとえば`function` キーワードの関数式では、値を返すコールバック関数を次のように書きます。`Array#map` は配列の要素を順番にコールバック関数へ渡し、そのコールバック関数が返した値を新しい配列にして返します。

```
const array = [1, 2, 3];
// 1, 2, 3と順番に値が渡されコールバック関数（匿名関数）が処理する
const doubleArray = array.map(function(value) {
  return value * 2; // 返した値をまとめた配列ができる
});
console.log(doubleArray); // => [2, 4, 6];
```

Arrow Functionでは処理が1つの式だけである場合に、`return` 文を省略し暗黙的にその式の評価結果を`return` の返り値とします。また、Arrow Functionは仮引数が1つである場合は`()`を省略できます。このような省略はコールバック関数を多用する場合にコードの見通しを良くします。

```
const array = [1, 2, 3];
const doubleArray = array.map(value => value * 2);
console.log(doubleArray); // => [2, 4, 6];
```

Arrow Functionは`function` キーワードの関数式に比べて、できることとできないことがはっきりしています。たとえば、`function` キーワードでは非推奨としていた`arguments` 変数を参照できますが、Arrow Functionでは参照できません。Arrow Functionでは、人による解釈や実装の違いが生まれにくくなります。

また、`function` キーワードとArrow Functionの大きな違いとして、`this` という特殊なキーワードに関する挙動の違いあります。`this` については「[関数とスコープ](#)」の章で解説しますが、Arrow Functionではこの`this` の問題の多くを解決できるという利点があります。

そのため、Arrow Functionで問題ない場合はArrow Functionで書き、そうでない場合は`function` キーワードを使うことを推奨します。

## コールバック関数

関数はファーストクラスであるため、その場で作った匿名関数を関数の引数（値）として渡すことができます。引数として渡される関数のことをコールバック関数と呼びます。一方、コールバック関数を引数として使う関数やメソッドのことを高階関数と呼びます。

```
function 高階関数(コールバック関数) {
  コールバック関数();
}
```

たとえば、`Array#forEach` メソッドはコールバック関数を引数として受け取る高階関数です。`forEach` メソッドは、配列の各要素に対してコールバック関数を一度ずつ呼び出します。

```
const array = [1, 2, 3];
const output = (value) => {
  console.log(value);
};
array.forEach(output);
// 次のような実行しているのと同じ
// output(1); => 1
// output(2); => 2
// output(3); => 3
```

毎回、関数を定義してその関数をコールバック関数として渡すのは、少し手間がかかります。そこで、関数はファーストクラスであることを利用して、コールバック関数となる匿名関数をその場で定義して渡せます。

```
const array = [1, 2, 3];
array.forEach((value) => {
  console.log(value);
});
```

コールバック関数は非同期処理においてもよく利用されます。非同期処理におけるコールバック関数の利用方法については「[非同期処理](#)」の章で解説します。

## メソッド

オブジェクトのプロパティである関数をメソッドと呼びます。JavaScriptにおいて、関数とメソッドの機能的な違いはありません。しかし、呼び方を区別したほうがわかりやすいため、ここではオブジェクトのプロパティである関数をメソッドと呼びます。

次のコードにおける `object` の `method` プロパティに関数を定義しています。この `object.method` プロパティがメソッドです。

```
const object = {
  method1: function() {
    // `function` キーワードでのメソッド
  },
  method2: () => {
    // Arrow Functionでのメソッド
  }
};
```

次のように空の `object` を宣言してから、`method` プロパティへ関数を代入しても同様です。

```
const object = {};
object.method = function() {
```

メソッドを呼び出す場合は、関数呼び出しと同様に `オブジェクト.メソッド名()` と書くことで呼び出せます。

```
const object = {
  method: function() {
    return "this is method";
  }
};
console.log(object.method()); // => "this is method"
```

## [ES2015] メソッドの短縮記法

先ほどの方法では、プロパティに関数を代入するという書き方になっていました。ES2015からは、メソッドとしてプロパティを定義するための短縮した書き方が追加されています。

次のように、オブジェクトリテラルの中で `メソッド名(){ /*メソッドの処理*/ }` と書くことができます。

```
const object = {
  method() {
    return "this is method";
  }
};
console.log(object.method()); // => "this is method"
```

この書き方はオブジェクトのメソッドだけではなく、クラスのメソッドと共に通の書き方となっています。メソッドを定義する場合は、できるだけこの短縮記法に統一した方がよいでしょう。

## まとめ

この章では、次の学びました。

- 関数の宣言方法
- 関数を値として使う方法
- コールバック関数
- 関数式とArrow Function
- メソッドの定義方法

基本的な関数の定義や値としての関数について学びました。JavaScriptでは、非同期処理を扱うことが多く、その場合にコールバック関数が使われます。Arrow Functionを使うことで、コールバック関数を短く簡潔に書くことができます。

JavaScriptでの、メソッドはオブジェクトのプロパティである関数のことです。ES2015からは、メソッドを定義する構文が追加されているため活用していきます。

オーバーロード<sup>6</sup> JavaScriptにはオーバーロードがありません。[☞](#)

# 文と式

本格的に基本文法について学ぶ前に、JavaScriptというプログラミング言語がどのような要素からできているかを見ていきましょう。

JavaScriptは、文（Statement）と式（Expression）から構成されています。

## 式

式（Expression）を簡潔に述べると、値を生成し、変数に代入できるものを言います。

42 のようなリテラルや `foo` といった変数、関数呼び出しが式です。また、`1 + 1` のような式と演算子の組み合わせも式と呼びます。

式の特徴として、式を評価すると結果の値を得ることができます。この結果の値を評価値と呼びます。

評価した結果を変数に代入できるものは式であるという理解で問題ありません。

```
// 1という式の評価値を表示
console.log(1); // => 1
// 1 + 1という式の評価値を表示
console.log(1 + 1); // => 2
// 式の評価値を変数に代入
const total = 1 + 1;
// 関数式の評価値(関数オブジェクト)を変数に代入
const fn = function() {
  return 1;
};
// fn() という式の評価値を表示
console.log(fn()); // => 1
```

## 文

文（Statement）を簡潔に述べると、処理を実行する1ステップが1つの文といえます。JavaScriptでは、文の末尾にセミコロン（;）を置くことで文と文に区切りを付けます。

ソースコードとして書かれた文を上から処理していくことで、プログラムが実行されます。

```
処理する文;
処理する文;
処理する文;
```

たとえば、if文やfor文などが文と呼ばれるものです。次のように、文の処理の一部として式を含むことがあります。

```
const isTrue = true;
// isTrueという式がif文の中に出てくる
if (isTrue) {
```

一方、if文などは文であり式になることはできません。

式ではないため、if文を変数へ代入することはできません。そのため、次のようなコードは `syntaxError` となります。

```
// 構文として間違っているため SyntaxError
var forIsNotExpression = if (true) { /* ifは文であるため式にはなれない */ }
```

## 式文

一方、式（Expression）は文（Statement）になることができます。文となった式のことを式文と呼び、基本的に文が書ける場所には式を書くことができます。

その際に、式文（Expression statement）は文の一種であるため、セミコロンで文を区切っています。

```
// 式文であるためセミコロンをついている
式;
```

式は文となることができますが、先ほどのif文のように文は式となることができません。

## ブロック文

次のような、文を { と } で囲んだ部分をブロックと言います。ブロックには、複数の文を書くことができます。

```
{
  文;
  文;
}
```

if文など文の中には、ブロックで終わるものがあります。

文の末尾にはセミコロンを付けるとしていましたが、例外としてブロックで終わる文の末尾には、セミコロンが不要となっています。

```
// ブロックで終わらない文なので、セミコロンが必要
if (true) console.log(true);
// ブロックで終わる文なので、セミコロンが不要
if (true) {
  console.log(true);
}
```

## function宣言（文）とfunction式

[関数と宣言](#)の章において、関数を定義する方法を学びました。functionキーワードから文を開始する関数宣言と、変数へ関数式を代入する方法があります。

関数宣言（文）と関数式は、どちらも function というキーワードを利用しています。

```
// learn関数を宣言する関数宣言文
function learn() {
}
// 関数式をread変数へ代入
const read = function() {
};
```

この文と式の違いを見ると、関数宣言文にはセミコロンがなく、関数式にはセミコロンがあります。このような、違いがなぜ生まれるのかは、ここまで的内容を適用することで説明できます。

関数宣言文で定義した learn 関数には、セミコロンがありません。これは、ブロックで終わる文にはセミコロンが不要であるためです。

一方、関数式を `read` 変数へ代入したものには、セミコロンがあります。

「ブロックで終わる関数であるためセミコロンが不要なのでは？」と思うかもしれません。

しかし、この匿名関数は式であり、この処理は変数を宣言する文の一部であることが分かります。つまり、次のように置き換えるても同じといえるため、末尾にセミコロンが必要となります。

```
function fn() {}  
// fn(式)の評価値を代入する変数宣言の文  
const read = fn;
```

## まとめ

この章では次のことを学びました。

- JavaScriptは文（Statement）と式（Expression）から構成される
- 文は式になることができない
- 式は文になることができる（式文）
- 文の末尾にはセミコロンを付ける
- ブロックで終わる文は例外的にセミコロンを付けなくてよい

JavaScriptには、特殊なルールにもとづき、セミコロンがない文も行末に自動でセミコロンが挿入されるという仕組みがあります。しかし、この仕組みは構文を正しく解析できない場合に、セミコロンを足すという挙動を含みます。これにより、意図しない挙動を生むことがあります。そのため、必ず文の末尾にはセミコロンを書くようにします。

エディタやIDEの中にはセミコロンの入力の補助をしてくれるものや、[ESLint](#)などのLintツールを使うことで、セミコロンが必要なのかをチェックできます。

セミコロンが必要か見分けるにはある程度慣れが必要ですが、ツールを使い静的にチェックすることもできます。そのため、ツールなどの支援を受けて経験的に慣れていくこともよい方法といえます。

# 条件分岐

この章ではif文やswitch文を使った条件分岐について学んでいきます。

## if文

if文を使うことで、プログラム内に条件分岐を書くことができます。

if文は次のような構文が基本形となります。 条件式 の評価結果が `true` であるならば、 実行する文 が実行されます。

```
if (条件式) {
    実行する文;
}
```

次のコードでは 条件式 が `true` であるため、ifの中身が実行されます。

```
if (true) {
    console.log("この文は実行されます");
}
```

実行する文 が1行のみの場合は、`{` と `}` のブロックを省略できます。しかし、どこまでがif文かがわかりにくくなるため、常にブロックで囲むことを推奨します。

```
if (true)
    console.log("この文は実行されます");
```

if文は 条件式 に比較演算子などを使い、その比較結果によって処理を分岐するためによく使われます。次のコードでは、`x` が `10` よりも大きな値である場合に、if文の中身が実行されます。

```
const x = 42;
if (x > 10) {
    console.log("xは10より大きな値です");
}
```

if文の 条件式 には `true` または `false` といった真偽値以外の値も指定できます。真偽値以外の値の場合、その値を暗黙的に真偽値へ変換した評価結果を使い、if文の判定を行います。

真偽値へ変換した結果が `true` となる値の種類は多いため、逆に変換した結果が `false` となる値を覚えるのが簡単です。JavaScriptでは次の値は `false` に変換され、これらの値はfalsyと呼ばれます。（「[暗黙的な型変換](#)の章」を参照）

- `false`
- `undefined`
- `null`
- `0`
- `NaN`
- `""` (空文字)

falsy以外の値は `true` へ変換されます。

そのため、`true`、`"文字列"`、`0`以外の数値などを 条件式 に指定した場合は、`true` へと変換されます。

次のコードは、条件式が `true` へと変換されるため、if文の中身は実行されます。

```

if (true) {
    console.log("この文は実行されます");
}
if ("文字列") {
    console.log("この文は実行されます");
}
if (42) {
    console.log("この文は実行されます");
}

```

falsyな値を 条件式 に指定した場合は、`false` へと変換されます。次のコードは、条件式が `false` へと変換されるため、if文の中身は実行されません。

```

if (false) {
    // この文は実行されません
}
if ("") {
    // この文は実行されません
}
if (0) {
    // この文は実行されません
}
if (null) {
    // この文は実行されません
}

```

## else if文

複数の条件分岐を書く場合は、if文に続けてelse if文を使うことでできます。たとえば、次の3つの条件分岐するプログラムを考えます。

- `version` が "ES5" ならば "ECMAScript 5" と出力
- `version` が "ES6" ならば "ECMAScript 2015" と出力
- `version` が "ES7" ならば "ECMAScript 2016" と出力

次のコードでは、if文とelse if文を使うことで3つの条件を書いています。

```

const version = "ES6";
if (version === "ES5") {
    console.log("ECMAScript 5");
} else if (version === "ES6") {
    console.log("ECMAScript 2015");
} else if (version === "ES7") {
    console.log("ECMAScript 2016");
}

```

## else文

if文とelse if文では、条件に一致した場合の処理をブロック内に書いていました。一方、条件に一致しなかった場合の処理は、else文を使うことでできます。

```

const string = "JavaScript";
if (string.length > 0) {
    console.log(`文字列 "${string}" があります`);
} else {
    console.log("空文字列です");
}

```

## ネストしたif文

`if`、`else if`、`else`文は 実行する文 としてさらにif文を書きネストできます。

ネストしたif文の例として、今年がうるう年かを判定してみましょう。

うるう年の条件は次のとおりです。

- 西暦で示した年が4で割り切れる年はうるう年です
- ただし、西暦で示した年が100で割り切れる年はうるう年ではありません
- ただし、西暦で示した年が400で割り切れる年はうるう年です

西暦で示した年は `new Date().getFullYear();` で取得できるため、この条件をif文で表現すると次のように書くことができます。

```
const year = new Date().getFullYear();
if (year % 4 === 0) { // 4で割り切れる
  if (year % 100 === 0) { // 100で割り切れる
    if (year % 400 === 0) { // 400で割り切れる
      console.log("うるう年です");
    } else {
      console.log("うるう年ではありません");
    }
  } else {
    console.log("うるう年です");
  }
} else {
  console.log("うるう年ではありません");
}
```

条件を上から順に書き下したため、ネストが深い文となってしまっています。一般にネストは少ない方が、読みやすいコードとなります。

条件を少し読み解くと、400で割り切れる年は無条件にうるう年であることがわかります。そのため、条件を並び替えることで、ネストするif文なしに書くことができます。

```
const year = new Date().getFullYear();
if (year % 400 === 0) { // 400で割り切れる
  console.log("うるう年です");
} else if (year % 100 === 0) { // 100で割り切れる
  console.log("うるう年ではありません");
} else if (year % 4 === 0) { // 4で割り切れる
  console.log("うるう年です");
} else { // それ以外
  console.log("うるう年ではありません");
}
```

## switch文

switch文は次のような構文を持ち、式 の評価結果が指定した値である場合に行う処理を並べて書きます。

```
switch (式) {
  case ラベル1:
    // `式`の評価結果が`ラベル1`と一致する場合に実行する文
    break;
  case ラベル2:
    // `式`の評価結果が`ラベル2`である場合に実行する文
    break;
  default:
    // どのcaseにも該当しない場合の処理
    break;
```

```
}
```

// break; 後はここから実行される

switch文はif文と同様に式の評価結果にもとづく条件分岐を扱います。またbreak文は、switch文から抜けswitch文の次の文から実行するためのものです。次の例ではversionの評価結果は"ES6"となるため、case "ES6":に続く文が実行されます。

```
const version = "ES6";
switch (version) {
  case "ES5":
    console.log("ECMAScript 5");
    break;
  case "ES6":
    console.log("ECMAScript 2015");
    break;
  case "ES7":
    console.log("ECMAScript 2016");
    break;
  default:
    console.log("知らないバージョンです");
    break;
}
// "ECMAScript 2015"と出力される
```

これはif文で次のように書いた場合と同じ結果になります。

```
const version = "ES6";
if (version === "ES5") {
  console.log("ECMAScript 5");
} else if (version === "ES6") {
  console.log("ECMAScript 2015");
} else if (version === "ES7") {
  console.log("ECMAScript 2016");
} else {
  console.log("知らないバージョンです");
}
```

switch文はやや複雑な仕組みであるためどのように処理されているかを見ていきます。まずswitch(式)の式を評価します。

```
switch (式) {
  // case
```

次に式の評価結果に一致するラベルを探索します。一致するラベルが存在する場合は、そのcase節を実行します。一致するラベルが存在しない場合は、default節が実行されます。

```
switch (式) {
  // if (式 === "ラベル1")
  case "ラベル1":
    break;
  // else if (式 === "ラベル2")
  case "ラベル2":
    break;
  // else
  default:
    break;
}
```

## break文

switch文のcase節では基本的に `break;` を使いswitch文を抜けるようにします。この `break;` は省略が可能ですが、省略した場合、後ろに続くcase節が条件に関係なく実行されます。

```
const version = "ES6";
switch (version) {
    case "ES5":
        console.log("ECMAScript 5");
    case "ES6": // 一致するケース
        console.log("ECMAScript 2015");
    case "ES7": // breakされないため条件無視して実行
        console.log("ECMAScript 2016");
    default: // breakされないため条件無視して実行
        console.log("しらないバージョンです");
}
/*
"ECMAScript 2015"
"ECMAScript 2016"
"しらないバージョンです"
と出力される
*/
```

このように `break;` を忘れてしまうと意図しない挙動となります。そのため、case節とbreak文が多用されているswitch文が出てきた場合、別の方で書けないかを考えるべきサインとなります。

一般にswitch文はif文の代用として使うのではなく、関数と組み合わせて値を返すパターンとして使うことが多いです。

```
function getECMAScriptName(version) {
    switch (version) {
        case "ES5":
            return "ECMAScript 5";
        case "ES6":
            return "ECMAScript 2015";
        case "ES7":
            return "ECMAScript 2016";
        default:
            return "しらないバージョンです";
    }
}
// 関数を実行して`return`された値を得る
getECMAScriptName("ES6"); // => "ECMAScript 2015"
```

関数については、n章で詳しく解説します。

□ 関数の章を書いたらn章を変更する

## 参考

- 関年 - Wikipedia
- C言語入門：うるう年判定プログラム:Geekなページ
- どうしてこんなキーワードがあるの？ - あどけない話
- switch - JavaScript | MDN
- 制御フローとエラー処理 - JavaScript | MDN

# ループと反復処理

プログラミングにおいて、同じ処理を繰り返すために同じコードを書く必要はありません。ループや再帰呼び出し、イテレータなどを使い、反復処理は抽象化します。ここでは、もっとも基本的な反復処理と制御文について学んでいきます。

## while文

while文は 条件式 が `true` であるならば、反復処理を行います。

```
while (条件式)
    実行する文;
```

while文の実行フローは次のようにになります。最初から 条件式 が `false` である場合は、何も実行せずwhile文は終了します。

1. 条件式 の評価結果が `true` なら処理を続け、 `false` なら終了
2. 実行する文 を実行
3. ステップ1へ戻る

次のコードでは `x` の値が10未満であるなら、コンソールへ繰り返しログが出力されます。また、 実行する文 にて、 `x` の値を増やし 条件式 が `false` となるようにしています。

```
let x = 0;
while (x < 10) {
    console.log(x);
    x += 1;
}
```

つまり、 実行する文 の中で 条件式 が `false` となるような処理を書かないと無限ループします。JavaScriptにはより安全な反復処理の書き方があるため、while文は使う場面が限られています。

安易にwhile文を使うよりも、他の書き方で解決できないかを考えてからでも遅くはないでしょう。

## do-while文

do-while文はwhile文と殆ど同じですが実行順序が異なります。

```
do {
    実行する文;
} while (条件式);
```

do-while文の実行フローは次のようにになります。

1. 実行する文 を実行
2. 条件式 の評価結果が `true` なら処理を続け、 `false` なら終了
3. ステップ1へ戻る

while文とは異なり、かならず最初に 実行する文 を処理します。

そのため、次のコードのように最初から 条件式 を満たさない場合でも、初回の 実行する文 が処理され、コンソールへ `1000` と出力されます。

```
const x = 1000;
do {
    console.log(x); // => 1000
} while (x < 10);
```

この仕組みを上手く利用し、ループの開始前とループ中の処理をまとめて書くことができます。しかし、while文と同じく他の書き方で解決できないかを考えてからでも遅くはないでしょう。

## for文

for文は繰り返す範囲を指定した反復処理を書くことができます。

```
for (初期化式; 条件式; 増分式)
    実行する文;
```

for文の実行フローは次のようにになります。

1. 初期化式 で変数の宣言
2. 条件式 の評価結果が `true` なら処理を続け、`false` なら終了
3. 実行する文 を実行
  - 複数行である場合は、`{` と `}` で囲んだブロック文にする必要があります
4. 増分式 で変数を更新
5. ステップ2へ戻る

次のコードでは、for文を使い1から10の合計値を計算しています。

```
let total = 0; // 初期値は0
for (let i = 0; i < 10; i++) {
    total += i + 1; // 1...10
}
console.log(total); // => 55
```

このコードは1から10の合計を電卓で計算すればいいので、普通は必要ありませんね。実際に扱うなら、数値の入った配列を受け取り、その合計を計算して返すという関数を実装することになります。

次のコードでは、任意の数値が入った配列を受け取り、その合計値を返す `sum` 関数を実装しています。

```
function sum(numbers) {
    let total = 0;
    for (let i = 0; i < numbers.length; i++) {
        total += numbers[i];
    }
    return total;
}

sum([1, 2, 3, 4, 5]); // => 15
```

反復処理の多くは、配列に入れた値を処理する方法と言いかえることができます。そのため、JavaScriptの配列である `Array` オブジェクトには反復処理のためのメソッドが備わっています。

## Array#forEach

`Array` オブジェクトは、`map`、`reduce` などの反復処理のためのメソッドが用意されています。`forEach` メソッドもそのひとつでfor文に近い反復処理を行います。

`forEach` メソッドは次のように書くことができます。

```
const array = [1, 2, 3, 4, 5];
array.forEach((currentValue, index, array) => {
  // 実行する文
});
```

JavaScriptでは、関数はファーストクラスであるため、その場で作った匿名関数（名前のない関数）を引数として渡すことができます。

引数として渡される関数のことをコールバック関数と呼びます。また、`forEach` メソッドのようなコールバック関数を引数として受け取る関数やメソッドのことを高階関数と呼びます。

```
const array = [1, 2, 3, 4, 5];
array.forEach(コールバック関数);
```

`forEach` メソッドのコールバック関数には、配列の先頭から順番に要素が渡されて実行されます。つまり、コールバック関数の `currentValue` には1から5の値が順番に渡されて実行されます。

```
[1, 2, 3].forEach(currentValue => {
  console.log(currentValue);
});
// 1
// 2
// 3
// と順番に出力される
```

先ほどのfor文で合計値を計算する `sum` 関数を `forEach` メソッドで書いてみます。

```
function sum(numbers) {
  let total = 0;
  numbers.forEach(num => {
    total += num;
  });
  return total;
}

sum([1, 2, 3, 4, 5]); // => 15
```

`forEach` は `条件式` がなく、配列のすべての要素を走査するため、for文よりもシンプルな処理です。

## break文

break文は処理中の文から抜けて次の文へ移行する制御文です。while、do-while、forの中で使い、処理中のループを抜けて次の文へ制御を移します。

```
while (true) {
  break; // *1 へ
}
// *1 次の文
```

switch文で出てきたものと同様で、処理中のループ文を終了できます。

次のコードでは配列の要素に1つでも偶数を含んでいるかを判定しています。

```

const numbers = [1, 5, 10, 15, 20];
// 偶数があるかどうか
let isEvenIncluded = false;
for (let i = 0; i < numbers.length; i++) {
  const number = numbers[i];
  if (number % 2 === 0) {
    isEvenIncluded = true;
    break;
  }
}
console.log(isEvenIncluded); // => true

```

1つでも偶数があるかが分かればいいため、配列内から最初の偶数を見つけたらfor文での反復処理を終了します。このような処理はベタ書きせずに、関数として実装するのが一般的です。

同様の処理を行う `isEvenIncluded` 関数を実装してみます。次のコードでは、break文が実行され、ループを抜けた後にreturn文で結果を返しています。

```

// `number`が偶数ならtrueを返す
function isEven(number) {
  return number % 2 === 0;
}
// `numbers`に偶数が含まれているならtrueを返す
function isEvenIncluded(numbers) {
  let isEvenIncluded = false;
  for (let i = 0; i < numbers.length; i++) {
    const number = numbers[i];
    if (isEven(number)) {
      isEvenIncluded = true;
      break;
    }
  }
  return isEvenIncluded;
}
const array = [1, 5, 10, 15, 20];
console.log(isEvenIncluded(array)); // => true

```

return文は現在の関数を終了させることができます。次のように書くこともできます。

```

function isEven(number) {
  return number % 2 === 0;
}
function isEvenIncluded(numbers) {
  for (let i = 0; i < numbers.length; i++) {
    const number = numbers[i];
    if (isEven(number)) {
      return true;
    }
  }
  return false;
}
const numbers = [1, 5, 10, 15, 20];
console.log(isEvenIncluded(numbers)); // => true

```

偶数を見つけたらすぐにreturnすることで一時変数が不要となり、より簡潔に書くことができます。

## Array#some

先ほどの `isEvenIncluded` は、偶数を見つけたら `true` を返す関数でした。 `Array` オブジェクトでは、`some` メソッドで同様のことが行えます。

`some` メソッドは、配列の各要素をテストする処理をコールバック関数として渡します。コールバック関数が一度でも `true` を返した時点で反復処理を終了し、`some` メソッドは `true` を返します。

```
const array = [1, 2, 3, 4, 5];
const isPassed = array.some((currentValue, index, array) => {
  // テストをパスするtrue、そうでないならfalseを返す
});
```

`some` メソッドを使うことで、配列に偶数が含まれているかは次のように書くことができます。受け取った値が偶数であるかをテストするコールバック関数として `isEven` 関数を渡します。

```
function isEven(number) {
  return number % 2 === 0;
}
const numbers = [1, 5, 10, 15, 20];
console.log(numbers.some(isEven)); // => true
```

## continue文

`continue`文は処理中の文をスキップして、そのループの 条件式 と移行する制御文です。`while`、`do-while`、`for`の中で使い、実行中のループの 条件式 へ制御を移します。

```
while (条件式) {
  continue; // `条件式` へ
}
```

次のコードでは、配列の中から偶数を集め、新しい配列を作り返しています。偶数ではない場合、処理中の`for`文をスキップしています。

```
// `number`が偶数ならtrueを返す
function isEven(number) {
  return number % 2 === 0;
}
// `numbers`に含まれている偶数だけを取り出す
function filterEven(numbers) {
  const results = [];
  for (let i = 0; i < numbers.length; i++) {
    const number = numbers[i];
    // 偶数ではないなら、次のループへ
    if (!isEven(number)) {
      continue;
    }
    // 偶数を`results`に追加
    results.push(number);
  }
  return results;
}
const array = [1, 5, 10, 15, 20];
console.log(filterEven(array)); // => [10, 20]
```

もちろん次のように、偶数なら `results` へ追加するという書き方も可能です。

```
if (isEven(number)) {
  results.push(number);
}
```

この場合、条件が複雑になってきた場合にネストが深くなつてコードが読みにくくなります。そのため、[ネストしたif文](#)のうるう年の例でも紹介したように、できるだけ早い段階でそれ以上処理を続けない宣言をすることで、複雑なコードになることを避けています。

## Array#filter

配列から特定の値だけを集めた新しい配列を作るには `filter` メソッドを利用できます。

`filter` メソッドには、配列の各要素をテストする処理をコールバック関数として渡します。コールバック関数が `true` を返した要素のみを集めた新しい配列を返します。

```
const array = [1, 2, 3, 4, 5];
// テストをパスしたものを集めた配列
const filteredArray = array.filter((currentValue, index, array) => {
    // テストをパスするならtrue、そうでないならfalseを返す
});
```

この `filter` メソッドを使うことで、次のように偶数を取り出す処理を書くことができます。

```
function isEven(number) {
    return number % 2 === 0;
}

const array = [1, 5, 10, 15, 20];
console.log(array.filter(isEven)); // => [10, 20]
```

## for...in文

for...in文はオブジェクトのプロパティに対して、順不同で反復処理を行います。

```
for (variable in object)
    実行する文;
```

次のコードでは `object` のプロパティ名を `key` 変数に代入し反復処理をしています。`object` には、3つのプロパティ名があるため3回繰り返されます。

```
const object = {
    "a": 1,
    "b": 2,
    "c": 3
};
for (const key in object) {
    const value = object[key];
    console.log(`key:${key}, value:${value}`);
}
// "key:a, value:1"
// "key:b, value:2"
// "key:c, value:3"
```

オブジェクトに対する反復処理のためにfor...in文は有用に見えますが、多くの問題を持っています。

JavaScriptでは、オブジェクトは荷らかのオブジェクトを継承しています。for...in文は、対象となるオブジェクトのプロパティを列挙する場合、すべての親オブジェクトまで探索し列挙します。そのため、オブジェクト自身が持っていないプロパティも列挙されてしまうことがあります。

この仕組みをプロトタイプチェーンといいますが、詳しくは第n章で解説します。

安全にオブジェクトのプロパティを列挙するには、`Object.keys` メソッド、`Object.values` メソッド、`Object.entries` メソッドなどが利用できます。

先ほどの例である、オブジェクトのキーと値を列挙するコードは`for...in`文を使わずに書くことができます。  
`Object.keys` メソッドは `object` 自身がもつ列挙可能なプロパティ名の配列を返します。そのため`for...in`文とは違い、親オブジェクトのプロパティは列挙されません。

```
const object = {
  "a": 1,
  "b": 2,
  "c": 3
};

Object.keys(object).forEach(key => {
  const value = object[key];
  console.log(`key:${key}, value:${value}`);
});

// "key:a, value:1"
// "key:b, value:2"
// "key:c, value:3"
```

また、`for...in`文は配列に対しても利用できますが、こちらも期待した結果にはなりません。

次のコードでは、配列の要素が列挙されそうですが、実際には配列のプロパティ名が列挙されます。`for...in`文が列挙する配列オブジェクトのプロパティ名は、要素のインデックスを文字列化した"0"、"1"となるため、その文字列が `num` へと順番に代入されます。そのため、数値と文字列の加算が行われ、意図した結果にはなりません。

```
const numbers = [5, 10];
let total = 0;
for (const num in numbers) {
  total += num;
}
console.log(total); // => "001"
```

配列の内容に対して反復処理を行う場合は、`for`文や `forEach` メソッド、後述する`for...of`文を使うべきでしょう。

このように`for...in`文は正しく扱うのが難しいですが、代わりとなる手段が豊富にあります。そのため、`for...in`文を使うことよりも他の方法を考えた方がよいでしょう。

## [ES2015] `for...of`文

最後に`for...of`文についてです。

JavaScriptでは、`Symbol.iterator` という特別な名前のメソッドを実装したオブジェクトを`iterable`と呼びます。`iterable` オブジェクトは、`for...of`文で反復処理できます。

`iterable`については`generator`と密接な関係がありますが、ここでは反復処理時の動作が定義されたオブジェクトと認識していれば問題ありません。

`iterable`オブジェクトは反復処理時に次の返す値を定義しています。それに対して、`for...of`文では、`iterable` から値を1つ取り出し、`variable` に代入し反復処理を行います。

```
for (variable of iterable)
  実行する文;
```

実はすでに`iterable`オブジェクトは登場していて、`Array`は`iterable`オブジェクトです。

次のように`for...of`文で、配列から値を取り出し反復処理を行うことができます。`for...in`文とは異なり、インデックス値ではなく配列の値を列挙します。

```
const array = [1, 2, 3];
for (const value of array) {
    console.log(value);
}
// 1
// 2
// 3
```

JavaScriptではStringオブジェクトもiterableです。そのため、文字列を1文字ずつ列挙できます。

```
const string = "吉野家";
for (const value of string) {
    console.log(value);
}
// "吉"
// "野"
// "家"
```

その他にも、`TypedArray`、`Map`、`Set`、DOM NodeListなど、iterableなオブジェクトとして実装されているものは多いです。`for...of`文はそれらに対して反復処理を行うことができます。

## [コラム] `let` ではなく `const` で反復処理をする

先ほどのfor文や`forEach`メソッドでは`let`を`const`に変更することはできませんでした。なぜなら、for文は一度定義した変数に値の代入を繰り返し行う処理といえるからです。`const`は再代入できない変数を宣言するキーワードであるためfor文とは相性がよくありません。

一度定義した変数に値を代入しつつ反復処理すると、変数へ値の上書きが必要となり`const`を使うことができません。そのため、一時的な変数を定義せずに反復処理した結果だけを受け取る方法が必要になります。

反復処理により新しい値を作るArrayメソッドとして`Array#reduce`メソッドがあります。

`reduce`メソッドは2つずつ要素を取り出し（左から右へ）、その値をコールバック関数に適用し、`次の値`として1つの値を返します。最終的な、`reduce`メソッドの返り値は、コールバック関数が最後に`return`した値となります。

```
const result = array.reduce((前回の値, 現在の値) => {
    return 次の値;
}, 初期値);
```

配列から合計値を返すものを`reduce`メソッドを使い実装してみましょう。

先ほどの配列の全要素の合計値を計算するものは`reduce`メソッドでは、次のように書くことができます。`初期値`に`0`を指定し、`前回の値`と`現在の値`を足していくことで合計を計算できます。`初期値`を指定していた場合は、最初の`前回の値`に初期値が、配列の先頭の値が`現在の値`となった状態で開始されます。

```
function sum(numbers) {
    return numbers.reduce((total, num) => {
        return total + num;
    }, 0); // 初期値が0
}

sum([1, 2, 3, 4, 5]); // => 15
```

`reduce`メソッドを使った例では、そもそも変数宣言をしていないことが分かります。`reduce`メソッドでは常に新しい値を返すことで、1つの変数の値を更新していく必要があります。これは`const`と同じく、一度作った変数の値を変更しないため、意図しない変数の更新を避けることにつながります。



# オブジェクト

## オブジェクトとは

オブジェクトはプロパティの集合です。プロパティとはキー（名前）と値から構成されるものを言います。キーには文字列またはSymbolが利用でき、値には任意のデータが利用できます。

オブジェクトを作成するには、オブジェクトリテラル（{}）を利用します。

```
// プロパティをもたない空のオブジェクトを作成
const object = {};
```

オブジェクトリテラルでは、任意のプロパティをもつオブジェクトを作成できます。プロパティは、オブジェクトリテラル（{}）の中にキーと値を : (コロン) で区切り記述します。

```
// プロパティをもつオブジェクトを定義する
const object = {
  // キー: 値
  "key": "value"
};
```

オブジェクトリテラルのプロパティ名（キー）はクオート（" や ' ）を省略できます。そのため、次のように書いても同じです。

```
// プロパティ名（キー）はクオートを省略することが可能
const object = {
  // キー: 値
  key: "value"
};
```

オブジェクトリテラルでは複数のプロパティ（キーと値の組み合わせ）をもつオブジェクトも作成できます。複数のプロパティを定義するには、それぞれのプロパティを , (カンマ) で区切ります。

```
const color = {
  // それぞれのキーと値の組み合わせを , で区切る
  red: "red",
  green: "green",
  blue: "blue",
};
```

プロパティの値に変数名を指定すれば、そのキーは指定した変数を参照します。

```
const name = "名前";
// `name`プロパティ
const object = {
  name: name
};
console.log(object); // => { name: "名前" }
```

またES2015からは、プロパティ名と値に指定する変数名が同じ場合は { name } のように省略して書くことができます。

```
const name = "名前";
// `name`というプロパティ名で`name`の変数を値に設定
```

```
const object = {
  name
};

console.log(object); // => { name: "名前" }
```

この省略記法は、モジュールや分割代入においても共通した表現です。そのため、`{}` の中にプロパティ名が単独で書かれてる場合は、この省略記法を利用していることに注意してください。

## プロパティへのアクセス

オブジェクトのプロパティにアクセスする方法として、ドット記法（`.`）を使う方法とブラケット記法（`[]`）があります。

```
const object = {
  key: "value"
};

// ドット記法で参照
console.log(object.key); // => "value"
// ブラケット記法で参照
console.log(object["key"]); // => "value"
```

ドット記法（`.`）では、プロパティ名が変数名と同じく識別子の命名規則を満たす必要があります。（詳細は[変数と宣言](#)の章を参照）

```
object.key; // OK!
// プロパティ名が数字から始まる識別子は利用できない
object.123; // NG!
```

一方、ブラケット記法では、`[` と `]` の間に任意の式を書くことができます。その式の評価結果をプロパティ名として利用できるため、プロパティ名に任意の文字列や変数を利用できます。

次のコードでは、プロパティ名に `myLang` という変数をブラケット記法で指定しています。ドット記法ではプロパティ名に変数は利用できないため、プロパティ名に変数を指定した場合はブラケット記法を利用します。

```
const languages = {
  ja: "日本語",
  en: "英語"
};

const myLang = "ja";
console.log(languages[myLang]); // => "日本語"
```

基本的にはドット記法（`.`）を使い、ドット記法で書けない場合はブラケット記法（`[]`）を使うといいでしょう。

## オブジェクトと分割代入

同じオブジェクトのプロパティを何度もアクセスする場合に、何度も `オブジェクト.プロパティ名` と書くと冗長となりやすいです。そのため、短い名前で利用できよう、そのプロパティを変数として定義し直すことがあります。

次のコードでは、`languages` オブジェクトのプロパティをそれぞれ変数 `ja` と `en` と定義し直しています。

```
const languages = {
  ja: "日本語",
  en: "英語"
};

const ja = languages.ja;
const en = languages.en;
```

```
console.log(ja); // => "日本語"
console.log(en); // => "英語"
```

このようなオブジェクトのプロパティを変数として定義し直すときには、分割代入（Destructuring assignment）が利用できます。

オブジェクトの分割代入では、左辺にオブジェクトリテラルのような構文で変数名を定義します。右辺のオブジェクトから対応するプロパティ名が、左辺で定義した変数に代入されます。

次のコードでは、先程のコードと同じように `languages` オブジェクトから `ja` と `en` プロパティを取り出して変数として定義しています。代入演算子のオペランドとして左辺と右辺それぞれに `ja` と `en` と書いていたのが、一度で書くだけで済むため短く書けます。

```
const languages = {
  ja: "日本語",
  en: "英語"
};
const { ja, en } = languages;
console.log(ja); // => "日本語"
console.log(en); // => "英語"
```

## プロパティの追加

オブジェクトは、一度作成した後もその値 자체を変更できるためミュータブル（mutable）の特性を持ちます。そのため、作成したオブジェクトに対して、後からプロパティを追加できます。

プロパティの追加方法は単純で、作成したいプロパティ名へ値を代入するだけです。そのとき、オブジェクトに指定したプロパティが存在しないなら、自動的にプロパティが作成されます。

プロパティの追加はドット記法、ブラケット記法どちらでも可能です。

```
// 空のオブジェクト
const object = {};
// `key`プロパティを追加し値を代入
object.key = "value";
console.log(object.key); // => "value"
```

先ほども紹介したように、ドット記法は変数の識別子として利用可能なプロパティ名しか利用できません。一方、ブラケット記法は `object[式]` の式の評価結果を文字列にしたものを利用できます。そのため、次のものをプロパティ名として扱う場合にはブラケット記法を利用します。

- 変数
- 識別子として書けない文字列
- Symbol

```
const key = "key-string";
const object = {};
// `key`の評価結果 "key-string" をプロパティ名に利用
object[key] = "value of key";
// 取り出すときも同じく `key`変数を利用
console.log(object[key]); // => "value of key"
// Symbolは例外的に文字列化されず扱える
const symbolKey = Symbol("シンボルは一意な値");
object[symbolKey] = "value of symbol";
console.log(object[symbolKey]); // => "value of symbol"
```

ブラケット記法を用いたプロパティ定義は、オブジェクトリテラルの中でも利用できます。オブジェクトリテラル内でのブラケット記法を使ったプロパティ名はComputed property namesと呼ばれます。Computed property namesはES2015から導入された記法ですが、式の評価結果をプロパティ名に使う点はブラケット記法と同じです。

```
const key = "key-string";
// Computed Propertyでプロパティを定義する
const object = {
  [key]: "value"
};
console.log(object[key]); // => "value"
```

JavaScriptのオブジェクトは、変更不可能と明示しない限り、変更可能なmutableの特性をもつことを紹介しました。そのため、関数が受け取ったオブジェクトに対して、勝手にプロパティを追加することもできてしまいます。

```
function doSomething(object) {
  object.key = "value";
  // 色々な処理...
}
const object = {};
doSomething(object); // objectが変更されている
console.log(object.key); // => "value"
```

このように、プロパティを初期化時以外に追加してしまうと、そのオブジェクトがどのようなプロパティを持っているかがわかりにくくなります。そのため、できる限り作成後に新しいプロパティの追加を行わないほうがよいでしょう。つまり、オブジェクトの作成時のオブジェクトリテラルの中でプロパティを定義することを推奨します。

## [コラム] constで定義したオブジェクトは変更可能

先ほどのコード例で、`const` で宣言したオブジェクトのプロパティがエラーなく変更できていることが分かります。次のコードを実行してみると、値であるオブジェクトのプロパティが変更できていることが分かります。

```
const object = { key: "value" };
object.key = "Hi!"; // constで定義したobjectが変更できる
console.log(object.key); // => "Hi!"
```

これは、JavaScriptの`const` は値を固定するのではなく、変数への再代入を防ぐためのものです。そのため、次のような`object`変数への再代入は防ぐことができますが、変数に代入された値であるオブジェクトの変更は防ぐことができません。（「[変数と宣言のconstについて](#)」を参照）

```
const object = { key: "value" };
object = {}; // => SyntaxError
```

作成したオブジェクトのプロパティの変更を防止するには`Object.freeze` メソッドを利用する必要があります。ただし、strict modeでないと例外が発生せず、無言で変更を無視するだけとなります。そのため、`Object.freeze` メソッドを利用する場合は必ずstrict modeと合わせて使います。

```
"use strict";
const object = Object.freeze({ key: "value" });
// freezeしたオブジェクトにプロパティを追加や変更できない
object.key = "value"; // => TypeError
```

## プロパティの存在を確認する

JavaScriptでは、存在しないプロパティに対してアクセスした場合に例外ではなく `undefined` を返します。次のコードでは、`object` には存在しない `notFound` プロパティにアクセスしているため、`undefined` という値が返ってきます。

```
const object = {};
console.log(object.notFound); // => undefined
```

このように、JavaScriptでは存在しないプロパティへアクセスした場合に例外が発生しません。プロパティ名を間違えた場合に単に `undefined` という値を返すため、気づきにくいという問題があります。

次のようにプロパティ名を間違えていた場合にも、例外が発生しないめ気づきにくいという問題が起きやすいです。さらにプロパティ名をネストしてアクセスした場合に、初めて例外が発生します。

```
const widget = {
  window: {
    title: "ウィジエットのタイトル"
  }
};
// `window`を`windw`と間違えているが、例外は発生しない
console.log(widget.windw); // => undefined
// さらにネストした場合に、例外が発生する
// `undefined.title`と書いたのと同じ意味となるため
console.log(widget.windw.title); // => TypeError: widget.windw is undefined
// 例外が発生した文以降は実行されない
```

`undefined` や `null` はオブジェクトではないため、存在しないプロパティへアクセスする例外が発生してしまいます。このような場合に、あるオブジェクトがあるプロパティを持っているを確認する方法がいくつかあります。

## undefinedとの比較

存在しないプロパティへアクセスした場合に、`undefined` を返すため実際にアクセスして比較することでも判定できます。

```
const object = { key: "value" };
// `key`プロパティが`undefined`ではないなら、プロパティが存在する?
if (object.key !== undefined) {
  console.log("`key`プロパティの値は`undefined`");
}
```

しかし、この方法はプロパティの値が `undefined` であった場合に、プロパティがないから `undefined` なのかが区別できないという問題があります。次のような例は、`key` プロパティは存在していますが、値が `undefined` であるため、存在の判定が上手くできていないことがわかります。

```
const object = { key: undefined };
// `key`プロパティの値が`undefined`
if (object.key !== undefined) {
  // 実行されない文
}
```

## in演算子を使う

`in` 演算子は、指定したオブジェクト上に指定したプロパティがあるかを判定できます。

```
"プロパティ名" in オブジェクト; // true or false
```

次のように、`object` に `key` プロパティが存在するなら、`true` を返します。

```
const object = { key: undefined };
// `key`プロパティを持っているならtrue
if ("key" in object) {
  console.log("`key`プロパティは存在する");
}
```

しかし、`in`演算子は、`for...in`文と同じく、対象となるオブジェクトのプロパティを列挙する場合、親オブジェクトまで探索し列挙します。そのため、`object`自身が持っていないなくても、親オブジェクトが持っているならば`true`を返します。

`object`自身がそのプロパティを持っているかを判定するには、`Object#hasOwnProperty`メソッドを使うのが確実です。

## Object#hasOwnProperty メソッド

`Object#hasOwnProperty`メソッドを使うことで、オブジェクト自身が指定したプロパティを持っているかを判定できます。

```
オブジェクト.hasOwnProperty("プロパティ名"); // true or false
```

`hasOwnProperty`メソッドは引数に存在を判定したいプロパティ名を渡し、該当するプロパティを持っている場合は`true`を返します。

```
const object = { key: "value" };
// `object`が`key`プロパティを持っているならtrue
if (object.hasOwnProperty("key")) {
  console.log("`object`は`key`プロパティを持っている");
}
```

## Object#toString メソッド

`Object#toString`メソッドは、オブジェクト自身を文字列化するメソッドです。`String`コンストラクタ関数を使うことでも文字列にすることができますが、どのような違いがあるのでしょうか？（「暗黙的な型変換」を参照）

実は`String`コンストラクタ関数は、引数に渡されたオブジェクトの`toString`メソッドを呼び出しています。そのため、`String`コンストラクタ関数と`toString`メソッドの結果はどちらも同じになります。

```
const object = { key: "value" };
console.log(object.toString()); // => "[object Object]"
// `String`コンストラクタ関数は`toString`メソッドを呼んでいる
console.log(String(object)); // => "[object Object]"
```

このことは、オブジェクトに`toString`メソッドを再定義してみると分かります。独自の`toString`メソッドを定義したオブジェクトを`String`コンストラクタ関数で文字列化してみます。すると、再定義した`toString`メソッドの返り値が、`String`コンストラクタ関数の返り値になることが分かります。

```
// 独自のtoStringメソッドを定義
const customObject = {
  toString() {
    return "value";
  }
};
console.log(String(customObject)); // => "value"
```

`Object` 以外の `Array` や `Number` などもそれぞれ独自の `toString` メソッドを定義しています。そのため、それぞれのオブジェクトで `toString` メソッドの結果は異なります。

```
const number = [1, 2, 3];
// Array#toStringが定義されているため、`Object#toString`とは異なる形式となる
console.log(number.toString()); // => "1,2,3";
```

## オブジェクトの静的メソッド

最後に `Object` の静的メソッドについて見てきましょう。

### オブジェクトの列挙

オブジェクトはプロパティの集合です。そのオブジェクトのプロパティを列挙する方法として、`Object.keys` メソッド、`Object.values` メソッド、`Object.entries` メソッドがあります。これらのメソッドは、そのオブジェクト自身がもつ列挙可能なプロパティだけを扱います。（「[ループと反復処理](#)」を参照）

それぞれ、オブジェクトのキー、値、キーと値の組み合わせを配列にして返します。

```
const object = {
  "one": 1,
  "two": 2,
  "three": 3
};
// `Object.keys`はキーの列挙した配列を返す
console.log(Object.keys(object)); // => ["one", "two", "three"]
// `Object.values` (ES2017) は値を列挙した配列を返す
console.log(Object.values(object)); // => [1, 2, 3]
// `Object.entries` (ES2017) は[キー, 値]の配列を返す
console.log(Object.entries(object)); // => [["one", 1], ["two", 2], ["three", 3]]
```

## オブジェクトのコピー/マージ

`Object.assign` を使うことで、あるオブジェクトを別のオブジェクトに代入（assign）できます。これを使うことでオブジェクトのコピーやオブジェクト同士のマージを行うできます。

`Object.assign` メソッドは、`target` オブジェクトに対して、1つ以上の `sources` オブジェクトを指定します。  
`sources` オブジェクト自身がもつ列挙可能なプロパティを第一引数の `target` オブジェクトに対してコピーします。  
`Object.assign` メソッドの返り値は、`target` オブジェクトになります。

```
Object.assign(target, ...sources);
```

## オブジェクトのマージ

具体的なオブジェクトのマージの例を見てきます。

次のコードでは、新しく作った空のオブジェクトを `target` にしています。この `target` に対して、`objectA` と `objectB` をマージしたものが `Object.assign` メソッドの返り値となります。

```
const objectA = { a: "a" };
const objectB = { b: "b" };
const merged = Object.assign({}, objectA, objectB);
console.log(merged); // => { a: "a", b: "b" }
```

第一引数には空のオブジェクトではなく、既存のオブジェクトを指定することもできます。第一引数に既存のオブジェクトを指定した場合は、指定されたオブジェクトのプロパティが変更されます。

次のコードでは第一引数に指定された `objectA` に対してプロパティが追加されています。

```
const objectA = { a: "a" };
const objectB = { b: "b" };
const merged = Object.assign(objectA, objectB);
console.log(merged); // => { a: "a", b: "b" }
// `objectA`が変更されている
console.log(objectA); // => { a: "a", b: "b" }
console.log(merged === objectA); // => true
```

空のオブジェクトを `target` にすることで、既存のオブジェクトには影響を与えずマージしたオブジェクトを作ることができます。そのため、`Object.assign` メソッドの第一引数には、空のオブジェクトリテラルを指定するのが典型的な利用方法です。

このとき、プロパティ名が重複した場合は、後ろのオブジェクトのプロパティにより上書きされます。JavaScriptでは、基本的な処理は左から順番に行います。そのため左から順にオブジェクトが代入されていくと考えるとよいです。

```
// `version`のプロパティ名が被っている
const objectA = { version: "a" };
const objectB = { version: "b" };
const merged = Object.assign({}, objectA, objectB);
// 後ろにある`objectB`のプロパティで上書きされる
console.log(merged); // => { version: "b" }
```

## オブジェクトのspread構文

ES2018ではオブジェクトのマージを構文として行える `...` (spread構文) が追加されました。ES2015で配列の要素を展開する `...` (spread構文) はサポートされていましたが、オブジェクトに対してもES2018でサポートされました。オブジェクトのspread構文は、オブジェクトリテラルの中に指定したオブジェクトのプロパティを展開できます。

オブジェクトのspread構文は、`Object.assign` とは異なり必ず新しいオブジェクトを作成し返します。なぜならspread構文はオブジェクトリテラルの中でのみ記述でき、オブジェクトリテラルは新しいオブジェクトを返すためです。

次のコードでは `objectA` と `objectB` をマージした新しいオブジェクトを返します。

```
const objectA = { a: "a" };
const objectB = { b: "b" };
const merged = {
  ...objectA,
  ...objectB
};
console.log(merged); // => { a: "a", b: "b" }
```

プロパティ名が被った場合の優先順位は、後ろにあるプロパティほど優先されます。そのため同じプロパティ名をもつオブジェクトをマージした場合には、後ろにあるオブジェクトによってプロパティが上書きされます。

```
// `version`のプロパティ名が被っている
const objectA = { version: "a" };
const objectB = { version: "b" };
const merged = {
  ...objectA,
  ...objectB,
  other: "other"
};
```

```
// 後ろにある`objectB`のプロパティで上書きされる
console.log(merged); // => { version: "b", other: "other" }
```

## オブジェクトの複製

JavaScriptには、オブジェクトを複製する関数は用意されていません。しかし、新しく空のオブジェクトを作成し、そこへ既存のオブジェクトのプロパティをコピーすれば、それはオブジェクトの複製しているといえます。次のように、`Object.assign` メソッドを使うことでオブジェクトを複製できます。

```
// `object`を浅く複製したオブジェクトを返す
const shallowClone = (object) => {
  return Object.assign({}, object);
};

const object = { a: "a" };
const cloneObject = shallowClone(object);
console.log(cloneObject); // => { a: "a" }
console.log(object === cloneObject); // => false
```

注意点として、`Object.assign` メソッドは `sources` オブジェクトのプロパティを浅くコピー（shallow copy）する点です。`sources` オブジェクト自身が持っている列挙できるプロパティをコピーするだけです。そのプロパティの値がオブジェクトである場合に、そのオブジェクトまでも複製するわけではありません。

```
const shallowClone = (object) => {
  return Object.assign({}, object);
};

const object = {
  level: 1,
  nest: {
    level: 2
  },
};
const cloneObject = shallowClone(object);
// `nest`オブジェクトは複製されていない
console.log(cloneObject.nest === object.nest); // => true
```

このような浅いコピーのことをshallow copyと呼び、逆にプロパティの値までも再帰的に複製してコピーすることを深いコピー（deep copy）と呼びます。shallowな実装を使い再帰的に処理することで、deepな実装を実現できます。次のコードでは、`shallowClone` を使い、`deepClone` を実現しています。

```
// `object`を浅く複製したオブジェクトを返す
const shallowClone = (object) => {
  return Object.assign({}, object);
};

// `object`を深く複製したオブジェクトを返す
function deepClone(object) {
  const newObject = shallowClone(object);
  // プロパティがオブジェクト型であるなら、再帰的に複製する
  Object.keys(newObject)
    .filter(k => typeof newObject[k] === "object")
    .forEach(k => newObject[k] = deepClone(newObject[k]));
  return newObject;
};

const object = {
  level: 1,
  nest: {
    level: 2
  }
};
const cloneObject = deepClone(object);
// `nest`オブジェクトも再帰的に複製されている
console.log(cloneObject.nest === object.nest); // => false
```

このように、JavaScriptのビルトインメソッドは浅い（shallow）な実装のみを提供し、深い（deep）な実装は提供していません。言語としては最低限の機能を提供し、より複雑な機能はユーザー側で実装するという形になることが多いです。

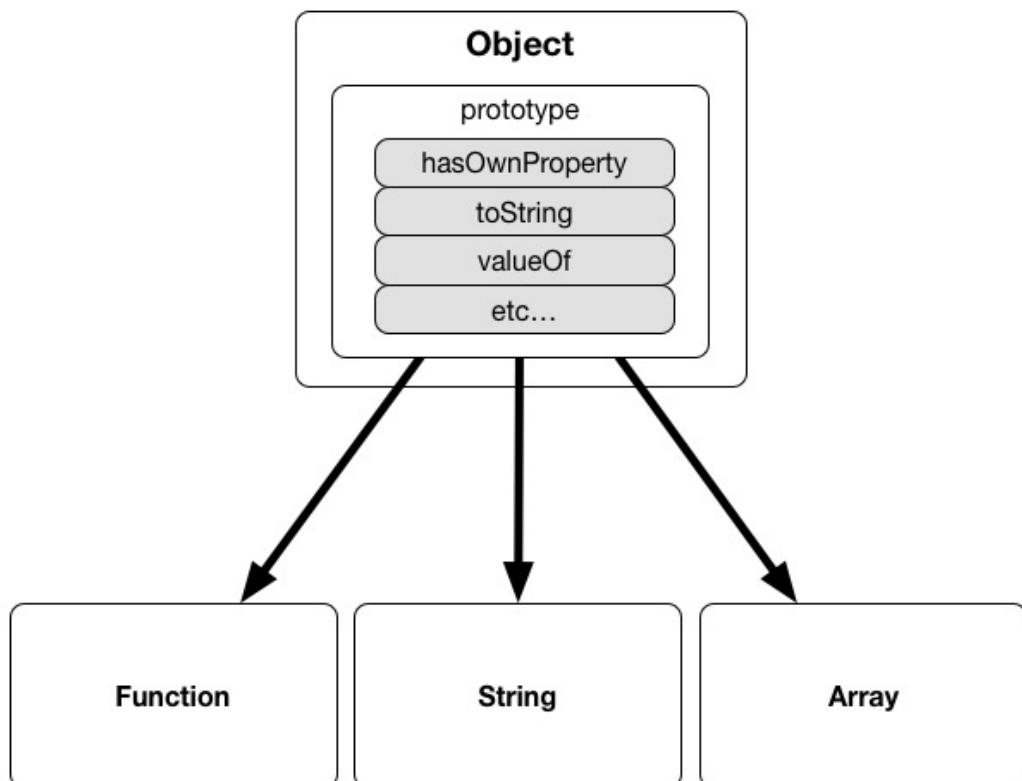
一方、JavaScriptという言語はコアにある機能が最低限であるため、ユーザーが作成した小さな機能をもつライブラリが数多く公開されています。それらのライブラリは `npm` と呼ばれるJavaScriptのパッケージ管理ツールで公開され、JavaScriptのエコシステムを築いています。

# プロトタイプオブジェクト

## Object はすべての元

ここまででは、`Object` 自身の機能について見てきましたが、`Object` には、他の `Array` や `String`、`Function` などの他のオブジェクトとは異なる特徴があります。それは、他のオブジェクトはすべて `Object` を継承しているという点です。

厳密には、すべてのオブジェクトは `Object` の `prototype` オブジェクトを継承しています。`prototype` オブジェクトはすべてのオブジェクトに備わっている特別なオブジェクトです。そのため、`Object` はすべてのオブジェクトが共通して利用できるプロパティやメソッドを提供するベースのオブジェクトともいえます。



具体的にどういうことかを見てみます。先ほども登場した、`Object#hasOwnProperty` メソッドは、`Object` の `prototype` オブジェクトに `hasOwnProperty` メソッドの定義があります。

```
// `Object`の`prototype`オブジェクトに`hasOwnProperty`メソッドの定義がある
console.log(typeof Object.prototype.hasOwnProperty); // => "function"
```

この `Object.prototype.hasOwnProperty` メソッドは、`Object` の `prototype` オブジェクトに定義されています。そのため、ほとんどのオブジェクトは `hasOwnProperty` メソッドを持っています。

```
// このような定義が自動的に行われているイメージ
// `Object`の`prototype`オブジェクトに`hasOwnProperty`メソッドの定義を行う
Object.prototype.hasOwnProperty = (propertyName) => {
  // hasOwnPropertyの処理
};
```

`Object` のインスタンスは、この `prototype` オブジェクトに定義されたメソッドやプロパティをインスタンス化時に継承します。つまり、オブジェクトリテラルや `new Object` でインスタンス化したオブジェクトは、`Object.prototype` に定義されたものが利用できるということです。

```
// var object = new Object()と同じ
const object = {};
// インスタンスがprototypeオブジェクトに定義されたものを継承する
console.log(object.hasOwnProperty === Object.prototype.hasOwnProperty); // => true
```

そのため、`Object.prototype` に定義されている `toString` メソッドや `hasOwnProperty` メソッドが、`Object` のインスタンスで利用できます。

`Object` のインスタンス -> `Object.prototype`

## in 演算子と `Object#hasOwnProperty` メソッドの違い

先ほど学んだ `in` 演算子と `Object#hasOwnProperty` メソッドの違いからもここから生じています。

`hasOwnProperty` メソッドは、そのオブジェクト自身が指定したプロパティを持っているかを判定します。一方、`in` 演算子はオブジェクト自身が持っていないければ、そのオブジェクトの親オブジェクトまで順番に探索して持っているかを判定します。

```
const object = {};
// `object`のインスタンス自体に`toString`メソッドが定義されているわけではない
console.log(object.hasOwnProperty("toString")); // => false
// `in`演算子は指定されたプロパティ名が見つかるまで親を辿るため、`Object.prototype`まで見に行く
console.log("toString" in object); // => true
```

これにより `Object` のインスタンス自身が `toString` メソッドを持っているわけではない、`Object.prototype` が `toString` メソッドを持っていることが分かります。

## オブジェクトの継承元を明示する `Object.create` メソッド

`Object.create` メソッドを使うと、第一引数に指定した `prototype` オブジェクトを継承した新しいオブジェクトを作成できます。

先ほど、オブジェクトリテラルは `Object.prototype` オブジェクトを自動的に継承したオブジェクトを作成していることがわかりました。オブジェクトリテラルで作成する新しいオブジェクトは、`Object.create` メソッドを使うことで次のように書くことができます。

```
// var object = {} 同じ
const object = Object.create(Object.prototype);
// `object`は`Object.prototype`を継承している
console.log(object.hasOwnProperty === Object.prototype.hasOwnProperty); // => true
```

## ArrayもObjectを継承している

`Object` と `Object.prototype` の関係と同じく、`Array` コンストラクタも `Array.prototype` を持っています。そのため、`Array` コンストラクタのインスタンスは `Array.prototype` を継承します。さらに、`Array.prototype` は `Object.prototype` を継承しているため、`Array` のインスタンスは `Object.prototype` も継承しているのです。

`Array` のインスタンス -> `Array.prototype` -> `Object.prototype`

`Object.create` メソッドを使って `Array` と `Object` の関係をコードとして表現してみます。`Array` コンストラクタの実装などは実際のものとは異なるので、あくまで関係の例示でしかないことに注意してください。

```
// `Array`コンストラクタ自身は関数でもある
const Array = function() {};
// `Array.prototype`は`Object.prototype`を継承している
Array.prototype = Object.create(Object.prototype);
// `Array`のインスタンスは、`Array.prototype`を継承している
const array = Object.create(Array.prototype);
// `array`は`Object.prototype`を継承している
console.log(array.hasOwnProperty === Object.prototype.hasOwnProperty); // => true
```

このように、`Array` のインスタンスも `Object.prototype` を継承しているため、`Object.prototype` に定義されているメソッドを利用できます。

```
// var array = new Array(); 同じ
var array = [];
// `Array`のインスタンス -> `Array.prototype` -> `Object.prototype`
console.log(array.hasOwnProperty === Object.prototype.hasOwnProperty); // => true
```

この継承の仕組みは、プロトタイプ継承と呼ばれるJavaScriptのコアとなる概念です。詳しくは、[クラス]Iの章で詳しく解説します。

ここでは、`Object` はすべてのオブジェクトの親となるオブジェクトであるだけを覚えておくだけで問題ありません。これにより、`Array` や `String` などのインスタンスも `Object.prototype` がもつメソッドを利用する点を覚えておきましょう。

## [コラム] `Object.prototype` を継承しないオブジェクト

`Object` はすべてのオブジェクトの親となるオブジェクトである言いましたが、例外もあります。

イディオムに近いのですが、`Object.create(null)` とすることで `Object.prototype` を継承しないオブジェクトを作ることができます。これにより、プロパティやメソッドをなどを全く持たない本当に空のオブジェクトを作ることができます。

```
// 親がnull、つまり親がないオブジェクトを作る
const object = Object.create(null);
// Object.prototypeを継承しないため、hasOwnPropertyが存在しない
console.log(object.hasOwnProperty); // => undefined
```

`Object.create` メソッドはES5から導入されました。`Object.create` メソッドは `Object.create(null)` というイディオムで、一部ライブラリなどで `Map` オブジェクトの代わりとして利用されています。`Object` のインスタンスはデフォルトで `Object.prototype` を継承するため、`toString` などのプロパティ名がオブジェクトを作成した時点で存在します。`Object.create(null)` をつかうことで `Object.prototype` を継承しないオブジェクトを作成できるため、何もプロパティをもたないオブジェクトを作成できます。

```
// ただのオブジェクト
const object = {};
// "toString"という値を定義していないのに、"toString"が存在している
```

```
console.log(object["toString"]); // Function
// Mapのようなオブジェクト
const mapLike = Object.create(null);
// toStringキーは存在しない
console.log(mapLike["toString"]); // => undefined
```

しかし、ES2015からは、本物の `Map` が利用できるため、`Object.create(null)` を `Map` の代わりに利用する必要はありません。

```
const map = new Map();
// toStringキーは存在しない
console.log(map.has("toString")); // => false
```

# 配列

この章では、配列の基本的な操作と配列を扱う場合においてのパターンについて学びます。配列はJavaScriptの中でもよく使われるオブジェクトです。

配列とは値に順序をつけて格納できるオブジェクトです。配列に格納したそれぞれの値のことを要素、それぞれの要素の位置のことをインデックス（`index`）と呼びます。インデックスは`0`、`1`、`2`のように`0`から始まる値となります。

またJavaScriptにおける配列は可変長です。そのため配列を作成後に配列へ要素を追加したり、配列から要素を削除できます。

## 配列は特別なオブジェクト

JavaScriptでは、プリミティブ型のデータ以外はすべてオブジェクトです。そのため、配列もオブジェクトの一種です。このことは`typeof`演算子の結果を見てみることでもわかります。

```
typeof ["A", "B", "C"]; // => "object"
```

しかし、`Object`のインスタンスにはない`Array#forEach`などのメソッドや特殊な動作を持っています。

その特殊な動作が`length`プロパティです。配列には複数の要素を格納できますが、`length`プロパティはその配列の要素数を返します。

```
const array = ["A", "B", "C"];
console.log(array.length); // => 3
```

また、`length`プロパティへ値を代入できます。`length`プロパティへ値を代入は配列の要素を削除することに利用されることがあります。配列の`length`プロパティは特殊な動作となっているため、後ほど解説します。

```
const array = ["A", "B", "C"];
array.length = 0; // 配列を空にする
console.log(array); // => []
```

## オブジェクトが配列かどうかを判定する

配列の`length`プロパティは特殊な動作をしますが、独自の`length`プロパティを持ったオブジェクトを作ることができます。この2つのオブジェクトの違いはどのように見分けなければならないのでしょうか？

```
// 配列
const array = [];
// `length`を持つオブジェクト
const object = {
  length: 0
};
```

先ほど示したように`typeof`ではオブジェクトと配列の区別は付きません。また、`length`プロパティが存在するかでは、それが配列であるとは判断できません。

そのため、あるオブジェクトが配列なのかを知りたい場合には、`Array.isArray`メソッドを利用します。

`Array.isArray`メソッドは引数が配列ならば`true`を返します。

```
const array = [];
console.log(Array.isArray(array)); // => true
// 配列のようなオブジェクト
const object = {
  length: 0
};
console.log(Array.isArray(object)); // => false
```

## [コラム] TypedArray

JavaScriptの配列は可変長のみですが、`TypedArray` という固定長でかつ型付きの配列を扱う別のオブジェクトが存在します。`TypedArray` はバイナリデータを扱うためのオブジェクトで、WebGLやバイナリを扱う場面で利用されます。`TypedArray` は文字列や数値などのプリミティブ型の値はそのままでは扱えないため、扱う値は`TypedArray`オブジェクトという形式にする必要があります。そのため、通常の配列とは異なる使い勝手や用途が存在します。

JavaScriptで配列といった場合には`Array`を示します。

## 配列の作成とアクセス

配列の作成と要素へのアクセス方法はデータ型とリテラルすでに紹介していますが、もう一度振り返ってみましょう。

配列の作成には配列リテラルを使うのが簡単です。配列リテラル（`[ ]`）の中に要素をカンマ（`,`）区切りで記述するだけです。

```
const emptyArray = [];
const numbers = [1, 2, 3];
const matrix = [
  [0, 1],
  [0, 1]
]; // 2次元配列
```

作成した配列の要素へインデックスとなる数値を、`配列[インデックス]`と記述することで、そのインデックスにある要素を配列から読み取ることができます。配列の先頭要素のインデックスは`0`となります。配列のインデックスは、`0`以上`2^32 - 1`未満の整数となります。

```
const array = ["one", "two", "three"];
console.log(array[0]); // => "one"
```

先ほど学んだように、配列の`length`プロパティは配列の要素の数を返します。そのため、配列の最後の要素へアクセスするには`array.length - 1`をインデックスとして指定します。

```
const array = ["one", "two", "three"];
console.log(array[array.length - 1]); // => "three"
```

一方、存在しないインデックスにアクセスした場合はどうなるでしょうか？JavaScriptでは、存在しないインデックスに対してアクセスした場合に例外ではなく`undefined`を返します。

```
const array = ["one", "two", "three"];
// `array`にはインデックスが100の要素は定義されていない
console.log(array[100]); // => undefined
```

これは、配列がオブジェクトであることを考えると、次のように存在しないプロパティへのアクセスと同じということが分かります。オブジェクトでも、存在しないプロパティへのアクセスした場合には`undefined`が返ってきます。

```

const object = {
  "0": "one",
  "1": "two",
  "2": "three",
  "length": 3
};
// object[100]はobject["100"]としてアクセスされる
// objectにはプロパティ名が"100"のものがないため、undefinedが返る
console.log(object[100]); // => undefined

```

また、配列は常に `length` の数だけ要素を持っているとは限りません。次のように、配列リテラルでは値を省略することで、未定義の要素を含めることができます。このような、配列の中に隙間があるものを疎な配列と呼びます。一方、隙間がなくすべてのインデックスに要素がある配列を密な配列と呼びます。

```

// 未定義の箇所が1つ含まれる疎な配列
// インデックスが1の値を省略しているので、カンマが2つ続いていることに注意
const sparseArray = [1,, 3];
console.log(sparseArray.length); // => 3
// 1番目の要素は存在しないため undefined が返る
console.log(sparseArray[1]); // => undefined

```

## 配列と分割代入

配列の指定したインデックスの値を変数として定義し直す場合には、分割代入（Destructuring assignment）が利用できます。

配列の分割代入では、左辺に配列リテラルのような構文で定義したい変数名を書きます。右辺の配列から対応するインデックスの要素が、左辺で定義した変数に代入されます。

次のコードでは、左辺に定義した変数に対して、右辺の配列から対応するインデックスの要素が代入されます。

`first` にはインデックスが `0` の要素、`second` にはインデックスが `1` の要素、`third` にはインデックスが `2` の要素が代入されます。

```

const array = ["one", "two", "three"];
const [first, second, third] = array;
console.log(first); // => "one"
console.log(second); // => "two"
console.log(third); // => "three"

```

## [コラム] `undefined`の要素と未定義の要素の違い

疎な配列で該当するインデックスに要素がない場合は `undefined` を返します。しかし、`undefined` という値も存在するため、配列に `undefined` という値がある場合に区別できません。

次のコードでは、`undefined` という値を要素として定義した密な配列と、要素そのものがない疎な配列を定義しています。どちらも要素にアクセスした結果は `undefined` となり、区別できていません。

```

// 要素として`undefined`を持つ密な配列
const denseArray = [1, undefined, 3];
// 要素そのものがない疎な配列
const sparseArray = [1, , 3];
console.log(denseArray[1]); // => undefined
console.log(sparseArray[1]); // => undefined

```

この違いを見つける方法として利用できるのが `Object#hasOwnProperty` メソッドです。 `hasOwnProperty` メソッドを使うことで、配列の指定したインデックスに要素自体が存在するかを判定できます。

```
const denseArray = [1, undefined, 3];
const sparseArray = [1, , 3];
// 要素自体は`undefined`値が存在する
console.log(denseArray.hasOwnProperty(1)); // => true
// 要素自体がない
console.log(sparseArray.hasOwnProperty(1)); // => false
```

## 配列から要素を検索

配列からある要素があるかを探索したい場合に、主に次の3つの目的に分類できます。

- その要素のインデックスが欲しい場合
- その要素自体が欲しい場合
- その要素が含まれているかという真偽値が欲しい場合

配列にはそれぞれに対応したメソッドが用意されているため、目的別に見ていきます。

### インデックスを取得

ある要素が配列のどの位置にあるかを知りたい場合、`Array#indexOf` メソッドや `Array#findIndex` メソッドを利用します。要素の位置のことをインデックス（`index`）と呼ぶため、メソッド名にも `index` という名前が入っています。

次のコードでは、`Array#indexOf` メソッドを利用して、配列の中から "JavaScript" という文字列のインデックスを取得しています。`indexOf` メソッドは引数と厳密等価演算子（`==`）で一致する要素があるなら、その要素のインデックスを返し、該当する要素がない場合は `-1` を返します。`indexOf` メソッドは先頭から探索して見つかった要素のインデックスを返します。`indexOf` メソッドには対となる `Array#lastIndexOf` メソッドがあり、`lastIndexOf` メソッドは末尾から探索した結果を得ることができます。

```
const array = ["Java", "JavaScript", "Ruby"];
const indexOfJS = array.indexOf("JavaScript");
console.log(indexOfJS); // => 1
console.log(array[indexOfJS]); // => "JavaScript"
// "JS" という要素はないため `-1` が返される
console.log(array.indexOf("JS")); // => -1
```

`indexOf` メソッドは配列からプリミティブな要素は発見できますが、オブジェクトは持っているプロパティが同じでも別オブジェクトだと異なるものとして扱われます。次のコードを見ると、同じプロパティをもつ異なるオブジェクトは、`indexOf` メソッドでは見つけることができません。これは、異なる参照をもつオブジェクト同士は `==` で比較しても一致しないためです。

```
const object = { key: "value" };
const array = ["A", "B", object];
console.log(array.indexOf({ key: "value" })); // => -1
// リテラルは新しいオブジェクトを作るため異なるオブジェクトを比較している
console.log(object === { key: "value" }); // => false
// 等価のオブジェクト
console.log(array.indexOf(object)); // => 2
```

このように、異なるオブジェクトだが値が同じものを見つけたい場合には、`Array#findIndex` メソッドが利用できます。`findIndex` メソッドは関数には配列の各要素をテストする関数をコールバック関数として渡します。

`indexOf` メソッドとは異なり、テストする処理を自由に書くことができます。これにより、異なるオブジェクトだ

が値が同じという要素を配列から見つけて、その要素のインデックスを得ることができます。

```
// colorプロパティを持つオブジェクトの配列
const colors = [
  { "color": "red" },
  { "color": "green" },
  { "color": "blue" }
];
// `color`プロパティが"blue"のオブジェクトのインデックスを取得
const indexOfBlue = colors.findIndex((object) => {
  return object.color === "blue";
});
console.log(indexOfBlue); // => 2
console.log(colors[indexOfBlue]); // => { "color": "blue" }
```

## 条件に一致する要素を取得

配列から要素を取得する方法としてインデックスを使うこともできます。先ほどのように `findIndex` メソッドでインデックスを取得、そのインデックスで配列へアクセスすればよいだけです。

しかし、`findIndex` メソッドを使い要素を取得するケースでは、そのインデックスが欲しいのか、またはその要素自体が欲しいのかがコードとして明確ではありません。

より明確に要素自体が欲しいということを表現するには、`Array#find` を使うことができます。`find` メソッドは、`findIndex` メソッドと同様にテストする関数をコールバック関数として渡します。`find` メソッドの返り値は、要素そのものとなり、要素が存在しない場合は `undefined` を返します。

```
// colorプロパティを持つオブジェクトの配列
const colors = [
  { "color": "red" },
  { "color": "green" },
  { "color": "blue" }
];
// `color`プロパティが"blue"のオブジェクトを取得
const blueColor = colors.find((object) => {
  return object.color === "blue";
});
console.log(blueColor); // => { "color": "blue" }
// 該当する要素がない場合は`undefined`を返す
console.log(colors.find((object) => object.color === "white")); // => undefined
```

## 指定範囲の要素を取得

配列から指定範囲の要素を取り出す方法として `Array#slice` メソッドが利用できます。`slice` メソッドは第一引数に開始位置、第二引数に終了位置を指定することで、その範囲を取り出した新しい配列を返します。第二引数は省略でき、省略した場合は配列の末尾が終了位置となります。

```
const array = ["A", "B", "C", "D", "E"];
// インデックス1から4の範囲を取り出す
console.log(array.slice(1, 4)); // => ["B", "C", "D"]
// 第二引数を省略した場合は、第一引数から末尾の要素までを取り出す
console.log(array.slice(1)); // => ["B", "C", "D", "E"]
// マイナスを指定すると後ろからの数えた位置となる
console.log(array.slice(-1)); // => ["E"]
// 第一引数 > 第二引数の場合、常に空配列を返す
console.log(array.slice(4, 1)); // => []
```

## 真偽値を取得

最後に、ある要素が配列に含まれているかを知る方法について見ていきます。インデックスや要素が取得できれば、その要素は配列に含まれているということは分かります。

しかし、ある要素が含まれているかだけを知りたい場合に、`Array#findIndex` メソッドや `Array#find` メソッドは過剰な機能を持っています。そのコードを読んだ人は取得したインデックスや要素を何に使うのかが明確ではありません。

次のコードは、`Array#indexOf` メソッドを利用し、該当する要素が含まれているかを判定しています。`indexOf` メソッドの結果を `indexOfJS` に代入していますが、含まれているかを判定する以外には利用していません。コードを隅々まで読まないといけないため、意図が明確ではなくコードの読みづらさにつながります。

```
const array = ["Java", "JavaScript", "Ruby"];
// `indexOf` メソッドは含まれていないときのみ`-1`を返すことを利用
const indexOfJS = array.indexOf("JavaScript");
if (indexOfJS !== -1) {
  console.log("配列にJavaScriptが含まれている");
  // ... 色々な処理 ...
  // `indexOfJS` は、含まれているのかの判定以外には利用していない
}
```

しかし、ES2015からは `Array#include` メソッドである要素が含まれているかを判定できます。`include` メソッドは真偽値を返すので、`indexOf` メソッドを使った場合に比べて意図が明確になります。そのため、前述のコードは次のように `include` メソッドを使うべきでしょう。

```
const array = ["Java", "JavaScript", "Ruby"];
// `include` は含まれているなら`true`を返す
if (array.includes("JavaScript")) {
  console.log("配列にJavaScriptが含まれている");
}
```

`include` メソッドは、`indexOf` メソッドと同様で、異なるオブジェクトだが値が同じものを見つける場合には利用できません。`Array#find` メソッドのようにテストするコールバック関数を利用して、真偽値を得るには `Array#some` メソッドを利用できます。

`Array#some` メソッドはテストする関数をコールバック関数にマッチする要素があるなら `true` を返し、存在しない場合は `false` を返します。（「ループと反復処理」の章）を参照）

```
// colorプロパティを持つオブジェクトの配列
const colors = [
  { "color": "red" },
  { "color": "green" },
  { "color": "blue" }
];
// `color` プロパティが"blue"のオブジェクトがあるかどうか
const isIncludedBlueColor = colors.some((object) => {
  return object.color === "blue";
});
console.log(isIncludedBlueColor); // => true
```

## 追加と削除

配列は可変長であるため、作成後の配列に対して要素を追加、削除できます。

要素を配列の末尾へ追加するには `Array#push` が利用できます。一方、末尾から要素を削除するには `Array#pop` が利用できます。

```
const array = ["A", "B", "C"];
```

```
array.push("D"); // "D"を末尾に追加
console.log(array); // => ["A", "B", "C", "D"]
const poppedItem = array.pop(); // 最末尾の要素を削除し、その要素を返す
console.log(poppedItem); // => "D"
console.log(array); // => ["A", "B", "C"]
```

要素を配列の先頭へ追加するには `Array#unshift` が利用できます。一方、配列の先頭から要素を削除するには `Array#shift` が利用できます。

```
const array = ["A", "B", "C"];
array.unshift("S"); // "S"を先頭に追加
console.log(array); // => ["S", "A", "B", "C"]
const shiftedItem = array.shift(); // 先頭の要素を削除
console.log(shiftedItem); // => "S"
console.log(array); // => ["A", "B", "C"]
```

## 配列同士を結合

`Array#concat` メソッドを使うことで配列と配列を結合した新しい配列を作成できます。

```
const array = ["A", "B", "C"];
const newArray = array.concat(["D", "E"]);
console.log(newArray); // => ["A", "B", "C", "D", "E"]
```

また、`concat` メソッドは配列だけではなく任意の値を要素として結合できます。

```
const array = ["A", "B", "C"];
const newArray = array.concat("新しい要素");
console.log(newArray); // => ["A", "B", "C", "新しい要素"]
```

## 配列から要素を削除

### Array#splice

配列の先頭や末尾の要素を削除する場合は `Array#shift` や `Array#pop` で行えます。しかし、配列の任意のインデックスにある要素を削除することはできません。配列の任意のインデックスの要素削除するには `Array#splice` を利用できます。

`Array#splice` メソッドを利用すると、削除した要素を自動で詰めることができます。`Array#splice` メソッドは、`index` 番目から `削除する数` だけ要素を取り除き、必要ならば要素を同時に追加できます。

```
const array = [];
array.splice(インデックス, 削除する要素数);
// 削除と同時に要素の追加もできる
array.splice(インデックス, 削除する要素数, ...追加する要素);
```

たとえば、配列のインデックスが `1` の要素を削除するには、インデックス `1` から `1` つの要素を削除するという指定をする必要があります。このとき、削除した要素は自動で詰められるため、疎な配列にはなりません。

```
const array = [1, 2, 3];
// 1番目から1つの要素を削除
array.splice(1, 1);
console.log(array); // => [1, 3]
console.log(array.length); // => 2
console.log(array[1]); // => 3
```

```
// すべて削除
array.splice(0, array.length);
console.log(array.length); // => 0
```

## length プロパティへの代入

配列のすべての要素を削除することは `Array#splice` で行うことができますが、配列の `length` プロパティへの代入を利用した方法もあります。

```
const array = [1, 2, 3];
array.length = 0; // 配列を空にする
console.log(array); // => []
```

配列の `length` プロパティへ 要素数 を代入すると、その要素数に配列が切り詰められます。つまり、`length` プロパティへ `0` を代入すると、インデックスが `0` 以降の要素がすべて削除されます。

## 空の配列を代入

さいごに、その配列の要素を削除するのではなく、新しい空の配列を変数へ代入する方法です。次のコードでは、`array` 変数に空の配列を代入することで、`array` は空の配列を参照させることができます。

```
let array = [1, 2, 3];
console.log(array.length); // => 3
// 新しい配列で変数を上書き
array = [];
console.log(array.length); // => 0
```

元々、`array` 変数が参照していた `[1, 2, 3]` はどこからも参照されなくなり、ガベージコレクションによりメモリから解放されます。

また、`var` で宣言していた変数を `const` にした場合は、再代入できないためこの手法は使うことができません。そのため、再代入をしたい場合は `let` または `var` で変数する必要があります。

```
const array = [1, 2, 3];
console.log(array.length); // => 3
// `const`で宣言された変数を再代入できない
array = []; // TypeError: invalid assignment to const `array` が発生
```

## 破壊的なメソッドと非破壊的なメソッド

これまで紹介してきた配列を変更するメソッドには、破壊的なメソッドと非破壊的メソッドがあります。この破壊的なメソッドと非破壊的メソッドの違いを知ることは、意図しない結果を避けるために重要です。破壊的なメソッドとは、配列オブジェクトそのものを変更し、変更した配列または変更箇所を返すメソッドです。非破壊的メソッドとは、配列オブジェクトのコピーを作成してから変更し、そのコピーの配列を返すメソッドです。

破壊的なメソッドの例として、配列に要素を追加する `Array#push` メソッドがあります。`push` メソッドは、`myArray` の配列そのものへ要素を追加しています。その結果 `myArray` の参照する配列が変更されるため破壊的なメソッドです。

```
const myArray = ["A", "B", "C"];
const result = myArray.push("D");
// `push`の返り値は配列ではなく、追加後の配列のlength
console.log(result); // => 4
// `myArray`が参照する配列そのものが変更されている
console.log(myArray); // => ["A", "B", "C", "D"]
```

非破壊的なメソッドの例として、配列に要素を結合する `Array#concat` メソッドがあります。`concat` メソッドは、`myArray` をコピーした配列に対して要素を結合しその配列を返します。その結果 `myArray` の参照する配列は変更されないため非破壊的なメソッドです。

```
const myArray = ["A", "B", "C"];
// `concat`の返り値は結合済みの新しい配列
const newArray = myArray.concat("D");
console.log(newArray); // => ["A", "B", "C", "D"]
// `myArray`は変更されていない
console.log(myArray); // => ["A", "B", "C"]
// `newArray`と`myArray`は異なる配列オブジェクト
console.log(myArray === newArray); // => false
```

JavaScriptにおいて破壊的なメソッドと非破壊的メソッドを名前から見分ける方法はありません。また、返り値が配列の破壊的なメソッドもあるため、返り値からも判別できません。たとえば、`Array#sort` メソッドは返り値がソート済みの配列ですが破壊的です。次に紹介するメソッドは破壊的なメソッドであり、その他のメソッドは非破壊的なメソッドです。

メソッド名	返り値
<code>Array.prototype.pop</code>	配列の末尾の値
<code>Array.prototype.push</code>	変更後の配列のlength
<code>Array.prototype.splice</code>	取り除かれた要素を含む配列
<code>Array.prototype.reverse</code>	反転した配列
<code>Array.prototype.shift</code>	配列の先頭の値
<code>Array.prototype.sort</code>	ソートした配列
<code>Array.prototype.unshift</code>	変更後の配列のlength
<code>Array.prototype.copyWithin</code>	変更後の配列
<code>Array.prototype.fill</code>	変更後の配列

破壊的メソッドは意図せぬ副作用を与えてしまうことがあるため、そのことを意識して利用する必要があります。たとえば、配列から特定のインデックスの要素を削除する `removeAtIndex` という関数を提供したいとします。

```
// `array`の`index`番目の要素を削除した配列を返す関数
function removeAtIndex(array, index) { /* 実装 */ }
```

次のように破壊的なメソッドである `Array#splice` メソッドで要素を削除すると、引数として受け取った配列にも影響を与えます。この場合 `removeAtIndex` 関数には副作用があるため、破壊的であることについてのコメントがあると親切です。

```
// `array`の`index`番目の要素を削除した配列を返す関数
// 引数の`array`は破壊的に変更される
function removeAtIndex(array, index) {
    array.splice(index, 1);
    return array;
}
const array = ["A", "B", "C"];
// `array`から1番目の要素を削除した配列を取得
const newArray = removeAtIndex(array, 1);
console.log(newArray); // => ["A", "C"]
// `array`自体にも影響を与える
console.log(array); // => ["A", "C"]
```

一方、非破壊的メソッドは配列のコピーを作成するため元々の配列に対して影響はありません。この`removeAtIndex`関数を非破壊的ものにするには、受け取った配列をコピーしてから変更を加える必要があります。

JavaScriptには`copy`メソッドはそのもの存在しませんが、配列をコピーする方法として`Array#slice`メソッドと`Array#concat`メソッドが利用されています。`slice`メソッドと`concat`メソッドは引数なしで呼び出すと、その配列のコピーを返します。

```
const myArray = ["A", "B", "C"];
// `slice`は`myArray`のコピーを返す - `myArray.concat()`でも同じ
const copiedArray = myArray.slice();
myArray.push("D");
console.log(myArray); // => ["A", "B", "C", "D"]
// `array`のコピーである`copiedArray`には影響がない
console.log(copiedArray); // => ["A", "B", "C"]
// コピーであるため参照は異なる
console.log(copiedArray === myArray); // => false
```

コピーした配列に変更を加えることで、`removeAtIndex`関数を非破壊的な関数として実装できます。非破壊的であれば引数の配列への副作用がないので、注意させるようなコメントは不要です。

```
// `array`の`index`番目の要素を削除した配列を返す関数
function removeAtIndex(array, index) {
    // コピーを作成してから変更する
    const copiedArray = array.slice();
    copiedArray.splice(index, 1);
    return copiedArray;
}
const array = ["A", "B", "C"];
// `array`から1番目の要素を削除した配列を取得
const newArray = removeAtIndex(array, 1);
console.log(newArray); // => ["A", "C"]
// 元の`array`には影響がない
console.log(array); // => ["A", "B", "C"]
```

このようにJavaScriptの配列には破壊的なメソッドと非破壊的メソッドが混在しています。そのため、統一的なインターフェースで扱えないのが現状です。このような背景もあるため、JavaScriptには配列を扱うためのさまざまライブラリが存在します。[immutable-array-prototype](#)や[Immutable.js](#)など非破壊的に配列を扱う目的のライブラリがあります。

## 配列を反復処理するメソッド

「[ループと反復処理](#)の章」において配列を反復処理する方法を一部解説していましたが、あらためて関連する`Array`メソッドを見ていきます。反復処理の中でもよく利用される`Array#forEach`、`Array#map`、`Array#filter`、`Array#reduce`について見ていきます。どのメソッドも共通して引数にコールバック関数を受け取るため高階関数と呼ばれます。

### Array#forEach

`Array#forEach`は配列の要素を先頭から順番にコールバック関数へ渡し、反復処理を行うメソッドです。

次のようにコールバック関数には`要素`、`インデックス`、`配列`が引数として渡され、配列要素の先頭から順番に反復処理します。

```
const array = [1, 2, 3];
array.forEach((currentValue, index, array) => {
    console.log(currentValue, index, array);
});
// コンソールの出力
// 1, 0, [1, 2, 3]
```

```
// 2, 1, [1, 2, 3]
// 3, 2, [1, 2, 3]
```

## Array#map

`Array#map` は配列の要素を順番にコールバック関数へ渡し、コールバック関数が返した値から新しい配列を返す非破壊的なメソッドです。配列の各要素を加工したい場合に利用します。

次のようにコールバック関数には `要素`, `インデックス`, `配列` が引数として渡され、配列要素の先頭から順番に反復処理します。`map` メソッドの返り値は、それぞれのコールバック関数が返した値を集めた新しい配列です。

```
const array = [1, 2, 3];
// 各要素に10を乗算した新しい配列を作成する
const newArray = array.map((currentValue, index, array) => {
  return currentValue * 10;
});
console.log(newArray); // => [10, 20, 30]
// 元の配列とは異なるインスタンス
console.log(array !== newArray); // => true
```

## Array#filter

`Array#filter` は配列の要素を順番にコールバック関数へ渡し、コールバック関数が `true` を返した要素だけを集めた新しい配列を返す非破壊的なメソッドです。配列から不要な要素を取り除いた配列を作成したい場合に利用します。

次のようにコールバック関数には `要素`, `インデックス`, `配列` が引数として渡され、配列要素の先頭から順番に反復処理します。`filter` メソッドの返り値は、コールバック関数が `true` を返した要素だけを集めた新しい配列です。

```
const array = [1, 2, 3];
// 奇数の値をもつ要素だけを集めた配列を返す
const newArray = array.filter((currentValue, index, array) => {
  return currentValue % 2 === 1;
});
console.log(newArray); // => [1, 3]
// 元の配列とは異なるインスタンス
console.log(array !== newArray); // => true
```

## Array#reduce

`Array#reduce` は累積値（アキュムレータ）と配列の要素を順番にコールバック関数へ渡し、1つの累積値を返します。配列から配列以外を含む任意の値を作成した場合に利用します。

ここまで紹介した反復処理のメソッドとはことなり、コールバック関数には `累積値`, `要素`, `インデックス`, `配列` を引数として渡します。`reduce` メソッドの第二引数には `accumulator` の初期値となる値を渡せます。

次のコードでは、`reduce` メソッドは配列の各要素を加算した1つの数値を返します。つまり配列から配列要素の合計値というNumber型の返しています。

```
const array = [1, 2, 3];
// すべての要素を加算した値を返す
// accumulatorの初期値は`0`
const totalValue = array.reduce((accumulator, currentValue, index, array) => {
  return accumulator + currentValue;
}, 0);
console.log(totalValue); // => 0 + 1 + 2 + 3
```

`Array#reduce` メソッドはやや複雑ですが、配列以外の値も返せるという特徴があります。

## [コラム] Array-likeオブジェクト

配列のように扱えるが配列ではないオブジェクトのことを、Array-likeオブジェクトと呼びます。Array-likeオブジェクトとは配列のようにインデックスにアクセスでき、配列のように`length`プロパティも持っています。しかし、配列のインスタンスではないため、`Array`メソッドは持っていないオブジェクトのことです。

機能	Array-likeオブジェクト	配列
インデックスアクセス ( <code>array[0]</code> )	できる	できる
長さ ( <code>array.length</code> )	持っている	持っている
<code>Array</code> メソッド ( <code>Array#forEach</code> など)	持っていない場合もある	持っている

Array-likeオブジェクトの例として`arguments`があります。`arguments`オブジェクトは、`function`で宣言した関数の中から参照できる変数です。`arguments`オブジェクトには関数の引数に渡された値が順番に格納されていて、配列のように引数へアクセスできます。

```
function myFunc() {
  console.log(arguments[0]); // => "a"
  console.log(arguments[1]); // => "b"
  console.log(arguments[2]); // => "c"
  // 配列ではないため、配列のメソッドは持っていない
  console.log(typeof arguments.forEach); // => "undefined"
}
myFunc("a", "b", "c");
```

Array-likeオブジェクトか配列なのかを判別するには`Array.isArray`メソッドを利用できます。`Array-like`オブジェクトは配列ではないので結果は常に`false`となります。

```
function myFunc() {
  console.log(Array.isArray([1, 2, 3])); // => true
  console.log(Array.isArray(arguments)); // => false
}
myFunc("a", "b", "c");
```

Array-likeオブジェクトは配列のようで配列ではないというもどかしさをもつオブジェクトです。`Array.from`メソッドを使うことでArray-likeをオブジェクト配列に変換して扱うことができます。一度配列に変換してしまえば`Array`メソッドも利用できます。

```
function myFunc() {
  // Array-likeオブジェクトを配列へ変換
  const argumentsArray = Array.from(arguments);
  console.log(Array.isArray(argumentsArray)); // => true
  // 配列のメソッドを利用できる
  argumentsArray.forEach(arg => {
    console.log(arg);
  });
}
myFunc("a", "b", "c");
```

## メソッドチェーンと高階関数

配列で頻出するパターンとしてメソッドチェーンがあります。メソッドチェーンとは名前のとおり、メソッドの呼び出しを行いその結果の値に対してさらにメソッドを呼び出すパターンのことを言います。

次のコードでは、`Array#concat` メソッドの返り値、つまり配列に対してさらに `concat` メソッドを呼び出すというメソッドチェーンが行われています。

```
const array = ["a"].concat("b").concat("c");
console.log(array); // => ["a", "b", "c"]
```

このコードの `concat` メソッドの呼び出しを分解してみると何がおこなわれているのか分かりやすいです。

`concat` メソッドの返り値は結合した新しい配列です。先ほどのメソッドチェーンでは、その新しい配列に対してさらに `concat` メソッドで値を結合しているということが分かります。

```
// メソッドチェーンを分解した例
// 一時的な`abArray`という変数が増えている
const abArray = ["a"].concat("b");
console.log(abArray); // => ["a", "b"]
const abcArray = abArray.concat("c");
console.log(abcArray); // => ["a", "b", "c"]
```

メソッドチェーンを利用することで処理の見た目を簡潔にできます。メソッドチェーンを利用した場合も最終的な処理結果は同じですが、途中の一時的な変数を省略できます。先ほどの例では `abArray` という一時的な変数をメソッドチェーンでは省略できています。

メソッドチェーンは配列に限ったものではありませんが、配列では頻出するパターンです。なぜなら、配列に含まれるデータを表示する際には、最終的に文字列や数値など別のデータへ加工することが殆どであるためです。配列には配列を返す高階関数が多く実装されているため、配列を柔軟に加工できます。

次のコードでは、ECMAScriptのバージョン名と発行年数が定義された `ecmaScriptVersions` という配列が定義されています。この配列から 2000 年以前に発行されたECMAScriptのバージョン名の一覧を取り出すことを考えてみます。目的の一覧を取り出すには「2000年以前のデータに絞り込む」と「データから `name` を取り出す」という2つの加工処理を組み合わせる必要があります。

この2つの加工処理は `Array#filter` メソッドと `Array#map` メソッドで実現できます。`filter` メソッドで配列から 2000 年以前というルールで絞り込み、`map` メソッドでそれぞれの要素から `name` プロパティを取り出せます。どちらのメソッドも配列を返すのでメソッドチェーンで処理を繋げることができます。

```
// ECMAScriptのバージョン名と発行年
const ECMA_scriptVersions = [
  { name: "ECMA_script 1", year: 1997 },
  { name: "ECMA_script 2", year: 1998 },
  { name: "ECMA_script 3", year: 1999 },
  { name: "ECMA_script 5", year: 2009 },
  { name: "ECMA_script 5.1", year: 2011 },
  { name: "ECMA_script 2015", year: 2015 },
  { name: "ECMA_script 2016", year: 2016 },
  { name: "ECMA_script 2017", year: 2017 },
];
// メソッドチェーンで必要な加工処理を並べている
const versionNames = ECMA_scriptVersions
  // 2000年以下のデータに絞り込み
  .filter(ECMA_script => ECMA_script.year <= 2000)
  // それぞれの要素から`name`プロパティを取り出す
  .map(ECMA_script => ECMA_script.name);
console.log(versionNames); // => ["ECMA_script 1", "ECMA_script 2", "ECMA_script 3"]
```

メソッドチェーンを使うことで複数の処理からなるものをひとつのまとった処理のように見せることができます。長過ぎるメソッドチェーンは長すぎる関数と同じように読みにくくなりますが、適度な単位のメソッドチェーンは処理をスッキリ見せるパターンとして利用されています。



# 文字列

この章ではJavaScriptにおける文字列について学んでいきます。文字列の表現やその背景にあるUnicodeを見ていき、文字列の操作方法について学びます。そして、文字列を編集して自由な文字列を作れるようになることがこの章の目的です。

## 文字列を作成する

文字列を作成するには文字列リテラルを利用します。文字列リテラルには3種類ありますが、まずは`"`（ダブルクオート）と`'`（シングルクオート）について見ていきます。（[データ型とリテラル](#)を参照）

`"`（ダブルクオート）と`'`（シングルクオート）に意味的な違いはありません。そのため、どちらを使うかは好みやプロジェクトごとのコーディング規約によって異なります。この書籍では、`"`（ダブルクオート）を主に文字列リテラルとして利用します。

```
const double = "文字列";
console.log(double); // => "文字列"
const single = '文字列';
console.log(single); // => '文字列'
// どちらも同じ文字列
console.log(double === single); // => true
```

文字列リテラルは同じ記号が対となるため、次のように文字列の中に同じ記号が出現した場合は、`\\"`のように`\`（バックスラッシュ）を使いエスケープする必要があります。

```
const string = "This book is \"js-primer\"";
console.log(string); // => 'This book is "js-primer"'
```

## 文字列を結合する

文字列を結合する簡単な方法は文字列結合演算子（`+`）を使う方法です。

```
const string = "a" + "b";
console.log(string); // => "ab"
```

変数と文字列を結合したい場合も文字列結合演算子で行うことができます。

```
const name = "JavaScript";
console.log("Hello " + name + "!"); // => "Hello JavaScript!"
```

特定の書式に値を埋め込みために文字列結合を行う場合には、テンプレートリテラルを使うとより宣言的に書くことができます。

テンプレートリテラルは```（バッククオート）で文字列を作成できる点は、`"`（ダブルクオート）や`'`（シングルクオート）と同じです。加えて、テンプレートリテラルは文字列中に変数を埋め込むことができます。

テンプレートリテラル中に `${変数名}` で書かれた変数は評価時に展開されます。つまり、先ほどの文字列結合は次のように書くことができます。

```
const name = "JavaScript";
console.log(`Hello ${name}!`); // => "Hello JavaScript!"
```

## 文字列とは

今まで何気なく「文字列」という言葉を利用していましたが、ここでいう文字列とはどのようなものでしょうか？

「文字列」とは「文字」が順番に並んでいるものです。これは、配列は配列の要素が順番に並んでいるという関係によく似ています。文字列においても、配列と同様にインデックスを指定することで、指定したインデックスにある文字へアクセスできます。

`文字列[インデックス]` という記述することでアクセスでき、インデックスの値は `0` 以上 `2^53 - 1` 未満の整数が指定できます。

```
const string = "文字列";
// 配列と同じようにインデックスでアクセスできる
console.log(string[1]); // => "字"
```

文字列と文字の関係を簡単に紹介しましたが、実際の文字列はもっと複雑です。私達は視覚的に文字を認識しますが、コンピュータでは文字の形ではなく、「ひらがなの『あ』という種類の文字」といった情報をやり取りします。また、視覚的に見えない制御文字や結合文字のように情報を組み合わせて扱うものも存在します。

制御文字など含めた文字は、JavaScriptエンジン上で一意なビット列に変換されて扱われます。文字をビット列へ変換することを符号化（エンコード）と呼びます。

文字とビット列の組み合わせを定義したものが文字コードであり、JavaScriptでは文字コードとしてUnicodeを採用しています。また、Unicodeには文字をエンコードする方式としてUTF-8、UTF-16、UTF-32などがありますが、JavaScriptではUTF-16を採用しています。

まとめるとJavaScriptはUnicodeのUTF-16を採用しており、文字列はUTF-16でエンコードされたデータとしてやり取りされます。

## Code Point

Unicodeでは、文字と1対1で対応するビット列を表のようなもので管理されています。たとえば、"A"という文字は表の56の位置にあるといった、文字とビット列の対応位置が決められています。この、対応表における位置のことを符号位置（Code Point）と呼びます。

ES2015で追加された `String#codePointAt` メソッドを使うことで、その文字のCode Pointを取得できます。

```
// 文字列"あ"の0番目のCode Pointを取得
console.log("あ".codePointAt(0)); // => 12354
```

逆に、`String.fromCodePoint` メソッドを使うことで、指定したCode Pointの文字を取得できます。

```
// 符号位置12354の文字を取得する
console.log(String.fromCodePoint(12354)); // => "あ"
```

また、文字列リテラル中にはUnicodeエスケープシーケンスで、直接Code Pointを書くこともできます。Code Pointは `\u{Code Pointの16進数の値}` で書くことができます。

```
// "あ"のCode Pointは12354
// 12354の16進数表現は3042
console.log("\u{3042}"); // => "あ"
```

Code Pointの16進数表現は次のようにして求めることができます。

```
// "あ"のCode Pointは12354
const codePointOfあ = "あ".codePointAt(0);
// 12354の16進数表現は"3042"
const hexOfCodePoint = codePointOfあ.toString(16);
// \はエスケープシーケンスであるため、\自体を表現するにはエスケープが必要
console.log("\u" + hexOfCodePoint + "}); // => "\u{3042}"
```

直接キーボードから入力が難しい特殊な文字や絵文字などは、Unicodeエスケープシーケンスを使うことでソースコード上に安全に書くことができます。

## Code Unit

符号単位（Code Unit）は、文字を構成する最小の単位ですが、解説をする前にまずUnicodeの歴史を振り返る必要があります。

Unicodeは元々16ビットつまり最大65,536文字で世界中の文字が取まるという前提で、文字とCode Pointの組み合わせを定義していました。しかし、今もなお増えている文字が65,536文字で取まるわけもなく、1文字（1つのCode Point）が16ビット1つで表現できるという前提は崩れてしまいました。そこで、16ビットを2つ並べることによって扱える文字数を増やすエンコード方式がUnicodeに追加されました。このエンコード方式がUTF-16です。

JavaScriptの仕様であるECMAScriptもUTF-16を採用しているため、この16ビットを2つ並べることで1文字（1つのCode Point）を表現できます。JavaScriptにおいては、この16ビット1つのことを符号単位（Code Unit）と呼び、文字列における最小の単位です。そのため、すべての文字列はCode Unitが並んでいるものとして扱われます。

16ビット1つで表現できる文字の場合は、Code PointとCode Unitが同じ値を示します。

Code UnitもCode Pointと同じく、Unicodeエスケープシーケンスとして文字列リテラルに書くことができます。Code Pointとよく似ていますが、\u4桁の16進数と文字列リテラル中に書くことができます。

```
// Code Unit
"\u3042"; // => "あ"
// Code Point
"\u{3042}"; // => "あ"
```

1つの16ビットで表現できない文字をUTF-16では2つの16ビットで表現します。この表現方法をサロゲートペアと呼び、Code Unit2つで1つのCode Pointを表現します。

UTF-16では次の範囲をサロゲートペアの領域としています。

- \uD800 ~ \uDBFF : 上位サロゲートの範囲
- \uDC00 ~ \uDEFF : 下位サロゲートの範囲

これは、文字列中に上位サロゲートであるCode Unitが登場したとき、次のインデックスにある下位サロゲートであるCode Unitを組み合わせて1文字（厳密にはCode Point）とするということです。

具体的にサロゲートペアの文字である「鰐（ほつけ）」は次の2つのCode Unitで表現できます。

```
// 上位サロゲート + 下位サロゲートの組み合わせ
"\uD867\uDE3D"; // => "鰐"
// Code Pointでの表現
"\u{29e3d}"; // => "鰐"
```

このようにCode Unitは歴史的な経緯もあり、1つまたは2つのCode Unitで1つのCode Pointを表現します。JavaScriptでは基本的にStringメソッドは、文字列をCode Unitが並んでいるものとして扱います。

```
"文字列";
// == 内部的にはCode Unitが並んでいるもの
"\u6587\u5b57\u5217"; // => "文字列"
```

```
// インデックスでのアクセスもCode Unitごと
"𩿱"[0]; // => "\uD867"
```

次の3つは例外として、文字列をCode Pointが並んでいるように扱います。

- Iterator (`for...of` や `Array.from` など)
- メソッドに `CodePoint` という名前を含むもの
- `u` (Unicode) フラグが有効化されている正規表現

## 文字列とCode UnitとCode Point

ここまでを踏まえて、JavaScriptにおける文字列とは何かということをまとめると次のように言い表せます。

- 「文字列」は「Code Unit」が順番に並んだもの
- 「文字列」は「Code Point」ごとに扱う方法が別途用意されている

このように文字列を扱うStringメソッドにおいては、各Code Unitごとに処理されている前提が、直感的ではない結果を発生させることができます。

この章では、どのような場面でこの仕組みを意識するのかを考えつつStringメソッドについて見てていきます。 Unicodeについて詳しくは[プログラマのための文字コード技術入門](#)などを参照してください。

## 文字列の分解と結合

文字列を配列へ分解するには `String#split` メソッドを利用できます。一方、配列の要素を結合し文字列にするには `Array#join` メソッドが利用できます。

この2つはよく組み合わせて利用されるため、あわせてみていきます。

`String#split` メソッドは、第一引数に指定した区切り文字で文字列を分解した配列を返します。次のコードでは、文字列を`・`で区切った配列を作成しています。

```
const strings = "赤・青・緑".split("・");
console.log(strings); // => ["赤", "青", "緑"]
```

分解してできた文字列の配列を結合して文字列を作る際に、`Array#join` メソッドがよく利用されます。

`Array#join` メソッドの第一引数には区切り文字を指定し、その区切り文字で結合した文字列を返します。

この2つを合わせれば、区切り文字を`・`から`、`へ変換する処理を次のように書くことができます。

```
const string = "赤・青・緑".split("・").join("、");
console.log(string); // => "赤、青、緑"
```

`String#split` メソッドの第一引数には正規表現を指定することもできます。これを利用すると、次のように文字列をスペースで区切るような処理が簡単に書くことができます。

```
// 文字列を1つ以上のスペースを区切りにして分解する
const strings = "a b c d".split(/\s+/);
console.log(strings); // => ["a", "b", "c", "d"]
```

## String#split と空文字

`String#split` メソッドでは、空文字（`""`）を区切り文字として指定し、文字列を文字の配列にする方法として紹介されることがあります。

```
// 空文字("")で文字列を分解する
const characters = "文字列".split("");
console.log(characters); // => ["文", "字", "列"]
```

しかし、この空文字での区切り方には問題があります。

一部を除いた文字列操作は、基本的に文字列をCode Unitが並んでいるものとして扱います。つまり、`split` メソッドも各Code Unitごとに文字列を分解しています。

次のコードを見ると、`string.split("")` は各文字ごとで分解するのではなく、各Code Unitごとに分解していることが分かります。

```
// "鰐"はサロゲートペアであるため2つのCode Unit (\uD867\uDE3D) からなる
// サロゲートペアを含む文字列を各Code Unitに分解
const codeUnitElements = "鰐のひらき".split("");
// サロゲートペアを各CodeUnitに分解したため、文字化けしている
console.log(codeUnitElements); // ["𠮷", "の", "ひ", "ら", "き"]
```

サロゲートペアを含んだ文字列をそれぞれのCode Pointへ分解するには、`Iterator`を利用するが簡単です。文字列は`Iterator`を実装している`Iterable`という特性をもち、また文字列の`Iterator`はそれぞれのCode Pointごとに列挙します。

そのため、`Iterable`を扱える`Array.from` メソッドや`... (spread構文)` を利用することで、文字列をそれぞれのCode Pointごとに分解できます。

```
const string = "鰐のひらき";
// Array.fromメソッドで文字列を分解
console.log(Array.from(string)); // => ["鰐", "の", "ひ", "ら", "き"]
// ... (spread構文) で文字列を展開しものを配列にする
console.log([...string]); // => ["𠮷", "の", "ひ", "ら", "き"]
// for...ofもIteratorを列挙するため、Code Pointごとで列挙できる
for (const codePoint of string) {
  console.log(codePoint);
}
```

絵文字などサロゲートペアを含む文字列をそれぞれのCode Unitで分解すると、加工して結合すると化けてしまうなどの問題が発生しやすいです。`Iterator`を利用すればサロゲートペアもそれぞれのCode Pointで扱うことができます。

しかし、JavaScriptにおいて、見た目どおりの文字ごとに処理を行う標準的な方法は用意されていません。結合文字などを考慮した文字について、詳しくは[JavaScript has a Unicode problem · Mathias Bynens](#)を参照してください。

## 文字列の長さ

`string.length` プロパティは文字列の要素数を返します。文字列のそれぞれの要素はCode Unitであるため、`length` プロパティはCode Unitの数を返します。つまり、サロゲートペアを含む文字列は視覚的な長さとは異なる値になります。

```
console.log("文字列".length); // => 3
// 評価結果の文字列の要素数 (Code Unit数) であるため1つ
console.log("\u{3042}".length); // => 1
// サロゲートペアを含むためCode Unitは6つ
console.log("𠮷のひらき".length); // => 6;
```

これは、`文字列.split("").length` と同じ結果です。

## Code Pointの数

一般に言われる文字列の長さとは視覚的な文字の数を表すことが多いため、`String#length` だけでは文字列の長さを得ることが難しいです。

たとえば、絵文字はUnicodeとして定められており、これらはサロゲートペアとして表現されています。そのため、絵文字などサロゲートペアを含む文字列が日常的に使われるようになった今では、Code Unitの数を文字列の長さとしたときに直感と反する場合が増えてきています。

たとえば、Twitterにおける140文字の文字数にはCode Pointの数をベースにしています。（[twitter-text](#)というライブラリとして公開されています。）`Array.from` メソッドを利用すれば、文字列におけるCode Pointの数は次のようにして取得できます。

```
// Code Pointごとの配列にする
// Array.fromメソッドはIteratorを配列にする
const codePoints = Array.from("𩫔のひらぎ");
console.log(codePoints.length); // => 5
```

Code Pointの数を数えた場合も、結合文字など視覚的に見えないものを1つと数えてしまします。そのため、文字として数えたくないものは正規表現で取り除く必要があるなど、視覚的な文字列の長さを数えるにはさらなる工夫が必要になります。

ECMAScriptが参照するUnicodeの仕様も更新されて続けています。そのため、文字列の長さを正確に測るにはある程度の妥協が必要になります。

## 文字列の比較

文字列の比較には`==`（厳密比較演算子）を利用します。

```
console.log("文字列" === "文字列"); // => true
// 文字列の評価結果を比較するため、評価結果の文字列はどちらも同じ
console.log("𩫔のひらぎ" === "\u{29e3d}のひらぎ"); // => true
// 一致しなければfalseとなる
console.log("JS" === "ES"); // => false
```

次の条件を満たしていれば同じ文字列となります。

- 文字列の要素であるCode Unitが同じ順番で並んでいるか
- 文字列の長さ（`length`）は同じか

また、`==`などの比較演算子だけではなく、`>`、`<`、`>=`、`<=`など大小の関係演算子で文字列同士を比較することもできます。

これらの関係演算子も、文字列の要素であるCode Unitの数値を先頭から順番に比較します。しかし、これらの関係演算子は暗黙的な型変換を行うため事前に文字列同士であるかのチェックが必要です。

文字列からCode Unitの数値を取得するには`String#charCodeAt` メソッドを利用します。

```
// "A"と"B"のCode Unitは65と66
console.log("A".charCodeAt(0)); // => 65
console.log("B".charCodeAt(0)); // => 66
// "A" (65) は"B" (66) よりCode Unitの値が小さい
console.log("A" > "B"); // => false
// 先頭から順番に比較し C > D が falseであるため
console.log("ABC" > "ABD"); // => false
```

このように、JavaScriptの文字列比較はCode Unitがベースとなります。

この「先頭から順番に比較する」という仕様は、数字の文字列を数値順にソートしたい時などに問題が起きます。次のように、`["10", "2", "1"]` という数字の配列を `Array#sort` メソッドで昇順ソートした場合、直感的には `["1", "2", "10"]` となることを期待します。しかし、実際の結果は `["1", "10", "2"]` となります。

```
const numberStrings = ["10", "2", "1"];
// Array#sortは、デフォルトでは配列の要素を`<`で比較する
// 文字列同士を`<`で比較し、Code Unitの値で昇順にした配列を返している
console.log(numberStrings.sort()); // => ["1", "10", "2"]
```

なぜなら、`"10" < "2"` を比較した場合に、先頭の文字（Code Unit）から順番に比較されるためです。まず `"10"[0] < "2"[0]` が比較され `true` となり、`"10"` が `"2"` より小さいと判定されます。

```
console.log("10" < "2"); // => true
// 数値同士なら10の方が大きい
console.log(10 < 2); // => false
```

数字の比較だけではなく、文字列の比較は地域や言語によって「自然な結果」が異なります。たとえば、`a`（アクセント付きa）と`z`は、アメリカ英語のアルファベットでは`a`の方が後ろの位置にありますが、スウェーデンアルファベットでは`a`の方が前になります。

このように文字列の並び順をひとつとっても、地域や言語によって異なるため、ローカライズする必要があります。JavaScriptでは、ECMAScriptの関連仕様として国際化API（ECMAScript Internationalization API）があります。  
(Internationalizationは長いためしばしばi18nと省略されることがあります)

この書籍では詳しく紹介しませんが、国際化APIは `Intl` オブジェクトにあり、言語に依存した整形や比較などが利用できます。`Intl` オブジェクトはECMAScriptの仕様ではなく、[ECMA-402](#)と呼ばれる「ECMAScriptに関連する仕様」という位置づけになっています。そのため、すべての実行環境で実装されているわけではありません。

ブラウザにおけるサポート状況については[Can I use...](#)で見ることができます。

先ほどの数字のソートについては、国際化APIのひとつである `Intl.Collator` コンストラクタを利用してことで地域化できます。`Intl.Collator` はさまざまなオプションを持ちますが、`numeric` オプションを `true` にすることで数字を数値として比較できます。

```
// numericをtrueとすると数字が数値として比較されるようになる
const collator = new Intl.Collator("ja", { numeric: true });
// collator.compareはsortに渡せる関数となっている
const sortedValues = ["1", "10", "2"].sort(collator.compare);
console.log(sortedValues); // => ["1", "2", "10"]
```

文字列の比較においては、単純な比較であれば、`==`（厳密比較演算子）や`>`（大なり演算子）を利用します。その地域や言語においてのより自然な形を求める場合は、ローカライズするために国際化APIなどを利用できます。

## 部分文字列の取得

文字列からその一部を取り出したい場合には、`String#slice` メソッドや`String#substring` メソッドが利用できます。

`slice` メソッドについては、すでに配列で学んでいますが、基本的な動作は文字列でも同様です。まずは `slice` メソッドについて見てきます。

`String#slice` メソッドは、第一引数に開始位置、第二引数に終了位置を指定しその範囲を取り出し新しい文字列を返します。第二引数は省略でき、省略した場合は文字列の末尾が終了位置となります。

位置にマイナスの値を指定した場合は文字列の末尾から数えた位置となります。また、第一引数の位置が第二引数の位置より大きい場合、常に空の文字列を返します。

```
const string = "ABCDE";
console.log(string.slice(1)); // => "BCDE"
console.log(string.slice(1, 5)); // => "BCDE"
// マイナスを指定すると後ろからの位置となる
console.log(string.slice(-1)); // => "E"
// 位置:1から4の範囲を取り出す
console.log(string.slice(1, 4)); // => "BCD"
// 第一引数 > 第二引数の場合、常に空文字を返す
console.log(string.slice(4, 1)); // => ""
```

`String#substring` メソッドは、`slice` メソッドと同じく第一引数に開始位置、第二引数に終了位置を指定しその範囲を取り出し新しい文字列を返します。第二引数を省略した場合の挙動も同様で、省略した場合は文字列の末尾が終了位置となります。

`slice` メソッドとは異なる点として、位置にマイナスの値を指定した場合は常に `0` として扱われます。また、第一引数の位置が第二引数の位置より大きい場合、第一引数と第二引数を入れ替わるという予想しにくい挙動となります。

```
const string = "ABCDE";
console.log(string.substring(1)); // => "BCDE"
console.log(string.substring(1, 5)); // => "BCDE"
// マイナスを指定すると0として扱われる
console.log(string.substring(-1)); // => "ABCDE"
// 位置:1から4の範囲を取り出す
console.log(string.substring(1, 4)); // => "BCD"
// 第一引数 > 第二引数の場合、引数を入れ替わる
// string.substring(1, 4)と同じ結果になる
console.log(string.substring(4, 1)); // => "BCD"
```

このように、マイナスの位置や引数が交換される挙動は分かりやすいものとはいえません。そのため、`slice` メソッドと `substring` メソッドに指定する引数は、どちらとも同じ結果となる範囲に限定した方が直感的な挙動となります。つまり、位置は0以上の値でかつ、第二引数を指定する場合は `第一引数の位置 < 第二引数の位置` にするということです。

`slice` メソッドと `substring` メソッドの引数に直接 `1` や `4` といった位置を指定することは少ないです。次のように、`String#indexof` メソッドなど位置を取得するものと組み合わせて使うことが多いでしょう。

```
const url = "https://example.com?param=1";
const indexOfQuery = url.indexOf("?");
const queryString = url.slice(indexOfQuery);
console.log(queryString); // => "?param=1"
```

また、配列とは異なりプリミティブ型の値である文字列は、`slice` メソッドと `substring` メソッド共に非破壊的です。機能的な違いが殆どないため、どちらを利用するかは好みの問題となるでしょう。

## 文字列の検索

文字列の検索方法として、大きく分けて文字列による検索と正規表現による検索があります。

### 文字列による検索

文字列による検索は「文字列」から「部分文字列」を検索できます。`String` メソッドには検索したい状況に応じたものが用意されています。

#### インデックスの取得

検索した結果「部分文字列」の開始インデックスを取得する `String#indexOf` メソッドと `String#lastIndexOf` メソッドがあります。これは、配列の `Array#indexOf` メソッドと同じで、厳密等価演算子（`==`）で一致する文字列のインデックスを取得します。一致する文字列がない場合は `-1` を返します。

- 文字列`.indexOf("部分文字列")`：先頭からの検索し、インデックスを返す
- 文字列`.lastIndexOf("部分文字列")`：末尾から検索し、インデックスを返す

どちらのメソッドも一致する文字列が複数個ある場合でも、指定した部分文字列を一度見つけた時点で検索は終了します。

```
// 検索対象となる文字列
const string = "にわにはにわにわとりがいる";
// indexOfは先頭から検索しインデックスを返す - "***にわ**にはにわにわとりがいる"
// "にわ"の先頭のインデックスを返すため 0 となる
console.log(string.indexOf("にわ")); // => 0
// lastIndexOfは末尾から検索しインデックスを返す - "にわにはにわ**にわ**とりがいる"
console.log(string.lastIndexOf("にわ")); // => 6
// 該当する部分文字列が見つからない場合は -1 を返す
console.log(string.indexOf("キーワード")); // => -1
```

検索している部分文字列の長さは固定であるため、一致した文字列は自明ですが、`String#slice` と取得したインデックスを組み合わせることで検索結果を取得できます。

```
const string = "JavaScript";
const searchWord = "Script";
const index = string.indexOf("Script");
if (index !== -1) {
  console.log(string.slice(index, index + searchWord.length)); // => searchword
} else {
  console.log(`${searchWord}は見つかりませんでした`);
}
```

ES2015より前では `String#indexOf` メソッドしか固定文字列の検索できませんでした。そのため、`string.indexOf("検索文字列") !== -1` で "検索文字列" が含まれているかを表現するイディオムがありました。しかし、ES2015以降は `String#include` メソッドなど、より適切な真偽値を取得するメソッドが追加されています。

```
// indexOfで含まれているかを判定する表現するイディオム
console.log("にわにはにわとりがいる".indexOf("にわ") !== -1); // => true
// String#includeによる同等の表現
console.log("にわにはにわとりがいる".includes("にわ")); // => true
```

## 真偽値の取得

「文字列」に「部分文字列」が含まれているかを検索する方法がいくつか用意されています。

- 文字列`.startsWith("部分文字列")`：先頭にあるかの真偽値を返す[ES2015]
- 文字列`.endsWith("部分文字列")`：終端にあるかの真偽値を返す[ES2015]
- 文字列`.includes("部分文字列")`：含むかの真偽値を返す[ES2015]

具体的な例をいくつか見てみましょう。

```
// 検索対象となる文字列
const string = "にわにはにわにわとりがいる";
// startsWith - 部分文字列が先頭ならtrue
console.log(string.startsWith("にわ")); // => true
console.log(string.startsWith("いる")); // => false
// endsWith - 部分文字列が末尾ならtrue
console.log(string.endsWith("にわ")); // => false
console.log(string.endsWith("いる")); // => true
```

```
// includes - 部分文字列が含まれるならtrue
console.log(string.includes("にわ")); // => true
console.log(string.includes("いる")); // => true
```

## 正規表現による検索

正規表現による検索は、正規表現オブジェクトを利用します。

正規表現オブジェクトは正規表現リテラルや `RegExp` コンストラクタを用いて生成できます。

正規表現リテラルは、`/` と `/` のリテラル内に正規表現のパターンを書くことで、正規表現オブジェクトを静的に生成します。正規表現のパターン内では、`+` や `\` (バックスラッシュ) から始まる特殊文字が特別な意味を持ちます。

次のコードでは、スペースやタブにマッチする特殊文字である `\s` を用い、3つ連続するホワイトスペースにマッチする正規表現を生成しています。

```
// 3つの連続するスペースにマッチする正規表現
const pattern = /\s{3}/;
```

一方、`RegExp` コンストラクタは、文字列から正規表現オブジェクトを動的に生成できます。正規表現リテラルでは、動的に正規表現オブジェクトを生成することはできません。そのため、`RegExp` コンストラクタは、変数をパターンに埋め込んだ正規表現を生成する際に利用されます。

注意点として、`\` (バックスラッシュ) 自体が、文字列中ではエスケープ文字であることに注意してください。そのため、`RegExp` コンストラクタの引数のパターン文字列において、バックスラッシュから始まる特殊文字はバックスラッシュを2つにする必要があります。

```
const spaceCount = 3;
// `/\\s{3}/` の正規表現を動的に生成する
// "\"がエスケープ文字であるため、\"自身を文字列として書くには、\"\\\"のように2つ書く
const pattern = new RegExp(`\\s{${spaceCount}}`);
```

`RegExp` コンストラクタは動的に正規表現オブジェクトを生成できますが、正規表現の特殊文字のエスケープが必要になります。そのため、正規表現リテラルで表現できる場合は、リテラルを利用したほうが簡潔です。パターンに変数を利用する場合など、動的でないと表現できないものは `RegExp` コンストラクタを利用します。

## マッチした文字列を取得

`String#indexOf` メソッドの正規表現版ともいえる `String#search` メソッドがあります。

- `String#indexOf(部分文字列)` : 部分文字列にマッチした文字列のインデックスを返す
- `String#search(/パターン/)` : 正規表現のパターンにマッチした文字列のインデックスを返す

文字列による検索は、検索しマッチした文字列の長さが決まっているため、`indexOf` メソッドでインデックスを取得することに意味がありました。しかし、正規表現による検索は、パターンによる検索であるため、検索しマッチした文字列の長さは固定ではありません。つまり、次のように `String#search` メソッドでインデックスのみを取得しても、実際にマッチした文字列が分かりません。

```
const string = "abc123def";
const searchPattern = /\d+/";
const index = string.search(searchPattern); // => 3
// `index` だけではマッチした文字列が分からない
// そのため`マッチした文字列の長さ`が`String#search`では分からない
string.slice(index, index + マッチした文字列の長さ); // マッチした文字列は取得できない
```

そのため、マッチした文字列そのものを取得するには `RegExp#exec` メソッドか `String#match` メソッドを利用します。これらのメソッドは、正規表現の繰り返す `g` フラグ (globalの略称) と組み合わせてよく利用されます。

- `String#match(正規表現)` : 文字列中でマッチするものを検索する
  - マッチした文字列の配列を返す
  - マッチしない場合は `null` を返す。
  - `g` フラグが有効化されている時は、マッチしたすべての結果を配列で返す
- `RegExp#exec(文字列)` : 文字列中でマッチするものを検索する
  - マッチした文字列の配列を返す
  - マッチしない場合は `null` を返す
  - `g` フラグが有効化されている時は、正規表現オブジェクト自身が最後にマッチしたインデックスを記憶する

通常の検索では、検索結果が見つかった時点で検索が終了します。しかし、正規表現の `g` フラグを有効化することで、検索結果を見つけた場合も検索を続けることができます。

たとえば、`/[a-zA-Z]+/` という正規表現は `a` から `z` のどれかの文字が1つ以上連続しているものにマッチします。`String#match` メソッドは、`g` フラグのなしではマッチする最初の結果のみを返しますが、`g` フラグありではすべての結果を返します。

```
const string = "ABC あいう DE えお";
// gフラグなしでは、最初の結果のみを持つ配列を返す
const results = string.match(/[a-zA-Z]+/);
console.log(results); // => ["ABC"]
// aからZのどれかの文字が1つ以上連続するパターンにマッチするものを繰り返した(gフラグ)結果を返す
const resultsWithG = string.match(/[a-zA-Z]+/g);
console.log(resultsWithG[0]); // => "ABC"
console.log(resultsWithG[1]); // => "DE"
```

`RegExp#exec` メソッドも、`g` フラグの有無によって挙動が変化します。`g` フラグなしではマッチした最初の結果のみを取得します。しかし、`g` フラグありでは最後にマッチした末尾のインデックスを正規表現オブジェクトの `lastIndex` プロパティに記憶します。次に、`exec` メソッドを呼び出すと最後にマッチした末尾のインデックスから検索が開始されます。

```
const string = "ABC あいう DE えお";
// gフラグなしでは、最初の結果のみを持つ配列を返す
const results = /[a-zA-Z]+/.exec(string);
console.log(results); // => ["ABC"]
// gフラグが有効化されているパターン
const alphabetsPattern = /[a-zA-Z]+/g;
// まだ一度も検索していないので、lastIndexは0となり先頭から検索開始される
console.log(alphabetsPattern.lastIndex); // => 0
// gフラグありでも、一回目の結果は同じだが、`lastIndex`プロパティが更新される
console.log(alphabetsPattern.exec(string)); // => ["ABC"]
console.log(alphabetsPattern.lastIndex); // => 3
// 2回目の検索が、`lastIndex`の値のインデックスから開始される
console.log(alphabetsPattern.exec(string)); // => ["DE"]
```

どちらのメソッドも `g` フラグによって挙動が変わり、`RegExp#exec` メソッドに `lastIndex` プロパティを変更するという副作用を持ちます。

## マッチした一部の文字列を取得

どちらのメソッドも正規表現のパターン内にかかれた ( ) で囲んだ部分を取得できます。この `\パターン(パターン)` のように括弧で囲んだ部分を取り出せるようにすることをキャプチャリングと呼びます。

正規表現のパターン全体に一致する文字列は必要ないですが、`()`で囲んだ部分（キャプチャした部分）の文字列だけが欲しいという場合に利用できます。`String#match` メソッド、`RegExp#exec` メソッドどちらもマッチした結果として配列を返します。

そのマッチしてるパターンにキャプチャが含まれている場合は、次のように返り値の配列へキャプチャした部分が追加されていきます。

```
const [マッチした文字列, ...キャプチャされた文字列] = 文字列.match(/パターン(キャプチャ)/);
```

具体的な例を見てみましょう。

```
// "ECMAScript (数字+)"にマッチするが、欲しい文字列は数字の部分のみ
const pattern = /ECMAScript (\d+)/i;
// 戻り値は0番目がマッチした全体、1番目がキャプチャの1番目というように対応している
// [マッチした全部の文字列, キャプチャの1番目, キャプチャの2番目 ...]
// `pattern.exec("ECMAScript 6")`も戻り値は同じ
const [all, capture1] = "ECMAScript 6".match(pattern);
console.log(all); // => "ECMAScript 6"
console.log(capture1); // => "6"
```

## 真偽値を取得

正規表現オブジェクトを使い、そのパターンにマッチするかをテストするには、`RegExp#test` メソッドを利用できます。

正規表現のパターンには、位置を指定する特殊文字があります。そのため、「文字列による検索」で登場したメソッドは、すべての特殊文字と`RegExp#test` メソッドで表現できます。

- `String#startsWith` : `/^パターン/.test(文字列)`
- `String#endsWith` : `/パターン$/.test(文字列)`
- `String#includes` : `/パターン/.test(文字列)`

具体的な例を見てみましょう。

```
// 検索対象となる文字列
const string = "にわにはにわにねとりがいる";
// ^ - 部分文字列が先頭ならtrue
console.log(/^にわ/.test(string)); // => true
console.log(/^いる/.test(string)); // => false
// $ - 部分文字列が末尾ならtrue
console.log(/にわ$/_.test(string)); // => false
console.log(/いる$/_.test(string)); // => true
// 部分文字列が含まれるならtrue
console.log(/にわ/.test(string)); // => true
console.log(/いる/.test(string)); // => true
```

その他にも、正規表現では繰り返しや文字の集合などを特殊文字で表現できるため、`String` メソッドによる検索より曖昧検索が簡単に書くことができます。

## 文字列と正規表現どちらを使うべきか

`String` メソッドでの検索と同等のことは、正規表現でもできることがわかりました。`String` メソッドと正規表現で同じ結果が得られる場合はどちらを利用するのがよいでしょうか？

正規表現は曖昧な検索に強く、特殊文字を使うことで柔軟な検索結果を得ることができます。一方、曖昧であるため、コードを見ても何を検索しているかが正規表現のパターン自体から分からぬことがあります。

次の例は、`/`から始まり`/`で終わる文字列かを判定する正規表現とStringメソッドを使った方法を比べたものです。  
(これは意図的に正規表現に不利な例となっています)

正規表現の場合、`/^\.*\/$/`のようにパターンそのものを見ても何をしたいのかはひと目では分かりにくいです。  
Stringメソッドの場合は、`/`から始まり`/`で終わるかを判定することがそのままコードにあらわれています。

```
const string = "/正規表現のような文字列/";
// 正規表現で`/`から始まり`/`で終わる文字列のパターン
const regExpLikePattern = /\^/.*/\$/;
// RegExp#testメソッドでパターンにマッチするかを判定
console.log(regExpLikePattern.test(string)); // => true
// Stringメソッドで同等の判定をする関数
const isRegExpLikeString = (string) => {
  return string.startsWith("/") && string.endsWith("/");
};
console.log(isRegExpLikeString(string)); // => true
```

このように、正規表現は柔軟で便利ですが、コード上から意図が消えてしまいやすいです。そのため、正規表現を扱う際にはコメントや変数名で具体的な意図を補足する必要があります。

「Stringメソッドと正規表現で同じ結果が得られる場合はどちらを利用するのがよいでしょうか？」という疑問に戻ります。Stringメソッドで表現できることはStringメソッドで表現し、柔軟性や曖昧な検索が必要な場合はコメントとともに正規表現を利用するという方針を推奨します。

正規表現についてより詳しくは[正規表現 - JavaScript | MDN](#)や、コンソールで実行しながら試せる[regex101](#)のようなサイトを参照してください。

## 文字列の置換/削除

文字列を一部を置換や削除するには`String#replace`メソッドを利用します。「データ型とリテラル」で説明したようにプリミティブ型である文字列は不变な特性をもちます。そのため、文字列から一部の文字を削除するような操作はできません。

つまり、`delete`演算子は文字列に対して利用できません。`strict mode`では削除出来ないプロパティを削除しようとするエラーが発生します。（`strict mode`でない場合はエラーも発生せず単に無視されます。）

```
"use strict";
const string = "文字列";
// 文字列の0番目を削除を試みるがStrict modeは例外が発生
delete string[0]; // => Error
```

代わりに、`String#replace`メソッドなどで削除したい文字を取り除いた新しい文字列を返すことで削除を表現します。`replace`メソッドは、文字列から第一引数の`検索文字列`または正規表現にマッチする部分を、第二引数の`置換文字列`へ置換します。第一引数には、文字列または正規表現を指定できます。

```
文字列.replace("検索文字列", "置換文字列");
文字列.replace(/パターン/, "置換文字列");
```

次のように、`replace`メソッドで、削除したい部分を空文字へ置換することで、文字列を削除できます。

```
const string = "文字列";
// "文字"を""(空文字)へ置換することで"削除"を表現
const newString = string.replace("文字", "");
console.log(newString); // => "列"
```

`replace` メソッドには正規表現も指定できます。`g` フラグを有効化した正規表現を渡すことで、文字列からパターンにマッチするものをすべて削除できます。

```
// 検索対象となる文字列
const string = "にわにはにわにわとりがいる";
// 文字列を指定した場合は、最初に一致したものだけが置換される
console.log(string.replace("にわ", "niwa")); // => "niwaにはにわにわとりがいる"
// `g` フラグなしの場合は、最初に一致したものだけが置換される
console.log(string.replace(/にわ/, "niwa")); // => "niwaにはにわにわとりがいる"
// `g` フラグで繰り返し置換を行う
console.log(string.replace(/にわ/g, "niwa")); // => "niwaにはniwaniwaとりがいる"
```

`replace` メソッドでは、キャプチャした文字列を利用しさらに複雑な置換処理をおこなうこともできます。

`replace` メソッドの第二引数にはコールバック関数を渡すことができます。第一引数の `パターン` にマッチした部分がコールバック関数の返り値で置換されます。コールバック関数の第一引数には `パターン` に一致した文字列全体、第二引数以降へキャプチャした文字列が順番に入ります。

```
const 置換した結果の文字列 = 文字列.replace(/(パターン)/, (all, ...captures) => {
    return 置換したい文字列;
});
```

例として、2017-03-01 を 2017年03月01日 に置換する処理を書いてみましょう。

`/(\d{4})-(\d{2})-(\d{2})/` という正規表現が "2017-03-01" という文字列へマッチします。コールバック関数の `year`、`month`、`day` にはそれぞれキャプチャした文字列が入り、マッチした文字列全体がコールバック関数の返り値に置換されます。

```
function toDateJa(dateString) {
    // パターンにマッチしたときのみ、コールバック関数で置換処理が行われる
    return dateString.replace(/(\d{4})-(\d{2})-(\d{2})/, (all, year, month, day) => {
        // `all`には、マッチした文字列全体が入っているが今回は利用しない
        // `all`が次の返す値で置換されるイメージ
        return `${year}年${month}月${day}日`;
    });
}
// マッチしない文字列の場合は、そのままの文字列が返る
console.log(toDateJa("本日八晴天ナリ")); // => "本日八晴天ナリ"
// マッチした場合は置換した結果を返す
console.log(toDateJa("今日は2017-03-01です")); // => "今日は2017年03月01日です"
```

## 文字列の組み立て

最後に文字列の組み立てについて見ていきましょう。最初に述べたようにこの章の目的は、「自由な文字列を作れるようになること」です。

文字列を単純に結合したり置換することで新しい文字列を作れることがわかりました。一方、構造的な文字列の場合は単純に結合するだけでは意味が壊なってしまうことがあります。

ここで構造的な文字列とは、URL文字列やファイルパス文字列といった構造をもつ文字列です。たとえば、URL文字列は次のような構造を持っており、それぞれの要素に入る文字列の種類などが制限されています。(「[URL Standard](#)」を参照)

```
"http://example.com/index.html"  
  ^^^^  ^^^^^^^^^^  
|          |          ^^^^^^  
scheme    host    pathname
```

これらの文字列を作成する場合は、文字列結合演算子（+）で単純に結合するよりも専用の関数を用意する方が安全です。

たとえば、次のように `baseURL` と `pathname` を渡し、それらを結合したURLにあるリソースを取得する `getResource` 関数があるとします。この `getResource` 関数には、ベースURLとベースURLからのパスを引数にそれぞれ渡して利用します。

```
// `baseURL` と `pathname` にあるリソースを取得する
function getResource(baseURL, pathname) {
  const url = baseURL + pathname;
  console.log(url); // => "http://example.com/resouces/example.js"
  // 省略) リソースを取得する処理...
}

const baseURL = "http://example.com/resouces";
const pathname = "/example.js";
getResource(baseURL, pathname);
```

しかし、人によっては、`baseURL` の末尾には / が含むと考える場合もあります。この場合は `getResource` 関数の内部で、`baseURL` と `pathname` を結合してできたURLは異なります。そのため、意図しないURLからリソースを取得するという問題が発生します。

```
// `baseURL` と `pathname` にあるリソースを取得する
function getResource(baseURL, pathname) {
  const url = baseURL + pathname;
  // `/` と `/` が2つ重なってしまっている
  console.log(url); // => "http://example.com/resouces//example.js"
  // 省略) リソースを取得する処理...
}

const baseURL = "http://example.com/resouces/";
const pathname = "/example.js";
getResource(baseURL, pathname);
```

この問題が難しいところは、結合してできた `url` は文字列としては正しいためエラーではないということです。つまり、一見すると問題ないように見えますが、実際に動かしてみて初めて分かるような問題が生じやすいです。

そのため、このような構造的な文字列を扱う場合は、専用の関数や専用のオブジェクトを利用することでより安全に文字列を処理できます。

先ほどのような、URL文字列の結合を安全に行うには、入力される `baseURL` 文字列の揺れを吸収する仕組みを作成します。次の `baseJoin` 関数はベースURLとパスを結合した文字列を返しますが、ベースURLの末尾に / があるかの揺れを吸収しています。

```
// ベースURLとパスを結合した文字列を返す
function baseJoin(baseURL, pathname) {
  // 末尾に / がある場合はそれを削除してから結合する
  const stripSlashBaseUrl = baseURL.replace(/\$/ , "");
  return stripSlashBaseUrl + pathname;
}

// `baseURL` と `pathname` にあるリソースを取得する
function getResource(baseURL, pathname) {
  const url = baseJoin(baseURL, pathname);
  // baseURLの末尾に `/` あってもなくても同じ結果となる
  console.log(url); // => "http://example.com/resouces/example.js"
  // 省略) リソースを取得する処理...
}

const baseURL = "http://example.com/resouces/";
const pathname = "/example.js";
getResource(baseURL, pathname);
```

ECMAScriptの範囲ではありませんが、URLやファイルパスといった典型的なものに対してはすでに専用のものがあります。URLを扱うものとしてブラウザ上のAPIである[URLオブジェクト](#)、Node.jsのコアモジュールである[Pathモジュール](#)などがあります。構造が決まっている文字列において、それ向けの仕組みがある場合はそちらを利用するをお勧めします。

## タグ付きテンプレート関数

文字列操作を行う場合にコンテキストをもつ文字列では気をつける必要があります。しかし、文字列処理をする際に毎回関数で囲んで書くとコードの見た目が分かりにくい場合もあります。

次のようなユーザー入力を受け取り構築されるURLを考えてみましょう。次の `input` にはユーザー入力、つまり外部から受け取った任意の文字列が入ります。

```
// ユーザー入力
const input = "test";
// ユーザー入力を使ってURLを構築
const searchURL = `https://example.com/search?query=${input}&sort=desc`;
```

このとき、単純な文字列結合だとユーザー入力によってはURLが壊れてしまいます。なぜなら、ユーザー入力に `&` や `/` などが含まれているとURLの意味合いが変わってしまったり、URLには含められない文字列があるためです。

```
// ユーザ入力
const input = "/";
// URLエスケープせずに結合した場合
const URL = `https://example.com/search?query=${input}&sort=desc`;
// `query`のパラメータのはずがパスの区切り文字と解釈されてしまう
console.log(URL); // => "https://example.com/search?query=/&sort=desc"
```

そのため、URLのパラメータなどにユーザー入力を含めるためにはURLエスケープする必要があります。JavaScriptでは、`encodeURIComponent` 関数を使うことで文字列をURL中に埋め込んでも安全な文字列へエスケープできます。

```
// ユーザ入力
const input = "/";
// URLエスケープして結合した場合
const URL = `https://example.com/search?query=${encodeURIComponent(input)}&sort=desc`;
// `/`が`%2F`へURLエスケープされている
console.log(URL); // => "https://example.com/search?query=%2F&sort=desc"
```

このように、変数（ユーザー入力など）をURLエスケープしてから文字列結合してもよいのですが、変数の数だけURLエスケープの関数を書く必要があります。そのため、変数をひとつでもURLエスケープし忘れるとななります。

このようなエスケープ忘れなどの問題は、デフォルトを安全側に倒すことでコーディングミスが起きても問題をレベルを小さくする工夫が必要です。ここでは、タグ付きテンプレート（Tagged Template）を使い、テンプレート中の変数を自動でエスケープする処理を加えることができます。

タグ付きテンプレートとは、`タグ関数`テンプレート`` という形式で記述する関数とテンプレートリテラルをあわせた表現です。関数の呼び出しに `タグ関数(`テンプレート`)` ではなく、`タグ関数`テンプレート`` という書式を使っていることに注意してください。

通常の関数として呼び出した場合、引数にはただの文字列が渡ってきます。

```
function tag(string) {
  console.log(string); // => "template 0 literal 1"
}
tag(`template ${0} literal ${1}`);
```

しかし、`()`ではなく `タグ関数`テンプレート`` と記述することで、`タグ関数` が受け取る引数にはタグ付きテンプレート向けの値が渡ってきます。そのため、タグ付きテンプレートで利用する関数のことを、タグ関数（Tag function）と呼び分けることにします。

```
// タグ関数は引数の形が決まっていること以外は関数と同じ
function tag(strings, ...values) {
    // stringsには文字列部分が${}で区切られて順番に入る
    console.log(strings); // => ["template ", " literal "", """]
    // valuesには${}の評価値が順番に入る
    console.log(values); // => [0, 1]
}
tag`template ${0} literal ${1}`;
```

タグ関数の引数は特殊な形をしていることが分かります。引数をどう扱うかを見ていくために、タグ付きテンプレートの内容をそのまま結合して返す `stringRaw` というタグ関数を実装してみます。`Array#reduce` メソッドを使うことで、テンプレートの文字列と変数を順番に結合できます。

```
// テンプレートを順番どおりに結合した文字列を返すタグ関数
function stringRaw(strings, ...values) {
    return strings.reduce((result, string, i) => {
        return result + values[i - 1] + string;
    });
}
// 関数`テンプレートリテラル` という形で呼び出す
stringRaw`template ${0} literal ${1}`; // => "template 0 literal 1"
```

ここで実装した `stringRaw` 関数と同様のものが、`String.raw` という名前でビルトイン関数として提供されています。

```
String.raw`template ${0} literal ${1}`; // => "template 0 literal 1"
```

それでは、テンプレート中の変数をURLエスケープするタグ付きテンプレートを実装してみましょう。

`encodeURIComponent` 関数を利用し、変数をURLエスケープする `escapeURL` タグ関数を定義します。このタグ関数を使うことで、テンプレートリテラル中の変数がURLエスケープできます。

```
// 変数をURLエスケープするタグ関数
function escapeURL(strings, ...values) {
    return strings.reduce((result, string, i) => {
        return result + encodeURIComponent(values[i - 1]) + string;
    });
}

const input = "A&B";
// escapeURLタグ関数を使ったタグ付きテンプレート
const escapedURL = escapeURL`https://example.com/search?q=${input}&sort=desc`;
console.log(escapedURL); // => "https://example.com/search?q=A%26B&sort=desc"
```

このようにタグ付きテンプレートリテラルを使うことで、コンテキストに応じた処理を付け加えることができます。この機能はJavaScript内にHTMLなどの別の言語やDSL（ドメイン固有言語）を埋め込む際に利用されることが多いです。

## おわりに

この章では、JavaScriptにおける文字列とは何かやUnicodeとの関係について紹介しました。文字列処理を行う `String` メソッドにはさまざまなものがあり、正規表現との組み合わせ使うものも含まれます。

正規表現については、正規表現のみでひとつの本が作れるようなJavaScript言語内にある別言語です。 詳細は[正規表現 - JavaScript | MDN](#)などを参照してください。

文字列は一見単純なオブジェクトに見えますが、多くの場合コンテキストを含んでいるため扱い方はさまざまです。 タグ付きテンプレートリテラルを利用することで、テンプレート中の変数は自動でエスケープするといったコンテキストを実現できます。 文字列を安全に扱うためには、コンテキストに応じた処理が必要になります。

## 参考

- [What every JavaScript developer should know about Unicode](#)
- [Unicodeについて](#)
  - 「文字数」ってなあに？～String, NSString, Unicodeの基本～ - Qiita
  - プログラマのための文字コード技術入門 | Gihyo Digital Publishing ... 技術評論社の電子書籍
  - 文字コード「超」研究 改訂第2版【委託】 - 達人出版会
  - Unicode のサロゲートペアとは何か - ひだまりソケットは壊れない
  - 文字って何かね？ - Qiita
  - ものかの >> archive >> Unicode正規化 その I
  - 結合文字列をUnicode正規化で合成する方法の危険性 - Qiita
  - Unicode（絵文字） - CyberLibrarian
  - Unicode Emoji
- [国際化APIについて](#)
  - [Intl - JavaScript | MDN](#)
  - [andyearnshaw/Intl.js](#)
  - [ECMAScript® 2017 Internationalization API Specification](#)
  - [ウェブサイトをグローバル化するために便利なIntl APIの話 - Qiita](#)
  - [カスタムの大文字と小文字の対応規則および並べ替え規則.aspx "カスタムの大文字と小文字の対応規則および並べ替え規則"](#)
- [文字列の検索について](#)
  - [四章第一回 文字列の操作 — JavaScript初級者から中級者になろう — uhyohyo.net](#)

# ラッパーオブジェクト

JavaScriptのデータ型はプリミティブ型とオブジェクトにわけられます。（詳細は「[データ型とリテラル](#)」を参照）

次のコードでは文字列リテラルでプリミティブ型の値である文字列を定義しています。プリミティブ型の値である文字列は `String` オブジェクトのインスタンスではありません。しかし、プリミティブ型の文字列においても、`String` オブジェクトのインスタンスマソッドである `toUpperCase` メソッドを呼び出せます。

```
// String#toUpperCaseを呼び出している
"string".toUpperCase(); // => "STRING"
```

プリミティブ型である文字列が `String` のインスタンスマソッドを呼び出せることは一見不思議です。

この章では、プリミティブ型の値がなぜオブジェクトのメソッドを呼び出せるのかについて解説します。

## プリミティブ型とラッパーオブジェクト

プリミティブ型のデータのうち、真偽値（Boolean）、数値（Number）、文字列（String）、シンボル（Symbol）にはそれぞれ対応するオブジェクトが存在します。たとえば、文字列に対応するオブジェクトとして、`String` オブジェクトがあります。

この `String` オブジェクトを `new` することで `String` オブジェクトのインスタンスを作ることができます。

```
// "string"の値をラップしたStringのインスタンスを生成
const string = new String("string");
// StringのインスタンスマソッドであるtoUpperCaseを呼び出す
string.toUpperCase(); // => "STRING"
```

このようにインスタンス化されたものは、プリミティブ型の値を包んだ（ラップした）オブジェクトといえます。そのため、このようなオブジェクトをプリミティブ型の値に対してのラッパーオブジェクトと呼ばれます。

ラッパーオブジェクトとプリミティブ型の対応は次のとおりです。

ラッパーオブジェクト	プリミティブ型	例
<code>Boolean</code>	真偽値	<code>true</code> や <code>false</code>
<code>Number</code>	数値	<code>1</code> や <code>2</code>
<code>String</code>	文字列	<code>"文字列"</code>
<code>Symbol</code>	シンボル	<code>Symbol("説明")</code>

注記: `undefined` と `null` に対応するラッパーオブジェクトはありません。

ひとつ注意点として、ラッパーオブジェクトは名前のとおりオブジェクトです。そのため、次のように `typeof` 演算子でラッパーオブジェクトを見ると `"object"` です。

```
const string = "文字列";
console.log(typeof string); // => "string";
const stringWrapper = new String("文字列");
console.log(typeof stringWrapper); // => "object";
```

## プリミティブ型の値からラッパーオブジェクトへの自動変換

JavaScriptでは、プリミティブ型の値に対してプロパティアクセスする時、自動で対応するラッパーオブジェクトに変換されます。たとえば `"string"` という文字列は、自動的に `new String("string")` のようなラッパーオブジェクトへ変換されています。これにより、プリミティブ型の値である文字列が `String` のインスタンスメソッドを呼び出すことができます。

```
const str = "string";
// プリミティブ型の値に対してメソッド呼び出しを行うとき、次のような変換が行われる
str.toUpperCase();
// `str`へアクセスする際に、"string"がラッパーオブジェクトへ変換される
// ラッパーオブジェクトはStringのインスタンスなのでメソッドを呼び出せる
(new String(str)).toUpperCase();
```

一方、明示的に作成したラッパーオブジェクトからプリミティブ型の値を取りだすこともできます。

ラッパーオブジェクト `.valueOf` メソッドを呼び出すことで、ラッパーオブジェクトから値を取り出せます。たとえば、次のように文字列のラッパーオブジェクトから `valueOf` メソッドで文字列を取りだせます。

```
const stringWrapper = new String("文字列");
// プリミティブ型の値を取得する
console.log(stringWrapper.valueOf()); // => "文字列"
```

このように、プリミティブ型の値はラッパーオブジェクトへの変換は自動的に行われます。<sup>1</sup>

JavaScriptでは、リテラルを使ったプリミティブ型の文字列とラッパーオブジェクトを使った文字列オブジェクトがあります。（真偽値や数値についても同様です）この2つを明示的に使い分ける利点はないため、常にリテラルを使うことを推奨します。理由として次の3つが挙げられます。

- 必要に応じて、プリミティブ型の文字列は自動的にラッパーオブジェクトに変換されるため
- `new String("string")` のようにラッパーオブジェクトのインスタンスを扱う利点がないため
- ラッパーオブジェクトを `typeof` 演算子で評価した結果が、プリミティブ型ではなく `"object"` となり混乱を生むため

これらの理由などから、プリミティブ型のデータはリテラルを使います。常にリテラルを使うことでラッパーオブジェクトは意識する必要がなくなります。

```
// OK: リテラルを使う
const string = "文字列";
// NG: ラッパーオブジェクトを使う
const stringWrapper = new String("文字列");
```

## まとめ

この章では、プリミティブ型の値がなぜメソッド呼び出しできるのかについて解説しました。その仕組みの背景にはプリミティブ型に対応したラッパーオブジェクトの存在があります。プリミティブ型の値のプロパティへアクセスする際に、自動的にラッパーオブジェクトへ変換されることでメソッド呼び出しなどが可能となっています。

「JavaScriptはすべてがオブジェクトである」と言われることがあります。プリミティブ型はオブジェクトではありませんが、プリミティブ型に対応したラッパーオブジェクトが用意されています。（`null` と `undefined` を除く）そのため、「すべてがオブジェクトのように見える」というのが正しい認識となるでしょう。

<sup>1</sup>. このような値型から参照型の変換は一般にボックス化(ボクシング)、逆の変換はボックス化解除(アンボクシング)と呼ばれます ↪



## 関数とスコープ<sup>°</sup>

定義された関数はそれぞれのスコープを持っています。スコープとは変数や関数の引数などを参照できる範囲を決めるものです。JavaScriptでは、新しい関数を定義するとその関数に紐付けられた新しいスコープを作成します。関数を定義するということは処理をまとめるというだけではなく、変数が有効な範囲を決める新しいスコープを作っているといえます。

スコープの仕組みを理解することは関数をより深く理解することにつながります。なぜなら関数とスコープは密接な関係を持っているためです。この章では関数とスコープの関係を中心に、スコープとはどのような働きをしていて、スコープ内では変数の名前から値がどのように取得されているのかを見ていきます。

JavaScriptのスコープは、ES2015において直感的に理解しやすい仕組みが整備されました。基本的にはES2015以降の仕組みを理解していればコードを書く場合には問題ありません。

しかし、既存のコードを理解するためには、ES2015よりも前に決められた古い仕組みについても知る必要があります。なぜなら、既存のコードは古い仕組みを使って書かれていることもあるためです。また、JavaScriptでは古い仕組みと新しい仕組みを混在して書くことができます。古い仕組みによるスコープは直感的でない挙動も多いため、コラムで補足していきます。

### スコープとは

スコープとは変数の名前や関数などの参照できる範囲を決めるものです。スコープの中で定義された変数はスコープ内でのみ参照でき、スコープの外側からは参照できません。

身近なスコープの例として関数によるスコープを見ていきます。次のコードには、`fn` 関数のブロック（`{` と `}`）内で変数 `x` を定義しています。この変数 `x` は `fn` 関数のスコープに定義されているため、`fn` 関数の内側では参照できます。一方、`fn` 関数の外側から変数 `x` は参照できないため `ReferenceError` をなげます。

```
function fn() {
  const x = 1;
  // fn関数のスコープ内から`x`は参照できる
  console.log(x); // => 1
}
fn();
// fn関数のスコープ外から`x`は参照できないためエラー
console.log(x); // => ReferenceError: x is not defined
```

このコードを見て分かるように、変数 `x` は `fn` 関数のスコープに紐付けて定義されます。そのため、変数 `x` は `fn` 関数のスコープ内でのみ参照できます。

関数は仮引数をもつことができますが、仮引数は関数のスコープに紐付けて定義します。そのため、仮引数はその関数の中でのみ参照が可能で、関数の外からは参照できません。

```
function fn(arg) {
  // fn関数のスコープ内から仮引数`arg`は参照できる
  console.log(arg); // => 1
}
fn(1);
// fn関数のスコープ外から`arg`は参照できないためエラー
console.log(arg); // => ReferenceError: arg is not defined
```

この関数によるスコープのことを関数スコープと呼びます。

変数と宣言の章にて、`let` や `const` は同じスコープ内に同じ名前の変数を二重に定義できないという話をしました。これは、各スコープには同じ名前の変数は1つしか宣言できないためです。（`var` による変数宣言と `function` による関数宣言は例外的に可能です）

```
// スコープ内に同じ"x"を定義すると SyntaxError となる
let x;
let x;
```

一方、スコープが異なれば同じ名前で変数を宣言できます。次の例では、`fnA` 関数と `fnB` 関数という異なるスコープで、それぞれ変数 `x` を定義できていることが分かります。

```
// 異なる関数のスコープには同じ"x"を定義できる
function fnA() {
  let x;
}

function fnB() {
  let x;
}
```

このように、スコープが異なれば同じ名前の変数を定義できます。スコープの仕組みがないと、グローバルな空間な一意な変数名を考える必要があります。スコープがあることで適切な名前の変数を定義できるようになるため、スコープの役割は重要です。

## ブロックスコープ

{ と } で囲んだ範囲をブロックと呼びます。（「文と式」の章を参照）ブロックもスコープを作成します。ブロック内で宣言された変数は、スコープ内でのみ参照でき、スコープの外側からは参照できません。

```
// ブロック内で定義した変数はスコープ内でのみ参照できる
{
  const x = 1;
  console.log(x); // => 1
}
// スコープの外から`x`を参照できないためエラー
console.log(x); // => ReferenceError: x is not defined
```

ブロックによるスコープのことをブロックスコープと呼びます。

`if`文や`while`文などもブロックスコープを作成します。単独のブロックと同じく、ブロックの中で宣言した変数は外から参照できません。

```
// if文のブロック内で定義した変数はブロックスコープの中でのみ参照できる
if (true) {
  const x = "inner";
  console.log(x); // => "inner"
}
console.log(x); // => ReferenceError: x is not defined
```

`for`文は、ループごとに新しいブロックスコープを作成します。このことは「各スコープには同じ名前の変数は1つしか宣言できない」のルールを考えてみると分かりやすいです。次のコードでは、ループ毎に `const` で `element` 変数を定義していますが、エラーなく定義できています。これは、ループ毎に別々のブロックスコープが作成され、変数の宣言もそれぞれ別々のスコープで行われるためです。

```
const array = [1, 2, 3, 4, 5];
// ループごとに新しいブロックスコープを作成する
for (const element of array) {
```

```
// forのブロックスコープの中でのみ`element`を参照できる
console.log(element);
}

// ループの外からはブロックスコープ内の変数は参照できない
console.log(element); // => ReferenceError: element is not defined
```

## スコープチェーン

関数やブロックはネスト（入れ子）して書けますが、同様にスコープもネストできます。次のコードではブロックの中にブロックを書いています。このとき外側のブロックスコープのことを `OUTER`、内側のブロックスコープのことを `INNER` と呼ぶことにします。

```
{
  // OUTERブロックスコープ
  {
    // INNERブロックスコープ
  }
}
```

スコープがネストしている場合に、内側のスコープから外側のスコープにある変数を参照できます。次のコードでは、内側のINNERブロックスコープから外側のOUTERブロックスコープに定義されている変数 `x` を参照できます。これは、ブロックスコープに限らず関数スコープでも同様です。

```
{
  // OUTERブロックスコープ
  const x = "x";
  {
    // INNERブロックスコープからOUTERブロックスコープの変数を参照できる
    console.log(x); // => "x"
  }
}
```

このとき、現在のスコープ（変数を参照する式が書かれているスコープ）から外側のスコープへと順番に変数が定義されているかを確認します。このとき、内側のINNERブロックスコープには変数 `x` はありませんが外側のOUTERブロックスコープに変数 `x` が定義されているため参照できます。つまり次のようなステップで参照したい変数を探索しています。

1. INNERブロックスコープに変数 `x` があるかを確認 => ない
2. ひとつ外側のOUTERブロックスコープに変数 `x` があるかを確認 => ある

一方、現在のスコープも含めどの外側のスコープに該当する変数が定義されていない場合は、`ReferenceError` の例外が発生します。次の例では、どのスコープにも存在しない `xyz` を参照しているため、`ReferenceError` の例外が発生します。

```
{
  // OUTERブロックスコープ
  {
    // INNERブロックスコープ
    console.log(xyz); // => ReferenceError: xyz is not defined
  }
}
```

このときも、現在のスコープ（変数を参照する式が書かれているスコープ）から外側のスコープへと順番に変数が定義されているかを確認します。しかし、どのスコープにも変数 `xyz` は定義されていないため、`ReferenceError` の例外が発生します。つまり次のようなステップで参照したい変数を探索しています。

1. INNERブロックスコープに変数 `xyz` があるかを確認 => ない

2. ひとつ外側のOUTERブロックスコープに変数 xyz があるかを確認 => ない
3. 一番外側のスコープにも変数 xyz は定義されていない => ReferenceError が発生

この内側から外側のスコープへと順番に変数が定義されているか探す仕組みのことをスコープチェーンと呼びます。

内側と外側のスコープ両方に同じ名前の変数が定義されている場合もスコープチェーンの仕組みで解決できます。次のコードでは、内側のINNERブロックスコープと外側のOUTERブロックスコープに同じ名前の変数 x が定義されています。スコープチェーンの仕組みにより、現在のスコープに定義されている変数 x を優先的に参照します。

```
{
  // OUTERブロックスコープ
  const x = "outer";
  {
    // INNERブロックスコープ
    const x = "inner";
    // 現在のスコープ(INNERブロックスコープ)にある`x`を参照する
    console.log(x); // => "inner"
  }
  // 現在のスコープ(OUTERブロックスコープ)にある`x`を参照する
  console.log(x); // => "outer"
}
```

このようにスコープは階層的な構造となっており、その際にどの変数が参照できるかはスコープチェーンによって解決されています。

## グローバルスコープ<sup>°</sup>

今までコードをプログラム直下に書いていましたが、ここにも暗黙的なグローバルスコープ（大域スコープ）と呼ばれるスコープが存在します。グローバルスコープとは名前のとおりもっとも外側にあるスコープで、プログラム実行時に暗黙的に作成されます。

```
// プログラム直下はグローバルスコープ
const x = "x";
console.log(x);
```

グローバルスコープに定義した変数はグローバル変数と呼ばれ、グローバル変数はあらゆるスコープから参照できる変数となります。なぜなら、スコープチェーンの仕組みにより、最終的にもっとも外側のグローバルスコープに定義されている変数を参照できるためです。

```
// グローバル変数はどのスコープからも参照できる
const globalVariable = "グローバル";

{
  // ブロックスコープ内には該当する変数が定義されてない -> 外側のスコープへ
  console.log(globalVariable); // => "グローバル"
}
function fn() {
  // 関数ブロックスコープ内には該当する変数が定義されてない -> 外側のスコープへ
  console.log(globalVariable); // => "グローバル"
}
fn();
```

グローバルスコープには自分で定義したグローバル変数以外に、プログラム実行時に自動的に定義されるビルトインオブジェクトがあります。ビルトインオブジェクトには大きく分けて2種類のものがあります。1つ目はECMAScript仕様が定義する `undefined` のような変数（「[undefinedはリテラルではない](#)」を参照）や `isNaN` のような関

数、`Array` や `RegExp` などのコンストラクタ関数です。もう一方は実行環境（ブラウザやNode.jsなど）が定義するオブジェクトで `document` や `module` などがあります。どちらもグローバルスコープに自動的に定義されているという点で大きな使い分けはないため、この章ではどちらもビルトインオブジェクトと呼ぶことにします。

ビルトインオブジェクトは、プログラム開始時にグローバルスコープへ自動的に定義されているためどのスコープからも参照できます。

```
// ビルトインオブジェクトは実行環境が自動的に定義している
// どこのスコープから参照してもReferenceErrorにはならない
console.log(undefined); // => undefined
console.log(Array); // => Array
```

自分で定義したグローバル変数とビルトインオブジェクトでは、グローバル変数が優先して参照されます。つまり次のようにビルトインオブジェクト同じ名前の変数を定義すると、定義した変数が参照されます。

```
// "Array"という名前の変数を定義
const Array = 1;
// 自分で定義した変数がビルトインオブジェクトより優先される
console.log(Array); // => 1
```

ビルトインオブジェクトと同じ名前の変数を定義したことにより、ビルトインオブジェクトを参照できなくなる問題は変数の隠蔽（shadowing）とも呼ばれます。この問題を回避する方法としては、むやみにグローバルスコープへ変数を定義しないことです。グローバルスコープでビルトインオブジェクトと名前が衝突するとすべてのスコープへ影響を与えますが、関数のスコープ内ではその関数の中だけの影響範囲はとどまります。

ビルトインオブジェクトと同じ名前を避けることは難しいです。なぜならビルトインオブジェクトは実行環境（ブラウザやNode.jsなど）がそれぞれ独自に定義したものが多く存在します。そのため、関数などを活用し小さなスコープを中心にしてプログラムを書くことで、ビルトインオブジェクトと同じ名前の変数があつても影響範囲が限定的な状態にすることが望ましいです。

## [コラム] 変数を参照できる範囲を小さくする

グローバル変数に限らず、特定の変数を参照できる範囲を小さくすることはよいことです。なぜなら、現在のスコープの変数を参照するつもりがグローバル変数を参照したり、その逆も起きることがあるからです。あらゆる変数がグローバルスコープにあると、どこでその変数が参照されているのかを把握できなくなります。これを避けるシンプルな考え方は、変数はできるだけ利用する近くのスコープ内に定義するということです。

次のコードでは、`doHeavyTask` 関数の実行時間を計測しようとしています。`Date.now` メソッドは現在の時刻をミリ秒にして返す関数で、実行後の時刻から実行前の時刻を引くことで間に行われた処理の実行時間を得ることができます。

```
function doHeavyTask() {
  // 計測したい処理
}
const startTime = Date.now();
doHeavyTask();
const endTime = Date.now();
console.log(`実行時間は${endTime - startTime}ミリ秒`);
```

このコードでは、計測処理以外で利用しない `startTime` と `endTime` という変数がグローバルスコープに定義されています。プログラム全体が短い場合はあまり問題になりませんが、プログラムが長くなっていくにつれ影響の範囲が広がっていきます。この2つの変数を参照できる範囲を小さくする簡単な方法はこの実行時間を計測する処理を関数にすることです。

```
// 実行時間を計測したい関数を引数に渡す
```

```

const measureTask = (taskFn) => {
  const startTime = Date.now();
  taskFn();
  const endTime = Date.now();
  console.log(`実行時間は${endTime - startTime}ミリ秒`);
};

function doHeavyTask() {
  // 計測したい処理
}

measureTask(doHeavyTask);

```

これにより、`startTime` と `endTime` という変数を外側のスコープから参照できなくなりました。また、実行時間を計測するという処理を関数にすることで再利用できます。

これは単純なように見えますが、コードの量が増えていくにつれ、人が一度に把握できる量にも限界がやってきます。そのため、人が一度に把握できる範囲のサイズに処理をまとめていくことが必要です。この問題を解決するアプローチとして、変数の参照できる範囲を小さくすることや処理を関数にまとめるという手法がよく利用されます。

## 関数スコープとvarの巻き上げ

変数宣言には `var`、`let`、`const` が利用できます。[変数と宣言](#)の章において、`let`は「よりよい `var`」と紹介したように、`var` を改善する目的で導入された構文です。`const` は再代入できないという点以外は `let` と同じ動作になります。そのため、`let` が使える場合に `var` を使う理由はありませんが、既存のコードや既存のライブラリなどでは `var` が利用されている場面もあるため、`var` の動作を理解する必要があります。

まず最初に、`let` と `var` で共通する動作を見ていきます。`let` と `var` どちらも、初期値を指定せずに宣言した変数の評価結果は暗黙的に `undefined` になります。また、`let` と `var` どちらも、変数宣言をした後に値を代入できます。

次のコードでは、それぞれ初期値を持たない変数を宣言した後に参照すると、変数の評価結果は `undefined` となっています。

```

let let_x;
var var_x;
// 宣言後にそれぞれの変数を参照すると`undefined`となる
console.log(let_x); // => undefined
console.log(var_x); // => undefined
// 宣言後に値を代入できる
let_x = "letのx";
var_x = "varのx";

```

次に、`let` と `var` で異なる動作を見ていきます。

`let` では、変数を宣言する前にその変数を参照すると `ReferenceError` となります。次のコードでは、変数を宣言する前に、変数 `x` を参照したため `ReferenceError` となっています。エラーメッセージから、変数 `x` が存在しないからエラーになっているのではなく、実際に宣言した行よりも前に参照したためエラーとなることが分かります。[TDZ](#)

```

console.log(x); // => ReferenceError: can't access lexical declaration `x` before initialization
let x = "letのx";

```

一方 `var` では、変数を宣言する前にその変数を参照しても `undefined` となります。次のコードは、変数を宣言する前に参照しているにもかかわらずエラーにはならず、変数 `x` の評価結果は `undefined` となります。

```

// var宣言よりも前に参照してもエラーにならない
console.log(x); // => undefined
var x = "varのx";

```

このように `var` で宣言された変数が宣言前に参照でき、その値が `undefined` となる特殊な動きをしていることが分かります。

この `var` の振る舞いを理解するために、変数宣言が宣言と代入の2つの部分から構成されていると考えてみましょう。`var` による変数宣言は、暗黙的に宣言部分がもっと近い関数またはグローバルスコープの先頭に巻き上げられ、代入部分はそのままの位置に残るという特殊な動作をします。

この動作により、変数 `x` を参照するコードよりも前に変数 `x` の宣言部分が移動し、変数 `x` の評価結果は暗黙的に `undefined` となっています。つまり、先ほどのコードは実際の実行時には、次のように解釈されて実行されていると考えられます。

```
// 解釈されたコード
// スコープの先頭に宣言部分が巻き上げられる
var x;
console.log(x); // => undefined
// 変数への代入はそのままの位置に残る
x = "varのx";
console.log(x); // => "varのx"
```

さらに、`var` 変数の宣言の巻き上げは、ブロックスコープを無視してもっと近い関数またはグローバルスコープに変数を紐付けます。そのため、次のようにブロック {} で `var` による変数宣言を囲んでも、もっとも近い関数スコープである `fn` 関数の直下に宣言部分が巻き上げられます。（if文やfor文におけるブロックスコープも同様に無視されます）

```
function fn() {
  // 内側のスコープにあるはずの変数 `x` が参照できる
  console.log(x); // => undefined
  {
    var x = "varのx";
  }
  console.log(x); // => "varのx"
}
fn();
```

つまり、先ほどのコードは実際の実行時には、次のように解釈されて実行されていると考えられます。

```
// 解釈されたコード
function fn() {
  // もっと近い関数スコープの先頭に宣言部分が巻き上げられる
  var x;
  console.log(x); // => undefined
  {
    // 変数への代入はそのままの位置に残る
    x = "varのx";
  }
  console.log(x); // => "varのx"
}
fn();
```

この変数の宣言部分がもっと近い関数またはグローバルスコープの先頭に移動しているように見える動作のことを変数の巻き上げ（hoisting）と呼びます。

このように `let`、`const` に対して `var` は異なった動作をしています。`var` は巻き上げによりブロックスコープを無視して、宣言部分を自動的にスコープの先頭に移動します。もっとも簡単な回避方法は `var` を使わないのですが、`var` を含んだコードではこの動作に気をつける必要があります。

## 関数宣言と巻き上げ

`function` キーワードを使った関数宣言も `var` と同様に、もっと近い関数またはグローバルスコープの先頭に巻き上げされます。次のコードでは、実際に `hello` 関数を宣言した行より前に関数を呼び出せます。

```
// `hello`関数の宣言よりも前に呼び出せる
hello(); // => "Hello"

function hello(){
    return "Hello";
}
```

これは、関数宣言は宣言そのものであるため、`hello` 関数そのものがスコープの先頭に巻き上げされます。つまり、先ほどのコードは実際の実行時には、次のように解釈されて実行されていると考えられます。

```
// 解釈されたコード
// `hello`関数の宣言が巻き上げされる
function hello(){
    return "Hello";
}

hello(); // => "Hello"
```

注意点として、`var` や `let` などで宣言された変数へ関数を代入した場合は `var` のルールで巻き上げされます。そのため、`var` で変数へ関数を代入する関数式では、`hello` 変数が巻き上げにより `undefined` となるため呼び出すことができません。（「[関数と宣言（関数式）](#)」を参照）

```
// `hello`変数は巻き上げされ、暗黙的に`undefined`となる
hello(); // => TypeError: hello is not a function

// `hello`変数へ関数を代入している
var hello = function(){
    return "Hello";
};
```

## [コラム] 即時実行関数

即時実行関数（IIFE, *Immediately-Invoked Function Expression*）は、グローバルスコープの汚染を避けるために生まれたイディオムです。

次のように、匿名関数を宣言した直後に呼び出すことで、任意の処理を関数のスコープに閉じて実行できます。関数スコープを作ることで `foo` 変数は匿名関数の外側からはアクセスできません。

```
(function() {
    // 関数のスコープ内でfoo変数を宣言している
    var foo = "foo";
    console.log(foo); // => "foo"
})();
// foo変数のスコープ外
console.log(typeof foo === "undefined"); // => true
```

関数を式として定義そのまま呼び出しています。`function` から始まってしまうとJavaScriptエンジンが関数宣言と解釈してしまうため、無害な括弧などで囲み関数式として解釈させるのが特徴的な記法です。これは次のように書いた場合と意味は同じですが、匿名関数を定義して実行するためより短く書けます。

```
function fn() {
    var foo = "foo";
    console.log(foo); // => "foo"
}
fn();
```

```
// foo変数のスコープ外
console.log(typeof foo === "undefined"); // => true
```

ECMAScript 5までは、変数を宣言する方法は `var` しか存在しません。即時実行関数は `var` によるグローバルスコープの汚染を防ぐために必要でした。

しかしECMAScript 2015で導入された `let` と `const` により、ブロックスコープに対して変数宣言できるようになりました。そのため、グローバルスコープの汚染を防ぐための即時実行関数は不要です。先ほどの即時実行関数は次のように `let` や `const` とブロックスコープで書き換えられます。

```
{
  // ブロックスコープ内でfoo変数を宣言している
  const foo = "foo";
  console.log(foo); // => "foo"
}
// foo変数のスコープ外
console.log(typeof foo === "undefined"); // => true
```

## クロージャー

最後にこの章ではクロージャーと呼ばれる関数とスコープに関する性質について見ていきます。クロージャーとは「外側のスコープにある変数への参照を保持できる」という関数がもつ性質のことです。

クロージャーは言葉で説明しただけでは分かりにくい性質です。このセクションでは、クロージャを使ったコードがどのようにして動くのかを理解することを目標にします。

次の例では `createCounter` 関数は、関数内で定義した `increment` 関数を返しています。その返された `increment` 関数を `myCounter` 変数を代入しています。この `myCounter` 変数を実行するたびに1,2,3と1ずつ増えた値を返しています。

さらに、もう一度 `createCounter` 関数を実行しその返り値を `newCounter` 変数に代入します。`newCounter` 変数も実行するたびに1ずつ増えていますが、`myCounter` 変数とその値を共有しているわけではないことが分かります。

```
// `increment`関数を定義し返す関数
function createCounter() {
  let count = 0;
  // `increment`関数は`count`変数を参照
  function increment() {
    count = count + 1;
    return count;
  }
  return increment;
}
// `myCounter`は`createCounter`が返した関数を参照
const myCounter = createCounter();
myCounter(); // => 1
myCounter(); // => 2
// 新しく`newCounter`を定義する
const newCounter = createCounter();
newCounter(); // => 1
newCounter(); // => 2
// `myCounter`と`newCounter`は別々の状態持っている
myCounter(); // => 3
newCounter(); // => 3
```

このように、まるで関数が状態（ここでは1ずつ増える `count` という値）を持っているように振る舞える仕組みの背景にはクロージャがあります。クロージャーを直感的には理解しにくいため、まずはクロージャを理解するために必要な「静的スコープ」と「メモリ管理の仕組み」について見ていきます。

## 静的スコープ<sup>°</sup>

クロージャーを理解するために、今まで意識したことなかったスコープの性質について見ていきます。JavaScriptのスコープでは、どの識別子がどの変数を参照するかが静的に決定されるという性質を持ちます。つまり、コードを実行する前にどの識別子がどの変数を参照しているかが分かるということです。

次のような例を見てみます。`printx` 関数内で変数 `x` を参照していますが、変数 `x` はグローバルスコープと関数 `run` の中にそれぞれ定義されています。このとき `printx` 関数内の `x` という識別子がどの変数 `x` を参照するかは静的に決定されます。

結論からいえば、`printx` 関数内にある識別子 `x` はグローバルスコープ (\*1) の変数 `x` を参照します。そのため、`printx` 関数の実行結果は常に `10` となります。

```
const x = 10; // *1

function printx() {
    // この識別子`x`は常に *1 の変数`x`を参照する
    console.log(x); // => 10
}

function run() {
    const x = 20; // *2
    printx(); // 常に10が表示される
}

run();
```

スコープチェーンの仕組みを思い出すと、この識別子 `x` は次のように名前解決されグローバルスコープの変数 `x` を参照することが分かります。

1. `printx` の関数スコープに変数 `x` が定義されていない
2. ひとつ外側のスコープ（グローバルスコープ）を確認する
3. ひとつ外側のスコープに `const x = 10;` が定義されているので、識別子 `x` はこの変数を参照する

つまり、`printx` 関数内に書かれた `x` という識別子は、`run` 関数を実行されるかは関係なく、静的に \*1 で定義された変数 `x` を参照することが決定されます。このように、どの識別子がどの変数を参照しているかを静的に決定する性質を静的スコープと呼びます。

この静的スコープの仕組みは `function` キーワードを使った関数宣言、メソッド、Arrow Functionなどすべての関数で共通する性質です。

## [コラム] 動的スコープ

多くの言語は静的スコープですが、BashやPerl4などは呼び出し元によって識別子がどの変数を参照するかが変わる仕組みを持っています。

次のコードは、仮にJavaScriptが呼び出し元によって参照する変数が変わることの結果を表した擬似的なコードです。JavaScriptは静的スコープであるので、実際には次のような結果にはなりません。識別子 `x` が呼び出し元のスコープを参照する仕組みである場合には次のような結果になります。

```
// JavaScriptが静的スコープでないとした場合の擬似的なコード例
const x = 10; // *1

function printx() {
    // この識別子`x`は呼び出し元によってどの変数`x`を参照するかが変わる
    console.log(x);
}

function run() {
    const x = 20;
```

```
//呼び出し元で変数`x`を定義している
printX();
}

printX(); //ここでは 10 が表示される
run(); //ここでは 20 が表示される
```

このように呼び出し方によって動的に参照する変数が変わる仕組みのことを動的スコープと呼びます。

JavaScriptは変数や関数の参照先は静的スコープの性質によって決定されます。しかし `this` という特別なキーワードだけは動的スコープのように、呼び出し元によって参照先が変わります。この `this` というキーワードについては次章で解説します。

## メモリ管理の仕組み

ほとんどのプログラミング言語では使わなくなった変数やデータを解放する仕組みを持っています。なぜなら、変数や関数を定義すると定義されたデータはメモリ上に確保されますが、ハードウェアのメモリは有限であるためです。そのため、メモリからデータが溢れないようにするために必要なタイミングで不要なデータをメモリから解放する必要があります。

不要なデータをメモリから解放する方法は言語によって異なりますが、JavaScriptではガベージコレクションが採用されています。ガベージコレクションとは、どこからも参照されなくなったデータを不要なデータと判断して自動でメモリ上から解放する仕組みのことです。

JavaScriptではガベージコレクションがあるため、手動でメモリから解放するコードは書く必要がありません。しかし、ガベージコレクションといったメモリ管理の仕組みを理解することは、スコープやクロージャーに関係するため大切です。

どのようなタイミングでメモリ上から不要なデータが解放されるのか、具体的な例を見てみましょう。

次の例では、最初に `"before text"` という文字列のデータがメモリ上に確保され、変数 `x` はそのメモリ上のデータを参照しています。その後、`"after text"` という新しい文字列のデータを作り、変数 `x` はその新しいデータへ参照先を変えています。

このとき、最初にメモリ上へ確保した `"before text"` という文字列のデータはどこからも参照されなくなっています。どこからも参照されなくなった時点で不要になったデータと判断されるためガベージコレクションの回収対象となります。その後、任意のタイミングでガベージコレクションによって回収されメモリ上から解放されます。[GC](#)

```
let x = "before text";
//変数`x`に新しいデータを代入する
x = "after text";
//このとき"before text"というデータはどこからも参照されなくなる
//その後、ガベージコレクションによってメモリ上から解放される
```

次にこのガベージコレクションと関数の関係性について考えてみましょう。よくある誤解として「関数の中で作成したデータは、その関数が実行し終了したら解放される」という誤解があります。関数の中で作成したデータは、その関数の実行が終了した時点では必ずしも解放されるわけではありません。

具体的に、「関数の実行が終了した際に解放される場合」と「関数の実装が終了しても解放されない場合」の例をそれぞれ見ていきましょう。

まずは、関数の実行が終了した際に解放されるデータの例です。次のコードでは、`printX` 関数の中で変数 `x` を定義しています。この変数 `x` は、`printX` 関数が実行されるたびに定義され、実行終了後にどこからも参照されなくなります。どこからも参照できなくなったものは、ガベージコレクションによって回収されメモリ上から解放されます。

```
function printX() {
  const x = "X";
```

```

    console.log(x); // => "X";
}

printX();
// この時点で`"X"`を参照するものはなくなる -> 解放される

```

次に、関数の実行が終了しても解放されないデータの例です。次のコードでは、`createArray` 関数の中で定義された変数 `tempArray` は、`createArray` 関数の返り値となっています。この、関数で定義された変数 `tempArray` は返り値として、別の変数 `array` に代入されています。つまり、変数 `tempArray` が参照している配列オブジェクトは、`createArray` 関数の実行終了後も変数 `array` から参照され続けています。ひとつでも参照されているならば、そのデータは自動的に解放されることはありません。

```

function createArray() {
  const tempArray = [1, 2, 3];
  return tempArray;
}
const array = createArray();
console.log(array); // => [1, 2, 3]
// 変数`array`が`[1, 2, 3]`という値を参照してる -> 解放されない

```

つまり、関数の実行が終了したことと関数内で定義したデータの解放のタイミングは直接関係ないことが分かります。そのデータがメモリ上から解放されるかどうかはあくまで、そのデータが参照されているかによって決定されます。

## クロージャーがなぜ動くのか

ここまでで「静的スコープ」と「メモリ管理の仕組み」について説明してきました。

- 静的スコープ: ある変数がどの値を参照するかは静的に決まる
- メモリ管理の仕組み: 参照されなくなったデータはガベージコレクションにより解放される

クロージャーとはこの2つの仕組みを利用して、関数内から特定の変数を参照し続けることで関数が状態をもつことができる仕組みのことを言います。

最初にクロージャーの例として紹介した `createCounter` 関数の例を改めて見てみましょう。

```

const createCounter = () => {
  let count = 0;
  return function increment() {
    // `increment`関数は外のスコープの変数`count`を参照している
    // これがクロージャーと呼ばれる
    count = count + 1;
    return count;
  };
}
// createCounter()の実行結果は、内側で定義されていた`increment`関数
const myCounter = createCounter();
// myCounter関数の実行結果は`count`の評価結果
myCounter(); // => 1
myCounter(); // => 2

```

つまり次のような参照の関係が `myCounter` 変数と `count` 変数の間にはあることがわかります。

- `myCounter` 変数は `createCounter` 関数の返り値である `increment` 関数を参照している
- `myCounter` 変数は `increment` 関数を経由して `count` 変数を参照している
- `myCounter` 変数実行した後も `count` 変数を参照している

`count` 変数を参照するものがいるため、`count` 変数は自動的に解放されません。そのため `count` 変数の値は保持され続け、`myCounter` 変数を実行するたびに1ずつ大きくなっています。

```
myCounter -> increment -> count
```

このように `count` 変数が自動解放されずに保持できているのは「( `increment` ) 関数が外側のスコープにある (`count`) 変数への参照を保持できる」ためです。このような性質のことをクロージャー（関数閉包）と呼びます。クロージャーは静的スコープと変数は参照され続ければデータは保持されるという2つの性質によって成り立っています。

JavaScriptの関数は静的スコープとメモリ管理という2つの性質を常に持っています。そのため、ある意味ではすべての関数がクロージャーとなります。ここでは関数が特定の変数を参照することで関数が状態をもっていることを指すことにします。

先ほどの例では `createCounter` 関数を実行するたびに、それぞれ `count` と `increment` 関数が定義しています。そのため、`createCounter` 関数の実行するとそれぞれ別々の `increment` 関数が定義され、別々の `count` 変数を参照しています。

次のように `createCounter` 関数を複数回呼び出してみると、別々の状態を持っていることが確認できます。

```
const createCounter = () => {
  let count = 0;
  return function increment() {
    // 変数`count`を参照し続けている
    count = count + 1;
    return count;
  };
};

// countUpとnewCountUpはそれぞれ別のincrement関数(内側にあるのも別のcount変数)
const countUp = createCounter();
const newCountUp = createCounter();
// 参照している関数(オブジェクト)は別であるため==は一致しない
console.log(countUp === newCountUp); // false
// それぞれの状態も別となる
countUp(); // => 1
newCountUp(); // => 1
```

## クロージャーの用途

クロージャーはさまざまな用途に利用されますが、次のような用途で利用されることが多いです。

- 関数に状態を持たせる手段として
- 外から参照できない変数を定義する手段として
- グローバル変数を減らす手段として
- 高階関数の一部部分として

これらはクロージャーの特徴でもあるので、同時に使われることがあります。

たとえば次の例では、`privateCount` という変数を関数の中に定義していますが、外からはその変数を直接参照はできません。言い換えると外から直接参照して値を変更することはできません。外から参照する必要がない変数をクロージャーとなる関数に閉じ込めることは、言い換えるとグローバルに定義する変数を減らすことができます。

```
const createCounter = () => {
  // 外のスコープから`privateCount`を直接参照できない
  let privateCount = 0;
  return () => {
    privateCount++;
    return `${privateCount}回目`;
  };
};

const counter = createCounter();
counter(); // => "1回目"
counter(); // => "2回目"
```

また、関数を返す関数のことを高階関数と呼びますが、クロージャの性質を使うことで次のように `n` より大きいかを判定する高階関数を作れます。最初から `greaterThan5` という関数を定義すればよいのですが、高階関数を使うことで文字列などの値と同じように関数を値としてやり取りできます。

```
function greaterThan(n) {
  return function(m) {
    return m > n;
  };
}
const greaterThan5 = greaterThan(5);
greaterThan5(5); // => false
greaterThan5(6); // => true
```

## [コラム] 状態をもつ関数オブジェクト

JavaScriptでは関数はオブジェクトの一種です。オブジェクトであるということは直接プロパティに値を入れることができます。つまり、状態を関数をもつ方法として、次のように直接関数に値を代入するという手段をとることができます。

```
function countUp() {
  // countプロパティを参照して変更する
  countUp.count = countUp.count + 1;
  return countUp.count;
}
// 関数オブジェクトにプロパティとして値を代入する
countUp.count = 0;
// 呼び出すことにcountが更新される
countUp(); // => 1
countUp(); // => 2
```

しかし、この方法は推奨されていません。理由としては、外から直接 `count` プロパティが変更できるためです。関数オブジェクトのプロパティは外からも参照でき、またそのプロパティ値を変更できます。その値を外から見えないように隠しているつもりなら、それを強制できるクロージャーが有効です。

```
function countUp() {
  // countプロパティを参照して変更する
  countUp.count = countUp.count + 1;
  return countUp.count;
}
countUp.count = 0;
// 呼び出すことにcountが更新される
countUp(); // => 1
// 直接値を変更できてしまう
countUp.count = 10;
countUp(); // => 11
```

## クロージャーのまとめ

クロージャーは、変数が参照する値は静的に決まる静的スコープという性質とデータは参照されていれば保持されるという2つの性質によって成り立っています。JavaScriptでは、関数を短く定義できるArrow Functionや高階関数であるメソッドなどクロージャーを自然と利用しやすい環境があります。関数を理解する上ではクロージャーを理解することは大切です。

TDZ. この仕組みはTemporal Dead Zoneと呼ばれます。 ↪

GC. ECMAScriptの仕様ではガベージコレクションの実装の規定はないため、実装依存の処理となります ↪



## 関数とthis

この章では `this` という特殊な動作をするキーワードについて見ていきます。 `this` は基本的にはメソッドの中で利用しますが、`this` は読み取り専用のグローバル変数のようなものでどこにでも書くことができます。加えて、`this` の参照先（評価結果）は条件によって異なります。

`this` の参照先は主に次の条件によって変化します。

- 実行コンテキストにおける `this`
- コンストラクタにおける `this`
- 関数とメソッドにおける `this`
- Arrow Functionにおける `this`

コンストラクタにおける `this` は次章のクラスで扱います。もっとも複雑な条件が存在するのは「関数とメソッドにおける `this`」です。そのためこの章では関数と `this` の関係を主に扱います。

この章では、さまざまな条件下で変わる `this` の参照先と関数やArrow Functionとの関係を見ていきます。

## 実行コンテキストと `this`

最初にJavaScriptとはの章において、JavaScriptには実行コンテキストとして"Script"と"Module"があるという話をしました。トップレベル（外側がグローバルスコープの場所）にある `this` は、実行コンテキストによって値が異なります。実行コンテキストの違いは意識しにくい部分であり、トップレベルで `this` を使うことは混乱を生むことになります。そのため、コードのトップレベルにおいては `this` を使うべきではありませんが、それぞれの実行コンテキストにおける動作を紹介します。

### スクリプトにおける `this`

実行コンテキストが"Script"である場合、そのコード直下に書かれた `this` はグローバルオブジェクトを参照します。グローバルオブジェクトとは、実行環境において異なるものが定義されています。ブラウザなら `window` オブジェクト、Node.jsなら `global` オブジェクトとなります。

ブラウザでは、`script` 要素の `type` 属性を指定しない場合は実行コンテキストが"Script"として実行されます。この `script` 要素の直下に書いた `this` はグローバルオブジェクトである `window` オブジェクトとなります。

```
<script>
// 実行コンテキストは"Script"
console.log(this); // => window
</script>
```

### モジュールにおける `this`

実行コンテキストが"Module"である場合、そのコード直下に書かれた `this` は常に `undefined` となります。

ブラウザでは、`script` 要素の `type="module"` 属性がついた場合は実行コンテキストが"Module"として実行されます。この `script` 要素の直下に書いた `this` は `undefined` となります。

```
<script type="module">
// 実行コンテキストは"Module"
console.log(this); // => undefined
</script>
```

このように、コード直下の `this` は実行コンテキストによって `undefined` となる場合があります。単純にグローバルオブジェクトを参照したい場合は、`this` ではなく `window` などのグローバルオブジェクトを直接参照した方がよいです。

## 関数とメソッドにおける `this`

関数を定義する方法として、`function` キーワードによる関数宣言と関数式、Arrow Functionなどがあります。`this` が参照先を決めるルールはArrow Functionとそれ以外の方法で異なります。

そのため、まずは関数定義の種類について振り返ってから、それぞれの `this` について見ていきましょう。

### 関数の種類

[関数と宣言](#)で詳しくは紹介していますが、関数の定義方法と呼び出し方について改めて振り返ってみましょう。関数を定義する場合には、次の3つの方法を利用します。

```
// `function` キーワードから始める関数宣言
function fn1() {}
// `function` を式として扱う関数式
const fn2 = function() {};
// Arrow Functionを使った関数式
const fn3 = () => {};
```

それぞれ定義した関数は `関数名()` と書くことで呼び出すことができます。

```
// 関数宣言
function fn() {}
// 関数呼び出し
fn();
```

### メソッドの種類

JavaScriptではオブジェクトのプロパティが関数である場合にそれをメソッドと呼びます。一般的にはメソッドも含めたものを関数といい、関数宣言などとプロパティである関数を区別する場合にメソッドと呼びます。

メソッドを定義する場合には、オブジェクトのプロパティに関数式を定義するだけです。

```
const object = {
  // `function` キーワードを使ったメソッド
  method1: function() {
  },
  // Arrow Functionを使ったメソッド
  method2: () => {
  }
};
```

これに加えてメソッドには短縮記法があります。オブジェクトリテラルの中で `メソッド名(){ /*メソッドの処理*/ }` と書くことで、メソッドを定義できます。

```
const object = {
  // メソッドの短縮記法で定義したメソッド
  method() {
  }
};
```

これらのメソッドは、`オブジェクト名.メソッド名()` と書くことで呼び出すことができます。

```
const object = {
  // メソッドの定義
  method() {
  }
};
// メソッド呼び出し
object.method();
```

関数定義とメソッドの定義についてまとめると、次のような種類があります。

名前	関数	メソッド
関数宣言( <code>function fn(){} </code> )		x
関数式( <code>const fn = function(){} </code> )		
Arrow Function( <code>const fn = () =&gt; {} </code> )		
メソッドの短縮記法( <code>const obj = { method(){} } </code> )	x	

そして、最初に書いたように `this` の挙動は、Arrow Functionの関数定義とそれ以外（`function` キーワードやメソッドの短縮記法）の関数定義で異なります。そのため、まずはArrow Function以外の関数やメソッドにおける `this` を見ていきます。

## Arrow Function以外の関数における `this`

Arrow Function以外の関数（メソッドも含む）における `this` は実行時に決まる値となります。言い方をかえると `this` は関数に渡される暗黙的な引数のようなもので、その渡される値は関数を実行する時に決まります。

次のコードは擬似的なものです。関数の中に書かれた `this` は、関数の呼び出し元から暗黙的に渡される値を参照することになります。このルールはArrow Function以外の関数やメソッドで共通した仕組みとなります。Arrow Functionで定義した関数やメソッドはこのルールとは別の仕組みとなります。

```
// 擬似的な`this`の値の仕組み
// 関数は引数として暗黙的に`this`の値を受け取るイメージ
function fn(暗黙的渡されるthisの値, 引数) {
  console.log(this); // => 暗黙的渡されるthisの値
}
// 暗黙的に`this`の値を引数として渡しているイメージ
fn(暗黙的に渡すthisの値, 引数);
```

関数における `this` の基本的な参照先（暗黙的に関数に渡す `this` の値）はベースオブジェクトとなります。ベースオブジェクトとは「メソッドを呼ぶ際に、そのメソッドのドット演算子または括弧演算子のひとつ左にあるオブジェクト」のことを言います。ベースオブジェクトがない場合の `this` は `undefined` となります。

たとえば、`fn()` のように関数を呼び出したとき、この `fn` 関数呼び出しのベースオブジェクトはないため、`this` は `undefined` となります。一方、`obj.method()` のようにメソッドを呼び出したとき、この `obj.method` メソッド呼び出しのベースオブジェクトは `obj` オブジェクトとなり、`this` は `obj` となります。

```
// `fn`関数はメソッドではないのでベースオブジェクトはない
fn();
// `obj.method`メソッドのベースオブジェクトは`obj`、
obj.method();
// `obj1.obj2.method`メソッドのベースオブジェクトは`obj2`、
// ドット演算子、括弧演算子どちらも結果は同じ
obj1.obj2.method();
obj1["obj2"]["method"]();
```

`this` は関数の定義ではなく呼び出し方で参照する値が異なります。これは、後述する「`this` が問題となるパターン」で詳しく紹介します。Arrow Function以外の関数では、関数の定義だけを見て `this` の値が何かということは決定できない点には注意が必要です。

## 関数宣言や関数式における `this`

まずは、関数宣言や関数式の場合を見ていきます。

次の例では、関数宣言と関数式で定義した関数の中の `this` をコンソールに出力しています。このとき、`fn1` と `fn2` はただの関数として呼び出されています。つまり、ベースオブジェクトがないため `this` は `undefined` となります。

```
"use strict";
function fn1() {
    return this;
}
const fn2 = function() {
    return this;
};
// 関数の中の`this`が参照する値は呼び出し方によって決まる
// `fn1`と`fn2`どちらもただの関数として呼び出している
// メソッドとして呼び出していないためベースオブジェクトはない
// ベースオブジェクトがない場合、`this`は`undefined`となる
console.log(fn1()); // => undefined
console.log(fn2()); // => undefined
```

これは、関数の中に関数を定義して呼び出す場合も同じです。

```
"use strict";
function outer() {
    console.log(this); // => undefined
    function inner() {
        console.log(this); // => undefined
    }
    // `inner`関数呼び出しのベースオブジェクトはない
    inner();
}
// `outer`関数呼び出しのベースオブジェクトはない
outer();
```

この書籍では注釈がないコードはstrict modeとして扱いますが、コード例に `"use strict";` とあらためてstrict modeの明示しています。なぜなら、strict modeではない場合 `this` は `undefined` ではなくグローバルオブジェクトとなってしまう問題があるためです（「[JavaScriptとは](#)を参照」）。これは、strict modeではない通常の関数呼び出しのみの問題であり、メソッドではこの暗黙的な型変換は行われません。これも、`this` をメソッド以外で使うべきではない理由の1つとなります。

## メソッド呼び出しにおける `this`

次に、メソッドの場合を見ていきます。メソッドの場合は、そのメソッドは何かしらのオブジェクトに所属しています。なぜなら、JavaScriptではオブジェクトのプロパティとして指定される関数のことをメソッドと呼ぶためです。

次の例では `method1` と `method2` はそれぞれメソッドとして呼び出されています。このとき、それぞれのベースオブジェクトは `object` となり、`this` は `object` となります。

```
const object = {
    // 関数式をプロパティの値にしたメソッド
    method1: function() {
        return this;
    },
}
```

```
// 短縮記法で定義したメソッド
method2() {
    return this;
}
};

// メソッド呼び出しの場合、それぞれの`this`はベースオブジェクト(`object`)を参照する
// メソッド呼び出しの`.`の左にあるオブジェクトがベースオブジェクト
object.method1(); // => object
object.method2(); // => object
```

これを利用すれば、メソッドの中から同じオブジェクトに所属する別のプロパティを `this` で参照できます。

```
const person = {
    fullName: "Brendan Eich",
    sayName: function() {
        // `person.fullName`と書いているのと同じ
        return this.fullName;
    }
};
// `person.fullName`を出力する
console.log(person.sayName()); // => "Brendan Eich"
```

このようにメソッドが所属するオブジェクトのプロパティを、`オブジェクト名.プロパティ名` の代わりに `this.プロパティ名` で参照できます。

オブジェクトは何重にもネストできますが、`this` はベースオブジェクトを参照するというルールは同じです。

次のコードを見てみると、ネストしたオブジェクトにおいてメソッド内の `this` がベースオブジェクトである `obj3` を参照していることが分かります。このときのベースオブジェクトはドットで繋いだ一番左の `obj1` ではなく、メソッドから見てひとつ左の `obj3` となります。

```
const obj1 = {
    obj2: {
        obj3: {
            method() {
                return this;
            }
        }
    }
};
// `obj1.obj2.obj3.method`メソッドの`this`は`obj3`を参照
console.log(obj1.obj2.obj3.method() === obj1.obj2.obj3); // => true
```

## this が問題となるパターン

`this` はその関数（メソッドも含む）呼び出しのベースオブジェクトを参照することがわかりました。`this` は所属するオブジェクトを直接書く代わりとして利用できますが、一方 `this` には色々な問題があります。

この問題の原因は `this` がどの値を参照するかは関数の呼び出し時に決まるという性質に由来します。この `this` の性質が問題となるパターンの代表的な2つの例とそれぞれの対策について見ていきます。

### 問題: `this` を含むメソッドを変数に代入した場合

JavaScriptではメソッドとして定義したものが、後からただの関数として呼び出されることがあります。なぜなら、メソッドは関数を値にもつプロパティのことで、プロパティは変数に代入し直すことができるためです。

そのため、メソッドとして定義した関数も、別の変数に代入してただの関数として呼び出されることがあります。この場合には、メソッドとして定義した関数であっても、実行時にはただの関数であるためベースオブジェクトが変わっています。これは `this` が定義した時点ではなく実行した時に決まるという性質そのものです。

具体的に、`this` が実行時に変わる例を見てましょう。次の例では、`person.sayName` メソッドを変数 `say` に代入してから実行しています。このときの `say` 関数( `sayName` メソッドを参照)のベースオブジェクトはありません。そのため、`this` は `undefined` となり、`undefined.fullName` は参照できずに例外をなげます。

```
"use strict";
const person = {
  fullName: "Brendan Eich",
  sayName: function() {
    // `this`は呼び出し元によってことなる
    return this.fullName;
  }
};
// `sayName`メソッドは`person`オブジェクトに所属する
// `this`は`person`オブジェクトとなる
console.log(person.sayName()); // => "Brendan Eich"
// `person.sayName`を`say`変数に代入する
const say = person.sayName;
// 代入したメソッドを関数として呼ぶ
// この`say`関数はどのオブジェクトにも所属していない
// `this`は`undefined`となるため例外を投げる
say(); // => TypeError: Cannot read property 'fullName' of undefined
```

結果的には、次のようなコードが実行されているのと同じです。次のコードでは、`undefined.fullName` を参照しようとすると例外が発生しています。

```
"use strict";
// const sayName = person.sayName; は次のようなイメージ
const say = function() {
  return this.fullName;
};
// `this`は`undefined`となるため例外をなげる
say(); // => TypeError: Cannot read property 'fullName' of undefined
```

このように、Arrow Function以外の関数において、`this` は定義した時ではなく実行した時に決定されます。そのため、関数に `this` を含んでいる場合、その関数は意図した呼ばれ方がされないと間違った結果が発生するという問題があります。

この問題の対処法としては大きく分けて2つあります。

ひとつはメソッドとして定義されている関数はメソッドとして呼ぶということです。メソッドをわざわざただの関数として呼ばなければそもそもこの問題は発生しません。

もうひとつは、`this` の値を指定して関数を呼べるメソッドで関数を実行する方法です。

## 対処法: call、apply、bindメソッド

関数やメソッドの `this` を明示的に指定して関数を実行する方法もあります。`Function` (関数オブジェクト) には `call`、`apply`、`bind` といった明示的に `this` を指定して関数を実行するメソッドが用意されています。

`call` メソッドは第一引数に `this` としたい値を指定し、残りの引数は呼び出す関数の引数となります。暗黙的に渡される `this` の値を明示的に渡せるメソッドといえます。

```
関数.call(thisの値, ...関数の引数);
```

次の例では `this` に `person` オブジェクトを指定した状態で `say` 関数を呼び出しています。`call` メソッドの第二引数で指定した値が、`say` 関数の仮引数 `message` に入ります。

```
"use strict";
function say(message) {
    return `${message} ${this.fullName}!`;
}
const person = {
    fullName: "Brendan Eich"
};
// `this`を`person`にして`say`関数を呼びだす
console.log(say.call(person, "こんにちは")); // => "こんにちは Brendan Eich!"
// `say`関数をそのまま呼び出すと`this`は`undefined`となるため例外が発生
say("こんにちは"); // => TypeError: Cannot read property 'fullName' of undefined
```

`apply` メソッドは第一引数に `this` とする値を指定し、第二引数に関数の引数を配列として渡します。

```
関数.apply(thisの値, [関数の引数1, 関数の引数2]);
```

次の例では `this` に `person` オブジェクトを指定した状態で `say` 関数を呼び出しています。`apply` メソッドの第二引数で指定した配列は、自動的に展開されて `say` 関数の仮引数 `message` に入ります。

```
"use strict";
function say(message) {
    return `${message} ${this.fullName}!`;
}
const person = {
    fullName: "Brendan Eich"
};
// `this`を`person`にして`say`関数を呼びだす
// callとは異なり引数を配列として渡す
console.log(say.apply(person, ["こんにちは"])); // => "こんにちは Brendan Eich!"
// `say`関数をそのまま呼び出すと`this`は`undefined`となるため例外が発生
say("こんにちは"); // => TypeError: Cannot read property 'fullName' of undefined
```

`call` メソッドと `apply` メソッドの違いは、関数の引数への値の渡し方が異なるだけです。また、どちらのメソッドも `this` の値が不要な場合は `null` を渡すのが一般的です。

```
function add(x, y) {
    return x + y;
}
// `this`は不要な場合はnullを渡す
add.call(null, 1, 2); // => 3
add.apply(null, [1, 2]); // => 3
```

最後に `bind` メソッドについてです。名前のとおり `this` の値を束縛 (bind) した新しい関数を作成します。

```
関数.bind(thisの値, ...関数の引数); // => thisや引数がbindされた関数
```

次の例では `this` を `person` オブジェクトに束縛した `say` 関数の関数を作っています。`bind` メソッドの第二引数以降に値を渡すことで、束縛した関数の引数も束縛できます。

```
function say(message) {
    return `${message} ${this.fullName}!`;
}
const person = {
    fullName: "Brendan Eich"
};
// `this`を`person`に束縛した`say`関数をラップした関数を作る
```

```
const sayPerson = say.bind(person, "こんにちは");
sayPerson(); // => "こんにちは Brendan Eich!"
```

この `bind` メソッドをただの関数で表現すると次のように書けます。`bind` は `this` や引数を束縛した関数を作るメソッドということがわかります。

```
function say(message) {
  return `${message} ${this.fullName}!`;
}
const person = {
  fullName: "Brendan Eich"
};
// `this`を`person`に束縛した`say`関数をラップした関数を作る
// say.bind(person, "こんにちは"); は次のようなラップ関数を作る
const sayPerson = () => {
  return say.call(person, "こんにちは");
};
sayPerson(); // => "こんにちは Brendan Eich!"
```

このように `call`、`apply`、`bind` メソッドを使うことで `this` を明示的に指定した状態で関数を呼び出せます。しかし、毎回関数を呼び出すたびにこれらのメソッドを使うのは、関数を呼び出すための関数が必要になってしまい手間がかかります。そのため、基本的には「メソッドとして定義されている関数はメソッドとして呼ぶこと」でこの問題を回避するほうがよいでしょう。その中で、どうしても `this` を固定したい場合には `call`、`apply`、`bind` メソッドを利用します。

## 問題: コールバック関数と `this`

コールバック関数の中で `this` を参照すると問題となる場合があります。この問題は、メソッドの中で `Array#map` メソッドなどコールバック関数を扱う場合に発生しやすいです。

具体的に、コールバック関数における `this` が問題となっている例を見てみましょう。次のコードでは `prefixArray` メソッドの中で `Array#map` メソッドを使っています。このとき、`Array#map` メソッドのコールバック関数の中で、`Prefixer` オブジェクトを参照するつもりで `this` を参照しています。

しかし、このコールバック関数における `this` は `undefined` となり、`this.prefix` は `undefined.prefix` であるため `TypeError`となります。

```
"use strict";
const Prefixer = {
  prefix: "pre",
  /**
   * `strings`配列の各要素にprefixをつける
   */
  prefixArray(strings) {
    return strings.map(function(string) {
      // コールバック関数における`this`は`undefined`となる(strict mode)
      // そのため`this.prefix`は`undefined.prefix`となり例外が発生する
      return this.prefix + "-" + string;
    });
  }
};
// `prefixArray`メソッドにおける`this`は`Prefixer`
Prefixer.prefixArray(["a", "b", "c"]); // => TypeError: Cannot read property 'prefix' of undefined
```

なぜコールバック関数の中での `this` が `undefined` となるのかを見ていきます。`Array#map` メソッドにはコールバック関数として、その場で定義した匿名関数を渡していることに注目してください。

```
// ...
prefixArray(strings) {
```

```
// 匿名関数をコールバック関数として渡している
return strings.map(function(string) {
    return this.prefix + "-" + string;
});
// ...
```

このとき、`Array#map` メソッドに渡しているコールバック関数は `callback()` のようにただの関数として呼び出されます。つまり、コールバック関数として呼び出すとき、この関数にはベースオブジェクトはありません。そのため `callback` 関数の `this` は `undefined` となります。

先ほどの匿名関数をコールバック関数として渡しているのは、一度 `callback` 変数に入れてから渡しても結果は同じです。

```
"use strict";
const Prefixer = {
    prefix: "pre",
    prefixArray(strings) {
        // コールバック関数は`callback()`のように呼び出される
        // そのためコールバック関数における`this`は`undefined`となる(strict mode)
        const callback = function(string) {
            return this.prefix + "-" + string;
        };
        return strings.map(callback);
    }
};
// `prefixArray`メソッドにおける`this`は`Prefixer`
Prefixer.prefixArray(["a", "b", "c"]); // => TypeError: Cannot read property 'prefix' of undefined
```

## 対処法: `this` を一時変数へ代入する

コールバック関数内での `this` の参照先が変わる問題への対処法として、`this` を別の変数に代入し、その `this` の参照先を保持するという方法があります。

`this` は関数の呼び出し元で変化し、その参照先は呼び出し元におけるベースオブジェクトです。`prefixArray` メソッドの呼び出しにおいては、`this` は `Prefixer` オブジェクトです。しかし、コールバック関数はあらためて関数として呼び出されるため `this` が `undefined` となってしまうのが問題でした。

そのため、最初の `prefixArray` メソッド呼び出しにおける `this` の参照先を一時変数として保存することでこの問題を回避できます。つぎのように、`prefixArray` メソッドの `this` を `that` 変数に保持しています。コールバック関数からは `this` の代わりに `that` 変数を参照することで、コールバック関数からも `prefixArray` メソッド呼び出しと同じ `this` を参照できます。

```
"use strict";
const Prefixer = {
    prefix: "pre",
    prefixArray(strings) {
        // `that`は`prefixArray`メソッド呼び出しにおける`this`となる
        // つまり`that`は`Prefixer`オブジェクトを参照する
        const that = this;
        return strings.map(function(string) {
            // `this`ではなく`that`を参照する
            return that.prefix + "-" + string;
        });
    }
};
// `prefixArray`メソッドにおける`this`は`Prefixer`
const prefixedStrings = Prefixer.prefixArray(["a", "b", "c"]);
console.log(prefixedStrings); // => ["pre-a", "pre-b", "pre-c"]
```

もちろん `Function#call` メソッドなどで明示的に `this` を渡して関数を呼び出すこともできます。また、`Array#map` メソッドなどは `this` となる値を引数として渡せる仕組みを持っています。そのため、つぎのように第二引数に `this` となる値を渡すことでも解決できます。

```
"use strict";
const Prefixer = {
  prefix: "pre",
  prefixArray(strings) {
    // `Array#map`メソッドは第二引数に`this`となる値を渡せる
    return strings.map(function(string) {
      // `this`が第二引数の値と同じになる
      // つまり`prefixArray`メソッドと同じ`this`となる
      return this.prefix + "-" + string;
    }, this);
  }
};
// `prefixArray`メソッドにおける`this`は`Prefixer`
const prefixedStrings = Prefixer.prefixArray(["a", "b", "c"]);
console.log(prefixedStrings); // => ["pre-a", "pre-b", "pre-c"]
```

しかし、これら解決方法はコールバック関数において `this` が変わることを意識して書く必要があります。そもそもの問題としてメソッド呼び出しとその中のコールバック関数における `this` が変わってしまうのが問題でした。ES2015では `this` を変えずにコールバック関数を定義する方法として、Arrow Functionが導入されました。

## 対処法: Arrow Functionでコールバック関数を扱う

通常の関数やメソッドは呼び出し時に暗黙的に `this` の値を受け取り、関数内の `this` はその値を参照します。一方、Arrow Functionはこの暗黙的な `this` の値を受け取ません。そのためArrow Function内の `this` は、スコープチェーンの仕組みと同様で外側の関数(この場合は `prefixArray` メソッド)に探索します。これにより、Arrow Functionで定義したコールバック関数は呼び出し方には関係なく、常に外側の関数の `this` をそのまま利用します。

Arrow Functionを使うことで、先ほどのコードは次のように書くことができます。

```
"use strict";
const Prefixer = {
  prefix: "pre",
  prefixArray(strings) {
    return strings.map((string) => {
      // Arrow Function自体は`this`を持たない
      // `this`は外側の`prefixArray`関数がもつ`this`を参照する
      // そのため`this.prefix`は"pre"となる
      return this.prefix + "-" + string;
    });
  }
};
// この時、`prefixArray`のベースオブジェクトは`Prefixer`となる
// つまり、`prefixArray`メソッド内の`this`は`Prefixer`を参照する
const prefixedStrings = Prefixer.prefixArray(["a", "b", "c"]);
console.log(prefixedStrings); // => ["pre-a", "pre-b", "pre-c"]
```

このように、Arrow Functionでのコールバック関数における `this` は簡潔です。そのため、コールバック関数内の `this` の対処法として `this` を代入する方法を紹介しましたが、ES2015からはArrow Functionを使うのがもっとも簡潔です。

このArrow Functionと `this` の関係についてもっと詳しく見ていきます。

## Arrow Functionと `this`

Arrow Functionで定義された関数やメソッドにおける `this` がどの値を参照するかは関数の定義時（静的）に決まります。一方、Arrow Functionではない関数においては、`this` は呼び出し元に依存するため関数の実行時（動的）に決まります。

Arrow Functionとそれ以外の関数で大きく違うことは、Arrow Functionは `this` を暗黙的な引数として受け付けないということです。そのため、Arrow Function内には `this` が定義されていません。このときの `this` は外側のスコープ（関数）の `this` を参照します。

これは、変数におけるスコープチェーンの仕組みと同様で、そのスコープに `this` が定義されていない場合には外側のスコープを探索するのと同じです。そのため、Arrow Function内の `this` の参照先は、常に外側のスコープ（関数）へと `this` の定義を探索しに行きます（詳細は[スコープチェーン](#)を参照）。また、`this` は読み取り専用のキーワードであるため、ユーザーが `this` という変数を定義できません。

```
const this = "thisは読み取り専用"; // => SyntaxError: Unexpected token this
```

これにより、通常の変数のように `this` がどの値を参照するかは静的（定義時）に決定できます（詳細は[静的スコープ](#)を参照）。つまり、Arrow Functionにおける `this` は「Arrow Function自身の外側のスコープに定義されたもっとも近い関数の `this` の値」となります。

具体的な例を元にArrow Functionにおける `this` の動きを見ていきましょう。

まずは、関数式のArrow Functionを見ていきましょう。

次の例では、関数式で定義したArrow Functionの中の `this` をコンソールに出力しています。このとき、`fn` の外側には関数はないため、「自身より外側のスコープ定義されたもっとも近い関数」の条件にあてはまるものはありません。このときの `this` はトップレベルに書かれた `this` と同じ値になります。

```
// Arrow Functionで定義した関数
const fn = () => {
    // この関数の外側には関数は存在しない
    // トップレベルの`this`と同じ値
    return this;
};
fn() === this; // => true
```

トップレベルに書かれた `this` の値は[実行コンテキスト](#)によって異なることを紹介しました。`this` の値は、実行コンテキストが"Script"ならばグローバルオブジェクトとなり、"Module"ならば `undefined` となります。

次の例のように、Arrow Functionを包むように通常の関数が定義されている場合はどうでしょうか。Arrow Functionにおける `this` は「自身の外側のスコープにあるもっとも近い関数の `this` の値」となるのは同じです。

```
"use strict";
function outer() {
    // Arrow Functionで定義した関数を返す
    return () => {
        // この関数の外側には`outer`関数が存在する
        // `outer`関数に`this`を書いた場合と同じ
        return this;
    };
}
// `outer`関数の返り値はArrow Functionにて定義された関数
const innerArrowFunction = outer();
console.log(innerArrowFunction()); // => undefined;
```

つまり、このArrow Functionにおける `this` は `outer` 関数で `this` を参照した場合と同じ値になります。

```
"use strict";
function outer() {
    // `outer`関数直下の`this`
```

```

const that = this;
// Arrow Functionで定義した関数を返す
return () => {
  // Arrow Function自身は`this`を持たない
  // `outer`関数に`this`を書いた場合と同じ
  return that;
};
}
// `outer()`と呼び出した時の`this`は`undefined`(strict mode)
const innerArrowFunction = outer();
console.log(innerArrowFunction()); // => undefined;

```

## メソッドとコールバック関数とArrow Function

メソッド内におけるコールバック関数はArrow Functionをもっと活用できるパターンです。 `function` キーワードでコールバック関数を定義すると、`this` の値はコールバック関数の呼びられ方を意識する必要があります。なぜなら、`function` キーワードで定義した関数における `this` は呼び出し方によって変わるためにです。

コールバック関数側から見ると、どのように呼ばれるかによって変わる `this` を使うことはエラーとなる場合もあるため使えません。そのため、コールバック関数の外側のスコープで `this` を一時変数に代入し、それを使うという回避を取っていました。

```

// `callback`関数を受け取り呼び出す関数
const callCallback = (callback) => {
  // `callback`を呼び出す実装
};

const object = {
  method() {
    callCallback(function() {
      // ここでの`this`は`callCallback`の実装に依存する
      // `callback()`のように単純に呼び出されるなら`this`は`undefined`になる
      // `Function#call`などを使い特定のオブジェクトを指定するかもしれない
      // この問題を回避するために`const that = this`のような一時変数を使う
    });
  }
};

```

一方、Arrow Functionでコールバック関数を定義した場合は、1つ外側の関数の `this` を参照します。このときの Arrow Functionで定義したコールバック関数における `this` は呼び出し方によって変化しません。そのため、`this` を一時変数に代入するなどの回避方法は必要ありません。

```

// `callback`関数を受け取り呼び出す関数
const callCallback = (callback) => {
  // `callback`を呼び出す実装
};

const object = {
  method() {
    callCallback(() => {
      // ここでの`this`は1つ外側の関数における`this`と同じ
    });
  }
};

```

このArrow Functionにおける `this` は呼び出し方の影響を受けません。つまり、コールバック関数がどのように呼ばれるかという実装について考えることなく `this` を扱うことができます。

```

const Prefixer = {
  prefix: "pre",
  prefixArray(strings) {

```

```

        return strings.map((string) => {
            // `Prefixer.prefixArray()` と呼び出されたとき
            // `this` は常に `Prefixer` を参照する
            return this.prefix + "-" + string;
        });
    }
};

const prefixedStrings = Prefixer.prefixArray(["a", "b", "c"]);
console.log(prefixedStrings); // => ["pre-a", "pre-b", "pre-c"]

```

## Arrow Functionは `this` をbindできない

Arrow Functionで定義した関数には `call`、`apply`、`bind` を使った `this` の指定は単に無視されます。これは、Arrow Functionは `this` をもつことができないためです。

次のように Arrow Functionで定義した関数に対して `call` で `this` をしても、`this` の参照先が代わっていないことが分かります。同様に `apply` や `bind` メソッドを使った場合も `this` の参照先が変わりません。

```

const fn = () => {
    return this;
};

// Scriptコンテキストの場合、スクリプト直下のArrow Functionの`this`はグローバルオブジェクト
console.log(fn()); // グローバルオブジェクト
// callで`this`を`{}`にしようとしても、`this`は変わらない
fn.call({}); // グローバルオブジェクト

```

最初に述べたように `function` キーワードで定義した関数は呼び出し時に、ベースオブジェクトが暗黙的な引数のように `this` の値として渡されます。一方、Arrow Functionの関数は呼び出し時に `this` を受け取らずに、定義時の Arrow Functionにおける `this` の参照先が静的に決定されます。

また、`this` が変わらないのはあくまで Arrow Functionで定義した関数だけで、Arrow Functionの `this` が参照する「自身の外側のスコープにあるもっとも近い関数の `this` の値」は `call` メソッドで変更できます。

```

const object = {
    method() {
        const arrowFunction = () => {
            return this;
        };
        return arrowFunction();
    }
};
// 通常の`this`は`object.method`の`this`と同じ
console.log(object.method()); // => object
// `object.method`の`this`を変更すれば、Arrow Functionの`this`も変更される
console.log(object.method.call("THAT")); // => "THAT"

```

## まとめ

`this` は状況によって異なる値を参照する性質を持ったキーワードであることを紹介しました。その `this` の評価結果をまとめると次の表のようになります。

実行コンテキスト	strict mode	コード	<code>this</code> の評価結果
Script	NO	<code>this</code>	global
Script	NO	<code>const fn = () =&gt; this</code>	global
Script	NO	<code>const fn = function(){ return this; }</code>	global

Script	YES	this	global
Script	YES	const fn = () => this	global
Script	YES	const fn = function(){ return this; }	undefined
Module	YES	this	undefined
Module	YES	const fn = () => this	undefined
Module	YES	const fn = function(){ return this; }	undefined
*	*	const obj = { method(){ return this; } }	obj
*	*	const obj = { method: function(){ return this; } }	obj

\*はどの場合でも this の評価結果に影響しないということを示しています

実際にブラウザで実行した結果は[What is this value in JavaScript?][1]というサイトで確認できます。

this はオブジェクト指向プログラミングの文脈でJavaScriptに導入されました。awbjs メソッド以外においても this は評価できますが、実行コンテキストやstrict modeなどによって結果が異なり混乱の元となります。そのため、メソッドではない通常の関数においては this を使うべきではありません。

また、メソッドにおいても this は呼び出し方によって異なる値となり、それにより発生する問題と対処法についてを紹介しました。コールバック関数における this はArrow Functionを使うことで分かりやすく解決できます。この背景にはArrow Functionで定義した関数は this を持たないという性質があります。

awbjs. ES 2015の仕様編集者であるAllen Wirfs-Brock氏もただの関数においては this を使うべきではないと述べている。 <https://twitter.com/awbjs/status/938272440085446657>; ↵

# クラス

「クラス」と一言にいってもさまざまであるため、ここでは構造、動作、状態を定義できるものを示すことにします。また、この章では概念を示す場合はクラスと呼び、クラスに関する構文（記述するコード）のことを `class` 構文と呼びます。

クラスとは動作や状態の初期値を定義した構造をことを言います。クラスからはインスタンスと呼ばれるオブジェクトを作成でき、インスタンスはクラスに定義した動作を継承し、状態は動作によって変化します。とても抽象的なことに見えますが、これは今までオブジェクトや関数を使って表現してきたものにも見えます。実際にJavaScriptではES2015より前までは `class` 構文はなく、関数を使いクラスのようなものを表現して扱っていました。

ES2015で `class` 構文が導入されました。この `class` 構文で定義したクラスは一種の関数オブジェクトです。

`class` 構文ではプロトタイプベースの継承の仕組みの上に関数でクラスを表現しています。そのため、`class` 構文はクラスを作るための関数定義や継承をパターン化した書き方といえます。[糖衣構文](#)

JavaScriptでは関数で学んだことの多くはクラスでもそのまま適応されます。また、関数の定義方法として関数宣言文と関数式があるように、クラスにもクラス宣言文とクラス式があります。この章では、`class` 構文でのクラスの定義や継承、クラスの性質について学んでいきます。

## クラスの定義

クラスを定義するには `class` 構文を使いますが、クラスの定義方法にはクラス宣言文とクラス式があります。

まずは、クラス宣言文によるクラスの定義方法を見ていきます。

クラス宣言文では `class` キーワードを使い、`class クラス名 { }` のようにクラスの構造を定義できます。クラスは必ずコンストラクタを持ち、`constructor` という名前のメソッドとして定義します。コンストラクタとは、そのクラスからインスタンスを作成する際にインスタンスに関する状態の初期化を行うメソッドです。

```
class MyClass {
  constructor() {
    // コンストラクタ関数の処理
  }
}
```

もうひとつの定義方法であるクラス式は、クラスを値として定義する方法です。クラス式ではクラス名を省略できます。これは関数式における匿名関数と同じです。

```
const MyClass = class MyClass {
  constructor() {}
};

const AnonymousClass = class {
  constructor() {}
};
```

コンストラクタ関数内で何も処理がない場合はコンストラクタの記述を省略できます。省略した場合には自動的に空のコンストラクタが定義されるため、クラスにはコンストラクタが必ず存在します。

```
class MyClassA {
  constructor() {
    // コンストラクタの処理が必要なら書く
  }
}
```

```
// コンストラクタの処理が不要な場合は省略できる
class MyClassB {
}
```

## クラスのインスタンス化

クラスは `new` 演算子でインスタンスであるオブジェクトを作成できます。 `class` 構文で定義したクラスからインスタンスを作成することをインスタンス化と呼びます。あるインスタンスが指定したクラスから作成されたものかを判定するには `instanceof` 演算子が利用できます。

```
class MyClass {
}
// `MyClass`をインスタンス化する
const myClass = new MyClass();
// 毎回新しいインスタンス(オブジェクト)を作成する
const myClassAnother = new MyClass();
// それぞれのインスタンスは異なるオブジェクト
console.log(myClass === myClassAnother); // => false
// クラスのインスタンスかどうかは`instanceof`演算子で判定できる
console.log(myClass instanceof MyClass); // => true
console.log(myClassAnother instanceof MyClass); // => true
```

このままでは何もできない空のクラスなので、値を持ったクラスを定義してみましょう。

クラスではインスタンスの初期化処理をコンストラクタ関数で行います。コンストラクタ関数は `new` 演算子でインスタンス化されるときに暗黙的によばれ、コンストラクタのなかでの `this` はこれから新しく作るインスタンスオブジェクトとなります。

次のコードでは `x` 座標と `y` 座標の値をもつ `Point` というクラスを定義しています。コンストラクタ関数 (`constructor`)の中でインスタンスオブジェクト (`this`) の `x` と `y` プロパティに値を代入して初期化しています。

```
class Point {
    // コンストラクタ関数の仮引数として`x`と`y`を定義
    constructor(x, y) {
        // コンストラクタ関数における`this`はインスタンスを示すオブジェクト
        // インスタンスの`x`と`y`プロパティにそれぞれ値を設定する
        this.x = x;
        this.y = y;
    }
}
```

この `Point` クラスのインスタンスを作成するには `new` 演算子を使います。`new` 演算子には関数呼び出しと同じように引数を渡すことができます。`new` 演算子の引数はクラスの `constructor` メソッド (コンストラクタ関数) の仮引数に渡されます。そして、コンストラクタのなかではインスタンスオブジェクト (`this`) の初期化処理を行います。

```
class Point {
    // 2. コンストラクタ関数の仮引数として`x`には`3`、`y`には`4`が渡る
    constructor(x, y) {
        // 3. インスタンス(`this`)の`x`と`y`プロパティにそれぞれ値を設定する
        this.x = x;
        this.y = y;
        // コンストラクタではreturn文は書かない
    }
}

// 1. コンストラクタを`new`演算子で引数とともに呼び出す
const point = new Point(3, 4);
```

```
// 4. `Point`のインスタンスである`point`の`x`と`y`プロパティには初期化された値が入る
console.log(point.x); // => 3
console.log(point.y); // => 4
```

このようにクラスからインスタンスを作成するには必ず `new` 演算子を使います。

一方、クラスは通常の関数として呼ぶことができません。これは、クラスのコンストラクタはインスタンス (`this`) の初期化する場所であり、通常の関数とは役割が異なるためです。

```
class MyClass {
    constructor() { }
}
// クラスのコンストラクタ関数として呼び出すことはできない
MyClass(); // => TypeError: class constructors must be invoked with |new|
```

コンストラクタは初期化処理を書く場所であるため、`return` 文で値を返すべきではありません。JavaScriptでは、コンストラクタで任意のオブジェクトを返すことが可能ですが行うべきではありません。なぜなら、コンストラクタは `new` 演算子で呼び出し、その評価結果はクラスのインスタンスを期待するのが一般的であるためです。

```
// 非推奨の例: コンストラクタで値を返すべきではない
class Point {
    constructor(x, y) {
        // `this`の代わりにただのオブジェクトを返せる
        return { x, y };
    }
}

// `new`演算子の結果はコンストラクタ関数が返したただのオブジェクト
const point = new Point(3, 4);
console.log(point); // => { x: 3, y: 4 }
// Pointクラスのインスタンスではない
console.log(point instanceof Point); // => false
```

### [Note] クラス名は大文字で始める

JavaScriptでは慣習としてクラス名は大文字で始まる名前を付けます。これは、変数名にキャメルケースを使う慣習があるのと同じで、名前自体には特別なルールがあるわけではありません。クラス名を大文字にしておき、そのインスタンスは小文字で開始すれば、名前が被らないため合理的な理由で好まれています。

```
class Thing {}
const thing = new Thing();
```

### [コラム] `class` 構文と関数でのクラスの違い

ES2015より前はこれらのクラスを `class` 構文ではなく、関数で表現していました。その表現方法は人によってさまざままで、これも `class` 構文という統一した表現が導入された理由の1つです。

次のコードでは先ほどの `class` 構文でのクラスを簡略化した関数での1つの実装例です。この関数でのクラス表現は、継承の仕組みなどは省かれていますが、`class` 構文とよく似ています。

```
// コンストラクタ関数
const Point = function PointConstructor(x, y) {
    // インスタンスの初期化処理
    this.x = x;
    this.y = y;
};

// `new`演算子でコンストラクタ関数から新しいインスタンスを作成
```

```
const point = new Point(3, 4);
```

大きな違いとして、`class` 構文で定義したクラスは関数として呼び出すことができません。クラスは `new` 演算子でインスタンス化して使うものなので、これはクラスの誤用を防ぐ仕様です。一方、関数でのクラス表現はただの関数なので、当然関数として呼び出せます。

```
// `class`構文でのクラス
class MyClass {
}
// 関数でのクラス表現
function MyClassLike() {
}
// 関数なので関数として呼び出せる
MyClassLike();
// クラスは関数として呼び出すと例外が発生する
MyClass(); // => TypeError: class constructors must be invoked with |new|
```

このように、`class` 構文で定義したクラスは一種の関数ですが、そのクラスはクラス以外には利用できないようになっています。

## クラスのプロトタイプメソッドの定義

クラスの動作はメソッドによって定義できます。`constructor` メソッドは初期化時に呼ばれる特殊なメソッドですが、`class` 構文ではクラスに対して自由にメソッドを定義できます。このクラスに定義したメソッドは各インスタンスがもつ動作となります。

次のように `class` 構文ではクラスに対してメソッドを定義できます。メソッドの中からクラスのインスタンスを参照するには、`constructor` メソッドと同じく `this` を使います。このクラスのメソッドにおける `this` は「[関数とthisの章](#)」で学んだメソッドと同じくベースオブジェクトを参照します。

```
class クラス {
  メソッド() {
    // ここでの`this`はベースオブジェクトを参照
  }
}

const インスタンス = new クラス();
// メソッド呼び出しのベースオブジェクト(`this`)は`インスタンス`となる
インスタンス.メソッド();
```

クラスのプロトタイプメソッド定義では、オブジェクトにおけるメソッドとは異なり `key : value` のように `:` 区切りでメソッドを定義できることに注意してください。つまり、次のような書き方は構文エラー（`syntaxError`）となります。

```
// クラスでは次のようにメソッドを定義できない
class クラス {
  // SyntaxError
  メソッド: () => {}
  // SyntaxError
  メソッド: function(){}
}
```

このようにクラスに対して定義したメソッドは、クラスの各インスタンスから共有されるメソッドとなります。このインスタンス間で共有されるメソッドのことをプロトタイプメソッドと呼びます。また、プロトタイプメソッドはインスタンスから呼び出せるメソッドであるためインスタンスマソッドとも呼ばれます。

この書籍では、プロトタイプメソッド（インスタンスマソッド）を `クラス#メソッド名` のように表記します。

次のコードでは、`Counter` クラスに `increment` メソッド（`Counter#increment` メソッド）を定義しています。`Counter` クラスのインスタンスはそれぞれ別々の状態（`count` プロパティ）を持ちます。

```
class Counter {
  constructor() {
    this.count = 0;
  }
  // `increment` メソッドをクラスに定義する
  increment() {
    // `this` は `Counter` のインスタンスを参照する
    this.count++;
  }
}
const counterA = new Counter();
const counterB = new Counter();
// `counterA.increment()` のベースオブジェクトは `counterA` のインスタンス
counterA.increment();
// 各インスタンスの `count` プロパティ（状態）は異なる
console.log(counterA.count); // => 1
console.log(counterB.count); // => 0
```

また `increment` メソッドはプロトタイプメソッドとして定義されています。プロトタイプメソッドは各インスタンス間で共有されます。そのため、次のように各インスタンスの `increment` メソッドの参照先は同じとなっていることが分かります。

```
class Counter {
  constructor() {
    this.count = 0;
  }
  increment() {
    this.count++;
  }
}
const counterA = new Counter();
const counterB = new Counter();
// 各インスタンスオブジェクトのメソッドは共有されている（同じ関数を参照している）
console.log(counterA.increment === counterB.increment); // => true
```

プロトタイプメソッドがなぜインスタンス間で共有されているのかは、クラスの継承の仕組みと関連するため後ほど詳細に解説します。

## クラスのインスタンスに対してメソッドを定義する

`class` 構文でのメソッド定義はプロトタイプメソッドとなり、インスタンス間で共有されます。

一方、クラスのインスタンスに対して直接メソッドを定義することも可能です。これは、コンストラクタ関数内でインスタンスオブジェクトに対してメソッドを定義するだけです。

次のコードでは、`Counter` クラスのコンストラクタ関数で、インスタンスオブジェクトに `increment` メソッドを定義しています。コンストラクタ関数内で `this` はインスタンスオブジェクトを示すため、`this` に対してメソッドを定義しています。

```
class Counter {
  constructor() {
    this.count = 0;
    this.increment = () => {
      // `this` は `constructor` メソッドにおける `this`（インスタンスオブジェクト）を参照する
      this.count++;
    };
  }
}
```

```

const counterA = new Counter();
const counterB = new Counter();
// `counterA.increment()`のベースオブジェクトは`counterA`インスタンス
counterA.increment();
// 各インスタンスのもつプロパティ(状態)は異なる
console.log(counterA.count); // => 1;
console.log(counterB.count); // => 0

```

この方法で定義した `increment` メソッドはインスタンスから呼び出せるため、インスタンスマソッドです。しかし、インスタンスオブジェクトに定義した `increment` メソッドはプロトタイプメソッドではありません。インスタンスオブジェクトのメソッドとプロトタイプメソッドには、いくつか異なる点があります。

プロトタイプメソッドは各インスタンスから共有されているため、各インスタンスからのメソッドの参照先が同じでした。しかし、インスタンスオブジェクトのメソッドはコンストラクタで毎回同じ挙動の関数（オブジェクト）を新しく定義しています。そのため、次のように各インスタンスからのメソッドの参照先も異なります。

```

class Counter {
  constructor() {
    this.count = 0;
    this.increment = () => {
      this.count++;
    };
  }
}
const counterA = new Counter();
const counterB = new Counter();
// 各インスタンスオブジェクトのメソッドの参照先は異なる
console.log(counterA.increment === counterB.increment); // => false

```

また、プロトタイプメソッドとはことなり、インスタンスオブジェクトへのメソッド定義はArrow Functionが利用できます。Arrow Functionには `this` が静的に決まるという性質があるため、メソッドにおける `this` の参照先をインスタンスに固定できます。なぜならArrow Functionで定義した `increment` メソッドはどのような呼び出し方をしても、必ず `constructor` における `this` となるためです。（「[Arrow Functionでコールバック関数を扱う](#)」を参照）

```

"use strict";
class ArrowClass {
  constructor() {
    // コンストラクタでの`this`は常にインスタンス
    this.method = () => {
      // Arrow Functionにおける`this`は静的に決まる
      // そのため`this`は常にインスタンスを参照する
      return this;
    };
  }
}
const instance = new ArrowClass();
const method = instance.method;
// ベースオブジェクトに依存しない
method(); // => instance

```

一方、プロトタイプメソッドにおける `this` はメソッド呼び出し時のベースオブジェクトを参照します。そのためプロトタイプメソッドは呼び出し方によって `this` の参照先が異なります。（[`this`を含むメソッドを変数に代入した場合の問題を参照](#)）

```

"use strict";
class PrototypeClass {
  method() {
    // `this`はベースオブジェクトを参照する
    return this;
  }
}

```

```
const instance = new PrototypeClass();
const method = instance.method;
// ベースオブジェクトはundefined
method(); // => undefined
```

## クラスのアクセサプロパティの定義

クラスに対してメソッドを定義できますが、メソッドは `メソッド名()` のように呼び出す必要があります。クラスでは、プロパティの参照（getter）、プロパティへの代入（setter）時に呼び出される特殊なメソッドを定義できます。このメソッドはプロパティのように振る舞うためアクセサプロパティと呼ばれます。

次のコードでは、プロパティの参照（getter）、プロパティへの代入（setter）に対するアクセサプロパティを定義しています。アクセサプロパティはメソッド名（プロパティ名）の前に `get` または `set` をつけるだけです。`getter`（`get`）には仮引数はありませんが、必ず値を返す必要があります。`setter`（`set`）の仮引数にはプロパティへ代入された値が入りますが、値を返す必要はありません。

```
class クラス {
  // getter
  get プロパティ名() {
    return 値;
  }
  // setter
  set プロパティ名(仮引数) {
    // setterの処理
  }
}
const インスタンス = new クラス();
インスタンス.プロパティ名; // getterが呼び出される
インスタンス.プロパティ名 = 値; // setterが呼び出される
```

次のコードでは、`NumberValue#value` をアクセサプロパティとして定義しています。`number.value` へアクセスした際にそれぞれ定義したgetterとsetterが呼ばれていることが分かります。このアクセサプロパティで実際に読み書きされているのは、`NumberValue` インスタンスの `_value` プロパティとなります。

```
class NumberValue {
  constructor(value) {
    this._value = value;
  }
  // `_value`プロパティの値を返すgetter
  get value() {
    console.log("getter");
    return this._value;
  }
  // `_value`プロパティに値を代入するsetter
  set value(newValue) {
    console.log("setter");
    this._value = newValue;
  }
}

const number = new NumberValue(1);
// "getter"とコンソールに表示される
console.log(number.value); // => 1
// "setter"とコンソールに表示される
number.value = 42;
// "getter"とコンソールに表示される
console.log(number.value); // => 42
```

## [コラム] プライベートプロパティ

`NumberValue#value` のアクセッサプロパティで実際に読み書きしているのは、`_value` プロパティです。このように、外から直接読み書きしてほしくないプロパティを`_`（アンダーバー）で開始するのはただの習慣であるため、構文としての意味はありません。

現時点（ECMAScript 2018）外から原理的に見ることができないプライベートプロパティ（hard private）を定義する構文はありません。また、現時点でも`WeakSet`などを使うことで擬似的なプライベートプロパティ（soft private）を実現できます。`WeakSet`については「[Map/Set](#)」の章で解説します。

## Array#length をアクセッサプロパティで再現する

`getter`や`setter`を利用しないと実現が難しいものとして`Array#length` プロパティがあります。`Array#length` プロパティへ値を代入すると、そのインデックス以降の値は自動的に削除される仕様があります。

次のコードでは、配列の要素数(`length` プロパティ)を小さくすると配列の要素が削除されています。

```
const array = [1, 2, 3, 4, 5];
// 要素数を減らすと、インデックス以降の要素が削除される
array.length = 2;
console.log(array.join(", ")); // => "1, 2"
// 要素数だけを増やしても、配列の中身は空要素が増えるだけ
array.length = 5;
console.log(array.join(", ")); // => "1, 2, , , "
```

この`length` プロパティの挙動を再現する`ArrayLike` クラスを実装してみます。`Array#length` プロパティは、`length` プロパティへ値を代入した際に次のようなことを行っています。

- 現在要素数より小さな要素数が指定された場合、その要素数を変更し、配列の末尾の要素を削除する
- 現在要素数より大きな要素数が指定された場合、その要素数だけを変更し、配列の実際の要素はそのままにする

つまり、`ArrayLike#length` の`setter`で要素の追加や削除を実装することで、配列のような`length` プロパティを実装できます。

```
/**
 * 配列のようなlengthを持つクラス
 */
class ArrayLike {
    constructor(items = []) {
        this._items = items;
    }

    get items() {
        return this._items;
    }

    get length() {
        return this._items.length;
    }

    set length(newLength) {
        const currentItemLength = this.items.length;
        // 現在要素数より小さな`newLength`が指定された場合、指定した要素数となるように末尾を削除する
        if (newLength < currentItemLength) {
            this._items = this.items.slice(0, newLength);
        } else if (newLength > currentItemLength) {
            // 現在要素数より大きな`newLength`が指定された場合、指定した要素数となるように末尾に空要素を追加する
            this._items = this.items.concat(new Array(newLength - currentItemLength));
        }
    }

    const arrayLike = new ArrayLike([1, 2, 3, 4, 5]);
    // 要素数を減らすとインデックス以降の要素が削除される
```

```
arrayLike.length = 2;
console.log(arrayLike.items.join(", ")); // => "1, 2"
// 要素数を増やすと末尾に空要素が追加される
arrayLike.length = 5;
console.log(arrayLike.items.join(", ")); // => "1, 2, , , "
```

このようにアクセッサプロパティは、プロパティのようありながら実際にアクセスした際には他のプロパティなどと連動する動作を実現できます。

## 静的メソッド

インスタンスマソッドはクラスをインスタンス化して利用します。一方、クラスをインスタンス化せずに利用できる静的メソッド（クラスメソッド）もあります。

静的メソッドの定義方法はメソッド名の前に、`static` をつけるだけです。

```
class クラス {
  static メソッド() {
    // 静的メソッドの処理
  }
}
// 静的メソッドの呼び出し
クラス.メソッド();
```

次のコードでは、配列をラップする `ArrayWrapper` というクラスを定義しています。`ArrayWrapper` はコンストラクタの引数として配列を受け取り初期化しています。このクラスに配列ではなく要素そのものを引数に受け取りインスタンス化できる `ArrayWrapper.of` という静的メソッドを定義しています。

```
class ArrayWrapper {
  constructor(array = []) {
    this.array = array;
  }

  // rest parametersとして要素を受け付ける
  static of(...items) {
    return new ArrayWrapper(items);
  }

  get length() {
    return this.array.length;
  }
}

// 配列を引数として渡している
const arrayWrapperA = new ArrayWrapper([1, 2, 3]);
// 要素を引数として渡している
const arrayWrapperB = ArrayWrapper.of(1, 2, 3);
console.log(arrayWrapperA.length); // => 3
console.log(arrayWrapperB.length); // => 3
```

クラスの静的メソッドにおける `this` は、そのクラス自身を参照します。そのため、先ほどのコードは `new ArrayWrapper` の代わりに `new this` と書くこともできます。

```
class ArrayWrapper {
  constructor(array = []) {
    this.array = array;
  }

  static of(...items) {
    // `this`は`ArrayWrapper`を参照する
```

```

        return new this(items);
    }

    get length() {
        return this.array.length;
    }
}

const arrayWrapper = ArrayWrapper.of(1, 2, 3);
console.log(arrayWrapper.length); // => 3

```

このように静的メソッドでの `this` はクラス自身を参照するため、インスタンスマソッドのようにインスタンスを参照することはできません。そのため静的メソッドは、クラスのインスタンスを作成する処理やクラスに関係する処理を書くために利用されます。

## 2種類のインスタンスマソッドの定義

クラスでは、2種類のインスタンスマソッドの定義方法があります。`class` 構文を使ったインスタンス間で共有されるプロトタイプメソッドの定義と、インスタンスオブジェクトに対するメソッドの定義です。

これらの2つの方法を同時に使い、1つのクラスに同じ名前でメソッドを定義した場合はどうなるでしょうか？次の `ConflictClass` ではプロトタイプメソッドとインスタンスに対して同じ `method` という名前のメソッドを定義しています。

```

class ConflictClass {
    constructor() {
        // インスタンスオブジェクトに`method`を定義
        this.method = () => {
            console.log("インスタンスオブジェクトのメソッド");
        };
    }

    // クラスのプロトタイプメソッドとして`method`を定義
    method() {
        console.log("プロトタイプのメソッド");
    }
}

const conflict = new ConflictClass();
conflict.method(); // どちらの`method`が呼び出される？

```

結論から述べるとこの場合はインスタンスオブジェクトに定義した `method` が呼び出されます。このとき、インスタンスの `method` プロパティを `delete` 演算子で削除すると、今度はプロトタイプメソッドの `method` が呼び出されます。

```

class ConflictClass {
    constructor() {
        this.method = () => {
            console.log("インスタンスオブジェクトのメソッド");
        };
    }

    method() {
        console.log("プロトタイプメソッド");
    }
}

const conflict = new ConflictClass();
conflict.method(); // "インスタンスオブジェクトのメソッド"
// インスタンスの`method`プロパティを削除
delete conflict.method;

```

```
conflict.method(); // "プロトタイプのメソッド"
```

この実行結果から次のことが分かります。

- プロトタイプメソッドとインスタンスオブジェクトのメソッドは上書きされずにどちらも定義されている
- インスタンスオブジェクトのメソッドがプロトタイプのメソッドよりも優先して呼ばれている

どちらも注意深く意識しないと気づきにくいですが、この挙動はJavaScriptの重要な仕組みであるため理解することは重要です。

この挙動はプロトタイプオブジェクトと呼ばれる特殊なオブジェクトとプロトタイプチェーンと呼ばれる仕組みで成り立っています。どちらもプロトタイプについていることからも分かるように、2つで1組のような仕組みです。

このセクションでは、プロトタイプオブジェクトとプロトタイプチェーンとはどのような仕組みなのかを見ていきま

## プロトタイプオブジェクト

プロトタイプメソッドとインスタンスオブジェクトのメソッドを同時に定義しても、互いのメソッドは上書きされるわけではありません。これは、プロトタイプメソッドはプロトタイプオブジェクトへ、インスタンスオブジェクトのメソッドはインスタンスオブジェクトへそれぞれ定義されるためです。

プロトタイプオブジェクトとは、JavaScriptの関数オブジェクトの `prototype` プロパティに自動的に作成される特殊なオブジェクトです。クラスも一種の関数オブジェクトであるため、自動的に `prototype` プロパティにプロトタイプオブジェクトが作成されています。

次のコードでは関数やクラス自身の `prototype` プロパティにプロトタイプオブジェクトが自動的に作成されていることが分かります。

```
function fn() {
}
// `prototype`プロパティにプロトタイプオブジェクトが存在する
console.log(typeof fn.prototype === "object"); // => true

class MyClass {
}
// `prototype`プロパティにプロトタイプオブジェクトが存在する
console.log(typeof MyClass.prototype === "object"); // => true
```

`class` 構文のメソッド定義は、このプロトタイプオブジェクトのプロパティとして定義されます。次のコードでは、クラスのメソッドがプロトタイプオブジェクトに定義されていることを確認できます。また、クラスには `constructor` メソッド（コンストラクタ）が必ず定義されます。この `constructor` プロパティはクラス自身を参照します。

```
class MyClass {
    method() { }
}

console.log(typeof MyClass.prototype.method === "function"); // => true
console.log(MyClass.prototype.constructor === MyClass); // => true
```

このように、プロトタイプメソッドとインスタンスオブジェクトのメソッドはそれぞれ異なるオブジェクトに定義されていることが分かります。そのため、2つの方法でメソッドを定義しても上書きされずにそれぞれ定義されます。

## プロトタイプチェーン

`class` 構文で定義したプロトタイプメソッドはプロトタイプオブジェクトに定義されます。しかし、インスタンス（オブジェクト）にはメソッドが定義されていないのに、インスタンスからクラスのプロトタイプメソッドを呼び出すことができます。

```
class MyClass {
  method() {
    console.log("プロトタイプのメソッド");
  }
}
const instance = new MyClass();
instance.method(); // "プロトタイプのメソッド"
```

このインスタンスからプロトタイプメソッドを呼び出せるのはプロトタイプチェーンと呼ばれる仕組みによるものです。プロトタイプチェーンは2つの処理から成り立ちます。

- インスタンスを作成時に、インスタンスの `[[Prototype]]` 内部プロパティへプロトタイプオブジェクトの参照を保存する処理
- インスタンスからプロパティ（またはメソッド）を参照する時に、`[[Prototype]]` 内部プロパティまで探索する処理

## インスタンス作成とプロトタイプチェーン

クラスから `new` 演算子によってインスタンスを作成する際に、インスタンスにはクラスのプロトタイプオブジェクトの参照が保存されます。このとき、インスタンスからクラスのプロトタイプオブジェクトへの参照は、インスタンスオブジェクトの `[[Prototype]]` という内部プロパティに保存されます。

`[[Prototype]]` 内部プロパティは仕様上定められた内部的な表現であるため、通常のプロパティのようにアクセスできません。しかし、`Object.getPrototypeOf(object)` メソッドで `object` の `[[Prototype]]` 内部プロパティを読み取れます。次のコードでは、インスタンスの `[[Prototype]]` 内部プロパティがクラスのプロトタイプオブジェクトを参照していることを確認できます。

```
class MyClass {
  method() {
    console.log("プロトタイプのメソッド");
  }
}
const instance = new MyClass();
// instanceの`[[Prototype]]`内部プロパティは`MyClass.prototype`と一致する
const Prototype = Object.getPrototypeOf(instance);
console.log(Prototype === MyClass.prototype); // => true
```

ここで重要なのは、インスタンスはどのクラスから作られたかやそのクラスのプロトタイプオブジェクトを知っているということです。

### Note: `[[Prototype]]` 内部プロパティを読み書きする

`Object.getPrototypeOf(object)` で `object` の `[[Prototype]]` を読み取ることができます。一方、`Object.setPrototypeOf(object, prototype)` で `object` の `[[Prototype]]` に `prototype` オブジェクトを書き込めます。また、`[[Prototype]]` 内部プロパティを通常のプロパティのように扱える `__proto__` という特殊なアクセサプロパティが存在します。

しかし、これらの `[[Prototype]]` 内部プロパティを直接読み書きすることは通常の用途では行いません。また、既存のビルトインオブジェクトの動作なども変更できるため、不用意に扱うべきではないでしょう。

## プロパティを参照とプロトタイプチェーン

オブジェクトのプロパティを参照するときに、オブジェクト自身がプロパティを持っていない場合でもそこで探索が終わるわけではありません。オブジェクトの [[Prototype]] 内部プロパティのプロトタイプオブジェクトに対しても探索を続けます。これは、スコープに指定した識別子の変数がなかった場合に外側のスコープへと探索するスコープチェーンと良く似た仕組みです。

つまり、オブジェクトがプロパティを探索するときは次のような順番で、それぞれのオブジェクトを調べます。すべてのオブジェクトにおいて見つからなかった場合の結果は `undefined` を返します。

1. `instance` オブジェクト自身
2. `instance` オブジェクトの [[Prototype]] の参照先（プロトタイプオブジェクト）

次のコードでは、インスタンスオブジェクト自身は `method` プロパティを持っていません。そのため、実際参照しているのはクラスのプロトタイプオブジェクトの `method` プロパティです。

```
class MyClass {
  method() {
    console.log("プロトタイプのメソッド");
  }
}
const instance = new MyClass();
// インスタンスには`method`プロパティがないため、プロトタイプオブジェクトの`method`が参照される
instance.method(); // "プロトタイプのメソッド"
// `instance.method`の参照はプロトタイプオブジェクトの`method`と一致する
const Prototype = Object.getPrototypeOf(instance);
console.log(instance.method === Prototype.method); // => true
```

このように、インスタンス（オブジェクト）に `method` が定義されていなくても、クラスのプロトタイプオブジェクトの `method` を呼び出すことができます。このプロパティを参照する際に、オブジェクト自身から [[Prototype]] 内部プロパティへと順番に探す仕組みのことをプロトタイプチェーンと呼びます。

プロトタイプチェーンの仕組みを擬似的なコードとして表現すると次のような動きをしています。

```
// プロトタイプチェーンの擬似的な動作の擬似的なコード
class MyClass {
  method() {
    console.log("プロトタイプのメソッド");
  }
}
const instance = new MyClass();
// `instance.method()`を行う際には、次のような呼び出し処理が行われている
// インスタンス自身が`method`プロパティを持っている場合
if (instance.hasOwnProperty("method")) {
  instance.method();
} else {
  // インスタンスの`[[Prototype]]`の参照先（`MyClass`のプロトタイプオブジェクト）を取り出す
  const prototypeObject = Object.getPrototypeOf(instance);
  // プロトタイプオブジェクトが`method`プロパティを持っている場合
  if (prototypeObject.hasOwnProperty("method")) {
    // `this`はインスタンス自身を指定して呼び出す
    prototypeObject.method.call(instance);
  }
}
```

プロトタイプチェーンの仕組みによって、プロトタイプオブジェクトに定義したプロトタイプメソッドがインスタンスから呼び出すことができています。

普段は、プロトタイプオブジェクトやプロトタイプチェーンといった仕組みを意識する必要はありません。

`class` 構文はこのようなプロトタイプを意識せずにクラスを利用できるように導入された構文です。しかし、プロトタイプベースである言語のJavaScriptではクラスをこのようなプロトタイプを使い表現していることは知っておくと

よいでしょう。

## 継承

`extends` キーワードを使うことで既存のクラスを継承できます。ここで継承とはクラスの構造や機能を引き継いだ新しいクラスを定義することを言います。

### 継承したクラスの定義

`extends` キーワードを使って既存のクラスを継承した新しいクラスを定義してみます。`class` 構文の右辺に `extends` キーワードで継承元となる親クラス（基底クラス）を指定することで、親クラスを継承した子クラス（派生クラス）を定義できます。

```
class 子クラス extends 親クラス {
}
```

次のコードでは、`Parent` クラスを継承した `Child` クラスを定義しています。子クラスである `Child` クラスのインスタンス化は通常のクラスと同じく `new` 演算子を使って行います。

```
class Parent {
}
class Child extends Parent {
}
const instance = new Child();
```

## super

`extends` を使って定義した子クラスから親クラスを参照するには `super` というキーワードを利用します。もっともシンプルな `super` を使う例としてコンストラクタの処理を見ていきます。

`class` 構文でも紹介しましたが、クラスは必ず `constructor` メソッド（コンストラクタ）をもちます。これは、継承した子クラスでも同じです。

次のコードでは、`Parent` クラスを継承した `Child` クラスのコンストラクタで、`super()` を呼び出しています。`super()` は子クラスから親クラスの `constructor` メソッドを呼び出します。（`super()` は子クラスのコンストラクタ以外では書くことができません）

```
// 親クラス
class Parent {
    constructor(...args) {
        console.log("Parentコンストラクタの処理", ...args);
    }
}
// Parentを継承したChildクラスの定義
class Child extends Parent {
    constructor(...args) {
        // Parentのコンストラクタ処理を呼び出す
        super(...args);
        console.log("Childコンストラクタの処理", ...args);
    }
}
const child = new Child("引数1", "引数2");
// "Parentコンストラクタの処理", "引数1", "引数2"
// "Childコンストラクタの処理", "引数1", "引数2"
```

`class` 構文でのクラス定義では、`constructor` メソッド（コンストラクタ）で何も処理を行わない場合は省略できることを紹介しました。これは、継承した子クラスでも同じです。

次のコードでは、`Child` クラスのコンストラクタでは何も処理を行っていません。そのため、`Child` クラスの `constructor` メソッドの定義を省略できます。

```
class Parent {}
class Child extends Parent {}
```

このように子クラスで `constructor` を省略した場合は次のように書いた場合と同じ意味になります。

```
class Parent {}
class Child extends Parent {
  constructor(...args) {
    super(...args); // 親クラスに引数をそのまま渡す
  }
}
```

## コンストラクタの処理順は親クラスから子クラスへ

コンストラクタの処理順は親クラスから子クラスへと順番が決まっています。

`class` 構文ではかならず親クラスのコンストラクタ処理（`super()` を呼び）を先に行い、その次に子クラスのコンストラクタ処理を行います。子クラスのコンストラクタでは、`this` を触る前に `super()` で親クラスのコンストラクタ処理を呼び出さないと `syntaxError` となるためです。

次のコードでは、`Parent` と `Child` でそれぞれインスタンス（`this`）の `name` プロパティに値を書き込んでいます。子クラスでは先に `super()` を呼び出してからでないと `this` を参照することはできません。そのため、コンストラクタの処理順は `Parent` から `Child` という順番に限定されます。

```
class Parent {
  constructor() {
    this.name = "Parent";
  }
}
class Child extends Parent {
  constructor() {
    // 子クラスでは`super()`を`this`に触る前に呼び出さなければならない
    super();
    // 子クラスのコンストラクタ処理
    // 親クラスで書き込まれた`name`は上書きされる
    this.name = "Child";
  }
}
const parent = new Parent();
console.log(parent.name); // => "Parent";
const child = new Child();
console.log(child.name); // => "Child";
```

## プロトタイプ継承

次のコードでは `extends` キーワードを使い `Parent` クラスを継承した `Child` クラスを定義しています。`Parent` クラスでは `method` を定義しているため、これを継承している `Child` クラスのインスタンスからも呼び出せます。

```
class Parent {
  method() {
    console.log("Parent#method");
  }
}
```

```
// `Parent`を継承した`Child`を定義
class Child extends Parent {
    // methodの定義はない
}
// `Child`のインスタンスは`Parent`のプロトタイプメソッドを継承している
const instance = new Child();
instance.method(); // "Parent#method"
```

このように、子クラスのインスタンスから親クラスのプロトタイプメソッドもプロトタイプチェーンの仕組みによって呼びだせます。

`extends` によって継承した場合、子クラスのプロトタイプオブジェクトの `[[Prototype]]` 内部プロパティには親クラスのプロトタイプオブジェクトが設定されます。このコードでは、`Child.prototype` オブジェクトの `[[Prototype]]` 内部プロパティには `Parent.prototype` が設定されます。

これにより、プロパティを参照する場合には次のような順番でオブジェクトを探索しています。

1. `instance` オブジェクト自身
2. `Child.prototype` (`instance` オブジェクトの `[[Prototype]]` の参照先)
3. `Parent.prototype` (`Child.prototype` オブジェクトの `[[Prototype]]` の参照先)

このプロトタイプチェーンの仕組みより、`method` プロパティは `Parent.prototype` オブジェクトに定義されたものを参照します。

このようにJavaScriptでは `class` 構文と `extends` キーワードを使うことでクラスの機能を継承できます。`class` 構文ではプロトタイプオブジェクトと参照する仕組みによって継承が行われています。そのため、この継承の仕組みをプロトタイプ継承と呼びます。

## 静的メソッドの継承

インスタンスはクラスのプロトタイプオブジェクトとの間にプロトタイプチェーンがあります。クラス自身（クラスのコンストラクタ）も親クラス自身（親クラスのコンストラクタ）との間にプロトタイプチェーンがあります。

これは簡単にいえば、静的メソッドも継承されるということです。

```
class Parent {
    static hello() {
        return "Hello";
    }
}
class Child extends Parent {}
console.log(Child.hello()); // => "Hello"
```

`extends` によって継承した場合、子クラスのコンストラクタの `[[Prototype]]` 内部プロパティには親クラスのコンストラクタが設定されます。このコードでは、`Child` コンストラクタの `[[Prototype]]` 内部プロパティに `Parent` コンストラクタが設定されます。

つまり、先ほどのコードでは `Child.hello` プロパティを参照した場合には次のような順番でオブジェクトを探索しています。

1. `Child` コンストラクタ
2. `Parent` コンストラクタ (`Child` コンストラクタの `[[Prototype]]` の参照先)

クラスのコンストラクタ同士にもプロトタイプチェーンの仕組みがあるため、子クラスは親クラスの静的メソッドを呼び出せます。

## super プロパティ

子クラスから親クラスのコンストラクタ処理を呼び出すには `super()` を使います。同じように、子クラスのプロトタイプメソッドからは、`super.プロパティ名` で親クラスのプロトタイプメソッドを参照できます。

次のコードでは、`Child#method` の中に `super.method()` と書くことで `Parent#method` を呼び出しています。このように、子クラスから継承元の親クラスのプロトタイプメソッドは `super.プロパティ名` で参照できます。

```
class Parent {
  method() {
    console.log("Parent#method");
  }
}
class Child extends Parent {
  method() {
    console.log("Child#method");
    // `this.method()``だと自分(`this`)のmethodを呼び出して無限ループする
    // そのため明示的に`super.method()``とParent#methodを呼びだす
    super.method();
  }
}
const child = new Child();
child.method();
// コンソールには次のように出力される
// "Child#method"
// "Parent#method"
```

プロトタイプチェーンでは、インスタンスからクラス、さらに親のクラスと継承関係をさかのぼるようにメソッドを探索すると紹介しました。このコードでは `child#method` が定義されているため、`child.method` は `child#method` を呼び出します。そして `child#method` は `super.method` を呼び出しているため、`Parent#method` が呼び出されます。

クラスの静的メソッド同士も同じように `super.method()` と書くことで呼び出せます。次のコードでは、`Parent` を継承した `Child` から親クラスの静的メソッドを呼び出しています。

```
class Parent {
  static method() {
    console.log("Parent.method");
  }
}
class Child extends Parent {
  static method() {
    console.log("Child.method");
    // `super.method()``で`Parent.method`を呼びだす
    super.method();
  }
}
Child.method();
// コンソールには次のように出力される
// "Child.method"
// "Parent.method"
```

## 継承の判定

あるクラスが指定したクラスをプロトタイプ継承しているかは `instanceof` 演算子を使って判定できます。

次のコードでは、`Child` のインスタンスは `Child` クラスと `Parent` クラスを継承したオブジェクトであることを確認しています。

```
class Parent {}
class Child extends Parent {}

const parent = new Parent();
const child = new Child();
// `Parent`のインスタンスは`Parent`のみを継承したインスタンス
```

```

console.log(parent instanceof Parent); // => true
console.log(parent instanceof Child); // => false
// `Child`のインスタンスは`Child`と`Parent`を継承したインスタンス
console.log(child instanceof Parent); // => true
console.log(child instanceof Child); // => true

```

より具体的な継承の使い方については「[ユースケース:Todoアプリ](#)」見ておきます。

## ビルトインオブジェクトの継承

ここまで自身が定義したクラスを継承してきましたが、ビルトインオブジェクトのコンストラクタも継承できます。ビルトインオブジェクトには `Array`、`String`、`Object`、`Number`、`Error`、`Date` などのコンストラクタがあります。`class` 構文ではこれらのビルトインオブジェクトを継承できます。

次のコードでは、ビルトインオブジェクトである `Array` を継承して独自のメソッドを加えた `MyArray` クラスを定義しています。継承した `MyArray` は `Array` の性質—つまりメソッドや状態管理についての仕組みを継承しています。継承した性質に加えて、`MyArray#first` や `MyArray#last` といったアクセサプロパティを追加しています。

```

class MyArray extends Array {
  get first() {
    if (this.length === 0) {
      return undefined;
    } else {
      return this[0];
    }
  }

  get last() {
    if (this.length === 0) {
      return undefined;
    } else {
      return this[this.length - 1];
    }
  }
}

// Arrayを継承しているのでArray.fromも継承している
// Array.fromはIterableなオブジェクトから配列インスタンスを作成する
const array = MyArray.from([1, 2, 3, 4, 5]);
console.log(array.length); // => 5
console.log(array.first); // => 1
console.log(array.last); // => 5

```

`Array` を継承した `MyArray` は、`Array` が元々もつ `length` プロパティや `Array.from` メソッドなどを継承し利用できます。

糖衣構文。`class` 構文でのみしか実現できない機能ではなく、読みやすさや分かりやすさのために導入された構文という側面もあるためJavaScriptの `class` 構文は糖衣構文と呼ばれることがあります。 ↩

## 例外処理

この章ではJavaScriptにおける例外処理について学びます。

### try...catch構文

try...catch構文は例外が発生しうるブロックをマークし、例外が発生したときの処理を記述するための構文です。次のように、try文にはひとつのtryブロックがあり、tryブロック内で発生した例外をcatch節でキャッチします。

tryブロック内で例外が発生すると、それ以降の文は実行されずcatch節に処理が移ります。finally節が存在するときには、例外がなげられたかどうかにかかわらず、かならずtry文の最後に実行されます。

```
try {
  console.log("この文は実行されます");
  // 未定義の関数を呼び出してReferenceError例外が発生する
  undefinedFunction();
  console.log("この文は実行されません");
} catch (error) {
  // 例外が発生したあとはこのブロックが実行される
  console.log("この文は実行されます");
  console.log(error instanceof ReferenceError); // => true
  console.log(error.message); // => "undefinedFunction is not defined"
} finally {
  // このブロックはかならず実行される
  console.log("この文は実行されます");
}
```

また、catch節とfinally節のうち、片方が存在していれば、もう片方の節は省略できます。finally節のみを書いた場合は例外がキャッチされないため、finally節を実行後に例外が発生します。

```
// catch節のみ
try {
  undefinedFunction();
} catch (error) {
  console.log(error);
}

// finally節のみ
try {
  undefinedFunction();
} finally {
  console.log("この文は実行されます");
}

// 上記のtry-finallyで例外がキャッチされていないため
console.log("この文は実行されません");
```

### throw文

throw文を使うとユーザーが例外を投げることができます。例外として投げられたオブジェクトは、catch節で関数の引数のようにアクセスできます。catch節でオブジェクトを参照できる識別子を[例外識別子](#)と呼びます。

```
try {
  // 独自の例外を投げる
  throw new Error("例外が投げられました");
} catch (error) {
```

```
// catch節のスコープでerrorにアクセスできる
console.log(error.message); // => "例外が投げられました"
}
```

## エラーオブジェクト

`throw` 文ではエラーオブジェクトを例外として投げることができます。

### Error

`Error` オブジェクトのインスタンスは `Error` を `new` して作成します。コンストラクタの第一引数には、エラーの内容をあらわす文字列を渡します。渡した文字列は `Error#message` プロパティに格納されます。

次の例では、`assertPositiveNumber` 関数でエラーオブジェクトを作成し、例外として `throw` しています。投げられたオブジェクトは `catch` 節の例外識別子から取得でき、エラーメッセージが確認できます。

```
// 渡された数値が0未満であれば例外を投げる関数
function assertPositiveNumber(num) {
    if (num < 0) {
        throw new Error(`${num} is not positive.`);
    }
}

try {
    assertPositiveNumber(-1);
} catch (error) {
    console.log(error instanceof Error); // => true
    console.log(error.message); // => "-1 is not positive."
}
```

`throw` 文はあらゆるオブジェクトを例外として投げられますが、基本的には `Error` オブジェクトのインスタンスを投げることが推奨されます。その理由は後述するスタックトレースのためです。`Error` オブジェクトはインスタンスの作成時に、そのインスタンスが作成されたファイル名や行数などのデバッグに役立つ情報をもっています。文字列のような `Error` オブジェクトでないオブジェクトを投げてしまうと、スタックトレースが得られません。

```
// 文字列を例外として投げるアンチパターンの例
try {
    throw "例外が投げられました";
} catch (error) {
    console.log(error); // => "例外が投げられました"
}
```

## ビルトインエラー

JavaScriptエンジンが投げる組み込みのエラーのことをビルトインエラーと呼びます。ビルトインエラーとして投げられるエラーオブジェクトは、すべて `Error` オブジェクトから派生したオブジェクトのインスタンスです。そのため、ユーザーが定義したエラーと同じように例外処理できます。

ビルトインエラーはいくつか種類がありますが、ここでは代表的なものを紹介します。

### ReferenceError

`ReferenceError` は存在しない変数や関数などの識別子が参照された場合のエラーです。たとえば次のようなコードを実行すると、`ReferenceError` 例外が投げられます。

```
try {
```

```
// 存在しない変数を参照する
console.log(x);
} catch (error) {
  console.log(error instanceof ReferenceError); // => true
  console.log(error.name); // => "ReferenceError"
  console.log(error.message); // エラーの内容が表示される
}
```

## SyntaxError

`SyntaxError`は構文的に不正なコードを解釈しようとした場合のエラーです。`SyntaxError`例外はJavaScriptを実行する前のパース段階で発生します。この構文エラーは`try...catch`文で`catch`することはできません。

```
// 正しくない構文
foo! bar!
```

たとえば次のようなコードを実行すると、`SyntaxError`例外が発生します。ここでは`eval`関数を使って動的にJavaScriptを解釈することで、実行時に`SyntaxError`を発生させています。

```
try {
  // eval関数は渡した文字列をJavaScriptとして実行する関数
  // 正しくない構文をパースさせ、SyntaxErrorを実行時に発生させる
  eval("foo! bar!");
} catch (error) {
  console.log(error instanceof SyntaxError); // => true
  console.log(error.name); // => "SyntaxError"
  console.log(error.message); // エラーの内容が表示される
}
```

## TypeError

`TypeError`は値が期待される型でない場合のエラーです。たとえば次のようなコードを実行すると、`TypeError`例外が投げられます。

```
try {
  // 関数でないオブジェクトを関数として呼び出す
  const fn = {};
  fn();
} catch (error) {
  console.log(error instanceof TypeError); // => true
  console.log(error.name); // => "TypeError"
  console.log(error.message); // エラーの内容が表示される
}
```

## ビルトインエラーを投げる

ユーザーがビルトインエラーのインスタンスを作成することもできます。通常の`Error`オブジェクトと同じように、それぞれのオブジェクトを`new`します。たとえば関数の引数を数値に限定したい場合は、次のように`TypeError`例外を投げるとよいでしょう。メッセージを確認しなくとも、エラーの名前だけで型に関する例外だとすぐにわかります。

```
// 文字列を反転する関数
function reverseString(str) {
  if (typeof str !== "string") {
    throw new TypeError(`"${str}" is not a string`);
  }
  return str.split("").reverse().join("");
}
```

```

try {
    // 数値を渡す
    reverseString(100);
} catch (error) {
    console.log(error instanceof TypeError); // => true
    console.log(error.name); // => "TypeError"
    console.log(error.message); // "100 is not a string"
}

```

## エラーとデバッグ

JavaScript開発においてデバッグ中に発生したエラーを理解することは非常に重要です。エラーがもつ情報を活用することで、ソースコードのどこでどのような例外が投げられたのか知ることができます。

エラーはすべて `Error` オブジェクトを拡張したオブジェクトで宣言されています。つまり、エラーの名前をあらわす `name` プロパティと内容をあらわす `message` プロパティをもっています。この2つのプロパティを確認することで、多くの場面で開発の助けとなるでしょう。

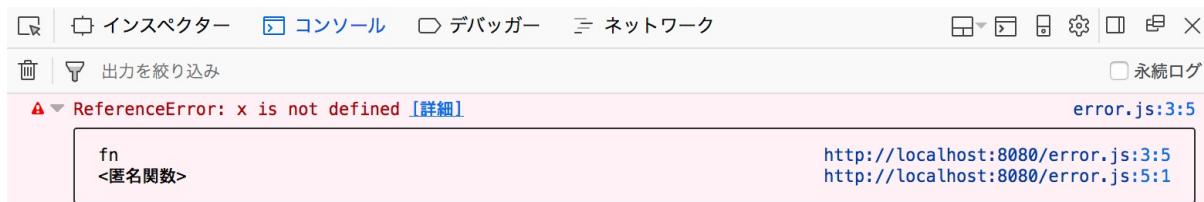
次のコードではtry文で囲っていない部分で例外を投げています。

```

function fn() {
    // 存在しない変数を参照する
    x++;
}
fn();

```

このスクリプトを読み込むと、投げられた例外についてのログがコンソールに出力されます。ここではFirefoxにおける実行例を示します。



»

このエラーログには次の情報が含まれています。

メッセージ	意味
<code>ReferenceError: x is not defined</code>	エラーの種類は <code>ReferenceError</code> で、 <code>x</code> が未定義であること
<code>error.js:3:5</code>	例外が <code>error.js</code> の3行目5列目で発生したこと。つまり <code>x++;</code> であること。

また、メッセージの後には例外のスタックトレースが表示されています。

- スタックトレースの最初の行が実際に例外が発生した場所です。つまり、3行目の `x++;` で例外が発生しています
- 次の行には、そのコードの呼び出し元が記録されています。つまり、3行目のコードを実行したのは5行目の `fn` 関数の呼び出しだす

このように、スタックトレースは上から下へ呼び出し元を辿れるように記録されています。

コンソールに表示されるエラーログには多くの情報が含まれています。MDNの[JavaScriptエラーリファレンス](#)には、ブラウザが投げるビルトインのエラーについて種類とメッセージが網羅されています。開発中にビルトインエラーが発生したときには、リファレンスを見て解決方法を探すとよいでしょう。

## 非同期処理

この章ではJavaScriptにおける非同期処理について学んで行きます。非同期処理はJavaScriptにおいてはとても重要な概念です。また、JavaScriptを扱うブラウザやNode.jsなどにおいて非同期処理のみのAPIも多いため、非同期処理を避けることはできません。そのため、非同期処理をあつかうためのパターンやPromiseというビルトインオブジェクト、さらにはAsync Functionとよばれる構文的なサポートがあります。

この章では非同期処理とはどのようなものかという話から、非同期処理での例外処理、非同期処理の扱い方を見ていきます。

## 同期処理

多くのプログラミング言語ではコードの評価の仕方として同期処理（sync）と非同期処理（async）という大きな分類があります。

今まで書いていたコードは同期処理と呼ばれているもので、コードを順番に文と式を評価したらその評価結果がその場で返されます。

同期処理ではコードを順番に処理していき、ひとつの処理が終わるまで次の処理は行いません。同期処理では実行している処理はひとつだけとなるため、とても直感的な動作となります。

一方、同期的にブロックする処理が行われていた場合には問題があります。同期処理ではひとつの処理が終わるまで次の処理を行うことができないためです。

次のコードの `blockTime` 関数は指定した `timeout` ミリ秒だけ無限ループを行い同期的にブロックする処理です。この `blockTime` 関数を呼び出すと、指定時間経過するまで次の処理（次の行）が呼ばれません。

```
// 指定した`timeout`ミリ秒経過するまで同期的にブロックする関数
function blockTime(timeout) {
    const startTime = Date.now();
    // `timeout`ミリ秒経過するまで無限ループをする
    while (true) {
        const diffTime = Date.now() - startTime;
        if (diffTime >= timeout) {
            return; // 指定時間経過したら関数の実行を終了
        }
    }
}
console.log("処理を開始");
blockTime(1000); // 他の処理を1000ミリ秒（1秒間）ブロックする
console.log("この行が呼ばれるまで処理が1秒間ブロックされる");
```

このような同期的にブロックするのは、ブラウザでは大きな問題となります。なぜなら、JavaScriptは基本的にブラウザのメインスレッド（UIスレッドとも呼ばれる）で実行されるためです。そのため、JavaScriptで同期的にブロックする処理を行うと他の処理ができなくなるため、画面がフリーズしたような体感を与えてしまいます。

さきほどの例では1秒間も処理をブロックしているため、1秒間スクロールやクリックなどの他の操作が効かないといった悪影響がでます。

## 非同期処理

非同期処理は、コードを順番に文と式を評価したら処理は開始されますが、その評価結果を返しません。（処理が開始されたことを表すオブジェクトなどを返すことはありますが、最終的な評価結果はすぐには手に入りません）

また非同期処理はコードを順番に処理していきますが、ひとつの非同期処理が終わるのを待たずに次の処理を評価します。つまり、非同期処理では同時に実行している処理は複数あります。

JavaScriptにおいて代表的な非同期処理を行う関数として `setTimeout` 関数があります。`setTimeout` 関数は `delay` ミリ秒後に、`コールバック関数` を呼び出すようにタイマーへ登録する非同期処理です。

```
setTimeout(コールバック関数, delay);
```

次のコードでは `setTimeout` 関数を使い10ミリ秒後に同期的にブロックを行います。`setTimeout` 関数でタイマーに登録したコールバック関数は非同期的なタイミングで呼ばれます。そのため `setTimeout` 関数の次の行に書かれている同期的処理は、非同期処理よりも先に実行されます。

```
// 指定した`timeout`ミリ秒経過するまで同期的にブロックする関数
function blockTime(timeout) {
  const startTime = Date.now();
  while (true) {
    const diffTime = Date.now() - startTime;
    if (diffTime >= timeout) {
      return; // 指定時間経過したら関数の実行を終了
    }
  }
}

console.log("1. setTimeoutのコールバック関数を10ミリ秒後に実行します");
setTimeout(() => {
  console.log("3. ブロックする処理を開始します");
  blockTime(1000); // 他の処理を1秒間ブロックする
  console.log("4. ブロックする処理が完了しました");
}, 10);
// ブロックする処理は非同期なタイミングで呼び出されるので、次の行が先に実行される
console.log("2. 同期的な処理を実行します");
```

このコードを実行した結果のコンソールログは次のようにになります。

1. `setTimeout` のコールバック関数を10ミリ秒後に実行します
2. 同期的な処理を実行します
3. ブロックする処理を開始します
4. ブロックする処理が完了しました

このように、非同期処理（`setTimeout` のコールバック関数）は、コードの見た目上の並びとは異なる順番で実行されることがわかります。

## 非同期処理はメインスレッドで実行される

JavaScriptにおいて多くの非同期処理はメインスレッドで実行されます。メインスレッドはUIスレッドとも呼ばれ、重たいJavaScriptの処理はメインスレッドで実行する他の処理（画面の更新など）をブロックする問題について紹介しました。（ECMAScriptの仕様として規定されているわけではないため、すべてがメインスレッドで実行されているわけではありません）

非同期処理は名前から考えるとメインスレッド以外で実行されるように見えますが、基本的には非同期処理も同期処理と同じようにメインスレッドで実行されます。このセクションでは非同期処理がどのようにメインスレッドで実行されているかを簡潔に見ていきます。

次のコードは、`setTimeout` 関数でタイマーに登録したコールバック関数が呼ばれるまで、実際にどの程度の時間がかかったかを計測しています。また、`setTimeout` 関数でタイマーに登録した次の行で同期的にブロックする処理を実行しています。

非同期処理（コールバック関数）がメインスレッド以外のスレッドで実行されるならば、この非同期処理はメインスレッドでの同期的にブロックする処理の影響を受けないはずです。しかし、実際にはこの非同期処理もメインスレッドで実行された同期的にブロックする処理の影響を受けます。

次のコードを実行すると `setTimeout` 関数で登録したコールバック関数は、タイマーに登録した時間（10ミリ秒後）よりも大きく遅れて呼び出されます。

```
// 指定した`timeout`ミリ秒経過するまで同期的にブロックする関数
function blockTime(timeout) {
    const startTime = Date.now();
    while (true) {
        const diffTime = Date.now() - startTime;
        if (diffTime >= timeout) {
            return; // 指定時間経過したら関数の実行を終了
        }
    }
}

const startTime = Date.now();
// 10ミリ秒後にコールバック関数を呼び出すようにタイマーに登録する
setTimeout(() => {
    const endTime = Date.now();
    console.log(`非同期処理のコールバックが呼ばれるまで${endTime - startTime}ミリ秒かかりました`);
}, 10);
console.log("ブロックする処理を開始します");
blockTime(1000); // 1秒間処理をブロックする
console.log("ブロックする処理が完了しました");
```

多くの環境では、このときの非同期処理のコールバックが呼ばれるまでは1000ミリ秒以上かかります。このように非同期処理も同期処理の影響を受けることからも同じスレッドで実行されていることがわかります。

JavaScriptでは一部の例外を除き非同期処理が並行処理（concurrent）として扱われます。並行処理とは、処理を一定の単位ごとに分けて処理を切り替えながら実行することです。そのため非同期処理の実行中にとても重たい処理があると、非同期処理の切り替えが遅れるという現象を引き起こします。

このようにJavaScriptの非同期処理も基本的には1つのメインスレッドで処理されていると考えても間違いないでしょう。これは `setTimeout` 関数のコールバック関数から外側のスコープのデータへのアクセス方法に制限がないことからもわかります。もし非同期処理が別スレッドで行われるならば自由なデータへのアクセスは競合状態（レースコンディション）を引き起こしてしまうためです。

ただし、非同期処理の中にもメインスレッドとは別のスレッドで実行できるAPIが実行環境によっては存在します。たとえばブラウザでは[Web Worker API](#)を使いメインスレッド以外でJavaScriptを実行できるため、非同期処理を並列処理（Parallel）できます。並列処理とは、排他的に複数の処理を同時に実行することです。

Web WorkerでのJavaScriptはメインスレッドのJavaScriptとは異なるスレッドで実行されるため、お互いに同期的なブロックする処理の影響を受けにくくなります。ただし、Web Workerとメインスレッドでのデータのやり取りには `postMessage` メソッドを利用する必要があります。そのため、`setTimeout` 関数のコールバック関数とは異なりデータへのアクセス方法にも制限がつきます。

このように、非同期処理のすべてをひとくくりにはできませんが、基本的な非同期処理（タイマーなど）はメインスレッドで実行されているという性質を知ることは大切です。JavaScriptの大部分の非同期処理は非同期的なタイミングで実行される処理であると理解しておく必要があります。

## 非同期処理と例外処理

非同期処理は処理の流れが同期処理とは異なることについて紹介しました。これは非同期処理における例外処理においても大きな影響を与えます。

同期処理では、`try...catch`構文を使うことで同期的に発生した例外はキャッチできます。（詳細は「例外処理」の章を参照）

```
try {
  throw new Error("同期的なエラー");
} catch (error) {
  console.log("同期的なエラーをキャッチできる");
}
console.log("この文は実行されます");
```

非同期処理では、`try...catch`構文を使っても非同期的に発生した例外をキャッチできません。次のコードでは、10ミリ秒後に非同期的なエラーを発生させています。しかし、`try...catch`構文では次のような非同期エラーをキャッチすることはできません。

```
try {
  setTimeout(() => {
    throw new Error("非同期的なエラー");
  }, 10);
} catch (error) {
  console.log("この文は実行されません");
}
console.log("この文は実行されます");
```

`try`ブロックはそのブロック内で発生した例外をキャッチする構文です。しかし、`setTimeout`関数で登録されたコールバック関数が実際に実行され例外を投げるのは、すべての同期処理が終わった後となります。つまり、`try`ブロックで例外が発生しようとマークした範囲外で例外が発生します。

そのため、`setTimeout`関数のコールバック関数における例外は、次のようにコールバック関数内で同期的なエラーとしてキャッチする必要があります。

```
// 非同期処理の外
setTimeout(() => {
  // 非同期処理の中
  try {
    throw new Error("エラー");
  } catch (error) {
    console.log("エラーをキャッチできる");
  }
}, 10);
console.log("この文は実行されます");
```

このようにコールバック関数内でエラーをキャッチはできますが、非同期処理の外からは非同期処理の中で例外が発生したかが分かりません。そのため、非同期処理の中で例外が発生した場合に、その例外を非同期処理の外へ伝える方法が必要です。

この非同期処理で発生した例外の扱い方についてはさまざまなパターンがあります。この章では主要な非同期処理と例外の扱い方としてエラーファーストコールバック、Promise、Async Functionの3つを見ていきます。現実のコードではすべてのパターンが使われています。そのため、非同期処理の選択肢を増やす意味でもそれぞれを理解することは重要です。

## エラーファーストコールバック

ECMAScript 2015（ES2015）でPromiseが仕様に入るまで、非同期処理中に発生した例外を扱う統一的な方法は存在しませんでした。ES2015より前までは、エラーファーストコールバックという非同期処理中に発生した例外を扱う方法を決めたルールが広く使われていました。

エラーファーストコールバックとは、次のような非同期処理におけるコールバック関数の呼び出し方を決めたルールです。

- 処理が失敗した場合は、コールバック関数の1番目の引数にエラーオブジェクトを渡して呼び出す
- 処理が成功した場合は、コールバック関数の1番目の引数には `null` を渡し、2番目以降の引数に成功時の結果を渡して呼び出す

つまり、ひとつのコールバック関数で失敗した場合と成功した場合の両方を扱うルールとなります。

たとえば、Node.jsでは `fs.readFile` 関数というファイルシステムからファイルをロードする非同期処理を行う関数があります。指定したパスのデータを読むため、ファイルが存在しない場合やアクセス権限の問題から読み取りに失敗することがあります。そのため、`fs.readFile` 関数の第2引数にわたすコールバック関数にはエラーファーストコールバックスタイルの関数を渡します。

ファイルを読み込むことに失敗した場合は、コールバック関数の1番目の引数に `Error` オブジェクトが渡されます。ファイルを読み込むことに成功した場合は、コールバック関数の1番目の引数に `null`、2番目の引数に読み込んだデータを渡します。

```
fs.readFile("./example.txt", (error, data) => {
  if (error) {
    // 読み込み中にエラーが発生しました
  } else {
    // データを読み込むことができた
  }
});
```

このエラーファーストコールバックはNode.jsでは広く使われ、Node.jsの標準APIにおいても非同期処理を行う関数では利用されています。詳しい扱い方については[ユースケース: Node.jsでCLIアプリケーション](#)にて紹介します。

実際にエラーファーストコールバックで非同期な例外処理を扱うコードを書いてみましょう。

次のコードの `dummyFetch` 関数は、擬似的なリソースの取得を行う非同期な処理です。第1引数に任意のパスを受け取り、第2引数にエラーファーストコールバックスタイルの関数を受けとります。

この `dummyFetch` 関数は、任意のパスにマッチするリソースがある場合には、第2引数のコールバック関数に `null` とレスポンスオブジェクトを渡して呼び出します。一方、任意のパスにマッチするリソースがない場合には、第2引数のコールバック関数にはエラーオブジェクトを渡して呼び出します。

```
/** 
 * 1000ミリ秒未満のランダムなタイミングでレスポンスを擬似的にデータ取得する関数
 * 指定した`path`にデータがある場合は`callback(null, レスポンス)`を呼ぶ
 * 指定した`path`にデータがない場合は`callback(エラー)`を呼ぶ
 */
function dummyFetch(path, callback) {
  setTimeout(() => {
    // /success から始まるパスにはリソースがあるという設定
    if (path.startsWith("/success")) {
      callback(null, { body: `Response body of ${path}` });
    } else {
      callback(new Error("NOT FOUND"));
    }
  }, 1000 * Math.random());
}

// /success/data にリソースが存在するので、`response`にはデータが入る
dummyFetch("/success/data", (error, response) => {
  if (error) {
    console.log(error); // この文は実行されません
  } else {
    console.log(response); // => { body: "Response body of /success/data" }
  }
});
// /failure/data にリソースは存在しないので、`error`にはエラーオブジェクトが入る
```

```
dummyFetch("/failure/data", (error, response) => {
  if (error) {
    console.log(error.message); // => "NOT FOUND"
  } else {
    console.log(response); // この文は実行されません
  }
});
```

このようにコールバック関数の1番目の引数にはエラーオブジェクトまたは `null` を入れ、それ以降の引数にデータを渡すというルール化したものをエラーファーストコールバックと呼びます。

非同期処理中に例外が発生して生じたエラーをコールバック関数で受け取る方法は他にもやり方があります。たとえば、成功したときに呼び出すコールバック関数と失敗したときに呼び出すコールバック関数の2つを受け取る方法があります。さきほどの `dummyFetch` 関数を2種類のコールバック関数を受け取る形に変更すると次のような実装になります。

```
/** 
 * リソースの取得に成功した場合は`successCallback(レスポンス)`を呼び出す
 * リソースの取得に失敗した場合は`failureCallback(エラー)`を呼び出す
 */
function dummyFetch(path, successCallback, failureCallback) {
  setTimeout(() => {
    if (path.startsWith("/success")) {
      successCallback({ body: `Response body of ${path}` });
    } else {
      failureCallback(new Error("NOT FOUND"));
    }
  }, 1000 * Math.random());
}
```

このように非同期処理の中で例外が発生した場合に、その例外を非同期処理の外へ伝える方法はさまざまな手段が考えられます。エラーファーストコールバックはその形を決めただけの共通のルールの1つです。そのため、非同期処理ではエラーファーストコールバック以外の方法が使われていることもあります。一方で、非同期処理における例外処理のルールを決めるこのメリットとして、エラーハンドリングのパターン化ができます。

しかし、エラーファーストコールバックは非同期処理におけるエラーハンドリングの書き方を決めただけのルールで仕様ではありません。そのためエラーファーストコールバックというルールを破っても、構文として問題があるわけではありません。

しかしながら、最初に書いたようにJavaScriptでは非同期処理を扱うケースが多いです。そのため、ただのルールではなくECMAScriptの仕様として非同期処理を扱う方法が求められていました。そこで、ES2015では `Promise` という非同期処理を扱うビルトインオブジェクトが導入されました。

次のセクションでは、ES2015で導入された `Promise` について見ていきます。

## [ES2015] Promise

`Promise` はES2015で導入された非同期処理の結果を表現するビルトインオブジェクトです。

エラーファーストコールバックは非同期処理を扱うコールバック関数の最初の引数にエラーオブジェクトを渡すというルールでした。`Promise` はこれを発展させたもので、単なるルールではなくオブジェクトという形にして非同期処理を統一的なインターフェースで扱うことを目的にしています。

`Promise` はビルトインオブジェクトであるためさまざまなメソッドを持ちますが、まずはエラーファーストコールバックと `Promise` での非同期処理のコード例を比較してみます。

次のコードの `asyncTask` 関数はエラーファーストコールバックを受け取る非同期処理の例です。

エラーファーストコールバックでは、非同期処理が成功した場合は、1番目の引数に `null` を渡し2番目以降の引数に結果を渡します。一方、非同期処理が失敗した場合は、1番目の引数にエラーオブジェクトを渡すというルールでした。

```
// asyncTask関数はエラーファーストコールバックを受け取る
asyncTask((error, result) => {
  if (error) {
    // 非同期処理が失敗したときの処理
  } else {
    // 非同期処理が成功したときの処理
  }
});
```

次のコードの `asyncTask` 関数は `Promise` インスタンスを返す非同期処理の例です。 `Promise` では、非同期処理に成功したときの処理を `then` メソッドへコールバック関数を渡し、失敗したときの処理を `catch` メソッドへコールバック関数を渡します。

エラーファーストコールバックとはことなり、非同期処理（`asyncTask` 関数）は `Promise` インスタンスを返しています。その返された `Promise` インスタンスに対して、成功と失敗時の処理をそれぞれコールバック関数として渡すという形になります。

```
// asyncTask関数はPromiseインスタンスを返す
asyncTask().then(()=> {
  // 非同期処理が成功したときの処理
}).catch(() => {
  // 非同期処理が失敗したときの処理
});
```

`Promise` インスタンスのメソッドによって引数に渡せるものが決められているため、非同期処理の流れも一定のやり方に統一されます。また非同期処理（`asyncTask` 関数）はコールバック関数を受け取るのでなく、`Promise` インスタンスを返すという形に変わっています。この `Promise` という統一されたインターフェースがあることで、さまざまな非同期処理のパターンを形成できます。

つまり、複雑な非同期処理を上手くパターン化できるというのが `Promise` の役割であり、`Promise` を使う理由のひとつであるといえるでしょう。このセクションでは、非同期処理を扱うビルトインオブジェクトである `Promise` について見ていきます。

## Promise インスタンスの作成

`Promise` は `new` 演算子で `Promise` のインスタンスを作成して利用します。このときのコンストラクタには `resolve` と `reject` の2つの引数を取る `executor` とよばれる関数を渡します。`executor` 関数の中で非同期処理を行い、非同期処理が成功した場合は `resolve` 関数を呼び、失敗した場合は `reject` 関数を呼びます。

```
const executor = (resolve, reject) => {
  // 非同期の処理が成功したときはresolveを呼ぶ
  // または、非同期の処理が失敗したときにはrejectを呼ぶ
};
const promise = new Promise(executor);
```

この `Promise` インスタンスの `Promise#then` メソッドで、`Promise` が `resolve`（成功）、`reject`（失敗）したときに呼ばれるコールバック関数を登録します。`then` メソッドでは2つの引数を渡すことができ、第一引数には `resolve`（成功）に呼ばれるコールバック関数、第二引数には `reject`（失敗）に呼ばれるコールバック関数を渡します。

```
// `Promise`インスタンスを作成
const promise = new Promise((resolve, reject) => {
```

```
// 非同期の処理が成功したときはresolveを呼ぶ
// または、非同期の処理が失敗したときにはrejectを呼ぶ
});
const onFulfilled = () => {
  console.log("resolveされたときに呼ばれる");
};
const onRejected = () => {
  console.log("rejectされたときに呼ばれる");
};
// `then`メソッドで成功時と失敗時に呼ばれるコールバック関数を登録
promise.then(onFulfilled, onRejected);
```

`Promise` コンストラクタの `resolve` と `reject`、`then` メソッドの `onFulfilled` と `onRejected` は次のような関係となります。

- `resolve` (成功) した時
  - `onFulfilled` が呼ばれる
- `reject` (失敗) した時
  - `onRejected` が呼ばれる

## Promise#then と Promise#catch

`Promise` のようにコンストラクタに関数を渡すパターンは今までなかったので、`then` メソッドの使い方について具体的な例を紹介します。また、`then` メソッドのエイリアスでもある `catch` メソッドについても見ていきます。

次のコードの `dummyFetch` 関数は `Promise` のインスタンスを作成して返します。`dummyFetch` 関数はリソースの取得に成功した場合は `resolve` 関数を呼び、失敗した場合は `reject` 関数を呼びます。

`resolve` に渡した値は、`then` メソッドの1番目のコールバック関数 (`onFulfilled`) に渡されます。`reject` に渡したエラーオブジェクトは、`then` メソッドの2番目のコールバック関数 (`onRejected`) に渡されます。

```
/** 
 * 1000ミリ秒未満のランダムなタイミングでレスポンスを擬似的にデータ取得する関数
 * 指定した`path`にデータがある場合は成功として`resolve`を呼ぶ
 * 指定した`path`にデータがない場合は失敗として`reject`を呼ぶ
 */
function dummyFetch(path) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (path.startsWith("/success")) {
        resolve({ body: `Response body of ${path}` });
      } else {
        reject(new Error("NOT FOUND"));
      }
    }, 1000 * Math.random());
  });
}
// `then`メソッドで成功時と失敗時に呼ばれるコールバック関数を登録
// /success/data のリソースは存在するので成功しonFulfilledが呼ばれる
dummyFetch("/success/data").then(function onFulfilled(response) {
  console.log(response); // => { body: "Response body of /success/data" }
}, function onRejected(error) {
  console.log(error); // この文は実行されません
});
// /failure/data のリソースは存在しないのでonRejectedが呼ばれる
dummyFetch("/failure/data").then(function onFulfilled(response) {
  console.log(response); // この文は実行されません
}, function onRejected(error) {
  console.log(error); // Error: "NOT FOUND"
});
```

`Promise#then` メソッドは成功 (`onFulfilled`) と失敗 (`onRejected`) のコールバック関数の2つを受け取りますが、どちらの引数も省略できます。

次のコードの `delay` 関数は一定時間後に解決 (`resolve`) される `Promise` インスタンスを返します。この `Promise` インスタンスに対して `then` メソッドで成功時のコールバック関数だけを登録しています。

```
function delay(timeoutMs) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve();
    }, timeoutMs);
  });
}

// `then` メソッドで成功時のコールバック関数だけを登録
delay(10).then(() => {
  console.log("10ミリ秒後に呼ばれる");
});
```

一方、`then` メソッドでは失敗時のコールバック関数だけの登録もできます。このとき `then(undefined, onRejected)` のように第1引数には `undefined` を渡す必要があります。`then(undefined, onRejected)` と同様のことを行う方法として `Promise#catch` メソッドが用意されています。

次のコードでは `then` メソッドと `catch` メソッドで失敗時のエラー処理をしていますが、どちらも同じ意味となります。`then` メソッドに `undefined` を渡すのはわかりにくいため、失敗時の処理だけを登録する場合は `catch` メソッドの利用を推奨しています。

```
function errorPromise(message) {
  return new Promise((resolve, reject) => {
    reject(new Error(message));
  });
}

// 非推奨: `then` メソッドで失敗時のコールバック関数だけを登録
errorPromise("thenでエラーハンドリング").then(undefined, (error) => {
  console.log(error.message); // => "thenでエラーハンドリング"
});

// 推奨: `catch` メソッドで失敗時のコールバック関数を登録
errorPromise("catchでエラーハンドリング").catch(error => {
  console.log(error.message); // => "catchでエラーハンドリング"
});
```

## Promiseと例外

`Promise` ではコンストラクタの処理で例外が発生した場合に自動的に例外がキャッチされます。例外が発生した `Promise` インスタンスは `reject` 関数を呼びだしたのと同じように失敗したものとして扱われます。そのため、`Promise` 内で例外が発生すると `then` メソッドの第二引数や `catch` メソッドで登録したエラー時のコールバック関数が呼び出されます。

```
function throwPromise() {
  return new Promise((resolve, reject) => {
    // Promiseコンストラクタの中で例外は自動的にキャッチされrejectを呼ぶ
    throw new Error("例外が発生");
    // 例外が発生するとそれ以降の処理は実行されない
    console.log("この文は実行されません");
  });
}

throwPromise().catch(error => {
  console.log(error.message); // => "例外が発生"
});
```

このようにPromiseにおける処理では `try...catch` 構文を使わなくとも、自動的に例外がキャッチされます。

## Promiseの状態

Promiseの `then` メソッドや `catch` メソッドによる処理がわかったところで、`Promise` インスタンスの状態について整理していきます。

`Promise` インスタンスには、内部的に次の3つの状態が存在します。

- Fulfilled
  - `resolve` (成功) したときの状態。このとき `onFulfilled` が呼ばれる
- Rejected
  - `reject` (失敗) または例外が発生したときの状態。このとき `onRejected` が呼ばれる
- Pending
  - FulfilledまたはRejectedではない状態
  - `new Promise` でインスタンスを作成したときの初期状態

これらの状態はECMAScriptの仕様として決められている内部的な状態です。しかし、この状態をPromiseのインスタンスから取り出す方法はありません。そのためAPIとしてこの状態を直接扱うことはできませんが、Promiseについて理解するのに役に立ちます。

`Promise` インスタンスの状態は作成時にPendingとなり、一度でもFulfilledまたはRejectedへ変化すると、それ以降状態は変化しなくなります。そのため、FulfilledまたはRejectedの状態であることをSettled（不变）と呼びます。

一度でもSettled（FulfilledまたはRejected）となつた `Promise` インスタンスは、それ以降別の状態には変化しません。そのため、`resolve` を呼び出した後に `reject` を呼び出しても、その `Promise` インスタンスは最初に呼び出した `resolve` によってFulfilledのままとなります。

次のコードでは、`reject` を呼び出しても状態が変化しないため、`then` で登録した`onRejected` のコールバック関数は呼び出されません。`then` メソッドで登録したコールバック関数は状態が変化した場合に一度だけ呼び出されます。

```
const promise = new Promise((resolve, reject) => {
  // 非同期でresolveする
  setTimeout(() => {
    resolve();
    // 既にresolveされているため無視される
    reject(new Error("エラー"));
  }, 16);
});
promise.then(() => {
  console.log("Fulfilledとなった");
}, (error) => {
  // この行は呼び出されない
});
```

同じように、`Promise` コンストラクタ内で `resolve` を何度も呼び出しても、その `Promise` インスタンスの状態は一度しか変化しません。そのため、次のように `resolve` を何度も呼び出しても、`then` で登録したコールバック関数は一度しか呼び出されません。

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve();
    resolve(); // 2度目以降のresolveやrejectは無視される
  }, 16);
});
promise.then(() => {
  console.log("最初のresolve時の一度しか呼ばれない");
}, (error) => {
```

```
// この行は呼び出されない
});
```

このように `Promise` インスタンスの状態が変化した時に、一度だけ呼ばれるコールバック関数を登録するのが `then` や `catch` メソッドとなります。

また `then` や `catch` メソッドはすでに `Settled` へと状態が変化済みの `Promise` インスタンスに対してもコールバック関数を登録できます。状態が変化済みの `Promise` インスタンスを作成する方法として `Promise.resolve` と `Promise.reject` メソッドがあります。

## Promise.resolve

`Promise.resolve` メソッドは `Fulfilled` の状態となった `Promise` インスタンスを作成します。

```
const fulFilledPromise = Promise.resolve();
```

`Promise.resolve` メソッドは `new Promise` の糖衣構文（シンタックスシュガー）です。そのため、`Promise.resolve` メソッドは次のコードと同じ意味になります。

```
const fulFilledPromise = new Promise((resolve) => {
  resolve();
});
```

`Promise.resolve` メソッドは引数に `resolve` される値を渡すこともできます。

```
// `resolve(42)` された `Promise` インスタンスを作成する
const fulFilledPromise = Promise.resolve(42);
fulFilledPromise.then(value => {
  console.log(value); // => 42
});
```

`Promise.resolve` メソッドで作成した `Fulfilled` の状態となった `Promise` インスタンスに対しても `then` メソッドでコールバック関数を登録できます。状態が変化済みの `Promise` インスタンスに `then` メソッドで登録したコールバック関数は、常に非同期なタイミングで実行されます。

```
const promise = Promise.resolve();
promise.then(() => {
  console.log("2. コールバック関数が実行されました");
});
console.log("1. 同期的な処理が実行されました");
```

このコードを実行すると、すべての同期的な処理が実行された後に、`then` メソッドのコールバック関数が非同期なタイミングで実行されることがわかります。

`Promise.resolve` メソッドは `new Promise` の糖衣構文であるため、この実行順序は `new Promise` を使った場合も同じです。次のコードはさきほどの `Promise.resolve` メソッドを使ったものと同じ動作になります。

```
const promise = new Promise((resolve) => {
  console.log("1. resolveします");
  resolve();
});
promise.then(() => {
  console.log("3. コールバック関数が実行されました");
});
console.log("2. 同期的な処理が実行されました");
```

このコードを実行すると、まず `Promise` のコンストラクタ関数が実行され、続いて同期的な処理が実行されます。最後に `then` メソッドで登録していたコールバック関数が非同期に呼ばれることがわかります。

## Promise.reject

`Promise.reject` メソッドは `Rejected` の状態となった `Promise` インスタンスを作成します。

```
const rejectedPromise = Promise.reject(new Error("エラー"));
```

`Promise.reject` メソッドは `new Promise` の糖衣構文（シンタックスシュガー）です。そのため、`Promise.reject` メソッドは次のコードと同じ意味になります。

```
const rejectedPromise = new Promise((resolve, reject) => {
  reject(new Error("エラー"));
});
```

`Promise.reject` メソッドで作成した `Rejected` 状態の `Promise` インスタンスに対しても `then` や `catch` メソッドでコールバック関数を登録できます。`Rejected` 状態へ変化済みの `Promise` インスタンスに登録したコールバック関数は、常に非同期なタイミングで実行されます。これは `Fulfilled` の場合と同様です。

```
Promise.reject(new Error("エラー")).catch(() => {
  console.log("2. コールバック関数が実行されました");
});
console.log("1. 同期的な処理が実行されました");
```

`Promise.resolve` や `Promise.reject` を使うことで短くかけるため、テストコードなどで利用されることがあります。また、`Promise.reject` は次に解説する Promise チェーンにおいて、Promise の状態を操作することに利用できます。

## Promise チェーン

Promise は非同期処理における統一的なインターフェイスを提供するビルトインオブジェクトです。Promise による統一的な処理方法は複数の非同期処理を扱う場合に特に効力を発揮します。これまで、1つの `Promise` インスタンスに対して `then` や `catch` メソッドで1組のコールバック処理を登録するだけでした。

非同期処理が終わったら次の非同期処理というように、複数の非同期処理を順番に扱いたい場合もあります。Promise ではこのような複数の非同期処理からなる一連の非同期処理を簡単に書く方法が用意されています。

この仕組みのキーとなるのが `then` や `catch` メソッドは常に新しい `Promise` インスタンスを作成して返すという仕様です。そのため `then` メソッドの返り値である `Promise` インスタンスにさらに `then` メソッドで処理を登録できます。これはメソッドチェーンと呼ばれる仕組みですが、この書籍では Promise をメソッドチェーンでつなぐことを Promise チェーンと呼びます（詳細は「[配列](#)」の章を参照）。

次のコードでは、`then` メソッドで Promise チェーンをしています。Promise チェーンでは、Promise が失敗（`Rejected` 状態）しない限り、順番に `then` メソッドで登録した成功時のコールバック関数を呼び出します。

```
// Promiseインスタンスでメソッドチェーン
Promise.resolve()
  // thenメソッドは新しいPromiseインスタンスを返す
  .then(() => {
    console.log(1);
  })
  .then(() => {
    console.log(2);
 });
```

このPromiseチェーンは、次のコードのように毎回新しい変数に入れて処理をつなげるのと結果的には同じ意味となります。

```
// Promiseチェーンを変数に入れた場合
const firstPromise = Promise.resolve();
const secondPromise = firstPromise.then(() => {
  console.log(1);
});
const thridPromise = secondPromise.then(() => {
  console.log(2);
});
// それぞれ新しいPromiseインスタンスが作成される
console.log(firstPromise === secondPromise); // => false
console.log(secondPromise === thridPromise); // => false
```

もう少し具体的なPromiseチェーンの例を見ていきましょう。

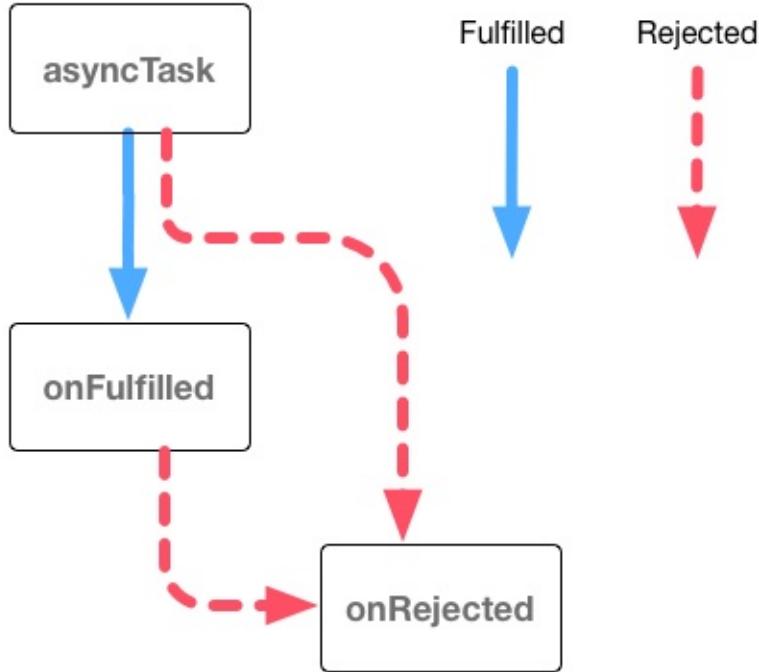
次のコードの `asyncTask` 関数はランダムで `Fulfilled` または `Rejected` 状態の `Promise` インスタンスを返します。この関数が返す `Promise` インスタンスに対して、`then` メソッドで成功時の処理を書いています。`then` メソッドの返り値は新しい `Promise` インスタンスであるため、続けて `catch` メソッドで失敗時の処理を書けます。

```
// ランダムでFulfilledまたはRejectedの`Promise`インスタンスを返す関数
function asyncTask() {
  return Math.random() > 0.5
    ? Promise.resolve("成功")
    : Promise.reject(new Error("失敗"));
}

// asyncTask関数は新しい`Promise`インスタンスを返す
asyncTask()
  .then(function onFulfilled(value) {
    console.log(value); // => "成功"
  })
  // catchメソッドは新しい`Promise`インスタンスを返す
  .catch(function onRejected(error) {
    console.log(error.message); // => "失敗"
  });
}
```

`asyncTask` 関数が成功 (`resolve`) した場合は `then` メソッドで登録した成功時の処理だけが呼び出され、`catch` メソッドで登録した失敗時の処理は呼び出されません。一方、`asyncTask` 関数が失敗 (`reject`) した場合は `then` メソッドで登録した成功時の処理は呼び出されずに、`catch` メソッドで登録した失敗時の処理だけが呼び出されます。

先ほどのコードにおけるPromiseの状態とコールバック関数は次のような処理の流れとなります。



Promiseの状態がRejectedとなった場合は、もっとも近い失敗時の処理( `catch` または `then` の第二引数)が呼び出されます。このとき間にある成功時の処理 ( `then` の第一引数) はスキップされます。

次のコードでは、RejectedのPromiseに対して `then -> then -> catch` とPromiseチェーンで処理を記述しています。このときもっとも近い失敗時の処理 ( `catch` ) が呼び出されますが、間にある2つの成功時の処理 ( `then` ) は実行されません。

```
// RejectedなPromiseは次の失敗時の処理までスキップする
const rejectedPromise = Promise.reject(new Error("失敗"));
rejectedPromise.then(() => {
  // このthenのコールバック関数は呼び出されません
}).then(() => {
  // このthenのコールバック関数は呼び出されません
}).catch(error => {
  console.log(error.message); // => "失敗"
});
```

Promiseのコンストラクタの処理の場合と同様に、`then` や `catch` のコールバック関数内で発生した例外は自動的にキャッチされます。例外が発生したとき、`then` や `catch` メソッドはRejectedなPromiseインスタンスを返します。そのため、例外が発生するとともっとも近くの失敗時の処理( `catch` または `then` の第二引数)が呼び出されます。

```
Promise.resolve().then(() => {
  // 例外が発生すると、thenメソッドはRejectedなPromiseを返す
  throw new Error("例外");
}).then(() => {
  // このthenのコールバック関数は呼び出されません
}).catch(error => {
  console.log(error.message); // => "例外"
});
```

また、Promiseチェーンで失敗を `catch` メソッドなどでキャッチすると、次に呼ばれるのは成功時の処理です。これは、`then` や `catch` メソッドはFulfilled状態のPromiseインスタンスを作成して返すためです。そのため、一度キャッチするとそこからはもとの `then` で登録した処理が呼ばれるPromiseチェーンに戻ります。

```
Promise.reject(new Error("エラー")).catch(error => {
```

```

    console.log(error); // Error: エラー
}).then(() => {
  console.log("thenのコールバック関数が呼び出される");
});

```

このように `Promise#then` メソッドや `Promise#catch` メソッドをつないで、成功時や失敗時の処理を書いていくことを Promiseチェーンと呼びます。

## Promiseチェーンで値を返す

Promiseチェーンではコールバックで返した値を次のコールバックへ引数として渡せます。

`then` や `catch` メソッドのコールバック関数は数値、文字列、オブジェクトなどの任意の値を返せます。このコールバック関数が返した値は、次の `then` のコールバック関数へ引数として渡されます。

```

Promise.resolve(1).then((value) => {
  console.log(value); // => 1
  return value * 2;
}).then(value => {
  console.log(value); // => 2
  return value * 2;
}).then(value => {
  console.log(value); // => 4
  // 値を返さない場合は undefined を返すのと同じ
}).then(value => {
  console.log(value); // => undefined
});

```

ここでは `then` メソッドを元に解説しますが、`catch` メソッドは `then` メソッドの糖衣構文であるため同じ動作となります。Promiseチェーンで一度キャッチすると、次に呼ばれるのは成功時の処理となります。そのため、`catch` メソッドで返した値は次の `then` メソッドのコールバック関数に引数として渡されます。

```

Promise.reject(new Error("失敗")).catch(error => {
  // 一度catchすれば、次に呼ばれるのは成功時のコールバック
  return 1;
}).then(value => {
  console.log(value); // => 1
  return value * 2;
}).then(value => {
  console.log(value); // => 2
});

```

## コールバック関数で `Promise` インスタンスを返す

Promiseチェーンで一度キャッチすると、次に呼ばれるのは成功時の処理(`then` メソッド)でした。これは、コールバック関数で任意の値を返すと、その値で `resolve` されたFulfilled状態の `Promise` インスタンスを作成するためです。しかし、コールバック関数で `Promise` インスタンスを返した場合は例外的に異なります。

コールバック関数で `Promise` インスタンスを返した場合は、同じ状態をもつ `Promise` インスタンスが `then` や `catch` メソッドの返り値となります。つまり `then` メソッドでRejected状態の `Promise` インスタンスを返した場合は、次に呼ばれるのは失敗時の処理となります。

次のコードでは、`then` メソッドのコールバック関数で `Promise.reject` メソッドを使いRejectedな `Promise` インスタンスを返しています。Rejectedな `Promise` インスタンスは、次の `catch` メソッドで登録した失敗時の処理を呼び出すまで、`then` メソッドの成功時の処理をスキップします。

```

Promise.resolve().then(function onFulfilledA() {
  return Promise.reject(new Error("失敗"));
}).then(function onFulfilledB() {

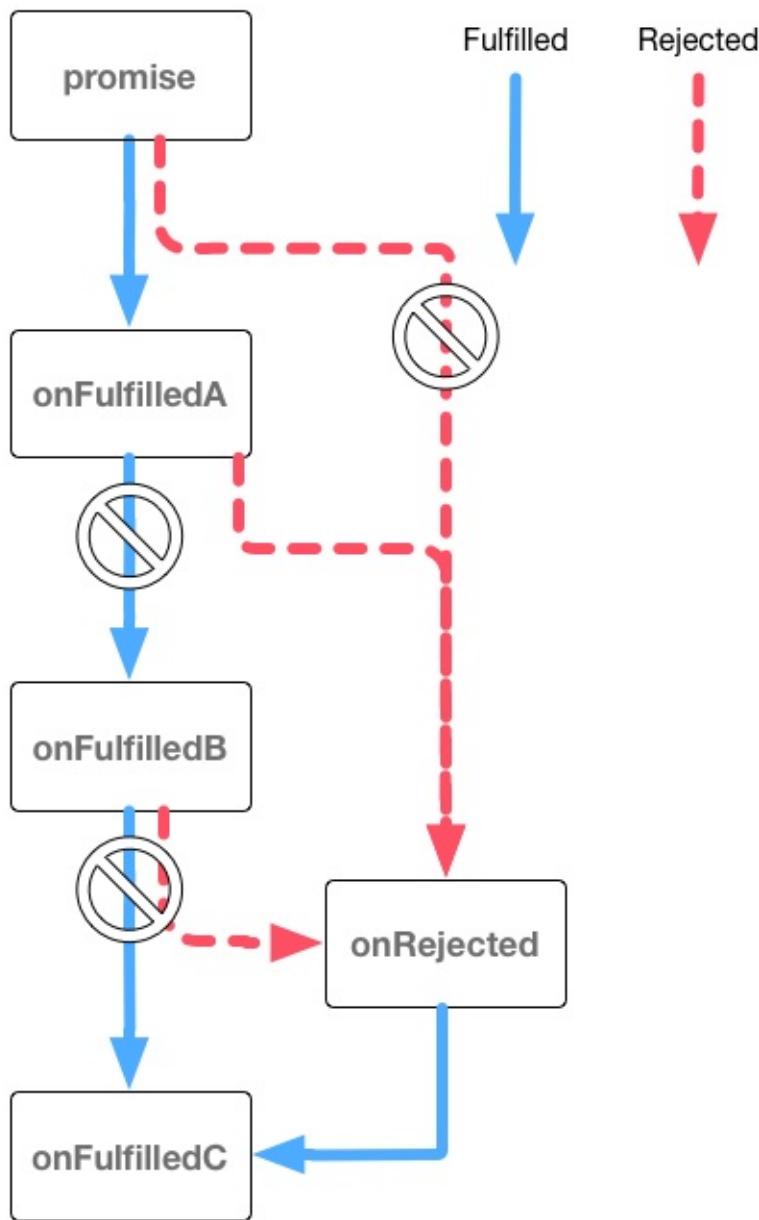
```

```

    console.log("onFulfilledBは呼び出されません");
}).catch(function onRejected(error) {
  console.log(error.message); // => "失敗"
}).then(function onFulfilledC() {
  console.log("onFulfilledCは呼び出されます");
});

```

このコードにおけるPromiseの状態とコールバック関数は次のような処理の流れとなります。



通常は一度 `catch` すると次に呼び出されるのは成功時の処理でした。この `Promise` インスタンスを返す仕組みを使うことで、`catch` してもそのまま `Rejected`な状態を継続するために利用できます。

次のコードでは `catch` メソッドでログを出力しつつ `Promise.reject` メソッドを使って `Rejected`な `Promise` インスタンスを返しています。これによって、`asyncFunction` で発生したエラーのログを取りながら、Promiseチェーンはエラーのまま処理を継続できます。

```

function asyncFunction() {
  return Promise.reject(new Error("エラー"));
}

```

```

function main() {
  return asyncFunction().catch(error => {
    // asyncFunctionで発生したエラーのログを出力する
    console.log(error);
    // Promiseチェーンはそのままエラーを継続させる
    return Promise.reject(error);
  });
}

// mainはRejectedなPromiseを返す
main().then(() => {
  console.log("この行は呼び出されません");
}).catch(error => {
  console.log("メインの処理が失敗した");
});

```

## Promiseチェーンの最後で処理を書く

`Promise#finally` メソッドは成功時、失敗時どちらの場合でも呼び出すコールバック関数を登録できます。  
`try...catch...finally` 構文の `finally` 節と同様の役割をもつメソッドです。

次のコードでは、リソースを取得して `then` で成功時の処理、`catch` で失敗時の登録しています。また、リソースを取得中かどうかを判定するためのフラグを `isLoading` という変数で管理しています。成功失敗どちらにもかかわらず、取得が終わったら `isLoading` は `false` にします。このとき `then` や `catch` それぞれの処理で `isLoading` へ `false` を代入もできますが、`Promise#finally` メソッドを使うことで一箇所にまとめられます。

```

function dummyFetch(path) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (path.startsWith("/resource")) {
        resolve({ body: `Response body of ${path}` });
      } else {
        reject(new Error("NOT FOUND"));
      }
    }, 1000 * Math.random());
  });
}

// リソースを取得中かどうかのフラグ
let isLoading = true;
dummyFetch("/resource/A").then(response => {
  console.log(response);
}).catch(error => {
  console.log(error);
}).finally(() => {
  isLoading = false;
  console.log("Promise#finally");
});

```

## Promiseチェーンで直列処理

Promiseチェーンで非同期処理の流れを書く大きなメリットは、非同期処理のさまざまなパターンに対応できることです。

ここでは、典型的な例として複数の非同期処理を順番に処理していく直列処理を考えていきましょう。Promiseで直列的な処理と言っても難しいことはなく、単純に `then` で非同期処理をつないでいくだけです。

次のコードでは、Resource A と Resource B を順番に取得しています。それぞれ取得したリソースを変数 `results` に追加し、すべて取得し終わったらコンソールに出力します。

```

function dummyFetch(path) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (path.startsWith("/resource")) {

```

```

        resolve({ body: `Response body of ${path}` });
    } else {
        reject(new Error("NOT FOUND"));
    }
}, 1000 * Math.random());
});
}

const results = [];
// Resource Aを取得する
dummyFetch("/resource/A").then(response => {
    results.push(response.body);
    // Resource Bを取得する
    return dummyFetch("/resource/B");
}).then(response => {
    results.push(response.body);
}).then(() => {
    console.log(results); // => ["Response body of /resource/A", "Response body of /resource/B"]
});

```

## Promise.all で複数のPromiseをまとめる

`Promise.all` を使うことで複数のPromiseを使った非同期処理をひとつのPromiseとして扱えます。

`Promise.all` メソッドは `Promise` インスタンスの配列を受け取り、新しい `Promise` インスタンスを返します。その配列のすべての `Promise` インスタンスが `Fulfilled` となった場合は、返り値の `Promise` インスタンスも `Fulfilled` となります。一方で、ひとつでも `Rejected` となった場合は、返り値の `Promise` インスタンスも `Rejected` となります。

返り値の `Promise` インスタンスに `then` メソッドで登録したコールバック関数には、`Promise` の結果をまとめた配列が渡されます。このときの配列の要素の順番は `Promise.all` メソッドに渡した配列の `Promise` の要素の順番と同じになります。

```

// `timeoutMs`ミリ秒後にresolveする
function delay(timeoutMs) {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve(timeoutMs);
        }, timeoutMs);
    });
}
const promise1 = delay(1);
const promise2 = delay(2);
const promise3 = delay(3);

Promise.all([promise1, promise2, promise3]).then(function(values) {
    console.log(values); // => [1, 2, 3]
});

```

先程の `Promise` チェーンでリソースを取得する例では、Resource Aを取得し終わってからResource Bを取得というように逐次的でした。しかし、Resource AとBどちらを先に取得しても問題ない場合は、`Promise.all` メソッドを使い2つの `Promise` を1つの `Promise` としてまとめられます。また、Resource AとBを同時に取得すればより早い時間で処理が完了します。

次のコードでは、Resource AとBを同時に取得開始しています。両方のリソースの取得が完了すると、`then` のコールバック関数にはAとBの結果が配列として渡されます。

```

function dummyFetch(path) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (path.startsWith("/resource")) {
                resolve({ body: `Response body of ${path}` });
            } else {

```

```

        reject(new Error("NOT FOUND"));
    }
}, 1000 * Math.random());
});
}

const fetchedPromise = Promise.all([
  dummyFetch("/resource/A"),
  dummyFetch("/resource/B")
]);
fetchedPromise.then(([responseA, responseB]) => {
  console.log(responseA.body); // => "Response body of /resource/A"
  console.log(responseB.body); // => "Response body of /resource/B"
});

```

渡したPromiseがひとつでもRejectedとなった場合は、失敗時の処理が呼び出されます。

```

function dummyFetch(path) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (path.startsWith("/resource")) {
        resolve({ body: `Response body of ${path}` });
      } else {
        reject(new Error("NOT FOUND"));
      }
    }, 1000 * Math.random());
  });
}

const fetchedPromise = Promise.all([
  dummyFetch("/resource/A"),
  dummyFetch("/not_found/B") // Bは存在しないため失敗する
]);
fetchedPromise.then(([responseA, responseB]) => {
  console.log("この行は呼び出されません");
}).catch(error => {
  console.log(error); // Error: NOT FOUND
});

```

## Promise.race

`Promise.all` メソッドは複数のPromiseがすべて完了するまで待つ処理でした。`Promise.race` メソッドでは複数のPromiseを受け取りますが、Promiseが1つでも完了した（Settle状態となった）時点で次の処理を実行します。

`Promise.race` メソッドは `Promise` インスタンスの配列を受け取り、新しい `Promise` インスタンスを返します。この新しい `Promise` インスタンスは配列のなかでも一番最初にSettle状態へとなった `Promise` インスタンスと同じ状態になります。

- 配列のなかでも一番最初にSettleとなったPromiseがFulfilledの場合は、新しい `Promise` インスタンスもFulfilledへ
  - 配列のなかでも一番最初にSettleとなったPromiseがRejectedの場合は、新しい `Promise` インスタンスもRejectedへ
- つまり、複数のPromiseによる非同期処理を同時に実行して競争（race）させて、一番最初に完了した `Promise` インスタンスに対する次の処理を呼び出します。

次のコードでは、`delay` 関数という `timeoutMs` ミリ秒後にFulfilledとなる `Promise` インスタンスを返す関数を定義しています。`Promise.race` メソッドは1ミリ秒、32ミリ秒、64ミリ秒、128ミリ秒後に完了する `Promise` インスタンスの配列を受け取っています。この配列の中でも一番最初に完了するのは、1ミリ秒後にFulfilledとなる `Promise` インスタンスです。

```

// `timeoutMs`ミリ秒後にresolveする
function delay(timeoutMs) {
  return new Promise((resolve) => {

```

```

        setTimeout(() => {
            resolve(timeoutMs);
        }, timeoutMs);
    });
}

// 一つでも resolve または reject した時点で次の処理を呼び出す
const racePromise = Promise.race([
    delay(1),
    delay(32),
    delay(64),
    delay(128)
]);
racePromise.then(value => {
    // もっとも早く完了するのは1ミリ秒
    console.log(value); // => 1
});

```

このときに、一番最初に `resolve` された値で `racePromise` も `resolve` されます。そのため、`then` メソッドのコールバック関数に `1` という値が渡されます。

他の `timeout` 関数が作成した `Promise` インスタンスも32ミリ秒、64ミリ秒、128ミリ秒後に `resolve` されます。しかし、`Promise` インスタンスは一度Settled（FulfilledまたはRejected）となると、それ以降は状態も変化せず `then` のコールバック関数も呼び出しません。そのため、`racePromise` は何度も `resolve` されますが、初回以外は無視されるため `then` のコールバック関数は一度しか呼び出されません。

`Promise.race` メソッドを使うことでPromiseを使った非同期処理のタイムアウトが実装できます。タイムアウトとは、一定時間経過しても処理が終わっていないならエラーとして扱う処理のことと言います。

次のコードでは `timeout` 関数と `dummyFetch` 関数が返す `Promise` インスタンスを `Promise.race` メソッドで競争させています。`dummyFetch` 関数のランダムな時間をかけてリソースを取得し `resolve` する `Promise` インスタンスを返します。`timeout` 関数は指定ミリ秒経過すると `reject` する `Promise` インスタンスを返します。

この2つの `Promise` インスタンスを競争させて、`dummyFetch` が先に完了すれば処理は成功、`timeout` が先に完了すれば処理は失敗というタイムアウト処理が実現できます。

```

// `timeoutMs`ミリ秒後にresolveする
function timeout(timeoutMs) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            reject(new Error(`Timeout: ${timeoutMs}ミリ秒経過`));
        }, timeoutMs);
    });
}

function dummyFetch(path) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (path.startsWith("/resource")) {
                resolve({ body: `Response body of ${path}` });
            } else {
                reject(new Error("NOT FOUND"));
            }
        }, 1000 * Math.random());
    });
}

// 500ミリ秒以内に取得できなければ失敗時の処理が呼ばれる
Promise.race([
    dummyFetch("/resource/data"),
    timeout(500),
]).then(response => {
    console.log(response.body); // => "Response body of /resource/data"
}).catch(error => {
    console.log(error.message); // => "Timeout: 500ミリ秒経過"
});

```

このようにPromiseを使うことで非同期処理のさまざまなパターンが形成できます。より詳しいPromiseの使い方については[JavaScript Promiseの本](#)というオンラインで公開されている文書にまとめられています。

一方でPromiseはただのビルトインオブジェクトであるため、非同期処理間の連携を行うにはPromiseチェーンのように少し特殊な書き方や見た目になります。また、エラーハンドリングについても `Promise#catch` メソッドや `Promise#finally` メソッドなど `try...catch` 構文とよく似た名前を使います。しかし、Promiseは構文ではなくただのオブジェクトであるため、それらをメソッドチェーンとして実現しないといけないといった制限があります。

ES2017ではこのPromiseチェーンの不格好な見た目を解決するためにAsync Functionと呼ばれる構文が導入されました。

## Async Function

Async Functionとは非同期処理を行う関数を定義する構文です。Async Functionは通常の関数とは異なり、必ず `Promise` インスタンスを返す関数を定義します。

Async Functionは次のように関数の前に `async` をつけることで定義できますが、この `doAsync` 関数は常に `Promise` インスタンスを返します。

```
async function doAsync() {
    return "値";
}
// doAsync関数はPromiseを返す
doAsync().then(value => {
    console.log(value); // => "値"
});
```

このAsync Functionは次のように書いた場合と同じ意味になります。Async Functionでは `return` した値の代わりに、`Promise.resolve(返り値)` のように返り値をラップした `Promise` インスタンスを返します。

```
// 通常の関数でPromiseインスタンスを返している
function doAsync() {
    return Promise.resolve("値");
}
doAsync().then(value => {
    console.log(value); // => "値"
});
```

重要なこととしてAsync FunctionはPromiseの上に作られた構文です。そのためAsync Functionを理解するには、Promiseを理解する必要があることに注意してください。

またAsync Function内では `await` 式というPromiseを使った非同期処理が完了するまで待つ構文が利用できます。`await` 式を使うことで非同期処理を同期処理のように扱えるため、Promiseチェーンで実現していた処理の流れを読みやすくかけます。

このセクションではAsync Functionと `await` 式について見てきます。

## Async Functionの定義

Async Functionは関数の定義に `async` キーワードをつけることで定義できます。JavaScriptの関数定義には関数宣言や関数式、Arrow Function、メソッドの短縮記法などがあります。どの定義方法でも `async` キーワードを前につけるだけでAsync Functionとして定義できます。

```
// 関数宣言のAsync Function版
async function fn1() {}
```

```
// 関数式のAsync Function版
const fn2 = async function() {};
// Arrow FunctionのAsync Function版
const foo = async() => {};
// メソッドの短縮記法のAsync Function版
const メソッド = { async foo() {} };
```

これらのAsync Functionは必ずPromiseを返すこととその関数の中では `await` 式が利用できる点以外は、通常の関数と同じ性質を持ちます。

## Async FunctionはPromiseを返す

Async Functionとして定義した関数は必ず `Promise` インスタンスを返します。具体的にはAsync Functionが返す値は次の3つのケースが考えられます。

1. Async Functionは値をreturnした場合、その返り値をもつFulfilledなPromiseを返します
2. Async FunctionがPromiseをreturnした場合、その返り値のPromiseをそのまま返します
3. Async Function内で例外が発生した場合は、そのエラーをもつRejectedなPromiseを返します

次のコードでは、Async Functionがそれぞれの返り値によってどのような `Promise` インスタンスを返すかを確認できます。この1から3の挙動は `Promise#then` メソッドの返り値とそのコールバック関数の関係とほぼ同じです。

```
// 1. resolveFnは値を返している
// 何もreturnしていない場合はundefinedを返したのと同じ扱いとなる
async function resolveFn() {
    return "返り値";
}
resolveFn().then(value => {
    console.log(value); // => "返り値"
});

// 2. rejectFnはPromiseインスタンスを返している
function rejectFn() {
    return Promise.reject(new Error("エラーメッセージ"));
}

// rejectFnはRejectedなPromiseを返すのでcatchできる
rejectFn().catch(error => {
    console.log(error.message); // => "エラーメッセージ"
});

// 3. exceptionFnは例外を投げている
async function exceptionFn() {
    throw new Error("例外が発生しました");
    console.log("この行は呼ばれません");
}

// Async Functionで例外が発生するとRejectedなPromiseが返される
exceptionFn().catch(error => {
    console.log(error.message); // => "例外が発生しました"
});
```

どの場合でもAsync Functionは必ずPromiseを返すことがわかります。このようにAsync Functionを呼び出す側から見れば、Async FunctionはPromiseを返すただの関数と何も変わりません。

## `await` 式

Async Functionの関数内では `await` 式を利用できます。`await` 式は右辺の `Promise` インスタンスが `Fulfilled` または `Rejected` になるまでその場で非同期処理の完了を待ちます。そして `Promise` インスタンスの状態が変わると、次の行の処理を再開します。

```
async function asyncMain() {
  // PromiseがFulfilledまたはRejectedとなるまで待つ
  await Promiseインスタンス;
  // Promiseインスタンスの状態が変わったら処理を再開する
}
```

普通の処理の流れでは非同期処理を実行した場合にその非同期処理の完了を待つことなく、次の行（次の文）を実行します。しかし `await` 式では非同期処理を実行し完了するまで、次の行（次の文）を実行しません。そのため `await` 式を使うことで非同期処理が同期処理のように上から下へと順番に実行するような処理順で書けます。

```
// async functionは必ずPromiseを返す
async function doAsync() {
  // 非同期処理
}

async function asyncMain() {
  // doAsyncの非同期処理が完了するまでまつ
  await doAsync();
  // 次の行はdoAsyncの非同期処理が完了されるまで実行されない
  console.log("この行は非同期処理が完了後に実行される");
}
```

`await` 式は式であるため右辺（`Promise` インスタンス）の評価結果を値として返します（式については「文と式」を参照）。この `await` 式の評価方法は評価する `Promise` の状態（`Fulfilled` または `Rejected`）によって異なります。

`await` 式の右辺の `Promise` が `Fulfilled` となった場合は、`resolve` された値が `await` 式の返り値となります。

次のコードでは、`await` 式の右辺にある `Promise` インスタンスは `42` という値で `resolve` されています。そのため `await` 式の返り値は `42` となり、`value` 変数にもその値が入ります。

```
async function asyncMain() {
  const value = await Promise.resolve(42);
  console.log(value); // => 42
}
asyncMain(); // Promiseインスタンスを返す
```

これは `Promise` を使って書くと次のコードと同様の意味となります。`await` 式を使うことでコールバック関数を使わずに非同期処理の流れを表現できていることがわかります。

```
function asyncMain() {
  return Promise.resolve(42).then(value => {
    console.log(value); // => 42
  });
}
asyncMain(); // Promiseインスタンスを返す
```

`await` 式の右辺の `Promise` が `Rejected` となった場合は、その場でエラーを `throw` します。また `Async Function` では関数内で発生した例外は自動的にキャッチされます。そのため `await` 式で `Promise` が `Rejected` となった場合は、その `Async Function` が `Rejected` な `Promise` を返すことになります。

次のコードでは、`await` 式の右辺にある `Promise` インスタンスが `Rejected` の状態になっています。そのため `await` 式は エラー を `throw` するため、`asyncMain` 関数は `Rejected` な `Promise` を返します。

```
async function asyncMain() {
  const value = await Promise.reject(new Error("エラーメッセージ"));
```

```

    console.log("この行は実行されません");
}
// Async Functionは自動的に例外をキャッチできる
asyncMain().catch(error => {
  console.log(error.message); // => "エラーメッセージ"
});

```

`await` 式がエラーを `throw` するということは、そのエラーは `try...catch` 構文でキャッチできます（詳細は「[try...catch構文](#)」の章を参照）。通常の非同期処理では完了する前に次の行が実行されてしまうため `try...catch` 構文ではエラーをキャッチできませんでした。そのためPromiseでは `catch` メソッドを使いPromise内で発生したエラーをキャッチしていました。

次のコードでは、`await` 式で発生した例外を `try...catch` 構文でキャッチしています。

```

async function asyncMain() {
  // await式のエラーはtry...catchできる
  try {
    const value = await Promise.reject(new Error("エラーメッセージ"));
    console.log("この行は実行されません");
  } catch (error) {
    console.log(error.message); // => "エラーメッセージ"
  }
}
asyncMain().catch(error => {
  console.log("この行は実行されません");
});

```

このように `await` 式を使うことで、`try...catch` 構文のように非同期処理を同期処理と同じ構文を使って扱えます。またコードの見た目も同期処理と同じように、その行（その文）の処理が完了するまで次の行を評価しないという分かりやすい形になるのは大きな利点です。

## await 式はAsync Functionの中でのみ利用可能

`await` 式はAsync Functionの直下でのみで利用可能です。Async Functionではない通常の関数で `await` 式を使うと Syntax Errorとなります。

```

function main(){
  // Syntax Error
  await Promise.resolve();
}

```

Async Functionは関数内で `await` を使っているのとは関係なく、必ず関数自体はPromiseを返します。そのAsync Function内で `await` 式を使って処理を待っている間も、関数の外側では通常通り処理が進んでいます。

```

async function asyncMain() {
  // 中でawaitしても、Async Functionの外側の処理まで止まるわけではない
  await new Promise((resolve) => {
    setTimeout(resolve, 16);
  });
}
console.log("1. asyncMain関数を呼び出します");
// async functionは外から見れば単なるpromiseを返す関数
asyncMain().then(() => {
  console.log("3. asyncMain関数が完了しました");
});
// async functionの外側の処理は次の行へ進む
console.log("2. asyncMain関数外では、次の行が同期的に呼び出される");

```

このように `await` 式を非同期処理を一時停止しても、`Async Function` 外の処理が停止するわけではありません。`Async Function` 外の処理も停止できてしまうと、JavaScriptでは基本的にメインスレッドで多くの処理をするためのUIを含めた他の処理が止まってしまいます。これが `await` 式が`Async Function`の範囲外で利用できない理由の一つです。

これと同じ理由で次のようなコールバック関数では `await` 式が利用できないことに注意してください。

次のコードでは `await` 式は `asynMain` 関数の直下ではなく、`forEach` メソッドのコールバック関数にかかりています。そのため`Async Function`の直下ではないため、次のコードはSyntax Errorとなります。

```
// コールバック関数で構文エラーとなる例
async function asynMain(){
  const promises = [Promise.resolve(1), Promise.resolve(2), Promise.resolve(3)];
  promises.forEach(promise => {
    // Syntax Error
    await promise;
  });
}
```

このコードを動作するように書くには次のようにコールバック関数に対して `async` キーワードをつける必要があります。

```
// 正しいAsync Functionとコールバック関数の書き方
async function asynMain() {
  const promises = [Promise.resolve(1), Promise.resolve(2), Promise.resolve(3)];
  promises.forEach(async promise => {
    await promise;
  });
}
```

## Promiseチェーンを `await` 式で表現する

`Async Function` と `await` 式を使うことでPromiseチェーンとして表現していた非同期処理を同期処理のような見た目でかけます。

たとえば、次のようなリソースAとリソースBを順番に取得する処理をPromiseチェーンで書くと次のようになります。

```
function dummyFetch(path) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (path.startsWith("/resource")) {
        resolve({ body: `Response body of ${path}` });
      } else {
        reject(new Error("NOT FOUND"));
      }
    }, 1000 * Math.random());
  });
}

// リソースAとリソースBを順番に取得する
function fetchResources() {
  const results = [];
  return dummyFetch("/resource/A").then(response => {
    results.push(response.body);
    return dummyFetch("/resource/B");
  }).then(response => {
    results.push(response.body);
  }).then(() => {
    return results;
  });
}

// リソースを取得して出力する
```

```
fetchResources().then((results) => {
  console.log(results); // => ["Response body of /resource/A", "Response body of /resource/B"]
});
```

このコードと同じ処理を Async Function と `await` 式で書くと次のように書けます。

```
function dummyFetch(path) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (path.startsWith("/resource")) {
        resolve({ body: `Response body of ${path}` });
      } else {
        reject(new Error("NOT FOUND"));
      }
    }, 1000 * Math.random());
  });
}

// リソースAとリソースBを順番に取得する
async function fetchResources() {
  const results = [];
  const responseA = await dummyFetch("/resource/A");
  results.push(responseA.body);
  const responseB = await dummyFetch("/resource/B");
  results.push(responseB.body);
  return results;
}

// リソースを取得して出力する
fetchResources().then((results) => {
  console.log(results); // => ["Response body of /resource/A", "Response body of /resource/B"]
});
```

Promise チェーンで `fetchResources` 関数書いた場合はコールバックの中で処理を行うためややこしい見た目になります。一方、Async Function と `await` 式で書いた場合は、取得と追加を順番に行うだけとなりネストがなく見た目はシンプルです。

このように Async Function と `await` 式でも非同期処理を同期処理のような見た目で書けます。一方で同期処理のような見た目となるため、複数の非同期処理を順番に行うようなケースでは無駄な待ち時間を作ってしまうコードを書きやすいです。

先ほど `fetchResources` 関数ではリソースAを取得し終わってからリソースBを取得していました。特に取得順が関係無い場合はリソースAとリソースBを同時に取得できます。

Promise チェーンでは `Promise.all` メソッドを使い、リソースAとリソースBを取得する非同期処理を1つの `Promise` インスタンスにまとめることで同時に取得していました。`await` 式が評価するのは `Promise` インスタンスであるため、`await` 式は `Promise.all` メソッドなど `Promise` インスタンスを返す処理と組み合わせて利用できます。

そのため、先ほど `fetchResources` 関数でリソースを同時に取得する場合は、次のように書けます。`Promise.all` メソッドは複数の `Promise` を配列で受け取り、それを1つの `Promise` としてまとめたものを返す関数です。`Promise.all` メソッドの返す `Promise` インスタンスを `await` することで、非同期処理の結果を配列としてまとめて取得できます。

```
function dummyFetch(path) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (path.startsWith("/resource")) {
        resolve({ body: `Response body of ${path}` });
      } else {
        reject(new Error("NOT FOUND"));
      }
    }, 1000 * Math.random());
  });
}

// リソースAとリソースBを同時に取得する
```

```

async function fetchResources() {
  // Promise.allは [ResponseA, ResponseB] のように結果を配列にしたPromiseインスタンスを返す
  const responses = await Promise.all([
    dummyFetch("/resource/A"),
    dummyFetch("/resource/B")
  ]);
  return responses.map(response => {
    return response.body;
  });
}
// リソースを取得して出力する
fetchResources().then((results) => {
  console.log(results); // => ["Response body of /resource/A", "Response body of /resource/B"]
});

```

このようにAsync Functionや `await` 式は既存のPromiseと組み合わせて利用できます。 Async Functionも内部的に Promiseの仕組みを利用しているため、両者は対立関係ではなく共存関係です。

## コールバック関数とAsync Function

Async Functionと `await` 式はPromiseチェーンに比べてコードを読みやすくしますが、すべてのケースでそうとはいえない。ここでは `await` 式とコールバック関数の組み合わせにおいての直感的ではない動作を紹介します。

次のコードは  `AsyncStorage` という擬似的に非同期で読み書きするストレージクラスを使う例です。`main` 関数では `saveUsers` 関数でユーザーデータを保存し、保存が完了後にユーザーデータが読み出せるかをチェックしています。

```

class AsyncStorage {
  constructor() {
    this.dataMap = new Map();
  }
  async save(key, value) {
    return new Promise(resolve => {
      setTimeout(() => {
        this.dataMap.set(key, value);
        resolve();
      }, 100);
    });
  }
  async load(key) {
    return new Promise(resolve => {
      setTimeout(() => {
        resolve(this.dataMap.get(key));
      }, 50);
    });
  }
}
// Async Storageを作成する
const storage = new AsyncStorage();
// 1. AsyncStorageにデータを保存する
async function saveUsers(users) {
  users.forEach(async(user) => {
    await storage.save(user.id, user);
  });
}
// 2. AsyncStorageからデータを読み取る
async function loadUser(userId) {
  return storage.load(userId);
}
async function main() {
  const users = [{ id: 1, name: "John" }, { id: 5, name: "Smith" }, { id: 7, name: "Ayo" }];
  await saveUsers(users);
  // idが5のユーザーデータを取り出す
  const user = await loadUser(5);
}

```

```
// しかしあとで保存が完了していないためundefinedとなる
console.log(user); // => undefined
}
main();
```

このコードは `users` (ユーザーデータ) をストレージに保存し、保存が完了後にすぐ読み出せることを意図しています。しかし、`saveUsers` 関数を呼び出し後に `loadUser` でユーザーデータを読み出しても `undefined` が返されます。これは保存が完了する前に、ユーザーデータを読み取ろうとしてしまったため空の値である `undefined` が返されています。

なぜ意図したように動いていないかというと `saveUsers` 関数の実装に問題があるためです。

`saveUsers` 関数を詳しく見ていきます。`forEach` メソッドのコールバック関数として Async Function を渡しています。Async Function の中で `await` 式を利用して非同期処理の完了を待っています。しかし、この非同期処理の完了を待つの Async Function の中だけで、外側では `save` メソッドの完了を待つことなく進みます。

次のように `saveUsers` 関数にコンソール出力を入れてみると動作が分かりやすいでしょう。`forEach` メソッドのコールバック関数が完了するのは、`saveUsers` 関数の呼び出しがすべて終わった後になります。そのため `await` 式で `saveUsers` 関数の完了を待ったつもりでも、その時点では Storage に値が保存されていません。

```
async function saveUsers(users) {
  console.log("1. saveUsers関数開始");
  users.forEach(async(user) => {
    // 非同期処理が完了するまで待つ
    await storage.save(user.id, user);
    console.log(`3. UserId:${user.id}を保存しました`);
  });
  console.log("2. saveUsers関数終了");
}
```

この問題を修正する方法はいくつかありますが、ここでは2種類の方法を見ていきます。

1つめの方法は、Async Function をコールバック関数に利用しない方法です。次のコードのように `forEach` メソッドではなく for ループを利用すれば、特別な工夫をせずにユーザーデータを保存できます。

```
async function saveUsers(users) {
  console.log("1. saveUsers関数開始");
  for (let i = 0; i < user.length; i++) {
    const user = users[i];
    // コールバック関数ではないので、`saveUsers` 関数の処理もここで一時停止する
    await storage.save(user.id, user);
    console.log(`2. UserId:${user.id}を保存しました`);
  };
  console.log("3. saveUsers関数終了");
}
```

2つめの方法は、Async Function を使ったコールバック関数の結果の Promise がすべて完了するのを待つ方法です。Async Function はそれぞれ Promise を返すため、すべての Promise の完了を明示的に待てばよいはずです。複数の Promise の完了を待つには `Promise.all` メソッドで 1 つの Promise にまとめて `await` 式でその Promise の完了を待てばよいだけです。

次のコードは `Array#forEach` メソッドではなく、`Array#map` メソッドを使いコールバック関数の結果を集めています。その集めた Promise を `Promise.all` メソッドで 1 つの Promise にして、`await` 式で完了するまで待つだけです。

```
async function saveUsers(users) {
  console.log("1. saveUsers関数開始");
  const promises = users.map(async user => {
    await storage.save(user.id, user);
    console.log(`2. UserId:${user.id}を保存しました`);
  });
}
```

```
// すべての保存処理のPromiseを完了を待つ
await Promise.all(promises);
console.log("3. saveUsers関数終了");
}
```

`AsyncStorage#save` メソッドはもともと Promise を返すため、次のように Async Function をコールバック関数にしなくても動作は同じです。

```
async function saveUsers(users) {
  const promises = users.map(user => {
    return storage.save(user.id, user);
  });
  await Promise.all(promises);
}
```

最後にもともとのコードの `saveUsers` 関数の修正し動作を確認してみます。

次のように `saveUsers` 関数の問題を修正することで、`saveUsers` 関数の完了する前にユーザーデータがストレージに保存できます。`await saveUsers(ユーザーデータ)` で完了を待つことで、次の行でストレージからデータを取り出したときに意図したようにデータを取得できます。

```
class AsyncStorage {
  constructor() {
    this.dataMap = new Map();
  }
  async save(key, value) {
    return new Promise(resolve => {
      setTimeout(() => {
        this.dataMap.set(key, value);
        resolve();
      }, 100);
    });
  }
  async load(key) {
    return new Promise(resolve => {
      setTimeout(() => {
        resolve(this.dataMap.get(key));
      }, 50);
    });
  }
}
// Async Storageを作成する
const storage = new AsyncStorage();
// 1. AsyncStorageにデータを保存する
async function saveUsers(users) {
  // Storege#saveはそれぞれPromiseを返すためAsync Functionをコールバックにしなくても良い
  const promises = users.map(user => storage.save(user.id, user));
  await Promise.all(promises);
  return; // 返り値は明示的になしにしているため、undefinedでresolveされるPromiseを返す
}
// 2. AsyncStorageからデータを読み取る
async function loadUser(userId) {
  return storage.load(userId);
}
async function main() {
  const users = [{ id: 1, name: "John" }, { id: 5, name: "Smith" }, { id: 7, name: "Ayo" }];
  await saveUsers(users);
  // idが5のユーザーデータを取り出す
  const user = await loadUser(5);
  console.log(user); // => { id: 5, name: "Smith" }
}
main();
```

Async Functionは非同期処理のコードフローを分かりやすくしますが、コールバック関数に利用した際には分かりにくくする場合もあります。そのため、無理してすべてをAsync Functionで書かずにPromiseの仕組みそのものを利用することも重要です。 Async FunctionはPromiseの上に作られた仕組みであるため、両者と一緒に利用することも考えてみてください。

# Map/Set

JavaScriptでデータの集まりを扱うコレクションは配列だけではありません。この章では、マップ型のコレクションである `Map` と、セット型のコレクションである `Set` について学びます。

## Map

`Map` はマップ型のコレクションを扱うためのビルトインオブジェクトです。マップとは、キーと値の組み合せからなる抽象データ型です。他のプログラミング言語の文脈では辞書やハッシュマップ、連想配列などと呼ばれることもあります。

### マップの作成と初期化

`Map` オブジェクトを `new` することで、新しいマップを作ることができます。作成されたばかりのマップは何ももつていません。そのため、マップのサイズを返す `size` プロパティは `0` を返します。

```
const map = new Map();
console.log(map.size); // => 0
```

`Map` オブジェクトを `new` で初期化するときに、コンストラクタに初期値を渡すことができます。コンストラクタ引数として渡すことができるはエントリーの配列です。エントリーとは、ひとつのキーと値の組み合せを `[キー, 値]` という形式の配列で表現したものです。そのため、次の例のように配列の配列を渡すことになります。

```
const map = new Map([[{"key1": "value1"}, {"key2": "value2"}]]);
// 2つのエントリーで初期化されている
console.log(map.size); // => 2
```

### 要素の追加と取り出し

`Map` には新しい要素を追加したり、追加した要素を取り出したりするためのメソッドがあります。`set` メソッドは特定のキーと値をもつ要素をマップに追加します。ただし、同じキーで複数回 `set` メソッドを呼び出した際は後から追加された値で上書きされます。

`get` メソッドは特定のキーに紐付いた値を取り出します。また、特定のキーに紐付いた値をもっているかどうかを確認する `has` メソッドがあります。

```
const map = new Map();
// 新しい要素の追加
map.set("key", "value1");
console.log(map.size); // => 1
console.log(map.get("key")); // => "value1"
// 要素の上書き
map.set("key", "value2");
console.log(map.get("key")); // => "value2"
// キーの存在確認
console.log(map.has("key")); // => true
console.log(map.has("foo")); // => false
```

`delete` メソッドは追加した要素を削除します。`delete` メソッドに渡されたキーと、そのキーに紐付いた値がマップから削除されます。また、マップがもつすべての要素を削除するための `clear` メソッドがあります。

```
const map = new Map();
```

```
map.set("key1", "value1");
map.set("key2", "value2");
console.log(map.size); // => 2
map.delete("key1");
console.log(map.size); // => 1
map.clear();
console.log(map.size); // => 0
```

## マップの反復処理

マップがもつ要素を列挙するメソッドとして、`forEach`、`keys`、`values`、`entries`があります。

`forEach` メソッドはマップがもつすべての要素を、マップへの追加順に反復します。コールバック関数には引数として値、キー、マップの3つが渡されます。配列の`forEach` メソッドと似ていますが、インデックスの代わりにキーが渡されます。配列は順序により要素を特定しますが、マップはキーにより要素を特定するためです。

```
const map = new Map([["key1", "value1"], ["key2", "value2"]]);
const results = [];
map.forEach((value, key) => {
  results.push(` ${key}: ${value}`);
});
console.log(results); // => ["key1: value1", "key2: value2"]
```

`keys` メソッドはマップがもつすべての要素のキーを挿入順に並べたIteratorオブジェクトを返します。同様に、`values` メソッドはマップがもつすべての要素の値を挿入順に並べたIteratorオブジェクトを返します。これらの戻り値はIteratorオブジェクトであって配列ではありません。そのため次の例のように、`for...of`文で反復処理をおこなったり、`Array.from` メソッドに渡して配列に変換して使ったりします。

```
const map = new Map([["key1", "value1"], ["key2", "value2"]]);
const keys = [];
// keysメソッドの戻り値を反復する
for (const key of map.keys()) {
  keys.push(key);
}
console.log(keys); // => ["key1", "key2"]
// keysメソッドの戻り値から配列を作る
const keysArray = Array.from(map.keys());
console.log(keysArray); // => ["key1", "key2"]
```

`entries` メソッドはマップがもつすべての要素をエントリーとして挿入順に並べたIteratorオブジェクトを返します。先述のとおりエントリーは`[キー, 値]`のような配列です。そのため、配列の分割代入を使うとエントリーからキーと値を簡潔に取り出せます。

```
const map = new Map([["key1", "value1"], ["key2", "value2"]]);
const entries = [];
for (const [key, value] of map.entries()) {
  entries.push(` ${key}: ${value}`);
}
console.log(entries); // => ["key1: value1", "key2: value2"]
```

また、マップ自身も`iterable`なオブジェクトなので、`for...of`文で反復できます。マップを`for...of`文で反復したときは、すべての要素をエントリーとして挿入順に反復します。つまり、`entries` メソッドの戻り値を反復するときと同じ結果が得られます。

```
const map = new Map([["key1", "value1"], ["key2", "value2"]]);
const results = [];
for (const [key, value] of map) {
  results.push(` ${key}: ${value}`);
}
```

```
console.log(results); // => ["key1:value1", "key2:value2"]
```

## マップとしてのObjectとMap

ES2015で `Map` が導入されるまで、JavaScriptにおいてマップ型を実現するために `object` が利用されてきました。何かをキーにして値にアクセスするという点で、`Map` と `Object` はよく似ています。ただし、マップとしての `Object` にはいくつかの問題があります。

- `Object` のプロトタイプから継承されたプロパティによって、意図しないマッピングを生じる危険性があります
- また、プロパティとしてデータをもつため、キーとして使えるのは文字列か `Symbol` に限られます

`Object` にはプロトタイプがあるため、いくつかのプロパティは初期化されたときから存在します。`Object` をマップとして使うと、そのプロパティと同じ名前のキーを使おうとしたときに問題があります。

たとえば `constructor` という文字列は `Object.prototype.constructor` プロパティと衝突してしまいます。そのため `constructor` のような文字列をオブジェクトのキーに使うことで意図しないマッピングを生じる危険性があります。この問題はマップとして使う `Object` のインスタンスを `Object.create(null)` のように初期化して作ることで回避されてきました。

```
const map = {};
// マップがキーをもつことを確認する
function has(key) {
  return typeof map[key] !== "undefined";
}
console.log(has("foo")); // => false
// Objectのプロパティが存在する
console.log(has("constructor")); // => true
```

これらの問題を解決するために `Map` が導入されました。`Map` はプロパティとは異なる仕組みでデータを格納します。そのため、`Map` のプロトタイプがもつメソッドやプロパティとキーが衝突することはありません。また、`Map` はマップのキーとしてあらゆるオブジェクトを使うことができます。

他にも `Map` には次のような利点があります。

- マップのサイズを簡単に知ることができます
- マップがもつ要素を簡単に列挙できる
- オブジェクトをキーにすると参照ごとに違うマッピングができる

たとえばショッピングカートのような仕組みを作るとき、次のように `Map` を使って商品のオブジェクトと注文数をマッピングできます。

```
// ショッピングカートを表現するクラス
class ShoppingCart {
  constructor() {
    // 商品とその数をもつマップ
    this.items = new Map();
  }
  // カートに商品を追加する
  addItem(item) {
    const count = this.items.get(item) || 0;
    this.items.set(item, count + 1);
  }
  // カート内の合計金額を返す
  getTotalPrice() {
    return Array.from(this.items).reduce((total, [item, count]) => {
      return total + item.price * count;
    }, 0);
  }
  // カートの中身を文字列にして返す
  toString() {
```

```

        return Array.from(this.items).map(([item, count]) => {
            return `${item.name}:${count}`;
        }).join(",");
    }
}

const shoppingCart = new ShoppingCart();
// 商品一覧
const shopItems = [
    { name: "みかん", price: 100 },
    { name: "りんご", price: 200 },
];

// カートに商品を追加する
shoppingCart.addItem(shopItems[0]);
shoppingCart.addItem(shopItems[0]);
shoppingCart.addItem(shopItems[1]);

// 合計金額を表示する
console.log(shoppingCart.getTotalPrice()); // => 400
// カートの中身を表示する
console.log(shoppingCart.toString()); // => "みかん:2,りんご:1"

```

`Object` をマップとして使うときに起きる多くの問題は、`Map` オブジェクトを使うことで解決しますが、常に `Map` が `Object` の代わりになるわけではありません。マップとしての `Object` には次のような利点があります。

- リテラル表現があるため作成しやすい
- 規定のJSON表現があるため、`JSON.stringify` 関数を使ってJSONに変換するのが簡単である
- ネイティブAPI・外部ライブラリを問わず、多くの関数がマップとして `Object` を渡される設計になっている

次の例では、ログインフォームのsubmitイベントを受け取ったあと、サーバーにPOSTリクエストを送信しています。サーバーにJSON文字列を送るために、`JSON.stringify` 関数を使います。そのため、`Object` のマップを作つてフォームの入力内容をもたせています。このような簡易なマップにおいては、`Object` を使うほうが適切でしょう。

```

// URLとObjectのマップを受け取ってPOSTリクエストを送る関数
function sendPOSTRequest(url, data) {
    // XMLHttpRequestを使ってPOSTリクエストを送る
    const httpRequest = new XMLHttpRequest();
    httpRequest.setRequestHeader("Content-Type", "application/json");
    httpRequest.send(JSON.stringify(data));
    httpRequest.open("POST", url);
}

// formのsubmitイベントを受け取る関数
function onLoginFormSubmit(event) {
    const form = event.target;
    const data = {
        userName: form.elements.userName,
        password: form.elements.password,
    };
    sendPOSTRequest("/api/login", data);
}

```

## WeakMap

`WeakMap`は、`Map`と同じくマップを扱うためのビルトインオブジェクトです。`Map`と違う点は、キーを弱い参照(Weak Reference)でもつことです。

`弱い参照`とは、ガベージコレクタによるオブジェクトの解放を妨げないための特殊な参照です。あるオブジェクトへの参照がすべて弱い参照のとき、そのオブジェクトはいつでもガベージコレクタによって解放できます。弱い参照は、不要になったオブジェクトを参照し続けて発生するメモリリークを防ぐために使われます。`WeakMap`では不要になったキーとそれに紐付いた値が自動的に削除されるため、メモリリークを引き起こす心配がありません。

`WeakMap` は `Map` と似ていますが `iterable` ではありません。そのため、キーを列挙する `keys` メソッドや、データの数を返す `size` プロパティなどは存在しません。また、キーを弱い参照でもつ特性上、キーとして使えるのは参照型のオブジェクトだけです。

`WeakMap` の主な使い方のひとつは、あるオブジェクトに紐付くオブジェクトを管理することです。たとえば次の例では、オブジェクトが発火するイベントのリスナー関数（イベントリスナー）をマップで管理しています。イベントリスナーとは、イベントが発生したときに呼び出される関数のことです。このマップを `Map` で実装してしまうと、`targetObj` がマップから削除されるまでイベントリスナーはメモリ上に残り続けます。ここで `WeakMap` を使うと、`addListener` 関数に渡された `listener` は `targetObj` が解放された際、自動的に解放されます。

```
// イベントリスナーを管理するマップ
const listenersMap = new WeakMap();

// 渡されたオブジェクトに紐付くイベントリスナーを追加する
function addListener(targetObj, listener) {
    const listeners = listenersMap.get(targetObj) || [];
    listenersMap.set(targetObj, listeners.concat(listener));
}

// 渡されたオブジェクトに紐付くイベントリスナーを呼び出す
function triggerListeners(targetObj) {
    if (listenersMap.has(targetObj)) {
        listenersMap.get(targetObj)
            .forEach((listener) => listener());
    }
}

// 上記関数の実行例

let eventTarget = {};
// イベントリスナーを追加する
addListener(eventTarget, () => {
    console.log("イベントが発火しました");
});
// eventTargetに紐付いたイベントリスナーが呼び出される
triggerListeners(eventTarget);
// eventTargetの参照が変われば自動的にイベントリスナーが解放される
eventTarget = null;
```

また、あるオブジェクトから計算した結果を保存する用途でもよく使われます。次の例ではHTML要素の高さを計算した結果を保存して、2回目以降に同じ計算をしないようにしています。

```
const cache = new WeakMap();

function getHeight(element) {
    if (cache.has(element)) {
        return cache.get(element);
    }
    const height = element.getBoundingClientRect().height;
    // elementオブジェクトに対して高さを紐付けて保存している
    cache.set(element, height);
    return height;
}
```

## [コラム] キーの等価性とNaNオブジェクト

`Map` に値をセットする際のキーにはあらゆるオブジェクトが使えますが、一部のオブジェクトについては扱いに注意が必要です。

マップが特定のキーをすでにもっているか、つまり挿入と上書きの判定は基本的に `==` 演算子と同じです。ただし `NaN` オブジェクトの扱いだけが例外的に違います。`Map` におけるキーの比較では、`NaN` 同士は常に等価であるとみなされます。この挙動は [Same-value-zero](#) アルゴリズムと呼ばれます。

```
const map = new Map();
map.set(NaN, "value");
// NaNは==で比較した場合は常にfalse
console.log(NaN === NaN); // => false
// MapはNaN同士を比較できる
console.log(map.get(NaN)); // => "value"
```

## Set

`Set`はセット型のコレクションを扱うためのビルトインオブジェクトです。セットとは、重複する値がないことを保証したコレクションのことをいいます。`Set`は追加した値を列挙できるので、値が重複しないことを保証する配列のようなものとしてよく使われます。ただし、配列と違って要素は順序をもたず、インデックスによるアクセスはできません。

### セットの作成と初期化

`Set`オブジェクトを `new` することで、新しいセットを作ることができます。作成されたばかりのセットは何ももつていません。そのため、セットのサイズを返す `size` プロパティは0を返します。

```
const set = new Set();
console.log(set.size); // => 0
```

`Set`オブジェクトを `new` で初期化するときに、コンストラクタに初期値を渡すことができます。コンストラクタ引数として渡すことができる的是可iterableオブジェクトです。次の例ではiterableオブジェクトである配列を初期値として渡しています。

```
const set = new Set(["value1", "value2", "value2"]);
// "value2"が重複しているので、セットのサイズは2になる
console.log(set.size); // => 2
```

### 値の追加と取り出し

作成したセットに値を追加するには `add` メソッドを使います。先述のとおり、セットは重複する値をもたないことが保証されます。そのため、すでにセットがもっている値を `add` メソッドに渡した際は無視されます。

また、セットが特定の値をもっているかどうかを確認する `has` メソッドがあります。

```
const set = new Set();
// 値の追加
set.add("a");
console.log(set.size); // => 1
// 重複する値は追加されない
set.add("a");
console.log(set.size); // => 1
// 値の存在確認
console.log(set.has("a")); // => true
console.log(set.has("b")); // => false
```

セットから値を削除するには、`delete` メソッドを使います。`delete` メソッドに渡された値がセットから削除されます。また、セットがもつすべての値を削除するための `clear` メソッドがあります。

```
const set = new Set();
set.add("a");
set.add("b");
```

```

console.log(set.size); // => 2
set.delete("a");
console.log(set.size); // => 1
set.clear();
console.log(set.size); // => 0

```

## セットの反復処理

セットがもつすべての値を反復するにはfor...of文を使います。for...of文でセットを反復したときは、セットへの追加順に値が取り出されます。

```

const set = new Set();
set.add("a");
set.add("b");
const results = [];
for (const value of set) {
  results.push(value);
}
console.log(results); // => ["a", "b"]

```

セットがもつ要素を列挙するメソッドとして、`forEach`、`keys`、`values`、`entries`があります。これらは`Map`との類似性のために存在しますが、セットにはマップにおけるキー相当のものはありません。そのため、`keys`メソッドは`values`メソッドのエイリアスになっており、セットがもつすべての値を挿入順に列挙するIteratorオブジェクトを返します。また、`entries`メソッドは`[値, 値]`という形のエントリーを挿入順に列挙するIteratorオブジェクトを返します。ただし、`Set`自身が`iterable`であるため、これらのメソッドが必要になることはないでしょう。

## WeakSet

`WeakSet`は弱い参照で値をもつセットです。`WeakSet`は`Set`と似ていますが、`iterable`ではないので追加した値を反復できません。つまり、`WeakSet`は値の追加と削除、存在確認以外のことができません。データの格納ではなく、データの一意性を確認することに特化したセットといえるでしょう。

また、弱い参照で値をもつ特性上、値として使えるのは参照型のオブジェクトだけです。

# JSON

## JSONとは

JSONはJavaScript Object Notationの略で、JavaScriptのオブジェクト表記をベースに作られた軽量なデータフォーマットです。JSONの仕様はECMA-404によって標準化されています。人間にとて読み書きが容易で、マシンにとっても簡単にパースや生成を行なえる形式になっているため、多くのプログラミング言語がJSONを扱う機能を備えています。

JSONはJavaScriptのオブジェクトリテラル、配列リテラル、各種プリミティブ型の値を組み合わせたものです。ただしJSONとJavaScriptは一部の構文に違いがあります。たとえばJSONでは、オブジェクトリテラルのキーを必ずダブルクオートで囲まなければいけません。また、小数点から書き始める数値リテラルや、先頭がゼロから始まる数値リテラルも使えません。これらは機械がパースしやすくするために仕様で定められた制約です。

```
{
  "object": {
    "number": 1,
    "string": "js-primer",
    "boolean": true,
    "null": null,
    "array": [1, 2, 3]
  }
}
```

JSONの細かい仕様に関しては[json.orgの日本語ドキュメント](#)にわかりやすくまとまっているので、参考にするとよいでしょう。

## JSON オブジェクト

JavaScriptでJSONを扱うには、ビルトインの[JSONオブジェクト](#)を利用します。`JSON`オブジェクトはグローバルオブジェクトなので、どのスコープからでもアクセスできます。`JSON`オブジェクトはJSON形式の文字列とJavaScriptのオブジェクトを相互に変換するための`parse`メソッドと `stringify`メソッドを提供します。

### JSON文字列をオブジェクトに変換する

`JSON.parse`メソッドは引数に与えられた文字列をJSONとしてパースし、その結果をJavaScriptのオブジェクトとして返す関数です。次のコードは簡単なJSON形式の文字列をJavaScriptのオブジェクトに変換する例です。

```
// JSONはダブルクオートのみを許容するため、シングルクオートでJSON文字列を記述
const json = '{ "id": 1, "name": "js-primer" }';
const obj = JSON.parse(json);
console.log(obj.id); // => 1
console.log(obj.name); // => "js-primer"
```

文字列がJSONの配列を表す場合は、`JSON.parse`メソッドの返り値も配列になります。

```
const json = "[1, 2, 3]";
console.log(JSON.parse(json)); // => [1, 2, 3]
```

与えられた文字列がJSON形式でパースできない場合は例外が投げられます。また、実際のアプリケーションでJSONを扱うのは、外部のプログラムとデータを交換する用途がほとんどです。外部のプログラムが送ってくるデータが常にJSONとして正しい保証はありません。そのため、`JSON.parse` メソッドは基本的にtry-catch文で例外処理をするべきです。

```
const userInput = "not json value";
try {
  const json = JSON.parse(userInput);
} catch (error) {
  console.log("パースできませんでした");
}
```

## オブジェクトをJSON文字列に変換する

`JSON.stringify` メソッドは第1引数に与えられたオブジェクトをJSON形式の文字列に変換して返す関数です。HTTP通信でサーバーにデータを送信するときや、アプリケーションが保持している状態を外部に保存するときなどに必要になります。次のコードはJavaScriptのオブジェクトをJSON形式の文字列に変換する例です。

```
const obj = { id: 1, name: "js-primer", bio: null };
console.log(JSON.stringify(obj)); // => '{"id":1,"name":"js-primer","bio":null}'
```

`JSON.stringify` メソッドにはオプショナルな引数が2つあります。第2引数はreplacer引数とも呼ばれ、変換後のJSONに含まれるプロパティ関数あるいは配列を渡せます。関数を渡した場合は引数にプロパティのキーと値が渡され、その返り値によって文字列に変換される際の挙動をコントロールできます。次の例は値がnullであるプロパティを除外してJSONに変換するreplacer引数の例です。replacer引数の関数でundefinedが返されたプロパティは、変換後のJSONに含まれなくなります。

```
const obj = { id: 1, name: "js-primer", bio: null };
const replacer = (key, value) => {
  if (value === null) {
    return undefined;
  }
  return value;
};
console.log(JSON.stringify(obj, replacer)); // => '{"id":1,"name":"js-primer"}'
```

replacer引数に配列を渡した場合はプロパティのホワイトリストとして使われ、その配列に含まれる名前のプロパティだけが変換されます。

```
const obj = { id: 1, name: "js-primer", bio: null };
const replacer = ["id", "name"];
console.log(JSON.stringify(obj, replacer)); // => '{"id":1,"name":"js-primer"}'
```

第3引数はspace引数とも呼ばれ、変換後のJSON形式の文字列を読みやすくフォーマットする際のインデントを設定できます。数値を渡すとその数値分の長さのスペースで、文字列を渡すとその文字列でインデントされます。次のコードはスペース2個でインデントされたJSONを得る例です。

```
const obj = { id: 1, name: "js-primer" };
// replacer引数を使わない場合はnullを渡して省略するのが一般的です
console.log(JSON.stringify(obj, null, 2));
/*
{
  "id": 1,
  "name": "js-primer"
}
*/
```

また、次のコードはタブ文字でインデントされたJSONを得る例です。

```
const obj = { id: 1, name: "js-primer" };
console.log(JSON.stringify(obj, null, "\t"));
/*
{
  "id": 1,
  "name": "js-primer"
}
*/
```

## [コラム] JSONにシリアル化できないオブジェクト

`JSON.stringify` メソッドはJSONで表現可能な値だけをシリアル化します。そのため、値が関数や `symbol`、あるいは `undefined` であるプロパティなどは変換されません。ただし、配列の値としてそれらが見つかったときには例外的に `null` に置き換えられます。またキーが `symbol` である場合にもシリアル化の対象外になります。代表的な変換の例を次の表とサンプルコードに示します。

シリアル化前の値	シリアル化後の値
文字列・数値・真偽値	対応する値
<code>null</code>	<code>null</code>
配列	配列
オブジェクト	オブジェクト
関数	変換されない（配列のときは <code>null</code> ）
<code>undefined</code>	変換されない（配列のときは <code>null</code> ）
<code>Symbol</code>	変換されない（配列のときは <code>null</code> ）
<code>RegExp</code>	<code>{}</code>
<code>Map</code> , <code>Set</code>	<code>{}</code>

```
// 値が関数のプロパティ
console.log(JSON.stringify({ x: function() {} })); // => '{}'
// 値がSymbolのプロパティ
console.log(JSON.stringify({ x: Symbol("") })); // => '{}'
// 値がundefinedのプロパティ
console.log(JSON.stringify({ x: undefined })); // => '{}'
// 配列の場合
console.log(JSON.stringify({ x: [10, function() {}] })); // => '{"x":[10,null]}'
// キーがSymbolのプロパティ
JSON.stringify({ [Symbol("foo")]: "foo" }); // => '{}'
// 値がRegExpのプロパティ
console.log(JSON.stringify({ x: /foo/ })); // => '{"x":{}}'
// 値がMapのプロパティ
const map = new Map();
map.set("foo", "foo");
console.log(JSON.stringify({ x: map })); // => '{"x":{}}'
```

オブジェクトがシリアル化される際は、そのオブジェクトの列挙可能なプロパティだけが再帰的にシリアル化されます。`RegExp`や`Map`、`Set`などのインスタンスは列挙可能なプロパティを持たないため、空のオブジェクトに変換されます。

また、`JSON.stringify` メソッドがシリアル化に失敗することもあります。よくあるのは、参照が循環しているオブジェクトをシリアル化しようとしたときに例外が投げられるケースです。たとえば次の例のように、あるオブジェクトのプロパティを再帰的に辿って自分自身が見つかるような場合はシリアル化が不可能となります。

`JSON.parse` メソッドだけでなく、`JSON.stringify` メソッドも例外処理をおこなって安全に使いましょう。

```
const obj = { foo: "foo" };
obj.self = obj;
try {
  JSON.stringify(obj);
} catch (error) {
  console.log(error); // => "TypeError: Converting circular structure to JSON"
}
```

## [コラム] `toJSON` メソッドを使ったシリアル化

オブジェクトが`toJSON` メソッドを持っている場合、`JSON.stringify` メソッドは既定の文字列変換ではなく`toJSON` メソッドの返り値を使います。次の例のように、引数に直接渡されたときだけでなく引数のプロパティとして登場したときにも再帰的に処理されます。

```
const obj = {
  foo: "foo",
  toJSON() {
    return "bar";
  }
};
console.log(JSON.stringify(obj)); // => '"bar"'
console.log(JSON.stringify({ x: obj })); // => '{"x":"bar"}'
```

`toJSON` メソッドは自作のクラスを特殊な形式でシリアル化する目的などに使われます。

# Date

この章では、JavaScriptで日付や時刻を扱うための[Date](#)について学びます。

## Dateオブジェクト

`Date` オブジェクトは `String` や `Array` などと同じく、ビルトインのグローバルオブジェクトです。そのため、スクリプト中のどこからでも呼び出して使えます。

`Date` オブジェクトをインスタンス化することで、ある特定の時刻を表すオブジェクトが得られます。`Date` における「時刻」は、UTC（協定世界時）の1970年1月1日0時0分0秒を基準とした相対的なミリ秒として保持されます。このミリ秒の値のことを、本章では「時刻値」と呼びます。`Date` オブジェクトのインスタンスはそれぞれがひとつの時刻値をもち、その時刻値をもとに日付や時・分などを扱うメソッドを提供します。

### インスタンスの作成

`Date` オブジェクトのインスタンスは、常に`new`演算子を使って作成します。`Date` オブジェクトのインスタンス作成には、大きく分けて2つの種類があります。ひとつは現在の時刻をインスタンス化するもの、もうひとつは任意の時刻をインスタンス化するものです。

#### 現在の時刻をインスタンス化する

`Date` を`new`するときにコンストラクタ引数を何も渡さない場合、作成されるインスタンスは現在の時刻を表すものになります。`Date` オブジェクトのインスタンスではなく現在の時刻の時刻値だけが欲しい場合には、`Date.now` メソッドの戻り値を使います。作成したインスタンスがもつ時刻値は、`getTime` メソッドで取得できます。また、`toISOString` メソッドを使うと、その時刻をUTCにおけるISO 8601形式の文字列に変換できます。ISO 8601とは国際規格となっている文字列の形式で、`2006-01-02T15:04:05.999+09:00` のように時刻を表現します。人間が見ても分かりやすい文字列であるため、広く利用されています。

```
// 現在の時刻を表すインスタンスを作成する
const now = new Date();
// 時刻値だけが欲しい場合にはDate.nowメソッドを使う
console.log(Date.now());

// 時刻値を取得する
console.log(now.getTime());
// 時刻をISO 8601形式の文字列で表示する
console.log(now.toISOString());
```

#### 任意の時刻をインスタンス化する

コンストラクタ引数を渡すことで、任意の時刻を表すインスタンスを作成できます。`Date` のコンストラクタ関数はオーバーロードされており、渡す引数によって時刻の指定方法が変わります。オーバーロードは次の3種類があります。

- 時刻値を渡すもの
- 時刻を示す文字列を渡すもの
- 時刻の部分（年・月・日など）をそれぞれ数値で渡すもの

1つめは、コンストラクタ関数にミリ秒を表す数値型の引数を渡したときに適用されます。渡した数値をUTCの1970年1月1日0時0分0秒を基準とした時刻値として扱います。この方法は実行環境による挙動の違いが起きないので安全です。また、時刻値を直接指定するので、他2つの方法と違ってタイムゾーンを考慮する必要がありません。

```
// 時刻のミリ秒値を直接指定する形式
// 1136214245999はUTCにおける"2006年1月2日15時04分05秒999"を表す
const date = new Date(1136214245999);
// 末尾の'Z'はUTCであることを表す
console.log(date.toISOString()); // => "2006-01-02T15:04:05.999Z"
```

2つめは文字列型の引数を渡したときに適用されます。RFC2822やISO 8601の形式にしたがった文字列を渡すと、その文字列をパースして得られる時刻値を使って、`Date` のインスタンスを作成します。

次のコードでは、ISO 8601形式の文字列を渡して`Date` のインスタンスを作成します。タイムゾーンを含む文字列の場合は、そのタイムゾーンにおける時刻として時刻値を計算します。文字列からタイムゾーンが読み取れない場合は、実行環境のタイムゾーンによって時刻値を計算するため注意が必要です。また、ISO 8601形式以外の文字列のパースは、ブラウザごとに異なる結果を返す可能性があるため注意しましょう。

```
// UTCにおける"2006年1月2日15時04分05秒999"を表すISO 8601形式の文字列
const inUTC = new Date("2006-01-02T15:04:05.999Z");
console.log(inUTC.toISOString()); // => "2006-01-02T15:04:05.999Z"

// 上記の例とは異なり、UTCであることを表す'Z'がついていないことに注意
// Asia/Tokyo(+09:00)で実行すると、UTCにおける表記は9時間前の06時04分05秒になる
const inLocal = new Date("2006-01-02T15:04:05.999");
console.log(inLocal.toISOString()); // "2006-01-02T06:04:05.999Z" (Asia/Tokyoの場合)
```

3つめは、時刻を次のように、年・月・日などの部分ごとの数値で指定する方法です。

```
new Date(year, month, day, hour, minutes, seconds, milliseconds);
```

コンストラクタ関数に2つ以上の引数を渡すと、このオーバーロードが適用されます。日を表す第3引数から後の引数は省略可能ですが、日付だけはデフォルトで1が設定され、その他は0が設定されます。また、月を表す第2引数は0から11までの数値で指定することにも注意しましょう。

先述した2つの方法と違い、この方法はタイムゾーンを指定できません。渡した数値は常にローカルのタイムゾーンにおける時刻とみなされます。結果が実行環境に依存してしまうため、基本的にこの方法は使うべきではありません。時刻を部分ごとに指定したい場合は、`Date.UTC`メソッドを使うとよいでしょう。渡す引数の形式は同じですが、`Date.UTC` メソッドは渡された数値をUTCにおける時刻として扱い、その時刻値を返します。

```
// 実行環境における"2006年1月2日15時04分05秒999"を表す
// タイムゾーンを指定することはできない
const date1 = new Date(2006, 0, 2, 15, 4, 5, 999);
console.log(date1.toISOString()); // "2006-01-02T06:04:05.999Z" (Asia/Tokyoの場合)

// Date.UTCメソッドを使うとUTCに固定できる
const ms = Date.UTC(2006, 0, 2, 15, 4, 5, 999);
// 時刻値を渡すコンストラクタと併用する
const date2 = new Date(ms);
console.log(date2.toISOString()); // => "2006-01-02T15:04:05.999Z"
```

なお、どのオーバーロードにもあてはまらない引数や、時刻としてパースできない文字列を渡した際にも、`Date` のインスタンスは作成されます。ただし、このインスタンスがもつ時刻は不正であるため、`getTime` メソッドは`Nan`を返し、`toString` メソッドは`Invalid Date`という文字列を返します。

```
// 不正なDateインスタンスを作成する
const invalid = new Date("");
console.log(invalid.getTime()); // => NaN
console.log(invalid.toString()); // => "Invalid Date"
```

## Dateのインスタンスマソッド

`Date` オブジェクトのインスタンスは多くのメソッドをもっていますが、ほとんどは `getHours` と `setHours` のような、時刻の各部分を取得・更新するためのメソッドです。

次の例は、日付を決まった形式の文字列に変換しています。`getMonth` メソッドや `setMonth` メソッドのように月を数値で扱うメソッドは、0から11の数値で指定することに注意しましょう。ある `Date` のインスタンスの時刻が何月かを表示するには、`getMonth` メソッドの戻り値に1を足す必要があります。

```
// YYYY/MM/DD形式の文字列に変換する関数
function formatDate(date) {
  const yyyy = new String(date.getFullYear());
  // String#padStartメソッド (ES2017) で2桁になるように0埋めする
  const mm = new String(date.getMonth() + 1).padStart(2, "0");
  const dd = new String(date.getDate()).padStart(2, "0");
  return `${yyyy}/${mm}/${dd}`;
}

const date = new Date("2006-01-02T15:04:05.999");
console.log(formatDate(date)); // => "2006/01/02"
```

`getTimezoneOffset` メソッドは、実行環境のタイムゾーンのUTCからのオフセット値を分単位の数値で返します。たとえばAsia/TokyoタイムゾーンはUTC+9時間なのでオフセット値は-9時間となり、`getTimezoneOffset` メソッドの戻り値は `-540` です。

```
// getTimezoneOffsetはインスタンスマソッドなので、インスタンスが必要
const now = new Date();
// 時間単位にしたタイムゾーンオフセット
const timezoneOffsetInHours = now.getTimezoneOffset() / 60;
// UTCの現在の時間を計算できる
console.log(`Hours in UTC: ${now.getHours() + timezoneOffsetInHours}`);
```

## 現実のユースケースとDate

ここまで `Date` オブジェクトとインスタンスマソッドについて述べましたが、多くのユースケースにおいては機能が不十分です。たとえば次のような場合に、`Date` では直感的に記述できません。

- 任意の書式の文字列から時刻に変換するメソッドがない
- 「時刻を1時間進める」のように時刻を前後にずらす操作を提供するメソッドがない
- 任意のタイムゾーンにおける時刻を計算するメソッドがない
- `YYYY/MM/DD` のようなフォーマットに基づいた文字列への変換を提供するメソッドがない

そのため、JavaScriptにおける日付・時刻の処理は、標準の`Date`ではなくライブラリを使うことが一般的になっています。代表的なライブラリとしては、`moment.js`や`js-joda`、`date-fns`などがあります。

```
// moment.jsで現在時刻のmomentオブジェクトを作る
const now = moment();
// addメソッドで10分進める
const future = now.add(10, "minutes");
// formatメソッドで任意の書式の文字列に変換する
console.log(future.format("YYYY/MM/DD"));
```

Date

---

# Math

この章では、JavaScriptで数学的な定数と関数を提供する組み込みのオブジェクトである[Math](#)について学びます。

## Mathオブジェクト

`Math` オブジェクトはビルトインのグローバルオブジェクトですが、コンストラクタではありません。つまり `Math` オブジェクトはインスタンスを作らず、すべての定数や関数は `Math` オブジェクトの静的なプロパティやメソッドとして提供されています。たとえば、`Math.PI` プロパティは円周率πをあらわす定数であり、`Math.sin` メソッドはラジアン値から正弦を計算する関数です。次の例では、90度における正弦を計算しています。90度の正弦は1なので、`sin90` 変数は1を返します。

```
const rad90 = Math.PI * 90 / 180;
const sin90 = Math.sin(rad90);
console.log(sin90); // => 1
```

三角関数をはじめとした多くの関数や定数が `Math` オブジェクトから提供されています。この章ではそれらのうちよく使われるものについてユースケースを交えて紹介します。網羅的な解説については[MDNのリファレンス](#)を参照してください。

### 乱数を生成する

`Math` オブジェクトの主な用途のひとつは、`Math.random` メソッドによる乱数の生成です。`Math.random` メソッドは、0以上1未満の範囲内で、疑似ランダムな浮動小数点数を返します。乱数生成のシードには現在時刻が使われます。

```
for (let i = 0; i < 5; i++) {
  // 毎回ランダムな浮動小数点数を返す
  console.log(Math.random());
}
```

次の例では、`Math.random` メソッドを使って、任意の範囲で乱数を生成しています。

```
// minからmaxまでの乱数を返す関数
function getRandom(min, max) {
  return Math.random() * (max - min) + min;
}
// 1以上5未満の浮動小数点数を返す
console.log(getRandom(1, 5));
```

### 数値の大小を比較する

`Math.max` メソッドは引数として渡された複数の数値のうち、最大のものを返します。同様に、`Math.min` メソッドは引数として渡された複数の数値のうち、最小のものを返します。

```
console.log(Math.max(1, 10)); // => 10
console.log(Math.min(1, 10)); // => 1
```

これらのメソッドは可変長の引数を取るため、任意の個数の数値を比較できます。数値の配列の中から最大・最小の値を取り出す際には、`...` (spread構文) を使うと簡潔に記述できます。

```
const numbers = [1, 2, 3, 4, 5];
```

```
console.log(Math.max(...numbers)); // => 5
console.log(Math.min(...numbers)); // => 1
```

# ECMAScript

ここまでJavaScriptの基本文法について見てきましたが、その文法を定めるECMAScriptという仕様自体がどのように変化していくのかを見ていきましょう。

ECMAScriptはEcma Internationalという団体によって標準化されている仕様です。Ecma InternationalはECMAScript以外にもC#やDartなどの標準化作業を行っています。Ecma International中のTechnical Committee 39（TC39）という技術委員会が中心となって、ECMAScript仕様についてを議論しています。この技術委員会はMicrosoft、Mozilla、Google、AppleといったブラウザベンダーやECMAScriptに関心のある企業などによって構成されます。

## ECMAScriptのバージョンの歴史

ここで、簡単にECMAScriptのバージョンの歴史を振り返ってみましょう。

バージョン	リリース時期
1	1997年6月
2	1998年6月
3	1999年12月
4	破棄 <a href="#">ES4</a>
5	2009年12月
5.1	2011年6月
2015	2015年6月
2016	2016年6月
2017	2017年6月

ES5.1からES2015ができるまで4年もの歳月がかかっているのに対して、ES2015以降は毎年リリースされています。毎年安定したリリースを行えるようになったのは、ES2015以降は仕様策定プロセスの変更が行われたためです。

## Living StandardとなるECMAScript

現在、ECMAScriptの仕様書のドラフトはGitHub上の[tc39/ecma262](#)で管理されており日々更新されています。そのため、本当の意味での最新のECMAScript仕様は<https://tc39.github.io/ecma262/>となります。このように更新ごとにバージョン番号を付けずに、常に最新版を公開する仕様のことをLiving Standardと呼びます。

ECMAScriptはLiving Standardですが、これに加えてECMAScript 2017のようにバージョン番号をつけたものも公開されています。このバージョン付きECMAScriptは、毎年決まった時期のドラフトを元にしたスナップショットのようなものです。

ブラウザなどに実際にJavaScriptとして実装される際には、Living StandardのECMAScriptを参照しています。これは、ブラウザ自体も日々更新されるものであり、決まった時期にしかリリースされないバージョン付きよりもLiving Standardの方が適当であるためです。

## 仕様策定のプロセス

ES2015以前はすべての仕様の合意が取れるまで延々と議論を続けすべてが決まってからリリースされていました。そのため、ES2015がリリースされるまでには6年もの歳月がかかり言語の進化が停滞していました。この問題を解消するために、TC39は毎年リリースするためにECMAScriptの策定プロセスを変更しました。

この策定プロセスはES2015がリリース後に適応され、このプロセスで初めてリリースされたのがES2016となります。ES2016以降では、次のような仕様策定のプロセスで議論を進めて仕様が決定されています。[process](#)

仕様に追加する機能（API、構文など）をそれぞれ個別のプロポーザル（提案書）として進めていきます。現在策定中のプロポーザルはGitHub上の[tc39/proposals](#)に一覧が公開されています。それぞれのプロポーザルは責任者であるチャンピオンとステージ（Stage）と呼ばれる0から4の5段階の状態を持ちます。

ステージ	ステージの概要
0	アイデアの段階
1	機能提案の段階
2	機能の仕様書ドラフトを作成した段階
3	仕様としては完成しており、ブラウザの実装やフィードバックを求める段階
4	仕様策定が完了し、2つ以上の実装が存在している。 正式にECMAScriptにマージできる段階

2ヶ月に一度行われるTC39のミーティングにおいて、プロポーザルごとにステージを進めるかどうかを議論します。このミーティングの議事録もGitHub上の[tc39/tc39-notes](#)にて公開されています。ステージ4となったプロポーザルはドラフト版である[tc39/ecma262](#)へマージされます。そして毎年の決まった時期にドラフト版を元にして [ECMAScript 20xx](#) としてリリースします。

この仕様策定プロセスの変更は、ECMAScriptに含まれる機能の形にも影響しています。

たとえば、`class`構文の策定は最大限に最小のクラス（maximally minimal classes）と呼ばれる形で提案されています。これによりES2015で`class`構文が導入されました。しかし、クラスとして合意が取れる最低限の機能だけの状態で入りました。その他のクラスの機能は別のプロポーザルとして提案され、ES2015以降に持ち越された形で議論が進められています。

このような合意が取れる最低限の形でプロポーザルを進めていくのには、ES4の苦い失敗が背景にあります。ES4ではECMAScriptに多くの変更を入れることを試みましたが、TC39内でも意見が分かれ最終的に合意できませんでした。これによりES4の策定に割いた数年分のリソースが無駄となってしまったという経緯があります。<sup>1</sup>

ES2016以降の策定プロセスでも、すべてのプロポーザルが仕様に入るわけではありません。<sup>2</sup>別の代替プロポーザルが出た場合や後方互換性と保てない場合などにプロポーザルの策定を中断する場合があります。しかし、この場合でもプロポーザルという単位であるため策定作業の無駄は最小限で済みます。このようにモジュール化されたプロポーザルは入れ替えがし易いという性質もあります。

## プロポーザルの機能を試す

ECMAScriptの策定プロセスのステージ4に「2つ以上の実装が存在している」という項目があります。そのためブラウザのJavaScriptエンジンには、策定中のプロポーザルが実装されている場合があります。多くの場合は試験的なフラグ付きで実装されておりフラグを有効化することで、試すことができるようになっています。

またTranspilerやPolyfillといった手段で、プロポーザルの機能をエミュレートできる場合があります。

Transpilerとは、新しい構文を既存の機能で再現できるようにソースコードを変換するツールのことです。たとえば、ES2015で`class`構文が導入されたが、ES5では`class`は予約語であるため構文エラーとなり実行することはできません。Transpilerでは、`class`構文を含むソースコードを`function`キーワードを使い擬似的に再現するコードへ変換します。TranspilerとしてはBabelやTypeScriptなどが有名です。

Polyfillとは、新しい関数やメソッドなどの仕様を満たすような実装を提供するライブラリのことです。たとえば、ES2016では `Array#includes` というメソッドが追加されました。構文とは異なり `Array#includes` のようなメソッドはビルトインオブジェクトを書き換えることで実装できます。Polyfillを提供するものとしては[core-js](#)や[polyfill.io](#)などがあります。

注意点としてはTranspilerやPolyfillはあくまで既存の機能で新しい機能を再現を試みているだけに過ぎません。そのため、既存の機能で再現ができないプロポーザル（機能）はTranspilerやPolyfillでは再現できません。また、完全な再現はできていないことがあるためTranspilerやPolyfillを新しい機能を学ぶために使うべきではありません。

## 仕様や策定プロセスを知る意味

こうしたECMAScriptという仕様や策定プロセスを知る意味は何があるのでしょうか？主に次のような理由で知る意味があると考えています。

- 言語を学ぶため
- 言語が進化しているため
- 情報の正しい状態を調べるため

### 言語を学ぶため

もっとも単純な理由はJavaScriptという言語そのものを学ぶためです。言語の詳細を知りたい場合にはECMAScriptという仕様を参照できます。

しかしながら、JavaScriptにおいては言語機能に関しては[MDN Web Docs](#)という優れたリファレンスサイトなどがあります。そのため、使い方を覚えたいなどの範囲ではECMAScriptの仕様そのものを参照する機会は少ないでしょう。

### 言語が進化しているため

ECMAScriptはLiving Standardであり、日々更新されています。これは、言語仕様に新しい機能や修正などが常に行われていることを表しています。

ECMAScriptは後方互換性を尊重するため、今学んでいることが無駄になるわけではありません。しかしながら言語自体も進化していることは意識しておくとよいでしょう。

ECMAScriptのプロポーザル（機能）は問題を解決するために提案されます。そのプロポーザルがECMAScriptにマージされ利用できる場合、その機能が何を解決するために導入されたのか知ることは大切です。その際には、ECMAScriptの策定プロセスを知っておくことが調べることに役立ちます。

この仕様はなぜこうなったのかということを知りたいと思ったときに、その機能がどのような経緯で入ったのかを調べる手段をもつことは大切です。特にES2015以降は策定プロセスもGitHubを利用したオープンなものとなり、過去の記録なども探しやすくなっています。

### 情報の正しい状態を調べるため

JavaScriptは幅広く使われている言語であるため、世の中には膨大な情報があります。そして、検索して見つかる情報には正しいものや間違ったものが混在しています。

その中においてECMAScriptの仕様やその策定中のプロポーザルに関する情報は状態が明確です。基本的にECMAScriptの仕様に入ったものは、後方互換性を維持するために破壊的変更は殆ど行なえません。プロポーザルはステージという明示された状態があり、ステージ4未満の場合はまだ安定していないことが分かります。

そのため、問題を見つけた際に該当する仕様やプロポーザルを確認してみることは重要です。

これはECMAScriptにかぎらず、ウェブやブラウザに関する情報に関しては同じことがいえます。ブラウザ関してはHTML、DOM API、CSSなどもオープンな仕様とそれぞれ策定プロセスが存在しています。

## まとめ

JavaScriptと一言にいってもECMAScript、ブラウザ、Node.js、WebAssembly、 WebGL、WebRTCなど幅広い分野があります。そのためすべてのことを知っている必要はありませんし、知っている人もおそらくいないでしょう。このような状況下においては知識そのものよりも、それについて知りたいと思ったときに調べる方法を持っていることが大切です。

なにごとも突然全く新しい概念が増えるわけではなく、ものごとには過程が存在します。 ECMAScriptにおいては策定プロセスという形でどのような段階であるかが公開されています。つまり、仕様にいきなり新しい機能が増えるのではなくプロポーザルという段階を踏んでいます。

日々変化しているソフトウェアにおいては、自身に適切な調べ方をもつことが大切です。

1. ES2015の仕様編集者であるAllen Wirfs-Brock氏の書いた[Programming Language Standardization](#)に詳細が書かれています。 ↵
2. [Inactive Proposals](#)に策定を中止したプロポーザルの一覧が公開されています。 ↵
- process. この策定プロセスは<https://tc39.github.io/process-document/>に詳細が書かれています。 ↵
- ES4. ECMAScript 4は複雑で大きな変更が含まれており、合意を得ることできずに仕様策定が破棄されました。 ↵

## 第一部: おわりに

第一部 基本文法では、ECMAScriptという仕様の範囲でのJavaScriptの文法や使い方について見てきました。第二部 ユースケースでは、第一部で学んだ基本文法を応用した小さなアプリケーションを実装しながらJavaScriptについて理解を深めていきます。また第二部からはNode.jsやブラウザといった実行環境の固有なAPIの利用やライブラリの利用方法についても見ていきます。

一方で、第一部ではECMAScriptのすべての文法やビルトインオブジェクトを紹介したわけではありません。紹介していない文法やビルトインオブジェクトにもとても有用なものが数多くあります。

たとえば[Proxy](#)や[Reflect](#)といったビルトインオブジェクトは、オブジェクトの基本的な操作（プロパティの取得や代入など）に対して独自の動作を定義できます。また、ビルトインオブジェクトの `Object` にも[Object.defineProperty](#)メソッドという、オブジェクトの記述子（descriptor）を変更できるものがあります。オブジェクトの記述子（descriptor）を変更することで、オブジェクトのプロパティを変更できなくなるとといったオブジェクトのメタ的な動作を設定できます。

その他にもありますが、これらのAPIはアプリケーションよりもライブラリを作成するに利用することが多いです。また第二部のユースケースでも登場しないため、この書籍では省略させていただきました。これらのAPIは必要となつた際に使い方を調べて覚えていくのがよいです。

JavaScriptのほとんどのAPIについては何度も登場している[MDN Web Docs](#)というリファレンスに大部分が記載されています。MDNにはECMAScriptの機能だけではなく、ブラウザ固有の機能であるDOM APIと呼ばれるものについても含まれています。そのため、MDNは実質的にJavaScriptの公式リファレンスと考えられます。

第二部ではNode.jsやブラウザ固有のDOM APIについても触れていきます。これらの実行環境に依存するAPIはかなりの数が存在するため、APIの調べ方を知ることが重要です。

たとえば、Node.jsには公式のリファレンスガイドとして[Node.js Documentation](#)があります。ブラウザならば先ほども紹介した[MDN Web Docs](#)が実質的な公式のリファレンスガイドです。まずは使い方を知るために公式のリファレンスガイドを参照してみてください。

また利用しているライブラリやツールの使い方について調べる場合には、そのライブラリなどの公式サイトやリポジトリを見てみることが大切です。これは「[ECMAScriptの章](#)」でも紹介していますが、ECMAScriptやJavaScriptは常に変化しています。そのため、ライブラリによってはいつのまにかDeprecated（非推奨）となっている場合もあるため、まずは元となるものをみることが重要です。

調べ方に正解はありません。しかし、調べたいと思ったときに調べることができるよう、調べ方を知っておくことが重要です。

## 第二部: ユースケース

基本文法で学んだことを応用し、具体的なユースケースを元に学んでいきます。

# アプリケーション開発の準備

これまでに学んだJavaScriptの基本構文は、実行環境を問わずに使えるものです。しかしこの後に続くユースケースの章では、具体的な実行環境としてWebブラウザとNode.jsの2つを扱います。また、ブラウザで実行するアプリケーションであっても、その開発にはツールとしてのNode.jsが欠かせません。このセクションではユースケースの学習へ進むために必要なアプリケーション開発環境の準備をおこないます。

## Node.jsのインストール

Node.jsはサーバーサイドJavaScript実行環境のひとつで、次のような特徴があります。

- WebブラウザのChromeと同じV8 JavaScriptエンジンで動作する
- オープンソースで開発されている
- OSを問わずクロスプラットフォームで動作する

Node.jsはサーバーサイドで使うために開発されました。しかし今ではコマンドラインツールやElectronなどのデスクトップアプリケーションにも利用されています。そのため、Node.jsはサーバーサイドに限らずクライアントサイドのJavaScript実行環境としても幅広く使われています。

Node.jsは多くの他のプログラミング言語と同じように、実行環境をマシンにインストールすることで使用できます。公式の[ダウンロードページ](#)から、開発用のマシンに合わせたインストーラをダウンロードして、インストールしましょう。

Node.jsにはLTS (Long-Term Support) 版と最新版の2つのリリース版があります。LTS (Long-Term Support) 版は2年間のメンテナンスとサポートが宣言されたバージョンです。具体的には、後方互換性を壊さない範囲でのアップデートと、継続的なセキュリティパッチの提供が行われます。一方で、最新版はNode.jsの最新の機能を使用できますが、常に最新のバージョンしかメンテナンスされません。ほとんどのユーザーは、LTS版を用いることが推奨されます。Node.jsでの開発が初めてであれば、迷わずLTS版のインストーラをダウンロードしましょう。この章では執筆時点の最新LTS版であるバージョン8.12系で動作するように開発します。

インストールが完了すると、コマンドラインで node コマンドが使用可能になっているはずです。次のコマンドを実行して、インストールされたNode.jsのバージョンを確認しましょう。

```
$ node -v  
v8.12.0
```

また、Node.jsにはnpmというパッケージマネージャーが同梱されています。Node.jsをインストールすると、node コマンドだけでなくnpmを使うための npm コマンドも使えるようになっています。次のコマンドを実行して、インストールされたnpmのバージョンを確認しましょう。

```
$ npm -v  
6.4.1
```

npmや npm コマンドについての詳細は[公式ドキュメント](#)や[npmのGitHubリポジトリ](#)を参照してください。Node.jsのライブラリのほとんどはnpmを使ってインストールできます。実際に、ユースケースの章ではnpmを使ってライブラリをインストールして利用します。

## npxコマンドによるnpmパッケージの実行

Node.jsを使ったコマンドラインツールは数多く公開されており、npmでインストールすることによりコマンドとして実行できるようになります。ところで、Node.jsのインストールにより、`npx`というコマンドも使えるようになっています。`npx` コマンドを使うと、npmで公開されている実行可能なパッケージのインストールと実行がコマンドひとつで省略できます。この後のユースケースでもそれらのツールを使うので、ここでツールの実行を試してみましょう。

ここでは例として`@js-primer/hello-world`というサンプル用のパッケージを実行します。`npx` コマンドでコマンドラインツールを実行するには、次のように `npx` コマンドにパッケージ名を渡して実行します。

```
$ npx @js-primer/hello-world
npx: 1個のパッケージを7.921秒でインストールしました。
Hello World!
```

このように、`npx` コマンドを使うことによりnpmで公開されているコマンドラインツールを簡単に実行できます。

## [コラム] コマンドラインツールのインストールと実行

npmで公開されているコマンドラインツールを実行する方法は `npx` コマンドだけではありません。`npm install` コマンドを使ってパッケージをインストールし、インストールされたパッケージのコマンドを実行する方法があります。通常の `npm install` コマンドは実行したカレントディレクトリにパッケージをインストールしますが、`--global` フラグを加えるとパッケージをグローバルインストールします。グローバルインストールされたパッケージのコマンドは、`node` コマンドや `npm` コマンドと同じように、任意の場所から実行できます。

次の例では `@js-primer/hello-world` パッケージをグローバルインストールしています。その後、パッケージに含まれている `js-primer-hello-world` コマンドを絶対パスの指定なしで呼び出しています。

```
$ npm install --global @js-primer/hello-world
$ js-primer-hello-world
Hello World!
```

# モジュール

JavaScriptにおけるモジュールは、保守性・名前空間・再利用性のために使われます。

- 保守性: 依存性の高いコードの集合を一箇所にまとめ、それ以外のモジュールへの依存性を減らすことができます
- 名前空間: モジュールごとに分かれたスコープがあり、グローバルの名前空間を汚染しません
- 再利用性: 便利な変数や関数を複数の場所にコピー・ペーストせず、モジュールとして再利用できます

ひとつのJavaScriptのモジュールはひとつのJavaScriptファイルに対応します。モジュールは変数や関数などを外部にエクスポートできます。また、別のモジュールで宣言された変数や関数などをインポートできます。この章ではECMAScriptモジュール（ESモジュール、JSモジュールとも呼ばれる）について見ていきます。ECMAScriptモジュールは、JavaScriptファイルをモジュール化する言語標準の機能です。

## ESモジュール

ESモジュールは、[export文](#)によって変数や関数などをエクスポートできます。また、[import文](#)を使って別のモジュールからエクスポートされたものをインポートできます。

まずは `export` 文について見ていきましょう。

### export文

`export` 文は変数や関数などをエクスポートし、別のモジュールから参照できるようにします。エクスポートの方法は、名前付きエクスポートとデフォルトエクスポートの2種類があります。名前付きエクスポートは、モジュールごとに複数の変数や関数などをエクスポートできます。デフォルトエクスポートは、モジュールごとにひとつしかエクスポートできません。それぞれについて見ていきましょう。

### 名前付きエクスポート

名前付きエクスポートには、すでに宣言した変数名と同じ名前でエクスポートする方法と、異なる名前でエクスポートする方法の2つがあります。次の例は、すでに宣言されている変数をエクスポートする構文です。`export` 文のあとに続けて`{}`を書き、その中にエクスポートする変数を入れます。

```
const foo = "foo";
function bar() { };
export { foo, bar };
```

また、次のように、宣言とエクスポートを同時に行うこともできます。

```
// 変数の宣言のみ
export let foo; // varも使用可
// 宣言と代入
export const bar = "bar"; // var, letも使用可
// 関数の宣言
export function fn() { }
// クラスの宣言
export class ClassName { }
```

もうひとつ的方法は、エクスポートする対象にエイリアスをつけて、宣言した変数名と違う名前でエクスポートする構文です。エクスポートする時にエイリアスをつけるには、次のような構文を使います。`as` のあとにエクスポートしたい名前を記述します。

```
const internalFoo = "foo";
function internalBar() { };
export { internalFoo as foo, internalBar as bar };
```

## デフォルトエクスポート

デフォルトエクスポートには、`export default` 文を使ってエクスポートする方法と、エイリアスに `default` を指定して名前付きエクスポートする方法の2つがあります。

`export default` 文は、後に続く式の評価結果をデフォルトエクスポートします。次の例では、すでに宣言されている変数をデフォルトエクスポートします。

```
const foo = "foo";
export default foo;
```

また、次のように関数とクラスは宣言とデフォルトエクスポートを同時に行うことができます。このとき関数やクラスは名前を省略できます。

```
// 関数の宣言
export default function () { }

// クラスの宣言
export default class {}
```

ただし、変数宣言は宣言とデフォルトエクスポートを同時にすることはできません。なぜなら、変数宣言はカンマ区切りで複数の変数を定義してしまうためです。次の例は実行できない不正なコードです。

```
// 関数の宣言はデフォルトエクスポートできない
export default const foo = "foo", bar = "bar";
```

エイリアスを使ってデフォルトエクスポートする方法では、次のように `as default` を付与します。`default` というエイリアスがつけられた名前付きエクスポートは、デフォルトエクスポートと同じ意味になります。

```
const foo = "foo";
function bar() { };
// fooは名前付き、barはデフォルトとしてエクスポートする
export { foo, bar as default };
```

## 再エクスポート

再エクスポートとは、別のモジュールからエクスポートされたものを、改めて自分自身からエクスポートしなおすことです。複数のモジュールからエクスポートされたものをまとめたモジュールを作るときなどに使われます。

再エクスポートするは次のように `export` 文のあとに `from` を続けて、別のモジュール名を指定します。

```
// ./myModule.jsのすべての名前付きエクスポートを再エクスポートする
export * from "./myModule.js";
// ./myModule.jsの名前付きエクスポートを選んで再エクスポートする
export { foo, bar } from "./myModule.js";
// ./myModule.jsの名前付きエクスポートにエイリアスをつけて再エクスポートする
export { foo as myModuleFoo, bar as myModuleBar } from "./myModule.js";
// ./myModule.jsのデフォルトエクスポートをデフォルトエクスポートとして再エクスポートする
export { default } from "./myModule.js";
// ./myModule.jsのデフォルトエクスポートを名前付きエクスポートとして再エクスポートする
export { default as myModuleDefault } from "./myModule.js";
// ./myModule.jsの名前付きエクスポートをデフォルトエクスポートとして再エクスポートする
export { foo as default } from "./myModule.js";
```

## import文

`import` 文は、別のモジュールからエクスポートされた変数や関数などを自身のモジュールにインポートします。インポートした変数や関数は、そのモジュールの先頭で宣言されたものと同じように扱えます。

エクスポートと同じように、インポートにも名前付きインポートとデフォルトインポートの2種類があります。それらに加え、指定したモジュールからすべてのエクスポートをまとめてインポートする方法と、副作用のためのインポートがあります。それぞれについて見てきましょう。

### 名前付きインポート

名前付きインポートは、指定したモジュールから名前を指定してインポートします。名前を指定してインポートするには、`import` 文のあとに続けて`{}`を書き、その中にインポートしたい名前付きエクスポートの名前を入れます。次の例では、`./myModule.js` モジュールから名前付きエクスポートされた`foo`と`bar`をインポートしています。

```
// 名前付きエクスポートされたfooとbarをインポートする
import { foo, bar } from "./myModule.js";
```

エクスポートするときと同じように、インポートするときにも別の名前をつけることができます。

```
// fooとして名前付きエクスポートされた変数をmyModuleFooとしてインポートする
import { foo as myModuleFoo } from "./myModule.js";
```

### デフォルトインポート

デフォルトインポートは、指定したモジュールのデフォルトエクスポートをインポートします。デフォルトインポートには、専用の構文を使う方法と、名前付きインポートで`default`を指定する方法の2つがあります。

デフォルトインポート専用の構文では、`import` 文のあとに任意の名前をつけてデフォルトエクスポートをインポートします。

```
// myModuleDefaultとしてデフォルトエクスポートをインポートする
import myModuleDefault from "./myModule.js";
```

もうひとつ的方法は、`default`を指定して名前付きインポートする方法です。デフォルトエクスポートは`default`という名前の名前付きエクスポートとして扱うこともできます。次のように、名前付きインポートの構文で`default`を指定し、エイリアスをつけてインポートできます。ただし、`default`は予約語なので、この方法では必ず`as`構文を使ってエイリアスをつける必要があります。

```
// myModuleDefaultとしてデフォルトエクスポートをインポートする
import { default as myModuleDefault } from "./myModule.js";
```

これら2つの構文は同時に記述できます。次のようにデフォルトインポートの構文と名前付きインポートを構文をカシマでつなげます。

```
// myModuleDefaultとしてデフォルトエクスポートをインポートし、
// 名前付きエクスポートされたfooをインポートする
import myModuleDefault, { foo } from "./myModule.js";
```

ESモジュールでは、エクスポートされていないものはインポートできません。なぜならESモジュールはJavaScriptのパース段階で依存関係が解決され、インポートする対象が存在しない場合はパースエラーとなるためです。デフォルトインポートは、指定したモジュールがデフォルトエクスポートをしている必要があります。同様に名前付きインポ

ートは、指定したモジュールが指定した名前付きエクスポートをしている必要があります。

## すべてをインポート

`import * as` 構文は、すべての名前付きエクスポートをまとめてインポートします。この方法では、モジュールごとの名前空間となるオブジェクトを宣言します。エクスポートされた変数や関数などにアクセスするには、その名前空間オブジェクトのプロパティを使います。

```
// すべての名前付きエクスポートをmyModuleオブジェクトとしてまとめてインポートする
import * as myModule from "./myModule.js";
// fooとして名前付きエクスポートされた変数にアクセスする
console.log(myModule.foo);
```

## 副作用のためのインポート

モジュールの中には、グローバルのコードを実行するだけで何もエクスポートしないものがあります。たとえば次のような、グローバル変数を操作するためのモジュールなどです。

```
import { foo } from "./myModule.js";

// グローバル変数を操作する
window.foo = foo;
```

このようなモジュールをインポートするには、副作用のためのインポート構文を使います。この構文では、モジュールのグローバルコードを実行するだけで何もインポートしません。

```
// ./sideEffects.jsのグローバルコードが実行される
import "./sideEffects.js";
```

## ESモジュールを実行する

作成したESモジュールを実行するためには、起点となるJavaScriptファイルをESモジュールとしてWebブラウザに読み込ませる必要があります。Webブラウザは `script` タグによってJavaScriptファイルを読み込み、実行します。次のように `script` タグに `type="module"` 属性を付与すると、WebブラウザはJavaScriptファイルをESモジュールとして読み込みます。

```
<!-- myModule.jsをECMAScriptモジュールとして読み込む -->
<script type="module" src="./myModule.js"></script>
<!-- インラインでも同じ -->
<script type="module">
import { foo } from "./myModule.js";
</script>
```

`type="module"` 属性が付与されない場合は通常のスクリプトとして扱われ、ECMAScriptモジュールの機能は使えません。スクリプトとして読み込まれたJavaScriptで `import` 文や `export` 文を使用すると、シンタックスエラーが発生します。

また、インポートされるモジュールの取得はネットワーク経由で解決されます。そのため、モジュール名はJavaScriptファイルの絶対URLあるいは相対URLを指定します。詳しくは[Todoアプリのユースケース](#)を参照してください。

## CommonJSモジュール

CommonJSモジュールとは、[Node.js](#)環境で利用されているモジュール化の仕組みです。CommonJSモジュールはESモジュールの仕様が策定されるよりもずっと古くから使われています。Node.jsの標準パッケージや[NPM](#)で配布されるサードパーティパッケージは、CommonJSモジュールとして提供されていることがほとんどです。

CommonJSモジュールはNode.jsのグローバル変数である `module` 変数を使って変数や関数などをエクスポートします。次のように `module.exports` プロパティに代入されたオブジェクトが、そのJavaScriptファイルからエクスポートされます。複数の名前付きエクスポートが可能なESモジュールと違い、CommonJSでは `module.exports` プロパティの値だけがエクスポートの対象です。

```
module.exports = {
  foo: "foo"
};
```

モジュールをインポートするには、`require` グローバル関数を使います。次のように `require` 関数にモジュール名を渡し、戻り値としてエクスポートされたオブジェクトを受け取ります。

```
const myModule = require("./myModule.js");
```

Node.jsではESモジュールもサポートする予定ですが、現在はまだ安定した機能としてサポートされていません。

## [コラム] モジュールバンドラー

モジュールバンドラーとは、JavaScriptのモジュール依存関係を解決し、複数のモジュールをひとつのJavaScriptファイルに結合するツールのことです。モジュールバンドラーは起点となるモジュールが依存するモジュールを次々にたり、適切な順序になるように結合（バンドル）します。

NPMによって多くのJavaScriptライブラリがNode.js向けに配布されていますが、これらはほぼすべてCommonJSモジュールです。それらのライブラリを使ったアプリケーションをWebブラウザで実行するためには、CommonJSモジュールを解決し、ひとつのJavaScriptファイルに結合する必要がありました。結果的に、Node.js向けでないアプリケーションもモジュール化することが一般的になり、モジュールバンドラーはJavaScript開発において無くてはならないものになりました。

モジュールバンドラーにはCommonJSだけでなくESモジュールにも対応したものもあります。また、バンドルする際にJavaScriptコードの最適化を行うなどバンドル以外の機能をもつものもあります。[JavaScriptモジュールについてのドキュメント](#)では、WebにおけるJavaScriptのモジュールと、バンドルする目的などについて詳しくまとめられています。

## ユースケース: Ajax通信

ここではウェブブラウザ上でAjax通信を行うユースケースとして、GitHubのユーザーIDからプロフィール情報を取得するアプリケーションを作成します。

作成するアプリケーションは次の要件を満たすものとします。

- GitHubのユーザーIDをテキストボックスに入力できる
- 入力されたユーザーIDを元にGitHubからユーザー情報を取得する
- 取得したユーザー情報をアプリケーション上で表示する

## エントリポイント

エントリポイントとは、アプリケーションの中で一番最初に呼び出される部分のことです。アプリケーションを作成するにあたり、まずはエントリポイントを用意しなければなりません。

Webアプリケーションにおいては、常にHTMLドキュメントがエントリポイントとなります。ウェブブラウザによりHTMLドキュメントが読み込まれたあとに、HTMLドキュメント中で読み込まれたJavaScriptが実行されます。

## HTMLファイルの用意

エントリポイントとして、まずは最低限の要素だけを配置したHTMLファイルを作成しましょう。`body`要素の一番下で読み込んでいるindex.jsが、今回のアプリケーションの処理を記述するJavaScriptファイルです。

```
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Ajax Example</title>
  </head>
  <body>
    <h2>GitHub User Info</h2>
    <script src="index.js"></script>
  </body>
</html>
```

index.jsには、スクリプトが正しく読み込まれたことを確認できるよう、コンソールにログを出力する処理だけを書いておきます。

```
console.log("index.js: loaded");
```

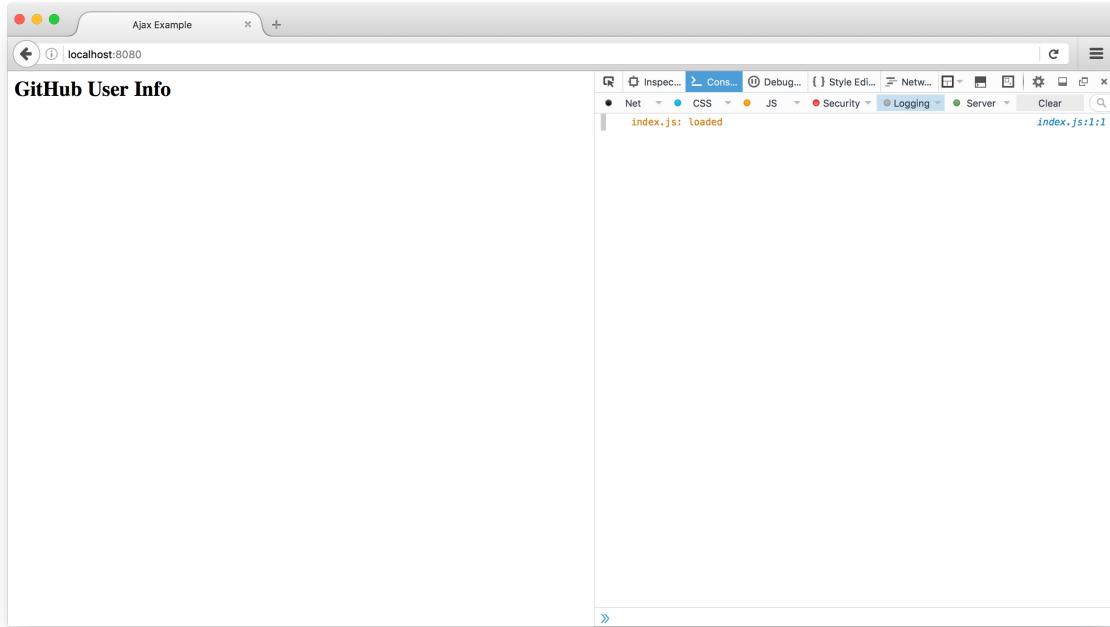
開発用のローカルサーバーを立ち上げて、ウェブブラウザでindex.htmlにアクセスしてみましょう。ローカルサーバーを立ち上げずに直接HTMLファイルを開くこともできますが、その場合`file:///`から始まるURLになります。

`file`スキーマでは[Same Origin Policy](#)により、多くの場面でアプリケーションは正しく動作しません。本章はローカルサーバーを立ち上げた上で、`http`スキーマのURLでアクセスすることを前提としています。

index.htmlにアクセスすると、次の画像のようにログが表示されます。Console APIで出力したログを確認するには、ウェブブラウザの開発者ツールを開く必要があります。ほとんどのブラウザで開発者ツールが同梱されていますが、本章ではFirefoxを使って確認します。Firefoxの開発者ツールは次のいずれかの方法で開きます。

- Firefoxメニュー（メニューバーがある場合やmacOSでは、ツールメニュー）のWeb開発サブメニューで“Webコンソール”を選択する
- キーボードショートカットCtrl+Shift+K（macOSではCommand+Option+K）を押下する

詳細は[“Webコンソールを開く”](#)を参照してください。



# HTTP通信

アプリケーションが実行できるようになったので、次はGitHubのAPIを呼び出す処理を実装していきます。当然ですが、GitHubのAPIを呼び出すためにはHTTP通信を行う必要があります。ウェブブラウザ上でJavaScriptからHTTP通信を行うには `XMLHttpRequest` という機能を使います。

## XMLHttpRequest

`XMLHttpRequest` (XHR) はクライアントとサーバー間でデータをやり取りするためのAPIです。XHRを使うことで、ページ全体を再読み込みすることなくURLからデータを取得できます。

GitHubが提供している、ユーザー情報を取得するためのWebAPIを呼び出すコードは次のようになります。リクエストを送信するためには、まず `XMLHttpRequest` クラスのインスタンスを作ります。作成したXHRのインスタンスに、リクエストメソッドとURLを与えることで、HTTPリクエストが組み立てられます。URLをテンプレート文字列にしているのは、変数でユーザーIDを設定するためです。URLをオープンして組み立てられたXHRは、最後に `send` することでサーバーとの通信を開始します。

```
const request = new XMLHttpRequest();
request.open("GET", `https://api.github.com/users/${userId}`);
request.send();
```

このように、XHRを使ったHTTP通信は基本的に3ステップで行われます。

1. XHRのインスタンスを生成 (`new XMLHttpRequest`)
2. リクエストを初期化 (`request.open` メソッド)
3. リクエストを送信 (`request.send` メソッド)

## レスポンスの受け取り

GitHubのAPIに対してHTTPリクエストを送信しましたが、まだレスポンスを受け取る処理を書いていません。次はサーバーから返却されたレスポンスのログをコンソールに出力する処理を実装します。

非同期的なXHRの場合、レスポンスはXHRが発火するイベントのコールバック内で受け取れます。実はXHRにはHTTP通信を同期的に実行するモードも存在しますが、一般的にはXHRを同期的に行うことはありません。なぜならWebブラウザ上で実行されるJavaScriptはシングルスレッドであり、HTTP通信を行っている間はすべての処理がブロックされてしまうからです。そもそも本章のテーマでもあるAJAXの根幹はAsynchronous (非同期的) であることなので、ここで登場するXHRはすべて非同期的とします。

送信したXHRにHTTPレスポンスが返却されると、`load` イベントが発生します。XHRインスタンスの `addEventListener` メソッドにイベント名とコールバック関数を渡すことで、指定したイベントを受け取れます。コールバック関数の第1引数には `Event` オブジェクトが渡されます。このオブジェクトの `target` プロパティには、[イベントを発生させたオブジェクト \(EventTarget\)](#) であるXHRのインスタンスがセットされています。XHRの `responseText` プロパティからは、HTTPレスポンスが文字列で取得できます。また、`status` プロパティからはHTTPレスポンスのステータスコードが取得できます。

```
const request = new XMLHttpRequest();
request.open("GET", `https://api.github.com/users/${userId}`);
request.addEventListener("load", (event) => {
    console.log(event.target.status); // => 200
    console.log(event.target.responseText); // => "..."
});
request.send();
```

HTTPレスポンスの内容によらず、`responseText` は常に文字列としてレスポンスを読み取れます。一方、XHRには`response` というプロパティもあります。リクエストを送るときに`responseType` を指定しておくと、その型にしたがって変換されたオブジェクトが`response` から取得できます。`response` プロパティは便利なAPIなのですが、サーバー側の実装やブラウザの実装などに依存するため、本章では扱いません。興味があれば、詳細については[XMLHttpRequest.response](#)を参照してください。

## エラーハンドリング

HTTP通信にはエラーがつきものです。もちろんXHRを使った通信においても、エラーをハンドリングする方法があります。XHRではレスポンスの受け取りと同じように、イベントリスナによってエラーのハンドリングが可能です。サーバーとの通信に際してエラーが発生した場合は、`error` イベントが発生されます。

```
request.addEventListener("error", () => {
  console.log("Network Error!");
});
```

注意すべき点は、サーバーがレスポンスとして返却するエラーは、通常のレスポンスと同様に`load` イベントで受け取ることです。たとえば、ステータスコードが200以外である場合のハンドリングは次のように行います。

```
request.addEventListener("load", (event) => {
  if (event.target.status !== 200) {
    console.log(`Error: ${event.target.status}`);
    return;
  }
});
```

ここまでの中をまとめ、GitHubからユーザー情報を取得する関数を`getUserInfo` という名前で定義します。

```
function getUserInfo(userId) {
  const request = new XMLHttpRequest();
  request.open("GET", `https://api.github.com/users/${userId}`);
  request.addEventListener("load", (event) => {
    if (event.target.status !== 200) {
      console.log(`${event.target.status}: ${event.target.statusText}`);
      return;
    }
    console.log(event.target.status);
    console.log(event.target.responseText);
  });
  request.addEventListener("error", () => {
    console.error("Network Error");
  });
  request.send();
}
```

`index.js`では関数を定義しているだけで、呼び出しあは行っていません。ページを読み込むたびにGitHubのAPIを呼び出してしまって、呼び出し回数の制限を超えるてしまうおそれがあります。そこで`getUserInfo` 関数を呼び出すため、HTMLドキュメント側にボタンを追加します。ボタンの`click`イベントで`getUserInfo` 関数を呼び出し、固定のユーザーIDを引数として与えています。

```
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Ajax Example</title>
  </head>
  <body>
```

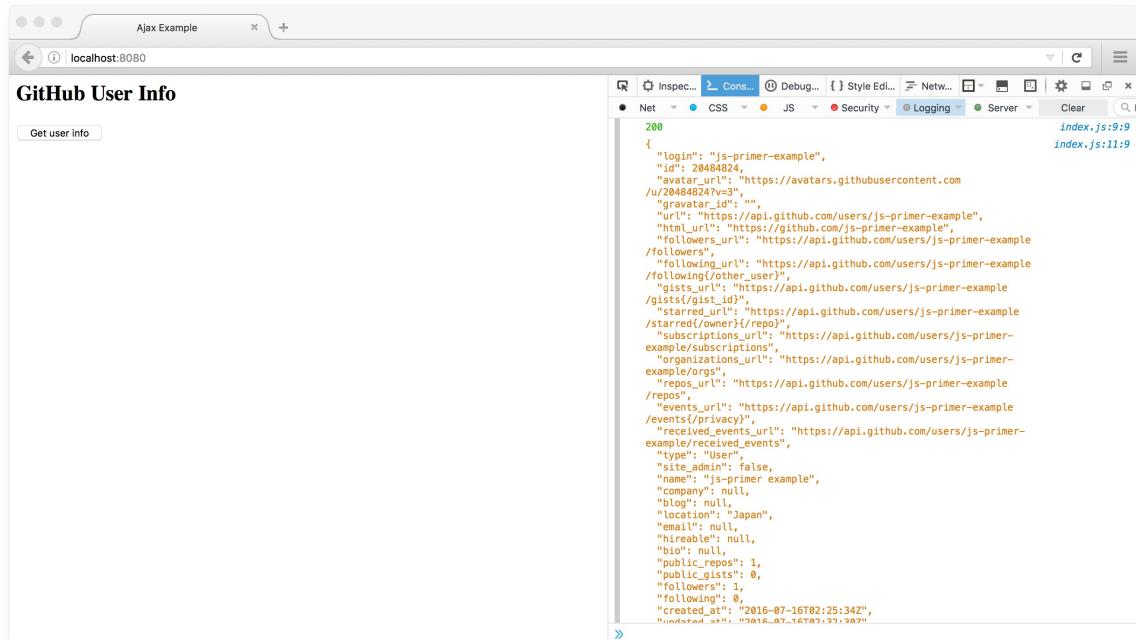
```

<h2>GitHub User Info</h2>

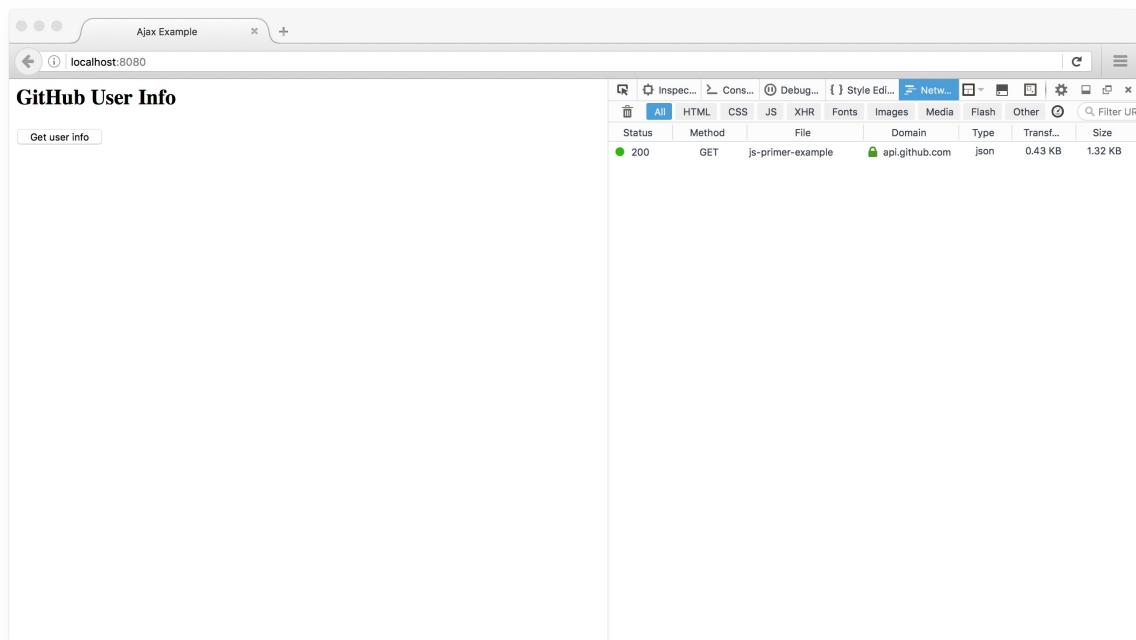
<button onclick="getUserInfo('js-primer-example');">Get user info</button>
<script src="index.js"></script>

```

準備ができたら、ローカルサーバーを立ち上げてindex.htmlにアクセスしましょう。ボタンを押すとHTTP通信が行われ、コンソールにステータスコードとレスポンスのログが出力されます。



また、開発者ツールのネットワーク画面を開くと、確かにGitHubのサーバーに対してHTTP通信が行われていることを確認できます。





## データを表示する

XHRを使ってサーバーからデータを取得できたので、データをHTMLに整形して表示してみましょう。

### レスポンスをオブジェクトに変換する

まずはGitHub APIからのデータをJavaScriptで扱うために、レスポンス文字列をオブジェクトに変換します。 GitHub APIのレスポンスはJSON形式なので、オブジェクトへの変換には`JSON.parse()`を使います。

```
request.addEventListener("load", (event) => {
  if (event.target.status !== 200) {
    console.log(`[${event.target.status}]: ${event.target.statusText}`);
    return;
  }

  const userInfo = JSON.parse(event.target.responseText);
});
```

## HTMLを組み立てる

HTML文字列の生成にはテンプレートリテラルを使います。テンプレートリテラルは文字列中の改行が可能なため、HTMLのインデントを表現できて見通しが良くなります。また、文字列の埋め込みも簡単なため、HTMLのテンプレートに対して動的なデータを当てはめるのに適しています。

次のコードではGitHubのユーザー情報から組み立てるHTMLのテンプレートを宣言しています。

```
const view = `

<h4>${userInfo.name} (@${userInfo.login})</h4>

<dl>
  <dt>Location</dt>
  <dd>${userInfo.location}</dd>
  <dt>Repositories</dt>
  <dd>${userInfo.public_repos}</dd>
</dl>
`;
```

このテンプレートに `userInfo` オブジェクトを適用すると、次のようなHTML文字列になります。

```
<h4>js-primer example (@js-primer-example)</h4>

<dl>
  <dt>Location</dt>
  <dd>Japan</dd>
  <dt>Repositories</dt>
  <dd>1</dd>
</dl>
```

## HTML文字列をDOMに追加する

次に、生成したHTML文字列をDOMツリーに追加して表示します。まずは動的にHTMLをセットするために、目印となる要素をindex.htmlに追加します。今回は `result` というidを持ったdiv要素（以降 `div#result` と表記します）を配置します。

```
<body>
  <h2>GitHub User Info</h2>

  <button onclick="getUserInfo('js-primer-example');">Get user info</button>

  <div id="result"></div>

  <script src="index.js"></script>
</body>
```

ここから、`div#result` 要素の子要素としてHTML文字列を挿入することになります。`document.getElementById`メソッドを使い、id属性が設定された要素にアクセスします。`div#result` 要素が取得できたら、先ほど生成したHTML文字列を `innerHTML` プロパティにセットします。

```
const result = document.getElementById("result");
result.innerHTML = view;
```

JavaScriptによってHTML要素をDOMに追加する方法は、大きく分けて2つあります。ひとつは、今回のようにHTML文字列を`Element#innerHTML`プロパティにセットする方法です。もうひとつは、文字列ではなく`Element`オブジェクトを生成して手続き的にツリー構造を構築する方法です。後者はセキュリティ的に安全ですが、コードは少し冗長になります。今回は `Element#innerHTML` プロパティを使いつつ、セキュリティのための処理を行うこととします。

## HTML文字列をエスケープする

`Element#innerHTML` に文字列をセットすると、その文字列はHTMLとして解釈されます。たとえばGitHubのユーザー名に `<` 記号や `>` 記号が含まれていると、意図しない構造のHTMLになる可能性があります。これを回避するためには、文字列をセットする前に、特定の記号を安全な表現に置換する必要があります。この処理を一般にHTMLのエスケープと呼びます。

多くのViewライブラリは内部にエスケープ機構を持っていて、動的にHTMLを組み立てるときにはデフォルトでエスケープをしてくれます。または、[エスケープ用のライブラリ](#)を利用するケースも多いでしょう。今回のように独自実装するのは特別なケースで、一般的にはライブラリが提供する機能を使うのがほとんどです。

次のように、特殊な記号に対するエスケープ処理を `escapeSpecialChars` 関数として宣言します。

```
function escapeSpecialChars(str) {
  return str
    .replace(/&/g, "&amp;")
    .replace(/</g, "&lt;")
    .replace(/>/g, "&gt;")
    .replace(/\"/g, "&quot;")
    .replace(/\'/g, "&#039;");
}
```

この `escapeSpecialChars` 関数を、HTML文字列の中で `userInfo` から値を注入しているすべての箇所で行います。ただし、テンプレートリテラル中で挿入している部分すべてに関数を適用するのは手間ですし、メンテナンス性もよくありません。そこで、[テンプレートリテラルをタグ付け](#)することで、明示的にエスケープ用の関数を呼び出す必要がないようにします。タグ付けされたテンプレートリテラルは、テンプレートによる値の埋め込みを関数の呼び出しとして扱えます。

次の `escapeHTML` はテンプレートリテラルにタグ付けするためのタグ関数です。タグ関数は第一引数に文字列リテラルの配列、第二引数に埋め込まれる値の配列が与えられます。`escapeHTML` 関数では、文字列リテラルと値が元の順番どおりに並ぶように文字列を組み立てつつ、値が文字型であればエスケープするようにしています。

```
function escapeHTML(strings, ...values) {
  return strings.reduce((result, string, i) => {
    const value = values[i - 1];
    if (typeof value === "string") {
      return result + escapeSpecialChars(value) + string;
    } else {
      return result + String(value) + string;
    }
  });
}
```

`escapeHTML` 関数はタグ関数なので、通常の`()`による呼び出しへなく、テンプレートリテラルに対してタグ付けして使います。テンプレートリテラルのバッククオート記号の前に関数を書くと、関数がタグ付けされます。

```
const view = escapeHTML`  

<h4>${userInfo.name} (${userInfo.login})</h4>  

<dl>  

  <dt>Location</dt>  

  <dd>${userInfo.location}</dd>  

  <dt>Repositories</dt>  

  <dd>${userInfo.public_repos}</dd>  

</dl>  

`;  

const result = document.getElementById("result");
result.innerHTML = view;
```

これでHTML文字列の生成とエスケープができました。ここまでで、`index.js`と`index.html`は次のようになっています。

```
function getUserInfo(userId) {
  const request = new XMLHttpRequest();
  request.open("GET", `https://api.github.com/users/${userId}`);
  request.addEventListener("load", (event) => {
    if (event.target.status !== 200) {
      console.log(` ${event.target.status}: ${event.target.statusText}`);
      return;
    }

    const userInfo = JSON.parse(event.target.responseText);

    const view = escapeHTML`  

<h4>${userInfo.name} (${userInfo.login})</h4>  

<dl>  

  <dt>Location</dt>  

  <dd>${userInfo.location}</dd>  

  <dt>Repositories</dt>  

  <dd>${userInfo.public_repos}</dd>  

</dl>  

`;  

    const result = document.getElementById("result");
    result.innerHTML = view;
  });
  request.addEventListener("error", () => {
    console.error("Network Error");
  });
  request.send();
}
```

```

}

function escapeSpecialChars(str) {
    return str
        .replace(/&/g, "&amp;")
        .replace(/</g, "&lt;")
        .replace(/>/g, "&gt;")
        .replace(/\"/g, "&quot;")
        .replace(/\'/g, "&#039;");
}

function escapeHTML(strings, ...values) {
    return strings.reduce((result, string, i) => {
        const value = values[i - 1];
        if (typeof value === "string") {
            return result + escapeSpecialChars(value) + string;
        } else {
            return result + String(value) + string;
        }
    });
}

```

```

<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Ajax Example</title>
    </head>
    <body>
        <h2>GitHub User Info</h2>

        <button onclick="getUserInfo('js-primer-example');">Get user info</button>

        <div id="result"></div>

        <script src="index.js"></script>
    </body>
</html>

```

アプリケーションを開いてボタンを押すと、次のようにユーザー情報が表示されます。

### GitHub User Info

js-primer example (@js-primer-example)



Location  
Japan  
Repositories  
1



# Promiseを活用する

ここまでで、XHRを使ってAjax通信を行い、サーバーから取得したデータを表示できました。最後に、Promiseを使ってソースコードを整理することで、エラーハンドリングをしっかりと行います。

## 関数の分割

まずは、大きくなりすぎた `getUserInfo` 関数を整理しましょう。この関数では、XHRを使ったデータの取得・HTML文字列の組み立て・組み立てたHTMLの表示を行っています。そこで、HTML文字列を組み立てる `createView` 関数とHTMLを表示する `displayView` 関数を作り、処理を分割します。

また、後述するエラーハンドリングを行いやすくするため、アプリケーションにエントリポイントを設けます。

`index.js` に新しく `main` 関数を作り、その中に `getUserInfo` 関数を呼び出すようにします。

```
function main() {
  getUserInfo("js-primer-example");
}

function getUserInfo(userId) {
  const request = new XMLHttpRequest();
  request.open("GET", `https://api.github.com/users/${userId}`);
  request.addEventListener("load", (event) => {
    if (event.target.status !== 200) {
      console.log(` ${event.target.status}: ${event.target.statusText}`);
      return;
    }

    const userInfo = JSON.parse(event.target.responseText);

    const view = createView(userInfo);
    displayView(view);
  });
  request.addEventListener("error", () => {
    console.error("Network Error");
  });
  request.send();
}

function createView(userInfo) {
  return escapeHTML`

#### ${userInfo.name} (@${userInfo.login})

![${userInfo.login}](${userInfo.avatar_url})

<dt>Location</dt>
<dd> ${userInfo.location}</dd>
<dt>Repositories</dt>
<dd> ${userInfo.public_repos}</dd>


`;
}

function displayView(view) {
  const result = document.getElementById("result");
  result.innerHTML = view;
}
```

## XHRをPromiseでラップする

次に、`getUserInfo` 関数で行っているXHRの処理を整理します。これまでXHRのコールバック関数の中で処理を行っていましたが、これをPromiseを使った処理に書き換えます。コールバック関数を使うと、ソースコードのネストが深くなったり、例外処理が複雑になったりします。Promiseを用いることで、可読性を保ちながらエラーハンドリングを簡単に行えます。

コールバック関数を使う形式のAPIをPromiseに置き換えるのは、次のコードのように `new Promise()` を用いるのが一般的です。Promiseのコンストラクタには、`resolve` と `reject` の2つの関数オブジェクトを引数とする関数を渡します。ひとつめの引数は非同期処理が成功したときに呼び出す関数で、ふたつめは失敗した時に呼び出す関数です。

```
new Promise((resolve, reject) => {
  // ここで非同期処理を行う
});
```

Promiseのコンストラクタに渡す関数で、XHRの処理を行います。作成されたPromiseは成功か失敗のどちらかで完了させなければなりません。非同期処理が成功したら第1引数の `resolve` 関数を、失敗なら第2引数の `reject` 関数を呼び出します。

作成したPromiseのオブジェクトを `return` することで、`getUserInfo` 関数はPromiseを返す関数になりました。`getUserInfo` 関数がPromiseを返すことによって、それを呼び出す `main` 関数の方で非同期処理の結果を扱えるようになります。

```
function getUserInfo(userId) {
  return new Promise((resolve, reject) => {
    const request = new XMLHttpRequest();
    request.open("GET", `https://api.github.com/users/${userId}`);
    request.addEventListener("load", (event) => {
      if (event.target.status !== 200) {
        console.log(`[${event.target.status}]: ${event.target.statusText}`);
        reject(); // ステータスコードが200じゃないので失敗
      }
    });

    const userInfo = JSON.parse(event.target.responseText);

    const view = createView(userInfo);
    displayView(view);
    resolve(); // 完了
  });
  request.addEventListener("error", () => {
    console.error("Network Error");
    reject(); // 通信エラーが発生したので失敗
  });
  request.send();
});
```

## エラーハンドリング

このままではPromiseに置き換えた意味がないので、Promiseを使ったエラーハンドリングを行いましょう。Promiseのコンテキスト内で発生したエラーは、`Promise#catch` メソッドを使って一箇所で受け取れます。次のコードでは、`getUserInfo` 関数から返されたPromiseオブジェクトを使い、エラーが起きた時にログを出力します。`reject` 関数に渡したエラーは `catch` のコールバック関数で第1引数として受け取れます。

```
function main() {
  getUserInfo("js-primer-example")
    .catch((error) => {
      console.error(`エラーが発生しました (${error})`);
    });
}

function getUserInfo(userId) {
```

```

return new Promise((resolve, reject) => {
  const request = new XMLHttpRequest();
  request.open("GET", `https://api.github.com/users/${userId}`);
  request.addEventListener("load", (event) => {
    if (event.target.status !== 200) {
      reject(new Error(`[${event.target.status}]: ${event.target.statusText}`));
    }

    const userInfo = JSON.parse(event.target.responseText);

    const view = createView(userInfo);
    displayView(view);
    resolve();
  });
  request.addEventListener("error", () => {
    reject(new Error("ネットワークエラー"));
  });
  request.send();
});
}

```

## Promiseチェーンへの置き換え

Promiseは `Promise#then` メソッドを使うことで、複数の処理の連鎖を表現できます。複数の処理を `then` で分割し、連鎖させたものを、ここではPromiseチェーンと呼びます。基本的に、`then` はコールバック関数の戻り値をそのまま次の `then` へ渡します。ただし、コールバック関数の戻り値がPromiseである場合はその完了を待ち、Promiseの結果の値を次の `then` に渡します。つまり、`then` のコールバック関数が同期処理から非同期処理に変わったとしても、次の `then` が受け取る値の型は変わらないということです。

Promiseチェーンを使って処理を分割する利点は、同期処理と非同期処理を区別せずに連鎖できることです。一般に、同期的に書かれた処理を後から非同期処理へと変更することは、全体を書き換える必要があるため難しいです。そのため、最初から処理を分けておき、処理を `then` を使って繋ぐことで、変更に強いコードを書くことができます。どのように処理を区切るかは、それぞれの関数が受け取る値の型と、返す値の型に注目するのがよいでしょう。Promiseチェーンで処理を分けることで、それぞれの処理が簡潔になりコードの見通しがよくなります。

さて、今の `getUserInfo` 関数では `load` イベントのコールバック関数でHTMLの組み立てと表示も行っています。これをPromiseチェーンを使うように書き換えると、次のようにできます。`getUserInfo` 関数では、`resolve` 関数に `userInfo` を渡し、次の `then` でコールバック関数の引数として受け取っています。同じように、`userInfo` を受け取った関数は `createView` 関数を呼び出し、その戻り値を次の `then` に渡しています。

```

function main() {
  getUserInfo("js-primer-example")
    .then((userInfo) => createView(userInfo))
    .then((view) => displayView(view))
    .catch((error) => {
      console.error(`エラーが発生しました (${error})`);
    });
}

function getUserInfo(userId) {
  return new Promise((resolve, reject) => {
    const request = new XMLHttpRequest();
    request.open("GET", `https://api.github.com/users/${userId}`);
    request.addEventListener("load", (event) => {
      if (event.target.status !== 200) {
        reject(new Error(`[${event.target.status}]: ${event.target.statusText}`));
      }

      const userInfo = JSON.parse(event.target.responseText);
      resolve(userInfo);
    });
    request.addEventListener("error", () => {

```

```

        reject(new Error("ネットワークエラー"));
    });
    request.send();
});
}

```

## [コラム] Fetch API

Fetch APIとは、ページの外部からリソースを取得するためのインターフェースを定義した、Webブラウザの標準APIです。Fetch APIはfetch関数など、リソースを取得するためのAPIを定義しています。fetch関数はPromiseを返すのが特徴です。たとえば、本章で扱ったXHRによるgetUserInfo関数は、fetch関数を使うと次のようになります。

```

function getUserInfo(userId) {
    // 暗黙にGETリクエストとなる
    // Responseオブジェクトがthenに渡される
    return fetch(`https://api.github.com/users/${userId}`)
        .then(response => {
            if (!response.status !== 200) {
                throw new Error(`${response.status}: ${response.statusText}`);
            }
            // jsonメソッドは、レスポンスボディをJSONとしてパースしたPromiseオブジェクトを返す
            return response.json();
        }, error => {
            throw new Error("ネットワークエラー");
        });
}

```

今回のユースケースではFetchへの置き換えが可能ですが、コーラルバック関数をPromiseでラップする手法を学ぶために、あえてXHRを利用しています。また、ログレスイベントやリクエストの中止などXHRでしか使えない機能もあるため、常にFetchで置き換えられるわけではありません。

Fetchの詳しい使い方については[Fetchに関するドキュメント](#)を参照してください。

## ユーザーIDを変更できるようにする

仕上げとして、今までjs-primer-exampleで固定していたユーザーIDを変更できるようにしましょう。index.htmlに<input>タグを追加し、JavaScriptから値を取得するためにuserIdというIDを付与しておきます。

```

<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Ajax Example</title>
    </head>
    <body>
        <h2>GitHub User Info</h2>

        <input id="userId" type="text" value="js-primer-example">
        <button onclick="main();">Get user info</button>

        <div id="result"></div>

        <script src="index.js"></script>
    </body>
</html>

```

index.jsにも<input>タグから値を受け取るための処理を追加すると、最終的に次のようにになります。

```

function main() {
  const userId = getUserId();
  getuserInfo(userId)
    .then((userInfo) => createView(userInfo))
    .then((view) => displayView(view))
    .catch((error) => {
      console.error(`エラーが発生しました (${error})`);
    });
}

function getUserInfo(userId) {
  return new Promise((resolve, reject) => {
    const request = new XMLHttpRequest();
    request.open("GET", `https://api.github.com/users/${userId}`);
    request.addEventListener("load", (event) => {
      if (event.target.status !== 200) {
        reject(new Error(`#${event.target.statusText}`));
      }

      const userInfo = JSON.parse(event.target.responseText);
      resolve(userInfo);
    });
    request.addEventListener("error", () => {
      reject(new Error("ネットワークエラー"));
    });
    request.send();
  });
}

function getUserId() {
  const value = document.getElementById("userId").value;
  return encodeURIComponent(value);
}

function createView(userInfo) {
  return escapeHTML`

#### ${userInfo.name} (@${userInfo.login})

![ ${userInfo.login}]( ${userInfo.avatar_url})
<dl>
  <dt> Location </dt>
  <dd> ${userInfo.location} </dd>
  <dt> Repositories </dt>
  <dd> ${userInfo.public_repos} </dd>
</dl>
`;
}

function displayView(view) {
  const result = document.getElementById("result");
  result.innerHTML = view;
}

function escapeSpecialChars(str) {
  return str
    .replace(/&/g, "&")
    .replace(/</g, "<")
    .replace(/>/g, ">")
    .replace(/"/g, """)
    .replace(/\'/g, "'");
}

function escapeHTML(strings, ...values) {
  return strings.reduce((result, string, i) => {
    const value = values[i - 1];
    if (typeof value === "string") {
      return result + escapeSpecialChars(value) + string;
    } else {
      return result + String(value) + string;
    }
  });
}

```

```
        }
    });
}
```

アプリケーションを実行すると、次のようにになります。要件を満たすことができたので、このアプリケーションはこれで完成です。

### GitHub User Info

[js-primer-example](#) [Get user info](#)

**js-primer example (@js-primer-example)**



Location  
Japan  
Repositories  
1

## ユースケース: Node.jsでCLIアプリケーション

ここではNode.jsでCLI（コマンドラインインターフェース）アプリケーションを開発します。 CLIのユースケースとしてMarkdown形式のテキストファイルをHTMLテキストに変換するツールを作成します。

作成するアプリケーションは次の要件を満たすものとします。

- コマンドライン引数として変換対象のファイルパスを受け取る
- Markdown形式のファイルを読み込み、変換したHTMLを標準出力に表示する
- 変換の設定をコマンドライン引数でオプションとして与えられる

# Node.jsでHello World

実際にアプリケーションを作成するまえに、まずはHello Worldアプリケーションを通じてNode.jsのCLIアプリケーションの基本を学びましょう。

## Hello World

まずはNode.jsでHello Worldアプリケーションを作ってみましょう。具体的には、実行すると標準出力に"Hello World!"という文字列を表示するCLIアプリケーションを記述します。はじめに用意するのは、アプリケーションのエントリポイントとなるJavaScriptファイルです。適当なディレクトリに `main.js` という名前でファイルを作成し、次のように記述します。

```
console.log("Hello World!");
```

Webブラウザの実行環境では、`console.log` 関数の出力先はブラウザのコンソールでしたが、Node.js環境では標準出力になります。このコードは、標準出力に `Hello World!` という文字列を出力するものです。

JavaScriptのコードをNode.jsで実行するには、`node` コマンドを使用します。次のコマンドを実行して、Node.jsで `main.js` を実行します。

```
$ node main.js
Hello World!
```

Node.jsの基本は、エントリポイントとなるJavaScriptファイルを作成し、そのファイルを `node` コマンドの引数に渡して実行するという流れです。また、WebブラウザのJavaScriptと同じく、コードは1行目から順に実行されます。

## Node.jsとブラウザのグローバルオブジェクト

Node.jsはChromeと同じV8エンジンを利用しているため、ECMAScriptで定義されている基本構文はほとんどブラウザと同じように使えます。ただし、ブラウザ環境とNode.js環境では利用できるグローバルオブジェクトが違うため、アプリケーションを開発するときにはその違いを理解しなくてはなりません。

ECMAScriptで定義されているグローバルオブジェクトはブラウザとNode.jsどちらの環境にも存在します。たとえば `Boolean`、`String` などのラッパーオブジェクトや、`Map`、`Array`、`Promise` のようなビルトインオブジェクトがそれにあたります。

Webブラウザ環境のグローバルオブジェクトは `window` オブジェクトですが、Node.jsでは`global`と呼ばれるオブジェクトがグローバルオブジェクトになります。ブラウザの `window` オブジェクトにはたとえば次のようなプロパティや関数があります。

- `document`
- `XMLHttpRequest`

一方、Node.jsの `global` オブジェクトにはたとえば次のようなプロパティや関数があります。

- `process`
- `Buffer`

それぞれのグローバルオブジェクトにあるプロパティなどは、同じ名前でグローバル変数や関数としてアクセスできます。たとえば `window.document` プロパティは、グローバル変数の `document` としてアクセスすることもできます。

また、ECMAScriptで定義されたものではありませんが、ほぼ同等の機能と名前をもつプロパティや関数がブラウザとNode.jsどちらにもある場合もあります。たとえば次のようなものです。

- Console API
- `setTimeout` 関数

これらを踏まえた上で、次のセクションからCLIアプリケーションの開発をはじめていきましょう。

## コマンドライン引数を処理する

このユースケースで作成するCLIアプリケーションの目的は、コマンドライン引数として与えられたファイルを変換することです。このセクションではコマンドライン引数を受け取って、それをパースするところまでを行います。

### process オブジェクトとコマンドライン引数

コマンドライン引数を扱う前に、まずは `process` オブジェクトについて触れておきます。`process` オブジェクトは Node.js 実行環境のグローバル変数のひとつです。`process` オブジェクトが提供するのは、現在の Node.js の実行プロセスについて、情報の取得と操作を行う API です。 詳細は [公式ドキュメント](#) を参照してください。

コマンドライン引数へのアクセスを提供するのは、`process` オブジェクトの `argv` プロパティで、文字列の配列になっています。例として、次のように `process.argv.js` を記述します。

```
console.log(process.argv);
```

このスクリプトを次のようなコマンドで実行します。

```
$ node process.argv.js one two=three four
```

すると、出力結果は次のようになります。

```
[  
  '/usr/local/bin/node', // Node.jsの実行プロセスのパス  
  '/Users/laco/nodecli/argument-parse/src/process-argv.js', // 実行したスクリプトファイルのパス  
  'one', // 1番目の引数  
  'two=three', // 2番目  
  'four' // 3番目  
]
```

1番目と2番目の要素は常に `node` コマンドと実行されたスクリプトのファイルパスになります。つまりアプリケーションがコマンドライン引数として使うのは、3番目以降の要素です。

## コマンドライン引数をパースする

`process.argv` 配列を使えばコマンドライン引数を取得できますが、取得できるのは文字列の配列です。そのままではアプリケーションから扱いにくいため、コマンドライン引数をパースして整形する必要があります。文字列処理を自前で行うこともできますが、このような一般的な処理は既存のライブラリを使うと簡単に書けます。今回は `commander` というライブラリを使ってコマンドライン引数をパースします。

### npmを使ってパッケージをインストールする

Node.js のライブラリの多くは `npm` というパッケージマネージャーを使ってインストールできます。`npm` や `npn` コマンドについての詳細は [公式ドキュメント](#) や [npmのGitHubリポジトリ](#) を参照してください。Node.js をインストールすると、`node` コマンドだけでなく `npn` コマンドも使えるようになっています。

`npm` でパッケージをインストールする前に、まずは `npm` のパッケージ管理環境を作りましょう。`npm` では `package.json` というファイルを使って、依存するパッケージの種類やバージョンなどの情報を記録します。`npm init` コマンドは、`package.json` ファイルを生成してパッケージ管理環境を初期化するものです。通常は初期化する

ための値を対話式のプロンプトによって設定しますが、`--yes` オプションを付与するとすべてをデフォルト値にします。次のコマンドを実行して、`package.json` を生成します。

```
$ npm init --yes
```

生成された `package.json` ファイルは次のようにになっています。

```
{
  "name": "nodecli",
  "version": "1.0.0",
  "description": "",
  "main": "main.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

`package.json` ファイルが用意できたら、`npm install` コマンドを使って `commander` パッケージをインストールします。このコマンドの引数にはインストールするパッケージの名前とそのバージョンを@記号でつなげて指定できます。バージョンを指定せずにインストールすれば、その時点での最新の安定版が自動的に選択されます。

`package.json` にインストールしたパッケージの情報を保存するためには `--save` オプションを付与する必要があることに注意しましょう。次のコマンドを実行して、`commander` のバージョン 2.9 をインストールします。

```
$ npm install --save commander@2.9
```

インストールが完了すると、`package.json` ファイルは次のようにになっています。

```
{
  "name": "nodecli",
  "version": "1.0.0",
  "description": "",
  "main": "main.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "commander": "^2.9.0"
  }
}
```

また、npm のバージョンが 5 以上であれば `package-lock.json` ファイルが生成されています。このファイルは npm がインストールしたパッケージの、実際のバージョンを記録するためのものです。さきほど `commander` のバージョンは 2.9 としましたが、実際にインストールされるのは 2.9.x に一致する最新のバージョンです。`package-lock.json` ファイルには実際にインストールされたバージョンが記録されています。これによって、ふたたび `npm install` を実行したときに異なるバージョンがインストールされることを防ぎます。

## commander パッケージを使う

`npm install` コマンドでインストールされたパッケージは、`node_modules` というディレクトリの中に配置されています。`node_modules` ディレクトリに配置されたパッケージは、`require` 関数を使ってスクリプト中に読み込みます。`require` 関数は Node.js 環境のグローバル関数のひとつで、指定したパッケージのモジュールを読み込めます。

commanderパッケージを読み込むには、次のように記述します。

```
const program = require("commander");
```

commanderは `parse` メソッドを使ってコマンドライン引数をパースします。パースした後に `opts` メソッドを呼び出すと、定義したオプションと与えられた値をオブジェクトとして取り出することができます。次の `commander-flag.js` では、値をもたないオプションを真偽値にパースしています。

```
const program = require("commander");
program.option("--foo");
program.parse(process.argv);
const options = program.opts();
console.log(options.foo);
```

このスクリプトを次のように実行すると、`--foo` オプションがパースされ、`options.foo` プロパティとして扱えるようになっています。

```
$ node commander-flag.js --foo
true
```

もし、次のようなエラーが表示されたときは、commanderパッケージが `node_modules` ディレクトリ内にないことを示しています。commanderパッケージのインストールに失敗しているので、パッケージのインストールからやり直してみましょう。

```
Error: Cannot find module 'commander'
```

値をもつオプションをパースする場合は、次の `commander-param.js` のように記述します。

```
const program = require("commander");
program.option("--foo <text>");
program.parse(process.argv);
const options = program.opts();
console.log(options.foo);
```

`--foo` オプションに値を与えて実行すれば、文字列が `options.foo` プロパティにセットされていることがわかります。

```
$ node commander-param.js --foo bar
bar
```

このように、`process.argv` 配列を直接扱うよりも、commanderのようなライブラリを使うことで簡単にコマンドライン引数を処理できます。次のセクションからは、こうして受け取ったコマンドライン引数を使って、CLIアプリケーションを作成していきます。

## ファイルを読み込む

前のセクションではコマンドライン引数を受け取って、Node.jsのスクリプト中で利用できるようになりました。このセクションではコマンドライン引数に指定されたファイルを読み込んで、標準出力に表示してみましょう。

### ファイルパスを受け取る

まずは読み込むファイルのパスを受け取ります。前回のセクションでインストールしたcommanderを使って、コマンドライン引数からファイルパスを取得しましょう。オプションでないコマンドライン引数は `program.args` 配列から取得できます。次のように、ファイルパスを取得して標準出力に表示します。

```
const program = require("commander");

program.parse(process.argv);
const filePath = program.args[0];
console.log(filePath);
```

このスクリプトを次のように実行すると、引数に与えたファイルパスがそのまま表示されます。

```
$ node receive-path.js sample.md
sample.md
```

### fsモジュールを使ってファイルを読み込む

取得したファイルパスとともに、ファイルを読み込みます。Node.jsでファイルの読み書きをおこなうには、標準モジュールの [fsモジュール](#) を使います。まずは読み込むためのファイルを作成しましょう。`sample.md` という名前で `receive-path.js` と同じディレクトリに配置します。

```
# sample
```

#### fsモジュール

fsモジュールは、Node.jsでファイルの読み書きをおこなうための基本的な関数を提供するモジュールです。すべての関数は非同期形式と同期形式の両方が提供されています。非同期形式の関数は常にコールバック関数を受け取ります。コールバック関数の第1引数は必ずその処理で発生したエラーオブジェクトになっていて、処理の戻り値などがその後ろの引数につづきます。処理が成功したときには、第1引数は `null` または `undefined` になります。一方、同期形式の関数が処理に失敗したときは例外を発生させるので、try/catch文によって例外処理をおこなうことができます。

次のサンプルコードは非同期形式の関数の例です。

```
const fs = require("fs");

fs.readFile("sample.md", (err, file) => {});
```

そして、次のサンプルコードは同じ関数の同期形式の例です。

```
const fs = require("fs");
```

```
try {
  const file = fs.readFileSync("sample.md");
} catch (err) {
}
```

Node.jsはシングルスレッドなので、ノンブロッキング処理である非同期形式のAPIを選ぶことがほとんどです。Node.jsにはfsモジュール以外にも多くの非同期APIがあるので、非同期処理に慣れておきましょう。

## readFile関数を使う

ファイルを読み込むには、fsモジュールの `readFile` 関数を使います。`readFile` 関数にはいくつかのオーバーロードがあります。次の例では、ファイルパスとコールバック関数だけを渡していますが、このときのコールバック関数の第2引数は `Buffer` インスタンスになります。

```
const program = require("commander");
const fs = require("fs");

program.parse(process.argv);
const filePath = program.args[0];

fs.readFile(filePath, (err, file) => {
  console.log(file);
});
```

`sample.md` を引数に渡した実行結果は次のようにになります。`Buffer` インスタンスはファイルの中身をバイト列として保持していて、そのまま `console.log` 関数に渡しても人間が読める文字列にはなりません。

```
$ node read-file-1a.js sample.md
<Buffer 23 20 73 61 6d 70 6c 65>
```

`Buffer` インスタンスから文字列を取り出すには、`toString` メソッドを使います。`toString` メソッドはオプショナルな引数として文字エンコーディングを受け取れますが、何も指定しなかった場合は自動的にUTF-8として変換されます。次の例ではコールバック関数の中で `toString` メソッドを呼び出して、ファイルの中身をUTF-8の文字列として表示します。

```
const program = require("commander");
const fs = require("fs");

program.parse(process.argv);
const filePath = program.args[0];

fs.readFile(filePath, (err, file) => {
  console.log(file.toString());
});
```

`sample.md` を引数に渡した実行結果は次のようにになります。

```
$ node read-file-1b.js sample.md
# sample
```

ちなみに、次の例のように `readFile` 関数の第2引数で文字エンコーディング形式を指定することもできます。このときのコールバック関数の第2引数は、指定した文字エンコーディングでエンコードされた後の文字列が渡されます。

```
const program = require("commander");
const fs = require("fs");

program.parse(process.argv);
```

```
const filePath = program.args[0];

fs.readFile(filePath, "utf8", (err, file) => {
  console.log(file);
});
```

`sample.md` を引数に渡した実行結果は次のようにになります。

```
$ node read-file-2.js sample.md
# sample
```

## エラーハンドリング

先述のとおり、`fs`モジュールのコールバック関数の第1引数には常にエラーオブジェクトが渡されます。ファイルの読み書きは存在の有無や権限、ファイルシステムの違いなどによって例外が発生しやすいので、必ずエラーハンドリング処理を書きましょう。次の例は `err` オブジェクトが `null` または `undefined` ではないことだけをチェックするシンプルなエラーハンドリングです。エラーが発生していたときには表示し、`process.exit` 関数を使って実行しているプロセスを異常終了させています。

```
const program = require("commander");
const fs = require("fs");

program.parse(process.argv);
const filePath = program.args[0];

fs.readFile(filePath, "utf8", (err, file) => {
  if (err) {
    console.error(err);
    process.exit(err.code);
    return;
  }
  console.log(file);
});
```

存在しないファイルである `notfound.md` を引数に渡すと、次のようにエラーが発生して終了します。

```
$ node read-file-3.js notfound.md
Error: ENOENT: no such file or directory, open 'notfound.md'
```

これでコマンドライン引数に指定したファイルを読み込んで標準出力に表示できました。

# MarkdownをHTMLに変換する

前のセクションではコマンドライン引数で受け取ったファイルを読み込み、標準出力に表示しました。次は読み込んだMarkdownファイルをHTMLに変換して、その結果を標準出力に表示してみましょう。

## markedパッケージを使う

JavaScriptでMarkdownをHTMLへ変換するために、今回は `marked` というライブラリを使用します。`marked` のパッケージはnpmで配布されているので、`commander` と同様に `npm install` コマンドでパッケージをインストールしましょう。

```
$ npm install --save marked
```

インストールが完了したら、`Node.js` のスクリプトから読み込みます。前のセクションの最後で書いたスクリプトに、`marked` パッケージの読み込み処理を追加します。

```
const program = require("commander");
const fs = require("fs");
const marked = require("marked"); // markedパッケージを読み込む

program.parse(process.argv);
const filePath = program.args[0];

fs.readFile(filePath, "utf8", (err, file) => {
  if (err) {
    console.error(err);
    process.exit(err.code);
    return;
  }
  console.log(file);
});
```

`marked` パッケージから取得した `marked` オブジェクトは、Markdown文字列を引数に取りHTML文字列を返す関数です。次のように `readFile` 関数で読み込んだファイルの文字列を引数として渡せば、HTMLに変換できます。

```
const program = require("commander");
const fs = require("fs");
const marked = require("marked");

program.parse(process.argv);
const filePath = program.args[0];

fs.readFile(filePath, "utf8", (err, file) => {
  if (err) {
    console.error(err);
    process.exit(err.code);
    return;
  }
  const html = marked(file); // HTML文字列に変換する
  console.log(html);
});
```

## 変換オプションを作成する

markedにはMarkdownの変換オプションがあり、オプションの設定によって変換後のHTMLが変化します。いくつかのオプションについてアプリケーション中のデフォルトの設定を決め、さらにコマンドライン引数から設定を切り替えられるようにしてみましょう。今回扱うオプションは次の2つです。

- gfm
- sanitize

## gfmオプション

`gfm` オプションは、GitHubにおけるMarkdownの仕様(GitHub Flavored Markdown, GFM)に合わせて変換するかを決めるオプションです。markedのデフォルトでは `true` になっています。GFMは標準的なMarkdownにいくつかの拡張を加えたもので、代表的な拡張がURLの自動リンク化です。例として、次のようなMarkdownファイルを用意し、先ほどのスクリプトと、`gfm` オプションを `false` にしたスクリプトで結果の違いを見てみましょう。

```
# サンプルファイル

これはサンプルです。
https://jsprimer.net/

- サンプル1
- サンプル2
```

`gfm` オプションが有効のときは、URLの文字列が自動的に `<a>` タグのリンクに置き換わります。

```
<h1 id="-">サンプルファイル</h1>
<p>これはサンプルです。
<a href="https://jsprimer.net/">https://jsprimer.net/</a></p>
<ul>
<li>サンプル1</li>
<li>サンプル2</li>
</ul>
```

一方、次のように `gfm` オプションを `false` にすると、単なる文字列として扱われ、リンクには置き換わりません。

```
const program = require("commander");
const fs = require("fs");
const marked = require("marked");

program.parse(process.argv);
const filePath = program.args[0];

fs.readFile(filePath, "utf8", (err, file) => {
  if (err) {
    console.error(err);
    process.exit(err.code);
    return;
  }
  const html = marked(file, {
    gfm: false
  });
  console.log(html);
});
```

```
<h1 id="-">サンプルファイル</h1>
<p>これはサンプルです。
https://jsprimer.net/</p>
<ul>
<li>サンプル1</li>
<li>サンプル2</li>
</ul>
```

自動リンクの他にもいくつかの拡張がありますが、詳しくは[GitHub Flavored Markdown](#)のドキュメンテーションを参照してください。

## sanitizeオプション

`sanitize` オプションは出力されるHTMLを安全な形にサニタイズするためのオプションです。`sanitize` オプションが有効なとき、Markdownファイル中に書かれたHTMLタグはエスケープされ、単なる文字列として出力されます。例として次のようなMarkdownファイルの変換が `sanitize` オプションによってどう変わるかを見てみましょう。

```
# サンプルファイル

これはサンプルです。
https://jsprimer.net/

<p>これはHTMLです</p>

- サンプル1
- サンプル2
```

`sanitize` オプションのデフォルト値は `false` なので、何も指定しなければMarkdownファイル中のHTMLはそのまま出力されるHTML中でもタグとして残ります。

```
<h1 id="-">サンプルファイル</h1>
<p>これはサンプルです。
https://jsprimer.net/</p>
<p>これはHTMLです</p>

<ul>
<li>サンプル1</li>
<li>サンプル2</li>
</ul>
```

ここで次のように `sanitize` オプションを有効にすると、`< と >` がエスケープされてHTMLタグとして機能しなくなります。自由なHTMLを書かれては困る場合に有用なオプションです。

```
const program = require("commander");
const fs = require("fs");
const marked = require("marked");

program.parse(process.argv);
const filePath = program.args[0];

fs.readFile(filePath, "utf8", (err, file) => {
  if (err) {
    console.error(err);
    process.exit(err.code);
    return;
  }
  const html = marked(file, {
    gfm: false,
    sanitize: true
  });
  console.log(html);
});
```

```
<h1 id="-">サンプルファイル</h1>
<p>これはサンプルです。
https://jsprimer.net/</p>
<p>&lt;&gt;これはHTMLです&lt;&gt;</p>
```

```
</p>
<ul>
<li>サンプル1</li>
<li>サンプル2</li>
</ul>
```

## コマンドライン引数からオプションを受け取る

それぞれの変換オプションについて、コマンドライン引数で制御できるようにします。`gfm` オプションは `--gfm`、`sanitize` オプションは `--sanitize` と `-s` でコマンドラインから設定できるようにします。

```
program
  .option("--gfm", "GFMを有効にする")
  .option("-s, --sanitize", "サニタイズを行う");

program.parse(process.argv);
```

## デフォルト設定を定義する

毎回すべての設定を明示的に入力させるのは不便なので、それぞれの変換オプションのデフォルト設定を定義します。今回は `gfm` オプションと `sanitize` オプションをどちらもデフォルトで `false` にします。アプリケーション側でデフォルト設定を持っておくことで、将来的に `marked` の挙動が変わったときにも影響を受けにくくなります。

`marked` のオプションはオブジェクトを渡す形式です。オブジェクトのデフォルト値を明示的な値で上書きするときは `...` (spread構文) を使うと便利です。(オブジェクトのspread構文を参照) 次のようにデフォルトのオプションを表現したオブジェクトに対して、`program.opts` メソッドの戻り値で上書きします。

```
const markedOptions = {
  gfm: false,
  sanitize: false,
  // オプションのkey-valueオブジェクトをマージする
  ...program.opts()
};
```

あとは `markedOptions` オブジェクトから `marked` にオプションを渡すだけです。スクリプト全体は次のようになります。

```
const program = require("commander");
const fs = require("fs");
const marked = require("marked");

program
  .option("--gfm", "GFMを有効にする")
  .option("-s, --sanitize", "サニタイズを行う");

program.parse(process.argv);
const filePath = program.args[0];

const markedOptions = {
  gfm: false,
  sanitize: false,
  ...program.opts()
};

fs.readFile(filePath, "utf8", (err, file) => {
  if (err) {
    console.error(err);
    process.exit(err.code);
    return;
  }
```

```
    });
    const html = marked(file, {
      gfm: markedOptions.gfm,
      sanitize: markedOptions.sanitize
    });
    console.log(html);
});
```

定義したコマンドライン引数を使って、Markdownファイルを変換してみましょう。

```
# gfmオプションを有効にする
$ node main.js --gfm sample.md
# sanitizeオプションを短縮形で有効にする
$ node main.js -S sample.md
```

これでMarkdown変換の設定をコマンドライン引数でオプションとして与えられるようになりました。

# ユニットテストを記述する

このセクションでは、これまで作成したCLIアプリケーションにユニットテストを導入します。ユニットテストの導入とあわせて、ソースコードを整理してテストがしやすくなるようにモジュール化します。

## スクリプトをモジュールに分割する

前のセクションまでは、すべての処理を一つのJavaScriptファイルに記述していました。ユニットテストを行うためにはテスト対象がモジュールとして分割されていなければいけません。今回のアプリケーションでは、CLIアプリケーションとしてコマンドライン引数を処理する部分と、MarkdownをHTMLへ変換する部分に分割します。

Node.jsでは、複数のJavaScriptファイル間で変数や関数などをやりとりするために、モジュールという仕組みを利用します。モジュールとは変数や関数などを外部にエクスポートするJavaScriptファイルのことです。モジュールであるJavaScriptファイルは、別のJavaScriptファイルからインポートできます。モジュールからオブジェクトをエクスポートするには、Node.jsのグローバル変数である`moduleオブジェクト`を利用します。`module.exports`オブジェクトは、そのファイルからエクスポートされるオブジェクトを格納します。次のコードは簡単な関数をエクスポートするモジュールの例です。

```
// greet.js
module.exports = function greet(name) {
  return `Hello ${name}!`;
};
```

`require` 関数は別のJavaScriptファイルをモジュールとしてインポートできます。次の例では先ほどのモジュールをインポートして、エクスポートされた関数を取得しています。

```
const greet = require("./greet");
greet("World"); // => "Hello World!"
```

`module.exports` オブジェクトに直接代入するのではなく、そのプロパティとして任意の値をエクスポートすることもできます。次の例では2つの関数を同じファイルからエクスポートしています。

```
// functions.js
module.exports.foo = function() { /**/ };
module.exports.bar = function() { /**/ };
```

このようにエクスポートされたオブジェクトは、`require` 関数の戻り値のプロパティとしてアクセス可能になります。

```
const functions = require("./functions");
functions.foo();
functions.bar();
```

それではCLIアプリケーションのソースコードをモジュールに分割してみましょう。`md2html.js` という名前のJavaScriptファイルを作成し、次のようにMarkdownの変換処理を記述します。

```
const marked = require("marked");

module.exports = (markdown, options = {}) => {
  const markedOptions = {
    gfm: false,
    sanitize: false,
```

```

        ...options,
    });

    return marked(markdown, {
        gfm: markedOptions.gfm,
        sanitize: markedOptions.sanitize
    });
}

```

このモジュールがエクスポートするのは、与えられたオプションとともにMarkdown文字列をHTMLに変換する関数です。アプリケーションのエントリーポイントである `main.js` では、次のようにこのモジュールをインポートして使用します。

```

const program = require("commander");
const fs = require("fs");
const md2html = require("./md2html");

program
    .option("-gfm", "GFMを有効にする")
    .option("-S, --sanitize", "サニタイズを行う");

program.parse(process.argv);
const filePath = program.args[0];

fs.readFile(filePath, "utf8", (err, file) => {
    if (err) {
        console.error(err);
        process.exit(err.code);
        return;
    }
    const html = md2html(file, program.opts());
    console.log(html);
});

```

`marked` パッケージや、そのオプションに関する記述がひとつの `md2html` 関数に隠蔽され、`main.js` がシンプルになりました。そして `md2html.js` はアプリケーションから独立したひとつのモジュールとして切り出され、ユニットテストが可能になりました。

## ユニットテスト実行環境を作る

ユニットテストの実行にはさまざまな方法がありますが、このセクションではテスティングフレームワークとして [Mocha](#) を使って、ユニットテストの実行環境を作成します。Mochaが提供するテスト実行環境では、グローバルに `it` や `describe` などの関数が定義されます。`it` 関数はその内部でエラーが発生したとき、そのテストを失敗として扱います。つまり、期待する結果と異なるならエラーを投げ、期待どおりならエラーを投げないというテストコードを書くことになります。

テストコード中でエラーを投げるために、今回はNode.jsの標準モジュールのひとつである [assertモジュール](#) から提供される `assert` 関数を利用します。`assert` 関数は引数の評価結果が `false` であるとき、実行時にエラーを投げます。

Mochaによるテスト環境を作るために、まずは次のコマンドで `mocha` パッケージをインストールします。

```
$ npm install --save-dev mocha
```

`--save-dev` オプションは、パッケージを `devDependencies` としてインストールするためのものです。`package.json` の `devDependencies` には、そのパッケージを開発するときだけ必要な依存ライブラリを記述します。

ユニットテストを実行するには、Mochaが提供する `mocha` コマンドを使います。Mochaをインストールした後、`package.json` の `scripts` プロパティを次のように記述します。

```
{
  ...
  "scripts": {
    "test": "mocha"
  },
  ...
}
```

この記述により、`npm test` コマンドを実行したときに `mocha` コマンドが呼び出されます。試しに `npm test` コマンドを実行し、Mochaによるテストが行われることを確認しましょう。まだ何もテストを書いていないので、`0 passing` と表示されます。

```
$ npm test
> mocha

0 passing (2ms)
```

## ユニットテストを記述する

テストの実行環境ができたので、実際にユニットテストを記述します。Mochaのユニットテストは `test` ディレクトリの中にJavaScriptファイルを配置して記述します。`test/md2html-test.js` ファイルを作成し、`md2html.js` に対するユニットテストを次のように記述します。

```
const assert = require("assert");
const fs = require("fs");
const path = require("path");
const md2html = require("../md2html");

it("converts Markdown to HTML", () => {
  const sample = fs.readFileSync(path.resolve(__dirname, "./fixtures/sample.md"), "utf8");
  const expected = fs.readFileSync(path.resolve(__dirname, "./fixtures/expected.html"), "utf8");

  assert(md2html(sample) === expected);
});
```

`it` 関数で定義したユニットテストは、`md2html` 関数の変換結果が期待するものになっているかをテストしています。`test/fixtures` ディレクトリにはユニットテストで用いるファイルを配置しています。今回は変換元のMarkdownファイルと、期待する変換結果のHTMLファイルの2つが存在します。

ユニットテストを記述したら、もう一度改めて `npm test` コマンドを実行しましょう。1件のテストが通れば成功です。

```
$ npm test
> mocha

✓ converts Markdown to HTML
1 passing (18ms)
```

## なぜユニットテストをおこなうのか

ユニットテストを実施することには多くの利点があります。早期にバグが発見できることや、安心してリファクタリングをおこなえるようになるのはもちろんですが、ユニットテストが可能な状態を保つこと自体に意味があります。実際にテストをおこなわなくてもテストしやすいコードになるよう心がけることが、アプリケーションを適切にモジュール化する指針になります。

またユニットテストには生きたドキュメントとしての側面もあります。ドキュメントはこまめにメンテナンスされないとすぐに実際のコードと齟齬が生まれてしまいますが、ユニットテストはそのモジュールが満たすべき仕様を表すドキュメントとして機能します。

ユニットテストの記述は手間がかかるだけのようにも思えますが、中長期的にアプリケーションをメンテナンスする場合にはかかせないものです。そしてよいテストを書くためには、日頃からテストを書く習慣をつけておくことが重要です。

## まとめ

このユースケースの目標であるNode.jsを使ったCLIアプリケーションの作成と、ユニットテストの導入ができました。`npm`を使ったパッケージ管理や外部モジュールの利用、`fs`モジュールを使ったファイル操作など、多くの要素が登場しました。これらはNode.jsアプリケーション開発においてほとんどのユースケースで応用されるものなので、よく理解しておきましょう。

## ユースケース: Todoアプリ

ここではブラウザで動作するウェブアプリケーション（以下アプリ）のユースケースとして、Todoアプリを作成していきます。ここで作成するTodoアプリは、タスクを入力しそのタスクの完了状態をチェックボックスで管理するというアプリです。

Ajax通信のユースケースではGitHubからデータを取得し表示するだけであったため状態を管理する部分は殆どありませんでした。しかし、このTodoアプリはタスクの状態管理をするためアプリとしての状態を管理する必要があります。このユースケースを通して、どのように状態を管理し、表示や処理を変更するかといったアプリに作るにあたり必要になる設計や考え方についてを見ていきます。

作成するアプリは次の要件を満たすものとします。

- Todoアイテムを追加できる
- Todoアイテムの完了状態を更新できる
- Todoアイテムを削除できる

## エントリポイント

エントリポイントとは、アプリケーションの中で一番最初に呼び出される部分のことです。

[Ajax通信:エントリポイント](#)のユースケースでは、エントリポイントはHTML（`index.html`）のみでした。まずHTMLが読み込まれ、次にHTMLの中に書かれているJavaScriptファイルが読み込まれます。

今回のTodoアプリは処理をモジュール化し、それぞれのモジュールを別々のJavaScriptファイルとして作成していきます。JavaScriptモジュールはHTMLから`<script type="module">`で読み込むことができますが、`script`タグ毎に別々のモジュールスコープを持ちます。そのため、JavaScriptモジュールを別々の`script`タグで読み込むとモジュール同士でスコープが異なるため、モジュール同士で連携できません。

次のコードは、それぞれの`<script type="module">`同士のスコープが異なるため、別の`script`タグで定義した変数にアクセス出来ないことを示しています。これはJavaScriptのコードをファイルにして`src`属性で読み込んだ場合も同様です。

```
<script type="module">
  export const scopeA = "A";
</script>
<script type="module">
  // 異なるmoduleスコープの変数には直接はアクセスできない
  console.log(scopeA); // => ReferenceError: scopeA is not defined
</script>
```

そのため、HTMLから読み込むのは1つのJavaScriptファイル(`index.js`)として、この`index.js`から他のモジュールを読み込み利用します。このようにすることでモジュール間は1つの`<script type="module">`のスコープ内に収まるため、モジュール同士で連携できます。このHTMLから読み込むJavaScriptファイル（`index.js`）をJavaScriptにおけるエントリポイントとします。

つまり、今回作成するTodoアプリではエントリポイントとしてHTMLとJavaScriptの2つを用意します。

- `index.html`：もっとも最初に読み込まれるファイル、`index.js`を読み込む
- `index.js`：`index.html`から読み込まれるファイル、JavaScript間においては最初に読み込まれる

このセクションでは、この2つのエントリポイントを作成し読み込むところまでを確認します。

## プロジェクトディレクトリを作成

今回作成するアプリにはHTMLやCSS、JavaScriptなど複数のファイル必要となります。そのため、まずそれらを配置するディレクトリを作成します。

任意の名前で問題ありませんが、ここでは`todoapp`という名前のディレクトリを作成します。

## HTMLファイルの用意

エントリポイントとして、まずは最低限の要素だけを配置したHTMLファイルを作成しましょう。エントリポイントとなるHTMLとして`index.html`を作成し、次のような内容にします。`body`要素の一番下で`<script>`タグを使い読み込んでいる`index.js`が、今回のアプリケーションの処理を記述するJavaScriptファイルです。

```
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Todo App</title>
```

```
</head>
<body>
  <h1>Todo App</h1>
  <script type="module" src="index.js"></script>
</body>
</html>
```

`index.js` には、スクリプトが正しく読み込まれたことを確認できるように、コンソールにログを出力する処理だけを書いておきます。

```
console.log("index.js: loaded");
```

次はこのHTMLをブラウザで開きコンソールにログが出力されることを確認していきます。

## ローカルサーバでHTMLを確認する

ウェブブラウザで `index.html` を開くために開発用のローカルサーバを準備します。ローカルサーバーを立ち上げずに直接HTMLファイルを開くこともできますが、その場合 `file:///` から始まるURLになります。`file` スキーマでは[Same Origin Policy](#)により、JavaScriptモジュールを始め多くのAPIに制限がありアプリケーションは正しく動作しません。本章はローカルサーバーを立ち上げた上で、`http` スキーマのURLでアクセスすることを前提としています。

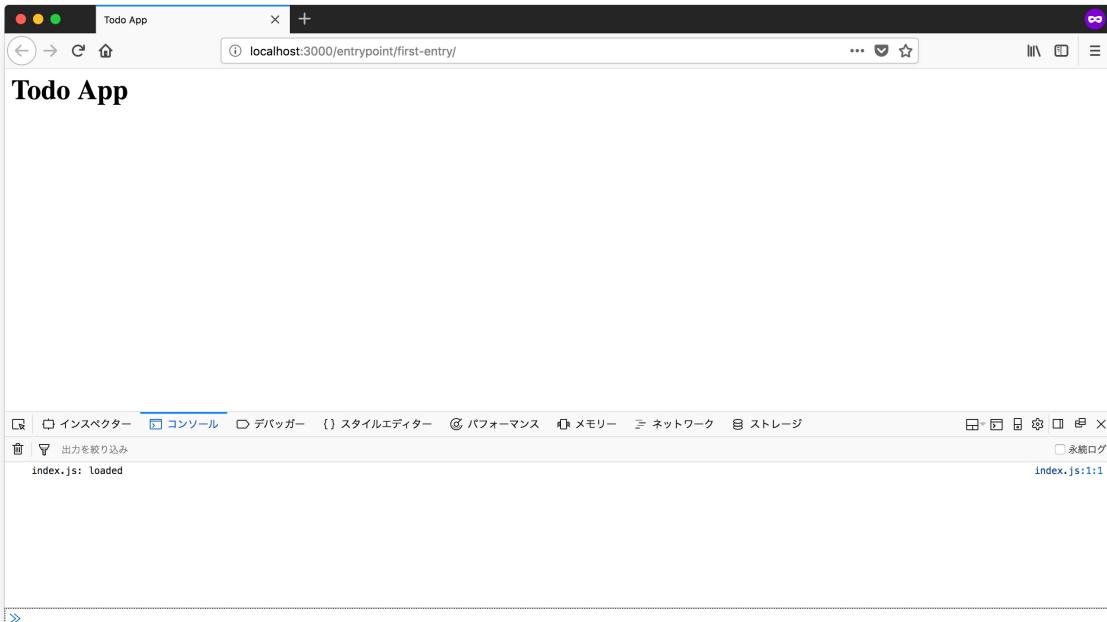
コマンドラインで `todoapp` ディレクトリへ移動し、次のコマンドでローカルサーバを起動します。`npx` コマンドを使い、この書籍用に作成された `@js-primer/local-server` というローカルサーバモジュールをダウンロードと同時に実行します。まだ `npx` コマンドの用意できていなければ、先に[アプリケーション開発の準備](#)を参照してください。

```
# todoapp/ディレクトリに移動する
$ cd todoapp/
# todoapp/をルートにしたローカルサーバを起動する
$ npx @js-primer/local-server

todoappのローカルサーバを起動しました。
次のURLをブラウザで開いてください。

URL: http://localhost:3000
```

起動したローカルサーバのURL（`http://localhost:3000`）へブラウザでアクセスしてみましょう。ブラウザには `index.html` の内容が表示され、開発者ツールのコンソールに `index.js: loaded` というログが出力されていることが確認できます。



## 開発者ツールでのコンソールログの確認方法

Console APIで出力したログを確認するには、ウェブブラウザの開発者ツールを開く必要があります。ほとんどのブラウザで開発者ツールが同梱されていますが、本章ではFirefoxを使って確認します。開発者ツールのコンソールタブを開くとConsole APIで出力したログを確認できます。

Firefoxの開発者ツールは次のいずれかの方法で開きます。

- Firefox メニュー（メニューバーがある場合やmacOSでは、ツールメニュー）の Web 開発サブメニューで "Web コンソール" を選択する
- キーボードショートカット `ctrl+Shift+K` (macOSでは `Command+Option+K`) を押下する

詳細は["Webコンソールを開く"](#)を参照してください。

### [エラー例] コンソールログが表示されない

HTMLは表示されるがコンソールログに `index.js: loaded` が表示されない場合は次のような問題であるかを確認して見てください。

- `index.js` の読み込みに失敗している
- JavaScriptモジュールに非対応のブラウザを利用している

`index.js` の読み込みに失敗している

scirptタグに指定した `index.js` のパスにファイルが存在しているかを確認してください。`<script type="module" src="index.js">` とした場合は `index.html` と `index.js` は同じディレクトリに配置する必要があります。

また、**CORS policy Invalid**のようなエラーがコンソールに表示されている場合は、[Same Origin Policy](#)により `index.js` の読み込みが失敗しています。先ほども書いたように、`file:` から始まるページ上からはJavaScriptモジュールを読み込めないブラウザもあります。そのため、ローカルサーバを起動し、ローカルサーバ(`http:` から始まるURL)にアクセスしていることを確認してください。

JavaScriptモジュールに非対応のブラウザを利用している

JavaScriptモジュールはまだ新しい機能であるため、バージョンが60以上のFirefoxが必要です。バージョンが60未満のFirefoxではモジュールである `index.js` が読み込めないためコンソールログは出力されません。

今回のTodoアプリでは、ネイティブでJavaScriptモジュールに対応しているブラウザが必要です。[Can I Use](#)にてネイティブでJavaScriptモジュールに対応しているブラウザがまとめられています。非対応のブラウザでもBundlerと呼ばれるツールを使うことで対応できますが、本章では省略します。

## モジュールのエントリーポイントの作成

最後にエントリーポイントとなる `index.js` から別のJavaScriptファイルをモジュールとして読み込んで見ましょう。このアプリではJavaScriptモジュールが複数登場するため `src/` というディレクトリを作り、`src/` の下にJavaScriptモジュールを書くことにします。今回は `src/App.js` にファイルを作成し、これを `index.js` からモジュールとして読み込みます。

最終的に現在の `todoapp` ディレクトリは次のような構造になります。

```
todoapp
├── index.html
├── index.js
└── node_modules
├── package.json
└── src
    └── App.js
```

`src/App.js` にファイルを作成し、次のような内容のJavaScriptモジュールとします。モジュールは、基本的には何かしらを外部に公開（`export`）します。`App.js` は `App` というクラスを公開するモジュールとして、今回はコンソールログを出力するだけです。

```
console.log("App.js: loaded");
export class App {
  constructor() {
    console.log("App initialized");
  }
}
```

次に、この `src/App.js` を `index.js` から取り込み（`import`）します。`index.js` を次のように書き換え、`App.js` から `App` クラスを取り込みインスタンス化します。

```
import { App } from "./src/App.js";
const app = new App();
```

再度ローカルサーバのURL（`http://127.0.0.1:3030`）にブラウザでアクセスし、リロードしてみましょう。コンソールログには、次のように処理の順番どおりのログが表示されます。

```
App.js: loaded
App initialized
```

まず `index.js` から `src/App.js` の `App` クラスが取り込まれています。次に `App` クラスがインスタンス化されていることがログから確認できます。

これでHTMLとJavaScriptそれぞれのエントリーポイントの作成と動作を確認できました。

### [エラー例] App.jsの読み込みに失敗する

ディレクトリ構造や `import` 宣言で指定したファイルパスが異なると、ファイルを読み込むことができずにエラーとなってしまいます。この場合は開発者ツールを開き、コンソールにエラーが出ていないかを確認して見てください。

`import` 宣言を使ったJavaScriptモジュール読み込むで起きる典型的なエラーと対処を次にまとめています。

SyntaxError: import declarations may only appear at top level of a module

「`import` 宣言はモジュールのトップレベルでしか利用できません」というエラーがでています。このエラーがでているということは、`import` 宣言を使える条件を満たしていないということです。つまり、`import` 宣言がトップレベルではない所に書かれている、またはモジュールではない実行コンテキストで実行されているということです。

`import` 宣言がトップレベルではない所に書かれているというのは、関数の中などに `import` 宣言を行っています。この場合は `import` 宣言をトップレベル（ファイルの直下）に移動させてみてください。

モジュールではない実行コンテキストで実行されているというのは、裏を返せば実行コンテキストがScriptとなっているということです。JavaScriptには実行コンテキストとしてScriptとModuleがあります。`import` 宣言は実行コンテキストがModuleでないと利用できません。そのため、scriptタグの `type` 指定を忘れていないかをチェックしてみてください。

実行コンテキストをモジュールとして実行するには `<script type="module" src="index.js">` のように `type=module` を指定する必要があります。（`index.js` から `import` 宣言で読み込んだ `App.js` は実行コンテキストを引き継ぐため、モジュールの実行コンテキストで処理されます。）

モジュールのソース “<http://localhost:3000/src/App>” の読み込みに失敗しました。

`App.js` が読み込めていないというエラーがでています。エラーメッセージをよく見ると `App` となっていて `App.js` ではありません。

`import` 宣言では、読み込むファイルの拡張子を省略しません。そのため、`App` のように拡張子（`.js`）を省略して書いている場合はこのエラーが発生します。

```
// エラーとなる例
import { App } from "./src/App";
```

正しくは次のように拡張子まで含めたパスを記述します。また指定したパス（`./src/App.js`）にファイルが存在するかを確認してください。

```
// 正しい例
import { App } from "./src/App.js";
```

## まとめ

このセクションでは次のことを行いました。

- todoappという名前のプロジェクトディレクトリを作成した
- エントリポイントとなるindex.htmlを作成した
- JavaScriptのエントリーポイントとなるindex.jsを作成しindex.htmlから読み込んだ
- ローカルサーバを使ってindex.htmlが表示できた
- src/App.jsを作成し、index.jsからimport文で読み込めるのを確認した

現在のTodoアプリは次のURLで確認できます。

- <https://jsprimer.net/use-case/todoapp/entrypoint/module-entry/>



# アプリの構成要素

HTMLとJavaScriptのエントリーポイントを作成しましたが、次はこのTodoアプリの構成要素をあらためて見ていきましょう。

Todoアプリは、次のような機能を実装していきます。追加、更新、削除、現在の状態の表示など複数の機能を持っています。

- Todoアイテムを追加する
- Todoアイテムを更新する
- Todoアイテムを削除する
- Todoアイテム数（合計）の表示

また、アプリと呼ぶからには見た目もちょっとしたものにしないと雰囲気が出ません。このセクションでは、多くのウェブアプリケーションを構成するHTML、CSS、JavaScriptの役割について見てきます。このセクションで見た目だけで機能がないハリボテのTodoアプリは完成させ、次のセクションから実際にJavaScriptを使ってTodoアプリを実装していきます。

## HTMLとCSSとJavaScript

Todoアプリはブラウザで実行する向けに書いていきますが、ウェブアプリを作成するにはHTMLやCSS、JavaScriptを組み合わせて書いていきます。

- HTML: コンテンツの構造を記述するためのマークアップ言語
- CSS: HTMLの見た目を装飾するスタイルシート言語
- JavaScript: インタラクションといった動作を扱うプログラミング言語

多くのウェブアプリケーションはHTMLでコンテンツの構造を定義し、CSSで見た目を装飾し、JavaScriptで動作を付けることで実装されます。JavaScriptは動的にHTMLやCSSを定義できるため、役割が明確に見えない場合もあります。しかし、HTML、CSS、JavaScriptを組み合わせてアプリが作られていることには違いありません。

一方、ブラウザにはiOSやAndroidのようにOSが提供するようなUIフレームワークの標準はありません。そのためユーザーの実装したさまざまなUIフレームワークがあります。それらのフレームワークを使うかや自分でアプリごとに定義してユーザーインターフェースを作成します。これはTodoアプリという題材をとってみても、書き方が人によって全く異なる場合があります。

今回のTodoアプリは特別なフレームワークを使わずに、そのままのHTML、CSS、JavaScriptを組み合わせて書いていきます。

## Todoアプリの構造をHTMLで定義する

最初に今回作成するTodoアプリのHTMLの構造を定義しています。ここで定義したHTMLとCSSは最後までこの形のまま利用します。次のセクションから変更していくのはJavaScriptだけということになります。

エントリーポイントのセクションで作成した todoapp ディレクトリの `index.html` を次の内容に変更します。

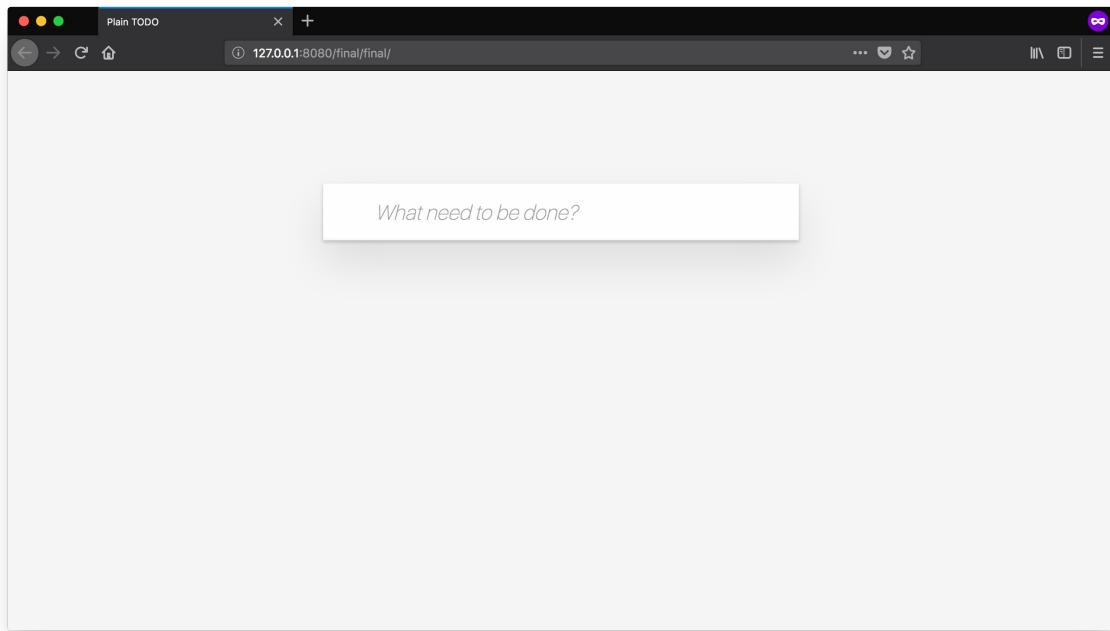
```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Todo App</title>
  <!-- 1. CSSファイルを読み込み -->
  <link href="https://jsprimer.net/use-case/todoapp/final/final/index.css" rel="stylesheet" />
</head>
```

```

<body>
  <!-- 2. class属性をCSSのために指定 -->
  <div class="todoapp">
    <!-- 3. id属性をJavaScriptのために指定 -->
    <form id="js-form">
      <input id="js-form-input" class="new-todo" type="text" placeholder="What need to be done?" autocomplete="off">
    </form>
    <!-- 4. TodoアプリのメインとなるTodoリスト -->
    <div id="js-todo-list" class="todo-list">
      <!-- 動的に更新されるTodoリスト -->
    </div>
    <footer class="footer">
      <!-- 5. Todoアイテム数の表示 -->
      <span id="js-todo-count">Todoアイテム数: 0</span>
    </footer>
  </div>
  <script src="./index.js" type="module"></script>
</body>
</html>

```

HTMLの内容を変更後にブラウザでアクセスすると次のような表示になります。この段階では動作となるJavaScriptは定義していませんが、見た目だけのTodoアプリはこれで完成です。



実際に変更したHTMLを上から順番に見てみましょう。

## 1. CSSファイルを読み込み

`head`要素の中で `link` タグを使い、外部のCSSファイルを読み込んでいます。今回読み込んでいるCSSファイルには、Todoアプリらしい表示に必要なCSSを定義したファイルになっています。

- <https://jsprimer.net/use-case/todoapp/final/final/index.css>

このCSSは動作には影響がないため、今回のユースケースでは外部ファイルをそのまま取り込むだけにし解説は省略します。CSSに定義したスタイルを正しく適応するには、`class`属性やHTML要素の構造が一致している必要があります。表示が崩れている場合は、`class`属性が正しいかやHTMLの構造が同じになっているかを確認して見てください。

## 2. class属性をCSSのために指定

`div` タグの `class` 属性に `todoapp` という値（クラス名）を設定しています。 `class` 属性は基本的にはCSSから装飾するための目印として利用されます。また、1ページの中で同じクラス名は複数の要素に対して設定できます。HTMLの `class` 属性はJavaScriptの `class` 構文とは無関係なことには注意が必要です。

今回の `todoapp` というクラス名をもつ要素を、CSSから `.todoapp` というCSSセレクタで指定できます。[CSSセレクタ](#)とは要素を指定するために利用できる表現をまとめたパターンです。今回の特定の「クラス名」をもつ要素の場合は `.クラス名`（クラス名の前にドット）で選択できます。

次のCSSコードでは、`todoapp` というクラス名をもつ要素の `background` プロパティの値を `black` にしています。つまり `todoapp` クラス名の要素の背景色を黒色にするという装飾をするということです。

```
.todoapp {
  background: black;
}
```

CSSセレクタではタグ名、`id` 属性や構造などに対する指定もできます。たとえば、特定の「`id名`」をもつ要素の場合は `#id名` で選択できます。

```
#id名 {
  /* CSSプロパティで装飾する */
}
```

## 3. id属性をJavaScriptのために指定

`id` 属性はその要素に対するユニークな識別子を付けるための属性です。`id` 属性はCSS、JavaScript、リンクのアンカーなどさまざまな用途で利用されます。また1ページの中で同じ`id`属性名を複数の要素に対して設定できません。

今回のTodoアプリではJavaScriptから要素を選択するために `id` 属性を設定しています。先ほどのCSSセレクタはCSSから要素を指定するだけではなく、JavaScriptから要素を指定する際にも利用できます。ブラウザのDOM APIの `document.querySelector` APIではCSSセレクタを使い要素を選択できます。

次のコードでは、`document.querySelector("cssセレクタ")` を利用し、特定の`id`属性名の要素を取得しています。

```
// id属性の値が"js-form"である要素を取得する
const form = document.querySelector("#js-form");
```

そのため、JavaScriptで参照する要素には `id` 属性を目印として付けています。それぞれの属性に `js-` というプロフィックスをついていることからも分かるように、JavaScriptから扱う部分には `js-` から始まる名前にしています。`id` 属性は同じページに1つしか存在できないため、この `id` 属性はJavaScriptのために付けているということを分かりやすくするための慣習です。

## 4. TodoアプリのメインとなるTodoリスト

`js-todo-list` という `id` 属性を付けた `div` 要素が今回のTodoアプリのメインとなるTodoリストです。この `div` 要素の中身はJavaScriptで動的に更新されるため、HTMLでは目印となる `id` 属性を付けています。

初期表示時はTodoリストの中身がまだ空であるため、何も表示されていません。（HTMLコメントは見た目には表示されません）

## 5. Todoアイテム数の表示

`js-todo-count` という `id` 属性を付けた `span` 要素は、現在のTodoリストのアイテム数を表示します。初期表示時は Todoリストが空であるため0コとなりますが、Todoアイテムを追加や削除する際には合わせて更新する必要があります。

## まとめ

このセクションではHTMLでアプリの構造を定義し、CSSでアプリのスタイルを定義しました。次のセクションではJavaScriptモジュールを作成していき、現在は空であるTodoリストを更新していきます。

現在のTodoアプリのディレクトリ構造は次のようにになります。

```
todoapp
├── index.html(今回の変更点)
├── index.js
├── node_modules
├── package.json
└── src
    └── App.js
```

現在のTodoアプリは次のURLで実際に確認できます。

- <https://jsprimer.net/use-case/todoapp/app-structure/todo-html/>

## フォームとイベント

ここからはJavaScriptでTodoアプリの動作を実際に作っていきます。

このセクションでは、前のセクションでHTMLに目印を付けたTodoリスト（`#js-todo-list`）に対してTodoアイテムを追加する処理を作っていきます。

### Todoアイテムの追加

ユーザーが次のような操作を行い、Todoアイテムを追加します。

1. 入力欄にTodoアイテムのタイトルを入力する
2. 入力欄でEnterを押し送信する
3. TodoリストにTodoアイテムが追加される

これをJavaScriptで実現するには次のことが必要です。

- `form`要素から送信（`submit`）されたことをイベントで受け取る
- `input`要素（入力欄）に入力された内容を取得する
- 入力内容をタイトルにしたTodoアイテムを作成し、Todoリスト（`#js-todo-list`）にTodoアイテム要素を追加する

まずは、`form`要素から送信されたイベントを受け取り、入力内容をコンソールログに表示してみることから始めてみましょう。

### 入力内容をコンソールに表示

`form`要素でEnterを押し送信すると `submit` イベントが発生します。この `submit` イベントは `addEventListener` メソッドを利用することで受け取れます。

```
// id="js-form`の要素を取得
const formElement = document.querySelector("#js-form");
// form要素から発生したsubmitイベントを受け取る
formElement.addEventListener("submit", (event) => {
  // イベントが発生した時に呼ばれるコールバック関数
});
```

フォームが送信されたときに入力内容をコンソールに表示するには、`addEventListener` コールバック関数内で入力内容をConsole APIで出力すればよいことになります。

入力内容は`input`要素の `value` プロパティから取得できます。

```
const inputElement = document.querySelector("#js-form-input");
console.log(inputElement.value); // => "input要素の入力内容"
```

これらを組み合わせて `App.js` に「入力内容をコンソールに表示」する機能を実装してみましょう。`App` クラスに `mount` というメソッドを定義して、その中に処理を書いていきましょう。

次のようにフォーム（`#js-form`）をEnterで送信すると、`input`要素（`#js-form-input`）の内容が開発者ツールのコンソールに表示されるという実装を行います。

src/App.js

```
export class App {
```

```

mount() {
  const formElement = document.querySelector("#js-form");
  const inputElement = document.querySelector("#js-form-input");
  formElement.addEventListener("submit", (event) => {
    // submitイベントの本来の動作を止める
    event.preventDefault();
    console.log(`入力欄の値: ${inputElement.value}`);
  });
}

```

このままでは、`App#mount` は呼び出されないため何も行われません。そのため、`index.js` も変更して、`App` クラスの `mount` メソッドを呼び出すようにします。

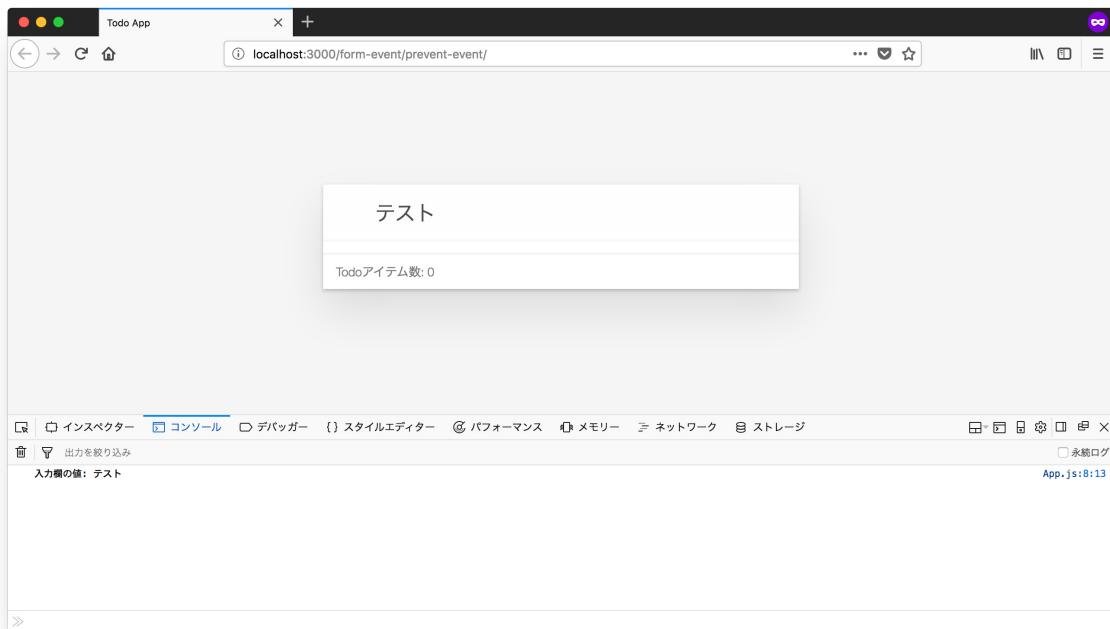
`index.js`

```

import { App } from "./src/App.js";
const app = new App();
app.mount();

```

これらの変更後にブラウザでページをリロードすると、`App#mount` が実行されるようになります。`submit` イベントがリッスンされているので、入力欄に何か入力してEnterで送信してみるとその内容がコンソールに表示されます。



先ほどの `App#mount` では、`submit` イベントのイベントリスナー内で `event.preventDefault` メソッドを呼び出しています。`event.preventDefault` メソッドは、`submit` イベントの発生元であるフォームがもつデフォルトの動作をキャンセルするメソッドです。

フォームがもつデフォルトの動作とは、フォームの内容を指定したURLへ送信するという動作です。ここでは `form` 要素に送信先が指定されていないため、現在のURLに対してフォームを送信が行われます。`event.preventDefault` メソッドを呼び出すことで、このデフォルトの動作をキャンセルしています。

```

formElement.addEventListener("submit", (event) => {
  // submitイベントの本来の動作を止める
  event.preventDefault();
  console.log(`入力欄の値: ${inputElement.value}`);
});

```

現在のURLに対してフォームを送信が行われると、結果的にページがリロードされてしまうため、`event.preventDefault()`を呼び出していました。これは`event.preventDefault()`をコメントアウトすると、ページがリロードされてしまうことが確認できます。

```
formElement.addEventListener("submit", (event) => {
  // preventDefaultしないとページがリロードされてしまう
  // event.preventDefault();
  console.log(`入力欄の値: ${inputElement.value}`);
});
```

ここまでで todoapp ディレクトリは次のような変更を加えました。

```
todoapp
└── index.html
└── index.js (App#mountの呼び出し)
└── package.json
└── src
    └── App.js (App#mountの実装)
```

ここまでTodoアプリは次のURLで実際に確認できます。

<https://jsprimer.net/use-case/todoapp/form-event/prevent-event/>

## 入力内容をTodoリストに表示

フォーム送信時に入力内容を取得する方法が分かったので、次はその入力内容をTodoリスト(`#js-todo-list`)に表示します。

HTMLではリストのアイテムを記述する際には`<li>`タグを使います。また後ほどTodoリストに表示するTodoアイテムの要素には、完了状態を表すチェックボックスや削除ボタンなども含めたいです。これらの要素を含むものを手続き的にDOM APIで作成すると見通しが悪くなるため、HTML文字列からHTML要素を生成するユーティリティモジュールを作成しましょう。

次の`html-util.js`を`src/view/html-util.js`というパスに作成します。

この`html-util.js`は「[ajaxapp: HTML文字列をDOMに追加する](#)」でも利用した`escapeSpecialChars`をベースにしています。ajaxappでの`escapeHTML`タグ関数では出力はHTML文字列でしたが、今回作成する`element`タグ関数の出力はHTML要素(Element)です。

これはTodoリスト(`#js-todo-list`)というすでに存在する要素に対して要素を追加するには、HTML文字列ではなく要素が必要になります。また、HTML文字列に対しては`addEventListener`でイベントをリッスンできません。そのため、チェックボックスの状態が変わったことや削除ボタンが押されたことを知る必要があるTodoアプリでは要素が必要になります。

`src/view/html-util.js`

```
export function escapeSpecialChars(str) {
  return str
    .replace(/&/g, "&amp;")
    .replace(/</g, "&lt;")
    .replace(/>/g, "&gt;")
    .replace(/\"/g, "&quot;")
    .replace(/\'/g, "&#039;");
}

export function htmlToElement(html) {
  const template = document.createElement("template");
```

```

        template.innerHTML = html;
        return template.content.firstChild;
    }

    /**
     * HTML文字列からDOM Nodeを作成して返す
     * @return {Element}
     */
    export function element(strings, ...values) {
        const htmlString = strings.reduce((result, string, i) => {
            const value = values[i - 1];
            if (typeof value === "string") {
                return result + escapeSpecialChars(value) + string;
            } else {
                return result + String(value) + string;
            }
        });
        return htmlToElement(htmlString);
    }

    /**
     * コンテナ要素の中身をbodyElementで上書きする
     * @param {Element} bodyElement コンテナ要素の中身となる要素
     * @param {Element} containerElement コンテナ要素
     */
    export function render(bodyElement, containerElement) {
        // rootElementの中身を空にする
        containerElement.innerHTML = "";
        // rootElementの直下にbodyElementを追加する
        containerElement.appendChild(bodyElement);
    }

```

`element` タグ関数では、同じファイルに定義した `htmlToElement` 関数を使ってHTML文字列からHTML要素を作成しています。`htmlToElement` 関数の中で利用している`template要素`はHTML5で追加された、HTML文字列の断片からHTML要素を作成できる要素です。

この `element` タグ関数を使うことで、次のようにHTML文字列からHTML要素を作成できます。作成した要素は、`appendChild` メソッドなどで既存の要素に子要素として追加できます。

```

// HTML文字列からHTML要素を作成
const newElement = element`<ul>
  <li>新しい要素</li>
</ul>`;
// 作成した要素を`document.body`の子要素として追加（appendChild）する
document.body.appendChild(newElement);

```

ブラウザが提供する `appendChild` メソッドは子要素を追加するだけであるため、すでに別の要素がある場合は末尾に追加されます。

このセクションではまだ利用しませんが、`html-util.js` には `render` という関数を定義しています。`render` 関数は指定したコンテナ要素（親となる要素）の子要素を上書きする関数となります。動作的には一度子要素をすべて消したあとに `appendChild` で子要素として追加しています。

```

// `ul`要素の空タグを作成
const newElement = element`<ul />`;
// `newElement`を`document.body`の子要素として追加する
// 既に`document.body`以下に要素がある場合は上書きされる
render(newElement, document.body);

```

最後に、この `element` タグ関数を使い、フォームから送信された入力内容をTodoリストに要素として追加してみます。

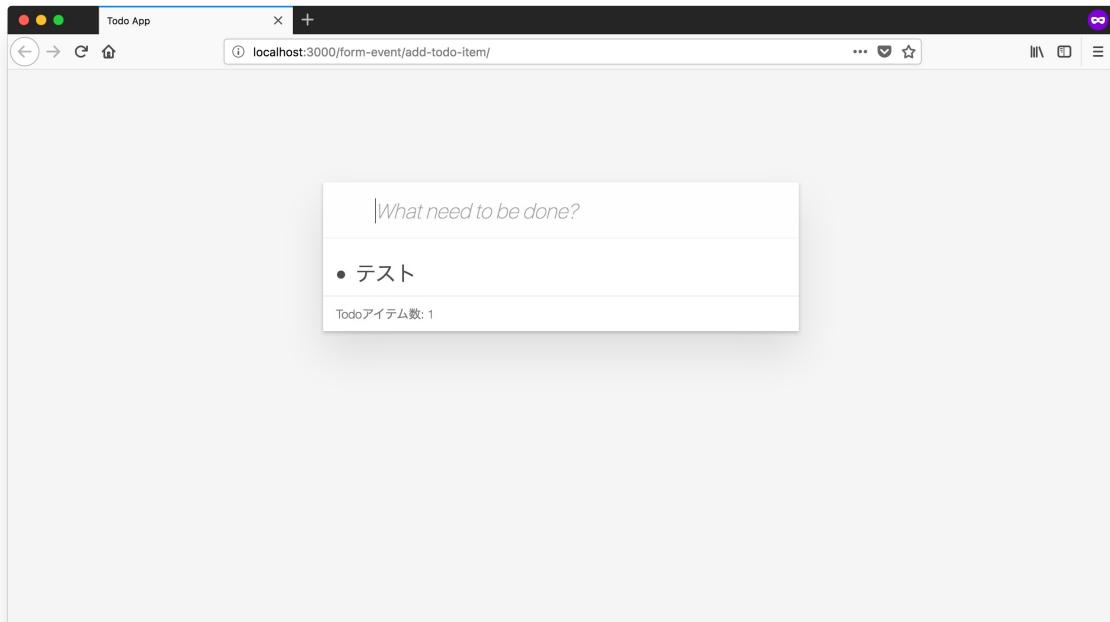
`App.js` から先ほど作成した `html-util.js` の `element` タグ関数を `import` します。次に `submit` イベントのリスナー関数で、Todoアイテムを表現する要素を作成し、Todoリスト(`#js-todo-list`)の子要素として追加(`appendChild`)します。最後にTodoアイテム数(`#js-todo-count`)のテキスト(`textContent`)を更新します。

src/App.js

```
import { element } from "./view/html-util.js";

export class App {
  mount() {
    const formElement = document.querySelector("#js-form");
    const inputElement = document.querySelector("#js-form-input");
    const containerElement = document.querySelector("#js-todo-list");
    const todoItemCountElement = document.querySelector("#js-todo-count");
    // Todoアイテム数
    let todoItemCount = 0;
    formElement.addEventListener("submit", (event) => {
      // 本来のsubmitイベントの動作を止める
      event.preventDefault();
      // 追加するTodoアイテムの要素(li要素)を作成する
      const todoItemElement = element`<li>${inputElement.value}</li>`;
      // Todoアイテムをcontainerに追加する
      containerElement.appendChild(todoItemElement);
      // Todoアイテム数を+1し、表示されてるテキストを更新する
      todoItemCount += 1;
      todoItemCountElement.textContent = `Todoアイテム数: ${todoItemCount}`;
      // 入力欄を空文字にしてリセットする
      inputElement.value = "";
    });
  }
}
```

これらの変更後にブラウザでページをリロードすると、入力内容を送信するたびにTodoリスト下へTodoアイテムが追加されます。また、入力内容を送信するたびに `todoItemCount` が加算され、Todoアイテム数の表示も更新されます。



このセクションでの変更点は次のとおりです。

todoapp

```
|__ index.html
|__ index.js
|__ package.json
└__ src
    |__ App.js(Todoアイテムの表示の実装)
    └__ view
        └__ html-util.js(追加)
```

現在のTodoアプリは次のURLで実際に確認できます。

<https://jsprimer.net/use-case/todoapp/form-event/add-todo-item/>

## まとめ

このセクションではform要素の `submit` イベントをリッスンし、入力内容を元にTodoアイテムをTodoリストの追加を実装しました。今回のTodoアイテムの追加のように多くのウェブアプリは、何らかのイベントが発生し、そのイベントをリッスンして表示を更新します。このようなイベントが発生したことを元に処理を進める方法をイベント駆動（イベントドリブン）と呼びます。

今回のTodoアイテムの追加では、`submit` イベントを入力にして、直接Todoリスト要素を追加するという方法を取っていました。このように直接DOMを更新するという方法はコードが短くなりますが、DOMのみにしか状態は残らないため柔軟性がなくなるという問題があります。

次のセクションではどのような問題がおきるかや、それを解決するための仕組みを見ていきます。

# イベントとモデル

Todoアイテムを追加する機能を実装しましたが、イベントを受け取り直接DOMを更新する方法は柔軟性がなくなるという問題があります。また「Todoアイテムの更新」という機能を実装するには追加したTodoアイテム要素を識別する方法が必要です。具体的には `id` 属性などユニークな識別子などがどこにもないため、特定のアイテムを指定する更新や削除が実装できません。

まずはどのような点で柔軟性の問題が起きやすかについて見ていきます。その後、柔軟性や識別子の問題を解決するためにモデルという概念を導入し、あらためて「Todoアイテムの追加」の機能を見ていきます。

## 直接DOMを更新する問題

[前回のセクション](#)では、操作した結果発生したイベントという入力に対して、DOM（表示）の更新という出力が1対1でおこなわれていました。つまりTodoリストにTodoアイテムが何個あるか、どのようなアイテムがあるかという状態がDOM上にしか存在しないことになります。

そのため、Todoアイテムの状態を更新するには、HTML要素にTodoアイテムの情報（タイトルや識別子となる`id`など）をすべて埋め込む必要があります。しかし、HTML要素に対して文字列しか埋め込めないため、Todoアイテムのデータを文字列にしないといけないという制限が発生します。

また操作と表示が1対1で更新される場合、1つの操作に対して複数の箇所の表示が更新されることもあります。今回のTodoアプリでもTodoリスト(`#js-todo-list`)とTodoアイテム数(`#js-todo-count`)の2箇所を更新する必要があることからも分かりります。

次の表に操作に対して更新する表示をまとめてみます。

機能	操作	表示
Todoアイテムの追加	フォームを入力し送信	Todoリスト( <code>#js-todo-list</code> )にTodoアイテム要素を作成し子要素として追加。合わせてTodoアイテム数( <code>#js-todo-count</code> )を更新
Todoアイテムの更新	チェックボックスをクリック	Todoリスト( <code>#js-todo-list</code> )にある指定したTodoアイテム要素のチェック状態を更新
Todoアイテムの削除	削除ボタンをクリック	Todoリスト( <code>#js-todo-list</code> )にある指定したTodoアイテム要素を削除。合わせてTodoアイテム数( <code>#js-todo-count</code> )を更新

これは表示を更新しなければいけない箇所が増えるほど、操作に対する処理が複雑化していくことが予想できます。

ここでは次の2つの問題が見つかりました。

- Todoリストの状態がDOM上にしか存在しないため、状態をすべてDOM上に文字列で埋め込まないといけない
- 操作に対して更新する表示箇所が増えてくると、表示の処理が複雑化する

## モデルを導入する

この問題を避けるために、Todoアイテムという情報をJavaScriptクラスとしてモデル化します。ここでモデルとはTodoアイテムやTodoリストといったものを表現し、操作や状態をもたせたオブジェクトという意味です。クラスでは操作はメソッドとして実装し、状態はインスタンスにプロパティの値で管理できるため、今回はクラスでモデルを表現します。

たとえば、Todoリストを表現するモデルとして `TodoListModel` クラスを考えます。TodoリストにはTodoアイテムを追加できるので `TodoListModel#addItem` メソッドを実装する必要があります。また、Todoリストからアイテムの一覧を取得できる必要もあるので `TodoListModel#getAllItems` メソッドも必要です。このようにTodoリストをクラスで表現する際にオブジェクトがどのような処理や状態をもつかを考え実装します。

このようにモデルを考え、先ほどの操作と表示の間にモデルを入れることを考えてみます。「フォームを入力し送信」という操作を行った場合、`TodoListModel` というモデルに対して `TodoItemModel` を追加します。そして、`TodoListModel` からアイテムの一覧を取得し、DOMを組み立て表示を更新します。

先ほどの表にモデルをいれてみます。操作に対するモデルの処理はさまざまですが、操作に対する表示の処理はどの場合も同じになります。これは表示箇所が増えた場合も表示の処理の複雑さが一定に保てるこことを意味しています。

機能	操作	モデルの処理	表示
Todoアイテムの追加	フォームを入力し送信	<code>TodoListModel</code> へ新しい <code>TodoItemModel</code> を追加	<code>TodoListModel</code> を元に表示を更新
Todoアイテムの更新	チェックボックスをクリック	<code>TodoListModel</code> の指定した <code>TodoItemModel</code> の状態を更新	<code>TodoListModel</code> を元に表示を更新
Todoアイテムの削除	削除ボタンをクリック	<code>TodoListModel</code> から指定の <code>TodoItemModel</code> を削除	<code>TodoListModel</code> を元に表示を更新

この表を元にあらためて先ほどの問題点を見ていきましょう。

Todoリストの状態がDOM上にしか存在しないため、状態をすべてDOM上に文字列で埋め込まないといけないモデルであるクラスのインスタンスを参照すれば情報が手に入ります。またモデルはただのJavaScriptクラスであるため、文字列ではない情報も保持できます。そのため、DOMにすべての情報を埋め込む必要はありません。

操作に対して更新する表示箇所が増えてくると、表示の処理が複雑化する

表示はモデルの状態を元にしてHTML要素を作成し表示を更新します。モデルの状態が変化していかなければ、表示は変わらなくても問題ありません。

そのため操作したタイミングではなく、モデルの状態が変化したタイミングで表示を更新すればよいはずです。具体的には「フォームを入力し送信」されたから表示を更新するのではなく、「`TodoListModel` というモデルの状態が変化」したから表示を更新すればよいはずです。

そのためには、`TodoListModel` というモデルの状態が変化したことを表示側から知る必要があります。ここで再び出てくるのがイベントです。

## モデルの変化を伝えるイベント

フォームを送信したらform要素から `submit` イベントが発生します。これと同じように `TodoListModel` の状態が変化したら自分自身へ `change` イベントをディスパッチします。表示側はそのイベントをリッスンしてイベントが発生したら表示を更新すればよいはずです。

`TodoListModel` の状態の変化とは、「`TodoListModel` に新しい `TodoItemModel` が追加される」などが該当します。先ほど表の「モデルの処理」は何かしら状態が変化しているので、表示を更新する必要があるわけです。

DOM APIのイベントの仕組みをモデルでも利用できれば、モデルが更新されたら表示を更新する仕組みを作れそうです。ブラウザのDOM APIではこのようなイベント仕組みをDOM Eventsと呼びます。Node.jsでは `events` と呼ばれるモジュールでAPIは異なりますが同様のイベントの仕組みが利用できます。ここではイベントの仕組みを理解するために、イベントのディスパッチとリッスンする機能をもつクラスを作ってみましょう。

とても難しく聞こえますが、今まで学んだクラスやコールバック関数などを使えば実現できます。

## EventEmitter

イベントの仕組みとは「イベントをディスパッチする側」と「イベントをリッスンする側」の2つの面から成り立ちます。場合によっては自分自身へのイベントをディスパッチし、自分自身でイベントをリッスンすることもあります。

このイベントの仕組みを言い換えると「イベントをディスパッチした（イベントが発生）ときにイベントをリッスンしているコールバック関数（イベントリスナー）を呼び出す」となります。

モデルが更新されたら表示を更新するには「`TodoListModel` が更新したときに指定したコールバック関数を呼び出すクラス」を作れば目的は達成できます。しかし、「`TodoListModel` が更新されたとき」というのはとても具体的な処理であるため、モデルを増やすたびに同じ処理をそれぞれのモデルへ実装する必要があります。

そのため、先ほどのイベントの仕組みを持った概念として `EventEmitter` というクラスを作成します。そして `TodoListModel` は作成した `EventEmitter` を継承することでイベントの仕組みを導入していきます。

- 親クラス (`EventEmitter`) : イベントをディスパッチした時、登録されているコールバック関数（イベントリスナー）を呼び出すクラス
- 子クラス (`TodoListModel`) : 値を更新した時、登録されているコールバック関数を呼び出すクラス

まずは、親クラスとなる `EventEmitter` を作成していきます。

`EventEmitter` はイベントの仕組みで書いたディスパッチ側とリッスン側の機能を持ったクラスとなります。

- ディスパッチ側: `addEventLister` メソッドは、指定した イベント名 に任意のコールバック関数を登録できる
- リッスン側: `emit` メソッドは、指定された イベント名 に登録済みのすべてのコールバック関数を呼び出す

これによって、`emit` メソッドを呼び出すと指定したイベントに関係する登録済みのコールバック関数を呼び出せます。このようなパターンはObserverパターンとも呼ばれ、ブラウザやNode.jsなど多くの実行環境で類似するAPIが存在します。

次のように `src/EventEmitter.js` へ `EventEmitter` クラスを定義します。

`src/EventEmitter.js`

```
export class EventEmitter {
  constructor() {
    // 登録する [イベント名, Set(リスナー関数)] を管理するMap
    this._listeners = new Map();
  }

  /**
   * 指定したイベントが実行されたときに呼び出されるリスナー関数を登録する
   * @param {string} type イベント名
   * @param {Function} listener イベントリスナー
   */
  addEventLister(type, listener) {
    // 指定したイベントに対応するSetを作成しリスナー関数を登録する
    if (!this._listeners.has(type)) {
      this._listeners.set(type, new Set());
    }
    const listenerSet = this._listeners.get(type);
    listenerSet.add(listener);
  }

  /**
   * 指定したイベントをディスパッチする
   * @param {string} type イベント名
   */
  emit(type) {
    // 指定したイベントに対応するSetを取り出し、すべてのリスナー関数を呼び出す
    const listenerSet = this._listeners.get(type);
    if (!listenerSet) {
      return;
    }
    listenerSet.forEach(listener => {
      listener();
    });
  }
}
```

```

        return;
    }
    listenerSet.forEach(listener => {
        listener.call(this);
    });
}

/**
 * 指定したイベントのイベントリスナーを解除する
 * @param {string} type イベント名
 * @param {Function} listener イベントリスナー
 */
removeEventLister(type, listener) {
    // 指定したイベントに対応するSetを取り出し、該当するリスナー関数を削除する
    const listenerSet = this._listeners.get(type);
    if (!listenerSet) {
        return;
    }
    listenerSet.forEach(ownListener => {
        if (ownListener === listener) {
            listenerSet.delete(listener);
        }
    });
}
}

```

この `EventEmitter` は次のようにイベントのリッスンとイベントのディスパッチの機能が利用できます。リッスン側は `addEventLister` メソッドでイベントの種類（`type`）に対するイベントリスナー（`listener`）を登録します。ディスパッチ側は `emit` メソッドでイベントをディスパッチし、イベントリスナーを呼び出します。

次のコードでは、`addEventLister` メソッドで `test-event` イベントに対して2つのイベントリスナーを登録しています。そのため、`emit` メソッドで `test-event` イベントをディスパッチすると、登録済みのイベントリスナーが呼び出されています。

`EventEmitter` の実行サンプル

```

import { EventEmitter } from "./EventEmitter.js";
const event = new EventEmitter();
// イベントリスナー（コールバック関数）を登録
event.addEventLister("test-event", () => console.log("One!"));
event.addEventLister("test-event", () => console.log("Two!"));
// イベントをディスパッチする
event.emit("test-event");
// コールバック関数がそれぞれ呼びだされ、コンソールには次のように出力される
// "One!"
// "Two!"

```

## EventEmitterを継承したTodoListモデル

次は作成した `EventEmitter` クラスを継承した `TodoListModel` クラスを作成しています。`src/model/` ディレクトリを新たに作成し、このディレクトリに各モデルクラスを実装したファイルを作成します。

作成するモデルは、Todoリストを表現する `TodoListModel` と各Todoアイテムを表現する `TodoItemModel` です。`TodoListModel` が複数の `TodoItemModel` を保持することでTodoリストを表現することになります。

- `TodoListModel` : Todoリストを表現するモデル
- `TodoItemModel` : Todoアイテムを表現するモデル

まずは `TodoItemModel` を `src/model/TodoItemModel.js` へ作成します。

`TodoItemModel` クラスは各Todoアイテムに必要な情報を定義します。各Todoアイテムにはタイトル (`title`)、アイテムの完了状態 (`completed`)、アイテムごとにユニークな識別子 (`id`) をもたせます。ただのデータの集合であるため、クラスではなくオブジェクトでも問題はありませんが、今回はクラスとして作成します。

次のように `src/model/TodoItemModel.js` へ `TodoItemModel` クラスを定義します。

`src/model/TodoItemModel.js`

```
// ユニークなIDを管理する変数
let todoIdx = 0;

export class TodoItemModel {
  /**
   * @param {string} title Todoアイテムのタイトル
   * @param {boolean} completed Todoアイテムが完了済みならばtrue、そうでない場合はfalse
   */
  constructor({ title, completed }) {
    // idは自動的に連番となりそれぞれのインスタンス毎に異なるものとする
    this.id = todoIdx++;
    this.title = title;
    this.completed = completed;
  }
}
```

次のコードでは `TodoItemModel` クラスはインスタンス化でき、それぞれの `id` が自動的に異なる値となっていることが確認できます。この `id` は後ほど特定のTodoアイテムを指定した更新する処理ときに、アイテムを区別する識別子として利用します。

```
import { TodoItemModel } from "./TodoItemModel";
const item = new TodoItemModel({
  title: "未完了のTodoアイテム",
  completed: false
});
const completedItem = new TodoItemModel({
  title: "完了済みのTodoアイテム",
  completed: true
});
// それぞれの`id`は異なる
console.log(item.id !== completedItem.id); // => true
```

次に `TodoListModel` を `src/model/TodoListModel.js` へ作成します。

`TodoListModel` クラスは、先ほど作成した `EventEmitter` クラスを継承します。`TodoListModel` クラスは `TodoItemModel` の配列を保持し、新しいTodoアイテムを追加する際はその配列に追加します。このとき `TodoListModel` の状態が変更したこと通知するために自分自身へ `change` イベントをディスパッチします。

`src/model/TodoListModel.js`

```
import { EventEmitter } from "../EventEmitter.js";

export class TodoListModel extends EventEmitter {
  /**
   * @param {TodoItemModel[]} [items] 初期アイテム一覧（デフォルトは空の配列）
   */
  constructor(items = []) {
    super();
    this.items = items;
  }

  /**
   * TodoItemの合計数を返す
   * @returns {number}
   */
}
```

```

    get totalCount() {
        return this.items.length;
    }

    /**
     * 表示できるTodoItemの配列を返す
     * @returns {TodoItemModel[]}
     */
    getTodoItems() {
        return this.items;
    }

    /**
     * TodoListの状態が更新されたときに呼び出されるリスナー関数を登録する
     * @param {Function} listener
     */
    onChange(listener) {
        this.addEventListener("change", listener);
    }

    /**
     * 状態が変更されたときに呼ぶ。登録済みのリスナー関数を呼び出す
     */
    emitChange() {
        this.emit("change");
    }

    /**
     * TodoItemを追加する
     * @param {TodoItemModel} todoItem
     */
    addTodo(todoItem) {
        this.items.push(todoItem);
        this.emitChange();
    }
}

```

次のコードは `TodoListModel` クラスを取り込み、新しい `TodoItemModel` を追加するサンプルコードです。

`TodoListModel#addTodo` メソッドで新しいTodoアイテムを追加した時に、`TodoListModel#onChange` で登録したイベントリスナーが呼び出されます。

```

import { TodoItemModel } from "./TodoItemModel";
import { TodoListModel } from "./TodoListModel";
// 新しいTodoリストを作成する
const todoListModel = new TodoListModel();
// 現在のTodoアイテム数は0
console.log(todoListModel.totalCount); // => 0
// Todoリストが変更されたら呼ばれるイベントリスナーを登録する
todoListModel.onChange(() => {
    console.log("TodoListの状態が変わりました");
});
// 新しいTodoアイテムを追加する
// => `onChange`で登録したイベントリスナーが呼び出される
todoListModel.addTodo(new TodoItemModel({
    title: "新しいTodoアイテム",
    completed: false
}));
// Todoリストにアイテムが増える
console.log(todoListModel.totalCount); // => 1;

```

これでTodoリストに必要なそれぞれのモデルクラスが作成できました。次はこれらのモデルを使い表示の更新を行ってみましょう。

## モデルを使って表示を更新する

さきほど作成した `TodoListModel` と `TodoItemModel` クラスを使い、「Todoアイテムの追加」を書き直してみます。

前回のセクションでは、フォームを送信すると直接DOMへ要素を追加しています。今回のセクションでは、フォームを送信すると `TodoListModel` へ `TodoItemModel` を追加します。`TodoListModel` に新しいTodoアイテムが増えると、`onChange` に登録したイベントリスナーが呼び出されるため、そのリスナー関数内でDOM（表示）を更新します。

まずは書き換え後の `App.js` を見ていきましょう。

```
import { TodoListModel } from "./model/TodoListModel.js";
import { TodoItemModel } from "./model/TodoItemModel.js";
import { element, render } from "./view/html-util.js";

export class App {
  constructor() {
    // 1. TodoListの初期化
    this.todoListModel = new TodoListModel();
  }
  mount() {
    const formElement = document.querySelector("#js-form");
    const inputElement = document.querySelector("#js-form-input");
    const containerElement = document.querySelector("#js-todo-list");
    const todoItemCountElement = document.querySelector("#js-todo-count");
    // 2. TodoListModelの状態が更新されたら表示を更新する
    this.todoListModel.onChange(() => {
      // TodoリストをまとめるList要素
      const todoListElement = element`<ul />`;
      // それぞれのTodoItem要素をtodoListElement以下へ追加する
      const todoItems = this.todoListModel.getTodoItems();
      todoItems.forEach(item => {
        const todoItemElement = element`<li>${item.title}</li>`;
        todoListElement.appendChild(todoItemElement);
      });
      // containerElementの中身をtodoListElementで上書きする
      render(todoListElement, containerElement);
      // アイテム数の表示を更新
      todoItemCountElement.textContent = `Todoアイテム数: ${this.todoListModel.totalCount}`;
    });
    // 3. フォームを送信したら、新しいTodoItemModelを追加する
    formElement.addEventListener("submit", (event) => {
      event.preventDefault();
      // 新しいTodoItemをTodoListへ追加する
      this.todoListModel.addTodo(new TodoItemModel({
        title: inputElement.value,
        completed: false
      }));
      inputElement.value = "";
    });
  }
}
```

変更後の `App.js` では大きく分けて3つの部分が変更されているので順番に見ていきましょう。

### 1. TodoListの初期化

作成した `TodoListModel` と `TodoItemModel` を取り込んでいます。

```
import { TodoListModel } from "./model/TodoListModel.js";
import { TodoItemModel } from "./model/TodoItemModel.js";
```

そして、`App` クラスのコンストラクタ内で `TodoListModel` を初期化しています。`App` のコンストラクタで `TodoListModel` を初期化しているのは、このTodoアプリでは開始時にTodoリストの中身が空の状態で開始されるのに合わせるためです。

```
class App {
  constructor() {
    // 1. TodoListの初期化
    this.todoListModel = new TodoListModel();
  }
  // ...省略..
}
```

## 2. TodoListModelの状態が更新されたら表示を更新する

`mount` メソッド内で `TodoListModel` が更新されたら表示を更新するという処理を実装します。

`TodoListModel#onChange` で登録したリスナー関数は、`TodoListModel` の状態が更新されたら呼び出されます。

このリスナー関数内では `TodoListModel#getTodoItems` でTodoアイテムを取得しています。そして、アイテム一覧から次のようなリスト要素（`todoListElement`）を作成しています。

```
<!-- todoListElementの実質的な中身 -->
<ul>
  <li>Todoアイテム 1のタイトル</li>
  <li>Todoアイテム 2のタイトル</li>
</ul>
```

この作成した `todoListElement` 要素を前回作成した、`html-util.js` の `render` 関数を使い `containerElement` の中身を上書きしてます。また、アイテム数は `TodoListModel#totalCount` で取得できるため、アイテム数だけを管理していた `todoItemCount` という変数は削除できます。

```
// render関数をimportに追加する
import { element, render } from './view/html-util.js';
// ...省略...
// `containerElement`の中身を`todoListElement`で上書きして表示を更新
render(todoListElement, containerElement);
// アイテム数の表示を更新
todoItemCountElement.textContent = `Todoアイテム数: ${this.todoListModel.totalCount}`;
```

## 3. フォームを送信したら、新しいTodoItemを追加する

前回のセクションでは、フォームを送信（`submit`）が行われると直接DOMへ要素を追加していました。今回のセクションでは、`TodoListModel` の状態が更新されたら表示を更新する仕組みがすでにできています。

そのため、`submit` イベントのリスナー関数内では `TodoListModel` に対して新しい `TodoItemModel` を追加するだけで表示が更新されます。直接DOMへ `appendChild` していた部分を `TodoListModel#addTodo` メソッドを使いモデルを更新する処理へ置き換えるだけです。

## まとめ

今回のセクションでは、[前回のセクション](#)と同等の機能をモデルとイベントの仕組みを使うようにリファクタリングしました。コード量は増えましたが、次に実装する「Todoアイテムの更新」や「Todoアイテムの削除」も同様の仕組みで実装できます。前回のセクションのように操作に対してDOMを直接更新した場合、追加は簡単ですが既存の要素を指定する必要がある更新や削除は難しくなります。

次のセクションでは、今回のモデルと同じように「表示」に関しても整理を行い、残りの「Todoアイテムの更新」や「Todoアイテムの削除」の機能を実装しています。

## このセクションのチェックボックス

- 直接DOMを更新する問題について理解した
- EventEmitterクラスでイベントの仕組みを実装した
- TodoListModelをEventEmitterクラスを継承して実装した
- Todoアイテムの追加の機能をモデルを使ってリファクタリングした

ここまでTodoアプリは次のURLで確認できます。

<https://jsprimer.net/use-case/todoapp/event-model/event-emitter/>

# Todoの更新と削除を実装する

このセクションではTodoアプリの残りの機能である「Todoアイテムの更新」と「Todoアイテムの削除」を実装していきます。

「Todoアイテムの更新」とは、チェックボックスをクリックして未完了だったらチェックを付けて完了済みに、逆完了済みのアイテムを未完了へとトグルする機能のことです。完了状態をTodoアイテムごとにもち、それぞれのTodoの進捗を管理できる機能です。

一方の「Todoアイテムの削除」はボタンをクリックしたらTodoアイテムを削除する機能です。不要となったTodoを削除して完了済みのTodoを取り除くなどに利用できる機能です。

まずは「Todoアイテムの更新」から実装します。その後「Todoアイテムの削除」を実装していきます。

## Todoアイテムの更新

現時点ではTodoアイテムの完了済みかが表示されていません。そのため、まずはTodoアイテムが完了済みかを表示する必要があります。HTMLの `<input type="checkbox">` 要素を使いチェックボックスを表示し、Todoアイテムごとの完了状態を表現します。

`<input type="checkbox">` は `checked` 属性がない場合はチェックが外れた状態のチェックボックスとなります。一方 `<input type="checkbox" checked>` のように `checked` 属性がある場合はチェックがついたチェックボックスとなります。

checked属性なし  checked属性あり

Todoアイテム要素である `<li>` 要素中に次のように `<input>` 要素を追加しチェックボックスを表示に追加します。チェックボックスである `<input>` 要素にはスタイルのために `class` 属性を `checkbox` とします。合わせて完了済みの場合は `<s>` 要素を使い打ち消し線を表示しています。

```
const todoItems = this.todoListModel.getTodoItems();
todoItems.forEach(item => {
    // 完了済みならchecked属性をつけ、未完了ならchecked属性を外す
    // input要素にはcheckboxクラスをつける
    const todoItemElement = item.completed
        ? element`<li><input type="checkbox" class="checkbox" checked><s>${item.title}</s></input></li>`
        : element`<li><input type="checkbox" class="checkbox">${item.title}</input></li>`;
    todoListElement.appendChild(todoItemElement);
});
```

`<input type="checkbox">` 要素はクリックするとチェックの表示がトグルします。しかし、モデルである `TodoItemModel` の `completed` プロパティの状態は自動では切り替わりません。これにより表示とモデルの状態が異なってしまうという問題が発生します。

この問題は次のような操作をしてみると確認できます。

1. Todoアイテムを追加する
2. Todoアイテムのチェックボックスにチェックを付ける
3. 別の新しいTodoアイテムを追加する
4. すべてのチェックボックスのチェックがリセットされてしまう

この問題を避けるためにも、`<input type="checkbox">` 要素がチェックされたらモデルの状態を更新する必要があります。

`<input type="checkbox">` 要素はチェックされたときに `change` イベントをディスパッチします。この `change` イベントをリッスン (listen) して、`TodoItem` モデルの状態を更新すればモデルと表示の状態を同期できます。

`input` 要素からディスパッチされる `change` イベントをリッスンする処理は次のようにかけます。

まずは `todoItemElement` 要素の下にある `input` 要素を `querySelector` メソッドで探索します。以前は `document.querySelector` で `document` 以下から CSS セレクタにマッチする要素を探していました。

`todoItemElement.querySelector` メソッドを使うことで、`todoItemElement` 下にある要素だけを対象に探索できます。

見つけた `input` 要素に対して `addEventListener` メソッドで `change` イベントが発生したときに呼ばれるコールバック関数を登録できます。この `addEventListener` メソッドは `XMLHttpRequest` の場合と同じくイベント名とコールバック関数を渡すことで、指定したイベントを受け取れます。（「[ユースケース: Ajax通信](#)」を参照）

このようなイベントが発生した際に呼ばれるコールバック関数のことをイベントリスナー（イベントをリッスンするものという意味）と呼びます。またイベントリスナーはイベントハンドラーとも呼ばれることがあります、この書籍ではこの2つの言葉は同じ意味として扱います。

```
const todoItemElement = element`<li><input type="checkbox" class="checkbox">${item.title}</input></li>`;
// クラス名checkboxを持つ要素を取得
const inputCheckboxElement = todoItemElement.querySelector(".checkbox");
// `<input type="checkbox">` のチェックが変更されたときに呼ばれるイベントリスナーを登録
inputCheckboxElement.addEventListener("change", () => {
  // チェックボックスの表示が変わったタイミングで呼び出される処理
  // TODO: ここでモデルを更新する処理を呼ぶ
});
```

ここまでをまとめると、Todo アイテムの更新は次の2つのステップで実装できます。

1. `TodoListModel` に指定した Todo アイテムの更新処理を追加する
2. チェックボックスの `change` イベントが発生したら、モデルの状態を更新する

ここから実際に Todo アイテムの更新を `todoapp` プロジェクトに実装していきます。

## TodoListModel に指定した Todo アイテムの更新処理を追加する

まずは、`TodoListModel` に指定した Todo アイテムを更新する `updateTodo` メソッドを追加します。

`TodoListModel#updateTodo` メソッドは、指定した `id` と一致する Todo アイテムの完了状態 (`completed` プロパティ) を更新します。

```
// =====
// TodoItemModel.js の既存の実装は省略
// =====
/**
 * 指定した id の TodoItem の completed を更新する
 * @param {{ id: number, completed: boolean }} item
 */
updateTodo({ id, completed }) {
  // `id` が一致する TodoItem を見つけ、あるなら完了状態の値を更新する
  const todoItem = this.items.find(todo => todo.id === id);
  if (!todoItem) {
    return;
  }
  todoItem.completed = completed;
  this.emitChange();
}
```

チェックボックスの `change` イベントが発生したら、Todoアイテムの完了状態を更新する

次に `input` 要素の `change` イベントのリスナー関数で、Todoアイテムの完了状態を更新します。

`App.js` で `todoItemElement` の子要素として `checkbox` というクラス名をつけた `input` 要素を追加します。この `input` 要素の `change` イベントが発生したら、`TodoListModel#updateTodo` メソッドを呼び出すようにします。チェックがトグルするたびに呼び出されるので、`completed` には現在の状態を反転（トグル）した値を渡します。

```
const todoItems = this.todoListModel.getTodoItems();
todoItems.forEach(item => {
  // 完了済みならchecked属性をつけ、未完了ならchecked属性を外す
  const todoItemElement = item.completed
    ? element`<li><input type="checkbox" class="checkbox" checked><s>${item.title}</s></input></li>`
    : element`<li><input type="checkbox" class="checkbox">${item.title}</input></li>`;
  // チェックボックスがトグルしたときのイベントにリスナー関数を登録
  const inputCheckboxElement = todoItemElement.querySelector(".checkbox");
  inputCheckboxElement.addEventListener("change", () => {
    // 指定したTodoアイテムの完了状態を反転させる
    this.todoListModel.updateTodo({
      id: item.id,
      completed: !item.completed
    });
  });
  todoListElement.appendChild(todoItemElement);
});
```

`TodoListModel#updateTodo` メソッド内では `emitChange` メソッドによって、`TodoListModel` の変更が通知されます。これによって `TodoListModel#onChange` で登録されているイベントリスナーが呼びだされ、表示が更新されます。

これで表示とモデルが同期でき「Todoアイテムの更新処理」が実装できました。

## 削除機能

次は「Todoアイテムの削除機能」を実装していきます。

基本的な流れは「Todoアイテムの更新機能」と同じです。`TodoListModel` にTodoアイテムを削除する処理を追加します。そして表示には削除ボタンを追加し、削除ボタンがクリックされたときの指定したTodoアイテムを削除する処理を呼び出します。

### TodoListModel に指定したTodoアイテムの削除する処理を追加する

まずは、`TodoListModel` に指定したTodoアイテムを削除する `deleteTodo` メソッドを追加します。

`TodoListModel#deleteTodo` メソッドは、指定したidと一致するTodoアイテムを削除します。

`items` というTodoアイテムの配列から指定したidと一致するTodoアイテムを取り除くことで削除しています。

```
// =====
// TodoItemModel.jsの既存の実装は省略
// =====
/** 
 * 指定したidのTodoItemを削除する
 * @param {{ id: number }}
 */
deleteTodo({ id }) {
  // `id`が一致するTodoItemを`this.items`から取り除き、削除する
  this.items = this.items.filter(todo => {
    return todo.id !== id;
  });
  this.emitChange();
}
```

```

    }
}

```

## 削除ボタンの click イベントが発生したら、Todoアイテムを削除する

次に `button` 要素の `click` イベントのリスナー関数でTodoアイテムを削除する処理を呼び出します。

`App.js` で `todoItemElement` の子要素として `delete` というクラス名をつけた `button` 要素を追加します。この要素がクリック（`click`）されたときに呼び出されるイベントリスナーを `addEventListener` メソッドで登録します。このイベントリスナーの中で `TodoListModel#deleteTodo` メソッドを呼び指定したidのTodoアイテムを削除します。

```

const todoItems = this.todoListModel.getTodoItems();
todoItems.forEach(item => {
    // 削除ボタン(x)をそれぞれ追加する
    const todoItemElement = item.completed
        ? element`<li><input type="checkbox" class="checkbox" checked>
          ${item.title}</s>
          <button class="delete">x</button>
        </input></li>`
        : element`<li><input type="checkbox" class="checkbox">
          ${item.title}
          <button class="delete">x</button>
        </input></li>`;
    // チェックボックスのトグル処理は変更なし
    const inputCheckboxElement = todoItemElement.querySelector(".checkbox");
    inputCheckboxElement.addEventListener("change", () => {
        this.todoListModel.updateTodo({
            id: item.id,
            completed: !item.completed
        });
    });
    // 削除ボタン(x)をクリック時にTodoListModelからアイテムを削除する
    const deleteButtonElement = todoItemElement.querySelector(".delete");
    deleteButtonElement.addEventListener("click", () => {
        this.todoListModel.deleteTodo({
            id: item.id
        });
    });
    todoItemElement.appendChild(todoItemElement);
});

```

`TodoListModel#deleteTodo` メソッド内では `emitChange` メソッドによって、`TodoListModel` の変更が通知されます。これにより表示が `TodoListModel` と同期するように更新され、表示からもTodoアイテムが削除できます。

これで「Todoアイテムの削除機能」が実装できました。

## まとめ

このセクションでは次のことができるようになりました。

- Todoアイテムの完了状態としてを表示に追加した
- チェックボックスが更新時のchangeイベントのリスナー関数でTodoアイテムの更新した
- Todoアイテムを削除するボタンとしてを表示に追加した
- 削除ボタンのclickイベントのリスナー関数でTodoアイテムを削除した
- Todoアイテムの追加、更新、削除の機能が動作するのを確認できた

このセクションでTodoアプリに必要な要件が実装できました。

- Todoアイテムを追加できる
- Todoアイテムの完了状態を更新できる

Todoアプリムを削除できる

最後のセクションでは、`App.js` のリファクタリングを行い継続的に開発できるアプリの作り方について見ていきます。

# Todoアプリのリファクタリング

前のセクションでTodoアプリの機能の実装できました。しかし、App.jsを見てみるとほとんどがHTML要素の作成処理になっています。このようなHTML要素の作成処理は表示する内容が増えるほど行数が線形的に増えています。このままTodoアプリを拡張していくとApp.jsが肥大化してコードが読みにくく、メンテナンス性が低下してしまいます。

App.jsの役割を振り返ってみましょう。`App` というクラスを持ち、このクラスではModelの初期化やHTML要素とModel間で発生するイベントを中継する役割をもっています。表示から発生したイベントをModelに伝え、Modelから発生した変更イベントを表示に伝えているという管理者といえます。

このセクションでは `App` クラスをイベントの管理者という役割に集中させるため、`App` クラスに書かれているHTML要素を作成する処理を別のクラスへ移動させるリファクタリングを行います。

## Viewクラス

`App` クラスの大部分の占めているのは `TodoItemModel` の配列に対応するTodoリストのHTML要素を作成する処理です。このような表示のための処理をViewクラスとしてモジュールにして、`App` クラスから作成したViewモジュールを使うような形にリファクタリングをしていきます。

Todoリストの表示は次の2つの部品（コンポーネント）から成り立っています。

- Todoアイテムコンポーネント
- TodoアイテムをリストとしてまとめたTodoリストコンポーネント

この部品に対応するように次のViewのモジュールを作成していきます。

- `TodoItemView` : Todoアイテムコンポーネント
- `TodoListView` : Todoリストコンポーネント

### TodoItemViewを作成する

まずは、Todoアイテムに対応する `TodoItemView` から作成しています。

`view/TodoItemView.js` ファイルを作成して、次のような `TodoItemView` クラスを `export` します。この `TodoItemView` はTodoアイテムに対応するHTML要素を返す `createElement` メソッドを持ちます。

`src/view/TodoItemView.js`

```
import { element } from "./html-util.js";

export class TodoItemView {
    /**
     * `todoItem`に対応するTodoアイテムのHTML要素を作成して返す
     * @param {TodoItemModel} todoItem
     * @param {function({id:string, completed: boolean})} onUpdateTodo チェックボックスの更新イベントリスナー
     * @param {function({id:string})} onDeleteTodo 削除ボタンのクリックイベントリスナー
     * @returns {Element}
     */
    createElement(todoItem, { onUpdateTodo, onDeleteTodo }) {
        const todoItemElement = todoItem.completed
            ? element`<li><input type="checkbox" class="checkbox" checked>
                <s>${todoItem.title}</s>
                <button class="delete">x</button>
            </input></li>`
            : element`<li><input type="checkbox" class="checkbox">
                ${todoItem.title}</li>`;
    }
}
```

```

        <button class="delete">x</button>
    </input></li>;
const inputCheckboxElement = todoItemElement.querySelector(".checkbox");
inputCheckboxElement.addEventListener("change", () => {
    // コールバック関数に変更
    onUpdateTodo({
        id: todoItem.id,
        completed: !todoItem.completed
    });
});
const deleteButtonElement = todoItemElement.querySelector(".delete");
deleteButtonElement.addEventListener("click", () => {
    // コールバック関数に変更
    onDeleteTodo({
        id: todoItem.id
    });
});
// 作成したTodoアイテムのHTML要素を返す
return todoItemElement;
}
}

```

`TodoItemView#createElement` メソッドの中身は元々 `App` クラスでのHTML要素を作成する部分を元にしています。  
`createElement` メソッドは、`TodoItemModel` のインスタンスだけではなく `onUpdateTodo` と `onDeleteTodo` のリスナー関数を受け取っています。この受け取ったリスナー関数はそれぞれ対応するイベントが発生した際に呼びだします。

このように引数としてリスナー関数を外から受け取ることで、イベントが発生したときの具体的な処理はViewクラスの外側に定義できます。

たとえば、この `TodoItemView` クラスは次のように利用できます。`TodoItemModel` のインスタンスとイベントリスナーのオブジェクトを受け取り、TodoアイテムのHTML要素を返します。

```

import { TodoItemModel } from "../model/TodoItemModel.js";
import { TodoItemView } from "./TodoItemView.js";

// TodoItemViewをインスタンス化
const todoItemView = new TodoItemView();
// 対応するTodoItemModelを作成する
const todoItemModel = new TodoItemModel({
    title: "あたらしいTodo",
    completed: false
});
// TodoItemModelからHTML要素を作成する
const todoItemElement = todoItemView.createElement(todoItemModel, {
    onUpdateTodo: () => {
        // チェックボックスが更新されたときに呼ばれるリスナー関数
    },
    onDeleteTodo: () => {
        // 削除ボタンがクリックされたときによばれるリスナー関数
    }
});
console.log(todoItemElement); // <li>要素が入る

```

## TodoListViewを作成する

次はTodoリストに対応する `TodoListView` を作成します。

`view/TodoListView.js` には次のような `TodoListView` クラスを `export` します。この `TodoListView` は `TodoItemModel` の配列に対応するTodoリストのHTML要素を返す `createElement` メソッドを持ちます。

src/view/TodoListView.js

```

import { element } from "./html-util.js";

```

```

import { TodoItemView } from "./TodoItemView.js";

export class TodoListView {
    /**
     * `todoItems`に対応するTodoリストのHTML要素を作成して返す
     * @param {TodoItemModel[]} todoItems TodoItemModelの配列
     * @param {function({id:string, completed: boolean})} onUpdateTodo チェックボックスの更新イベントリスナー
     * @param {function({id:string})} onDeleteTodo 削除ボタンのクリックイベントリスナー
     * @returns {Element} TodoItemModelの配列に対応したリストのHTML要素
     */
    createElement(todoItems, { onUpdateTodo, onDeleteTodo }) {
        const todoListElement = element`<ul />`;
        // todoItemsに対応するアイテム要素を作りリストへ追加する
        todoItems.forEach(todoItem => {
            const todoItemView = new TodoItemView();
            // todoItemに対応したHTML要素を作成する
            const todoItemElement = todoItemView.createElement(todoItem, {
                onDeleteTodo,
                onUpdateTodo
            });
            todoListElement.appendChild(todoItemElement);
        });
        // todoListElementを返す
        return todoListElement;
    }
}

```

`TodoListView#createElement` メソッドは `TodoItemView` を使いTodoアイテムのHTML要素作り、`<li>` 要素に追加していきます。この `TodoListView#createElement` メソッドも `onUpdateTodo` と `onDeleteTodo` のリスナー関数を受け取ります。しかし、`TodoListView` ではこのリスナー関数を `TodoItemView` にそのまま渡しています。なぜなら具体的なDOMイベントを発生させる要素が作られるのは `TodoItemView` の中となるためです。

## Appのリファクタリング

最後に作成した `TodoItemView` クラスと `TodoListView` クラスを使い `App` クラスをリファクタリングしていきます。

`App.js` を次のように `TodoListView` クラスを使うように書き換えます。`onChange` のリスナー関数で `TodoListView` クラスを使いTodoリストのHTML要素を作るよう変更します。このとき `TodoListView#createElement` メソッドには次のようにそれぞれ対応するコールバック関数をわたします。

- `onUpdateTodo` のコールバック関数では `TodoListModel#updateTodo` メソッドを呼ぶ
- `onDeleteTodo` のコールバック関数では `TodoListModel#deleteTodo` メソッドを呼ぶ

src/App.js

```

import { TodoListModel } from "./model/TodoListModel.js";
import { TodoItemModel } from "./model/TodoItemModel.js";
import { TodoListView } from "./view/TodoListView.js";
import { render } from "./view/html-util.js";

export class App {
    constructor() {
        this.todoListModel = new TodoListModel();
    }

    mount() {
        const formElement = document.querySelector("#js-form");
        const inputElement = document.querySelector("#js-form-input");
        const containerElement = document.querySelector("#js-todo-list");
        const todoItemCountElement = document.querySelector("#js-todo-count");
        this.todoListModel.onChange(() => {
            const todoItems = this.todoListModel.getTodoItems();
            const todoListView = new TodoListView();

```

```
// todoItemsに対応するTodoListViewを作成する
const todoListElement = todoListView.createElement(todoItems, {
    // Todoアイテムが更新イベントが発生したときによばれるリスナー関数
    onUpdateTodo: ({ id, completed }) => {
        this.todoListModel.updateTodo({ id, completed });
    },
    // Todoアイテムが削除イベントが発生したときによばれるリスナー関数
    onDeleteTodo: ({ id }) => {
        this.todoListModel.deleteTodo({ id });
    }
});
render(todoListElement, containerElement);
todoItemCountElement.textContent = `Todoアイテム数: ${this.todoListModel.totalCount}`;
});
formElement.addEventListener("submit", (event) => {
    event.preventDefault();
    this.todoListModel.addTodo(new TodoItemModel({
        title: inputElement.value,
        completed: false
    }));
    inputElement.value = "";
});
});
```

これで App クラスからHTML要素の作成処理がViewクラスに移動でき、App クラスにはModelとView間のイベントを管理するだけになりました。

## Appのイベントリスナーを整理する

App クラスで登録しているイベントのリスナー関数を見てみると次の4種類となっています。

イベントの流れ	リスナー関数	役割
Model -> View	this.todoListModel.onChange(listener)	TodoListModel が変更イベントを受け取る
View -> Model	formElement.addEventListener("submit", listener)	フォームの送信イベントを受け取る
View -> Model	onUpdateTodo: listener	Todoアイテムのチェックボックスの更新イベントを受け取る
View -> Model	Todoアイテムの削除イベントを受け取る	

イベントの流れがViewからModelとなっているリスナー関数が3箇所あり、それぞれリスナー関数はコード上バラバラな位置に書かれています。また、それぞれのリスナー関数はTodoアプリの機能と対応していることがわかります。これらのリスナー関数がTodoアプリの扱っている機能であるということをわかりやすくするため、リスナー関数を `App` クラスのメソッドとして定義しなおしてみましょう。

src/App.js

```
import { render } from "./view/html-util.js";
import { TodoListView } from "./view/TodoListView.js";
import { TodoItemModel } from "./model/TodoItemModel.js";
import { TodoListModel } from "./model/TodoListModel.js";

export class App {
    constructor() {
        this.todoListView = new TodoListView();
        this.todoListModel = new TodoListModel([]);
    }
}
```

```

    /**
     * Todoを追加時に呼ばれるリスナー関数
     * @param {string} title
     */
    handleAdd(title) {
        this.todoListModel.addTodo(new TodoItemModel({ title, completed: false }));
    };

    /**
     * Todoの状態を更新時に呼ばれるリスナー関数
     * @param {{ id:number, completed: boolean }}
     */
    handleUpdate({ id, completed }) {
        this.todoListModel.updateTodo({ id, completed });
    };

    /**
     * Todoを削除時に呼ばれるリスナー関数
     * @param {{ id: number }}
     */
    handleDelete({ id }) {
        this.todoListModel.deleteTodo({ id });
    };

    mount() {
        const formElement = document.querySelector("#js-form");
        const inputElement = document.querySelector("#js-form-input");
        const todoCountElement = document.querySelector("#js-todo-count");
        const todoListContainerElement = document.querySelector("#js-todo-list");
        this.todoListModel.onChange(() => {
            const todoItems = this.todoListModel.getTodoItems();
            const todoListElement = this.todoListView.createElement(todoItems, {
                // Appに定義したリスナー関数を呼び出す
                onUpdateTodo: ({ id, completed }) => {
                    this.handleUpdate({ id, completed });
                },
                onDeleteTodo: ({ id }) => {
                    this.handleDelete({ id });
                }
            });
            render(todoListElement, todoListContainerElement);
            todoCountElement.textContent = `Todoアイテム数: ${this.todoListModel.totalCount}`;
        });

        formElement.addEventListener("submit", (event) => {
            event.preventDefault();
            this.handleAdd(inputElement.value);
            inputElement.value = "";
        });
    }
}

```

このように App クラスのメソッドとしてリスナー関数を並べることで、Todoアプリの機能がコード上の見た目としてわかりやすくなりました。

## セクションのまとめ

このセクションでは、次のことを行いました。

- ModelとViewをモジュールに分割した
- Todoアプリの機能と対応するリスナー関数をAppクラスのメソッドへ移動した
- Todoアプリを完成させた

完成したTodoアプリは次のURLで確認できます。

- <https://jsprimer.net/use-case/todoapp/final/final/>

実はこのTodoアプリはまだいくつかアプリケーションとして、完成していない部分があります。

入力欄でEnterキーを連打すると、空のTodoアイテムが追加されてしまうのは意図しない挙動です。また、`App#mount` で `TodoListModel#onChange` などのイベントリスナーを登録していますが、そのイベントリスナーを解除していません。このTodoアプリではあまり問題にはなりませんが、イベントリスナーは登録したままだとメモリリークに繋がる場合もあります。

そのため、余力がある人は次の残ったTodoを完成させてみてください。

- タイトルが空の場合は、フォームを送信してもTodoアイテムを追加できないようにする
- `App#mount`でイベントリスナーを登録に対応して、`App#unmount`を追加しイベントリスナーを解除する

`App#mount` と対応する `App#unmount` を作成するTodoは、アプリケーションのライフサイクルを意識するという課題になります。ウェブページにも `load` というページ読み込みが完了した時に発生するイベントと、`unload` というページを破棄した時に発生するイベントがあります。Todoアプリも `mount` と `unmount` を実装し、次のようにウェブページのライフサイクルに合わせられます。

```
const app = new App();
// ページのロードが完了したときのイベント
window.addEventListener("load", () => {
  app.mount();
});
// ページがアンロードされたときのイベント
window.addEventListener("unload", () => {
  app.unmount();
});
```

残ったTodoも実装したものは、次のURLで確認できます。ぜひ、自分で実装してみてウェブページやアプリの動きについて考えてみて下さい。

- <https://jsprimer.net/use-case/todoapp/final/more/>

## Todoアプリのまとめ

今回は、Todoアプリを構成する要素をModelとViewという単位でモジュールに分けていました。モジュールを分けることでコードの見通しを良くしたり、Todoアプリにさらなる機能を追加しやすい形にしました。このようなモジュールの分け方などの設計には正解はなくさまざまな考え方があります。

今回Todoアプリという題材をユースケースに選んだのは、JavaScriptのウェブアプリケーションではよく利用されている題材であるためです。さまざまなライブラリを使ったTodoアプリの実装がTodoMVCと呼ばれるサイトにまとめられています。今回作成したTodoアプリは、TodoMVCからフィルター機能などを削ったものをライブラリを使わずに実装しました。[vanilajs](#)

現実では、ライブラリを全く使わずウェブアプリケーションを実装することは殆どありません。ライブラリを使うことで、`html-util.js` のようなものは自分で書く必要はなくなったり、最後の課題として残ったライフサイクルの問題などは解決しやすくなります。

しかし、ライブラリを使って開発する場合でも、第一部の基本文法や第二部のユースケースで紹介したようなJavaScriptの基礎は重要です。なぜならライブラリも、これらの基礎の上に実装されているためです。

また作るアプリケーションの種類や目的によって適切なライブラリは異なります。ライブラリによっては魔法のような機能を提供しているものもありますが、それらも何かしらの基礎となる技術があることは覚えておいてください。

この書籍ではJavaScriptの基礎を中心に紹介しましたが、「[ECMAScript](#)」の章で紹介したようにJavaScriptの基礎も年々更新されています。基礎が更新されると応用であるライブラリも新しいものが登場し、定番だったものも徐々に変化していきます。そのため知らなかつたものが出てくるのはJavaScript自体が成長しているということです。

この書籍を読んでまだ理解できなかつたことや知らなかつたことがあるのは問題ありません。知らなかつたことを見つけたときにそれが何かを必要に応じて調べられるということが、JavaScriptという変化していく言語やそれを利用する環境においては重要です。

vanilajs ライブラリやフレームワークをつかわずに実装したJavaScriptをVanilla JSと呼ぶことがあります。 ↪