



本扉





※本書の内容は、<https://jsprimer.net/> にて公開されています。
本文中に記載されている社名および商品名は、一般に開発メーカーの登録商標です。
なお、本文中では ™ ・ © ・ ® 表示を明記しておりません。



はじめに

本書の目的

この書籍の目的は、JavaScript というプログラミング言語を学ぶことです。先頭から順番に読んでいけば、JavaScript の文法や機能を一から学べるように書かれています。

JavaScript の文法といった書き方を学ぶことも重要ですが、実際にどのように使われているかを知ることが目的にしています。なぜなら、JavaScript のコードを読んだり書いたりするには、文法の知識だけでは足りないと考えているためです。そのため、「第 1 部 基本文法」では文法だけではなく現実の利用方法について言及し、「第 2 部 ユースケース」では小さなアプリケーションを例に現実と近い使い方を解説しています。

また、JavaScript は常に変化を取り入れている言語でもあり、言語自身や言語を取り巻く開発環境も変化しています。この書籍では、これらの JavaScript を取り巻く変化に対応できる基礎を身につけていくことを目的としています。そのため、単に書き方を学ぶのではなく、なぜ動かないのかや問題の調べ方にも焦点を当てていきます。

本書の目的ではないこと

ひとつの書籍で JavaScript のすべてを学ぶことはできません。なぜなら、JavaScript を使ってできる範囲があまりにも広いからです。そのため、この書籍では取り扱わない内容（目的外）を明確にしておきます。

- 他のプログラミング言語と比較するのが目的ではない
- ウェブブラウザについて学ぶのが目的ではない
- Node.js について学ぶのが目的ではない
- JavaScript のすべての文法や機能を網羅するのが目的ではない
- JavaScript のリファレンスとなることが目的ではない
- JavaScript のライブラリやフレームワーク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わけではない

この書籍は、リファレンスのようにすべての文法や機能を網羅していくことを目的にはしていません。JavaScript やブラウザの API に関しては、[MDN Web Docs](https://developer.mozilla.org/ja/)^{*1} (MDN) というすばらしいリファレンスがすでにあります。

ライブラリの使い方や特定のアプリケーションの作り方を学ぶことも目的ではありません。それらに

*1 <https://developer.mozilla.org/ja/>

はじめに

ついては、ライブラリのドキュメントや実在するアプリケーションから学ぶことを推奨しています。もちろん、ライブラリやアプリケーションについての別の書籍をあわせて読むのもよいでしょう。

この書籍は、それらのライブラリやアプリケーションが動くために利用している仕組みを理解する手助けをします。作り込まれたライブラリやアプリケーションは、一見するとまるで魔法のようにも見えます。実際には、何らかの仕組みがありその上で作られたものがライブラリやアプリケーションとして動いています。

具体的な仕組み自体までは解説しませんが、そこに仕組みがあることに気づき理解する手助けをします。

本書を誰が読むべきか

この書籍は、プログラミング経験のある人が JavaScript という言語を新たに学ぶことを念頭に書かれています。そのため、この書籍で初めてプログラミング言語を学ぶという人には、少し難しい部分があります。しかし、実際にプログラムを動かして学べるように書かれているため、プログラミング初心者も挑戦してみてもよいでしょう。

JavaScript を書いたことはあるが最近の JavaScript がよくわからないという人も、この書籍の読者対象です。2015 年に、JavaScript には ECMAScript 2015 と呼ばれる仕様の大きな変更が入りました。この書籍は、ECMAScript 2015 を前提とした JavaScript の入門書であり、必要な部分では今までの書き方との違いについても触れています。そのため、新しい書き方や何が今までと違うのかわからない場合にも、この書籍は役に立ちます。

この書籍は、JavaScript の仕様に対して真剣に向き合って書かれています。入門書であるからといって、極端に省略して不正確な内容を紹介することは避けています。そのため、JavaScript の熟練者であっても、この書籍を読むことで発見があるはずです。

本書の特徴

この書籍の特徴について簡単に紹介します。

ECMAScript 2015 と呼ばれる仕様の大きな更新が行われた際に、JavaScript には新しい書き方や機能が大きく増えました。今までの JavaScript という言語とは異なるものにも見えるほどです。

この書籍は、新しくなった ECMAScript 2015 以降を前提にして一から書かれています。今から JavaScript を学ぶなら、新しくなった ECMAScript 2015 を前提としたほうがよりスッキリと学べるためです。

また現在のウェブブラウザは、ECMAScript 2015 をサポートしています。そのため、この書籍では一から学ぶ上で知る必要がない古い書き方は紹介していないことがあります。しかし、既存のコードを読む際には古い書き方への理解も必要になるので、頻出するケースについては紹介しています。

一方で、近い未来に入るであろう JavaScript の新しい機能については触れていません。なぜなら、それは未来の話であるため不確定な部分が多く、実際の使われ方も予測できないためです。この書籍は、基本を学びつつ現実のユースケースから離れすぎないことを目的としています。

この書籍の文章やソースコードは、オープンソースとして GitHub の [asciidoctor/js-primer](https://github.com/asciidoctor/js-primer)^{*2}で公

^{*2} <https://github.com/asciidoctor/js-primer>

はじめに

開されています。また書籍の内容が jsprimer.net^{*3} という URL で公開されているため、ウェブブラウザで読めます。ウェブ版では、その場でサンプルコードを実行して JavaScript を学べます。

書籍の内容がウェブで公開されているため、書籍の内容を共有したいときに URL を貼れます。また、書籍の内容やサンプルコードは次のライセンスの範囲内で自由に利用できます。

ライセンス

この書籍に記述されているすべてのソースコードは、MIT ライセンスに基づいたオープンソースソフトウェアとして提供されます。また、この書籍の文章は Creative Commons の Attribution-NonCommercial 4.0 (CC BY-NC 4.0) ライセンスに基づいて提供されます。どちらも、著作権表示がされていればある程度自由に利用できるライセンスとなっています。

ライセンスについての詳細は、次のライセンス文書をご覧ください。

ライセンス文書

Source Code released under the MIT License. Copyright (c) 2016-present jsprimer project

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The text content released under the CC BY-NC 4.0.

Copyright (c) 2016-present jsprimer project

<https://creativecommons.org/licenses/by-nc/4.0/>

文章の間違いに気づいたら

まったくバグがないプログラムはないのと同様に、まったく間違いのない技術書は存在しません。この書籍もできるだけ間違い（特に技術的な間違い）を減らす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。

^{*3} <https://jsprimer.net/>

はじめに

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、文章の意味や意図がわからないといった疑問を持つこともあるでしょう。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。

- 書籍の GitHub リポジトリ: <https://github.com/asciidwango/js-primer>

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/10x4ckFyb0fB9cBM2>

あるいは、アスキードワンゴ編集部にメールを送ることも報告できます。

- アスキードワンゴ編集部メールアドレス: info@asciidwango.jp

問題を修正する

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。

そのため、問題を報告するだけでなく、修正内容を [Pull Request](#) することで問題を修正できます。

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いたします。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。

謝辞

この書籍は次の方々にレビューをしていただきました。

- mizchi (竹馬光太郎)
- 中西優介@better_than_i_w
- @tsinlrou
- sakito
- 川上和義
- 尾上洋介

この書籍をよりよいものにできたのは皆さんのご協力のおかげです。

また、この書籍は最初から [GitHub](#) に公開した状態で執筆が行われています。そのため、Issue で問題の報告や Pull Request で修正を送ってもらうなど、さまざまな人の助けによって成り立っています。この書籍に対してコントリビュートしてくれた方々に感謝します。

著者紹介

azu

ISO/IEC JTC 1/SC 22/ECMAScript Ad Hoc 委員会エキスパートで ECMAScript、JSON の仕様に関わる。2011 年に JSer.info を立ち上げ、継続的に JavaScript の情報を発信している。ライフワークとして OSS へのコントリビューションをしている。

- Twitter: https://twitter.com/azu_re
- GitHub: <https://github.com/azu>

Suguru Inatomi

長崎生まれ福岡育ち。2016 年より Angular 日本ユーザー会の代表を務める。2018 年に日本で一人目の Google Developers Expert for Angular に認定される。日々の仕事の傍ら、Angular をはじめとする OSS へのコントリビューションや翻訳、登壇、イベントの主催などの活動が続けている。

- Twitter: <https://twitter.com/laco2net>
- GitHub: <https://github.com/lacolaco>

目次

はじめに	iii
著者紹介	vii
第 1 部 基本文法	1
第 1 章 JavaScript とは	2
1.1 JavaScript と ECMAScript	2
1.2 JavaScript ってどのような言語?	3
第 2 章 コメント	6
2.1 一行コメント	6
2.2 複数行コメント	6
2.3 まとめ	7
第 3 章 変数と宣言	8
3.1 const	8
3.2 let	9
3.3 var	10
3.4 変数名に使える名前のルール	11
3.5 まとめ	13
第 4 章 値の評価と表示	14
4.1 この書籍で利用するブラウザ	14
4.2 ブラウザで JavaScript を実行する	14
4.3 Console API	18
4.4 ウェブ版の書籍でコードを実行する	19
4.5 コードの評価とエラー	19
4.6 まとめ	22
第 5 章 データ型とリテラル	24
5.1 データ型	24
5.2 リテラル	25

5.3	プリミティブ型とオブジェクト	32
5.4	まとめ	33
第 6 章	演算子	34
6.1	二項演算子	35
6.2	単項演算子 (算術)	36
6.3	比較演算子	39
6.4	ビット演算子	42
6.5	代入演算子 (=)	45
6.6	条件 (三項) 演算子 (?と:)	47
6.7	論理演算子	48
6.8	カンマ演算子 (,)	51
6.9	まとめ	51
第 7 章	暗黙的な型変換	53
7.1	暗黙的な型変換とは	54
7.2	明示的な型変換	55
7.3	明示的な変換でも解決しないこと	62
7.4	まとめ	63
第 8 章	関数と宣言	64
8.1	関数宣言	64
8.2	関数の引数	65
8.3	可変長引数	68
8.4	関数の引数と分割代入	70
8.5	関数はオブジェクト	71
8.6	メソッド	76
8.7	まとめ	77
第 9 章	文と式	78
9.1	式	78
9.2	文	79
9.3	function 宣言 (文) と function 式	80
9.4	まとめ	82
第 10 章	条件分岐	83
10.1	if 文	83
10.2	switch 文	87
10.3	まとめ	90
第 11 章	ループと反復処理	91
11.1	while 文	91
11.2	do-while 文	92

目次

11.3	for 文	93
11.4	配列の <code>forEach</code> メソッド	94
11.5	break 文	95
11.6	continue 文	97
11.7	for...in 文	99
11.8	for...of 文	101
11.9	まとめ	102
第 12 章	オブジェクト	104
12.1	オブジェクトを作成する	104
12.2	プロパティへのアクセス	106
12.3	オブジェクトと分割代入 ES2015	108
12.4	プロパティの追加	108
12.5	プロパティの存在を確認する	110
12.6	<code>toString</code> メソッド	113
12.7	オブジェクトの静的メソッド	114
12.8	まとめ	120
第 13 章	プロトタイプオブジェクト	121
13.1	<code>Object</code> はすべての元	121
13.2	まとめ	126
第 14 章	配列	128
14.1	配列の作成とアクセス	128
14.2	オブジェクトが配列かどうかを判定する	130
14.3	配列と分割代入	131
14.4	配列から要素を検索	131
14.5	追加と削除	135
14.6	配列同士を結合	136
14.7	配列の展開	136
14.8	配列をフラット化	137
14.9	配列から要素を削除	137
14.10	破壊的なメソッドと非破壊的なメソッド	139
14.11	配列を反復処理するメソッド	142
14.12	メソッドチェーンと高階関数	145
14.13	まとめ	146
第 15 章	文字列	148
15.1	文字列を作成する	148
15.2	エスケープシーケンス	149
15.3	文字列を結合する	150
15.4	文字へのアクセス	151

15.5	文字列とは	151
15.6	文字列の分解と結合	153
15.7	文字列の長さ	153
15.8	文字列の比較	154
15.9	文字列の一部を取得	155
15.10	文字列の検索	156
15.11	正規表現オブジェクト	158
15.12	文字列の置換/削除	166
15.13	文字列の組み立て	167
15.14	終わりに	171
第 16 章	文字列と Unicode	172
16.1	Code Point	172
16.2	Code Point と Code Unit の違い	173
16.3	サロゲートペア	175
16.4	Code Point を扱う	176
16.5	まとめ	179
第 17 章	ラッパーオブジェクト	180
17.1	プリミティブ型とラッパーオブジェクト	180
17.2	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	181
17.3	まとめ	182
第 18 章	関数とスコープ	183
18.1	スコープとは	183
18.2	ブロックスコープ	185
18.3	スコープチェーン	186
18.4	グローバルスコープ	187
18.5	関数スコープと var の巻き上げ	190
18.6	関数宣言と巻き上げ	192
18.7	クロージャー	194
18.8	メモリ管理の仕組み	197
18.9	まとめ	203
第 19 章	関数と this	204
19.1	実行コンテキストと this	204
19.2	関数とメソッドにおける this	205
19.3	Arrow Function 以外の関数における this	207
19.4	this が問題となるパターン	210
19.5	Arrow Function と this	218
19.6	まとめ	222

目次

第 20 章	クラス	224
20.1	クラスの定義	224
20.2	クラスのインスタンス化	225
20.3	クラスのプロトタイプメソッドの定義	229
20.4	クラスのアクセッサプロパティの定義	233
20.5	静的メソッド	236
20.6	2 種類のインスタンスメソッドの定義	238
20.7	プロトタイプオブジェクト	239
20.8	プロトタイプチェーン	240
20.9	継承	243
20.10	ビルトインオブジェクトの継承	249
20.11	まとめ	250
第 21 章	例外処理	251
21.1	try...catch 構文	251
21.2	throw 文	252
21.3	エラーオブジェクト	252
21.4	エラーとデバッグ	256
21.5	console.error とスタックトレース	257
21.6	まとめ	258
第 22 章	非同期処理: コールバック /Promise/Async Function	259
22.1	同期処理	259
22.2	非同期処理	260
22.3	非同期処理はメインスレッドで実行される	261
22.4	非同期処理と例外処理	263
22.5	エラーファーストコールバック	264
22.6	Promise	266
22.7	Async Function	287
22.8	Async Function の定義	287
22.9	Async Function は Promise を返す	288
22.10	await 式	289
22.11	Async Function と組み合わせ	293
22.12	まとめ	299
第 23 章	Map/Set	301
23.1	Map	301
23.2	Set	308
23.3	セットの作成と初期化	308
23.4	まとめ	311
第 24 章	JSON	313

24.1	JSON とは	313
24.2	JSON オブジェクト	314
24.3	JSON にシリアライズできないオブジェクト	316
24.4	toJSON メソッドを使ったシリアライズ	317
24.5	まとめ	317
第 25 章	Date	319
25.1	Date オブジェクト	319
25.2	現実のユースケースと Date	322
25.3	まとめ	323
第 26 章	Math	324
26.1	Math オブジェクト	324
26.2	まとめ	326
第 27 章	ECMAScript モジュール	327
27.1	ECMAScript モジュールの構文	327
27.2	ECMAScript モジュールを実行する	333
第 28 章	ECMAScript	334
28.1	ECMAScript のバージョンの歴史	334
28.2	Living Standard となる ECMAScript	335
28.3	仕様策定のプロセス	335
28.4	プロポーザルの機能を試す	336
28.5	仕様や策定プロセスを知る意味	337
第 2 部	ユースケース	340
第 29 章	アプリケーション開発の準備	342
29.1	Node.js のインストール	342
29.2	npx コマンドによる npm パッケージの実行	343
29.3	ローカルサーバーのセットアップ	344
29.4	まとめ	347
第 30 章	ユースケース: Ajax 通信	348
30.1	エントリーポイント	348
30.2	HTTP 通信	351
30.3	データを表示する	357
30.4	Promise を活用する	363
第 31 章	ユースケース: Node.js で CLI アプリケーション	371
31.1	Node.js で Hello World	371

目次

31.2	コマンドライン引数进行处理する	373
31.3	ファイルを読み込む	378
31.4	Markdown を HTML に変換する	382
31.5	ユニットテストを記述する	387
31.6	このセクションのチェックリスト	394
第 32 章	ユースケース: Todo アプリケーション	395
32.1	エントリーポイント	395
32.2	アプリの構成要素	402
32.3	Todo アイテムの追加を実装する	406
32.4	イベントとモデル	415
32.5	Todo アイテムの更新と削除を実装する	427
32.6	Todo アプリのリファクタリング	434
付録 A	参考リンク集	445
A.1	開発を補助するツール	445
A.2	JavaScript の実行プラットフォーム	448

第 1 部 基本文法

Part 1

第1章

JavaScript とは

Chapter 1

JavaScript を学びはじめる前に、まず JavaScript とはどのようなプログラミング言語なのかを紹介します。

JavaScript は主にウェブブラウザの中で動くプログラミング言語です。ウェブサイトで操作をしたら表示が書き換わったり、ウェブサイトのサーバーと通信してデータを取得したりと現在のウェブサイトには欠かせないプログラミング言語です。このような JavaScript を活用して作られたアプリケーションのように操作できるウェブサイトをウェブアプリとも言います。

JavaScript はウェブブラウザだけではなく、Node.js というサーバー側のアプリケーションを作る仕組みでも利用されています。また、デスクトップアプリやスマートフォンアプリ、IoT（Internet of Things）デバイスでも JavaScript を使って動かせるものがあります。このように、JavaScript はかなり幅広い環境で動いているプログラミング言語で、さまざまな種類のアプリケーションを作成できます。

1.1 JavaScript と ECMAScript

JavaScript という言語は **ECMAScript** という仕様によって動作が決められています。**ECMAScript** という仕様では、どの実行環境でも共通な動作のみが定義されているため、基本的にどの実行環境でも同じ動作をします。

一方で、実行環境によって異なる部分もあります。たとえば、ブラウザでは UI（ユーザーインターフェース）を操作するための JavaScript の機能が定義されていますが、サーバー側の処理を書く Node.js ではそれらの機能は不要です。このように、実行環境によって必要な機能は異なるため、それらの機能は実行環境ごとに定義（実装）されています。

そのため、「ECMAScript」はどの実行環境でも共通の部分、「JavaScript」は ECMAScript と実行環境の固有機能も含んだ範囲というのがイメージしやすいでしょう。

ECMAScript の仕様で定義されている機能を学ぶことで、どの実行環境でも対応できる基本的な部分を学べます。この書籍では、この違いを明確に区別する必要がある場合は「ECMAScript」と「JavaScript」という単語を使い分けます。そうでない場合、「JavaScript」という単語を使います。

また、この ECMAScript という仕様（共通の部分）も毎年アップデートされ、新しい文法や機能が追加されています。そのため、実行環境によっては古いバージョンの ECMAScript を実装したものとなっている場合があります。ECMAScript は 2015 年に ECMAScript 2015（ES2015）として大きく

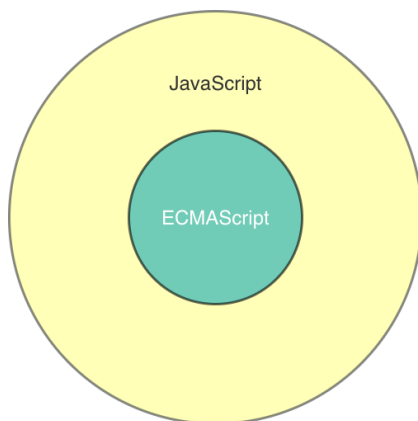


図 1.1 JavaScript と ECMAScript の範囲

アップデートされた仕様が公開されました。

今から JavaScript を学ぶなら、ES2015 以降を基本にしたほうがわかりやすいため、この書籍は ES2015 に基づいた内容となっています。また、既存のコードは ES2015 より前のバージョンを元にしたものも多いため、それらのコードに関しても解説しています。

まずは、JavaScript (ECMAScript) とはどのような言語なのかを大まかに見ていきます。

1.2 JavaScript ってどのような言語？

JavaScript は、元々 Netscape Navigator というブラウザのために開発されたプログラミング言語です。C、Java、Self、Scheme などのプログラミング言語の影響を受けて作られました。

JavaScript は、大部分がオブジェクト（値や処理を 1 つにまとめたものと考えてください）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オブジェクト、実行環境が定義したオブジェクト、ユーザー（つまりあなたです）の定義したオブジェクトが存在します。

この書籍の「第 1 部 基本文法」では ECMAScript の定義する構文やオブジェクトを学んでいきます。「第 2 部 ユースケース」ではブラウザや Node.js といった実行環境が定義するオブジェクトを学びながら、小さなアプリケーションを作成していきます。ユーザーの定義したオブジェクトは、コードを書いていくと自然と登場するため、適宜見ていきます。

次に、JavaScript の言語的な特徴を簡単に紹介していきます。

1.2.1 大文字と小文字を区別する

まず、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 `name` と `NAME` という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";
```

第1章 JavaScript とは

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

また、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持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（変数についての詳細は「[変数と宣言](#)」の章で解説します）。

1.2.2 予約語を持つ

JavaScript には特別な意味を持つキーワードがあり、これらは予約語とも呼ばれます。この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` も予約語のひとつとなっています。そのため、`const` という名前の変数名は宣言できません。

1.2.3 文はセミコロンで区切られる

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。特殊なルールに基づき、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^{*1}。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（詳細は「[文と式](#)」の章で解説します）。

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。これらの空白文字を文にいくつ置いても挙動に違いはありません。

たとえば、次の 1 足す 1 を行う 2 つの文は、+ の前後の空白文字の個数に違いはありますが、動作としてはまったく同じ意味となります。

```
// 式や文の間にスペースがいくつあっても同じ意味となる
1 + 1;
1  +  1;
```

空白文字の置き方には好みがあるため、人によって書き方が異なる場合もあります。複数人で開発する場合は、これらの空白文字の置き方を決めたコーディングスタイルを決めるとよいでしょう。コーディングスタイルの統一については「[付録 A 参考リンク集](#)」を参照してください。

1.2.4 strict mode

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。

"`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。また、後述する “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";
```

^{*1} Automatic Semicolon Insertion と呼ばれる仕組みです。

1.2 JavaScript ってどのような言語？

// 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、明らかな問題を含んだコードに対しては早期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 `const` 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ます。

```
"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。

1.2.5 実行コンテキスト: Script と Module

JavaScript の実行コンテキストとして“Script”と“Module”があります。コードを書く場合には、この2つの実行コンテキストの違いを意識することは多くありません。

“Script”の実行コンテキストは、多くの実行環境ではデフォルトの実行コンテキストです。“Script”の実行コンテキストでは、デフォルトは strict mode ではありません。

“Module”の実行コンテキストは、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、デフォルトが strict mode となり、古く安全でない構文や機能は一部禁止されています。また、モジュールの機能は“Module”の実行コンテキストでしか利用できません。モジュールについての詳細は「[ECMAScript モジュール](#)」の章で解説します。

1.2.6 JavaScript の仕様は毎年更新される

最後に、JavaScript の仕様である ECMAScript は毎年更新され、JavaScript には新しい構文や機能が増え続けています。そのため、この書籍で学んだ後もまだまだ知らなかったことが出てくるはずです。

一方で、ECMAScript は後方互換性が慎重に考慮されているため、過去に書いた JavaScript のコードが動かなくなる変更はほとんど入りません。そのため、この書籍で学んだことのすべてが無駄になることはありません。

ECMAScript の仕様がどのように策定されているかについては「[ECMAScript](#)」の章で解説します。

第2章

コメント

Chapter 2

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。この書籍でも、JavaScript のソースコードを解説するためにコメントを使っています。

コメントの書き方には、一行コメントと複数行コメントの2種類があります。

2.1 一行コメント

一行コメントは名前のとおり、一行ずつコメントを書く際に利用します。// 以降から行末まで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
// この部分はコードとして評価されない
```

2.2 複数行コメント

複数行コメントは名前のとおり、複数行のコメントを書く際に利用します。一行コメントとは違い複数行をまとめて書けるので、長い説明を書く際に利用されています。

/* 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コードとして評価されない
*/
```

複数行コメントの中に、複数行コメントを書くことはできません。次のように、複数行のコメントをネストして書いた場合は構文エラーとなります。

```
/* ネストされた /* 複数行コメント */ は書けない */
```

2.2.1 HTML-like コメント ES2015

ECMAScript 2015 (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。この HTML-like コメントは、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。

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 のコードとして実行される");
--> 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な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 HTML コメントで囲み、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ます。このようなサイトでも JavaScript が動作するという、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^{*1}](https://dev.mozilla.jp/2016/03/es6-in-depth-arrow-functions/)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ほとんどないため、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。一方で、仕様が更新されるたびに新しい機能が增えるため、それを学び続けることには変わりありません。

2.3 まとめ

この章では、ソースコードに説明を書けるコメントについて学びました。

- `//` 以降から行末までが一行コメント
- `/*` と`*/`で囲まれた範囲が複数行コメント
- HTML-like コメントは後方互換性のためだけに存在する

^{*1} <https://dev.mozilla.jp/2016/03/es6-in-depth-arrow-functions/>

第3章

変数と宣言

Chapter 3

プログラミング言語には、文字列や数値などのデータに名前をつけることで、繰り返し利用できるようにする変数という機能があります。

JavaScript には「これは変数です」という宣言をするキーワードとして、`const`、`let`、`var` の 3 つがあります。

`var` はもっとも古くからある変数宣言のキーワードですが、意図しない動作を作りやすい問題が知られています。そのため ECMAScript 2015 で、`var` の問題を改善するために `const` と `let` という新しいキーワードが導入されました。

この章では `const`、`let`、`var` の順に、それぞれの方法で宣言した変数の違いについて見ていきます。

3.1 `const` ES2015

`const` キーワードでは、再代入できない変数の宣言とその変数が参照する値（初期値）を定義できます。

次のように、`const` キーワードに続いて変数名を書き、代入演算子（`=`）の右辺に変数の初期値を書いて変数を定義できます。

```
const 変数名 = 初期値;
```

次のコードでは `bookTitle` という変数を宣言し、初期値が `"JavaScript の本"` という文字列であることを定義しています。

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

`const`、`let`、`var` どのキーワードも共通の仕組みですが、変数同士を、（カンマ）で区切ることでより、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` という変数を順番に定義しています。

```
const 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。

3.2 let

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

また、`const` は再代入できない変数を宣言するキーワードです。そのため、`const` キーワードで宣言した変数に対して、後から値を代入することはできません。

次のコードでは、`const` で宣言した変数 `bookTitle` に対して値を再代入しているため、次のようなエラー（`TypeError`）が発生します。エラーが発生するとそれ以降の処理は実行されなくなります。

```
const bookTitle = "JavaScript の本";  
bookTitle = "新しいタイトル"; // => 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。そのため、変数に対して値を再代入する必要がある場合は、`const` キーワードで変数宣言することを推奨しています。

変数に値を再代入したいケースとして、ループなどの反復処理の途中で特定の変数が参照する値を変化させたい場合があります。そのような場合には、変数への再代入が可能な `let` キーワードを利用します。

3.2 let ES2015

`let` キーワードでは、値の再代入が可能な変数を宣言できます。`let` の使い方は `const` 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、初期値を `"JavaScript の本"` という文字列であることを定義しています。

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

`let` は `const` とは異なり、初期値を指定しない変数も定義できます。初期値が指定されなかった変数はデフォルト値として `undefined` という値で初期化されます（`undefined` は値が未定義ということを表す値です）。

次のコードでは、`bookTitle` という変数を宣言しています。このとき `bookTitle` には初期値が指定されていないため、デフォルト値として `undefined` で初期化されます。

```
let bookTitle;  
// bookTitle は自動的に undefined という値になる
```

この `let` で宣言された `bookTitle` という変数には、代入演算子（`=`）を使うことで値を代入できます。代入演算子（`=`）の右側には変数へ代入する値を書きますが、ここでは `"JavaScript の本"` という文字列を代入しています。

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

第3章 変数と宣言

`let` で宣言した変数に対しては何度でも値の代入が可能です。

```
let count = 0;
count = 1;
count = 2;
count = 3;
```

3.3 var

`var` キーワードでは、値の再代入が可能な変数を宣言できます。`var` の使い方は `let` とほとんど同じです。

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

`var` では、`let` と同じように初期値がない変数を宣言でき、変数に対して値の再代入もできます。

```
var bookTitle;
bookTitle = "JavaScript の本";
bookTitle = "新しいタイトル";
```

3.3.1 var の問題

`var` は `let` とよく似ていますが、`var` キーワードには同じ名前の変数を再定義できてしまう問題があります。

`let` や `const` では、同じ名前の変数を再定義しようとする、次のような構文エラー (`SyntaxError`) が発生します。そのため、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。

```
// "x"という変数名で変数を定義する
let x;
// 同じ変数名の変数"x"を定義すると SyntaxError となる
let x; // => SyntaxError: redeclaration of let x
```

一方、`var` は同じ名前の変数を再定義できます。これは意図せずに同じ変数名で定義してもエラーとならずに、値を上書きしてしまいます。

```
// "x"という変数を定義する
var x = 1;
// 同じ変数名の変数"x"を定義できる
var x = 2;
// 変数 x は 2 となる
```

また `var` には変数の巻き上げと呼ばれる意図しない挙動があり、`let` や `const` ではこの問題が解消

されています。`var` による変数の巻き上げの問題については「[関数とスコープ](#)」の章で解説します。そのため、現時点では「`let` は `var` を改善したバージョン」ということだけ覚えておくといよいです。

このように、`var` にはさまざまな問題があります。また、ほとんどすべてのケースで `var` は `const` か `let` に置き換えが可能です。そのため、これから書くコードに対して `var` を利用することは避けたほうがよいでしょう。

なぜ `let` や `const` は追加されたのか？

ECMAScript 2015 では、`var` そのものを改善するのではなく、新しく `const` と `let` というキーワードを追加することで、`var` の問題を回避できるようにしました。`var` 自体の動作を変更しなかったのは、後方互換性のためです。

なぜなら、`var` の挙動自体を変更してしまうと、すでに `var` で書かれたコードの動作が変わってしまい、動かなくなるアプリケーションが出てくるためです。新しく `const` や `let` などのキーワードを ECMAScript 仕様に追加しても、そのキーワードを使っているソースコードは追加時点では存在しません^a。そのため、`const` や `let` が追加されても後方互換性には影響がありません。

このように、ECMAScript では機能を追加する際にも後方互換性を重視しているため、`var` 自体の挙動は変更されませんでした。

^a `let` や `const` は ECMAScript 2015 以前に予約語として定義されていたため、既存のコードと衝突する可能性はありませんでした。

3.4 変数名に使える名前のルール

ここまで `const`、`let`、`var` での変数宣言とそれぞれの特徴について見てきました。どのキーワードにおいても宣言できる変数に利用できる名前のルールは同じです。また、このルールは変数の名前や関数の名前といった JavaScript の識別子において共通するルールとなります。

変数名の名前（識別子）には、次のルールがあります。

1. 半角のアルファベット、`_`（アンダースコア）、`$`（ダラー）、数字を組み合わせた名前にする
2. 変数名は数字から開始できない
3. 予約語と被る名前は利用できない

変数の名前は、半角のアルファベットである `A` から `Z`（大文字）と `a` から `z`（小文字）、`_`（アンダースコア）、`$`（ダラー）、数字の `0` から `9` を組み合わせた名前にします。JavaScript では、アルファベットの `A` から `Z` の大文字と `a` から `z` の小文字は区別されます。

これらに加えて、ひらがなや一部の漢字なども変数名に利用できますが、全角の文字列が混在すると環境によって扱いにくいこともあるためお勧めしません。

```
var $; // OK: $が利用できる
var _title; // OK: _が利用できる
var jquery; // OK: 小文字のアルファベットが利用できる
var TITLE; // OK: 大文字のアルファベットが利用できる
```

第3章 変数と宣言

```
var es2015; // OK: 数字は先頭以外なら利用できる  
var 日本語の変数名; // OK: 一部の漢字や日本語も利用できる
```

変数名に数字を含めることはできますが、変数名を数字から開始することはできません。これは変数名と数値が区別できなくなってしまうためです。

```
var 1st; // NG: 数字から始まっている  
var 123; // NG: 数字のみで構成されている
```

また、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。予約語とは、`var` のように構文として意味を持つキーワードのことです。予約語の一覧は「[予約語 — JavaScript | MDN](#)」^{*1}で確認できますが、基本的には構文として利用される名前が予約されています。

```
var var; // NG: var は変数宣言のために予約されているので利用できない  
var if; // NG: if は if 文のために予約されているので利用できない
```

^{*1} https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Reserved_Words

const は定数ではない

`const` は「再代入できない変数」を定義する変数宣言であり、必ずしも定数を定義するわけではありません。定数とは、一度定義した名前（変数名）が常に同じ値を示すものです。JavaScript でも、`const` 宣言によって定数に近い変数を定義できます。次のように、`const` 宣言によって定義した変数を、変更できないプリミティブな値で初期化すれば、それは実質的に定数です。プリミティブな値とは、数値や文字列などオブジェクト以外のデータです（詳細は「[データ型とリテラル](#)」の章で解説します）。

```
// TEN_NUMBER という変数は常に 10 という値を示す
const TEN_NUMBER = 10;
```

しかし、JavaScript ではオブジェクトなども `const` 宣言できます。次のコードのように、オブジェクトという値そのものは、初期化したあとでも変更できます。

```
// const でオブジェクトを定義している
const object = {
  key: "値"
};
// オブジェクトそのものは変更できてしまう
object.key = "新しい値";
```

このように、`const` で宣言した変数が常に同じ値を示すとは限らないため、定数とは呼べません（詳細は「[オブジェクト](#)」の章で解説します）。

また `const` には、変数名の命名規則はなく、代入できる値にも制限はありません。そのため、`const` 宣言の特性として「再代入できない変数」を定義すると理解しておくのがよいでしょう。

3.5 まとめ

この章では、JavaScript における変数宣言を行うキーワードとして `const`、`let`、`var` があることについて学びました。

- `const` は、再代入できない変数を宣言できる
- `let` は、再代入ができる変数を宣言できる
- `var` は、再代入ができる変数を宣言できるが、いくつかの問題が知られている
- 変数の名前（識別子）には利用できる名前のルールがある

`var` はほとんどすべてのケースで `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。このため変数を宣言する場合には、まず `const` で定義できないかを検討し、できない場合は `let` を使うことを推奨しています。

第4章

値の評価と表示

Chapter 4

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、入力した値を評価してその結果を返すことを示しています。たとえば、次のような値の評価があります。

- `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
- `const x = 1;` という文を評価することで変数を定義するが、この文には返り値はありません

この値の評価方法を確認するために、ウェブブラウザ（以下ブラウザ）を使って JavaScript を実行する方法を見ていきます。

4.1 この書籍で利用するブラウザ

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/>

この書籍で紹介するサンプルコードのほとんどは、Google Chrome、Microsoft Edge、Safari などのブラウザの最新版でも動作します。一方で、古い JavaScript しかサポートしていない Internet Explorer では多くのコードは動作しません。

また、ブラウザによっては標準化されていないエラーメッセージの細かな違いや開発者ツールの使い方の違いなどもあります。この書籍では Firefox で実行した結果を記載しています。そのため、Firefox 以外のブラウザでは細かな違いがあることに注意してください。

4.2 ブラウザで JavaScript を実行する

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。

4.2 ブラウザで JavaScript を実行する



図 4.1 Firefox で Web コンソールを開いた状態

4.2.1 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、開発者ツールのウェブコンソールと呼ばれるパネルに REPL 機能が含まれています。REPL 機能を使うことで、試したいコードをその場で実行できるため、JavaScript の動作を理解するのに役立ちます。

REPL 機能を使うには、まず Firefox の開発者ツールを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）のウェブ開発サブメニューで“ウェブコンソール”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“[ウェブコンソールを開く](https://developer.mozilla.org/ja/docs/Tools/Web_Console/Opening_the_Web_Console)”^{*1}を参照してください。

開発者ツールの“コンソール”タブを選択すると、コマンドライン（二重山カッコから始まる欄）に任意のコードを入力して評価できます。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} https://developer.mozilla.org/ja/docs/Tools/Web_Console/Opening_the_Web_Console

第4章 値の評価と表示

`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、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 行ずつ実行しても、同じ変数名を定義したことになるため構文エラー (`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、開発者ツールを開いているウェブページでリロードすると REPL の実行状態もリセットされます。`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。

4.2.2 HTML ファイルを作成して 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す。ウェブサイトでは HTML に記述した `<script>` タグで JavaScript を読み込んで実行します。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 種類を使って、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](https://atom.io/)^{*2} や [Visual Studio Code](https://code.visualstudio.com/)^{*3} などの JavaScript に対応し

^{*2} <https://atom.io/>

^{*3} <https://code.visualstudio.com/>

4.2 ブラウザで JavaScript を実行する

たエディターを用意しておくともスムーズです。エディターはどんなものでも問題ありませんが、必ず文字コード（エンコーディング）は **UTF-8**、改行コードは **LF** にして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** の中には次のようなコードを書いておきます。

```
index.js
```

```
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** ファイルを読み込んで実行します。**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 ファイルをドラッグアンドドロップするか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:///**からはじ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“ウェブコンソール”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す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。

第4章 値の評価と表示

4.3 Console API

JavaScript の多くの実行環境では、Console API を使ってコンソールに表示します。`console.log(引数)` の引数にコンソール表示したい値を渡すことで、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換えます。そしてページをリロードすると、`1` という値を評価した結果がウェブコンソールに表示されます。

```
index.js
```

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

次のように引数に式を書いた場合は先に引数（`()` 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。

```
index.js
```

```
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。

```
index.js
```

```
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  
const total = 42 + 42;
```



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

4.4 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
```

4.5 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ほとんどありません。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ウェブ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。またエラーには大き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。

4.5.1 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 エンジンでは、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ったら、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)をつけ忘れていたため構文エラーが発生します。

```
index.js

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

第 4 章 値の評価と表示



図 4.2 コンソールに表示された SyntaxError

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に表示されます。

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なります。

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

~~~~~

| エラーの種類 | エラー内容の説明 | 行番号:列番号<br>ファイル名 |
|--------|----------|------------------|
|--------|----------|------------------|

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

Firefox では [詳細] というリンクがエラーメッセージによって表示されます。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<br><br>index.js:1:6 | エラーの種類は <b>SyntaxError</b> で、予期しないものが識別子（変数名）に指定されている<br>例外が <code>index.js</code> の 1 行目 6 列目で発生した |

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点、`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なコードであると解釈され、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。

`cosnt` という変数名 `a` という変数名 = 1;

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ほとんどです。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。

4.5.2 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）に起き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）のですが、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 `x` という存在しない変数を参照したため、**ReferenceError** という実行時エラーが発生しています。

第 4 章 値の評価と表示

```
index.js

const value = "値";
console.log(x); // => ReferenceError: x is not defined
```

ReferenceError: x is not defined{[]詳細[]} index.js:2:1

| メッセージ                            | 意味                                                            |
|----------------------------------|---------------------------------------------------------------|
| ReferenceError: x is not defined | エラーの種類は <b>ReferenceError</b> で、 <b>x</b> という未定義の識別子を参照したため発生 |
| index.js:2:1                     | 例外が <b>index.js</b> の 2 行目 1 列目で発生した                          |

**x** 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。そのため、実行して **x** という識別子を参照したときに、初めて **x** が存在するかが判明し、**x** が存在しない場合は **ReferenceError** となります。

この例では、**value** 変数を参照しているつもりで、**x** という存在しない変数を参照していたのが原因のようです。先ほどのコードは、次のように参照する変数を **value** に変更すれば、エラーが修正できます。

```
const value = "値";
console.log(value); // => "値"
```

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。そのため、エラーの原因を見つけることが大変な場合もあります。しかし、JavaScript はとてもよ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。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。

実行時エラーが発生した際には、発生したエラーの行数の周辺をよく見ることやエラーメッセージを調べてみるのが大切です。

4.6 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してみ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](https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Errors)<sup>\*4</sup>にまとめられています。また [Google](https://www.google.com/)<sup>\*5</sup>、[GitHub](https://github.com/)<sup>\*6</sup>、[Stack Overflow](https://stackoverflow.com/)<sup>\*7</sup>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けるようにしましょう。

---

<sup>\*4</sup> <https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Errors>

<sup>\*5</sup> <https://www.google.com/>

<sup>\*6</sup> <https://github.com/>

<sup>\*7</sup> <https://stackoverflow.com/>

## 第5章

## データ型とリテラル

# Chapter 5

### 5.1 データ型

JavaScript は動的型づけ言語に分類される言語であるため、静的型づけ言語のような**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**と呼びます。

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、真偽値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できないというイミュータブル（immutable）の特性を持ちます。JavaScript では、文字列も一度作成したら変更できないイミュータブルの特性を持ち、プリミティブ型の一種として扱われます。

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ミュータブル（mutable）の特性を持ちます。オブジェクトは、値そのものではなく値への参照を経由して操作されるため、参照型のデータとも言います。

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`: 値が存在しないこと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変な値のデータ型
- 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。

```
console.log(typeof true); // => "boolean"
```

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

残念ながら `typeof null` が `"object"` となるのは、歴史的経緯のある仕様のバグ<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。

一方で、配列とオブジェクトがどちらも `"object"` という判定結果になります。このように、`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。

基本的に `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す。オブジェクトの詳細な種類を判定できないことは、覚えておくとよいでしょう。各オブジェクトの判定方法については、それぞれのオブジェクトの章で見えます。

## 5.2 リテラル

プリミティブ型の値や一部のオブジェクトは、リテラルを使うことで簡単に定義できるようになっています。

リテラルとはプログラム上で数値や文字列など、データ型の値を直接記述できるように構文として定義されたものです。たとえば、`"と"` で囲んだ範囲が文字列リテラルで、これは文字列型のデータを表現しています。

次のコードでは、`"こんにちは"` という文字列型のデータを初期値に持つ変数 `str` を定義しています。

```
// "と"で囲んだ範囲が文字列リテラル
const str = "こんにちは";
```

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、よく利用される主要な値にはリテラルが用意されています。

次の4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
- null

また、オブジェクトの中でもよく利用されるものに関してはリテラル表現が用意されています。

---

<sup>\*1</sup> 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が壊れるため、修正が見送られ現在の挙動が仕様となりました。詳しくは <https://2ality.com/2013/10/typeof-null.html> を参照。

## 第5章 データ型とリテラル

- オブジェクト
- 配列
- 正規表現

これらのリテラルについて、まずはプリミティブ型から順番に見ていきます。

### 5.2.1 真偽値 (Boolean)

真偽値には `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で、見た目通りの意味となります。

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

### 5.2.2 数値 (Number)

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

#### 整数リテラル

整数リテラルには次の4種類があります。

- 10 進数: 数字の組み合わせ
  - ただし、複数の数字を組み合わせた際に、先頭を `0` から開始すると 8 進数として扱われる場合があります
  - 例) `0`、`2`、`10`
- 2 進数: `0b` (または `0B`) の後ろに、`0` または `1` の数字の組み合わせ
  - 例) `0b0`、`0b10`、`0b1010`
- 8 進数: `0o` (または `0O`) の後ろに、`0` から `7` までの数字の組み合わせ
  - `0o` は数字のゼロと小文字アルファベットの `o`
  - 例) `0o644`、`0o777`
- 16 進数: `0x` (または `0X`) の後ろに、`0` から `9` までの数字と `a` から `f` または `A` から `F` のアルファベットの組み合わせ
  - アルファベットの大文字・小文字の違いは値には影響しません
  - 例) `0x30A2`、`0xEEFF`

`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。

```
console.log(1); // => 1
console.log(10); // => 10
console.log(255); // => 255
```

`0b` からはじまる 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。`b` は 2 進数を表



す binary を意味しています。

```
console.log(0b1111); // => 15
console.log(0b10000000000); // => 1024
```

0o からはじまる 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。o は 8 進数を表す octal を意味しています。

```
console.log(0o644); // => 420
console.log(0o777); // => 511
```

次のように、0 からはじまり、0 から 7 の数字を組み合わせた場合も 8 進数として扱われます。しかし、この表記は 10 進数と紛らわしいものであったため、ES2015 で 0o の 8 進数リテラルが新たに導入されました。また、strict mode ではこの書き方は例外が発生するため、次のような 8 進数の書き方は避けるべきです。

```
// 非推奨な 8 進数の書き方
// strict mode は例外が発生
console.log(0644); // => 420
console.log(0777); // => 511
```

0x からはじまる 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。x は 16 進数を表す hex を意味しています。

```
console.log(0xFF); // => 255
// 小文字で書いても意味は同じ
console.log(0xff); // => 255
console.log(0x30A2); // => 12450
```

| 名前    | 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 | 文字のコードポイント、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](#)<sup>\*2</sup>を採用しています。浮動小数点数をリテラルとして書く場合には、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

<sup>\*2</sup> [https://ja.wikipedia.org/wiki/IEEE\\_754](https://ja.wikipedia.org/wiki/IEEE_754)

## 第5章 データ型とリテラル

0からはじ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からはじ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。



変数名を数字からはじ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からです。

eは指数（exponent）を意味する記号で、eのあとには指数部の値を書きます。たとえば、2e8は $2 \times 10^8$ の8乗となるので、10進数で表すと200000000となります。

```
2e8; // => 200000000
```

### 5.2.3 文字列 (String)

文字列リテラル共通のルールとして、同じ記号で囲んだ内容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、その評価結果はすべて同じ"文字列"になります。

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

#### ダブルクォートとシングルクォート

"（ダブルクォート）と'（シングルクォート）はまった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、\'のように\（バックスラッシュ）を使ってエスケープしなければなりません。

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

また、文字列内部に出現しない別のクォート記号を使うことで、エスケープをせずに書くこともできます。

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

ダブルクォートとシングルクォートどちらも、改行をそのままでは入力できません。次のように改行を含んだ文字列は定義できないため、構文エラー（SyntaxError）となります。

```
"複数行の
文字列を
入れたい"; // => SyntaxError: " string literal contains an unescaped line break
```

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";
```

シングルクォートとダブルクォートの文字列リテラルに改行を入れるには、エスケープシーケンスを使わないといけません。これに対して ES2015 から導入されたテンプレートリテラルでは、複数行の文字列を直感的に書くことができます。

### テンプレートリテラル ES2015

テンプレートリテラルは``` (バッククォート) 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のエスケープシーケンス (`\n`) を使わずにそのまま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も持っています。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 str = "文字列";
console.log(`これは${str}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、`\`を使ってエスケープする必要があります。

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

### 5.2.4 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` に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` 値を持つ `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
```

## 第5章 データ型とリテラル

## 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。

次のように、**undefined** はただのグローバル変数であるため、同じ **undefined** という名前のローカル変数を宣言できます。

```
function fn(){
    // undefined という名前の変数をエラーなく定義できる
    var undefined = "独自の未定義値";
    console.log(undefined); // => "独自の未定義値"
}
fn();
```

これに対して **true**、**false**、**null** などはグローバル変数ではなくリテラルであるため、同じ名前の変数を定義することはできません。リテラルは変数名として利用できない予約語のようなものであるため、再定義しようとすると構文エラー (SyntaxError) となります。

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

ここでは、説明のために **undefined** というローカル変数を宣言しましたが、**undefined** の再定義は非推奨です。無用な混乱を生むだけなので避けるべきです。

## 5.2.5 オブジェクトリテラル

JavaScript において、オブジェクトはあらゆるものの基礎となります。そのオブジェクトを作成する方法として、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 **{}** (中カッコ) を書くことで、新しいオブジェクトを作成できます。

```
const obj = {}; // 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** という値を持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 obj = {
    key: "value"
};
```

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

obj の key を参照するには、. (ドット) でつないで参照する方法と、[] (ブラケット) で参照する方法があります。

```
const obj = {
  "key": "value"
};
// ドット記法
console.log(obj.key); // => "value"
// ブラケット記法
console.log(obj["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"
};
// 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"
// NG: ドット記法では、数値からはじまる識別子は利用できない
object.123
```

オブジェクトはとても重要で、これから紹介する配列や正規表現もこのオブジェクトが元となっています。詳細は「[オブジェクト](#)」の章で解説します。ここでは、オブジェクトリテラル（{と}）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思ってください。

### 5.2.6 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[]でカンマ区切りの値を囲み、その値を持つ 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]; // 値を持った配列を作成
```

配列は 0 からはじまるインデックス（添字）に、対応した値を保持しています。作成した配列の要素を取得するには、配列に対して array[index] という構文で指定したインデックスの値を参照できます。

```
const array = ["index:0", "index:1", "index:2"];
// 0 番目の要素を参照
console.log(array[0]); // => "index:0"
```

## 第5章 データ型とリテラル

```
// 1 番目の要素を参照
console.log(array[1]); // => "index:1"
```

配列についての詳細は「[配列](#)」の章で解説します。

### 5.2.7 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) からはじ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できます。しかし、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。

## 5.3 プリミティブ型とオブジェクト

プリミティブ型は基本的にリテラルで表現しますが、真偽値 (**Boolean**)、数値 (**Number**)、文字列 (**String**) はそれぞれオブジェクトとして表現する方法もあります。これらはプリミティブ型の値をラップしたようなオブジェクトであるため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は、**new** 演算子と対応するコンストラクタ関数を利用して作成できます。たとえば、文字列のプリミティブ型に対応するコンストラクタ関数は **String** となります。

次のコードでは、**String** のラッパーオブジェクトを作成しています。ラッパーオブジェクトは、名前のとおりオブジェクトの一種であるため **typeof** 演算子の結果も **"object"** です。また、オブジェクトであるため **length** プロパティなどのオブジェクトが持つプロパティを参照できます。

```
// 文字列をラップしたString ラッパーオブジェクト
const str = new String("文字列");
// ラッパーオブジェクトは"object"型のデータ
console.log(typeof str); // => "object"
// String オブジェクトの length プロパティは文字列の長さを返す
console.log(str.length); // => 3
```

しかし、明示的にラッパーオブジェクトを使うべき理由はありません。なぜなら、JavaScript ではプリミティブ型のデータに対してもオブジェクトのように参照できる仕組みがあるためです。次のコードでは、プリミティブ型の文字列データに対しても **length** プロパティへアクセスできています。

```
// プリミティブ型の文字列データ
const str = "文字列";
// プリミティブ型の文字列は"string"型のデータ
console.log(typeof str); // => "string"
// プリミティブ型の文字列もlength プロパティを参照できる
console.log(str.length); // => 3
```

これは、プリミティブ型のデータのプロパティへアクセスする際に、対応するラッパーオブジェクトへ暗黙的に変換してからプロパティへアクセスするためです。また、ラッパーオブジェクトを明示的に作成するには、リテラルに比べて冗長な書き方が必要になります。このように、ラッパーオブジェクトを明示的に作成する必要はないため、常にリテラルでプリミティブ型のデータを表現することを推奨します。

このラッパーオブジェクトへの暗黙的な型変換の仕組みについては「[ラッパーオブジェクト](#)」の章で解説します。現時点では、プリミティブ型のデータであってもオブジェクトのようにプロパティ（メソッドなども含む）を参照できるということだけを知っていれば問題ありません。

## 5.4 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 種類のプリミティブ型とオブジェクトがある
- リテラルはデータ型の値を直接記述できる構文として定義されたもの
- プリミティブ型の真偽値、数値、文字列、null はリテラル表現がある
- オブジェクト型のオブジェクト、配列、正規表現にはリテラル表現がある
- プリミティブ型のデータでもプロパティアクセスができる

## 第6章

### 演算子

# Chapter 6

演算子はよく利用する演算処理を記号などで表現したものです。たとえば、足し算をする `+` も演算子の一種です。これ以外にも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持ちます。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、`+` 演算子の対象となっている `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す。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取るものもあります。たとえば、数値をインクリメントする `++` 演算子は、次のように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るため、呼び方を変えています。

この章では、演算子ごとにそれぞれ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、演算子をひとつと見つけた後に、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。

演算子の種類は多いため、すべての演算子の動作をここで覚える必要はありません。必要となったタイミングで、改めてその演算子の動作を見るのがよいでしょう。



## 6.1 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。

### 6.1.1 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
```

### 6.1.2 文字列結合演算子 (+)

数値の加算に利用したプラス演算子 (+) は、文字列の結合に利用できます。

文字列結合演算子 (+) 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"
```

つまり、プラス演算子 (+) は数値同士と文字列同士の演算を行います。

### 6.1.3 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
```

### 6.1.4 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
```

### 6.1.5 除算演算子 (/)

2つの数値を除算する演算子です。

## 第 6 章 演算子

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

### 6.1.6 剰余演算子 (%)

2 つの数値の余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
```

### 6.1.7 べき乗演算子 (\*\*) ES2016

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、関数と演算子が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。

## 6.2 単項演算子 (算術)

単項演算子は、1 つのオペランドを受け取って処理する演算子です。

### 6.2.1 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、単項プラス演算子をつ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。

### 6.2.2 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のは、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項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
```

## 第6章 演算子

### 6.2.3 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置くことで、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、オペランドの後ろ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言えます。

### 6.2.4 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
```

## 6.3 比較演算子

比較演算子はオペランド同士の値を比較し、真偽値を返す演算子です。

### 6.3.1 厳密等価演算子 (===)

厳密等価演算子は、左右の2つのオペランドを比較します。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、オペランドがどちらもオブジェクトであるときは、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
```

## 第6章 演算子

### 6.3.2 厳密不等価演算子 (!==)

厳密不等価演算子は、左右の2つのオペランドを比較します。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。

### 6.3.3 等価演算子 (==)

等価演算子 (`==`) は、2つのオペランドを比較します。同じデータ型のオペランドを比較する場合は、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をしてから比較します。

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（===）では二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、等価演算子（==）は暗黙的な型変換をするため、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。

### 6.3.4 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、暗黙的な型変換をしてから比較します。

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

## 第 6 章 演算子

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しない厳密不等価演算子 (`!==`) を利用します。

### 6.3.5 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
```

### 6.3.6 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
```

### 6.3.7 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
```

### 6.3.8 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
```

## 6.4 ビット演算子

ビット演算子はオペランドを符号つき 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 という数値は符号つき 32bit 整数では次のように表現されます。



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、次のようになります。

### 6.4.1 ビット論理積 (&)

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

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
```

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
```

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 の補数として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 (`~`) はビット演算以外にも使われていることがあります。

文字列 (String オブジェクト) が持つ `indexOf` メソッドは、マッチする文字列を見つけて、そのイ

## 第6章 演算子

インデックス（位置）を返すメソッドです。この `indexOf` メソッドは、検索対象が見つからない場合には `-1` を返します。

```
const str = "森森本森森";  
// 見つかった場合はインデックスを返す  
// JavaScript のインデックスは 0 から開始するので 2 を返す  
console.log(str.indexOf("本")); // => 2  
// 見つからない場合は -1 を返す  
console.log(str.indexOf("火")); // => -1
```

否定演算子 (`~`) は 1 の補数を返すため、`~(-1)` は 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 となるのは、その文字列が見つからなかった場合だけとなります。次のコードのような否定演算子 (`~`) と `indexOf` メソッドを使ったイディオムが一部では使われていました。

```
const str = "森森木森森";  
// indexOf メソッドは見つからなかった場合は -1 を返す  
if (str.indexOf("木") !== -1) {  
    console.log("木を見つけました");  
}  
// 否定演算子 (~) で同じ動作を実装  
// (~(-1)) は 0 となるため、見つからなかった場合は if 文の中身は実行されない  
if (~str.indexOf("木")) {  
    console.log("木を見つけました");  
}
```

ES2015 では、文字列 (String オブジェクト) に `includes` メソッドが実装されました。`includes` メソッドは指定した文字列が含まれているかを真偽値で返します。

```
const str = "森森木森森";  
if (str.includes("木")) {  
    console.log("木を見つけました");  
}
```

そのため、否定演算子 (`~`) と `indexOf` メソッドを使ったイディオムは、`includes` メソッドに置き換えられます。



## 第6章 演算子

```
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; と同じ
console.log(num); // => 11
```

### 6.5.1 分割代入 (Destructuring assignment) ES2015

今まで見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、左辺のオペランド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書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 obj = {
  "key": "value"
};
// プロパティ名 key の値を、変数 key として定義する
const { key } = obj;
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 obj = {
  "key": "value"
};
const key = obj.key;
```

## 6.6 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` のとき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` のとき処理する式の評価結果を返します。

条件式 ? `True` のとき処理する式 : `False` のとき処理する式;

`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つ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:"をつける
  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:";
```

## 第6章 演算子

```
    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"
```

## 6.7 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。

### 6.7.1 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それ以上評価しないことを短絡評価 (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` 型で `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ないかを検討するべきサインとなります。

## 6.7.2 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。");
}
```

## 6.7.3 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 str = "";
// 空文字列は falsy な値
console.log(!!str); // => 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

## 第6章 演算子

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 str = "";
// 空文字列でないことを判定
console.log(str.length > 0); // => false
```

#### 6.7.4 グループ化演算子（（と））

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

たとえば、次のようにグループ化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
```

[演算子の優先順位](#)<sup>\*1</sup>は ECMAScript 仕様で定義されていますが、演算子の優先度をすべて覚えるのは難しいです。演算子の優先順位の中でグループ化演算子は優先される演算子となり、グループ化演算子を使って優先順位を明示できます。

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

```
if (x || y && z) {
  // x が true または
  // y かつ z 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化演算子を使い、結合順を明示して書くようにしましょう。

```
if (x || (y && z)) {
  // x が true または
  // y かつ z が true
}
```

しかし、ひとつの式で多数の演算を行うよりも、式自体を分けたほうが読みやすい場合もあります。

次のように `a` と `b` が文字列型または `x` と `y` が数値型の場合に処理する `if` 文を考えてみます。グループ化演算子を使い、そのまま1つの条件式で書くことも可能ですが、読みにくくなってしまいます。

---

<sup>\*1</sup> [https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Operators/Operator\\_Precedence#Table](https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Operators/Operator_Precedence#Table)



```
if ((typeof a === "string" && typeof b === "string") || (typeof x === "number"
    && typeof y === "number")) {
    // a と b が文字列型 または
    // x と y が数値型
}
```

このように無理して1つの式(1行)で書くよりも、条件式を分解してそれぞれの結果を変数として定義したほうが読みやすくなる場合もあります。

```
const isAbString = typeof a === "string" && typeof b === "string";
const isXyNumber = typeof x === "number" && typeof y === "number";
if (isAbString || isXyNumber) {
    // a と b が文字列型 または
    // x と y が数値型
}
```

そのため、グループ化演算子ですべての条件をまとめるのではなく、それぞれの条件を分解して名前をつける(変数として定義する)ことも重要です。

## 6.8 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、`const` 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ほとんどないため、「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っていれば問題ありません<sup>\*2</sup>。

## 6.9 まとめ

この章では演算子について学びました。

- 演算子はよく利用する演算処理を記号などで表現したもの

---

<sup>\*2</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。<https://2ality.com/2014/01/eval.html>

## 第 6 章 演算子

- 四則演算から論理演算などさまざまな種類の演算子がある
- 演算子には優先順位が定義されており、グループ化演算子で明示できる

## 第7章

### 暗黙的な型変換

# Chapter 7

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（`==`）ではなく厳密等価演算子（`===`）の利用を推奨していました。これは厳密等価演算子（`===`）が暗黙的な型変換を行わずに、値同士を比較できるためです。

厳密等価演算子（`===`）では異なるデータ型を比較した場合に、その比較結果は必ず `false` となります。次のコードは、数値の `1` と文字列の `"1"` という異なるデータ型を比較しているので、結果は `false` となります。

```
// ===では、異なるデータ型の比較結果はfalse
console.log(1 === "1"); // => false
```

しかし、等価演算子（`==`）では異なるデータ型を比較した場合に、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。次のコードでは、数値の `1` と文字列の `"1"` の比較結果が `true` となっています。これは、等価演算子（`==`）は右辺の文字列 `"1"` を数値の `1` へと暗黙的な型変換してから、比較を行うためです。

```
// ==では、異なるデータ型は暗黙的な型変換をしてから比較される
// 暗黙的な型変換によって 1 == 1 のように変換されてから比較される
console.log(1 == "1"); // => true
```

このように、暗黙的な型変換によって意図しない結果となるため、比較には厳密等価演算子（`===`）を使うべきです。

別の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のような異なるデータ型同士の加算はエラーとなります。しかし、JavaScript では暗黙的な型変換が行われてから加算されるため、エラーなく処理されます。

次のコードでは、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。

```
// 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
```

## 第7章 暗黙的な型変換

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。このように、暗黙的な型変換はできる限り避けるべき挙動です。

この章では、次のことについて学んでいきます。

- 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こと

## 7.1 暗黙的な型変換とは

暗黙的な型変換とは次のことを言います。

- ある処理において、その処理過程で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を中心に見ていきます。

### 7.1.1 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
console.log(1 == "1"); // => true
console.log(0 == false); // => true
console.log(10 == ["10"]); // => true
```

このほか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を覚えるのは現実的ではありません。

しかし、等価演算子の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、暗黙的な型変換をせずに値を比較できます。

```
console.log(1 === "1"); // => false
console.log(0 === false); // => false
console.log(10 === ["10"]); // => false
```

厳密等価演算子（`===`）を使うことで、意図しない比較結果を避けることができます。そのため、比較には等価演算子（`==`）ではなく厳密等価演算子（`===`）を使うことを推奨します。

### 7.1.2 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型変換が行われる
"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イ演算子 (-) の定義はありません。そのため、マイナスイ演算子の対象となる数値への暗黙的な型変換が行われます。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型変換が行われる
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 + 演算子で演算する場合に、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"
console.log(x + z + y); // => "42"
```

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。

## 7.2 明示的な型変換

プリミティブ型へ明示的な型変換をする方法を見ていきます。

## 第 7 章 暗黙的な型変換

## 7.2.1 任意の値 → 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`
- `""` (空文字列)

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ときに、真偽値へと暗黙的に変換されてから判定されます。

```
// 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 (`===`) を使って比較します。

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

```
}
```

### 7.2.2 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` コンストラクタ関数は、数値以外にもいろいろ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("シンボルの説明文")); // => "Symbol(シンボルの説明文)"
// 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() {}); // "function() {}"
```

上記の結果からもわかるように `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の変換では、見た目どおりの文字列を得ることができます。

一方、オブジェクトに対しては、あまり意味のある文字列を返しません。オブジェクトに対しては `String` コンストラクタ関数より適切な方法があるためです。配列には `join` メソッド、オブジェクトには `JSON.stringify` メソッドなど、より適切な方法があります。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。

### 7.2.3 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られ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いってプラス演算子の結果が必ず文字列になるとは限らないことがわかります。

次のコードでは、シンボルを文字列結合演算子 (+) で文字列に変換できないという `TypeError` が発生しています。

```
"文字列と" + Symbol("シンボルの説明");
// => TypeError: can't convert symbol to string
```

## 第7章 暗黙的な型変換

この問題も `String` コンストラクタ関数を使って、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)"
```

## 7.2.4 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、ユーザー入力として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 num = Number(input);
console.log(typeof num); // => "number"
console.log(num); // 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て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を指定します。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
console.log(Number.parseInt("1", 10)); // => 1
// 余計な文字は無視してパースした結果を返す
console.log(Number.parseInt("42px", 10)); // => 42
console.log(Number.parseInt("10.5", 10)); // => 10
// 文字列をパースして浮動小数点数として取り出す
console.log(Number.parseFloat("1")); // => 1
console.log(Number.parseFloat("42.5px")); // => 42.5
console.log(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には、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` メソッドで判定できます。

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

### 7.2.5 NaN は Not a Number だけど Number 型

ここで、数値への型変換でたびたび現れる NaN という値について詳しく見ていきます。NaN は Not a Number の略称で、特殊な性質を持つ 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<sup>\\*1</sup>](https://en.cppreference.com/ja/cpp/numeric/NaN)で規定されており、JavaScript だけの性質ではありません。

NaN という値を作る方法は簡単で、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ないデータです。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
console.log(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ものがあります。この特徴を利用することで、ある値が NaN であるかを判定できます。

```
function isNaN(x) {
```

---

<sup>\*1</sup> [https://ja.wikipedia.org/wiki/IEEE\\_754](https://ja.wikipedia.org/wiki/IEEE_754)

## 第7章 暗黙的な型変換

```

    // NaN は自分自身と一致しない
    return x !== x;
}

console.log(isNaN(1)); // => false
console.log(isNaN("str")); // => false
console.log(isNaN({})); // => false
console.log(isNaN([])); // => false
console.log(isNaN(NaN)); // => true

```

同様の処理をする方法として `Number.isNaN` 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` 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` となったのかがわかりにくく、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)` と呼び出したときの結果が `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
console.log(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できません。つまり、次のように明示的な型変換をしても解決できないことがわかります。

```
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
console.log(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、大きく分けて次の2つがあります。

- 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ということですが、どちらも行うことがより安全なコードにつながります。

まずは、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**<sup>\*2</sup>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 文で例外を投げることで、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。
 * 1 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);
}

```

<sup>\*2</sup> <http://usejsdoc.org/>

## 第7章 暗黙的な型変換

```
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
console.log(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、エラーとなったとき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してしまうなど、驚くほど曖昧さを許容しています。そのため、大きな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が重要です。

## 7.3 明示的な変換でも解決しないこと

先ほどの例からもわかるように、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` になってしまうと `Number.isNaN` 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。

### 7.3.1 空文字列かどうかを判定する

たとえば、文字列が空文字列なのかを判定したい場合を考えてみましょう。`""`（空文字列）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` な値は空文字列以外にもあるため、明示的に変換したからといって空文字列だけを判定できるわけではありません。

次のコードでは、明示的な型変換をしています。0 も空文字列となってしまう意図しない挙動になっています。

```
// 空文字列かどうかを判定
function isEmptyString(str) {
  // str が falsy な値なら、isEmptyString 関数は true を返す
  return !Boolean(str);
}
// 空文字列列の場合は、true を返す
console.log(isEmptyString("")); // => true
// falsy な値の場合は、true を返す
console.log(isEmptyString(0)); // => true
// undefined の場合は、true を返す
console.log(isEmptyString()); // => true
```

ほとん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、空文字列とは「String 型で文字長が 0 の値」とであると定義することで、`isEmptyString` 関数を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、値が空文字列であるかを正しく判定できるようになりました。

```
// 空文字列かどうかを判定
function isEmptyString(str) {
  // String 型で length が 0 の値の場合は true を返す
  return typeof str === "string" && str.length === 0;
}
console.log(isEmptyString("")); // => true
// falsy な値でも正しく判定できる
console.log(isEmptyString(0)); // => false
console.log(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。

## 7.4 まとめ

この章では暗黙的な型変換と明示的な型変換について学びました。

- 暗黙的な型変換は意図しない結果となりやすいため避ける
- 比較には等価演算子 (`==`) ではなく、厳密等価演算子 (`===`) を利用する
- 演算子による暗黙的な型変換より、明示的な型変換を行う関数を利用する
- 真偽値を得るには、明示的な型変換以外の方法もある

## 第8章

### 関数と宣言

# Chapter 8

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する 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。

#### 8.1 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) {
    // 関数が呼び出されたときの処理
    // ...
    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つの要素で構成されています。

- 関数名 — 利用できる名前は変数名と同じ（「[変数名に使える名前のルール](#)」を参照）
- 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 — `{ }` 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）で区切ります。

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** 関数には **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 **return** 文で返しています。

```
function multiple(num) {  
    return num * 2;  
}  
// multiple 関数の戻り値は、num に 10 を入れて return 文で返した値  
console.log(multiple(10)); // => 20
```

関数で **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** 文では戻り値を省略できます。**return** 文の戻り値を省略した場合は、未定義の値である **undefined** を返します。

```
function fn() {  
    // 何も戻り値を指定してない場合は undefined を返す  
    return;  
    // すでに return されているため、この行は実行されません  
}  
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
```

## 8.2 関数の引数

JavaScript では、関数に定義した仮引数の個数と実際に呼び出したときの引数の個数が違ってても、関数を呼び出せます。そのため、引数の個数が合ってい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。

### 8.2.1 呼び出し時の引数が少ないとき

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。

## 第8章 関数と宣言

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;  
}  
  
console.log(echo(1)); // => 1  
console.log(echo()); // => undefined
```

複数の引数を受けつ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];  
}  
  
console.log(argumentsToArray(1, 2)); // => [1, 2]  
// 仮引数の x には 1、y には undefined が入る  
console.log(argumentsToArray(1)); // => [1, undefined]
```

### 8.2.2 デフォルト引数 ES2015

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。先ほどの `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;  
}  
  
console.log(echo(1)); // => 1
```



```
console.log(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 な値の場合に右辺のオペランドを評価します。falsy な値とは、真偽値へと変換すると false となる次のような値のことです。

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

OR 演算子 (||) を使った場合、次のように prefix に空文字列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場合があります。

```
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、このような挙動は起きなくなるため安全です。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;  
}
```

## 第 8 章 関数と宣言

```
}  
// 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"
```

### 8.2.3 呼び出し時の引数が多いとき

関数の仮引数に対して引数の個数が多い場合、あふれた引数は単純に無視されます。

次のコードでは、2 つの引数を足し算した値を返す `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
```

## 8.3 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** と呼びます。

```
// 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。

### 8.3.1 Rest parameters ES2015

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引数の値が、`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); // => ["a", "b", "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。ほかの仮引数と組み合わせる際には、必ず Rest parameters を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");
```

Rest parameters は、引数をまとめた配列を仮引数に定義する構文でした。一方で、配列を展開して関数の引数に渡す Spread 構文もあります。

Spread 構文は、配列の前に `...` をつけた構文のことで、関数には配列の値を展開したものが引数として渡されます。次のコードでは、`array` の配列を展開して `fn` 関数の引数として渡しています。

```
function fn(x, y, z) {
  console.log(x); // => 1
  console.log(y); // => 2
  console.log(z); // => 3
}
const array = [1, 2, 3];
// Spread 構文で配列を引数に展開して関数を呼び出す
fn(...array);
// 次のように書いたのと同じ意味
fn(array[0], array[1], array[2]);
```

### 8.3.2 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() {
```

## 第8章 関数と宣言

```
// 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つけるのかを仮引数だけを見て判断できない

**arguments** 変数は仮引数の定義とは関係なく、実際に渡された引数がすべて含まれています。そのため、関数の仮引数の定義部分だけ見ても、実際に関数の要求する引数がわからないという問題を作りやすいです。Rest parameters であれば、仮引数で可変長を受け入れることが明確になります。

このように、可変長引数が必要な場合は **arguments** 変数よりも、Rest parameters での実装を推奨します。

## 8.4 関数の引数と分割代入 **ES2015**

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。

```
// 第一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わかります。

```
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);
```

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。次のコードでは、引数に渡された配列の 1 番目の要素が **first** に、2 番目の要素が **second** に代入されます。

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

## 8.5 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 () をつけることで、関数として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。また、関数はほか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。

## 第8章 関数と宣言

```
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。

先ほどのコードでは、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できます。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。

### 8.5.1 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、関数式では関数を値として扱っています。これは、文字列や数値などの変数宣言と同じ定義方法です。

```
// 関数式  
const 変数名 = function() {  
    // 関数を呼び出したときの処理  
    // ...  
    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つけることができます。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に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
```

### 8.5.2 Arrow Function ES2015

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。

```
// Arrow Function を使った関数定義
const 変数名 = () => {
  // 関数を呼び出したときの処理
  // ...
  return 関数の返す値;
};
```

Arrow Function には書き方にいくつかの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) => { /* 仮引数が複数のとき */ };
// 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
```

## 第8章 関数と宣言

```
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。

- 名前をつけることができない（常に匿名関数）
- `this` が静的に決定できる（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` 変数を参照できない

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。配列の `map` メソッドは、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` 文を省略して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。

次のコードは、先ほどの `function` キーワードで書いたコールバック関数と同じ結果になります。

```
const array = [1, 2, 3];
// 仮引数が1つなので()を省略できる
// 関数の処理が1つの式なのでreturn文を省略できる
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 では参照できなくなっています。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があります。`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。



#### 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。そのため、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。

次のコードでは、**fn** という関数名を 2 つ定義していますが、最後に定義された **fn** 関数が優先されています。また、仮引数の定義が異なっても、関数の名前が同じなら上書きされます。

```
function fn(x) {  
    return `最初の関数 x: ${x}`;  
}  
function fn(x, y) {  
    return `最後の関数 x: ${x}, y: ${y}`;  
}  
console.log(fn(2, 10)); // => "最後の関数 x: 2, y: 10"
```

この関数定義の上書きは **function** キーワードでの関数宣言と **var** キーワードを使った関数式のみで発生します。**let** や **const** では同じ変数名の定義はエラーとなるため、このような関数定義の上書きもエラーとなります。

このように、同じ関数名で複数の関数を定義することは基本的に意味がないため避けるべきです。引数の違いで関数を呼び分けたい場合は、別々の名前で関数を定義するか関数の内部で引数の値で処理を分岐する必要があります<sup>a</sup>。

<sup>a</sup> JavaScript にはオーバーロードと呼ばれる機能はありません。

### 8.5.3 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、配列の **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);
```

## 第8章 関数と宣言

```
// 次のように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。

## 8.6 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。

次のコードでは、`obj` の `method1` プロパティと `method2` プロパティに関数を定義しています。この `obj.method1` プロパティと `obj.method2` プロパティがメソッドです。

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

次のように空オブジェクトの `obj` を定義してから、`method` プロパティへ関数を代入してもメソッドを定義できます。

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

### 8.6.1 メソッドの短縮記法 **ES2015**

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 obj = {
  method() {
    return "this is method";
  }
};
console.log(obj.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ほうがよいでしょう。

## 8.7 まとめ

この章では、次のことについて学びました。

- 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。

## 第9章

### 文と式

# Chapter 9

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。

#### 9.1 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が得られます。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
```

## 9.2 文

文 (Statement) を簡潔に述べると、処理する 1 ステップが 1 つの文と言えます。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つけます。

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はなりません。

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。次のようなコードは構文として問題があるため、構文エラー (SyntaxError) となります。

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

### 9.2.1 式文

一方で、式 (Expression) は文 (Statement) になれます。文となった式のことを**式文**と呼びます。基本的に文が書ける場所には式を書けます。

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けている  
式;
```

式は文になれますが、先ほどの if 文のように文は式になれません。

### 9.2.2 ブロック文

次のような、文を{ }で囲んだ部分を**ブロック**と言います。ブロックには、複数の文が書けます。

```
{  
  文;
```

## 第9章 文と式

```
    文;  
}
```

ブロック文は単独でも書けますが、基本的には if 文や for 文など他の構文と組み合わせて書くことがほとんどです。次のコードでは、if 文とブロック文を組み合わせることで、if 文の処理内容に複数の文を書いています。

```
// if 文とブロック文の組み合わせ  
if (true) {  
    console.log("文 1");  
    console.log("文 2");  
}
```

文の末尾にはセミコロンをつ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);  
}
```

## 9.3 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、ここまでの内容から説明できます。

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。

## 単独のブロック文の活用

アプリケーションのソースコードに if 文などと組み合わせない単独のブロック文を書くことはほとんどありません。しかし、REPL で同じコードの一部を変更して実行を繰り返している場合には、単独のブロック文が役に立つ機会もあります。

REPL では、次のように同じ変数名を再定義すると、構文エラーが発生します（詳細は「[変数と宣言](#)」の章の「[var の問題](#)」を参照）。そのため、同じコードを再び実行するには、ブラウザでページをリロードして変数定義をリセットしないといけませんでした。

```
// REPL での動作。>>は REPL の入力欄
>> const count = 1;
undefined
>> const count = 2;
SyntaxError: redeclaration of const count
```

この問題は単独のブロック文で変数定義を囲むことで回避できます。ブロック文（`{}`）の中で `let` や `const` を用いて変数を定義しても、そのブロック文の外には影響しません。そのため、次のようにブロック文で囲んでおけば、同じ変数名を定義しても構文エラー（`SyntaxError`）にはなりません。

```
// REPL での動作。>>は REPL の入力欄
>> {
    const count = 1;
}
undefined // ここでブロック内で定義した変数 count は参照できなくなる
>> {
    const count = 1;
}
undefined // ここでブロック内で定義した変数 count は参照できなくなる
```

これは、ブロックスコープという仕組みによるものですが、詳しい仕組みについては「[関数とスコープ](#)」の章で解説します。今は、ブロック文を使うと REPL での試行錯誤がしやすいということだけ知っていれば問題ありません。

「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わかります。つまり、次のように置き換えても同じと言えるため、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;
```

## 第9章 文と式

## 9.4 まとめ

この章では次のことについて学びました。

- JavaScript は文 (Statement) と式 (Expression) から構成される
- 文は式になれない
- 式は文になれる (式文)
- 文の末尾にはセミコロンをつける
- ブロックで終わる文は例外的にセミコロンをつけなくてよい

JavaScript には、特殊なルールに基づき、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持っています。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。

エディターや 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](http://eslint.org/)<sup>\*1</sup>などの Lint ツールを使うことで、セミコロンが必要な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、ツールを使って静的にチェックできます。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言えます。

---

\*1 <http://eslint.org/>



## 第10章

### 条件分岐

# Chapter 10

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。条件分岐を使うことで、特定の条件を満たすかどうかで行う処理を変更できます。

#### 10.1 if 文

if 文を使うことで、プログラム内に条件分岐を書けます。

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("この行は実行されます");  
}
```

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("この行は実行されます");
```

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");  
}
```

## 第 10 章 条件分岐

```
}
```

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てから、条件式として判定します。

真偽値へ変換すると `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。次の値は真偽値へと変換すると `false` となるため、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`
- `""` (空文字列)

これ以外の値は真偽値に変換すると `true` になります。そのため、**"文字列"** や `0` 以外の数値などを条件式に指定した場合は、`true` へと変換してから条件式として判定します。

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

```
if (true) {
    console.log("この行は実行されます");
}
if ("文字列") {
    console.log("この行は実行されます");
}
if (42) {
    console.log("この行は実行されます");
}
if (["配列"]) {
    console.log("この行は実行されます");
}
if ({ name: "オブジェクト" }) {
    console.log("この行は実行されます");
}
```

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) {
    // この行は実行されません
}
if ("" ) {
```

```
// この行は実行されません
}
if (0) {
    // この行は実行されません
}
if (undefined) {
    // この行は実行されません
}
if (null) {
    // この行は実行されません
}
```

### 10.1.1 else if 文

複数の条件分岐を書く場合は、if 文に続けて else if 文を使います。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。変数 `version` の値が “ES6” であるため、コンソールには “ECMAScript 2015” が出力されます。

```
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");
}
```

### 10.1.2 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。一方で条件に一致しなかった場合の処理は、else 文を使うことで書けます。

次のコードでは、変数 `num` の数値が 10 より大きいかを判定しています。`num` の値は 10 以下であるため、else 文で書いた処理が実行されます。

```
const num = 1;
```

## 第10章 条件分岐

```
if (num > 10) {  
    console.log(`num は 10 より大きいです: ${num}`);  
} else {  
    console.log(`num は 10 以下です: ${num}`);  
}
```

## ネストした if 文

if 文、else if 文、else 文はネストして書けます。次のように複数の条件を満たすかどうかを if 文のネストとして表現できます。

```
if (条件式 A) {  
    if (条件式 B) {  
        // 条件式 A と条件式 B が true ならば実行される文  
    }  
}
```

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

西暦での現在の年は `new Date().getFullYear()` で取得できます。このうるう年の条件を if 文で表現すると次のように書けます。

```
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(`${year}年はうるう年です`);  
        } else {  
            console.log(`${year}年はうるう年ではありません`);  
        }  
    } else {  
        console.log(`${year}年はうるう年です`);  
    }  
} else {  
    console.log(`${year}年は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います。一般的にはネストは浅いほうが、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(`${year}年はうるう年です`);
} else if (year % 100 === 0) { // 100 で割り切れる
  console.log(`${year}年は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(`${year}年はうるう年です`);
} else { // それ以外
  console.log(`${year}年はうるう年ではありません`);
}
```

## 10.2 switch 文

switch 文は、次のような構文で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 と一致す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基づく条件分岐を扱います。また break 文は、switch 文から抜けて switch 文の次の文から実行するためのものです。次のコードでは、**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":
```

## 第 10 章 条件分岐

```
        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
}
```

次に式の評価結果が厳密等価演算子 (===) で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;
}
```

### 10.2.1 break 文

switch 文の case 節では基本的に **break**; を使って switch 文を抜けるようにします。この **break**; は省略が可能です。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("知らないバージョンです");
}
/*
    "ECMAScript 2015"
    "ECMAScript 2016"
    "知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 case 節が実行されてしまいます。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、別の方法で書けないかを考えるべきサインとなります。

switch 文は if 文の代用として使うのではなく、次のように関数と組み合わせて条件に対する値を返すパターンとして使うことが多いです。関数については「[関数と宣言](#)」の章で紹介します。

```
function getECMAScriptName(version) {
```

## 第 10 章 条件分岐

```
    switch (version) {  
        case "ES5":  
            return "ECMAScript 5";  
        case "ES6":  
            return "ECMAScript 2015";  
        case "ES7":  
            return "ECMAScript 2016";  
        default:  
            return "知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"
```

### 10.3 まとめ

この章では条件分岐について学びました。

- if 文、else if 文、else 文で条件分岐した処理を扱える
- 条件式に指定した値は真偽値へと変換してから判定される
- 真偽値に変換すると **false** となる値を falsy と呼ぶ
- switch 文と case 節、default 節を組み合わせで条件分岐した処理を扱える
- case 節で break 文しない場合は引き続き case 節が実行される

条件分岐には if 文や switch 文を利用します。複雑な条件を定義する場合には、if 文のネストが深くなりやすいです。そのような場合には、条件式自体を見直してよりシンプルな条件にできないかを考えてみることも重要です。



## 第11章

### ループと反復処理

# Chapter 11

この章では、while 文や for 文などの基本的な反復処理と制御文について学んでいきます。

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繰り返し書く必要はありません。ループやイテレータなどを使い、反復処理として同じ処理を繰り返し実行できます。

また、for 文などのような構文だけではなく、配列のメソッドを利用して反復処理を行う方法もあります。配列のメソッドを使った反復処理もよく利用されるため、合わせて見ていきます。

#### 11.1 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** なら次のステップへ、**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;  
console.log(`ループ開始前の x の値: ${x}`);  
while (x < 10) {  
    console.log(x);  
    x += 1;  
}  
console.log(`ループ終了後の x の値: ${x}`);
```

## 第 11 章 ループと反復処理

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、ほか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。

## 無限ループ

反復処理を扱う際に、コードの書き間違いや条件式のミスなどから無限ループを引き起こしてしまう場合があります。たとえば、次のコードは条件式の評価結果が常に `true` になってしまうため、無限ループが発生してしまいます。

```
let i = 1;
// 条件式が常に true になるため、無限ループする
while (i > 0) {
  console.log(`${i}回目のループ`);
  i += 1;
}
```

無限ループが発生してしまったときは、あわてずにスクリプトを停止してからコードを修正しましょう。

ほとんどのブラウザは無限ループが発生した際に、自動的にスクリプトの実行を停止する機能が含まれています。また、ブラウザで該当のスクリプトを実行しているページ（タブ）またはブラウザそのものを閉じることで強制的に停止できます。Node.js で実行している場合は **Ctrl** + **C** を入力し、終了シグナルを送ることで強制的に停止できます。

無限ループが発生する原因のほとんどは条件式に関連する実装ミスです。まずは条件式の確認をしてみることで問題を解決できるはずです。

## 11.2 do-while 文

`do-while` 文は `while` 文とほとん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` なら次のステップへ、`false` なら終了
3. ステップ 1 へ戻る

`while` 文とは異なり、必ず最初に**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);
```

この仕組みをうまく利用して、ループの開始前とループ中の処理をまとめて書けます。しかし、while 文と同じくほか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。

## 11.3 for 文

for 文は繰り返す範囲を指定した反復処理を書けます。

```
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** なら次のステップへ、**false** なら終了
3. 実行する文 を実行
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 文で 1 から 10 までの値を合計して、その結果をコンソールへ出力しています。

```
let total = 0; // total の初期値は 0
// for 文の実行フロー
// i を 0 で初期化
// i が 10 未満（条件式を満たす）なら for 文の処理を実行
// i に 1 を足し、再び条件式の判定へ
for (let i = 0; i < 10; i++) {
  total += i + 1; // 1 から 10 の値を total に加算している
}
console.log(total); // => 55
```

このコードは 1 から 10 まで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。もう少し実用的なものを考えると、任意の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と良さそうです。

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。**numbers** 配列に含まれている要素を先頭から順番に変数 **total** へ加算することで合計値を計算しています。

## 第 11 章 ループと反復処理

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

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

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。そのため、配列のメソッドを使った反復処理も合わせて見ていきます。

## 11.4 配列の `forEach` メソッド

配列には `forEach` メソッドという `for` 文と同じように反復処理を行うメソッドがあります。`forEach` メソッドでの反復処理は、次のように書けます。

```
const array = [1, 2, 3];  
array.forEach(currentValue => {  
  // 配列の要素ごとに呼び出される処理  
});
```

JavaScript では、関数が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）を引数として渡せます。

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、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];  
// forEach は"コールバック関数"を受け取る高階関数  
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、配列の要素が先頭から順番に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 3 の値が順番に渡されます。

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

```
// と順番に出力される
```

先ほどの for 文の例と同じ数値の合計を返す `sum` 関数を `forEach` メソッドで実装してみます。

```
function sum(numbers) {  
  let total = 0;  
  numbers.forEach(num => {  
    total += num;  
  });  
  return total;  
}  
  
sum([1, 2, 3, 4, 5]); // => 15
```

`forEach` は for 文の条件式に相当するものではなく、必ず配列のすべての要素を反復処理します。変数 `i` といった一時的な値を定義する必要がないため、シンプルに反復処理を書けます。

## 11.5 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に、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 num = numbers[i];  
  if (num % 2 === 0) {  
    isEvenIncluded = true;  
    break;  
  }  
}  
  
console.log(isEvenIncluded); // => true
```

## 第 11 章 ループと反復処理

1 つでも偶数があるかがわかればよいので、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。このような処理は使い回せるように、関数として実装するのが一般的です。

同様の処理をする `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。

```
// 引数の num が偶数なら true を返す
function isEven(num) {
  return num % 2 === 0;
}

// 引数の 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 num = numbers[i];
    if (isEven(num)) {
      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るため、次のようにも書けます。`numbers` に 1 つでも偶数が含まれていれば結果は `true` となるため、偶数の値が見つかった時点で `true` を返しています。

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

function isEvenIncluded(numbers) {
  for (let i = 0; i < numbers.length; i++) {
    const num = numbers[i];
    if (isEven(num)) {
      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 することで一時的な変数が不要となり、より簡潔に書けました。

### 11.5.1 配列の some メソッド

先ほどの isEvenIncluded 関数は、偶数を見つけたら true を返す関数でした。配列では 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受け取ります。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 => {
  // テストをパスすると 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けます。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(num) {
  return num % 2 === 0;
}

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

## 11.6 continue 文

continue 文は現在の反復処理を終了して、次の反復処理を行います。continue 文は、while、do-while、for の中で使えます。

たとえば、while 文の処理中で continue 文が実行されると、現在の反復処理はその時点で終了します。そして、次の反復処理で条件式を評価するところからループが再開します。

```
while (条件式) {
  // 実行される処理
  continue; // 条件式へ
  // これ以降の行は実行されません
}
```

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

```
// number が偶数なら true を返す
```

## 第 11 章 ループと反復処理

```
function isEven(num) {
  return num % 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 num = numbers[i];
    // 偶数ではないなら、次のループへ
    if (!isEven(num)) {
      continue;
    }
    // 偶数を results に追加
    results.push(num);
  }
  return results;
}
const array = [1, 5, 10, 15, 20];
console.log(filterEven(array)); // => [10, 20]
```

もちろん、次のように `continue` 文を使わずに「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ってコードが読みにくくなります。そのため、[ネストした if 文のうろう年の例](#)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たい宣言をして、複雑なコードになることを避けています。

### 11.6.1 配列の `filter` メソッド

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
});
```



```
});
```

先ほどの `continue` 文を使った値の絞り込みは `filter` メソッドを使うとより簡潔に書けます。次のコードでは、`filter` メソッドを使って偶数だけに絞り込んでいます。

```
function isEven(num) {  
    return num % 2 === 0;  
}  
  
const array = [1, 5, 10, 15, 20];  
console.log(array.filter(isEven)); // => [10, 20]
```

## 11.7 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 (プロパティ in オブジェクト) {  
    実行する文;  
}
```

次のコードでは `obj` のプロパティ名を `key` 変数に代入して反復処理をしています。`obj` 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（ループのたびに毎回新しいブロックを作成しているため、ループごとに定義する変数 `key` は再定義エラーになりません。詳細は「[関数とスコープ](#)」の章の「[ブロックスコープ](#)」で解説します）。

```
const obj = {  
    "a": 1,  
    "b": 2,  
    "c": 3  
};  
// 注記: ループのたびに毎回新しいブロックに変数key が定義されるため、  
// 再定義エラーが発生しない  
for (const key in obj) {  
    const value = obj[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

## 第 11 章 ループと反復処理

ます。

JavaScript では、オブジェクトは何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に、親オブジェクトまで列挙可能なものがあるかを探索して列挙します。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い、意図しない結果になる場合があります。

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 文を使わずに書けます。`Object.keys` メソッドは引数のオブジェクト自身が持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 obj = {
  "a": 1,
  "b": 2,
  "c": 3
};
Object.keys(obj).forEach(key => {
  const value = obj[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) {
  // 0 + "0" + "1" という文字列結合が行われる
  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 文の利用は避け、Object.keys メソッドなどを使って配列として反復処理するなど別の方法を考えたほうがよいでしょう。

## 11.8 for...of 文 ES2015

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 に代入して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て反復処理を行えます。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 str = "吉野家";  
for (const value of str) {  
    console.log(value);  
}  
// "吉"  
// "野"  
// "家"
```

そのほかにも、TypedArray、Map、Set、DOM NodeList など、Symbol.iterator が実装されているオブジェクトは多いです。for...of 文は、それらの iterable オブジェクトで反復処理に利用できます。

## 第 11 章 ループと反復処理

## 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言えるからです。`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をすると、変数への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。

配列には、反復処理をして新しい値を作る `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関数を `reduce` メソッドを使って実装してみましょう。

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わかります。`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なくなります。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。

## 11.9 まとめ

この章では、for 文などの構文での反復処理と配列のメソッドを使った反復処理について比較しながら見てきました。for 文などの構文では `continue` 文や `break` 文が利用できますが、配列のメソッドではそれらは利用できません。一方で配列のメソッドは、一時的な変数を管理する必要がないことや、処理をコールバック関数として書くという違いがあります。

どちらの方法も反復処理においてはよく利用されます。どちらが優れているというわけでもないため、どちらの方法も使いこなせるようになることが重要です。また、配列のメソッドについては「[配列](#)」の章でも詳しく解説します。

## 第12章

## オブジェクト

# Chapter 12

オブジェクトはプロパティの集合です。プロパティとは名前（キー）と値（バリュー）が対になったものです。プロパティの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を指定できます。また、1つのオブジェクトは複数のプロパティを持てるため、1つのオブジェクトで多種多様な値を表現できます。

今までも登場してきた、配列や関数などもオブジェクトの一種です。JavaScript には、あらゆるオブジェクトの元となる `Object` という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は実行環境に組み込まれたオブジェクトのことです。`Object` というビルトインオブジェクトは ECMAScript の仕様で定義されているため、あらゆる JavaScript の実行環境で利用できます。

この章では、オブジェクトの作成や扱い方、`Object` というビルトインオブジェクトについて見ていきます。

### 12.1 オブジェクトを作成する

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。

```
// プロパティを持たない空のオブジェクトを作成
const obj = {};
```

オブジェクトリテラルでは、初期値としてプロパティを持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）で区切って記述します。

```
// プロパティを持つオブジェクトを定義する
const obj = {
  // キー: 値
  "key": "value"
};
```

オブジェクトリテラルのプロパティ名（キー）はクォート（`"`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）はクォートを省略することが可能
```

## 12.1 オブジェクトを作成する

```
const obj = {  
  // キー: 値  
  key: "value"  
};
```

ただし、ハイフンを含むプロパティ名（キー）はクォート（"や'）を省略できません。

```
const object = {  
  // キー: 値  
  my-prop: "value" // NG  
};
```

```
const obj = {  
  // キー: 値  
  "my-prop": "value" // OK  
};
```

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）を持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 = {  
  // それぞれのプロパティは、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 というプロパティ名で name の変数を値に設定したオブジェクト  
const obj = {  
  name: name  
};  
console.log(obj); // => { 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けます。次のコードは、プロパティ名 **name** に変数 **name** を値にしたプロパティを設定しています。

```
const name = "名前";  
// name というプロパティ名で name の変数を値に設定したオブジェクト  
const obj = {
```

## 第12章 オブジェクト

```
    name
  };
  console.log(obj); // => { 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、`{}`の中でプロパティ名が単独で書かれている場合は、この省略記法を利用していることに注意してください。

### 12.1.1 `{}`は `Object` のインスタンスオブジェクト

`Object` は JavaScript のビルトインオブジェクトです。オブジェクトリテラル (`{}`) は、このビルトインオブジェクトである `Object` を元にして新しいオブジェクトを作成するための構文です。

オブジェクトリテラル以外の方法として、`new` 演算子を使うことで、`Object` から新しいオブジェクトを作成できます。次のコードでは、`new Object()` でオブジェクトを作成していますが、これは空のオブジェクトリテラルと同じ意味です。

```
// プロパティを持たない空のオブジェクトを作成
// = Object からインスタンスオブジェクトを作成
const obj = new Object();
console.log(obj); // => {}
```

オブジェクトリテラルのほうが明らかに簡潔で、プロパティの初期値も指定できるため、`new Object()` を使う利点はありません。

`new Object()` でオブジェクトを作成することは、「`Object` のインスタンスオブジェクトを作成する」と言います。しかしながら、`Object` やインスタンスオブジェクトなどややこしい言葉の使い分けが必要となってしまいます。そのため、この書籍ではオブジェクトリテラルと `new Object` どちらの方法であっても、単に「オブジェクトを作成する」と呼びます。

オブジェクトリテラルは、`Object` から新しいインスタンスオブジェクトを作成していることを意識しておくといよいでしょう。

## 12.2 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 (`.`) とブラケット記法 (`[]`) があります。それぞれの記法で、オブジェクトの右辺へプロパティ名を指定すると、その名前を持ったプロパティの値を参照できます。

```
const obj = {
  key: "value"
};
// ドット記法で参照
console.log(obj.key); // => "value"
// ブラケット記法で参照
console.log(obj["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）。

```
obj.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
obj.123; // NG
// プロパティ名にハイフンを含む識別子は利用できない
obj.my-prop; // NG
```

一方、ブラケット記法では、[と] の間に任意の式を書けます。そのため、識別子の命名規則とは関係なく、任意の文字列をプロパティ名として指定できます。ただし、プロパティ名は文字列へと暗黙的に変換されることに注意してください。

```
const obj = {
  key: "value",
  123: 456,
  "my-key": "my-value"
};

console.log(obj["key"]); // => "value"
// プロパティ名が数字からはじまる識別子も利用できる
console.log(obj[123]); // => 456
// プロパティ名にハイフンを含む識別子も利用できる
console.log(obj["my-key"]); // => "my-value"
```

また、ブラケット記法ではプロパティ名に変数も利用できます。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。

```
const languages = {
  ja: "日本語",
  en: "英語"
};

const myLang = "ja";
console.log(languages[myLang]); // => "日本語"
```

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。基本的には簡潔なドット記法 (.) を使い、ドット記法で書けない場合はブラケット記法 ([ ]) を使うとよいでしょう。

## 12.3 オブジェクトと分割代入 ES2015

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` と書くのが冗長となりやすいです。そのため、短い名前でも利用できるように、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、変数 `ja` と `en` を定義し、その初期値として `languages` オブジェクトのプロパティを代入しています。

```
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、先ほど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、分割代入では一箇所に書くことができます。

```
const languages = {
  ja: "日本語",
  en: "英語"
};
const { ja, en } = languages;
console.log(ja); // => "日本語"
console.log(en); // => "英語"
```

## 12.4 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自体を変更できるというミュータブル（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 obj = {};
// key プロパティを追加して値を代入
obj.key = "value";
console.log(obj.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プロパティ名として利用できます。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
- 変数の識別子として扱えない文字列（詳細は「[変数と宣言](#)」の章を参照）
- Symbol

```
const key = "key-string";
const obj = {};
// key の評価結果 "key-string" をプロパティ名に利用
obj[key] = "value of key";
// 取り出すときも同じくkey 変数を利用
console.log(obj[key]); // => "value of key"
```

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。

次のコードでは、Computed property names を使って `key` 変数の評価結果である `"key-string"` をプロパティ名にしています。

```
const key = "key-string";
// Computed Property で key の評価結果 "key-string" をプロパティ名に利用
const obj = {
  [key]: "value"
};
console.log(obj[key]); // => "value"
```

JavaScript のオブジェクトは、作成後にプロパティが変更可能という `mutable` の特性を持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できてしまいます。

次のコードは、`changeProperty` 関数が引数として受け取ったオブジェクトにプロパティを追加している悪い例です。

## 第12章 オブジェクト

```
function changeProperty(obj) {  
    obj.key = "value";  
    // いろいろな処理...  
}  
  
const obj = {};  
changeProperty(obj); // obj のプロパティを変更している  
console.log(obj.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は追加しないほうがよいでしょう。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。

### 12.4.1 プロパティの削除

オブジェクトのプロパティを削除するには `delete` 演算子を利用します。削除したいプロパティを `delete` 演算子の右辺に指定して、プロパティを削除できます。

```
const obj = {  
    key1: "value1",  
    key2: "value2"  
};  
// key1 プロパティを削除  
delete obj.key1;  
// key1 プロパティが削除されている  
console.log(obj); // => { "key2": "value2" }
```

## 12.5 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。次のコードは、`obj` 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 obj = {};  
console.log(obj.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、間違いに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、例外が発生しません。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 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わかります。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わかります。

```
const obj = { key: "value" };
obj.key = "Hi!"; // const で定義したオブジェクト (obj) が変更できる
console.log(obj.key); // => "Hi!"
```

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 **obj** 変数への再代入は防げますが、変数に代入された値であるオブジェクトの変更は防げません（「**const は定数ではない**」を参照）。

```
const obj = { key: "value" };
obj = {}; // => 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。**Object.freeze** はオブジェクトを凍結します。凍結されたオブジェクトでプロパティの追加や変更を行うと例外が発生するようになります。

ただし、**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。**strict mode** でない場合は、凍結されたオブジェクトのプロパティを変更しても例外が発生せずに単に無視されます。

```
"use strict";
const object = Object.freeze({ key: "value" });
// freeze したオブジェクトにはプロパティの追加や変更ができない
object.key = "value"; // => TypeError: "key" is read-only
```

```
const widget = {
  window: {
    title: "ウィジェットのタイトル"
  }
};
// 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ません
```

## 第12章 オブジェクト

`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と例外が発生してしまいます。あるオブジェクトがあるプロパティを持っているかを確認する方法として、次の3つがあります。

- `undefined` との比較
- `in` 演算子
- `hasOwnProperty` メソッド

### 12.5.1 プロパティの存在確認: `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、実際にプロパティアクセスすることでも判定できそうです。次のコードでは、`key` プロパティの値が `undefined` ではないという条件式で、プロパティが存在するかを判定しています。

```
const obj = {
  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 (obj.key !== undefined) {
  // key プロパティが存在する? ときの処理
  console.log("key プロパティの値は undefined ではない");
}
```

しかし、この方法はプロパティの値が `undefined` であった場合に、プロパティそのものが存在しない場合と区別できないという問題があります。次のコードでは、`key` プロパティの値が `undefined` であるため、プロパティが存在しているにもかかわらず `if` 文の中は実行されません。

```
const obj = {
  key: undefined
};
// key プロパティの値が undefined である場合
if (obj.key !== undefined) {
  // この行は実行されません
}
```

このような問題があるため、プロパティが存在するかを判定するには `in` 演算子か `hasOwnProperty` メソッドを利用します。

### 12.5.2 プロパティの存在確認: `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
```

次のコードでは `obj` に `key` プロパティが存在するかを判定しています。`in` 演算子は、プロパティの値は関係なく、プロパティが存在した場合に `true` を返します。

```
const obj = { key: undefined };
// key プロパティを持っているなら true
if ("key" in obj) {
  console.log("key プロパティは存在する");
}
```

### 12.5.3 プロパティの存在確認: `hasOwnProperty` メソッド

オブジェクトの `hasOwnProperty` メソッドは、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。この `hasOwnProperty` メソッドの引数には、存在を判定したいプロパティ名を渡します。

```
const obj = {};
obj.hasOwnProperty("プロパティ名"); // true or false
```

次のコードでは `obj` に `key` プロパティが存在するかを判定しています。`hasOwnProperty` メソッドも、プロパティの値は関係なく、オブジェクトが指定したプロパティを持っている場合に `true` を返します。

```
const obj = { key: "value" };
// obj が key プロパティを持っているなら true
if (obj.hasOwnProperty("key")) {
  console.log("object は key プロパティを持っている");
}
```

`in` 演算子と `hasOwnProperty` メソッドは同じ結果を返していますが、厳密には動作が異なるケースもあります。この動作の違いを知るにはまずプロトタイプオブジェクトという特殊なオブジェクトについて理解する必要があります。次の章の「[プロトタイプオブジェクト](#)」で詳しく解説するため、次の章で `in` 演算子と `hasOwnProperty` メソッドの違いを見ていきます。

## 12.6 toString メソッド

オブジェクトの `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化できます。この2つにはどのような違いがあるのでしょうか？（`String` コンストラクタ関数については「[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

## 第12章 オブジェクト

ます。

```
const obj = { key: "value" };
console.log(obj.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(obj)); // => "[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わかります。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わかります。

```
// 独自の toString メソッドを定義
const customObject = {
  toString() {
    return "custom value";
  }
};
console.log(String(customObject)); // => "custom value"
```

## 12.7 オブジェクトの静的メソッド

最後にビルトインオブジェクトである `Object` の静的メソッドについて見ていきましょう。**静的メソッド**（スタティックメソッド）とは、インスタンスの元となるオブジェクトから呼び出せるメソッドのことです。

これまでの `toString` メソッドなどは、`Object` のインスタンスオブジェクトから呼び出すメソッドでした。これに対して、静的メソッドは `Object` そのものから呼び出せるメソッドです。

ここでは、オブジェクトの処理でよく利用されるいくつかの**静的メソッド**を紹介します。

### 12.7.1 オブジェクトの列挙

最初に紹介したように、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、次の3つの静的メソッドがあります。

- `Object.keys` メソッド: オブジェクトのプロパティ名の配列を返す
- `Object.values` メソッド **ES2017**: オブジェクトの値の配列を返す
- `Object.entries` メソッド **ES2017**: オブジェクトのプロパティ名と値の配列の配列を返す

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 obj = {
  "one": 1,
  "two": 2,
```



#### オブジェクトのプロパティ名は文字列化される

オブジェクトのプロパティへアクセスする際に、指定したプロパティ名は暗黙的に文字列に変換されます。ブラケット記法では、オブジェクトをプロパティ名に指定することもできますが、これは意図したようには動作しません。なぜなら、オブジェクトを文字列化すると "[object Object]" という文字列になるためです。

次のコードでは、`keyObject1` と `keyObject2` をブラケット記法でプロパティ名に指定しています。しかし、`keyObject1` と `keyObject2` はどちらも文字列化すると "[object Object]" という同じプロパティ名となります。そのため、プロパティは意図せず上書きされてしまいます。

```
const obj = {};  
const keyObject1 = { a: 1 };  
const keyObject2 = { b: 2 };  
// どちらも同じプロパティ名 ("[object Object]")に代入している  
obj[keyObject1] = "1";  
obj[keyObject2] = "2";  
console.log(obj); // { "[object Object]": "2" }
```

唯一の例外として、`Symbol` だけは文字列化されずにオブジェクトのプロパティ名として扱えます。

```
const obj = {};  
// Symbol は例外的に文字列化されず扱える  
const symbolKey1 = Symbol("シンボル 1");  
const symbolKey2 = Symbol("シンボル 2");  
obj[symbolKey1] = "1";  
obj[symbolKey2] = "2";  
console.log(obj[symbolKey1]); // => "1"  
console.log(obj[symbolKey2]); // => "2"
```

基本的にはオブジェクトのプロパティ名は文字列として扱われることを覚えておくとよいでしょう。また、`Map` というビルトインオブジェクトはオブジェクトをキーとして扱えます（詳細は「[Map/Set](#)」の章で解説します）。

```
    "three": 3  
  };  
  // Object.keys はキーを列挙した配列を返す  
  console.log(Object.keys(obj)); // => ["one", "two", "three"]  
  // Object.values は値を列挙した配列を返す  
  console.log(Object.values(obj)); // => [1, 2, 3]
```

## 第12章 オブジェクト

```
// Object.entries は [キー, 値] の配列を返す
console.log(Object.entries(obj)); // => [["one", 1], ["two", 2], ["three", 3]]
```

これらの静的メソッドと配列の `forEach` メソッドなどを組み合わせれば、プロパティに対して反復処理ができます。次のコードでは、`Object.keys` メソッドで取得したプロパティ名の一覧をコンソールへ出力しています。

```
const obj = {
  "one": 1,
  "two": 2,
  "three": 3
};
const keys = Object.keys(obj);
keys.forEach(key => {
  console.log(key);
});
// 次の値が順番に出力される
// "one"
// "two"
// "three"
```

## 12.7.2 オブジェクトのマージと複製

`Object.assign` メソッド **ES2015** は、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。このメソッドを使うことで、オブジェクトの複製やオブジェクト同士のマージができます。

`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持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。

```
const obj = 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いきます。

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。この空のオブジェクト（**target**）に **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も指定できます。第一引数に既存のオブジェクトを指定した場合は、その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れます。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、基本的に処理は先頭から後ろへと順番に行います。そのため、空のオブジェクトへ `objectA` を代入してから、その結果に `objectB` を代入するという形になります。

```
// version のプロパティ名が被っている
const objectA = { version: "a" };
const objectB = { version: "b" };
const merged = 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 構文でのマージ

ES2018 では、オブジェクトのマージを行うオブジェクトの... (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作成するためです。

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 = {
```

## 第12章 オブジェクト

```
    ...objectA,
    ...objectB
  };
  console.log(merged); // => { a: "a", b: "b" }
```

プロパティ名が被った場合は、後ろにあるオブジェクトが優先されます。そのため同じプロパティ名を持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をしていると言えます。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。

```
// 引数の obj を浅く複製したオブジェクトを返す
const shallowClone = (obj) => {
  return Object.assign({}, obj);
};
const obj = { a: "a" };
const cloneObj = shallowClone(obj);
console.log(cloneObj); // => { a: "a" }
// オブジェクトを複製しているので、異なるオブジェクトとなる
console.log(obj === cloneObj); // => 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。shallow copy とは、`sources` オブジェクトの直下にあるプロパティだけをコピーするということです。そのプロパティの値がオブジェクトである場合に、ネストした先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 = (obj) => {
  return Object.assign({}, obj);
};

const obj = {
  level: 1,
  nest: {
    level: 2
  },
};

const cloneObj = shallowClone(obj);
// nest オブジェクトは複製されていない
console.log(cloneObj.nest === obj.nest); // => true
```

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。deep copy は、再帰的に shallow copy することで実現できます。次のコードでは、deepClone を shallowClone を使うことで実現しています。

```
// 引数の obj を浅く複製したオブジェクトを返す
const shallowClone = (obj) => {
  return Object.assign({}, obj);
};

// 引数の obj を深く複製したオブジェクトを返す
function deepClone(obj) {
  const newObj = shallowClone(obj);
  // 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(newObj)
    .filter(k => typeof newObj[k] === "object")
    .forEach(k => newObj[k] = deepClone(newObj[k]));
  return newObj;
}

const obj = {
  level: 1,
  nest: {
    level: 2
  },
};

const cloneObj = deepClone(obj);
// nest オブジェクトも再帰的に複製されている
console.log(cloneObj.nest === obj.nest); // => false
```

## 第 12 章 オブジェクト

このように、JavaScript のビルトインメソッドは浅い (shallow) 実装のみを提供し、深い (deep) 実装は提供していないことが多いです。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式を取るためです。

JavaScript は言語仕様で定義されている機能が最低限であるため、それを補うようにユーザーが作成した小さな機能を持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。ライブラリの利用については「[ユースケース: Node.js で CLI アプリケーション](#)」の章で紹介します。

### 12.8 まとめ

この章では、オブジェクトについて学びました。

- `Object` というビルトインオブジェクトがある
- `{}` (オブジェクトリテラル) でのオブジェクトの作成や更新方法
- プロパティの存在を確認する `in` 演算子と `hasOwnProperty` メソッド
- オブジェクトのインスタンスメソッドと静的メソッド

JavaScript の `Object` は他のオブジェクトのベースとなるオブジェクトです。次の「[プロトタイプオブジェクト](#)」の章では、`Object` がどのようにベースとして動作しているのかを見ていきます。

## 第13章

### プロトタイプオブジェクト

# Chapter 13

「[オブジェクト](#)」の章では、オブジェクトの処理方法について見てきました。その中で、空のオブジェクトであっても `toString` メソッドなどを呼び出せていました。

```
const obj = {};  
console.log(obj.toString()); // "[object Object]"
```

オブジェクトリテラルで空のオブジェクトを定義しただけなのに、`toString` メソッドを呼び出せています。このメソッドはどこに実装されているのでしょうか？

また、JavaScript には `toString` 以外にも、オブジェクトに自動的に実装されるメソッドがあります。これらのオブジェクトに組み込まれたメソッドを**ビルトインメソッド**と呼びます。

この章では、これらのビルトインメソッドがどこに実装され、なぜ `Object` のインスタンスから呼び出せるのかを確認していきます。詳しい仕組みについては「[クラス](#)」の章で改めて解説するため、この章では大まかな動作の流れを理解することが目的です。

### 13.1 Object はすべての元

`Object` には、他の `Array`、`String`、`Function` など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。

正確には、ほとんどすべてのオブジェクトは `Object.prototype` プロパティに定義された `prototype` オブジェクトを継承しています。`prototype` オブジェクトとは、すべてのオブジェクトの作成時に自動的に追加される特殊なオブジェクトです。`Object` の `prototype` オブジェクトは、すべてのオブジェクトから利用できるメソッドなどを提供するベースオブジェクトとも言えます。

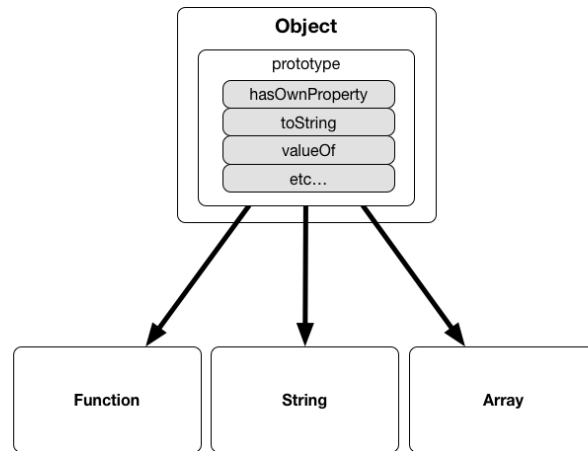
具体的にどういうことかを見えます。

先ほども登場した `toString` メソッドは、`Object` の `prototype` オブジェクトに定義があります。次のように、`Object.prototype.toString` メソッドの実装自体も参照できます。

```
// Object.prototype オブジェクトに toString メソッドの定義がある  
console.log(typeof Object.prototype.toString); // => "function"
```

このような `prototype` オブジェクトに組み込まれているメソッドは**プロトタイプメソッド**と呼ばれます。この書籍では `Object.prototype.toString` のようなプロトタイプメソッドを

## 第13章 プロトタイプオブジェクト

図 13.1 すべてのオブジェクトは `Object` の `prototype` を継承している

`Object#toString` と短縮して表記します。

```
Object.prototype.toString = Object#toString
```

`Object` のインスタンスは、この `Object.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したときに継承します。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。

次のコードでは、オブジェクトリテラルで作成（インスタンス化）したオブジェクトから、`Object#toString` メソッドを参照しています。このときに、インスタンスの `toString` メソッドと `Object#toString` は同じものとなるのがわかります。

```
const obj = {
  "key": "value"
};
// obj インスタンスは Object.prototype に定義されたものを継承する
// obj.toString は継承した Object.prototype.toString を参照している
console.log(obj.toString === Object.prototype.toString); // => true
// インスタンスからプロトタイプメソッドを呼び出せる
console.log(obj.toString()); // => "[object Object]"
```

このように `Object.prototype` に定義されている `toString` メソッドなどは、インスタンス作成時に自動的に継承されるため、`Object` のインスタンスから呼び出せます。これによりオブジェクトリテ



ラルで作成した空のオブジェクトでも、`Object#toString` メソッドなどを呼び出せるようになっていきます。

このインスタンスから `prototype` オブジェクト上に定義されたメソッドを参照できる仕組みを**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については「[クラス](#)」の章で扱うため、ここではインスタンスからプロトタイプメソッドを呼び出せるということがわかっていれば問題ありません。

### 13.1.1 プロトタイプメソッドとインスタンスメソッドの優先順位

プロトタイプメソッドと同じ名前のメソッドがインスタンスオブジェクトに定義されている場合もあります。その場合には、インスタンスに定義したメソッドが優先して呼び出されます。

次のコードでは、`Object` のインスタンスである `customObject` に `toString` メソッドを定義しています。実行してみると、プロトタイプメソッドよりも優先してインスタンスのメソッドが呼び出されていることがわかります。

```
// オブジェクトのインスタンスにtoString メソッドを定義
const customObject = {
  toString() {
    return "custom value";
  }
};
console.log(customObject.toString()); // => "custom value"
```

このように、インスタンスとプロトタイプオブジェクトで同じ名前のメソッドがある場合には、インスタンスのメソッドが優先されます。

### 13.1.2 `in` 演算子と `Object#hasOwnProperty` メソッドの違い

「[オブジェクト](#)」の章で学んだ `Object#hasOwnProperty` メソッドと `in` 演算子の挙動の違いについて見ていきます。2つの挙動の違いはこの章で紹介したプロトタイプオブジェクトに関係しています。

`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ければ、そのオブジェクトの継承元である `prototype` オブジェクトまで探索して持っているかを判定します。つまり、`in` 演算子はインスタンスに実装されたメソッドなのか、プロトタイプオブジェクトに実装されたメソッドなのかを区別しません。

次のコードでは、空のオブジェクトが `toString` メソッドを持っているかを `Object#hasOwnProperty` メソッドと `in` 演算子でそれぞれ判定しています。`hasOwnProperty` メソッドは `false` を返し、`in` 演算子は `toString` メソッドがプロトタイプオブジェクトに存在するため `true` を返します。

```
const obj = {};
// obj というオブジェクト自体に toString メソッドが定義されているわけではない
console.log(obj.hasOwnProperty("toString")); // => false
```

## 第13章 プロトタイプオブジェクト

```
// 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たどるため、  
// Object.prototype まで見に行く  
console.log("toString" in obj); // => true
```

次のように、インスタンスが `toString` メソッドを持っている場合は、`hasOwnProperty` メソッドも `true` を返します。

```
// オブジェクトのインスタンスにtoString メソッドを定義  
const obj = {  
  toString() {  
    return "custom value";  
  }  
};  
// オブジェクトのインスタンスがtoString メソッドを持っている  
console.log(obj.hasOwnProperty("toString")); // => true  
console.log("toString" in obj); // => true
```

### 13.1.3 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。

これまでの説明で、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けます。

```
// const obj = {} と同じ意味  
const obj = Object.create(Object.prototype);  
// obj は Object.prototype を継承している  
console.log(obj.hasOwnProperty === Object.prototype.hasOwnProperty); // => true
```

### 13.1.4 `Array` も `Object` を継承している

`Object` と `Object.prototype` の関係と同じように、ビルトインオブジェクト `Array` も `Array.prototype` を持っています。同じように、配列 (`Array`) 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` も継承しています。

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。この疑似コードは、`Array` コンストラクタの実装など、実際のものとは異なる部分があるため、あくまでイ

メージであることに注意してください。

```
// このコードはイメージです！
// 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。

次のコードでは、Array のインスタンスから Object#hasOwnProperty メソッドが参照できていることがわかります。

```
const array = [];
// Array のインスタンス -> Array.prototype -> Object.prototype
console.log(array.hasOwnProperty === Object.prototype.hasOwnProperty);
// => true
```

このような hasOwnProperty メソッドの参照が可能なのもプロトタイプチェーンという仕組みによるものです。

ここでは、Object.prototype 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こと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 が持つメソッドを利用できる点を覚えておきましょう。

また、Array.prototype などそれぞれ独自のメソッドを定義しています。たとえば、Array#toString メソッドもそのひとつです。そのため、配列のインスタンスで toString メソッドを呼び出すと Array#toString が優先して呼び出されます。

```
const numbers = [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(numbers.toString()); // => "1,2,3"
```

#### 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になるオブジェクトであると言いましたが、例外もあります。

イディオム(慣習的な書き方)ですが、Object.create(null) とすることで Object.prototype

## 第13章 プロトタイプオブジェクト

を継承しないオブジェクトを作成できます。これにより、プロパティやメソッドをまったく持たない本当に空のオブジェクトを作れます。

```
// 親がnull、つまり親がないオブジェクトを作る
const obj = Object.create(null);
// Object.prototype を継承しないため、hasOwnProperty が存在しない
console.log(obj.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した。`Map` とはキーと値の組み合わせを保持するためのオブジェクトです。

ただのオブジェクトも `Map` とよく似た性質を持っていますが、最初からいくつかのプロパティが存在し、アクセスできてしまいます。なぜなら、`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するためです。そのため、`Object.create(null)` で `Object.prototype` を継承しないオブジェクトを作成し、そのオブジェクトが `Map` の代わりとして使われていました。

```
// 空オブジェクトを作成
const obj = {};
// "toString"という値を定義してないのに、"toString"が存在している
console.log(obj["toString"]); // Function
// Map のような空オブジェクト
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。`Map` については「[Map/Set](#)」の章で詳しく紹介します。

```
const map = new Map();
// toString キーは存在しない
console.log(map.has("toString")); // => false
```

## 13.2 まとめ

この章では、プロトタイプオブジェクトについて学びました。

- プロトタイプオブジェクトはオブジェクトの作成時に自動的に作成される
- `Object` のプロトタイプオブジェクトには `toString` などのプロトタイプメソッドが定義されて

いる

- ほとんどのオブジェクトは `Object.prototype` を継承することで `toString` メソッドなどを呼び出せる
- プロトタイプメソッドとインスタンスメソッドではインスタンスメソッドが優先される
- `Object.create` メソッドを使うことでプロトタイプオブジェクトを継承しないオブジェクトを作成できる

プロトタイプオブジェクトに定義されているメソッドがどのように参照されているかを確認しました。このプロトタイプの詳しい仕組みについては「[クラス](#)」の章で改めて解説します。

## 第14章

### 配列

# Chapter 14

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。インデックスは先頭の要素から 0、1、2 のように 0 からはじまる連番となります。

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。

この章では、配列の基本的な操作と配列を扱う場合におけるパターンについて学びます。

#### 14.1 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](#)」の章の「[配列リテラル](#)」です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います。配列リテラル ([と]) 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];  
// 2次元配列 (配列の配列)  
const matrix = [  
  ["a", "b"],  
  ["c", "d"]  
];
```

作成した配列の要素への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、そのインデックスの要素を配列から読み取れます。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"
```

2次元配列（配列の配列）からの値の読み取りも同様に**配列 [インデックス]** でアクセスできます。  
配列 [0] [0] は、配列の 0 番目の要素である配列 (["a", "b"]) の 0 番目の要素を読み取ります。

```
// 2次元配列（配列の配列）
const matrix = [
  ["a", "b"],
  ["c", "d"]
];
console.log(matrix[0][0]); // => "a"
```

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利用できます。

```
const array = ["one", "two", "three"];
console.log(array.length); // => 3
// 配列の要素数 - 1 が 最後の要素のインデックスとなる
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のでしょうか？ 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アクセスしているのと原理は同じです。オブジェクトでも、存在しないプロパティへアクセスした場合には `undefined` が返ってきます。

```
const obj = {
  "0": "one",
  "1": "two",
  "2": "three",
  "length": 3
};
// obj["100"] は定義されていないため undefined が返る
console.log(obj[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。

## 第 14 章 配列

```
// 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
```

## 14.2 オブジェクトが配列かどうかを判定する

あるオブジェクトが配列かどうかを判定するには `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 obj = {};
const array = [];
console.log(Array.isArray(obj)); // => false
console.log(Array.isArray(array)); // => true
```

また、`typeof` 演算子では配列かどうかを判定することはできません。配列もオブジェクトの一種であるため、`typeof` 演算子の結果が `"object"` となるためです。

```
const array = [];
console.log(typeof array); // => "object"
```

### TypedArray ES2015

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` はバイナリデータのバッファを示すために使われるデータ型で、WebGL やバイナリを扱う場面で利用されます。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、通常の配列とは用途や使い勝手が異なります。また、`TypedArray` は `Array.isArray` のメソッドの結果が `false` となることから別物と考えてよいでしょう。

```
// TypedArray を作成
const typedArray = new Int8Array(8);
console.log(Array.isArray(typedArray)); // => false
```

そのため、JavaScript で配列といった場合には `Array` を示します。



## 14.3 配列と分割代入 ES2015

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"
```

## 14.4 配列から要素を検索

配列から指定した要素を検索する目的には、主に次の3つがあります。

- 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。

### 14.4.1 インデックスを取得

指定した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 ES2015 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検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** メソッドでは末尾から検索した結果が得られます。

```
const array = ["Java", "JavaScript", "Ruby"];
const indexOfJS = array.indexOf("JavaScript");
console.log(indexOfJS); // => 1
```

## 第 14 章 配列

## 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、区別できていないことがわかります。

```
// 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
```

```
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を発見できますが、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、同じプロパティを持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、異なる参照を持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 obj = { key: "value" };
const array = ["A", "B", obj];
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だと判定される
```

```
console.log(obj === { key: "value" }); // => false
// 等価のオブジェクトを検索してインデックスを返す
console.log(array.indexOf(obj)); // => 2
```

このように、異なるオブジェクトだが値は同じものを見つけない場合には、**Array#findIndex** メソッドが利用できます。**findIndex** メソッドの引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けます。これにより、プロパティの値が同じ要素を配列から見つけて、その要素のインデックスが得られます。

```
// 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((obj) => {
  return obj.color === "blue";
});
console.log(indexOfBlue); // => 2
console.log(colors[indexOfBlue]); // => { "color": "blue" }
```

#### 14.4.2 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し、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** メソッドを使って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** メソッド **ES2015** が使えます。**find** メソッドには、**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
```

## 第14章 配列

```
const blueColor = colors.find((obj) => {
  return obj.color === "blue";
});
console.log(blueColor); // => { "color": "blue" }
// 該当する要素がない場合はundefinedを返す
const whiteColor = colors.find((obj) => {
  return obj.color === "white";
});
console.log(whiteColor); // => undefined
```

### 14.4.3 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)); // => []
```

### 14.4.4 真偽値を取得

最後に、指定した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わかります。

しかし、指定した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には、取得したインデックスや要素を何に使うの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に-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");
  // ... いろいろな処理 ...
  // indexOfJS は、含まれているのかの判定以外には利用していない
}
```

そこで、ES2016 で導入された `Array#includes` メソッド **ES2016** を利用します。`Array#includes` メソッドは配列に指定要素が含まれているかを判定します。`includes` 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。前述のコードでは次のように `includes` メソッドを使うべきでしょう。

```
const array = ["Java", "JavaScript", "Ruby"];
// includes 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");
}
```

`includes` メソッドは、`indexOf` メソッドと同様、異なるオブジェクトだが値が同じものを見つけた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」の章の「配列の `some` メソッド」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((obj) => {
  return obj.color === "blue";
});
console.log(isIncludedBlueColor); // => true
```

## 14.5 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

## 第14章 配列

`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"]
```

## 14.6 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"]
```

## 14.7 配列の展開 **ES2015**

… (Spread 構文) を使うことで、配列リテラル中に既存の配列を展開できます。

次のコードでは、配列リテラルの末尾に配列を展開しています。これは、`Array#concat` メソッドで配列同士を結合するのと同じ結果になります。

```
const array = ["A", "B", "C"];
// Spread 構文を使った場合
```

```
const newArray = ["X", "Y", "Z", ...array];
// concat メソッドの場合
const newArrayConcat = ["X", "Y", "Z"].concat(array);
console.log(newArray); // => ["X", "Y", "Z", "A", "B", "C"]
console.log(newArrayConcat); // => ["X", "Y", "Z", "A", "B", "C"]
```

Spread 構文は、concat メソッドとは異なり、配列リテラル中の任意の位置に配列を展開できます。そのため、次のように要素の途中に配列を展開することも可能です。

```
const array = ["A", "B", "C"];
const newArray = ["X", ...array, "Z"];
console.log(newArray); // => ["X", "A", "B", "C", "Z"]
```

## 14.8 配列をフラット化 **ES2019**

**Array#flat** メソッド **ES2019** を使うことで、多次元配列をフラットな配列に変換できます。引数を指定しなかった場合は1段階のみのフラット化ですが、引数に渡す数値でフラット化する深さを指定できます。配列をすべてフラット化するには、無限を意味する **Infinity** を値として渡すことで実現できます。

```
const array = [["A"], "B", "C"];
// 引数なしは 1 を指定した場合と同じ
console.log(array.flat()); // => ["A", "B", "C"]
console.log(array.flat(1)); // => ["A", "B", "C"]
console.log(array.flat(2)); // => ["A", "B", "C"]
// すべてをフラット化するには Infinity を渡す
console.log(array.flat(Infinity)); // => ["A", "B", "C"]
```

また、**Array#flat** メソッドは必ず新しい配列を作成して返すメソッドです。そのため、これ以上フラット化できない配列をフラット化しても、同じ要素を持つ新しい配列を返します。

```
const array = ["A", "B", "C"];
console.log(array.flat()); // => ["A", "B", "C"]
```

## 14.9 配列から要素を削除

### 14.9.1 **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の要素を削除できません。配列の任意のインデックスの要素を削除するには **Array#splice** を利用できます。

## 第14章 配列

`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` メソッドは指定したインデックスから、指定した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 = ["a", "b", "c"];  
// 1番目から1つの要素("b")を削除  
array.splice(1, 1);  
console.log(array); // => ["a", "c"]  
console.log(array.length); // => 2  
console.log(array[1]); // => "c"  
// すべて削除  
array.splice(0, array.length);  
console.log(array.length); // => 0
```

### 14.9.2 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` で行えますが、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。

### 14.9.3 空の配列を代入

最後に、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` に空の配列を参照させています。

```
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、`const` で宣言した配列の場合は変数に対して再代入できないため、この手法は使えません。そのため、再代入をしたい場合は `let` または `var` で変数宣言をする必要があります。

```
const array = [1, 2, 3];  
console.log(array.length); // => 3  
// const で宣言された変数には再代入できない  
array = []; // TypeError: invalid assignment to const `array` が発生
```

## 14.10 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なメソッドがあります。この破壊的なメソッドと非破壊的な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。非破壊的なメソッドとは、配列オブジェクトのコピーを作成してから変更し、そのコピーした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` 変数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。`myArray` 変数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
```

第 14 章 配列

```
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なメソッドを名前から見分ける方法はありません。また、配列を返す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なメソッドです。

次の表で紹介するメソッド<sup>\*1</sup>は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> <b>ES2015</b> | 変更後の配列         |
| <code>Array.prototype.fill</code> <b>ES2015</b>       | 変更後の配列         |

破壊的な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、破壊的なもの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"]
```

<sup>\*1</sup> [https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global\\_Objects/Array/](https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/Array/)

```
// array 自体にも影響を与える
console.log(array); // => ["A", "C"]
```

一方、非破壊的な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` 関数を非破壊的なものにするには、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` メソッドそのものは存在しませんが、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。非破壊的な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な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なライブラリが存在します。非破壊的な配列を扱うライブラリの例として

## 第 14 章 配列

[immutable-array-prototype](#)<sup>\*2</sup>や [Immutable.js](#)<sup>\*3</sup>などがあります。

## 14.11 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ましたが、改めて関連する 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のが `Array#forEach`、`Array#map`、`Array#filter`、`Array#reduce` です。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。

### 14.11.1 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]
```

### 14.11.2 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;
});
```

---

<sup>\*2</sup> <https://github.com/azu/immutable-array-prototype>

<sup>\*3</sup> <https://facebook.github.io/immutable-js/>

```
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
```

### 14.11.3 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];
// 奇数の値を持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
```

### 14.11.4 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い場合に利用します。

ここまでで紹介した反復処理のメソッドとは異なり、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**累積値**の初期値となる値を渡します。

次のコードでは、**reduce** メソッドは初期値を 0 として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** 型の値を返しています。

```
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);
// 0 + 1 + 2 + 3 という式の結果が返り値になる
console.log(totalValue); // => 6
```

`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。

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 |
| Array 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持つオブジェクトです。Array.from メソッド **ES2015** を使うことで Array-like オブジェクトを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");
```

## 14.12 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、メソッドを呼び出した返り値に対してメソッド呼び出しをする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行われているのかわかりやすいです。`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わかります。

```
// 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ほとんどであるためです。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込む」

## 第14章 配列

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つなげられます。

```
// ECMAScript のバージョン名と発行年
const ECMAScriptVersions = [
  { name: "ECMAScript 1", year: 1997 },
  { name: "ECMAScript 2", year: 1998 },
  { name: "ECMAScript 3", year: 1999 },
  { name: "ECMAScript 5", year: 2009 },
  { name: "ECMAScript 5.1", year: 2011 },
  { name: "ECMAScript 2015", year: 2015 },
  { name: "ECMAScript 2016", year: 2016 },
  { name: "ECMAScript 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 = ECMAScriptVersions
  // 2000 年以下のデータに絞り込み
  .filter(ECMAScript => ECMAScript.year <= 2000)
  // それぞれの要素から name プロパティを取り出す
  .map(ECMAScript => ECMAScript.name);
console.log(versionNames); // => ["ECMAScript 1", "ECMAScript 2", "ECMAScript 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過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。

## 14.13 まとめ

この章では配列について学びました。

- 配列は順序を持った要素を格納できるオブジェクトの一種
- 配列には破壊的なメソッドと非破壊的なメソッドがある
- 配列には反復処理を行う高階関数となるメソッドがある
- メソッドチェーンは配列のメソッドが配列を返すことを利用している

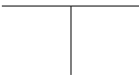
配列は JavaScript の中でもよく使われるオブジェクトで、メソッドの種類も多いです。この書籍でもすべてのメソッドは紹介していないため、詳しくは [Array についてのドキュメント](https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/Array)<sup>\*4</sup>も参照してみてください

<sup>\*4</sup> [https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global\\_Objects/Array](https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/Array)





ください。



## 第15章

## 文字列

# Chapter 15

この章では JavaScript における文字列について学んでいきます。まずは、文字列の作成方法や文字列の操作方法について見ていきます。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。

### 15.1 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。「データ型とリテラル」の章でも紹介しましたが、文字列リテラルには"（ダブルクォート）、'（シングルクォート）、`（バッククォート）の3種類があります。

まずは"（ダブルクォート）と'（シングルクォート）について見ていきます。

"（ダブルクォート）と'（シングルクォート）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、"（ダブルクォート）を主な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
```

ES2015 では、テンプレートリテラル`（バッククォート）が追加されました。`（バッククォート）を利用することで文字列を作成できる点は、他の文字列リテラルと同じです。

これに加えてテンプレートリテラルでは、文字列中に改行を入力できます。次のコードでは、テンプレートリテラルを使って複数行の文字列を見た目どおりに定義しています。

```
const multiline = `1 行目
2 行目
3 行目`;
// \n は改行を意味する
```

```
console.log(multiline); // => "1 行目\n2 行目\n3 行目"
```

どの文字列リテラルでも共通ですが、文字列リテラルは同じ記号が対になります。そのため、文字列の中にリテラルと同じ記号が出現した場合は、\（バックスラッシュ）を使ってエスケープする必要があります。次のコードでは、文字列中の"を\"のようにエスケープしています。

```
const str = "This book is \"js-primer\"";
console.log(str); // => 'This book is "js-primer"'
```

### 15.2 エスケープシーケンス

文字列リテラル中にはそのままでは入力できない特殊な文字もあります。改行もそのひとつで、"（ダブルクォート）と'（シングルクォート）の文字列リテラルには改行をそのまま入力できません（テンプレートリテラル中には例外的に改行をそのまま入力できます）。

次のコードは、JavaScript の構文として正しくないため、構文エラー（SyntaxError）となります。

```
// JavaScript エンジンが構文として解釈できないため、SyntaxError となる
const invalidString = "1 行目
2 行目
3 行目";
```

この問題を回避するためには、改行のような特殊な文字をエスケープシーケンスとして書く必要があります。エスケープシーケンスは、\と特定の文字を組み合わせることで、特殊文字を表現します。

次の表では、代表的なエスケープシーケンス<sup>\*1</sup>を紹介しています。エスケープシーケンスは、"（ダブルクォート）、'（シングルクォート）、`（バッククォート）すべての文字列リテラルの中で利用できます。

| エスケープシーケンス           | 意味                               |
|----------------------|----------------------------------|
| \'                   | シングルクォート                         |
| \"                   | ダブルクォート                          |
| \`                   | バッククォート                          |
| \\                   | バックスラッシュ（\そのものを表示する）             |
| \n                   | 改行                               |
| \t                   | タブ                               |
| \uXXXX               | Code Unit（\u と 4 桁の HexDigit）    |
| \u{X} ... \u{XXXXXX} | Code Point（\u{ }のカッコ中に HexDigit） |

このエスケープシーケンスを利用することで、先ほどの"（ダブルクォート）の中に改行（\n）を入力できます。

```
// 改行を\nのエスケープシーケンスとして入力している
```

<sup>\*1</sup> [https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global\\_Objects/String#%E3%82%A8%E3%82%B9%E3%82%B1%E3%83%BC%E3%83%97%E3%82%B7%E3%83%BC%E3%82%B1%E3%83%B3%E3%82%B9](https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/String#%E3%82%A8%E3%82%B9%E3%82%B1%E3%83%BC%E3%83%97%E3%82%B7%E3%83%BC%E3%82%B1%E3%83%B3%E3%82%B9)

## 第15章 文字列

```
const multiline = "1 行目\n2 行目\n3 行目";
console.log(multiline);
/* 改行した結果が出力される
1 行目
2 行目
3 行目
*/
```

また、\からはじまる文字は自動的にエスケープシーケンスとして扱われます。しかし、\aのように定義されていないエスケープシーケンスは、\が単に無視されaという文字列として扱われます。これにより、\（バックスラッシュ）そのものを入力していたつもりが、その文字がエスケープシーケンスとして扱われてしまう問題があります。

次のコードでは、\\_という組み合わせのエスケープシーケンスはないため、\が無視された文字列として評価されます。

```
console.log("\_(ツ)_/ ");
// \_(ツ)_/ のように\が無視されて表示される
```

\（バックスラッシュ）そのものを入力したい場合は、\\のようにエスケープする必要があります。

```
console.log("\\_(ツ)_/ ");
// \_(ツ)_/ と表示される
```

## 15.3 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 str = "a" + "b";
console.log(str); // => "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えます。

```
const name = "JavaScript";
console.log("Hello " + name + "!");// => "Hello JavaScript!"
```

特定の書式に文字列を埋め込むには、テンプレートリテラルを使うとより宣言的に書けます。

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けます。

```
const name = "JavaScript";
console.log(`Hello ${name}!`);// => "Hello JavaScript!"
```

### 15.4 文字へのアクセス

文字列の特定の位置にある文字にはインデックスを指定してアクセスできます。これは、配列における要素へのアクセスにインデックスを指定するのと同じです。

文字列では文字列 [インデックス] のように指定した位置（インデックス）の文字へアクセスできます。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 str = "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(str[0]); // => "文"
console.log(str[1]); // => "字"
console.log(str[2]); // => "列"
```

また、存在しないインデックスへのアクセスでは配列やオブジェクトと同じように `undefined` を返します。

```
const str = "文字列";
// 42 番目の要素は存在しない
console.log(str[42]); // => undefined
```

### 15.5 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？ コンピュータのメモリ上には文字列の「ア」といった文字をそのまま保存できないため、0 と 1 からなるビット列へ変換する必要があります。この文字から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。

一方で、変換後のビット列が何の文字なのかを管理する表が必要になります。この文字に対応するビット列（ID）の一覧表のことを符号化文字集合と呼びます。

次の表は、Unicode という文字コードにおける符号化文字集合からカタカナの一部分を取り出したものです\*2。Unicode はすべての文字に対して ID（Code Point）を振ることを目的に作成されている仕様です。

|      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 30A0 | = | ア | ア | イ | イ | ウ | ウ | エ | エ | オ | オ | カ | ガ | キ | ギ | ク |
| 30B0 | グ | ケ | ゲ | コ | ゴ | サ | ザ | シ | ジ | ス | ズ | セ | ゼ | ソ | ゾ | タ |
| 30C0 | ダ | チ | ヂ | ツ | ツ | ヅ | テ | デ | ト | ド | ナ | ニ | ヌ | ネ | ノ | ハ |

JavaScript（ECMAScript）は文字コードとして Unicode を採用し、文字をエンコードする方式と

\*2 Unicode のカタカナの一覧 <https://unicode-table.com/jp/#katakana> から取り出したテーブルです。

第 15 章 文字列

して UTF-16 を採用しています。UTF-16 とは、それぞれの文字を 16bit のビット列に変換するエンコード方式です。Unicode では 1 文字を表すのに使う最小限のビットの組み合わせを **Code Unit** (符号単位) と呼び、UTF-16 では各 Code Unit のサイズが 16bit (2 バイト) です。

次のコードは、文字列を構成する Code Unit を hex 値 (16 進数) にして表示する例です。`String#charCodeAt` メソッドは、文字列の指定インデックスの Code Unit を整数として返します。その Code Unit の整数値を `String#toString` メソッドで hex 値 (16 進数) にしています。

```
const str = "アオイ";
// それぞれの文字をCode Unit の hex 値 (16 進数) に変換する
// toString の引数に 16 を渡すと 16 進数に変換される
console.log(str.charCodeAt(0).toString(16)); // => "30a2"
console.log(str.charCodeAt(1).toString(16)); // => "30aa"
console.log(str.charCodeAt(2).toString(16)); // => "30a4"
```

逆に、Code Unit を hex 値 (16 進数) から文字へと変換するには `String.fromCharCode` メソッドを使います。次のコードでは、16 進数の整数リテラルである `0x` で記述した Code Unit から文字列へと変換しています (`0x` リテラルについては「[データ型とリテラル](#)」の章を参照)。

```
const str = String.fromCharCode(
  0x30a2, // アの Code Unit
  0x30aa, // オの Code Unit
  0x30a4 // イの Code Unit
);
console.log(str); // => "アオイ"
```

これらの結果をまとめると、この文字列と文字列を構成する UTF-16 の Code Unit との関係は次のようになります。

| インデックス                     | 0      | 1      | 2      |
|----------------------------|--------|--------|--------|
| 文字列                        | ア      | オ      | イ      |
| UTF-16 の Code Unit (16 進数) | 0x30A2 | 0x30AA | 0x30A4 |

このように、JavaScript における文字列は 16bit の Code Unit が順番に並んだものとして内部的に管理されています。これは、ECMAScript の内部表現として UTF-16 を採用しているだけで、JavaScript ファイル (ソースコードを書いたファイル) のエンコーディングとは関係ありません。そのため、JavaScript ファイル自体のエンコードは、UTF-16 以外の文字コードであっても問題ありません。

UTF-16 を利用していることは JavaScript の内部的な表現であるため、気にする必要がないようにも思えます。しかし、この JavaScript が UTF-16 を利用していることは、これから見ていく String の API にも影響しています。この UTF-16 と文字列については、次の章である「[文字列と Unicode](#)」で詳しく見ていきます。

ここでは、「JavaScript の文字列の各要素は UTF-16 の Code Unit で構成されている」ということ

だけを覚えておけば問題ありません。

## 15.6 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て文字列にするには `Array#join` メソッドを利用できます。

この2つはよく組み合わせて利用されるため、合わせて見ていきます。

`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` メソッドが利用できます。`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。`・` で文字列を分割 (`split`) してから、区切り文字を、にして結合 (`join`) すれば変換できます。

```
const str = "赤・青・緑".split("・").join("、");
console.log(str); // => "赤、青、緑"
```

`String#split` メソッドの第一引数には正規表現も指定できます。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を簡単に書けます。`/\s+/` は 1 つ以上のスペースにマッチする正規表現オブジェクトを作成する正規表現リテラルです。

```
// 文字間に 1 つ以上のスペースがある
const str = "a    b    c    d";
// 1 つ以上のスペースにマッチして分解する
const strings = str.split(/\s+/);
console.log(strings); // => ["a", "b", "c", "d"]
```

## 15.7 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。文字列の構成要素は Code Unit であるため、`length` プロパティは Code Unit の個数を返します。

次の文字列は 3 つの要素 (Code Unit) が並んだものであるため、`length` プロパティは 3 を返します。

```
console.log("文字列".length); // => 3
```

また、空文字列は要素数が 0 であるため、`length` プロパティの結果も 0 となります。

## 第 15 章 文字列

```
console.log("").length); // => 0
```

## 15.8 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。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

難しく書いていますが、同じ文字列同士なら`===`（厳密比較演算子）の結果は `true` となります。

```
console.log("文字列" === "文字列"); // => true
// 一致しなければ false となる
console.log("JS" === "ES"); // => false
// 文字列の長さが異なるので false となる
console.log("文字列" === "文字"); // => false
```

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の比較もできます。

これらの関係演算子も、文字列の要素である Code Unit 同士を先頭から順番に比較します。文字列から Code Unit の数値を取得するには `String#charCodeAt` メソッドを利用できます。

次のコードでは、ABC と ABD を比較した場合にどちらが大きい（Code Unit の値が大きい）かを比較しています。

```
// "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、関係演算子での文字列比較は Code Unit 同士を比較しています。この結果を予測するのは難しく、また直感的ではない結果が生まれることも多いです。文字の順番は国や言語によっても異なるため、国際化（Internationalization）に関する知識も必要です。

JavaScript においても、[ECMA-402](https://www.ecma-international.org/publications/standards/Ecma-402.htm)<sup>\*3</sup>という ECMAScript と関連する別の仕様として国際化についての取り決めがされています。この国際化に関する API を定義した `Intl`<sup>\*4</sup>というビルトインオブジェクトもありますが、この API についての詳細は省略します。

<sup>\*3</sup> <https://www.ecma-international.org/publications/standards/Ecma-402.htm>

<sup>\*4</sup> [https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global\\_Objects/Intl](https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/Intl)



## 15.9 文字列の一部を取得

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いきます。

`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た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 str = "ABCDE";
console.log(str.slice(1)); // => "BCDE"
console.log(str.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(str.slice(-1)); // => "E"
// インデックスが 1 から 4 の範囲を取り出す
console.log(str.slice(1, 4)); // => "BCD"
// 第一引数 > 第二引数の場合、常に空文字列を返す
console.log(str.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て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が入れ替わるという予想しにくい挙動となります。

```
const str = "ABCDE";
console.log(str.substring(1)); // => "BCDE"
console.log(str.substring(1, 5)); // => "BCDE"
// マイナスを指定すると 0 として扱われる
console.log(str.substring(-1)); // => "ABCDE"
// 位置:1 から 4 の範囲を取り出す
console.log(str.substring(1, 4)); // => "BCD"
// 第一引数 > 第二引数の場合、引数が入れ替わる
// str.substring(1, 4) と同じ結果になる
console.log(str.substring(4, 1)); // => "BCD"
```

## 第15章 文字列

このように、マイナスの位置や引数が交換される挙動はわかりやすいものとは言えません。そのため、`slice` メソッドと `substring` メソッドに指定する引数は、どちらも同じ結果となる範囲に限定したほうが直感的な挙動となります。つまり、指定するインデックスは0以上にして、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。

`String#slice` メソッドは `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。次のコードでは、?の位置を `indexOf` メソッドで取得し、それ以降の文字列を `slice` メソッドで切り出しています。

```
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。機能的な違いがほとんどないため、どちらを利用するかは好みの問題となるでしょう。

## 15.10 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。

指定した文字列が文字列中に含まれているかを検索する方法として、`String` メソッドには取得したい結果ごとにメソッドが用意されています。ここでは、次の3種類の結果を取得する方法について文字列と正規表現それぞれの検索方法を見ていきます。

- マッチした箇所のインデックスを取得
- マッチした文字列の取得
- マッチしたかどうかの真偽値を取得

### 15.10.1 文字列による検索

`String` オブジェクトには、指定した文字列で検索するメソッドが用意されています。

#### 文字列によるインデックスの取得

`String#indexOf` メソッドと `String#lastIndexOf` メソッドは、指定した文字列で検索し、その文字列が最初に現れたインデックスを返します。これらは配列の `Array#indexOf` メソッドと同じで、厳密等価演算子 (`===`) で文字列を検索します。一致する文字列がない場合は-1を返します。

- 文字列.`indexOf`("検索文字列"): 先頭から検索し、指定された文字列が最初に現れたインデックスを返す
- 文字列.`lastIndexOf`("検索文字列"): 末尾から検索し、指定された文字列が最初に現れた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、指定した検索文字列を最初に見つけた時

点で検索は終了します。

```
// 検索対象となる文字列
const str = "にわにはにわにわとりがいる";
// indexOf は先頭から検索して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(str.indexOf("にわ")); // => 0
// lastIndexOf は末尾から検索してインデックスを返す- "にわにはにわ**にわ**とりがいる"
console.log(str.lastIndexOf("にわ")); // => 6
// 指定した文字列が見つからない場合は -1 を返す
console.log(str.indexOf("未知のキーワード")); // => -1
```

### 15.10.2 文字列にマッチした文字列の取得

文字列を検索してマッチした文字列は、検索文字列そのものになるので自明です。

次のコードでは"Script"という文字列で検索していますが、その検索文字列にマッチする文字列は  
もちろん"Script"になります。

```
const str = "JavaScript";
const searchWord = "Script";
const index = str.indexOf(searchWord);
if (index !== -1) {
  console.log(`${searchWord}が見つかりました`);
} else {
  console.log(`${searchWord}は見つかりませんでした`);
}
```

#### 真偽値の取得

「文字列」に「検索文字列」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。次の 3  
つのメソッドは ES2015 で導入されました。

- String#startsWith(検索文字列) **ES2015**: 検索文字列が先頭にあるかの真偽値を返す
- String#endsWith(検索文字列) **ES2015**: 検索文字列が末尾にあるかの真偽値を返す
- String#includes(検索文字列) **ES2015**: 検索文字列を含むかの真偽値を返す

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 str = "にわにはにわにわとりがいる";
// startsWith - 検索文字列が先頭なら true
```

## 第 15 章 文字列

```
console.log(str.startsWith("にわ")); // => true
console.log(str.startsWith("いる")); // => false
// endsWith - 検索文字列が末尾なら true
console.log(str.endsWith("にわ")); // => false
console.log(str.endsWith("いる")); // => true
// includes - 検索文字列が含まれるなら true
console.log(str.includes("にわ")); // => true
console.log(str.includes("いる")); // => true
```

## 15.11 正規表現オブジェクト

文字列による検索では、固定の文字列にマッチするものしか検索できません。一方で正規表現による検索では、あるパターン（規則性）にマッチするという柔軟な検索ができます。

正規表現は正規表現オブジェクト（**RegExp** オブジェクト）として表現されます。正規表現オブジェクトはマッチする範囲を決める**パターン**と正規表現の検索モードを指定する**フラグ**の 2 つで構成されます。正規表現のパターン内では、次の文字は**特殊文字**と呼ばれ、特別な意味を持ちます。特殊文字として解釈されないように入力する場合には \（バックスラッシュ）でエスケープする必要があります。

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

正規表現オブジェクトを作成するには、正規表現リテラルと **RegExp** コンストラクタを使う 2 つの方法があります。

```
// 正規表現リテラルで正規表現オブジェクトを作成
const patternA = /パターン/フラグ;
// RegExp コンストラクタで正規表現オブジェクトを作成
const patternB = new RegExp("パターン文字列", "フラグ");
```

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作成できます。次のコードでは、+ という 1 回以上の繰り返しを意味する特殊文字を使い、a が 1 回以上連続する文字列にマッチする正規表現オブジェクトを作成しています。

```
const pattern = /a+/;
```

正規表現オブジェクトを作成するもうひとつの方法として **RegExp** コンストラクタがあります。**RegExp** コンストラクタでは、文字列から正規表現オブジェクトを作成できます。

次のコードでは、**RegExp** コンストラクタを使って a が 1 文字以上連続している文字列にマッチする正規表現オブジェクトを作成しています。これは先ほどの正規表現リテラルで作成した正規表現オブジェクトと同じ意味になります。

```
const pattern = new RegExp("a+");
```

### 15.11.1 正規表現リテラルと RegExp コンストラクタの違い

正規表現リテラルと `RegExp` コンストラクタの違いとして、正規表現のパターンが評価されるタイミングの違いがあります。正規表現リテラルは、ソースコードをロード（パース）した段階で正規表現のパターンが評価されます。一方で、`RegExp` コンストラクタでは通常の間数と同じように実際に `RegExp` コンストラクタを呼び出すまでパターンは評価されません。

単独の `[` という不正なパターンである正規表現を例に、評価されているタイミングの違いを見てみます。`[` は対になる `]` と組み合わせて利用する特殊文字であるため、正規表現のパターンに単独で書くと構文エラーの例外が発生します。

正規表現リテラルは、ソースコードのロード時に正規表現のパターンが評価されるため、次のように `main` 関数を呼び出していなくても構文エラー（`SyntaxError`）が発生します。

```
// 正規表現リテラルはロード時にパターンが評価され、例外が発生する
function main() {
  // [は対となる] を組み合わせる特殊文字であるため、単独で書けない
  const invalidPattern = /[;
}
```

```
// main 関数を呼び出さなくても例外が発生する
```

一方で、`RegExp` コンストラクタは実行時に正規表現のパターンが評価されるため、`main` 関数を呼び出すことで初めて構文エラー（`SyntaxError`）が発生します。

```
// RegExp コンストラクタは実行時にパターンが評価され、例外が発生する
function main() {
  // [は対となる] を組み合わせる特殊文字であるため、単独で書けない
  const invalidPattern = new RegExp("[");
}
```

```
// main 関数を呼び出すことで初めて例外が発生する
main();
```

これを言い換えると、正規表現リテラルはコードを書いた時点で決まったパターンの正規表現オブジェクトを作成する構文です。一方で、`RegExp` コンストラクタは変数と組み合わせるなど、実行時に変わることがあるパターンの正規表現オブジェクトを作成できます。

例として、指定個数のホワイトスペース（空白文字）が連続した場合にマッチする正規表現オブジェクトで比較してみます。

次のコードでは、正規表現リテラルを使って3つ連続するホワイトスペースにマッチする正規表現オブジェクトを作成しています。`\s` はスペースやタブなどのホワイトスペースにマッチする特殊文字です。また、`{数字}` は指定した回数だけ繰り返しを意味する特殊文字です。

## 第 15 章 文字列

```
// 3つの連続するスペースなどにマッチする正規表現
```

```
const pattern = /\s{3}/;
```

正規表現リテラルは、ロード時に正規表現のパターンが評価されるため、`\s` の連続する回数を動的に変更することはできません。一方で、`RegExp` コンストラクタは、実行時に正規表現のパターンが評価されるため、変数を含んだ正規表現オブジェクトを作成できます。

次のコードでは、`RegExp` コンストラクタで変数 `spaceCount` の数だけ連続するホワイトスペースにマッチする正規表現オブジェクトを作成しています。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では、バックスラッシュからはじまる特殊文字は`\` (バックスラッシュ) 自体をエスケープする必要があります。

```
const spaceCount = 3;
```

```
// /\s{3}/の正規表現を文字列から作成する
```

```
// "\"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}`);
```

このように、`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作成できますが、特殊文字のエスケープが必要となります。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パフォーマンスもよいです。正規表現のパターンに変数を利用する場合などは、`RegExp` コンストラクタを利用します。

### 15.11.2 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と対応した `String` オブジェクトまたは `RegExp` オブジェクトのメソッドを利用します。

#### 正規表現によるインデックスの取得

`String#indexOf` メソッドの正規表現版とも言える `String#search` メソッドがあります。`search` メソッドは正規表現のパターンにマッチした箇所のインデックスを返し、マッチする文字列がない場合は `-1` を返します。

- `String#indexOf(検索文字列)`: 指定された文字列にマッチした箇所のインデックスを返す
- `String#search(/パターン/)`: 指定された正規表現のパターンにマッチした箇所のインデックスを返す

次のコードでは、数字が3つ連続しているかを検索し、該当した箇所のインデックスを返しています。`\d` は、1文字の数字 (0 から 9) にマッチする特殊文字です。

```
const str = "ABC123EFG";
```

```
const searchPattern = /\d{3}/;
```

```
console.log(str.search(searchPattern)); // => 3
```

### 正規表現によるマッチした文字列の取得

文字列による検索では、検索した文字列そのものがマッチした文字列になります。しかし、`search` メソッドの正規表現による検索は、正規表現パターンによる検索であるため、検索して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` メソッドでマッチした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わかりません。

```
const str = "abc123def";
// 連続した数字にマッチする正規表現
const searchPattern = /\d+/;
const index = str.search(searchPattern); // => 3
// index だけではマッチした文字列の長さがわからない
str.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、マッチした文字列を取得する `RegExp#exec` メソッドと `String#match` メソッドが用意されています。これらのメソッドは、正規表現のマッチを文字列の最後まで繰り返す `g` フラグ (global の略称) と組み合わせることでよく利用されます。また、`g` フラグの有無によって返り値が変わるのも特徴的です。

- `String#match(正規表現)`: 文字列中でマッチするものを検索する
  - マッチした場合は、マッチした文字列を含んだ特殊な配列を返す
  - マッチしない場合は、`null` を返す
  - 正規表現の `g` フラグが有効化されているときは、マッチしたすべての結果を含んだ配列を返す
- `RegExp#exec(文字列)`: 文字列中でマッチするものを検索する
  - マッチした場合は、マッチした文字列を含んだ特殊な配列を返す
  - マッチしない場合は、`null` を返す
  - 正規表現の `g` フラグが有効化されているときは、マッチした末尾のインデックスを `lastIndex` プロパティに記憶する

`String#match` メソッドは正規表現の `g` フラグなしのパターンで検索した場合、マッチしたものが見つかった時点で検索が終了します。このときの `match` メソッドの返り値である配列は `index` プロパティと `input` プロパティが追加された特殊な配列となります。

次のコードの `/[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。

```
const str = "ABC あいう DE えお";
const alphabetsPattern = /[a-zA-Z]+/;
// g フラグなしでは、最初の結果のみを含んだ特殊な配列を返す
const results = str.match(alphabetsPattern);
console.log(results.length); // => 1
// マッチした文字列はインデックスでアクセスできる
```

## 第15章 文字列

```
console.log(results[0]); // => "ABC"
// マッチした文字列の先頭のインデックス
console.log(results.index); // => 0
// 検索対象となった文字列全体
console.log(results.input); // => "ABC あいう DE えお"
```

`String#match` メソッドは正規表現の `g` フラグありのパターンで検索した場合、マッチしたすべての結果を含んだ配列を返します。

次のコードの `/[a-zA-Z]+/g` 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繰り返しマッチします。このパターンにマッチする箇所は2つあるため、`String#match` メソッドの返り値である配列にも2つの要素が含まれています。

```
const str = "ABC あいう DE えお";
const alphabetsPattern = /[a-zA-Z]+/g;
// g フラグありでは、すべての検索結果を含む配列を返す
const resultsWithG = str.match(alphabetsPattern);
console.log(resultsWithG.length); // => 2
console.log(resultsWithG[0]); // => "ABC"
console.log(resultsWithG[1]); // => "DE"
// index と input は g フラグありの場合は追加されない
console.log(resultsWithG.index); // => undefined
console.log(resultsWithG.input); // => undefined
```

`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。

`RegExp#exec` メソッドは `g` フラグなしのパターンで検索した場合、マッチした最初の結果のみを含む特殊な配列を返します。このときの `exec` メソッドの返り値である配列が `index` プロパティと `input` プロパティが追加された特殊な配列となるのは、`String#match` メソッドと同様です。

```
const str = "ABC あいう DE えお";
const alphabetsPattern = /[a-zA-Z]+/;
// 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 = alphabetsPattern.exec(str);
console.log(results.length); // => 1
console.log(results[0]); // => "ABC"
// マッチした文字列の先頭のインデックス
console.log(results.index); // => 0
// 検索対象となった文字列全体
console.log(results.input); // => "ABC あいう DE えお"
```

`RegExp#exec` メソッドは `g` フラグありのパターンで検索した場合も、マッチした最初の結果のみを含む特殊な配列を返します。この点は `String#match` メソッドとは異なります。また、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。そしてもう一度 `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 str = "ABC あいう DE えお";
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が開始される
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
const result1 = alphabetsPattern.exec(str);
console.log(result1[0]); // => "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
const result2 = alphabetsPattern.exec(str);
console.log(result2[0]); // => "DE"
console.log(alphabetsPattern.lastIndex); // => 10
// 検索結果が見つからない場合は null を返し、lastIndex プロパティは 0 にリセットされる
const result3 = alphabetsPattern.exec(str);
console.log(result3); // => null
console.log(alphabetsPattern.lastIndex); // => 0
```

この `lastIndex` プロパティが検索ごとに更新される仕組みを利用することで、`exec` を反復処理してすべての検索結果を取得できます。`exec` メソッドはマッチしなければ `null` を返すため、マッチするものがなくなれば `while` 文から自動的に脱出します。

```
const str = "ABC あいう DE えお";
const alphabetsPattern = /[a-zA-Z]+/g;
let matches;
while (matches = alphabetsPattern.exec(str)) {
  console.log(`match: ${matches[0]}, lastIndex: ${alphabetsPattern.lastIndex}`);
}
// コンソールには次のよう出力される
// match: ABC, lastIndex: 3
// match: DE, lastIndex: 10
```

このように `String#match` メソッドと `RegExp#exec` メソッドはどちらも `g` フラグによって挙動が変わります。また `RegExp#exec` メソッドは、正規表現オブジェクトの `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

`String#match` メソッドと `RegExp#exec` メソッドのどちらも正規表現のキャプチャリングに対応しています。キャプチャリングとは、正規表現中で `/パターン 1(パターン 2)/` のようにカッコで囲んだ部

## 第15章 文字列

分を取り出すことです。このキャプチャリングによって、正規表現でマッチした一部分だけを取り出せます。

`String#match` メソッド、`RegExp#exec` メソッドのどちらもマッチした結果を配列として返します。

そのマッチしてい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 [マッチした全体の文字列, ... キャプチャされた文字列] = 文字列.match(/パターン(キャプチャ)/);
```

次のコードでは、`ECMAScript` 数字の数字部分だけを取り出そうとしています。`String#match` メソッドとキャプチャリングによって数字 `(\d)` にマッチする部分を取り出しています。

```
// "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+)/;
// 返り値は 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") も返り値は同じ
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って、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、パターンの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)`  
– `^` は先頭に一致する特殊文字
- `String#endsWith: /パターン$/ .test(文字列)`  
– `$` は末尾に一致する特殊文字
- `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 str = "にわにはにわにわとりがいる";
// ^ - 検索文字列が先頭なら true
console.log(/^にわ/.test(str)); // => true
console.log(/^いる/.test(str)); // => false
// $ - 検索文字列が末尾なら true
console.log(/にわ$/ .test(str)); // => false
```

```
console.log(/いる$/ .test(str)); // => true
// 検索文字列が含まれるならtrue
console.log(/にわ/ .test(str)); // => true
console.log(/いる/ .test(str)); // => true
```

そのほかにも、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柔軟な検索が可能です。

### 15.11.3 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られます。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わからないことがあります。

次の例は、/からはじまり/で終わる文字列かを判定しようとしています。この判定を正規表現とString メソッドを使ってそれぞれ実装しています（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わかりにくいです。String メソッドの場合は、/からはじまり/で終わるかを判定してることがそのままコードに表現できています。

```
const str = "/正規表現のような文字列/";
// 正規表現で/からはじまり/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(str)); // => true
// String メソッドで/からはじまり/で終わる文字列かを判定する関数
const isRegExpLikeString = (str) => {
  return str.startsWith("/") && str.endsWith("/");
};
console.log(isRegExpLikeString(str)); // => 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したほうがよいでしょう。

「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 [MDN の正規表現ドキュメント](https://developer.mozilla.org/ja/docs/Web/JavaScript/Guide/Regular_Expressions)<sup>\*5</sup>や、コンソールで実行しながら試せ

<sup>\*5</sup> [https://developer.mozilla.org/ja/docs/Web/JavaScript/Guide/Regular\\_Expressions](https://developer.mozilla.org/ja/docs/Web/JavaScript/Guide/Regular_Expressions)

## 第 15 章 文字列

る [regex101](https://regex101.com/)<sup>\*6</sup>のようなサイトを参照してください。

## 15.12 文字列の置換/削除

文字列の一部を置換したり削除するには `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変な特性を持ちます。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 では削除でき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 str = "文字列";
// 文字列の 0 番目の削除を試みるが Strict mode では例外が発生する
delete str[0];
// => TypeError: property 0 is non-configurable and can't be deleted
```

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、文字列と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列へ置換することで、文字列を削除できます。

```
const str = "文字列";
// "文字"を"" (空文字列) へ置換することで"削除"を表現
const newStr = str.replace("文字", "");
console.log(newStr); // => "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置換できます。

```
// 検索対象となる文字列
const str = "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のだけが置換される
console.log(str.replace("にわ", "niwa")); // => "niwa にはにわにわとりがいる"
// g フラグなし正規表現の場合は、最初に一致したもののだけが置換される
console.log(str.replace(/にわ/, "niwa")); // => "niwa にはにわにわとりがいる"
```

---

<sup>\*6</sup> <https://regex101.com/>

```
// g フラグあり正規表現の場合は、繰り返し置換を行う
console.log(str.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て複雑な置換処理もできます。
replace メソッドの第二引数にはコールバック関数を渡せます。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"という文字列にマッチします。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("本日ハ晴天ナリ")); // => "本日ハ晴天ナリ"
// 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"
```

## 15.13 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異なってしまうことがあります。

ここでの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文字列中にコンテキストを持っているものを指します。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

## 第15章 文字列

素に入る文字列の種類などが決められています（詳細は「[URL Standard<sup>\\*7</sup>](#)」を参照）。

```
"https://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ほうが安全です。

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 (`baseUrl`) とパス (`pathname`) を引数に渡して利用します。

```
// 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/が含まれると考える場合があります。`getResource` 関数は、`baseUrl` の末尾に/が含まれているケースを想定していませんでした。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わかるような

---

<sup>\*7</sup> <https://url.spec.whatwg.org/>

問題が生じやすいのです。

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作ることによって安全に文字列を処理します。

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` 文字列の表記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、/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/が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 を扱うものとしてウェブ標準 API である [URL](https://developer.mozilla.org/ja/docs/Web/API/URL) オブジェクト<sup>\*8</sup>、ファイルパスを扱うものとしては Node.js のコアモジュールである [Path](https://nodejs.org/api/path.html) モジュール<sup>\*9</sup>などがあります。専用の仕組みがある場合は、直接 + 演算子で結合するような文字列処理は避けるべきです。

### 15.13.1 タグつきテンプレート関数 ES2015

JavaScript では、テンプレートとなる文字列に対して一部分だけを変更する処理を行う方法として、タグつきテンプレート関数があります。タグつきテンプレート関数とは、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合わせた表現です。関数の呼び出しに関数`テンプレート`ではなく、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。

通常の間数として呼び出した場合、関数の引数にはただの文字列が渡ってきます。

```
function tag(str) {
  // 引数 str にはただの文字列が渡ってくる
```

<sup>\*8</sup> <https://developer.mozilla.org/ja/docs/Web/API/URL>

<sup>\*9</sup> <https://nodejs.org/api/path.html>

## 第15章 文字列

```

        console.log(str); // => "template 0 literal 1"
    }
    // () をつけて関数を呼び出す
    tag(`template ${0} literal ${1}`);

```

しかし、()ではなく関数`テンプレート`と記述することで、関数が受け取る引数にはタグつきテンプレート向けの値が渡ってきます。このとき、関数の第一引数にはテンプレートの中身が`\${}`で区切られた文字列の配列、第二引数以降は`\${}`の中に書いた式の評価結果が順番に渡されます。

```

// 呼び出し方によって受け取る引数の形式が変わる
function tag(strings, ...values) {
    // strings は文字列のパーツが`${}`で区切られた配列となる
    console.log(strings); // => ["template ", " literal ", ""]
    // values には`${}`の評価値が順番に入る
    console.log(values); // => [0, 1]
}
// () をつけずにテンプレートを呼び出す
tag`template ${0} literal ${1}`;

```

どちらも同じ関数ですが、関数`テンプレート`という書式で呼び出すと渡される引数が特殊な形になります。そのため、タグつ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。

まずは引数をどう扱うかを見ていくために、タグつ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) {
    // result の初期値は strings[0] の値となる
    return strings.reduce((result, str, i) => {
        console.log([result, values[i - 1], str]);
        // それぞれループで次のような出力となる
        // 1 回目: ["template ", 0, " literal "]
        // 2 回目: ["template 0 literal ", 1, ""]
        return result + values[i - 1] + str;
    });
}
// 関数`テンプレートリテラル` という形で呼び出す
console.log(stringRaw`template ${0} literal ${1}`);
// => "template 0 literal 1"

```



ここで実装した `stringRaw` タグ関数と同様のものが、`String.raw` メソッド<sup>ES2015</sup>として提供されています。

```
console.log(String.raw`template ${0} literal ${1}`);  
// => "template 0 literal 1"
```

タグつきテンプレート関数を利用することで、テンプレートとなる文字列に対して特定の形式に変換したデータを埋め込むといったテンプレート処理が行えます。

次のコードでは、テンプレート中の変数を URL エスケープしてから埋め込むタグつきテンプレート関数を定義しています。`encodeURIComponent` 関数は引数の値を URL エスケープする関数です。`escapeURL` では受け取った変数を `encodeURIComponent` 関数で URL エスケープしてから埋め込んでいます。

```
// 変数を URL エスケープするタグ関数  
function escapeURL(strings, ...values) {  
  return strings.reduce((result, str, i) => {  
    return result + encodeURIComponent(values[i - 1]) + str;  
  });  
}  
  
const input = "A&B";  
// escapeURL タグ関数を使ったタグつきテンプレート  
const escapedURL = escapeURL`https://example.com/search?q=${input}&sort=desc`;  
console.log(escapedURL); // => "https://example.com/search?q=A%26B&sort=desc"
```

このようにタグつ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つ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。

## 15.14 終わりに

この章では、JavaScript における文字列（`String` オブジェクト）について紹介しました。文字列を処理する `String` メソッドにはさまざまなものがあり、正規表現と組み合わせて使うものも含まれます。

正規表現は、正規表現のみで 1 冊の本が作れるような JavaScript 言語内にある別言語です。詳細は [MDN の正規表現ドキュメント](https://developer.mozilla.org/ja/docs/Web/JavaScript/Guide/Regular_Expressions)<sup>\*10</sup>なども参照してください。

文字列は一見単純なオブジェクトに見えますが、文字列には URL やパスといったコンテキストを持つ文字列もあります。それらの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。また、タグつ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を自動でエスケープするといった処理を実現できます。

<sup>\*10</sup> [https://developer.mozilla.org/ja/docs/Web/JavaScript/Guide/Regular\\_Expressions](https://developer.mozilla.org/ja/docs/Web/JavaScript/Guide/Regular_Expressions)

## 第 16 章

## 文字列と Unicode

# Chapter 16

「[文字列](#)」の章で紹介したように、JavaScript は文字コードとして Unicode を採用し、エンコード方式として UTF-16 を採用しています。この UTF-16 を採用しているのは、あくまで JavaScript の内部で文字列を扱う際の文字コード（内部コード）です。そのため、コードを書いたファイル自体の文字コード（外部コード）は、UTF-8 のように UTF-16 以外の文字コードであっても問題ありません。

「[文字列](#)」の章では、これらの文字コードは意識していなかったように、内部的にどのような文字コードで扱っているかは意識せずに文字列処理ができます。しかし、JavaScript の String オブジェクトにはこの文字コード（Unicode）に特化した API もあります。また、絵文字を含む特定の文字を扱う際や「文字数」を数えるという場合には、内部コードである UTF-16 を意識しないといけない場面があります。

この章では、文字列における Unicode を意識しないといけない場面について見ていきます。また、Unicode 自体も ECMAScript と同じように歴史がある仕様であり、Unicode のすべてを紹介するには膨大な文章が必要になります。そのため、この章は JavaScript における Unicode と UTF-16 に話を限定しています。

Unicode の歴史を含めた文字コード自体について詳しく知りたい方は「[プログラマのための文字コード技術入門](#)」\*1や「[文字コード「超」研究](#)」\*2等を参照してください。

### 16.1 Code Point

Unicode はすべての文字（制御文字などの画面に表示されない文字も含む）に対して ID を定義する目的で策定されている仕様です。この「文字」に対する「一意の ID」のことを **Code Point**（符号位置）と呼びます。

Code Point を扱うメソッドの多くは、ECMAScript 2015 で追加されています。ES2015 で追加された `String#codePointAt` メソッドや `String.fromCodePoint` メソッドを使うことで、文字列と Code Point を相互変換できます。

`String#codePointAt` メソッド **ES2015** は、文字列の指定インデックスにある文字の Code Point の値を返します。

\*1 『[改訂新版] プログラマのための文字コード技術入門』矢野啓介 著、技術評論社

\*2 『文字コード「超」研究 改訂第 2 版』深沢千尋 著、ラトルズ

```
// 文字列"あ"の Code Point を取得
console.log("あ".codePointAt(0)); // => 12354
```

一方の `String.fromCodePoint` メソッド **ES2015** は、指定した Code Point に対応する文字を返します。

```
// Code Point が 12354 の文字を取得する
console.log(String.fromCodePoint(12354)); // => "あ"
// 12354 を 16 進数リテラルで表記しても同じ結果
console.log(String.fromCodePoint(0x3042)); // => "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}` のようにエスケープシーケンスとして記述できます。Unicode エスケープシーケンスでは、Code Point の 16 進数の値が必要となります。`Number#toString` メソッドの引数に基数となる 16 を渡すことで、16 進数の文字列を取得できます。

```
// "あ"の Code Point は 12354
const codePointOfあ = "あ".codePointAt(0);
// 12354 の 16 進数表現は"3042"
const hexOfあ = codePointOfあ.toString(16);
console.log(hexOfあ); // => "3042"
// Unicode エスケープで"あ"を表現できる
console.log("\u{3042}"); // => "あ"
```

## 16.2 Code Point と Code Unit の違い

Code Point (符号位置) について紹介しましたが、JavaScript の文字列の構成要素は UTF-16 で変換された Code Unit (符号単位) です (詳細は「[文字列](#)」の章を参照)。ある範囲の文字列については、Code Point (符号位置) と Code Unit (符号単位) は結果として同じ値となります。

次のコードでは、アオイという文字列の各要素を Code Point と Code Unit として表示しています。`convertCodeUnits` 関数は文字列を Code Unit の配列にし、`convertCodePoints` 関数は文字列を Code Point の配列にしています。それぞれの関数の実装はまだ理解しなくても問題ありません。

```
// 文字列を Code Unit(16 進数) の配列にして返す
function convertCodeUnits(str) {
  const codeUnits = [];
  for (let i = 0; i < str.length; i++) {
    codeUnits.push(str.charCodeAt(i).toString(16));
  }
  return codeUnits;
}
```

第 16 章 文字列と Unicode

```
// 文字列を Code Point(16 進数) の配列にして返す
function convertCodePoints(str) {
  return Array.from(str).map(char => {
    return char.codePointAt(0).toString(16);
  });
}

const str = "アオイ";
const codeUnits = convertCodeUnits(str);
console.log(codeUnits); // => ["30a2", "30aa", "30a4"]
const codePoints = convertCodePoints(str);
console.log(codePoints); // => ["30a2", "30aa", "30a4"]
```

実行した結果をまとめてみると、この文字列においては Code Point と Code Unit が同じ値になっていることがわかります。

表 16.1 文字列における Code Unit と Code Point の表

| インデックス                       | 0      | 1      | 2      |
|------------------------------|--------|--------|--------|
| 文字列                          | ア      | オ      | イ      |
| Unicode の Code Point (16 進数) | 0x30A2 | 0x30AA | 0x30A4 |
| UTF-16 の Code Unit (16 進数)   | 0x30A2 | 0x30AA | 0x30A4 |

しかし、文字列によっては Code Point と Code Unit が異なる値となる場合があります。  
先ほどと同じ関数を使い、リンゴ🍏（リンゴの絵文字）という文字列を構成する Code Unit と Code Point を見比べてみます。

```
// 文字列を Code Unit(16 進数) の配列にして返す
function convertCodeUnits(str) {
  const codeUnits = [];
  for (let i = 0; i < str.length; i++) {
    codeUnits.push(str.charCodeAt(i).toString(16));
  }
  return codeUnits;
}

// 文字列を Code Point(16 進数) の配列にして返す
function convertCodePoints(str) {
  return Array.from(str).map(char => {
    return char.codePointAt(0).toString(16);
  });
}
```

```
const str = "リンゴ🍏";
const codeUnits = convertCodeUnits(str);
console.log(codeUnits); // => ["30ea", "30f3", "30b4", "d83c", "df4e"]
const codePoints = convertCodePoints(str);
console.log(codePoints); // => ["30ea", "30f3", "30b4", "1f34e"]
```

実行した結果をまとめてみると、この絵文字を含む文字列においては Code Point と Code Unit が異なる値となることがわかります。

表 16.2 絵文字を含んだ文字列における Code Unit と Code Point の表

| インデックス                       | 0      | 1      | 2      | 3       | 4      |
|------------------------------|--------|--------|--------|---------|--------|
| 文字列                          | リ      | ン      | ゴ      | 🍏       |        |
| Unicode の Code Point (16 進数) | 0x30ea | 0x30f3 | 0x30b4 | 0x1f34e |        |
| UTF-16 の Code Unit (16 進数)   | 0x30ea | 0x30f3 | 0x30b4 | 0xd83c  | 0xdf4e |

具体的には、Code Point の要素数が 4 つなのに対して、Code Unit の要素数が 5 つになっています。また、Code Point では 1 つの Code Point が 🍏 に対応していますが、Code Unit では 2 つの Code Unit で 🍏 に対応しています。JavaScript では「文字列は Code Unit が順番に並んだもの」として扱われるためこの文字列の要素数（長さ）は Code Unit の個数である 5 つとなっています。

ある 1 つの文字に対応する ID である Code Point を、16bit (2 バイト) の Code Unit で表現するのが UTF-16 というエンコード方式です。しかし、16bit (2 バイト) で表現できる範囲は、65536 種類 (2 の 16 乗) です。現在、Unicode に登録されている Code Point は 10 万種類を超えているため、すべての文字と Code Unit を 1 対 1 の関係で表すことができません。

このような場合に、UTF-16 では 2 つ Code Unit の組み合わせ (合計 4 バイト) で 1 つの文字 (1 つの Code Point) を表現します。この仕組みをサロゲートペアと呼びます。

### 16.3 サロゲートペア

サロゲートペアでは、2 つ Code Unit の組み合わせ (合計 4 バイト) で 1 つの文字 (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に利用する領域としています。

- \uD800~\uDBFF：上位サロゲートの範囲
- \uDC00~\uDFFF：下位サロゲートの範囲

文字列中に上位サロゲートと下位サロゲートの Code Unit が並んだ場合に、2 つの Code Unit を組み合わせて 1 文字 (Code Point) として扱います。

次のコードでは、サロゲートペアの文字である「鰯 (ほっけ)」を次の 2 つの Code Unit で表現しています。Code Unit のエスケープシーケンス (\uXXXX) を 2 つ並べることで鰯という文字を表現できます。一方で、ES2015 からは Code Point のエスケープシーケンス (\u{XXXX}) も書けるため、1 つの Code Point で鰯という文字を表現することもわかります。しかし、Code Point のエスケープシーケンスで書いた場合でも、内部的に Code Unit に変換された値で保持されることは変わりません。

## 第 16 章 文字列と Unicode

```
// 上位サロゲート + 下位サロゲートの組み合わせ
console.log("\uD867\uDE3D"); // => "𪛟"
// Code Point での表現
console.log("\u{29e3d}"); // => "𪛟"
```

先ほどの例で登場した 🍏 (リンゴの絵文字) もサロゲートペアで表現される文字です。

```
// Code Unit (上位サロゲート + 下位サロゲート)
console.log("\uD83C\uDF4E"); // => "🍏"
// Code Point
console.log("\u{1F34E}"); // => "🍏"
```

このようにサロゲートペアでは、2 つの Code Unit で 1 つの Code Point を表現します。

基本的には、文字列は Code Unit が順番に並んでいるものとして扱われるため、多くの **String** のメソッドは Code Unit ごとに作用します。また、インデックスアクセスも Code Unit ごととなります。そのため、サロゲートペアで表現されている文字列では、上位サロゲート (0 番目) と下位サロゲート (1 番目) へのインデックスアクセスになります。

```
// 内部的には Code Unit が並んでいるものとして扱われている
console.log("\uD867\uDE3D"); // => "𪛟"
// インデックスアクセスも Code Unit ごととなる
console.log("𪛟"[0]); // => "\uD867"
console.log("𪛟"[1]); // => "\uDE3D"
```

絵文字や「𪛟 (ほっけ)」などのサロゲートペアで表現される文字が文字列中に含まれると、Code Unit ごとに扱う文字列処理は複雑になります。

たとえば、**String#length** プロパティは文字列における Code Unit の要素数を数えるため、**"🍏".length** の結果は 2 となります。

```
console.log("🍏".length); // => 2
```

このような場合には、文字列を Code Point ごとに処理することを考える必要があります。

## 16.4 Code Point を扱う

文字列を Code Point が順番に並んだものとして扱うには、Code Point に対応したメソッドなどを利用する必要があります。

ES2015 から文字列を Code Point ごとに扱うメソッドや構文が追加されています。次に紹介するのは、文字列を Code Point ごとに扱います。

- **CodePoint** を名前に含むメソッド
- **u** (Unicode) フラグが有効化されている正規表現
- 文字列の Iterator を扱うもの (**Destructuring**, **for...of**, **Array.from** メソッドなど)

これらの Code Point を扱う処理と具体的な使い方を見ていきます。

### 16.4.1 正規表現の . と Unicode

ES2015 では、正規表現に `u` (Unicode) フラグが追加されました。この `u` フラグをつけた正規表現は、文字列を Code Point が順番に並んだものとして扱います。

具体的に `u` フラグの有無による、(改行文字以外のどの 1 文字にもマッチする特殊文字) の動作の違いを見ていきます。

`/(.)/` のひらき/というパターンで、`.` にマッチする部分を取り出すことを例に見ていきます。

まずは、`u` フラグをつけていない正規表現と `String#match` メソッドでマッチした範囲を取り出してみます。`match` メソッドの返す値は [マッチした全体の文字列, キャプチャされた文字列] です (詳細は「[文字列](#)」の章を参照)。

実際にマッチした結果を見てみると、`.` は鯿の下位サロゲートである `\ude3d` にマッチしていることがわかります (`\ude3d` は単独では表示できないため、文字化けのように表示されます)。

```
const [all, fish] = "鯿のひらき".match(/(.)/のひらき/);
console.log(all); // => "\ude3d のひらき"
console.log(fish); // => "\ude3d"
```

つまり、`u` フラグをつけていない正規表現は、文字列を Code Unit が順番に並んだものとして扱っています。

このような意図しない結果を避けるには、正規表現に `u` フラグをつけます。`u` フラグがついた正規表現は、文字列を Code Point ごとに扱います。そのため、任意の 1 文字にマッチする `.` が鯿という文字 (Code Point) にマッチします。

```
const [all, fish] = "鯿のひらき".match(/(.)/のひらき/u);
console.log(all); // => "鯿のひらき"
console.log(fish); // => "鯿"
```

基本的には正規表現に `u` フラグをつけて問題となるケースは少ないはずです。なぜなら、サロゲートペアの片方だけにマッチしたい正規表現を書くケースはまれであるためです。

### 16.4.2 Code Point の数を数える

`String#length` プロパティは、文字列を構成する Code Unit の個数を表すプロパティです。そのためサロゲートペアを含む文字列では、`length` の結果が見た目より大きな値となる場合があります。

```
// Code Unit の個数を返す
console.log("🍎".length); // => 2
console.log("\uD83C\uDF4E"); // => "🍎"
console.log("\uD83C\uDF4E".length); // => 2
```

JavaScript には、文字列における Code Point の個数を数えるメソッドは用意されていません。これ

## 第 16 章 文字列と Unicode

を行うには、文字列を Code Point ごとに区切った配列へ変換して、配列の長さを数えるのが簡潔なやり方です。

`Array.from` メソッド **ES2015** は、引数に iterable なオブジェクトを受け取り、それを元にした新しい配列を返します。iterable オブジェクトとは `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の総称で、`for...of` 文などで反復処理が可能なオブジェクトです（詳細は「[ループと反復処理](#)」の章の「[for...of 文](#)」を参照）。

文字列も iterable オブジェクトであるため、`Array.from` メソッドによって 1 文字（厳密には Code Point）ごとに区切った配列へと変換できます。先ほども紹介したように、文字列を iterable として扱う場合は Code Point ごとに処理を行います。

```
// 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("リンゴ🍏");
console.log(codePoints); // => ["リ", "ン", "ゴ", "🍏"]
// Code Point の個数を数える
console.log(codePoints.length); // => 4
```

しかし、Code Point の数を数えた場合でも、直感的な結果にならない場合もあります。なぜなら、Code Point には制御文字などの視覚的に見えないものも定義されているためです。そのため、文字として数えたくないものは無視するなど、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。残念ながら、ビルトインメソッドにはこれらを簡単に扱う方法は用意されていません。

### 16.4.3 Code Point ごとに反復処理をする

先ほど紹介した `Array.from` メソッドを使えば、文字列を Code Point で区切った文字の配列へと変換できます。配列にすれば、あとは「[ループと反復処理](#)」の章で学んだ方法を使って、Code Point ごとに反復処理ができます。

次のコードでは、文字列中に登場する 🍏 の個数を数えています。`countOfCodePoints` 関数は、`Array.from` で Code Point ごとの配列にし、配列を `codePoint` でフィルターした結果できた配列の要素数を返します。

```
// 指定した codePoint の個数を数える
function countOfCodePoints(str, codePoint) {
  return Array.from(str).filter(item => {
    return item === codePoint;
  }).length;
}
console.log(countOfCodePoints("🍏🍇🍏🍌🍏", "🍏")); // => 2
```

`for...of` による反復処理も文字列を Code Point ごとに扱えます。これは、`for...of` 文が対象を Iterator として列挙するためです。

先ほどのコードと同じ `countOfCodePoints` 関数を `for...of` を使って実装してみます。



```
// 指定した codePoint の個数を数える
function countOfCodePoints(str, codePoint) {
  let count = 0;
  for (const item of str) {
    if (item === codePoint) {
      count++;
    }
  }
  return count;
}

console.log(countOfCodePoints("🍌🍌🍌🍌", "🍌")); // => 2
```

## 16.5 まとめ

この章では、文字列と Unicode の関係について簡潔に紹介しました。Unicode にはこの章で紹介しきれなかった表現もあります。また、JavaScript には Unicode をキレイに扱う API が用意されているとは言い切れない部分もあります。

一方で「[文字列](#)」の章で紹介したように、Code Unit や Code Point を意識しなくても柔軟で強力な文字列処理ができます。しかし、近年は絵文字を利用するケースが多くなったため、Code Point を意識したプログラミングが必要となるケースも増えています。

Unicode は ECMAScript とは独立した仕様であるため、文字列を扱う悩みはプログラミング言語を問わずに出てくる共通の課題です。特に Java は JavaScript と同じく UTF-16 をエンコード方式として採用しているため、類似する問題が見られます。そのため、JavaScript で文字列処理の問題にぶつかった際には、他の言語ではどうしているかを調べることも重要です。

## 第17章

### ラッパーオブジェクト

# Chapter 17

JavaScript のデータ型はプリミティブ型とオブジェクトに分けられます（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メソッドである `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` のインスタンスメソッドを呼び出せるのは一見不思議です。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。

### 17.1 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れます。

```
// "input value"の値をラップした String のインスタンスを生成
const str = new String("input value");
// String のインスタンスメソッドである toUpperCase を呼び出す
str.toUpperCase(); // => "INPUT VALUE"
```

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）オブジェクトと言えます。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びます。

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。

17.2 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

| ラッパーオブジェクト | プリミティブ型 | 例            |
|------------|---------|--------------|
| Boolean    | 真偽値     | true や false |
| Number     | 数値      | 1 や 2        |
| String     | 文字列     | "文字列"        |
| Symbol     | シンボル    | Symbol("説明") |



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。

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。

```
// プリミティブの文字列は"string"型
const str = "文字列";
console.log(typeof str); // => "string"
// ラッパーオブジェクトは"object"型
const stringWrapper = new String("文字列");
console.log(typeof stringWrapper); // => "object"
```

17.2 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とき、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せるようになります。

```
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
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、
// 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
// つまり、上のコードは下のコードと同じ意味である
(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出すこともできます。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出せます。

```
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"
```

## 第 17 章 ラッパーオブジェクト

このように、プリミティブ型の値から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 には、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に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を意識する必要がなくなります。

```
// OK: リテラルを使う
const str = "文字列";

// NG: ラッパーオブジェクトを使う
const stringWrapper = new String("文字列");
```

## 17.3 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> このようなプリミティブ型からオブジェクト型への変換はボックス化（ボクシング）、逆にオブジェクト型からプリミティブ型への変換はボックス化解除（アンボクシング）と呼ばれます。

## 第18章

### 関数とスコープ

# Chapter 18

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ひもづけられた新しいスコープが作成されます。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言えます。

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からです。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取得する値がどのように決まるかを見ていきます。

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、古い仕組みについても補足していきます。

#### 18.1 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の内側でのみ参照でき、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。

次のコードでは、**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** が発生します。

```
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  
}
```

## 第 18 章 関数とスコープ

```
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
```

このコードを見てわかるように、変数 **x** は **fn** 関数のスコープにひもづけて定義されます。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**を持てますが、仮引数は関数のスコープにひもづけて定義されます。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
```

このような、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"a"を定義すると SyntaxError となる  
let a;  
let a;
```

一方、スコープが異なれば同じ名前の変数を宣言できます。次のコードでは、**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わかります。

```
// 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、グローバルな空間内で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同じ名前の変数をスコープごとに定義できるため、スコープの役割は重要です。

## 18.2 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」のルールを考えるとわかりやすいです。次のコードでは、ループごとに `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、ループごと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
```

## 18.3 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"  
  }  
}
```

変数を参照する際には、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。上記のコードでは、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、外側のどのスコープにも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ています。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** の例外が発生していました。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、変数を参照する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。

## 18.4 グローバルスコープ

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。

## 第18章 関数とスコープ

```
// プログラム直下はグローバルスコープ
const x = "x";
console.log(x); // => "x"
```

グローバルスコープで定義した変数は**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からです。

```
// 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
{
  // ブロックスコープ内には該当する変数が定義されていない -> 外側のスコープへ
  console.log(globalVariable); // => "グローバル"
}
// 関数スコープ
function fn() {
  // 関数スコープ内には該当する変数が定義されていない -> 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（「[データ型とリテラル](#)」の章の「[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。2つ目は実行環境（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(isNaN); // => isNaN
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と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ります。このように内側のスコープで外側のスコープと同じ名前の変数を定義することで、外側の変数が参照できなくなることを**変数の隠蔽** (shadowing) と呼びます。

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影響範囲がその関数の中だけにとどまります。

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には実行環境 (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するためです。関数などを活用して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っても影響範囲を限定できます。

#### 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のはよいことです。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、その逆も起こ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スコープ内に定義するというものです。

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す。`Date.now` メソッドを使った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が得られます。

```
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コールバック関数として引数に渡す
const measureTask = (taskFn) => {
```

## 第 18 章 関数とスコープ

```
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` という変数をグローバルスコープからなくせました。また、実行時間を計測するという処理を `measureTask` という関数にしたことで再利用できます。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必要があります。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。

## 18.5 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 `var` を改善したバージョン」と紹介したように、`let` は `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` の例外が発生して参照できません。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前に参照したためエラーとなっているのがわかりま<sup>\*1</sup>。

```
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わかります。

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、宣言部分が暗黙的にもっとも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も近い関数またはグローバルスコープに変数をひもづけます。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
```

---

<sup>\*1</sup> この仕組みは Temporal Dead Zone (TDZ) と呼ばれます。

## 第18章 関数とスコープ

```
    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() {
  // もっとも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**部分がもっとも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 **var** は **let**、**const** と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関数スコープの先頭に移動するという予測しにくい問題を持っています。この問題のもっとも簡単な回避方法は **var** を使わないことですが、**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。

## 18.6 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、もっとも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られます。つまり先ほどのコードは、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られる
function hello(){
    return "Hello";
}
```

```
hello(); // => "Hello"
```

`function` キーワードによる関数宣言も巻き上げられます。しかし、`var` による変数宣言の巻き上げとは異なり、問題となることはほとんどありません。なぜなら、実際に巻き上げられた関数を呼び出せるためです。

注意点として、`var` や `let` など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られます。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` となるため呼び出せません（「関数と宣言」の章の「関数式」を参照）。

```
// hello 変数は巻き上げられ、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";
};
```

#### 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によって `foo` 変数は匿名関数の外側からはアクセスできません。

```
// 匿名関数を宣言 + 実行を同時に行っている
(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"
```

## 第 18 章 関数とスコープ

```
}());  
// foo 変数のスコープ外  
console.log(typeof foo === "undefined"); // => true
```

関数を**式**として定義して、そのまま呼び出しています。`function` からはじまってしまうと JavaScript エンジンが**関数宣言**と解釈してしまうため、無害なカッコなどで囲んで**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短く書くことができ、余計な関数定義がグローバルスコープに残りません。

```
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置き換えられます。

```
{  
    // 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
```

## 18.7 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わ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」という関数を持つ性質のことです。

クロージャーは言葉で説明しただけではわかりにくい性質です。このセクションでは、クロージャーを使ったコードがどのように動くのかを理解することを目標にします。



次の例では `createCounter` 関数が、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` 変数に代入しています。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` 関数を実行して、その戻り値を `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わかります。

```
// increment 関数を定義して返す関数
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 は別々の状態を持っている
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ーがあります。クロージャーは直感的に理解しにくいので、まずはクロージャー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。

### 18.7.1 静的スコープ

クロージャーを理解するために、今まで意識してこなかったスコープの性質について見ていきます。JavaScript のスコープには、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があります。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わ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

## 第 18 章 関数とスコープ

コープと関数 `run` の中で、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。

結論から言えば、`printX` 関数中にある識別子 `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 が出力される
}

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てグローバルスコープの変数 `x` を参照することがわかります。

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` 関数中に書かれた `x` という識別子は、`run` 関数の実行とは関係なく、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。

### 動的スコープ

JavaScript は静的スコープです。しかし、動的スコープという呼び出し元により識別子がどの変数を参照するかが変わる仕組みを持つ言語もあります。

次のコードは、動的スコープの動きを説明する疑似的な言語のコード例です。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。

```
// 動的スコープの疑似的な言語のコード例 (JavaScript ではありません)
// 変数 x を宣言
var x = 10;

// printX という関数を定義
fn printX() {
  // 動的スコープの言語では、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
  // print 関数でコンソールへログを出力する
  print(x);
}

fn run() {
  // 呼び出し元のスコープで、変数x を定義している
  var x = 20;
  printX();
}

printX(); // ここでは 10 が出力される
run();   // ここでは 20 が出力される
```

このように関数呼び出し時に呼び出し元のスコープの変数を参照する仕組みを**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で決まるため、上記のような動的スコープではありません。しかし、JavaScript でも `this` という特別なキーワードだけは、呼び出し元によって動的に参照先が変わります。`this` というキーワードについては次の章で解説します。

## 18.8 メモリ管理の仕組み

プログラミング言語は、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だからです。そのため、メモリからデータがあふれないように、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。

## 第 18 章 関数とスコープ

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的にメモリ上から解放する仕組みのことです。

JavaScript にはガベージコレクションがあるため、手動でメモリを解放するコードを書く必要はありません。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てメモリ上から解放されます<sup>\*2</sup>。

```
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、その関数の実行が終了したら解放される」というのがあります。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「関数の実行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す。

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。どこからも参照されなくなったものは、ガベージコレクションによって回収されて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。

---

<sup>\*2</sup> 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。

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、そのデータが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いる -> 解放されない
```

つまり、関数の実行が終了した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わかります。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。

### 18.8.1 クロージャールがなぜ動くのか

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

クロージャールとはこの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持てる仕組みのことを言います。

最初にクロージャール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 createCounter 関数のスコープに定義された変数 count を
    // 参照している
    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
console.log(myCounter()); // => 1
```

## 第 18 章 関数とスコープ

```
console.log(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` 関数を經由して `count` 変数を参照している
- `myCounter` 変数を実行した後も `count` 変数への参照は保たれている

`myCounter` → `increment` → `count`

`count` 変数を参照するものがあるため、`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きます。

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` 関数内から外側の `createCounter` 関数スコープにある `count` 変数を参照している」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「参照され続けている変数のデータが保持される」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が、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持っていることを指します。

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` 関数が定義されます。そのため、`createCounter` 関数を実行すると、それぞれ別々の `increment` 関数が定義され、別々の `count` 変数を参照します。

次のように `createCounter` 関数を複数呼び出してみると、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  
console.log(countUp()); // => 1  
console.log(newCountUp()); // => 1
```

### 18.8.2 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。この `privateCount` 変数は、外のグローバルスコープからは直接参照できません。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で、グローバルに定義する変数を減らせています。

```
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();
console.log(counter()); // => "1 回目"
console.log(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、クロージャー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条件を後から定義できるなどの柔軟性が得られます。

```
function greaterThan(n) {
  return function(m) {
    return m > n;
  };
}

// 5 より大きな値かを判定する関数を作成する
const greaterThan5 = greaterThan(5);
console.log(greaterThan5(4)); // => false
console.log(greaterThan5(5)); // => false
console.log(greaterThan5(6)); // => true
```

## 第 18 章 関数とスコープ

クロージャーは、変数が参照する値が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 には、関数を短く定義できる Arrow Function や高階関数である `Array#forEach` 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が大切です。

## 状態を持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ため直接プロパティに値を代入できます。そのため、クロージャーを使わなくても、次のように関数にプロパティとして状態を持たせることが可能です。

```
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;  
}  
// 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;  
// 呼び出すごとに count が更新される  
console.log(countUp()); // => 1  
console.log(countUp()); // => 2
```

しかし、この方法は推奨されていません。なぜなら、関数の外から `count` プロパティを変更できるからです。関数オブジェクトのプロパティは外からも参照でき、そのプロパティ値は変更できます。関数の中でのみ参照可能な状態を扱いたい場合には、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;  
// 呼び出すごとに count が更新される  
console.log(countUp()); // => 1  
// 直接値を変更できてしまう  
countUp.count = 10;  
console.log(countUp()); // => 11
```



## 18.9 まとめ

この章では関数を中心にスコープについて学びました。

- 関数やブロックはスコープを持つ
- スコープはネストできる
- もっとも外側にはグローバルスコープがある
- スコープチェーンは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
- `var` キーワードでの変数宣言や `function` での関数宣言では巻き上げが発生する
- クロージャは静的スコープとメモリ管理の仕組みからなる関数を持つ性質

## 第19章

### 関数と this

# Chapter 19

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。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けます。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、次の章である「**クラス**」で扱います。この章ではさまざまな条件での **this** について扱いますが、**this** が実際に使われるのはメソッドにおいてです。そのため、あらゆる条件下での **this** の動きを覚える必要はありません。

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。また、実際にどのような状況で問題が発生するかを知り、**this** の動きを予測可能にするにはどのようにするかを見ていきます。

#### 19.1 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。どの実行コンテキストで JavaScript のコードを評価するかは、実行環境によってやり方が異なります。この章では、ブラウザの **script** 要素と **type** 属性を使い、それぞれの実行コンテキストを明示しながら **this** の動きを見ていきます。

トップレベル（もっとも外側のスコープ）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** を使うと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。

### 19.1.1 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、トップレベルのスコープに書かれた **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には、実行環境ごとに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** 属性を指定していない場合は、実行コンテキストが“Script”として実行されます。この **script** 要素の直下にした **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>
```

### 19.1.2 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、そのトップレベルのスコープに書かれた **this** は常に **undefined** となります。

ブラウザで、**script** 要素に **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** 要素の直下にした **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、トップレベルのスコープの **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ほうがよいです。

## 19.2 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 とそれ以外の関数定義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す。

### 19.2.1 関数の種類

「[関数と宣言](#)」の章で詳しく紹介していますが、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。

## 第19章 関数と this

```
// function キーワードからはじめる関数宣言
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せます。

```
// 関数宣言
function fn() {}
// 関数呼び出し
fn();
```

### 19.2.2 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**と言い、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 obj = {
  // 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 obj = {
  // 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せます。

```
const obj = {
```

```
// メソッドの定義
method() {
}

};
// メソッド呼び出し
obj.method();
```

関数定義とメソッドの定義についてまとめると、次のようになります。

| 名前                                    | 関数 | メソッド |
|---------------------------------------|----|------|
| 関数宣言 (function fn(){})                | ✓  | ✗    |
| 関数式 (const fn = function(){})         | ✓  | ✓    |
| Arrow Function(const fn = () => {})   | ✓  | ✓    |
| メソッドの短縮記法 (const obj = { method(){}}) | ✗  | ✓    |

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。

### 19.3 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。言い方を変えると **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ときに決まります。

次のコードは疑似的なものです。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。

```
// 疑似的な 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(暗黙的に渡される this の値, 仮引数) {
  console.log(this); // => 暗黙的に渡される 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ブラケット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

## 第19章 関数と this

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、ブラケット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注意が必要です。

## 19.3.1 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、関数宣言で関数 `fn1` と関数式で関数 `fn2` を定義し、それぞれの関数内で `this` を返します。定義したそれぞれの関数を `fn1()` と `fn2()` のようにただの関数として呼び出しています。このとき、ベースオブジェクトはないため、`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"; と改めて strict mode を明示しています。なぜなら、strict mode ではない状況で this が undefined の場合は、this がグローバルオブジェクトを参照するように変換される問題があるためです。

strict mode は、このような意図しにくい動作を防止するために導入されています。しかしながら、strict mode のメソッド以外の関数における this は undefined となるため使い道がありません。そのため、メソッド以外で this を使う必要はありません。

### 19.3.2 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、そのメソッドが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 obj となり、this は obj となります。

```
const obj = {
  // 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 (obj) 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
console.log(obj.method1()); // => obj
console.log(obj.method2()); // => obj
```

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

## 第19章 関数と this

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わかります。このときのベースオブジェクトはドットでつないだ一番左の **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
```

## 19.4 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** にはいろいろな問題があります。

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。



### 19.4.1 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持つプロパティのことで、プロパティは変数に代入し直せるためです。

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とき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いきます。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投げます。

```
"use strict";
const person = {
  fullName: "Brendan Eich",
  sayName: function() {
    // this は呼び出し元によって異なる
    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** を参照しようとして例外が発生しています。

```
"use strict";
// const say = person.sayName; は次のようなイメージ
const say = function() {
  return this.fullName;
};
```

## 第19章 関数と this

```
// this は undefined となるため例外を投げる
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** は定義したときではなく実行したときに決定されます。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。

1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1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、残りの引数には呼び出す関数の引数を指定します。暗黙的に渡される **this** の値を明示的に渡せるメソッドと言えます。

```
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 関数を呼び出す
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 関数を呼び出す
// 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 が不要な場合は、null を渡す
console.log(add.call(null, 1, 2)); // => 3
console.log(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 関数をラップした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");
console.log(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だということがわかります。

## 第19章 関数と this

```
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");
};

console.log(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。

## 19.4.2 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` メソッドなどの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、`undefined.prefix` は参照できないため `TypeError` の例外が発生します。

```
"use strict";

// strict mode を明示しているのは、this がグローバルオブジェクトに暗黙的に
// 変換されるのを防止するため
const Prefixer = {
  prefix: "pre",
  /**
   * strings 配列の各要素に prefix をつける
   */
  prefixArray(strings) {
    return strings.map(function(str) {
      // 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 + "-" + str;
    });
}
};
// 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(str) {
        return this.prefix + "-" + str;
    });
}
// ...

```

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。

先ほどの例では匿名関数をコールバック関数として直接メソッドに渡していますが、一度 `callback` 変数に入れてから渡しても結果は同じです。

```

"use strict";
// strict mode を明示しているのは、this がグローバルオブジェクトに暗黙的に
// 変換されるのを防止するため
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(str) {
            return this.prefix + "-" + str;
        };
        return strings.map(callback);
    }
}

```

## 第19章 関数と this

```

    };
    // 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改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。次のコードでは、**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(str) {
      // this ではなく that を参照する
      return that.prefix + "-" + str;
    });
  }
};
// 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、次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(str) {
        // 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 + "-" + str;
      }, 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、これらの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。そもそもメソッド呼び出しとその中で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

通常の間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** の値を受け取りません。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に外側の関数（この場合は **prefixArray** メソッド）を探索します。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けます。

```

"use strict";
const Prefixer = {
  prefix: "pre",
  prefixArray(strings) {
    return strings.map((str) => {
      // Arrow Function 自体は this を持たない
      // this は外側の prefixArray 関数を持つ this を参照する
      // そのため this.prefix は "pre" となる
      return this.prefix + "-" + str;
    });
  }
};
// このとき、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

```

## 第 19 章 関数と this

```
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。コールバック関数内での **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** の関係についてより詳しく見ていきます。

## 19.5 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のは、Arrow Function は **this** を暗黙的な引数として受けつ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します。そのため、Arrow Function 内の **this** の参照で、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「[関数とスコープ](#)」の章の「[スコープチェーン](#)」を参照）。また、**this** は ECMAScript のキーワードであるため、ユーザーは **this** という変数を定義できません。

```
// this はキーワードであるため、ユーザーは this という名前の変数を定義できない
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）に決定されます（詳細は「[関数とスコープ](#)」の章の「[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 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 を見ていきます。

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** の外側には関数がないため、「自身より外側のスコープに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;
};
console.log(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() と呼び出したときの this は undefined(strict mode)
const innerArrowFunction = outer();
console.log(innerArrowFunction()); // => undefined
```

### 19.5.1 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 をより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ば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ってしまう `this` を使うことはできません。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方法を取っていました。

```
// 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 obj = {
  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 などを使って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 obj = {
  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を考えると **this** を扱えます。

```
const Prefixer = {  
  prefix: "pre",  
  prefixArray(strings) {  
    return strings.map((str) => {  
      // Prefixer.prefixArray() と呼び出されたとき  
      // this は常に Prefixer を参照する  
      return this.prefix + "-" + str;  
    });  
  }  
};  
  
const prefixedStrings = Prefixer.prefixArray(["a", "b", "c"]);  
console.log(prefixedStrings); // => ["pre-a", "pre-b", "pre-c"]
```

### 19.5.2 Arrow Function は this を bind できない

Arrow Function で定義した関数では `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** を持てないためです。

次のように Arrow Function で定義した関数に対して `call` で **this** を指定しても、**this** の参照先が代わっていないことがわかります。同様に `apply` や `bind` メソッドを使った場合も **this** の参照先は変わりません。

```
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  
console.log(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では呼び出し時に、ベースオブジェクトが **this** の値として暗黙的な引数のように渡されます。一方、Arrow Function の関数は呼び出し時に **this** を受け取らず、**this** の参照先は定義時に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

第 19 章 関数と this

更できます。

```
const obj = {
  method() {
    const arrowFunction = () => {
      return this;
    };
    return arrowFunction();
  }
};
// 通常の this は obj.method の this と同じ
console.log(obj.method()); // => obj
// obj.method の this を変更すれば、Arrow Function の this も変更される
console.log(obj.method.call("THAT")); // => "THAT"
```

19.6 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                                                | this の評価結果 |
|----------|-----------------|----------------------------------------------------|------------|
| Script   | * <sup>*1</sup> | this                                               | global     |
| Script   | *               | const fn = () => this                              | global     |
| Script   | NO              | const fn = function(){ return this; }              | global     |
| Script   | YES             | const fn = function(){ return this; }              | undefined  |
| Script   | *               | const obj = { method: () => { return this; } }     | global     |
| Module   | YES             | this                                               | undefined  |
| Module   | YES             | const fn = () => this                              | undefined  |
| Module   | YES             | const fn = function(){ return this; }              | undefined  |
| Module   | YES             | const obj = { method: () => { return this; } }     | undefined  |
| *        | *               | const obj = { method(){ return this; } }           | obj        |
| *        | *               | const obj = { method: function(){ return this; } } | obj        |

実際にブラウザで実行した結果は「[What is 'this' value in JavaScript](#)」<sup>\*2</sup>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。メソッド以外にお

<sup>\*1</sup> \*はどの場合でも this の評価結果に影響しないことを示しています。

<sup>\*2</sup> <https://azu.github.io/what-is-this/>

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<sup>\*3</sup>。

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 を使うことでわ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。

---

<sup>\*3</sup> ES2015 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>

## 第20章

### クラス

# Chapter 20

「クラス」と一言にい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指すことにします。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を定義した構造です。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思えますが、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です。JavaScript では ES2015 より前までは `class` 構文はなく、関数を使ってクラスのようなものを表現して扱っていました。

ES2015 でクラスを表現するための `class` 構文が導入されましたが、この `class` 構文で定義したクラスは関数オブジェクトの一種です。`class` 構文ではプロトタイプベースの継承の仕組みを使って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言えます<sup>\*1</sup>。

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。このように関数とクラスは似ている部分が多いです。

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。

### 20.1 クラスの定義

クラスを定義するには `class` 構文を使います。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。`constructor` メソッドに定義した処理は、クラスをインスタンス化したときに自動的に呼び出されます。

<sup>\*1</sup> `class` 構文でしか実現できない機能はなく、読みやすさやわ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（シンタックスシュガー）と呼ばれることがあります。

```
class MyClass {
    constructor() {
        // コンストラクタ関数の処理
        // インスタンス化されるときに自動的に呼び出される
    }
}
```

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。省略した場合でも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 {

}
```

## 20.2 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();
```

## 第 20 章 クラス

```
// 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
```

このままでは何も処理が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する際に自動的に呼び出されます。コンストラクタ関数内での **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** 座標の値を持つ **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、コンストラクタの中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**) を初期化する場所であり、通常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、コンストラクタ関数が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。

次のコードのようにコンストラクタで返した値が **new** 演算子で呼び出した際の返り値となります。このような書き方は混乱を生むため避けるべきです。

```
// 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
```

## 第 20 章 クラス

クラス名は大文字ではじめる

JavaScript では慣習としてクラス名には大文字ではじまる名前をつけます。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という合理的な理由で好まれています。



```
class Thing {}  
const thing = new Thing();
```

## 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で、これも class 構文という統一した記法が導入された理由の 1 つです。

次のコードは、関数でクラスを実装した 1 つの例です。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。

```
// 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();  
  
// class 構文でのクラス  
class MyClass {  
}  
// 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、関数でクラスのようなものを実装した場合には、関数として呼び出してしまう問題があります。このような問題を避けるためにもクラスは `class` 構文を使って実装します。

## 20.3 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作成したインスタンスが持つ動作となります。

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ない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**インスタンスメソッド**とも呼ばれます。

## 第20章 クラス

この書籍では、プロトタイプメソッド（インスタンスメソッド）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 インスタンス
counterA.increment();
// 各インスタンスの持つプロパティ（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わかります。

```
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密接に関係しています。プロトタイプメソッドの仕組みについては後ほど解説します。

### 20.3.1 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方法もあります。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である `this` 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();
// 各インスタンスの持つプロパティ（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、インスタンスメソッドです。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() {
```

## 第20章 クラス

```

        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); // => true

```

また、プロトタイプメソッドとは異なり、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（「[関数と this](#)」の章の「[対処法: 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;
// 呼び出し方法（ベースオブジェクト）に依存しないため、this がインスタンスを参照する
console.log(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（「[関数と this](#)」の章の「[問題: this を含むメソッドを変数に代入した場合](#)」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

```

このように、インスタンスに対して Arrow Function でメソッドを定義することで **this** の参照先を固定化できます。

## 20.4 クラスのアクセッ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**アクセッサプロパティ**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) に対するアクセッサプロパティを定義しています。アクセッ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する値が入りますが、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、**NumberWrapper** クラスの **value** プロパティをアクセッサプロパティとして定義しています。**value** プロパティへアクセスした際にそれぞれ定義した getter と setter が呼ばれているのがわかります。このアクセッサプロパティで実際に読み書きされているのは、**NumberWrapper** インスタンスの **\_value** プロパティとなります。

## 第 20 章 クラス

```
class NumberWrapper {
  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 numberWrapper = new NumberWrapper(1);
// "getter"とコンソールに表示される
console.log(numberWrapper.value); // => 1
// "setter"とコンソールに表示される
numberWrapper.value = 42;
// "getter"とコンソールに表示される
console.log(numberWrapper.value); // => 42
```

## プライベートプロパティ

`NumberWrapper#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 2019) では、外から原理的に参照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。しかし、現時点でも `WeakSet` などを使うことで疑似的なプライベートプロパティを実現できます。`WeakSet` については「[Map/Set](#)」の章で解説します。

### 20.4.1 `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要素は自動的に削除される仕様になっています。



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

`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
```

## 第20章 クラス

```
// 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では、プロパティのようでありながら実際にアクセスした際には他のプロパティと連動する動作を実現できます。

## 20.5 静的メソッド

インスタンスメソッドは、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って初期化しています。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って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

ます。

```
class ArrayWrapper {
  constructor(array = []) {
    this.array = array;
  }

  // rest parameters として要素を受けつける
  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;
  }
}
```

## 第 20 章 クラス

```
}

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、クラスのインスタンスは参照できません。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する処理を書くために利用されます。

## 20.6 2 種類のインスタンスメソッドの定義

クラスでは、2 種類のインスタンスメ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 2 つ定義した場合はどうなるのでしょうか？

次のコードでは、`ConflictClass` クラスに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(); // "プロトタイプメソッド"
```

この実行結果から次のことがわかります。

- 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オブジェクト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**とについていることからわかるように、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す。

## 20.7 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。なぜなら、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については「**プロトタイプオブジェクト**」の章で簡単に紹介していましたが、改めて解説していきます。

**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

## 第20章 クラス

`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わかります。

```
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` メソッドもプロトタイプオブジェクトに定義されており、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  
// クラス#constructor はクラス自身を参照する  
console.log(MyClass.prototype.constructor === MyClass); // => true
```

このように、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、インスタンスオブジェクトのメソッドとは異なるオブジェクトに定義されています。そのため、それぞれの方法でメソッドを定義しても、上書きされることはありません。

## 20.8 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せます。

```
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"
```

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）を参照するときに、`[[Prototype]]` 内部プロパティまで探索する処理

### 20.8.1 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へ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 ECMAScript の仕様で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にはアクセスできません。ここでは説明のために、`[[プロパティ名]]` という書式で ECMAScript の仕様上に存在する内部プロパティを表現しています。

`[[Prototype]]` 内部プロパティへプロパティのようににはアクセスできませんが、`Object.getPrototypeOf` メソッドで `[[Prototype]]` 内部プロパティを参照できます。


次のコードでは、`instance` オブジェクトの `[[Prototype]]` 内部プロパティを取得しています。その取得した結果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 MyClassPrototype = Object.getPrototypeOf(instance);  
console.log(MyClassPrototype === 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。

## 第 20 章 クラス

[[Prototype]] 内部プロパティを読み書きする

 Object.getPrototypeOf(オブジェクト) でオブジェクトの [[Prototype]] を読み取ることができます。一方、Object.setPrototypeOf(オブジェクト, プロトタイプオブジェクト) でオブジェクトの [[Prototype]] にプロトタイプオブジェクトを設定できます。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ッ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。

**20.8.2 プロパティの参照とプロトタイプチェーン**

プロトタイプオブジェクトのプロパティがどのようにインスタンスから参照されるかを見ていきます。

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（仕様上の内部的なプロパティ）の参照先である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）
3. どこにもなかった場合は **undefined**

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、実際に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疑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動作の疑似的なコード
class MyClass {
  method() {
    console.log("プロトタイプのメソッド");
  }
}

const instance = new MyClass();
// instance.method() を実行する場合
// 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をインスタンスから呼び出せます。

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って表現していることは知っておくとよいでしょう。

## 20.9 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。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です。

## 第 20 章 クラス

### 20.9.1 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();
```

### 20.9.2 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）を持ちます。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。

```
// 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）で何も処理し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。

次のコードの `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。`constructor` メソッドの引数をすべて受け取り、そのまま `super` へ引数の順番を維持して渡します。

```
class Parent {}
class Child extends Parent {
    constructor(...args) {
        super(...args); // 親クラスに引数をそのまま渡す
    }
}
```

### 20.9.3 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 構文では必ず親クラスのコンストラクタ処理（`super()` の呼び出し）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";
    }
}
```

## 第20章 クラス

```
}  
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"
```

## 20.9.4 プロトタイプ継承

次のコードでは `extends` キーワードを使って `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出せます。

`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により、`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を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。

### 20.9.5 静的メソッドの継承

インスタンスとクラスのプロトタイプオブジェクトとの間には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。

簡単に言えば、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。

### 20.9.6 `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` の中で `super.method()` と書くことで `Parent#method` を呼び出

## 第20章 クラス

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 を呼び出す
    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 を呼び出す
    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"
```

### 20.9.7 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](#)」の章で見てください。

## 20.10 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` の性質である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` といったアクセッ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;
    }
  }
}
```

## 第 20 章 クラス

```
        } 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 が元々持つ length プロパティや Array.from メソッドなどを継承しているので利用できます。

## 20.11 まとめ

この章ではクラスについて学びました。

- JavaScript のクラスはプロトタイプベース
- クラスは `class` 構文で定義できる
- クラスで定義したメソッドはプロトタイプオブジェクトとプロトタイプチェーンの仕組みで呼び出せる
- アクセッサプロパティは `getter` と `setter` のメソッドを定義することでプロパティのように振る舞う
- クラスは `extends` で継承できる
- クラスのプロトタイプメソッドと静的メソッドはどちらも継承される



## 第 21 章

### 例外処理

# Chapter 21

この章では JavaScript における例外処理について学びます。

#### 21.1 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。

`try...catch` 構文の `try` ブロック内で例外が発生すると、`try` ブロック内のそれ以降の処理は実行されず、`catch` 節に処理が移行します。`catch` 節は、`try` ブロック内で例外が発生すると、発生したエラーオブジェクトとともに呼び出されます。`finally` 節は、`try` ブロック内で例外が発生したかどうかには関係なく、必ず `try` 文の最後に実行されます。

次のコードでは、`try` ブロックで例外が発生し、`catch` 節の処理が実行され、最後に `finally` 節の処理が実行されます。

```
try {
  console.log("try 節:この行は実行されます");
  // 未定義の関数を呼び出して ReferenceError 例外が発生する
  undefinedFunction();
  // 例外が発生したため、この行は実行されません
} 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("catch 節:この行は実行されます");
  console.log(error instanceof ReferenceError); // => true
  console.log(error.message); // => "undefinedFunction is not defined"
} finally {
  // このブロックは例外の発生に関係なく必ず実行される
  console.log("finally 節:この行は実行されます");
}
```

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`

## 第 21 章 例外処理

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.error(error);
}
// finally 節のみ
try {
    undefinedFunction();
} finally {
    console.log("この行は実行されます");
}
// finally 節のみでは例外がキャッチされないため、この行は実行されません
```

## 21.2 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。

次のコードでは、`catch` 節の `error` 識別子でキャッチしたエラーオブジェクトを参照しています。

```
try {
    // 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"
}
```

## 21.3 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。ここでは、`throw` 文で例外として投げられるエラーオブジェクトについて見ていきます。

### 21.3.1 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、エラーメッセージとなる文字列を渡します。渡したエラーメッセージは `Error#message` プロパ

ティに格納されます。

次のコードでは、`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 (`error`) からエラーオブジェクトを取得でき、エラーメッセージが確認できます。

```
// 渡された数値が 0 以上ではない場合に例外を投げる関数
function assertPositiveNumber(num) {
  if (num < 0) {
    throw new Error(`${num} is not positive.`);
  }
}

try {
  // 0 未満の値を渡しているので、関数が例外を投げる
  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 `Error` オブジェクトのインスタンスを投げることを推奨します。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持っています。文字列のような `Error` オブジェクトではないオブジェクトを投げしまうと、スタックトレースが得られません。

そのため、次のように `throw` 文で `Error` オブジェクトではないものを投げるのは非推奨です。

```
// 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) {
  // catch 節の例外識別子は、投げられた値を参照する
  console.log(error); // => "例外が投げられました"
}
```

### 21.3.2 ビルトインエラー

エラーには状況に合わせたいくつかの種類があり、これらはビルトインエラーとして定義されています。ビルトインエラーとは、ECMAScript 仕様や実行環境に組み込みで定義されているエラーオブジェクトです。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

## 第 21 章 例外処理

トを継承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。

ビルトインエラーに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。次のコードでは、存在しない変数を参照しているため **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); // エラーメッセージが表示される
}
```

**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。基本的に **SyntaxError** 例外は、JavaScript を実行する前のパース段階で発生します。そのため、実行前に発生する例外である **SyntaxError** を `try...catch` 文では `catch` できません。

```
// JavaScript として正しくない構文をパースすると SyntaxError が発生する
foo! bar!
```

次のコードでは、`eval` 関数を使って実行時に **SyntaxError** を発生させています。`eval` 関数は渡した文字列を JavaScript として実行する関数です。実行時に発生した **SyntaxError** は、`try...catch` 文でも `catch` できます。

```
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); // エラーメッセージが表示される
}
```

## TypeError

**TypeError** は値が期待される型ではない場合のエラーです。次のコードでは、関数ではないオブジェクトを関数呼び出ししているため、**TypeError** 例外が投げられます。

```
try {
  // 関数では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); // エラーメッセージが表示される
}
```

### 21.3.3 ビルトインエラーを投げる

ビルトインエラーのインスタンスを作成し、そのインスタンスを例外として投げることもできます。通常の **Error** オブジェクトと同じように、それぞれのビルトインエラーオブジェクトを **new** してインスタンスを作成できます。

たとえば関数の引数を文字列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ても、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 Array.from(str).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"
}
```

21.4 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。エラーが持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を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、エラーの名前を表す **name** プロパティと内容を表す **message** プロパティを持っています。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...catch** 文で囲っていない部分で例外が発生しています。

```
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。

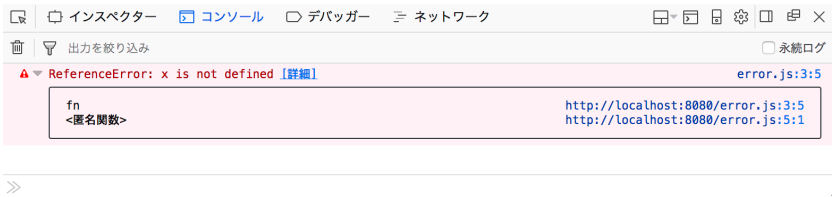


図 21.1 コンソールでのエラー表示 (Firefox)

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                                                             |
|----------------------------------|----------------------------------------------------------------|
| ReferenceError: x is not defined | エラーの種類は <b>ReferenceError</b> で、 <b>x</b> が未定義であること。           |
| error.js:3:5                     | 例外が <b>error.js</b> の 3 行目 5 列目で発生したこと。つまり <b>x++</b> ; 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。スタックトレースとは、プログラムの実行過程を記録した内容で、どの処理によってエラーが発生したかが書かれています。

- 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** 関数の呼び出しです

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たど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](#)<sup>\*1</sup>には、ブラウザが投げるビルトインエラーの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。

## 21.5 console.error とスタックトレース

console.error メソッドではメッセージと合わせてスタックトレースをコンソールへ出力できます。次のコードを実行して、console.log と console.error の出力結果を見比べてみます。

```
function fn() {  
  console.log("メッセージ");  
  console.error("エラーメッセージ");  
}  
  
fn();
```

このコードを Firefox で実行するとコンソール出力は次の図のようになります。

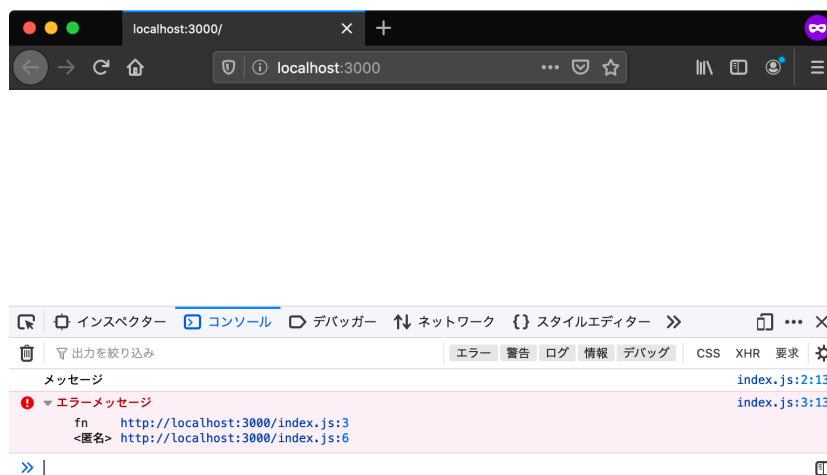


図 21.2 console.log と console.error の出力結果

console.log はメッセージだけなのに対して、console.error ではメッセージと共にスタックトレースが出力されます。そのため、エラーが発生した場合のコンソールへのメッセージ出力に console.error を利用することでデバッグがしやすくなります。

また、ほとんどのブラウザには console.log や console.error の出力をフィルターリングでき

<sup>\*1</sup> <https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Errors>

## 第 21 章 例外処理

る機能が備わっています。ただのログ出力には `console.log` を使い、エラーに関するログ出力には `console.error` と使うことで、ログの重要度が区別しやすくなります。

## 21.6 まとめ

この章では、例外処理とエラーオブジェクトについて学びました。

- `try...catch` 構文は `try` ブロック内で発生した例外を処理できる
- `catch` 節と `finally` 節は、両方またはどちらか片方を記述する
- `throw` 文は例外を投げることができ、`Error` オブジェクトを例外として投げる
- `Error` オブジェクトには、ECMAScript 仕様や実行環境で定義されたビルトインエラーがある
- `Error` オブジェクトには、スタックトレースが記録され、デバッグに役立てられる



## 第22章

# 非同期処理: コールバック /Promise/Async Function

この章では JavaScript の非同期処理について学んでいきます。非同期処理は JavaScript におけるとても重要な概念です。また、JavaScript を扱うブラウザや Node.js などの API には非同期処理のものも多いため、非同期処理を避けることはできません。そのため、非同期処理を扱うためのエラーファーストコールバックや Promise というビルトインオブジェクト、さらには Async Function と呼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。

### 22.1 同期処理

多くのプログラミング言語に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、次の処理へ進む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実行し、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、指定時間が経過するまで次の処理（次の行）は呼ばれません。

```
// 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) {
```

## 第 22 章 非同期処理: コールバック /Promise/Async Function

```
        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");
```

同期的にブロックする処理があると、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。メインスレッドは表示の更新といった UI に関する処理も行っています。そのため、メインスレッドが他の処理で専有されると、表示が更新されなくなりフリーズしたようになります。

先ほどの例では 1 秒間も処理をブロックしているため、1 秒間スクロールなどの操作が効かないといった悪影響がでます。

## 22.2 非同期処理

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が複数あります。

JavaScript において非同期処理の代表的な関数として `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` 関数を使って 10 ミリ秒後に、1 秒間ブロックする処理を実行しています。`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");
```

## 22.3 非同期処理はメインスレッドで実行される

```
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。

## 22.3 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同期的にブロックする処理の影響を受けないはずですが、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) {
```

## 第 22 章 非同期処理: コールバック /Promise/Async Function

```
        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、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っ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ます。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 を実行できます。この Web Worker における非同期処理は**並列処理** (Parallel) です。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 ではメインスレッドとは異なる Worker スレッドで実行されるため、メインスレッドは Worker スレッドの同期的に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` という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けます。

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。

## 22.4 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がキャッチできます（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("この行は実行されます");
```

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できません。

```
try {
  setTimeout(() => {
    throw new Error("非同期的なエラー");
  }, 10);
} catch (error) {
  // 非同期エラーはキャッチできないため、この行は実行されません
}
console.log("この行は実行されます");
```

`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て例外を投げるのは、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");
  }
});
```

## 第 22 章 非同期処理: コールバック /Promise/Async Function

```
    }  
  }, 10);  
  console.log("この行は実行されます");
```

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わかりません**。非同期処理の外から例外が起きたことを知るためには、非同期処理の中で例外が発生したこと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が重要です。

## 22.5 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仕様はありませんでした。このため、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の関数があります。指定したパスのファイルを読むため、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` 関数の第二引数に渡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ました  
  }  
});
```

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 でも利用されています。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」の章にて紹介します。

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、疑似的なリソースの取得をする非同期な処理です。第一引数に任意のパスを受け取り、第二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取ります。

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、第二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、第二引数のコールバック関数に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疑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 からはじ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) {
    // この行は実行されません
  } 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 {
    // この行は実行されません
  }
});
```

## 第 22 章 非同期処理: コールバック /Promise/Async Function

```
    }
  });
```

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ほかにもあります。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。先ほどの `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**へ伝える方法に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**共通のルール**の 1 つです。ルールを決めることのメリットとして、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**ただのルール**であって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、問題があるわけではありません。

しかしながら、最初書いたように JavaScript では非同期処理を扱うケースが多いため、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。

## 22.6 Promise ES2015

**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は次のようなルールでした。

- 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す
- 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

```
// 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 **asyncPromiseTask** 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コールバック関数として **then** メソッドへ渡し、失敗したときの処理を同じくコールバック関数として **catch** メソッドへ渡します。

エラーファーストコールバックとは異なり、非同期処理 (**asyncPromiseTask** 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。

```
// asyncPromiseTask 関数は Promise インスタンスを返す
asyncPromiseTask().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 (**asyncPromiseTask** 関数) はコールバック関数を受け取るのではなく、**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うま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言えるでしょう。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** を見ていきます。

### 22.6.1 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` と呼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` 関数を呼び出します。

```
const executor = (resolve, reject) => {  
  // 非同期の処理が成功したときは resolve を呼ぶ  
  // 非同期の処理が失敗したときは 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` メソッドの第一引数には `resolve` (成功) 時に呼ばれるコールバック関数、第二引数には `reject` (失敗) 時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  
  // 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) したとき
  - `onFulfilled` が呼ばれる
- `reject` (失敗) したとき
  - `onRejected` が呼ばれる

### 22.6.2 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疑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、成功として Resolved 状態の Promise オブジェクトを返す
 * 指定した path にデータがない場合、失敗として Rejected 状態の Promise オブジェクトを返す
 */
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) {
  // この行は実行されません
});

// /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) {
  // この行は実行されません
}, function onRejected(error) {
  console.log(error); // Error: "NOT FOUND"
```

## 第 22 章 非同期処理: コールバック /Promise/Async Function

```
});
```

`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)` のように第一引数には `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。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ので、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"
});
```

### 22.6.3 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** 関数を呼び出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、これ以降のコンストラクタの処理は実行されません
  });
}

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** 構文を使わなくても、自動的に例外がキャッチされます。

### 22.6.4 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** (不変) と呼びます。

## 第 22 章 非同期処理: コールバック /Promise/Async Function

一度でも **Settled (Fulfilled または Rejected)** となった **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();
    // すでに 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(); // 二度目以降の resolve や reject は無視される
  }, 16);
});
promise.then(() => {
  console.log("最初の resolve 時に一度だけ呼ばれる");
}, (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ときに、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。

### 22.6.5 `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。糖衣構文とは、同じ意味の処理を元の構文よりシンプルに書ける別の書き方のことです。`Promise.resolve` メソッドは次のコードの糖衣構文です。

```
// const fulfilledPromise = Promise.resolve(); と同じ意味
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、先ほどの `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();
});
```

## 第 22 章 非同期処理: コールバック /Promise/Async Function

```
});  
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的に呼ばれることがわかります。

### 22.6.6 `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、常に非同期的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` は短く書けるため、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` の状態を操作するのに利用できます。

### 22.6.7 `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ー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` メソッドの戻り値である **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](#)」の章の「[メソッドチェーンと高階関数](#)」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** な状態）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。そのため、次のコードでは、1、2 と順番にコンソールへログが出力されます。

```
// Promise インスタンスでメソッドチェーン
Promise.resolve()
  .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 thirdPromise = 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 === thirdPromise); // => 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` メソッドの戻り値は新しい **Promise** インスタンスであるため、続けて `catch` メソッド

## 第 22 章 非同期処理: コールバック /Promise/Async Function

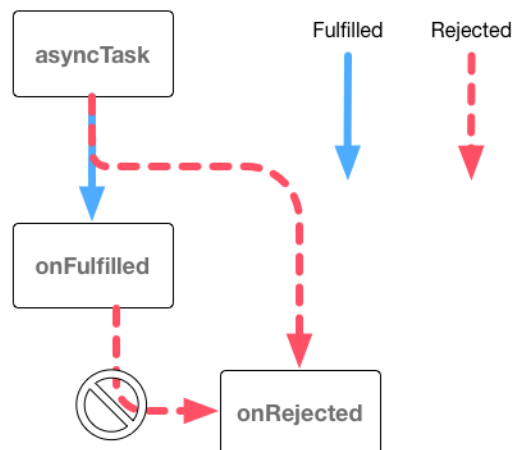
で失敗時の処理を書けます。

```
// 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 メソッドは新しい Promise インスタンスを返す
  .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` メソッドなどで一度キャッチすると、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元の `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。

## 第 22 章 非同期処理: コールバック /Promise/Async Function

**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。

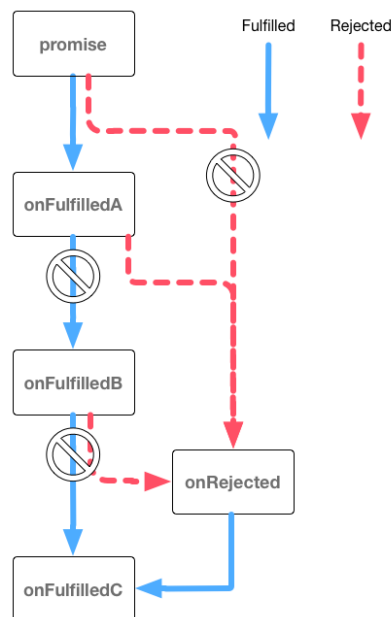
コールバック関数で Promise インスタンスを返した場合は、同じ状態を持つ Promise インスタンス

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です。

次のコードでは、`then` メソッドのコールバック関数で `Promise.reject` メソッドを使って **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

## 第 22 章 非同期処理: コールバック /Promise/Async Function

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 main() {
  return Promise.reject(new Error("エラー"));
}
// main は Rejected な Promise を返す
main().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);
}).then(() => {
  // 前の catch で Rejected な Promise が返されたため、この行は実行されません
}).catch(error => {
  console.log("メインの処理が失敗した");
});
```

**Promise チェーンの最後に処理を書く ES2018**

`Promise#finally` メソッドは成功時、失敗時どちらの場合でも呼び出されるコールバック関数を登録できます。`try...catch...finally` 構文の `finally` 節と同様の役割を持つメソッドです。

```
// promise には Resolved または Rejected な Promise インスタンスがランダムで入る
const promise = Math.random() < 0.5 ? Promise.resolve() : Promise.reject();
promise.then(() => {
  console.log("Promise#then");
}).catch((error) => {
  console.log("Promise#catch");
}).finally(() => {
  // 成功、失敗どちらの場合でも呼び出される
  console.log("Promise#finally");
});
```

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` で失敗時の処理を登録しています。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。`then` と `catch` の両方で `isLoading` へ `false` を代入できますが、`Promise#finally` メソッドを使うことで代入を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.error(error);
}).finally(() => {
    isLoading = false;
    console.log("Promise#finally");
});
```

### 22.6.8 Promise チェーンで逐次処理

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逐次処理を考えていきましょう。Promise で逐次的な処理とい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());
    });
}
```

## 第 22 章 非同期処理: コールバック /Promise/Async Function

```
    });  
  }  
  
  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"]  
  });
```

## 22.6.9 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]  
});
```

先ほどの 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** メソッドを使って複数の 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 の結果を Destructuring で responseA, responseB に代入している  
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")) {
```

## 第 22 章 非同期処理: コールバック /Promise/Async Function

```
        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]) => {
    // この行は実行されません
}).catch(error => {
    console.error(error); // Error: NOT FOUND
});
```

### 22.6.10 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、配列の中で一番最初に **Settle** 状態となった Promise インスタンスと同じ状態になります。

- 配列の中で一番最初に **Settle** となった Promise が **Fulfilled** の場合は、新しい Promise インスタンスも **Fulfilled** になる
- 配列の中で一番最初に **Settle** となった Promise が **Rejected** の場合は、新しい Promise インスタンスも **Rejected** になる

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);
    });
  }
  // 1 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 ミリ秒後
    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。ここでのタイムアウトとは、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です。

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** 関数では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 reject する
function timeout(timeoutMs) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      reject(new Error(`Timeout: ${timeoutMs}ミリ秒経過`));
    }, timeoutMs);
  });
}
```

## 第 22 章 非同期処理: コールバック /Promise/Async Function

```
    }, 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](https://azu.github.io/promises-book/)」<sup>\*1</sup>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するには 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。

---

\*1 <https://azu.github.io/promises-book/>

## 22.7 Async Function ES2017

ES2017 では、Async Function という非同期処理を行う関数を定義する構文が導入されました。Async Function は通常関数とは異なり、必ず **Promise** インスタンスを返す関数を定義する構文です。

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。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 の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書けます。

このセクションでは Async Function と **await** 式について見ていきます。

## 22.8 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。

## 第 22 章 非同期処理: コールバック /Promise/Async Function

```
// 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 fn3 = async() => {};
// メソッドの短縮記法の Async Function 版
const obj = { async method() {} };
```

これらの Async Function は、次の点以外は通常の関数と同じ性質を持ちます。

- Async Function は必ず **Promise** インスタンスを返す
- Async Function 内では **await** 式が利用できる

## 22.9 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 が値を return した場合、その返り値を持つ **Fulfilled** な Promise を返す
2. Async Function が Promise を return した場合、その返り値の Promise をそのまま返す
3. Async Function 内で例外が発生した場合は、そのエラーを持つ **Rejected** な Promise を返す

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はほぼ同じです。

```
// 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
async 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");
    // 例外が発生したため、この行は実行されません
}

// 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。

## 22.10 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て完了するまで、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() {
```

## 第 22 章 非同期処理: コールバック /Promise/Async Function

```
// doAsync の非同期処理が完了するまで待つ
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の完了後に実行される");
}
```

`await` 式は、`await` の右辺 (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」の章を参照)。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` 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` 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 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` 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` します。そのエラーを自動的にキャッチするため `asyncMain` 関数は Rejected な Promise を返します。

```
async function asyncMain() {
  // await 式で評価した右辺の Promise が Rejected となったため、例外が throw される
  const value = await Promise.reject(new Error("エラーメッセージ"));
  // await 式で例外が発生したため、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「例外処理」の章の「`try...catch` 構文」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` メソッドを使って 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。そのため、`asyncMain` 関数は Resolved な Promise を返し、`catch` メソッドのコールバック関数は呼び出されません。

```
async function asyncMain() {
  // await 式のエラーは try...catch できる
  try {
    // await 式で評価した右辺の Promise が Rejected となったため、例外が throw
    // される
    const value = await Promise.reject(new Error("エラーメッセージ"));
    // await 式で例外が発生したため、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 は Resolved な Promise を返す
asyncMain().catch(error => {
  // すでに try...catch されているため、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わかりやすい形になるのは大きな利点です。

### 22.10.1 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目で書けます。まずは、Promise チェーンで複数の非同期処理を逐次的に行うケースを見ていきます。その後に、同様の処理を Async Function と await 式で書き直して比較してみます。

次のコードの `fetchAB` 関数はリソース A とリソース B を順番に取得する処理を Promise チェーンで書いています。

## 第 22 章 非同期処理: コールバック /Promise/Async Function

```
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 fetchAB() {
  const results = [];
  return dummyFetch("/resource/A").then(response => {
    results.push(response.body);
    return dummyFetch("/resource/B");
  }).then(response => {
    results.push(response.body);
    return results;
  });
}

// リソースを取得して出力する
fetchAB().then((results) => {
  console.log(results);
  // => ["Response body of /resource/A", "Response body of /resource/B"]
});
```

同様の処理を Async Function と `await` 式で書くと次のように書けます。`await` 式を使ってリソースが取得できるまで待ち、その結果を変数 `results` に追加していくという形で逐次処理が実装できます。

```
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());
  });
}
```

```
    }  
    }, 1000 * Math.random());  
  });  
}  
// リソース A とリソース B を順番に取得する  
async function fetchAB() {  
  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  
fetchAB().then((results) => {  
  console.log(results);  
  // => ["Response body of /resource/A", "Response body of /resource/B"]  
});
```

Promise チェーンで `fetchAB` 関数書いた場合は、コールバックの中で処理するためややこしい見た目になりがちです。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。

## 22.11 Async Function と組み合わせ

これまでで基本的な Async Function の動きを見てきましたが、他の構文や Promise API と組み合わせた Async Function の使い方を見ていきましょう。

### 22.11.1 Async Function と反復処理

複数の非同期処理を行う際に、Async Function は for ループなどの反復処理と組み合わせることが可能です。

次のコードでは、指定したリソースのパスの配列を渡してそれらを順番に取得する `fetchResource` 関数を実装しています。Async Function 内で for 文を使った反復処理を行い、for ループの中で `await` 文を使ってリソースの取得を待ち、その結果を追加しています。

```
function dummyFetch(path) {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      if (path.startsWith("/resource")) {  
        resolve({ body: `Response body of ${path}` });  
      }  
    }, 1000 * Math.random());  
  });  
}
```

## 第 22 章 非同期処理: コールバック /Promise/Async Function

```
        } else {
            reject(new Error("NOT FOUND"));
        }
    }, 1000 * Math.random());
});
}
// 複数のリソースを順番に取得する
async function fetchResources(resources) {
    const results = [];
    for (let i = 0; i < resources.length; i++) {
        const resource = resources[i];
        // ループ内で非同期処理の完了を待っている
        const response = await dummyFetch(resource);
        results.push(response.body);
    }
    // 反復処理がすべて終わったら結果を返す
    // (返り値となる Promise を results で resolve する)
    return results;
}
// 取得したいリソースのパス配列
const resources = [
    "/resource/A",
    "/resource/B"
];
// リソースを取得して出力する
fetchResources(resources).then((results) => {
    console.log(results);
    // => ["Response body of /resource/A", "Response body of /resource/B"]
});
```

Async Function では、非同期処理であっても for ループのような既存の構文と組み合わせて利用することが簡単です。Promise のみの場合は、Promise チェーンでコールバック関数を使った反復処理を実装する必要があります。

### 22.11.2 Promise API と Async Function を組み合わせる

Async Function と `await` 式を使うことで、非同期処理を同期処理のような見方で書けます。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反復処理する場合に無駄な待ち時間を作ってしまうコードを書きやすくなります。

先ほどの `fetchResources` 関数ではリソースを順番に 1 つずつ取得していました。たとえば、リ

ソース A と B を取得しようとした場合にかかる時間は、リソース A と B の取得時間の合計となります。このとき、リソース A に 1 秒、リソース B に 2 秒かかるとした場合、すべてのリソースを取得するのに 3 秒かかります。

取得する順番に意味がない場合は、複数のリソースを同時に取得することで余計な待ち時間を解消できます。先ほどの例ならば、リソース A と B を同時に取得すれば、最大でもリソース B の取得にかかる 2 秒程度ですべてのリソースが取得できるはずです。

Promise チェーンでは `Promise.all` メソッドを使って、複数の非同期処理を 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` 式も `Promise.all` メソッドと組み合わせて利用できます。

次のコードでは、`Promise.all` メソッドと Async Function を組み合わせて、同時にリソースを取得する `fetchAllResources` 関数を実装しています。`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());
  });
}

// 複数のリソースをまとめて取得する
async function fetchAllResources(resources) {
  // リソースを同時に取得する
  const promises = resources.map(function(resource) {
    return dummyFetch(resource);
  });
  // すべてのリソースが取得できるまで待つ
  // Promise.all は [ResponseA, ResponseB] のように結果が配列となる
  const responses = await Promise.all(promises);
  // 取得した結果からレスポンスのボディだけを取り出す
  return responses.map((response) => {
    return response.body;
  });
}
```

## 第 22 章 非同期処理: コールバック /Promise/Async Function

```
const resources = [
  "/resource/A",
  "/resource/B"
];
// リソースを取得して出力する
fetchAllResources(resources).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 API 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になります。

### 22.11.3 `await` 式は Async Function の中でのみ利用可能

`await` 式を利用する際には、`await` 式は Async Function の中でのみ利用可能な点に注意が必要です。

次のコードのように、Async Function ではない通常の関数で `await` 式を使うと構文エラー (`SyntaxError`) となります。これは、間違った `await` 式の使い方を防止するための仕様です。

```
// async ではない関数では await 式は利用できない
function main(){
  // SyntaxError: await is only valid in async functions
  await Promise.resolve();
}
```

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どおり処理が進みます。次のコードを実行してみると、Async Function 内で `await` しても、Async Function 外の処理は停止していないことがわかります。

```
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 の外側の処理はそのまま進む  
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` 式で Async Function 内の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、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 の外で利用できない理由の 1 つです。

この仕様は、Async Function をコールバック関数内で利用しようとしたときに混乱を生む場合があります。具体例として、先ほどの逐次的にリソースを取得する `fetchResources` 関数を見えます。

先ほどの `fetchResources` 関数では `for` ループと `await` 式を利用していました。このときに `for` ループの代わりに `Array#forEach` メソッドは利用できません。

単純に `fetchResources` 関数の `for` ループから `Array#forEach` メソッドに書き換えて見ると、構文エラー (`SyntaxError`) が発生してしまいます。これは `await` 式が Async Function の中でのみ利用できる構文であるためです。

```
async function fetchResources(resources) {  
  const results = [];  
  // Syntax Error となる例  
  resources.forEach(function(resources) {  
    const resource = resources[i];  
    // Async Function ではないスコープで await 式を利用しているため Syntax Error  
    // となる  
    const response = await dummyFetch(resource);  
    results.push(response.body);  
  });  
  return results;  
}
```

そのため、`Array#forEach` メソッドのコールバック関数も Async Function として定義しないと、コールバック関数では `await` 式が利用できません。

この構文エラーは `Array#forEach` メソッドのコールバック関数を Async Function にすることで解決できます。しかし、コールバック関数を Async Function にしただけでは、`fetchResources` 関数は常に空の配列で解決される Promise を返すという意図しない挙動となります。

```
function dummyFetch(path) {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      if (path.startsWith("/resource")) {  
        resolve({ body: `Response body of ${path}` });  
      } else {
```

## 第 22 章 非同期処理: コールバック /Promise/Async Function

```
        reject(new Error("NOT FOUND"));
    }
    }, 1000 * Math.random());
});
}
// リソースを順番に取得する
async function fetchResources(resources) {
    const results = [];
    // コールバック関数を Async Function に変更
    resources.forEach(async function(resource) {
        // await 式を利用できるようになった
        const response = await dummyFetch(resource);
        results.push(response.body);
    });
    return results;
}
const resources = ["/resource/A", "/resource/B"];
// リソースを取得して出力する
fetchResources(resources).then((results) => {
    // しかし、results は空になってしまう
    console.log(results); // => []
});
```

なぜこのようになるかを `fetchResources` 関数の動きから見てみましょう。

`forEach` メソッドのコールバック関数として Async Function を渡し、コールバック関数中で `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はコールバック関数 Async Function の中だけで、コールバック関数の外側では `fetchResources` 関数の処理が進んでいます。

次のように `fetchResources` 関数にコンソールログを入れてみると動作がわ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、`fetchResources` 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、`forEach` メソッドのコールバック関数でリソースの取得が完了する前に、`fetchResources` 関数はその時点の `results` である空の配列で解決してしまいます。

```
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());
  });
}
// リソースを順番に取得する
async function fetchResources(resources) {
  const results = [];
  console.log("1. fetchResources を開始");
  resources.forEach(async function(resource) {
    console.log(`2. ${resource}の取得開始`);
    const response = await dummyFetch(resource);
    // dummyFetch が完了するのは、fetchResources 関数が返した Promise が
    // 解決された後
    console.log(`5. ${resource}の取得完了`);
    results.push(response.body);
  });
  console.log("3. fetchResources を終了");
  return results;
}
const resources = ["/resource/A", "/resource/B"];
// リソースを取得して出力する
fetchResources(resources).then((results) => {
  console.log("4. fetchResources の結果を取得");
  console.log(results); // => []
});
```

このように、Async Function とコールバック関数を組み合わせる場合には気をつける必要があります。

この問題を解決するため、最初の `fetchResources` 関数のように、コールバック関数を使わずにすむ `for` ループと `await` 式を組み合わせる方法があります。また、`fetchAllResources` 関数のように、複数の非同期処理を 1 つの Promise にまとめることでループ中に `await` 式を使わないようにする方法があります。

## 22.12 まとめ

この章では、非同期処理に関するコールバック関数、Promise、Async Function について学びました。

- 非同期処理はその処理が終わるのを待つ前に次の処理を評価すること
- 非同期処理であってもメインスレッドで実行されることがある
- エラーファーストコールバックは、非同期処理での例外を扱うルールの一つ

## 第 22 章 非同期処理: コールバック /Promise/Async Function

- Promise は、ES2015 で導入された非同期処理を扱うビルトインオブジェクト
- Async Function は、ES2017 で導入された非同期処理を扱う構文
- Async Function は Promise の上に作られた構文であるため、Promise と組み合わせて利用する

Promise や Async Function の応用パターンについては「[JavaScript Promise の本](#)」<sup>\*2</sup>も参照してください。

---

<sup>\*2</sup> <https://azu.github.io/promises-book/>

## 第23章

### Map/Set<sup>ES2015</sup>

# Chapter 23

JavaScript でデータの集まりを扱うコレクションは配列ではありません。この章では、ES2015 で導入された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。

## 23.1 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、キーと値の組み合わ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。

### 23.1.1 マップの作成と初期化

**Map** オブジェクトを **new** することで、新しいマップを作れます。作成されたばかりのマップは何も持っていない。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せます。コンストラクタ引数として渡せるのはエンタリーの配列です。エンタリーとは、1つのキーと値の組み合わせを【キー，値】という形式の配列で表現したものです。

次のコードでは、**Map** に初期値となるエンタリー（配列）の配列を渡しています。

```
const map = new Map([["key1", "value1"], ["key2", "value2"]]);
// 2つのエンタリーで初期化されている
console.log(map.size); // => 2
```

### 23.1.2 要素の追加と取り出し

**Map** には新しい要素を **set** メソッドで追加でき、追加した要素を **get** メソッドで取り出せます。

## 第 23 章 Map/Set

`set` メソッドは特定のキーと値を持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ひもづいた値を取り出します。また、特定のキーにひもづいた値を持っているかを確認する `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` メソッドはマップから要素を削除します。`delete` メソッドに渡されたキーと、そのキーにひもづいた値がマップから削除されます。また、マップが持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
```

### 23.1.3 マップの反復処理

マップが持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持つすべての要素を、マップへの挿入順に反復処理します。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。配列はインデックス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持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持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` 文で反復処理を行ったり、`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 (Iterator) を反復処理する
for (const key of map.keys()) {
  keys.push(key);
}
console.log(keys); // => ["key1","key2"]
// keys メソッドの戻り値 (Iterator) から配列を作成する
const keysArray = Array.from(map.keys());
console.log(keysArray); // => ["key1","key2"]
```

`entries` メソッドはマップが持つすべての要素をエンタリーとして挿入順に並べた **Iterator** オブジェクトを返します。先述のとおりエンタリーは [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タリーからキーと値を簡単に取り出せます。

```
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` 文で反復処理できます。マップを `for...of` 文で反復したときは、すべての要素をエンタリーとして挿入順に反復処理します。つまり、`entries` メソッドの戻り値を反復処理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"]
```

## 第 23 章 Map/Set

## 23.1.4 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 `prototype` オブジェクト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る
- また、プロパティとしてデータを持つため、キーとして使えるのは文字列か `Symbol` に限られる

`Object` には `prototype` オブジェクトがあるため、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となります（詳細は「[オブジェクト](#)」の章の「[プロパティの存在を確認する](#)」を参照）。

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。

```
const map = {};  
// マップがキーを持つことを確認する  
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
```

このマップとして使うオブジェクトの問題は、`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によって回避されてきました（詳細は「[プロトタイプオブジェクト](#)」の章の「[Object.prototype を継承しないオブジェクト](#)」を参照）。

ES2015 では、これらの問題を根本的に解決する `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` のプロトタイプが持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` ではマップのキーとしてあらゆるオブジェクトを使えます。

ほかにも `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る
- マップが持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() {
        // 商品とその数を持つマップ
        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: "リンゴ", 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, リンゴ:1"
```

## 第 23 章 Map/Set

`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` のマップを作ってフォームの入力内容を持たせています。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);
}
```

### 23.1.5 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）で持つことです。

**弱い参照**とは、ガベージコレクション（GC）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。GC によりメモリから解放できるオブジェクトは、どこからも参照されていないものだけです。このときオブジェクトへの弱い参照があったとしてもそのオブジェクトは解放されます。



そのため、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してしまう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ひもづ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。

次のコードでは、最初に **obj** には **{}** を設定し、**WeakMap** ではその **obj** をキーにして値 ("value") を設定しています。次に **obj** に別の値（ここでは **null**）を代入すると、**obj** が元々参照していた **{}** という値はどこからも参照されなくなります。このとき **WeakMap** は **{}** への弱い参照を持っていますが、弱い参照は GC を妨げないため、**{}** は不要になった値として GC によりメモリから解放されます。

同時に、**WeakMap** は解放されたオブジェクト (**{}**) をキーにしてひもづいていた値 ("value") を破棄できます。ただし、どのタイミングで実際にメモリから解放するかは、JavaScript エンジンの実装に依存します。

```
const map = new WeakMap();
// キーとなるオブジェクト
let obj = {};
// {} への参照をキーに値をセットする
map.set(obj, "value");
// {} への参照を破棄する
obj = null;
// GC が発生するタイミングで WeakMap から値が破棄される
```

**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、キーを弱い参照で持つ特性上、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、クラスにプライベートの値を格納することです。**this**（クラスインスタンス）を **WeakMap** のキーにすることで、インスタンスの外からはアクセスできない値を保持できます。また、クラスインスタンスが参照されなくなったときには自動的に解放されます。

次のコードでは、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 **WeakMap** 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、明示的に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** メソッドに渡された **listener** は **EventEmitter** インスタンスが参照されなくなった際、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();

class EventEmitter {
  addListener(listener) {
    // this にひもづいたリスナーの配列を取得する
    const listeners = listenersMap.get(this) || [];
    const newListeners = listeners.concat(listener);
    // this をキーに新しい配列をセットする
```

## 第 23 章 Map/Set

```
        listenersMap.set(this, newListeners);
    }
}

// 上記クラスの実行例

let eventEmitter = new EventEmitter();
// イベントリスナーを追加する
eventEmitter.addListener(() => {
    console.log("イベントが発火しました");
});
// eventEmitter への参照がなくなったことで自動的にイベントリスナーが解放される
eventEmitter = null;
```

また、あるオブジェクトから計算した結果を一時的に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ひもづけて保存している
    cache.set(element, height);
    return height;
}
```

## 23.2 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言います。[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持たず、インデックスによるアクセスはできません。

## 23.3 セットの作成と初期化

[Set](#) オブジェクトを `new` することで、新しいセットを作れます。作成されたばかりのセットは何も持っていません。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。

#### キーの等価性と NaN

**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。このときのマップが特定のキーをすでに持っているか、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、キーが **NaN** 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](https://developer.mozilla.org/ja/docs/Web/JavaScript/Equality_comparisons_and_when_to_use_them#Same-value-zero_equality) アルゴリズム<sup>a</sup>と呼ばれます。次のコードでは、**NaN** 同士の`===`の比較結果が `false` になるのに対して、**Map** のキーでは **NaN** 同士の比較結果が一致していることがわかります。

```
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.has(NaN)); // => true
console.log(map.get(NaN)); // => "value"
```

<sup>a</sup> [https://developer.mozilla.org/ja/docs/Web/JavaScript/Equality\\_comparisons\\_and\\_when\\_to\\_use\\_them#Same-value-zero\\_equality](https://developer.mozilla.org/ja/docs/Web/JavaScript/Equality_comparisons_and_when_to_use_them#Same-value-zero_equality)

```
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せます。コンストラクタ引数として渡せるのは `iterable` オブジェクトです。

次のコードでは `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。また、**Set** では重複する同じ値を持たないことを保証するため、同じ値は 1 つのみ格納されます。

```
// "value2"が重複するため、片方は無視される
const set = new Set(["value1", "value2", "value2"]);
// セットのサイズは 2 になる
console.log(set.size); // => 2
```

### 23.3.1 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持たないことが保証されます。そのため、すでにセットが持っている値を `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持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();
```

## 第 23 章 Map/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、セットが持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
```

### 23.3.2 セットの反復処理

セットが持つ値を反復処理するには、**forEach** メソッドが利用できます。**forEach** メソッドではセットが持つすべての要素を、セットへの挿入順に反復します。

```
const set = new Set(["a", "b"]);
const results = [];
set.forEach((value) => {
  results.push(value);
});
console.log(results); // => ["a","b"]
```

セットから Iterator オブジェクトを作成するメソッドとして **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がありません。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、セットが持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]** という形のエンタ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有用なケースは少ないでしょう。

```
const set = new Set(["a", "b"]);
// keys で列挙
const keysResults = [];
for (const value of set.keys()) {
  keysResults.push(value);
}
console.log(keysResults); // => ["a","b"]
// entries で列挙
const entryResults = [];
for (const entry of set.entries()) {
  // entry は [値, 値] という配列
  entryResults.push(entry);
}
console.log(entryResults); // => [["a","a"], ["b", "b"]]
```

Set オブジェクト自身も iterable なオブジェクトであるため `for...of` 文で反復処理できます。`for...of` 文で Set オブジェクトを反復処理したときも、セットへの挿入順に値が取り出されます。

```
const set = new Set(["a", "b"]);
const results = [];
for (const value of set) {
  results.push(value);
}
console.log(results); // => ["a","b"]
```

### 23.3.3 WeakSet

[WeakSet](#) は弱い参照で値を持つセットです。`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処理できません。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言えるでしょう。

また、弱い参照で値を持つ特性上、`WeakSet` の値として使えるのは参照型のオブジェクトだけです。

## 23.4 まとめ

この章では Map と Set について学びました。

- Map はキーと値の組み合わせからなるコレクションを扱うビルトインオブジェクト
- Map のキーはプロトタイプオブジェクトのプロパティと名前が衝突しないため意図しないマッピングを避けられる
- WeakMap はキーを弱い参照で持つ Map と同様のビルトインオブジェクト

## 第 23 章 Map/Set

- **Set** は重複する値がないことを保証した順序を持たないコレクションを扱うビルトインオブジェクト
- **WeakSet** は値を弱い参照で持つ **Set** と同様のビルトインオブジェクト

## 第24章

## JSON

# Chapter 24

この章では、JavaScript と密接な関係にある JSON というデータフォーマットについて見ていきます。

### 24.1 JSON とは

JSON は JavaScript Object Notation の略で、JavaScript のオブジェクトリテラル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<sup>\\*1</sup>](http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf) として標準化されています。JSON は、人間にとっ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える形式になっています。そのため、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ォートで囲まなければいけません。また、小数点から書きはじめる数値リテラルや、先頭がゼロからはじ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](http://www.json.org/json-ja.html) の日本語ドキュメント<sup>\*2</sup>にわかりやすくまとまっているので、参考にとするとよいでしょう。

<sup>\*1</sup> <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>

<sup>\*2</sup> <http://www.json.org/json-ja.html>

## 第 24 章 JSON

## 24.2 JSON オブジェクト

JavaScript で JSON を扱うには、ビルトインオブジェクトである **JSON オブジェクト** を利用します。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。

### 24.2.1 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 はダブルクォートのみを許容するため、シングルクォートで 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` 構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");
}
```

### 24.2.2 オブジェクトを JSON 文字列に変換する

**JSON.stringify** メソッドは第一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ンな引数が2つあります。第二引数は 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"}
```

第三引数は 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" };
```

第 24 章 JSON

```
console.log(JSON.stringify(obj, null, "\t"));
/*
{
  "id": 1,
  "name": "js-primer"
}
*/
```

24.3 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               |
| null       | null                |
| 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 null） |
| undefined  | 変換されない（配列のときは null） |
| Symbol     | 変換されない（配列のときは null） |
| RegExp     | {}                  |
| Map, Set   | {}                  |

```
// 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ライズされる際は、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ライズされます。`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` メソッドがシリアライズ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ライズ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たどって自分自身が見つかるような場合はシリアライズ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行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.error(error);
  // => "TypeError: Converting circular structure to JSON"
}
```

## 24.4 toJSON メソッドを使ったシリアライズ

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` メソッドの戻り値を使います。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ライズする目的などに使われます。

## 24.5 まとめ

この章では、JSON について学びました。

## 第 24 章 JSON

- JSON は JavaScript のオブジェクトリテラルをベースに作られた軽量なデータフォーマット
- JSON オブジェクトを使ったシリアライズとデシリアライズ
- JSON 形式にシリアライズできないオブジェクトもある
- `JSON.stringify` はシリアライズ対象の `toJSON` メソッドを利用する

## 第25章

### Date

# Chapter 25

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。

## 25.1 Date オブジェクト

`Date` オブジェクトは `String` や `Array` などと同じく、ECMAScript で定義されたビルトインオブジェクトです。

`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持ち、その時刻値を元に日付や時・分などを扱うメソッドを提供します。

### 25.1.1 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。1 つは現在の時刻をインスタンス化するもの、もう 1 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持つ時刻値は、`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<sup>\*1</sup>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。人間が見てもわ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
```

<sup>\*1</sup> [https://ja.wikipedia.org/wiki/ISO\\_8601](https://ja.wikipedia.org/wiki/ISO_8601)

## 第 25 章 Date

```
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渡すデータ型や引数によって時刻の指定方法が変わります。`Date` は次の 3 種類を引数としてサポートしています。

- 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、他の 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<sup>\\*2</sup>](https://tools.ietf.org/html/rfc2822#section-3.3)や [ISO 8601](#) の形式に従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"
```

---

<sup>\*2</sup> <https://tools.ietf.org/html/rfc2822#section-3.3>

```
// 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。日を表す第三引数から後ろ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、そのほかには0が設定されます。また、月を表す第二引数は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当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持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"
```

### 25.1.2 Date のインスタンスメソッド

Date オブジェクトのインスタンスは多くのメソッドを持っていますが、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 = 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 = String(date.getMonth() + 1).padStart(2, "0");
  const dd = 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 はインスタンスメソッドなので、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}`);
```

## 25.2 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メ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](https://momentjs.com/)<sup>\*3</sup>や [js-joda](https://github.com/js-joda/js-joda)<sup>\*4</sup>、[date-fns](https://date-fns.org/)<sup>\*5</sup>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"));
```

## 25.3 まとめ

この章では、`Date` オブジェクトについて学びました。

- `Date` オブジェクトのインスタンスはある特定の時刻を表すビルトインオブジェクト
- `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ている
- `Date` コンストラクタで任意の時間を表す `Date` インスタンスを作成できる
- `Date` インスタンスメソッドにはさまざまなものがあるが、現実のユースケースでは機能が不十分になりやすい
- ビルトインオブジェクトの `Date` のみではなく、ライブラリも合わせて利用するのが一般的

---

\*3 <https://momentjs.com/>

\*4 <https://github.com/js-joda/js-joda>

\*5 <https://date-fns.org/>

## 第 26 章

### Math

# Chapter 26

この章では、JavaScript で数学的な定数と関数を提供するビルトインオブジェクトである [Math](#) について学びます。

## 26.1 Math オブジェクト

`Math` オブジェクトはビルトイン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  $\pi$  を表す定数であり、`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](#)<sup>\*1</sup>を参照してください。

### 26.1.1 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。

```
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());
}
```

<sup>\*1</sup> [https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global\\_Objects/Math](https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/Math)

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));
```

### 26.1.2 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
```

### 26.1.3 数値を整数にする

`Math` オブジェクトには数値を整数に丸めるためのメソッドがいくつかあります。代表的なものは、小数点以下を切り捨てる `Math.floor` メソッド、小数点以下を切り上げる `Math.ceil` メソッド、そして四捨五入を行う `Math.round` メソッドです。

`Math.floor` メソッドは、引数として渡した数以下で最大の整数を返します。このような関数は底関数と呼ばれます。正の数である `1.3` は `1` になりますが、負の数である `-1.3` はより小さい整数の `-2` に丸められます。

次の `Math.ceil` メソッドは、引数として渡した数以上で最小の整数を返します。このような関数は天井関数と呼ばれます。正の数である `1.3` は `2` になりますが、負の数である `-1.3` はより大きい整数の `-1` に丸められます。

`Math.round` メソッドは、一般的な四捨五入の処理を行います。小数部分が `0.5` よりも小さな場合は切り捨てられ、それ以外は切り上げられます。

```
// 底関数
console.log(Math.floor(1.3)); // => 1
console.log(Math.floor(-1.3)); // => -2
```

## 第 26 章 Math

```
// 天井関数
console.log(Math.ceil(1.3)); // => 2
console.log(Math.ceil(-1.3)); // => -1
// 四捨五入
console.log(Math.round(1.3)); // => 1
console.log(Math.round(1.6)); // => 2
console.log(Math.round(-1.3)); // => -1
```

また、`Math.trunc` メソッド **ES2015** は、渡された数字の小数点以下を単純に切り落とした整数を返します。そのため、引数が正の値の場合は `Math.floor` メソッドと同じになり、そうでない場合は `Math.ceil` メソッドと同じになります。

```
// 単純に小数部分を切り落とす
console.log(Math.trunc(1.3)); // => 1
console.log(Math.trunc(-1.3)); // => -1
```

## 26.2 まとめ

この章では、`Math` オブジェクトについて学びました。紹介したメソッドは `Math` オブジェクトの一部にすぎないため、そのほかにもメソッドが用意されています。

- `Math` は数学的な定数や関数を提供するビルトインオブジェクト
- `Math` はコンストラクタではないためインスタンス化できない
- 疑似乱数の生成、数値の比較、数値の計算などを行うメソッドが提供されている

## 第27章

# ECMAScript モジュール ES2015

# Chapter 27

ECMAScript モジュールは「ユースケース: Todo アプリ」の章で実際に動かしながら学ぶため、ここでは構文の説明とモジュールのイメージをつかむのが目的です。この章のサンプルコードを実際に動かすためにはローカルサーバーなどの準備が必要です。そのため、ユースケースの章を先に読んでから戻ってきてもかまいません。

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せます
- 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アンドペーストせず、モジュールとして再利用できます

1つの JavaScript モジュールは1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、ES2015 で導入された JavaScript ファイルをモジュール化する言語標準の機能です。

## 27.1 ECMAScript モジュールの構文

ECMAScript 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。インポートとエクスポートはそれぞれに **名前つき** と **デフォルト** という2種類の方法があります。

まずは名前つきエクスポート／インポート文について見ていきましょう。

### 27.1.1 名前つきエクスポート／インポート

**名前つきエクスポート**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。次の例では、`foo` 変数と `bar` 関数をそれぞれ名前つきエクスポートしています。**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ることで、宣言済みの変数を名前つきエクスポートできます。

## 第 27 章 ECMAScript モジュール

named-export.js

```
const foo = "foo";  
// 宣言済みのオブジェクトを名前つきエクスポートする  
export { foo };
```

また、名前つきエクスポートでは **export** 文を宣言の前につけると、宣言と同時に名前つきエクスポートできます。

named-export-declare.js

```
// 宣言と同時に名前つきエクスポートする  
export function bar() { };
```

**名前つきインポート**は、指定したモジュールから名前を指定して選択的にインポートできます。次の例では `my-module.js` から名前つきエクスポートされたオブジェクトの名前を指定して名前つきインポートしています。**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つきエクスポートの名前を入れます。複数の値をインポートしたい場合は、それぞれの名前をカンマで区切ります。

my-module.js

```
export const foo = "foo";  
export function bar() { }
```

named-import.js

```
// 名前つ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 "./my-module.js";  
console.log(foo); // => "foo"  
console.log(bar); // => "bar"
```

**名前つきエクスポート／インポートのエイリアス**

名前つきエクスポート／インポートには**エイリアス**の仕組みがあります。エイリアスを使うと、宣言済みの変数を違う名前で名前つきエクスポートできます。エイリアスをつけるには、次のように **as** のあとにエクスポートしたい名前を記述します。

```
named-export-alias.js
```

```
const internalFoo = "foo";  
// internalFoo 変数を foo として名前つきエクスポートする  
export { internalFoo as foo };
```

また、名前つきインポートしたオブジェクトにも別名をつけることができます。インポートでも同様に、`as` のあとに別名を記述します。

```
named-import-alias.js
```

```
// foo として名前つきエクスポートされた変数を myFoo としてインポートする  
import { foo as myFoo } from "./named-export-alias.js";  
console.log(myFoo); // => "foo"
```

### 27.1.2 デフォルトエクスポート／インポート

次に、デフォルトエクスポート／インポートについて見ていきましょう。デフォルトエクスポートは、モジュールごとに1つしかエクスポートできない特殊なエクスポートです。次の例は、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ています。`export default` 文で、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。

```
default-export.js
```

```
const foo = "foo";  
// foo 変数の値をデフォルトエクスポートする  
export default foo;
```

また、`export` 文を宣言の前につけると、宣言と同時にデフォルトエクスポートできます。このとき関数やクラスの名前を省略できます。

```
// 宣言と同時に関数をデフォルトエクスポートする  
export default function() {}
```

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行うことはできません。なぜなら、変数宣言はカンマ区切りで複数の変数を定義できてしまうためです。次の例は実行できない不正なコードです。

```
// 変数宣言と同時にデフォルトエクスポートはできない
```

## 第 27 章 ECMAScript モジュール

```
export default const foo = "foo", bar = "bar";
```

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に名前をつけてインポートします。次の例では `my-module.js` のデフォルトエクスポートに `myModule` という名前をつけてインポートしています。`import` 文のあとに任意の名前をつけることで、デフォルトエクスポートをインポートできます。

```
my-module.js
```

```
export default {  
  baz: "baz"  
};
```

```
default-import.js
```

```
// デフォルトエクスポートをmyModuleとしてインポートする  
import myModule from "./my-module.js";  
console.log(myModule); // => { baz: "baz" }
```

実はデフォルトエクスポートは、`default` という固有の名前による名前つきエクスポートと同じものです。そのため、名前つきエクスポートで `as default` とエイリアスをつけることでデフォルトエクスポートすることもできます。

```
default-export-alias.js
```

```
const foo = "foo";  
// foo 変数の値をデフォルトエクスポートする  
export { foo as default };
```

同様に、名前つきインポートにおいても `default` という名前がデフォルトインポートに対応しています。次のように、名前つきインポートで `default` を指定するとデフォルト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。

```
default-import-alias.js
```

```
// デフォルトエクスポートをmyModuleとしてインポートする  
import { default as myModule } from "./my-module.js";  
console.log(myModule); // => { baz: "baz" }
```



また、名前つきインポートとデフォルトインポートの構文は同時に記述できます。次のように2つの構文をカンマでつなげます。

```
default-import-with-named.js
```

```
// myModule としてデフォルトインポートし、  
// foo を名前つきインポートする  
import myModule, { foo } from "./my-module.js";  
console.log(foo); // => "foo"  
console.log(myModule); // => { baz: "baz" }
```

ECMAScript 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 ECMAScript 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、インポート先の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。同様に名前つきインポートは、インポート先のモジュールが指定した名前つきエクスポートをしている必要があります。

### 27.1.3 その他の構文

ECMAScript モジュールには名前つきとデフォルト以外にもいくつかの構文があります。

#### 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インポートしたものを、改めて自分自身からエクスポートし直すことです。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。

```
// ./my-module.js のすべての名前つきエクスポートを再エクスポートする  
export * from "./my-module.js";  
// ./my-module.js の名前つきエクスポートを選んで再エクスポートする  
export { foo, bar } from "./my-module.js";  
// ./my-module.js の名前つ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 "./my-module.js";  
// ./my-module.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 "./my-module.js";  
// ./my-module.js のデフォルトエクスポートを名前つきエクスポートとして再エクスポートする  
export { default as myModuleDefault } from "./my-module.js";  
// ./my-module.js の名前つ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
```

## 第 27 章 ECMAScript モジュール

```
export { foo as default } from "./my-module.js";
```

## すべてをインポート

`import * as` 構文は、すべての名前つ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。また、先ほどのとおり、`default` という固有名を使うとデフォルトエクスポートにもアクセスできます。

```
my-module.js
```

```
export const foo = "foo";
export function bar() { }
export default {
  baz: "baz"
};
```

```
namespace-import.js
```

```
// すべての名前つきエクスポートを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 "./my-module.js";
// foo として名前つきエクスポートされた値にアクセスする
console.log(myModule.foo); // => "foo"
// default としてデフォルトエクスポートされた値にアクセスする
console.log(myModule.default); // => { baz: "baz" }
```

## 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、グローバル変数进行操作するためのモジュールなどです。

```
side-effects.js
```

```
// グローバル変数进行操作する（副作用）
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、指定したモジュールを読み込んで実行するだけで、何もインポートしません。

```
// ./side-effects.js のグローバルコードが実行される
import "./side-effects.js";
```

## 27.2 ECMAScript モジュールを実行する

作成した ECMAScript 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 ECMAScript モジュールとしてウェブブラウザに読み込ませる必要があります。ウェブブラウザは `script` 要素によって JavaScript ファイルを読み込み、実行します。次のように `script` 要素に `type="module"` 属性を付与すると、ウェブブラウザは JavaScript ファイルを ECMAScript モジュールとして読み込みます。

```
<!-- my-module.js を ECMAScript モジュールとして読み込む -->
<script type="module" src="./my-module.js"></script>
<!-- インラインでも同じ -->
<script type="module">
import { foo } from "./my-module.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。

ウェブブラウザの環境では、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「ユースケース: [Todo アプリ](#)」を参照してください。

# 第 28 章

## ECMAScript

# Chapter 28

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を見ていきましょう。

ECMAScript は [Ecma International](#)<sup>\*1</sup> 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しています。Ecma International の中の 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。

### 28.1 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 <sup>*2</sup> |
| 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       |
| 以下毎年リリース |                  |

ES5.1 から ES2015 がでるまで 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 以降に仕様策定プロセスの変更が行われたためです。

<sup>\*1</sup> <http://www.ecma-international.org/>

<sup>\*2</sup> ECMAScript 4 は複雑で大きな変更が含まれており、合意を得ることができずに仕様が破棄されました。

28.2 Living Standard となる ECMAScript

現在、ECMAScript の仕様書のドラフトは GitHub 上の [tc39/ecma262](#)<sup>\*3</sup>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つけずに、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。このバージョンつき 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つきよりも Living Standard のほうが適当であるためです。

28.3 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長い時間がかかり、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形へと 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 のリリース後に適用され、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まっています<sup>\*4</sup>。

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。

<sup>\*3</sup> <https://github.com/tc39/ecma262>  
<sup>\*4</sup> 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。

## 第 28 章 ECMAScript

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が、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になってしまったという経緯があります<sup>\*5</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>\*6</sup>。別の代替プロポーザルが出た場合や後方互換性を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やすいという性質もあります。

## 28.4 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つ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きます。

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` 構文が導入されましたが、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` キーワードを使って疑似的に再現するコードへ変換します。Transpiler としては [Babel](https://babeljs.io/)<sup>\*7</sup>や [TypeScript](https://www.typescriptlang.org/)<sup>\*8</sup>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](https://core-js.org/)<sup>\*9</sup>や [polyfill.io](https://polyfill.io/)<sup>\*10</sup>などが有名です。

注意点としては Transpiler や Polyfill は、あくまで既存の機能を用いて新しい機能の再現を試みているだけにすぎません。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。

<sup>\*5</sup> ES2015 の仕様編集者である Allen Wirfs-Brock 氏の書いた [Programming Language Standardization](http://wirfs-brock.com/allen/files/papers/standpats-asianplop2016.pdf) に詳細が書かれています (<http://wirfs-brock.com/allen/files/papers/standpats-asianplop2016.pdf>)。

<sup>\*6</sup> [Inactive Proposals](https://github.com/tc39/proposals/blob/master/inactive-proposals.md) 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 (<https://github.com/tc39/proposals/blob/master/inactive-proposals.md>)。

<sup>\*7</sup> <https://babeljs.io/>

<sup>\*8</sup> <https://www.typescriptlang.org/>

<sup>\*9</sup> <https://github.com/zloirock/core-js>

<sup>\*10</sup> <https://polyfill.io/v2/docs/>

## 28.5 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に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

### 28.5.1 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 の言語機能に関しては [MDN Web Docs](https://developer.mozilla.org/ja/)<sup>\*11</sup> 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。

### 28.5.2 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いよいでしょう。

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を知ることが大切です。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持つことは大切です。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。

### 28.5.3 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 の仕様に入るものは、後方互換性を維持するために破壊的変更はほとんど行えません。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わかります。

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することが重要です。

---

<sup>\*11</sup> <https://developer.mozilla.org/ja/>

## 第 28 章 ECMAScript

これは ECMAScript に限らず、ウェブやブラウザに関する情報については同じことが言えます。ブラウザに関しては HTML、DOM API、CSS などのオープンな仕様とそれぞれの策定プロセスが存在しています。

### 28.5.4 まとめ

JavaScript と一言に言っても ECMAScript、ウェブブラウザ、Node.js、WebAssembly、WebGL、WebRTC など幅広い分野があります。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。

何ごととも突然まった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持つことが大切です。



## 第 1 部のおわりに

第 1 部の基本文法では、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 2 部のユースケースでは、第 1 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 2 部からは Node.js やブラウザの実行環境にある固有の 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。

第 1 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。

そのほかにもありますが、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際に利用することが多いです。また第 2 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 は必要となった際に使い方を調べて覚えていくのがよいでしょう。

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](#)<sup>\*12</sup> 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 2 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](#)<sup>\*13</sup> があります。ブラウザなら先ほども紹介した [MDN Web Docs](#) が実質的な公式のリファレンスガイドです。まずは使い方を知らずとも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見るのが重要です。

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に、調べ方を知っておくことが重要です。

<sup>\*12</sup> <https://developer.mozilla.org/ja/>

<sup>\*13</sup> <https://nodejs.org/api/>

## 第 2 部 ユースケース

---

*Part2*

第 1 部の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。

## 目次

### アプリケーション開発の準備

アプリケーション開発のために Node.js と npm のインストールなどの準備方法を紹介します。

### ユースケース: Ajax 通信

ウェブブラウザ上で Ajax 通信をする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ながら、非同期処理について紹介します。

### ユースケース: Node.js で CLI アプリケーション

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する例として、Markdown を HTML に変換するツールを作成していきます。また、Node.js や npm の使い方を紹介します。

### ユースケース: Todo アプリケーション

ブラウザで動作するウェブアプリケーションの例として Todo アプリを作成しながら、モジュールを使ったコード管理について紹介します。

## 第29章

### アプリケーション開発の準備

# Chapter 29

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ウェブブラウザと [Node.js](#)<sup>\*1</sup> 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行います。

#### 29.1 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。

- ウェブブラウザの 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](#)<sup>\*2</sup>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、迷わずに 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 12.13.0 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

<sup>\*1</sup> <https://nodejs.org/ja/>

<sup>\*2</sup> <https://nodejs.org/ja/download/>

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（\$はコマンドラインの入力欄を表す記号であるため、実際に入力する必要はありません）。

```
$ node -v  
v12.13.0
```

また、Node.js には [npm](#)<sup>\*3</sup> 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` を扱うための `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  
6.12.0
```

`npm` や `npm` コマンドについての詳細は[公式ドキュメント](#)<sup>\*4</sup>や [npm の GitHub リポジトリ](#)<sup>\*5</sup>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。

## 29.2 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をまとめてできます。この後のユースケースでも `npx` コマンドでツールを利用するため、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。

---

\*3 <https://www.npmjs.com/>

\*4 <https://docs.npmjs.com/>

\*5 <https://github.com/npm/cli>

## 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!
```

## 29.3 ローカルサーバーのセットアップ

「値の評価と表示」の章では、`index.html` と `index.js` というファイルを作成してブラウザで表示していました。このときローカルに作成した HTML ファイルをそのままブラウザで読み込むと、ブラウザのアドレスバーは `file:///`からはじまる URL になります。`file` スキーマでは [Same Origin Policy](#)<sup>\*6</sup>というセキュリティ的な制限により、多くの場面でアプリケーションは正しく動作しません。

これからユースケースの章で書いていくアプリケーションは、[Same Origin Policy](#) の制限を避けるために、`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。開発用のローカルサーバーを使うことで、ローカルに作成した HTML ファイルも `http` スキーマの URL で表示できます。

ここでは、これからのユースケースで利用する開発用のローカルサーバーをセットアップする方法を見ていきます。

### 29.3.1 HTML ファイルの用意

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。ここでは `index.html` というファイル名で作成し、HTML ファイル内には次のように記述しています。この HTML ファイルでは `script` 要素を使って `index.js` というファイル名の JavaScript ファイルを読み込んでいます。

index.html

```
<html lang="ja">
  <head>
    <meta charset="utf-8" />
```

<sup>\*6</sup> [https://developer.mozilla.org/ja/docs/Web/Security/Same-origin\\_policy](https://developer.mozilla.org/ja/docs/Web/Security/Same-origin_policy)

```
<title>index.html</title>
</head>
<body>
  <h1>ローカルサーバで配信中</h1>
  <script src="index.js"></script>
</body>
</html>
```

同じように `index.js` というファイル名で JavaScript ファイルを作成します。この `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。

```
index.js

console.log("index.js: loaded");
```

### 29.3.2 ローカルサーバーを起動する

先ほど作成した `index.html` と同じディレクトリで、ローカルサーバーを起動します。次のコマンドでは、`@js-primer/local-server` というこの書籍用に作成されたローカルサーバーモジュールをダウンロードと同時に実行します。このローカルサーバーモジュールは、`http` スキーマの URL でローカルファイルへアクセスできるように、実行したディレクトリにあるファイルを配信する機能を持ちます。

```
# からはじまる行はコメントなので実行はしなくてよい
# cd コマンドでファイルがあるディレクトリまで移動
$ cd "index.html があるディレクトリのパス"
```

```
# npx コマンドでローカルサーバーを起動
$ npx @js-primer/local-server
```

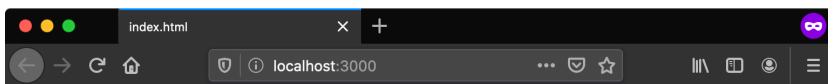
js-primer のローカルサーバーを起動しました。  
次の URL をブラウザで開いてください。

URL: `http://localhost:3000`

起動したローカルサーバーの URL (`http://localhost:3000`) へブラウザでアクセスすると、先ほどの `index.html` の内容が表示されます。多くのサーバーでは、`http://localhost:3000`

## 第 29 章 アプリケーション開発の準備

のようにファイルパスを指定せずにアクセスすると、`index.html` を配信する機能を持っています。`@js-primer/local-server` もこの機能を持つため、`http://localhost:3000` と `http://localhost:3000/index.html` のどちらの URL も同じ `index.html` を配信しています。



## ローカルサーバで配信中

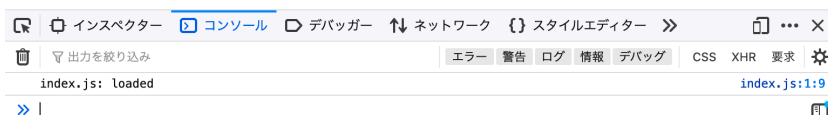


図 29.1 ログが表示されている Web コンソール

`index.html` にアクセスできたら、正しく `index.js` が読み込まれているかを確認してみましょう。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、この書籍では 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）のウェブ開発サブメニューで“ウェブコンソール”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“[ウェブコンソールを開く](#)”<sup>\*7</sup>を参照してください。

## 29.3.3 ローカルサーバを終了する

最後に、起動したローカルサーバを終了します。ローカルサーバを起動したコマンドラインで、`Ctrl+C`を押下することで終了できます。

複数のローカルサーバを同時に起動することも可能ですが、複数のサーバで同じポート番号を利用することはできません。ポートとは、先ほど起動したローカルサーバの URL で `:3000` となっている部分のことで、これは 3000 番ポートでローカルサーバを起動したことを意味しています。

`@js-primer/local-server` は、デフォルトのポート（3000 番ポート）がすでに使用されているな

<sup>\*7</sup> [https://developer.mozilla.org/ja/docs/Tools/Web\\_Console/Opening\\_the\\_Web\\_Console](https://developer.mozilla.org/ja/docs/Tools/Web_Console/Opening_the_Web_Console)



ら、使われていないポートを探してローカルサーバーを起動します。また、`--port` オプションで任意のポート番号でローカルサーバーを起動できます。

```
$ npx @js-primer/local-server --port 8000
```

この書籍では、`@js-primer/local-server` をデフォルトのポート番号である 3000 番ポートを利用する前提で進めていきます。使わなくなったローカルサーバーは `Ctrl`+`C` で終了しておくことで、アクセスする URL (ポート番号) が書籍と同じ状態で進められます。

## 29.4 まとめ

この章では、これからのユースケースの章で必要な環境を準備しました。

- Node.js の LTS 版をインストールした
- npm と npx でモジュールのインストールと実行をした
- `@js-primer/local-server` モジュールを使ってローカルサーバーを起動して終了した

npm では、すでに多種多様なローカルサーバーモジュールが公開されています。この書籍では、利用するローカルサーバーの機能で違いが出ないように `@js-primer/local-server` というこの書籍用のローカルサーバーモジュールを利用します。

## 第30章

### ユースケース: Ajax 通信

# Chapter 30

ここではウェブブラウザ上で Ajax 通信をする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

### 30.1 エントリーポイント

エントリーポイントとは、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、まずはエントリーポイントを用意しなければなりません。

ウェブ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ー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。

#### 30.1.1 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 JavaScript など複数のファイルが必要となります。そのため、まずそれらのファイルを置くためのディレクトリを作成します。

ここでは `ajaxapp` という名前で新しいディレクトリを作成します。ここからは作成した `ajaxapp` ディレクトリ以下で作業していきます。

またこのプロジェクトで作成するファイルは、必ず文字コード（エンコーディング）を **UTF-8**、改行コードを **LF** にしてファイルを保存します。

#### 30.1.2 HTML ファイルの用意

エントリーポイントとして、まずは最低限の要素だけを配置した HTML ファイルを `index.html` というファイル名で作成しましょう。`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。

```
index.html
```

```
<html lang="ja">
  <head>
    <meta charset="utf-8" />
    <title>Ajax Example</title>
  </head>
  <body>
    <h2>GitHub User Info</h2>
    <script src="index.js"></script>
  </body>
</html>
```

次に同じディレクトリに `index.js` というファイルを作成します。`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。

```
index.js
```

```
console.log("index.js: loaded");
```

`ajaxapp` ディレクトリのファイル配置が次のようになっていれば問題ありません。

```
ajaxapp
├── index.html
└── index.js
```

次はこの `index.html` をブラウザで表示して、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。

### 30.1.3 ローカルサーバーで HTML を確認する

ウェブブラウザで `index.html` を開く前に、開発用のローカルサーバーを準備します。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、その場合は `file:///`からはじまる URL になります。`file` スキーマでは [Same Origin Policy](#) のセキュリティ制限により、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。

コマンドラインで `ajaxapp` ディレクトリへ移動し、次のコマンドでローカルサーバーを起動します。次のコマンドでは、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ーモジュールをダウンロードと同時に実行します。まだ `npm` コマンドの用意ができていなければ、先に「[アプリケーション開発の準備](#)」を参照してください。

## 第 30 章 ユースケース: Ajax 通信

```
$ npx @js-primer/local-server
```

起動したローカルサーバーの URL (<http://localhost:3000>) へブラウザでアクセスすると、`"index.js: loaded"`とコンソールにログが出力されます。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）のウェブ開発サブメニューで“ウェブコンソール”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“[ウェブコンソールを開く](#)”<sup>\*1</sup>を参照してください。



図 30.1 ログが表示されているウェブコンソール

### 30.1.4 ウェブブラウザと DOM

HTML ドキュメントをブラウザで読み込むとき、[DOM](#) と呼ばれるプログラミング用のデータ表現が生成されます。**DOM (Document Object Model)** とは、HTML ドキュメントのコンテンツと構造を JavaScript から操作できるオブジェクトです。DOM では HTML ドキュメントのタグの入れ子関係を木構造で表現するため、DOM が表現する HTML タグの木構造を **DOM ツリー** と呼びます。

たとえば、DOM には HTML ドキュメントそのものを表現する `document` グローバルオブジェクトがあります。`document` グローバルオブジェクトには、指定した HTML 要素を取得したり、新しく

<sup>\*1</sup> [https://developer.mozilla.org/ja/docs/Tools/Web\\_Console/Opening\\_the\\_Web\\_Console](https://developer.mozilla.org/ja/docs/Tools/Web_Console/Opening_the_Web_Console)

HTML 要素を作成するメソッドが実装されています。`document` グローバルオブジェクトを使うことで、先ほどの `index.html` に書かれた HTML を JavaScript から操作できます。

```
// CSS セレクタを使って DOM ツリー中の h2 要素を取得する
const heading = document.querySelector("h2");
// h2 要素に含まれるテキストコンテンツを取得する
const headingText = heading.textContent;

// button 要素を作成する
const button = document.createElement("button");
button.textContent = "Push Me";
// body 要素の子要素として button を挿入する
document.body.appendChild(button);
```

JavaScript と DOM はウェブアプリケーション開発において切っても切り離せない関係です。動的なウェブアプリケーションを作るためには、JavaScript による DOM の操作が不可欠です。今回のユースケースでも GitHub の API から取得したデータを元に、動的に DOM ツリーを操作して画面の表示を更新します。

しかし、DOM は言語機能（ECMAScript）ではなくブラウザが実装している API です。そのため、DOM を持たない Node.js などの実行環境では使えず、`document` のようなグローバルオブジェクトも存在しないことには注意が必要です。

### 30.1.5 このセクションのチェックリスト

このセクションでは、エントリーポイントとなる HTML を作成し、JavaScript モジュールのエントリーポイントとなる JavaScript ファイルを読み込むところまでを実装しました。

- `ajaxapp` という名前のプロジェクトディレクトリを作成した
- エントリーポイントとなる `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
- ローカルサーバーを使ってブラウザで `index.html` を表示した
- `index.js` からコンソールに出力されたログを確認した
- JavaScript から HTML ドキュメントを操作する DOM について学んだ

## 30.2 HTTP 通信

ローカルサーバーで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。GitHub の API を呼び出すためには HTTP 通信をする必要があります。ウェブブラウザ上で JavaScript から HTTP 通信するために、[Fetch API](#) という機能を使います。

## 第 30 章 ユースケース: Ajax 通信

## 30.2.1 Fetch API

**Fetch API** は HTTP 通信を行ってリソースを取得するための API です。Fetch API を使うことで、ページ全体を再読み込みすることなく指定した URL からデータを取得できます。Fetch API は同じく HTTP 通信を扱う [XMLHttpRequest](#) と似た API ですが、より強力で柔軟な操作が可能です。

リクエストを送信するためには、`fetch` メソッドを利用します。`fetch` メソッドに URL を与えることで、HTTP リクエストが作成され、サーバーとの HTTP 通信を開始します。

GitHub にはユーザー情報を取得する API として、`https://api.github.com/users/GitHub ユーザー ID` という URL が用意されています。GitHub のユーザー ID には、英数字と - (ハイフン) 以外は利用できないため、ユーザー ID は `encodeURIComponent` 関数を使ってエスケープしたものを結合します。`encodeURIComponent` は / や % など URL として特殊な意味を持つ文字列をただの文字列として扱えるようにエスケープする関数です。

次のコードでは、指定した GitHub ユーザー ID の情報を取得する URL に対して `fetch` メソッドで、GET の HTTP リクエストを行っています。

```
const userId = "任意の GitHub ユーザー ID";
fetch(`https://api.github.com/users/${encodeURIComponent(userId)}`);
```

## 30.2.2 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。

`fetch` メソッドは `Promise` を返します。この `Promise` インスタンスはリクエストのレスポンスを表す `Response` オブジェクトで `resolve` されます。送信したリクエストにレスポンスが返却されると、`then` コールバックが呼び出されます。

次のように、`Response` オブジェクトの `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。また、`Response` オブジェクトの `json` メソッドも `Promise` を返します。これは、HTTP レスポンスボディを JSON としてパースしたオブジェクトで `resolve` されます。ここでは、書籍用に用意した `js-primer-example` という GitHub アカунトのユーザー情報を取得しています。

```
const userId = "js-primer-example";
fetch(`https://api.github.com/users/${encodeURIComponent(userId)}`)
  .then(response => {
    console.log(response.status); // => 200
    return response.json().then(userInfo => {
      // JSON パースされたオブジェクトが渡される
      console.log(userInfo); // => {...}
    });
  });
```

### 30.2.3 エラーハンドリング

HTTP 通信にはエラーがつきものです。そのため Fetch API を使った通信においても、エラーをハンドリングする必要があります。たとえば、サーバーとの通信に際してネットワークエラーが発生した場合は、ネットワークエラーを表す `NetworkError` オブジェクトで `reject` された `Promise` が返されます。すなわち、`then` メソッドの第二引数か `catch` メソッドのコールバック関数が呼び出されます。

```
const userId = "js-primer-example";
fetch(`https://api.github.com/users/${encodeURIComponent(userId)}`)
  .then(response => {
    console.log(response.status);
    return response.json().then(userInfo => {
      console.log(userInfo);
    });
  }).catch(error => {
    console.error(error);
  });
```

一方で、リクエストが成功したかどうかは `Response` オブジェクトの `ok` プロパティで認識できます。`ok` プロパティは、HTTP ステータスコードが 200 番台であれば `true` を返し、それ以外の 400 や 500 番台などなら `false` を返します。次のように、`ok` プロパティが `false` となるエラーレスポンスをハンドリングできます。

```
const userId = "js-primer-example";
fetch(`https://api.github.com/users/${encodeURIComponent(userId)}`)
  .then(response => {
    console.log(response.status);
    // エラーレスポンスが返されたことを検知する
    if (!response.ok) {
      console.error("エラーレスポンス", response);
    } else {
      return response.json().then(userInfo => {
        console.log(userInfo);
      });
    }
  }).catch(error => {
    console.error(error);
  });
```

ここまでの内容をまとめ、GitHub からユーザー情報を取得する関数を `fetchUserInfo` という名前

## 第 30 章 ユースケース: Ajax 通信

で定義します。

```
function fetchUserInfo(userId) {
  fetch(`https://api.github.com/users/${encodeURIComponent(userId)}`)
    .then(response => {
      console.log(response.status);
      // エラーレスポンスが返されたことを検知する
      if (!response.ok) {
        console.error("エラーレスポンス", response);
      } else {
        return response.json().then(userInfo => {
          console.log(userInfo);
        });
      }
    })
    .catch(error => {
      console.error(error);
    });
}
```

index.js では関数を定義しているだけで、呼び出しは行っていません。

ページを読み込むたびに GitHub の API を呼び出すと、呼び出し回数の制限を超えるおそれがあります。呼び出し回数の制限を超えると、API からのレスポンスがステータスコード 403 のエラーになってしまいます。

そのため、HTML ドキュメント側に手動で `fetchUserInfo` 関数を呼び出すためのボタンを追加します。ボタンの click イベントで `fetchUserInfo` 関数を呼び出し、取得したいユーザー ID を引数として与えています。例として `js-primer-example` という書籍用に用意した GitHub アカウントを指定しています。

```
<html lang="ja">
  <head>
    <meta charset="utf-8" />
    <title>Ajax Example</title>
  </head>
  <body>
    <h2>GitHub User Info</h2>

    <button onclick="fetchUserInfo('js-primer-example');">Get user info</button>
    <script src="index.js"></script>
  </body>
</html>
```



準備ができれば、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。

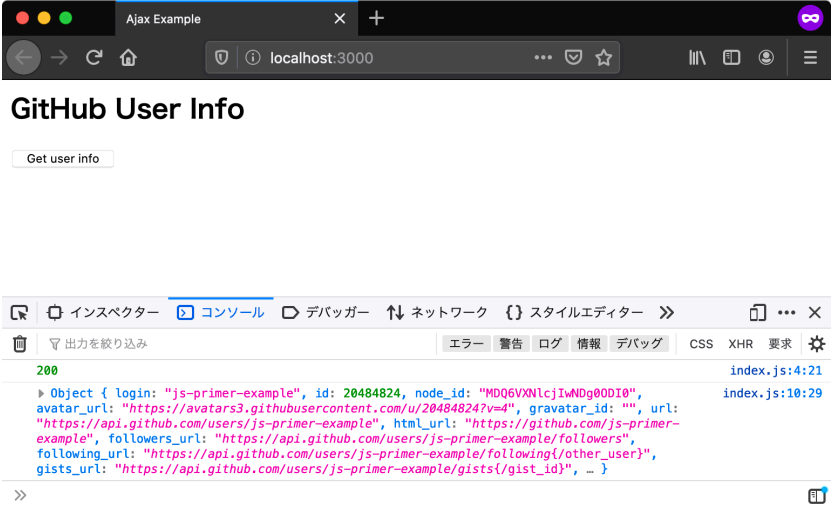


図 30.2 Fetch で取得したデータの表示

また、開発者ツールのネットワークパネルを開くと、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。



図 30.3 開発者ツールで HTTP 通信の記録を確認する

## XMLHttpRequest

[XMLHttpRequest \(XHR\)](#) は Fetch API と同じく HTTP 通信を行うための API です。Fetch API が標準化される以前は、ブラウザとサーバーの間で通信を行うには XHR を使うのが一般的でした。このセクションで扱った Fetch API による `fetchUserInfo` 関数は、XHR を使うと次のように書けます。

```
function fetchUserInfo(userId) {
  // リクエストを作成する
  const request = new XMLHttpRequest();
  request.open("GET",
    `https://api.github.com/users/${encodeURIComponent(userId)}`);
  request.addEventListener("load", () => {
    // リクエストが成功したかを判定する
    // Fetch API の response.ok と同等の意味
    if (request.status >= 200 && request.status < 300) {
      // レスポンス文字列を JSON オブジェクトにパースする
      const userInfo = JSON.parse(request.responseText);
      console.log(userInfo);
    } else {
      console.error("エラーレスポンス", request.statusText);
    }
  });
  request.addEventListener("error", () => {
    console.error("ネットワークエラー");
  });
  // リクエストを送信する
  request.send();
}
```

Fetch API は XHR を置き換えるために作られたもので、多くのユースケースでは XHR を使う必要はなくなっています。ただし、古いブラウザでは Fetch API が実装されていないため、ブラウザの互換性を保つために XHR を使う場面もまだあります。XHR の詳しい使い方については、[XHR の利用についてのドキュメント](#)<sup>a</sup>を参照してください。

<sup>a</sup> [https://developer.mozilla.org/ja/docs/Web/API/XMLHttpRequest/Using\\_XMLHttpRequest](https://developer.mozilla.org/ja/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest)

### 30.2.4 このセクションのチェックリスト

- [Fetch API](#) を使って HTTP リクエストを送った

- GitHub の API から取得したユーザー情報の JSON オブジェクトをコンソールに出力した
- Fetch API の呼び出しに対するエラーハンドリングを行った
- `fetchUserInfo` 関数を宣言し、ボタンのクリックイベントで呼び出した

## 30.3 データを表示する

前のセクションでは、Fetch API を使って GitHub の API からユーザー情報を取得しました。このセクションでは取得したデータを HTML に整形して、アプリケーションにユーザー情報を表示してみましょう。

### 30.3.1 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、変数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あ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>
</div>`;


```

このテンプレートに `userInfo` オブジェクトの値をあてはめると、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>
```

### 30.3.2 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。

index.html

```
<html lang="ja">
  <head>
    <meta charset="utf-8" />
    <title>Ajax Example</title>
  </head>
  <body>
    <h2>GitHub User Info</h2>

    <button onclick="fetchUserInfo('js-primer-example');">Get user info</button>
    <!-- 整形した HTML の挿入先 -->
    <div id="result"></div>

    <script src="index.js"></script>
  </body>
</html>
```

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 に追加する方法には、大きく分けて 2 つあります。1 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 1 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エスケープ処理を行います。

### 30.3.3 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、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](#)<sup>\*2</sup>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づけ](#)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。タグづ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` 関数はテンプレートリテラルにタグづけするためのタグ関数です。タグ関数には、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、値が文字列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, str, i) => {
    const value = values[i - 1];
    if (typeof value === "string") {
      return result + escapeSpecialChars(value) + str;
    } else {
      return result + String(value) + str;
    }
  });
}
```

<sup>\*2</sup> <https://github.com/teppeis/htmlspecialchars>

## 第 30 章 ユースケース: Ajax 通信

```
    });
  }

```

`escapeHTML` 関数はタグ関数なので、通常の `()` による呼び出しではなく、テンプレートリテラルに対してタグづけして使います。テンプレートリテラルのバッククォート記号の前に関数を書くと、関数がタグづけされます。

```
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。これらの処理を前のセクションで作成した `fetchUserInfo` 関数の中で呼び出します。ここまでで、`index.js` と `index.html` は次のようになっています。

index.js

```
function fetchUserInfo(userId) {
  fetch(`https://api.github.com/users/${encodeURIComponent(userId)}`)
    .then(response => {
      if (!response.ok) {
        console.error("エラーレスポンス", response);
      } else {
        return response.json().then(userInfo => {
          // HTML の組み立て
          const view = escapeHTML`
            <h4>${userInfo.name} (@${userInfo.login})</h4>
            
            <dl>
              <dt>Location</dt>

```

```
        <dd>${userInfo.location}</dd>
        <dt>Repositories</dt>
        <dd>${userInfo.public_repos}</dd>
    </dl>
    `;
    // HTML の挿入
    const result = document.getElementById("result");
    result.innerHTML = view;
    });
    }
    }).catch(error => {
        console.error(error);
    });
}

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, str, i) => {
        const value = values[i - 1];
        if (typeof value === "string") {
            return result + escapeSpecialChars(value) + str;
        } else {
            return result + String(value) + str;
        }
    });
}
```

---

## 第 30 章 ユースケース: Ajax 通信

index.html

```
<html lang="ja">
  <head>
    <meta charset="utf-8" />
    <title>Ajax Example</title>
  </head>
  <body>
    <h2>GitHub User Info</h2>

    <button onclick="fetchUserInfo('js-primer-example');">Get user info</button>
    <!-- 整形した HTML の挿入先 -->
    <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。

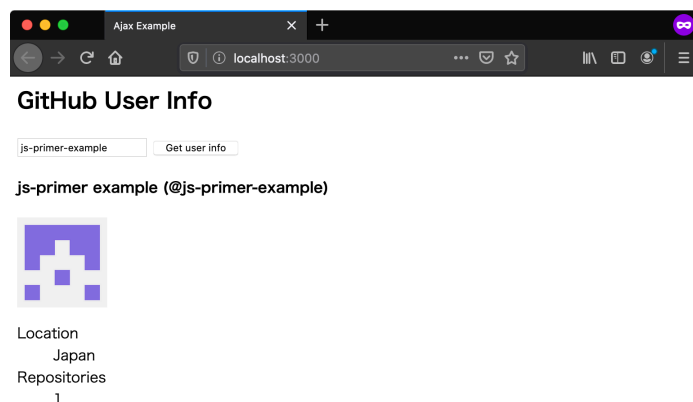


図 30.4 ユーザー情報の表示

## 30.3.4 このセクションのチェックリスト

- テンプレートリテラルを使って HTML 文字列を組み立てた
- innerHTML プロパティを使って HTML 文字列を DOM に追加した
- タグつきテンプレート関数を使って HTML 文字列をエスケープした



- `fetchUserInfo` 関数を呼び出し、HTML にユーザー情報が表示されることを確認した

## 30.4 Promise を活用する

ここまでのセクションで、Fetch API 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、Fetch API の戻り値でもある **Promise** を活用し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行います。

### 30.4.1 関数の分割

まずは、大きくなりすぎた `fetchUserInfo` 関数を整理しましょう。この関数では、Fetch API 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 の表示をしています。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ーポイントを設けます。`index.js` に新しく `main` 関数を作り、`main` 関数から `fetchUserInfo` 関数を呼び出すようにします。

```
function main() {
  fetchUserInfo("js-primer-example");
}

function fetchUserInfo(userId) {
  fetch(`https://api.github.com/users/${encodeURIComponent(userId)}`)
    .then(response => {
      if (!response.ok) {
        console.error("エラーレスポンス", response);
      } else {
        return response.json().then(userInfo => {
          // HTML の組み立て
          const view = createView(userInfo);
          // HTML の挿入
          displayView(view);
        });
      }
    })
    .catch(error => {
      console.error(error);
    });
}
```

## 第 30 章 ユースケース: Ajax 通信

```
function createView(userInfo) {
  return escapeHTML`
    <h4>${userInfo.name} (@${userInfo.login})</h4>
    
    <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;
}
```

ボタンの click イベントで呼び出す関数もこれまでの `fetchUserInfo` 関数から `main` 関数に変更します。

```
index.html
```

```
<html lang="ja">
  <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>
```

### 30.4.2 Promise のエラーハンドリング

次に `fetchUserInfo` 関数を変更し、Fetch API の戻り値でもある Promise オブジェクトを `return` します。この変更によって、`fetchUserInfo` 関数を呼び出す `main` 関数のほうで非同期処理の結果を扱えるようになります。Promise チェーンの中で投げられたエラーは、`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、`fetchUserInfo` 関数から返された Promise オブジェクトを、`main` 関数でエラーハンドリングしてログを出力します。`fetchUserInfo` 関数の `catch` メソッドでハンドリングしていたエラーは、`main` 関数の `catch` メソッドでハンドリングされます。一方、`Response#ok` で判定していた 400 や 500 などのエラーレスポンスがそのままでは `main` 関数でハンドリングできません。そこで、`Promise.reject` メソッドを使って Rejected な Promise を返し、Promise チェーンをエラーの状態にします。Promise チェーンがエラーとなるため、`main` 関数の `catch` でハンドリングできます。

```
function main() {
  fetchUserInfo("js-primer-example")
    .catch((error) => {
      // Promise チェーンの中で発生したエラーを受け取る
      console.error(`エラーが発生しました (${error})`);
    });
}

function fetchUserInfo(userId) {
  // fetch の戻り値の Promise を return する
  return fetch(`https://api.github.com/users/${encodeURIComponent(userId)}`)
    .then(response => {
      if (!response.ok) {
        // エラーレスポンスから Rejected な Promise を作成して返す
        return Promise.reject(new Error(`${response.status}:
          ${response.statusText}`));
      } else {
        return response.json().then(userInfo => {
          // HTML の組み立て
          const view = createView(userInfo);
          // HTML の挿入
          displayView(view);
        });
      }
    });
}
```

## 第 30 章 ユースケース: Ajax 通信

## Promise チェーンのリファクタリング

現在の `fetchUserInfo` 関数はデータの取得に加えて、HTML の組み立て (`createView`) と表示 (`displayView`) も行っています。`fetchUserInfo` 関数に処理が集中して見通しが悪いので、`fetchUserInfo` 関数はデータの取得だけを行うように変更します。併せて `main` 関数で、データの取得 (`fetchUserInfo`)、HTML の組み立て (`createView`) と表示 (`displayView`) という一連の流れを Promise チェーンで行うように変更していきます。

`Promise#then` メソッドでつながる Promise チェーンは、`then` に渡された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 Promise で解決された値を次の `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のは、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` を使ってつなぐことで、変更に強いコードを書けます。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。

`index.js` の `fetchUserInfo` 関数と `main` 関数を次のように書き換えます。まず、`fetchUserInfo` 関数が `Response#json` メソッドの戻り値をそのまま返すように変更します。`Response#json` メソッドの戻り値は JSON オブジェクトで解決される Promise なので、次の `then` ではユーザー情報の JSON オブジェクトが渡されます。次に、`main` 関数が `fetchUserInfo` 関数の Promise チェーンで、HTML の組み立て (`createView`) と表示 (`displayView`) を行うように変更します。

```
function main() {
  fetchUserInfo("js-primer-example")
    // ここでは JSON オブジェクトで解決される Promise
    .then((userInfo) => createView(userInfo))
    // ここでは HTML 文字列で解決される Promise
    .then((view) => displayView(view))
    // Promise チェーンでエラーがあった場合はキャッチされる
    .catch((error) => {
      console.error(`エラーが発生しました (${error})`);
    });
}

function fetchUserInfo(userId) {
  return fetch(`https://api.github.com/users/${encodeURIComponent(userId)}`)
    .then(response => {
      if (!response.ok) {
        return Promise.reject(new Error(`${response.status}:`);
      }
    });
}
```

```
        ${response.statusText}`));  
    } else {  
        // JSON オブジェクトで解決される Promise を返す  
        return response.json();  
    }  
});  
}
```

### Async Function への置き換え

Promise チェーンによって、Promise の非同期処理と同じ見目で同期処理を記述できるようになりました。さらに Async Function を使うと、同期処理と同じ見目で Promise の非同期処理を記述できるようになります。Promise の `then` メソッドによるコールバック関数の入れ子がなくなり、手続き的で可読性が高いコードになります。また、エラーハンドリングも同期処理と同じく `try...catch` 構文を使うことができます。

`main` 関数を次のように書き換えましょう。まず関数宣言の前に `async` をつけて Async Function にしています。次に `fetchUserInfo` 関数の呼び出しに `await` をつけます。これにより Promise に解決された JSON オブジェクトを `userInfo` 変数に代入できます。

もし `fetchUserInfo` 関数の中で例外が投げられた場合は、`try...catch` 構文でエラーハンドリングできます。このように、あらかじめ非同期処理の関数が Promise を返すようにしておくと、Async Function にリファクタリングしやすくなります。

```
async function main() {  
    try {  
        const userInfo = await fetchUserInfo("js-primer-example");  
        const view = createView(userInfo);  
        displayView(view);  
    } catch (error) {  
        console.error(`エラーが発生しました (${error})`);  
    }  
}
```

### 30.4.3 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` で固定としていたユーザー 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。

```
index.html
```

```
<html lang="ja">
```

## 第 30 章 ユースケース: Ajax 通信

```
<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なります。

```
index.js
```

```
async function main() {
  try {
    const userId = getUserId();
    const userInfo = await fetchUserInfo(userId);
    const view = createView(userInfo);
    displayView(view);
  } catch (error) {
    console.error(`エラーが発生しました (${error})`);
  }
}

function fetchUserInfo(userId) {
  return fetch(`https://api.github.com/users/${encodeURIComponent(userId)}`)
    .then(response => {
      if (!response.ok) {
        return Promise.reject(new Error(`${response.status}:
                                          ${response.statusText}`));
      } else {

```

```
        return response.json();
    }
    });
}

function getUserId() {
    const value = document.getElementById("userId").value;
    return encodeURIComponent(value);
}

function createView(userInfo) {
    return escapeHTML`
    <h4>${userInfo.name} (@${userInfo.login})</h4>
    
    <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, "&amp;")
        .replace(/</g, "&lt;")
        .replace(/>/g, "&gt;")
        .replace(/"/g, "&quot;")
        .replace(/'/g, "&#039;");
}

function escapeHTML(strings, ...values) {
```

## 第 30 章 ユースケース: Ajax 通信

```
return strings.reduce((result, str, i) => {
  const value = values[i - 1];
  if (typeof value === "string") {
    return result + escapeSpecialChars(value) + str;
  } else {
    return result + String(value) + str;
  }
});
}
```

アプリケーションを実行すると、次のようになります。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。



図 30.5 完成したアプリケーション

## 30.4.4 このセクションのチェックリスト

- HTML の組み立てと表示の処理を `createView` 関数と `displayView` 関数に分離した
- `main` 関数を宣言し、`fetchUserInfo` 関数が返す Promise のエラーハンドリングを行った
- Promise チェーンを使って `fetchUserInfo` 関数をリファクタリングした
- [Async Function](#) を使って `main` 関数をリファクタリングした
- `index.html` に `<input>` タグを追加し、`getUserId` 関数でユーザー ID を取得した



## 第31章

# ユースケース: Node.js で CLI アプリケーション

## Chapter 31

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

### 31.1 Node.js で Hello World

実際にアプリケーションを作成する前に、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。

#### 31.1.1 プロジェクトディレクトリの作成

今回作成する Node.js の CLI アプリケーションでは、JavaScript や Markdown などのファイルを扱います。そのため、まずそれらのファイルを置くためのディレクトリを作成します。

ここでは `nodecli` という名前で新しいディレクトリを作成します。ここからは作成した `nodecli` ディレクトリ以下で作業していきます。

またこのプロジェクトで作成するファイルは、必ず文字コード (エンコーディング) を **UTF-8**、改行コードを **LF** にしてファイルを保存します。

#### 31.1.2 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ーポイントとなる JavaScript ファイルです。`nodecli` 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。

## 第 31 章 ユースケース: Node.js で CLI アプリケーション

main.js

```
console.log("Hello World!");
```

ウェブブラウザの実行環境では、`console.log` メソッドの出力先はブラウザの開発者ツールのコンソールでした。Node.js 環境では、`console.log` メソッドの出力先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 では、エントリーポイントとなる 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のが基本です。また、ウェブブラウザの JavaScript と同じく、コードは 1 行目から順に実行されます。

### 31.1.3 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 という JavaScript エンジンを利用しています。そのため、ECMAScript で定義されている基本構文はブラウザと同じように使えます。しかし、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異なるため、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` や `String` などのラッパーオブジェクト、`Map` や `Promise` のようなビルトインオブジェクトはどちらの環境にも存在します。

しかし、実行環境によって異なるオブジェクトもあります。たとえばウェブ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、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、同じ名前でも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` としてもアクセスできます。

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持つプロパティ

や関数がブラウザと Node.js のどちらにもある場合があります。たとえば次のような API は同等の機能を提供しますが、メソッドの種類や返り値が異なります。

- 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。

#### 31.1.4 このセクションのチェックリスト

- `main.js` ファイルを作成した
- `node` コマンドで `main.js` を実行し、標準出力にログが出力されるのを確認した
- グローバルオブジェクトについて、ウェブブラウザと Node.js で実行環境による違いがあることを理解した

## 31.2 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。

### 31.2.1 `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ものは、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する API です。詳細は[公式ドキュメント](https://nodejs.org/dist/latest-v12.x/docs/api/process.html#process_process)<sup>\*1</sup>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。次のように `main.js` を変更し、`process.argv` をコンソールに出力しましょう。

```
main.js

// コンソールにコマンドライン引数を出力する
console.log(process.argv);
```

このスクリプトを次のようにコマンドライン引数をつけて実行してみましょう。

```
$ node main.js one two=three four
```

このコマンドの実行結果は次のようになります。

<sup>\*1</sup> [https://nodejs.org/dist/latest-v12.x/docs/api/process.html#process\\_process](https://nodejs.org/dist/latest-v12.x/docs/api/process.html#process_process)

## 第 31 章 ユースケース: Node.js で CLI アプリケーション

```
[
  '/usr/local/bin/node', // Node.js の実行プロセスのパス
  '/Users/laco/nodecli/main.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。

### 31.2.2 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、取得できる情報にはアプリケーションに不要なものも含まれています。また、文字列の配列として渡されるため、フラグのオンオフのような真偽値を受け取るときにも不便です。そのため、アプリケーションでコマンドライン引数を扱うときには、一度パースして扱いやすい値に整形するのが一般的です。

今回は `commander`<sup>\*2</sup> 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てみましょう。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。

#### commander パッケージをインストールする

`commander` は `npm` の `npm install` コマンドを使ってインストールできます。まだ `npm` の実行環境を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。

`npm` でパッケージをインストールする前に、まずは `package.json` というファイルを作成します。`package.json` とは、アプリケーションが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する JSON 形式のファイルです。`package.json` ファイルのひな形は、`npm init` コマンドで生成できます。通常は対話式のプロンプトによって情報を設定しますが、ここではすべてデフォルト値で `package.json` を作成する `--yes` オプションを付与します。

`nodecli` のディレクトリ内で、`npm init --yes` コマンドを実行して `package.json` を作成しましょう。

```
$ npm init --yes
```

生成された `package.json` ファイルは次のようになっています。

```
package.json
```

```
{
  "name": "nodecli",
```

<sup>\*2</sup> <https://github.com/tj/commander.js/>

```
"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。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\*3。

```
$ npm install 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なっています。

```
package.json
{
  "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"
  }
}
```

\*3 `--save` オプションをつけてインストールしたのと同じ意味。npm 5.0.0 からは `--save` がデフォルトオプションとなりました。

## 第 31 章 ユースケース: Node.js で CLI アプリケーション

```
}
```

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。先ほど `commander` のバージョンを 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、再び `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のを防ぎます。

### CommonJS モジュール

インストールした `commander` パッケージを使う前に、**CommonJS** モジュールのことを知っておきましょう。[CommonJS モジュール](#)<sup>\*4</sup>とは、[Node.js](#) 環境で利用されている JavaScript のモジュール化の仕組みです。CommonJS モジュールは基本文法で学んだ [ECMAScript モジュール](#) (ES module) の仕様が策定される前から Node.js で使われています。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。先ほどインストールした `commander` パッケージも、CommonJS モジュールとして利用できます。

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。CommonJS モジュールでは `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。複数の名前つきエクスポートが可能な ES Module とは異なり、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。

次の例では、`my-module.js` というファイルを作成し、`module.exports` でオブジェクトをエクスポートしています。

```
my-module.js

module.exports = {
  foo: "foo"
};
```

この CommonJS モジュールをインポートするには、Node.js 実行環境のグローバル関数である `require` 関数を使います。次のように `require` 関数にインポートしたいモジュールのファイルパスを渡し、戻り値としてエクスポートされた値をインポートできます。インポートするファイルパスに拡張子が必須な ES Module とは異なり、CommonJS の `require` 関数では拡張子である `.js` が省略可能です。

```
// my-module.js モジュールを myModule オブジェクトとしてインポートする。
const myModule = require("./my-module");
console.log(myModule.foo); // => "foo"
```

<sup>\*4</sup> <https://nodejs.org/docs/latest/api/modules.html>

また、`require` 関数には相対パスや絶対パス以外にも `npm` でインストールしたパッケージ名を指定できます。`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。`require` 関数にインストールしたパッケージ名を指定することで、`node_modules` ディレクトリに配置されたパッケージを読み込みます。

次の例では、先ほどインストールした `commander` パッケージを `node_modules` ディレクトリから読み込んでいます。

```
const program = require("commander");
```

このユースケースで今後登場するモジュールはすべて CommonJS モジュールです。Node.js では ES Module もサポートされる予定ですが、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。

#### コマンドライン引数からファイルパスを取得する

先ほどインストールした `commander` パッケージを使って、コマンドライン引数として渡されたファイルパスを取得しましょう。この CLI アプリケーションでは、処理の対象とするファイルパスを次のようなコマンドの形式で受け取ります。

```
$ node main.js ./sample.md
```

`commander` でコマンドライン引数をパースするためには、`parse` メソッドにコマンドライン引数を渡します。

```
// commander モジュールを program オブジェクトとしてインポートする
const program = require("commander");
// コマンドライン引数をパースする
program.parse(process.argv);
```

`parse` メソッドを呼び出すと、コマンドライン引数をパースした結果を `program` オブジェクトから取り出せるようになります。今回の例では、ファイルパスは `program.args` 配列に格納されています。`program.args` 配列には `--key=value` のようなオプションや `--flag` のようなフラグを取り除いた残りのコマンドライン引数が順番に格納されています。

それでは `main.js` を次のように変更し、コマンドライン引数で渡されたファイルパスを取得しましょう。

```
main.js

// commander モジュールを program としてインポートする
const program = require("commander");
// コマンドライン引数を commander でパースする
program.parse(process.argv);

// ファイルパスを program.args 配列から取り出す
const filePath = program.args[0];
```

## 第 31 章 ユースケース: Node.js で CLI アプリケーション

```
console.log(filePath);
```

次のコマンドを実行すると、`program.args` 配列に格納された `./sample.md` 文字列が取得されてコンソールに出力されます。`./sample.md` は `process.argv` 配列では 3 番目に存在していましたが、パース後の `program.args` 配列では 1 番目になって扱いやすくなっています。

```
$ node main.js ./sample.md
./sample.md
```

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宣言的にコマンドライン引数を定義して処理できます。次のセクションではコマンドライン引数から取得したファイルパスを元に、ファイルを読み込む処理を追加していきます。

### 31.2.3 このセクションのチェックリスト

- `process.argv` 配列に `node` コマンドのコマンドライン引数が格納されていることを確認した
- `npm` を使ってパッケージをインストールする方法を理解した
- `require` 関数を使ってパッケージのモジュールを読み込むことを確認した
- `commander` を使ってコマンドライン引数をパースできることを確認した
- コマンドライン引数で渡されたファイルパスを取得してコンソールに出力できた

## 31.3 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からファイルパスを取得して利用できるようになりました。このセクションでは渡されたファイルパスを元に Markdown ファイルを読み込んで、標準出力に表示してみましょう。

### 31.3.1 fs モジュールを使ってファイルを読み込む

前のセクションで取得できるようになったファイルパスを元に、ファイルを読み込みましょう。Node.js でファイルの読み書きを行うには、標準モジュールの `fs` モジュールを使います。まずは読み込む対象のファイルを作成しましょう。`sample.md` という名前で `main.js` と同じ `nodecli` ディレクトリに配置します。

```
sample.md
```

```
# 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行うための基本的な関数を提供するモジュールです。**fs** モジュールのメソッドとして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一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り、残りの引数は処理の戻り値となります。処理が成功したときには、第一引数は **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** 構文によって例外処理を行えます。

次のサンプルコードは、指定したファイルを読み込む非同期形式の **readFile** メソッドの例です。

```
const fs = require("fs");

fs.readFile("sample.md", (err, file) => {
  if (err) {
    console.error(err);
  } else {
    console.log(file);
  }
});
```

そして、次のサンプルコードは、同じく指定したファイルを読み込む同期形式の **readFileSync** メソッドの例です。

```
const fs = require("fs");

try {
  const file = fs.readFileSync("sample.md");
} catch (err) {
  // ファイルが読み込めないなどのエラーが発生したときに呼ばれる
}
```

Node.js はシングルスレッドなので、他の処理をブロックしにくい非同期形式の 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、非同期処理に慣れおきましょう。

### readFile 関数を使う

それでは **fs** モジュールの **readFile** メソッドを使って **sample.md** ファイルを読み込んでみましょう。次のように **main.js** を変更し、コマンドライン引数から取得したファイルパスを元にファイルを読み込んでコンソールに出力します。

## 第 31 章 ユースケース: Node.js で CLI アプリケーション

main.js

```
const program = require("commander");
// fs モジュールを fs オブジェクトとしてインポートする
const fs = require("fs");

// コマンドライン引数からファイルパスを取得する
program.parse(process.argv);
const filePath = program.args[0];

// ファイルを非同期で読み込む
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なります。文字列になっていないのは、コールバック関数の第二引数はファイルの中身を表す **Buffer** インスタンスだからです。**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ます。そのため、そのまま `console.log` メソッドに渡しても人間が読める文字列にはなりません。

```
$ node main.js sample.md
<Buffer 23 20 73 61 6d 70 6c 65>
```

`fs.readFile` 関数は引数によってファイルの読み込み方を指定できます。ファイルのエンコードを第二引数であらかじめ指定しておけば、自動的に文字列に変換された状態でコールバック関数に渡されます。次のように `main.js` を変更し、読み込まれるファイルを UTF-8 として変換させます。

main.js

```
const program = require("commander");
const fs = require("fs");

program.parse(process.argv);
const filePath = program.args[0];

// ファイルを UTF-8 として非同期で読み込む
fs.readFile(filePath, { encoding: "utf8" }, (err, file) => {
  console.log(file);
});
```

先ほどと同じコマンドをもう一度実行すると、実行結果は次のようになります。`sample.md` ファイルの中身を文字列として出力できました。

```
$ node main.js sample.md
# sample
```

### エラーハンドリング

先ほどの例では触れませんでした。が、`fs` モジュールのコールバック関数の第一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。

次のように `main.js` を変更します。`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エラーメッセージを表示し、`process.exit` 関数に終了ステータスを指定してプロセスを終了しています。ここでは、一般的なエラーを表す終了ステータスの `1` でプロセスを終了しています。

```
main.js
```

```
const program = require("commander");
const fs = require("fs");

program.parse(process.argv);
const filePath = program.args[0];

// ファイルを非同期で読み込む
fs.readFile(filePath, { encoding: "utf8" }, (err, file) => {
  if (err) {
    console.error(err.message);
    // 終了ステータス 1 (一般的なエラー) としてプロセスを終了する
    process.exit(1);
    return;
  }
  console.log(file);
});
```

存在しないファイルである `notfound.md` をコマンドライン引数に渡して実行すると、次のようにエラーが発生して終了します。

```
$ node main.js notfound.md
ENOENT: no such file or directory, open 'notfound.md'
```

## 第 31 章 ユースケース: Node.js で CLI アプリケーション

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。次のセクションでは読み込んだ Markdown ファイルを HTML に変換する処理を追加していきます。

### 31.3.2 このセクションのチェックリスト

- fs モジュールの `readFile` 関数を使ってファイルを読み込んだ
- UTF-8 形式のファイルの中身をコンソールに出力した
- `readFile` 関数の呼び出しにエラーハンドリング処理を記述した

## 31.4 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。

### 31.4.1 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`<sup>\*5</sup> 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@0.7
```

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しょう。次のように `main.js` を変更し、読み込んだ Markdown ファイルを `marked` を使って HTML に変換します。`marked` パッケージをインポートした `marked` 関数は、Markdown 文字列を引数にとり、HTML 文字列に変換して返します。

```
main.js

const program = require("commander");
const fs = require("fs");
// marked モジュールを marked オブジェクトとしてインポートする
const marked = require("marked");

program.parse(process.argv);
const filePath = program.args[0];

fs.readFile(filePath, { encoding: "utf8" }, (err, file) => {
  if (err) {
```

---

<sup>\*5</sup> <https://github.com/chjj/marked>

```
        console.error(err.message);
        process.exit(1);
        return;
    }
    // Markdown ファイルを HTML 文字列に変換する
    const html = marked(file);
    console.log(html);
  });
```

### 31.4.2 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。そこで、アプリケーション中でオプションのデフォルト値を決め、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。

今回のアプリケーションでは、例として `gfm` という marked のオプションを扱います。

#### 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](https://github.github.com/gfm/), GFM<sup>\*6</sup>）に合わせて変換するかを決めるオプションです。marked ではこの `gfm` オプションがデフォルトで `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。次のように `sample.md` を変更し、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。

sample.md

# 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">サンプルファイル</h1>`

<sup>\*6</sup> <https://github.github.com/gfm/>

## 第 31 章 ユースケース: Node.js で CLI アプリケーション

```
<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。

```
main.js

const program = require("commander");
const fs = require("fs");
const marked = require("marked");

program.parse(process.argv);
const filePath = program.args[0];

fs.readFile(filePath, { encoding: "utf8" }, (err, file) => {
  if (err) {
    console.error(err.message);
    process.exit(1);
    return;
  }
  // gfm オプションを無効にする
  const html = marked(file, {
    gfm: false
  });
  console.log(html);
});
```

```
<h1 id="サンプルファイル">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>
```

自動リンクのほかにもいくつかの拡張がありますが、詳しくは [GitHub Flavored Markdown](#) のドキュメントを参照してください。

#### コマンドライン引数からオプションを受け取る

次に、gfm オプションをコマンドライン引数で制御できるようにしましょう。アプリケーションのデフォルトでは gfm オプションを無効にした上で、次のように `--gfm` オプションを付与してコマンドを実行できるようにします。

```
$ node main.js --gfm sample.md
```

コマンドライン引数で `--gfm` のようなオプションを扱いたいときには、commander の `option` メソッドを使います。次のように必要なオプションを定義してからコマンドライン引数をパースすると、`program.opts` メソッドでパース結果のオブジェクトを取得できます。

```
// gfm オプションを定義する
program.option("--gfm", "GFM を有効にする");
// コマンドライン引数をパースする
program.parse(process.argv);
// オプションのパース結果をオブジェクトとして取得する
const options = program.opts();
console.log(options.gfm);
```

`--gfm` オプションはファイルパスを指定する `sample.md` の前後のどちらについていても動作します。なぜなら `program.args` 配列には `program.option` メソッドで定義したオプションが含まれないためです。`process.argv` 配列を直接使っているところのようなオプションの処理が面倒なので、commander のようなパース処理を挟むのが一般的です。

#### デフォルト設定を定義する

アプリケーション側でデフォルト設定を持つておくことで、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。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しょう。オブジェクトのデフォルト値を別のオブジェクトで上書きするときには... (spread 構文) を使うと便利です（「[オブジェクト](#)」の章の「[オブジェクトの spread 構文でのマージ](#)」を参照）。

```
// コマンドライン引数のオプションを取得し、デフォルトのオプションを上書きする
const cliOptions = {
  gfm: false,
  ...program.opts(),
};
```

こうして作成した `cliOptions` オブジェクトから、marked にオプションを渡しましょう。main.js

## 第 31 章 ユースケース: Node.js で CLI アプリケーション

の全体は次のようになります。

```
main.js

const program = require("commander");
const fs = require("fs");
const marked = require("marked");

// gfm オプションを定義する
program.option("--gfm", "GFM を有効にする");
program.parse(process.argv);
const filePath = program.args[0];

// コマンドライン引数のオプションを取得し、デフォルトのオプションを上書きする
const cliOptions = {
  gfm: false,
  ...program.opts(),
};

fs.readFile(filePath, { encoding: "utf8" }, (err, file) => {
  if (err) {
    console.error(err.message);
    process.exit(1);
    return;
  }
  const html = marked(file, {
    // オプションの値を使用する
    gfm: cliOptions.gfm,
  });
  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。

```
$ node main.js sample.md
<h1 id="サンプルファイル">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>
```

また、`gfm` オプションを付与して実行すると次のように出力されるはずです。

```
$ node main.js --gfm sample.md
<h1 id="サンプルファイル">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>
```

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。次のセクションではアプリケーションのコードを整理し、最後にユニットテストを導入します。

#### 31.4.3 このセクションのチェックリスト

- `marked` パッケージを使って Markdown 文字列を HTML 文字列に変換した
- コマンドライン引数で `marked` の変換オプションを設定した
- デフォルトオプションを定義し、コマンドライン引数で上書きできるようにした

## 31.5 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合わせて、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。

前のセクションまでは、すべての処理をひとつの 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。

### 31.5.1 CommonJS でのモジュール化

実際にアプリケーションのモジュール化をする前に、CommonJS でのモジュール化について簡単に振り返ります。

Node.js では、複数の JavaScript ファイル間で変数や関数などをやり取りするために、CommonJS モジュールという仕組みを利用します。CommonJS 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。

## 第 31 章 ユースケース: Node.js で CLI アプリケーション

次の `greet.js` というファイルは、`greet` 関数をエクスポートするモジュールの例です。

```
greet.js
```

```
// greet.js
module.exports = function greet(name) {
  return `Hello ${name}!`;
};
```

`require` 関数を使って、指定したファイルパスの JavaScript ファイルをモジュールとしてインポートできます。次のコードでは先ほどの `greet.js` のパスを指定してモジュールとしてインポートして、エクスポートされた関数を取得しています。

```
greet-main.js
```

```
const greet = require("./greet");
greet("World"); // => "Hello World!"
```

`module.exports` オブジェクトそのものに代入するのではなく、`module.exports` オブジェクトのプロパティに代入することでも任意の値をエクスポートできます。次の `functions.js` というファイルでは、`foo` と `bar` の 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() {
  console.log("foo 関数が呼び出されました");
};
module.exports.bar = function() {
  console.log("bar 関数が呼び出されました");
};
```

このようにエクスポートされたオブジェクトは、`require` 関数の戻り値であるオブジェクトのプロパティとしてアクセスできます。次のコードでは先ほどの `functions.js` をインポートして取得したオブジェクトから `foo` と `bar` 関数をプロパティとして取得しています。

```
functions-main.js
```

```
const functions = require("./functions");
functions.foo();
```

```
functions.bar();
```

---

### 31.5.2 アプリケーションをモジュールに分割する

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 `marked` を使った Markdown の変換処理を記述します。

```
md2html.js
```

```
const marked = require("marked");

module.exports = (markdown, cliOptions) => {
  return marked(markdown, {
    gfm: cliOptions.gfm,
  });
};
```

---

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を元に 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。

```
main.js
```

```
const program = require("commander");
const fs = require("fs");
// md2html モジュールをインポートする
const md2html = require("./md2html");

program.option("--gfm", "GFM を有効にする");
program.parse(process.argv);
const filePath = program.args[0];

const cliOptions = {
  gfm: false,
  ...program.opts(),
};
```

## 第 31 章 ユースケース: Node.js で CLI アプリケーション

```
fs.readFile(filePath, "utf8", (err, file) => {
  if (err) {
    console.error(err);
    process.exit(1);
    return;
  }
  // md2html モジュールを使って HTML に変換する
  const html = md2html(file, cliOptions);
  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。

### 31.5.3 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。このセクションではテストフレームワークとして [Mocha](https://mochajs.org/)<sup>\*7</sup> 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。

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.strictEqual` メソッドを利用します。`assert.strictEqual` メソッドは第一引数と第二引数の評価結果が `===` で比較して異なる場合に、例外を投げる関数です。

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@6
```

`--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": {
```

---

<sup>\*7</sup> <https://mochajs.org/>

```
    "test": "mocha test/"
  },
  ...
}
```

この記述により、`npm test` コマンドを実行すると、`mocha` コマンドで `test/` ディレクトリにあるテストファイルを実行します。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。まだテストファイルを作っていないので、`Error: No test files found` というエラーが表示されます。

```
$ npm test
> mocha
```

```
Error: No test files found
```

#### 31.5.4 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 (GFM=false)", () => {
  const sample = fs.readFileSync(path.resolve(__dirname,
    "./fixtures/sample.md"), "utf8");
  const expected = fs.readFileSync(path.resolve(__dirname,
    "./fixtures/expected.html"), "utf8");

  assert.strictEqual(md2html(sample, { gfm: false }), expected);
});

it("converts Markdown to HTML (GFM=true)", () => {
  const sample = fs.readFileSync(path.resolve(__dirname,
    "./fixtures/sample.md"), "utf8");
  const expected = fs.readFileSync(path.resolve(__dirname,
    "./fixtures/expected-gfm.html"), "utf8");
```

## 第 31 章 ユースケース: Node.js で CLI アプリケーション

```
    assert.strictEqual(md2html(sample, { gfm: true }), 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 ファイルが存在します。

次のように変換元の Markdown ファイルを `test/fixtures/sample.md` に配置します。

```
test/fixtures/sample.md
```

```
# 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
```

そして、期待する変換結果の HTML ファイルも `test/fixtures` ディレクトリに配置します。`gfm` オプションの有無にあわせて、`expected.html` と `expected-gfm.html` の 2 つを次のように作成しましょう。

```
test/fixtures/expected.html
```

```
<h1 id="サンプルファイル">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>
```

```
test/fixtures/expected-gfm.html
```

```
<h1 id="サンプルファイル">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>
```

---

ユニットテストの準備ができたなら、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
✓ converts Markdown to HTML (GFM=true)

2 passing (31ms)
```

ユニットテストが通らなかった場合は、次のことを確認してみましょう。

- `test/fixtures` ディレクトリに `sample.md` と `expected.html`、`expected-gfm.html` というファイルを作成したか
- それぞれのファイルは文字コードが UTF-8 で、改行コードが LF になっているか
- それぞれのファイルの末尾に余計な改行文字が入っていないか

### 31.5.5 なぜユニットテストを行うのか

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行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。実際にテストを行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。

### 31.5.6 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

## 第 31 章 ユースケース: Node.js で CLI アプリケーション

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。

### 31.6 このセクションのチェックリスト

- Markdown の変換処理を CommonJS モジュールとして `md2html.js` に切り出し、`main.js` から読み込んだ
- `mocha` パッケージをインストールし、`npm test` コマンドで `mocha` コマンドを実行できることを確認した
- `md2html` 関数のユニットテストを作成し、テストの実行結果を確認した



## 第32章

### ユースケース: Todo アプリケーション

ここではブラウザで動作するウェブアプリケーションのユースケースとして、Todo アプリケーション（以下 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て表示するだけであったため、状態を管理する部分はほとん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を作るにあたって必要になる設計や考え方について見ていきます。

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

#### 32.1 エントリーポイント

エントリーポイントとは、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](#)」のユースケースでは、エントリーポイントは 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 `script` 要素で指定した JavaScript ファイルが読み込まれます。

今回の Todo アプリは JavaScript の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` 要素ごとに別々のモジュールスコープを持ちます。モジュールスコープとは、モジュールのトップレベルに自動的に作成されるスコープで、グローバルスコープの下に作られます。JavaScript モジュールを別々の `script` 要素で読み込むと、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` 要素で定義した変数にアクセスできないことを示しています。これは JavaScript モジュールをファイル

## 第 32 章 ユースケース: Todo アプリケーション

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アクセスできない
  console.log(scopeA); // => ReferenceError: scopeA is not defined
</script>
```

このようにモジュールを別々の `script` 要素で扱っているとモジュール同士は連携できません。そのため、HTML では `script` 要素で `index.js` のみを読み込み、この `index.js` から `import` 文で他のモジュールを読み込みます。`import` 文を使うことで、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 におけるエントリーポイントとします。

つまり、今回作成する Todo アプリではエントリーポイントとして HTML と JavaScript の 2 つを用意します。

- `index.html`: 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 では最初に読み込まれる

このセクションでは、この 2 つのエントリーポイントを作成して読み込むところまでを確認します。

### 32.1.1 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 JavaScript など複数のファイルが必要となります。そのため、まずそれらのファイルを置くためのディレクトリを作成します。

ここでは `todoapp` という名前で新しいディレクトリを作成します。ここからは作成した `todoapp` ディレクトリ以下で作業していきます。

またこのプロジェクトで作成するファイルは、必ず文字コード (エンコーディング) を **UTF-8**、改行コードを **LF** にしてファイルを保存します。

### 32.1.2 HTML ファイルの用意

エントリーポイントとして、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。エントリーポイントとなる HTML として `index.html` を `todoapp` ディレクトリに作成し、次のような内容にします。`body` 要素の一番下で `script` 要素を使って読み込んでいる `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。

```
index.html
```

```
<html lang="ja">
  <head>
```

```
<meta charset="utf-8" />
<title>Todo App</title>
</head>
<body>
  <h1>Todo App</h1>
  <script type="module" src="index.js"></script>
</body>
</html>
```

次に `index.js` を `todoapp` ディレクトリに作成し、次のような内容にします。`index.js` にはスク립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ります。

```
index.js

console.log("index.js: loaded");
```

ここまでの `todoapp` ディレクトリのファイル配置は次のようになっています。

```
todoapp
├── index.html
└── index.js
```

次はこの `index.html` をブラウザで開いて、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。

### 32.1.3 ローカルサーバーで HTML を確認する

ウェブブラウザで `index.html` を開く前に、開発用のローカルサーバーを準備します。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:///`からはじまる URL になります。`file` スキーマでは [Same Origin Policy](#) により、JavaScript モジュールが正しく動作しません。そのため、本章ではローカルサーバーを立ち上げた上で、`http` からはじまる 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ーを起動します。`npx` コマンドを使って、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ーモジュールをダウンロードと同時に実行します。まだ `npx` コマンドが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](#)」の章を参照してください。

```
# todoapp/ディレクトリに移動する
$ cd todoapp/
```

## 第 32 章 ユースケース: Todo アプリケーション

```
# todoapp/をルートにしたローカルサーバーを起動する
$ npx @js-primer/local-server
```

todoapp のローカルサーバーを起動しました。  
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ーの 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。

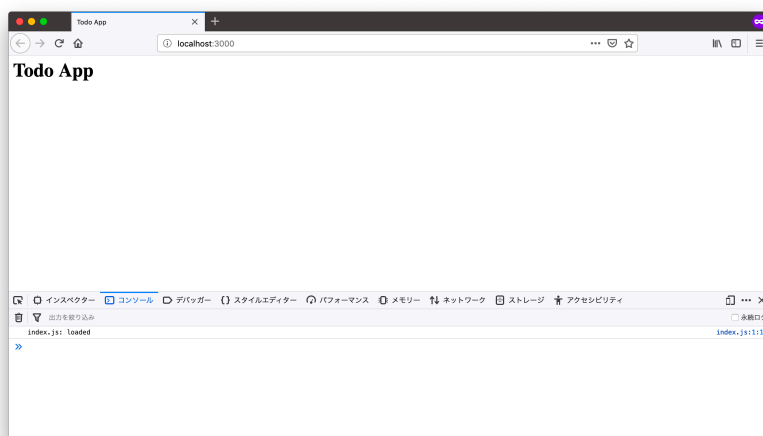


図 32.1 Web コンソールにログが表示されている

**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に開発者ツールが同梱されていますが、本書では 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）のウェブ開発サブメニューで“ウェブコンソール”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 MDN の「[ウェブコンソールを開く<sup>\\*1</sup>](#)」を参照してください。

#### 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、次のような問題に該当していないかを確認してください。

#### エラー例 index.js の読み込みに失敗している

`script` 要素の `src` 属性に指定した `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。先ほども紹介したように、`file:`からはじまるページ上からは JavaScript モジュールは正しく動作しません。そのため、ローカルサーバーを起動し、ローカルサーバー (`http:`からはじまる URL) にアクセスしていることを確認してください。

#### エラー例 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、JavaScript モジュールである `index.js` が読み込ま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<sup>\\*2</sup>](#)にネイティブで 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。

### 32.1.4 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みましょう。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` というファイルを作成し、これを `index.js` からモジュールとして読み込みます。

次のようなファイル配置となるように `src/App.js` を作成します。

```
todoapp
├── index.html
├── index.js
└── src
    └── App.js
```

`src/App.js` 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。App.js は App

<sup>\*1</sup> [https://developer.mozilla.org/ja/docs/Tools/Web\\_Console/Opening\\_the\\_Web\\_Console](https://developer.mozilla.org/ja/docs/Tools/Web_Console/Opening_the_Web_Console)

<sup>\*2</sup> <https://caniuse.com/#feat=es6-module>

## 第 32 章 ユースケース: Todo アプリケーション

というクラスを名前つきエクスポートしているモジュールです。また、**App** クラスのコンストラクタにはコンソールログを出力するコードを確認用に書いておきます。

src/App.js

```
console.log("App.js: loaded");
export class App {
  constructor() {
    console.log("App initialized");
  }
}
```

次に、この **src/App.js** を **index.js** から利用するために **import** します。**index.js** を次のように書き換え、**App.js** から **App** クラスをインポートしてインスタンス化します。

index.js

```
import { App } from "../src/App.js";
const app = new App();
```

再度ローカルサーバーの URL (<http://localhost:3000>) 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出力されます。

```
App.js: loaded
App initialized
```

まず **index.js** から **src/App.js** が名前つきエクスポートしている **App** クラスを名前つきインポートしています。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。

**App.js の読み込みに失敗する**

ここまでの JavaScript モジュールの読み込みでエラーが発生して動かない場合には、次のことを確認します。

ディレクトリ構造や **import** 文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ません。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みてください。

**import** 文を使った JavaScript のモジュール読み込み時に起きる典型的なエラーと対処を次にまとめています。

**エラー例** `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」というエラーが出ています。このエラーが出ているということは、import 文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 文がトップレベルではないところに書かれている、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。

関数の中などに import 宣言していると、import 宣言がトップレベルではないためエラーが発生します。この場合は import 文をトップレベル（プログラムの直下）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 文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 要素の type 属性に module 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 文で読み込んだ 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）。

**エラー例** 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 を読み込めないというエラーが出ています。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 文では、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";
```

### 32.1.5 セクションのまとめ

このセクションでは、エントリーポイントとなる HTML を作成し、JavaScript モジュールのエントリーポイントとなる JavaScript ファイルを読み込むところまでを実装しました。

### 32.1.6 このセクションのチェックリスト

- todoapp という名前のプロジェクトディレクトリを作成した
- エントリーポイントとなる 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
- ローカルサーバーを使って index.html を表示した
- 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

## 第 32 章 ユースケース: Todo アプリケーション

ここまでの Todo アプリは次の URL で確認できます。

- <https://jsprimer.net/use-case/todoapp/entrypoint/module-entry/>

## 32.2 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改めて見ていきましょう。

Todo アプリには、次のような機能を実装していきます。Todo アイテムの追加、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、まず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いきます。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 アプリを完成させ、次のセクションから実際に JavaScript を使って Todo アプリの機能を実装していきます。

### 32.2.1 HTML と CSS と JavaScript

Todo アプリはブラウザで動くアプリケーションとして作成しますが、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。今回は HTTP 通信などはいらないクライアントサイドのみで解決するウェブアプリなので、サーバーサイドの言語は登場しません。

- 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 で動作をつけることで実装されます。そのため、ウェブアプリは HTML、CSS、JavaScript を組み合わせて作られています。

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。また、ユーザーが実装したさまざまな種類の UI フレームワークがあります。そのため、Todo アプリという題材をとってみても、フレームワークや人によって書き方がまったく異なる場合もあります。

今回の Todo アプリは特別な UI 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。



### 32.2.2 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。

index.html

```
<html lang="ja">
  <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 -->
      </footer>
    </div>
  </body>
</html>
```

## 第 32 章 ユースケース: Todo アプリケーション

```
<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。まだ JavaScript で Todo アプリの機能は実装していませんが、見た目だけの Todo アプリはこれで完成です。

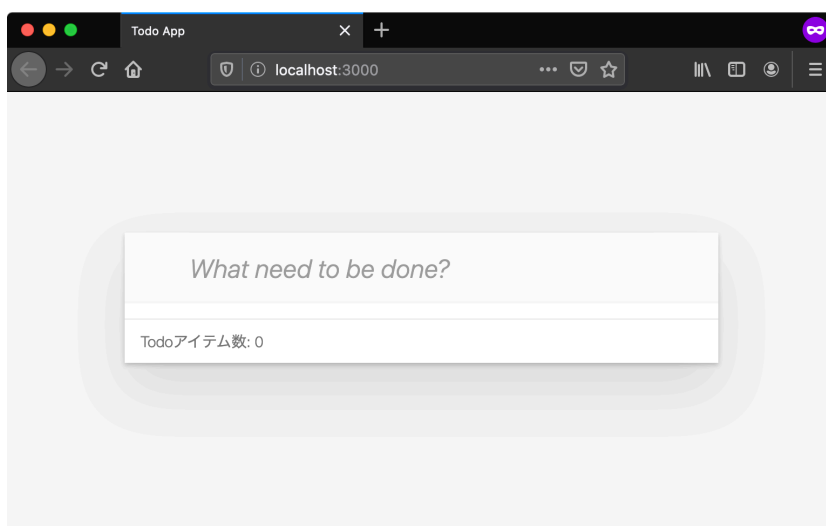


図 32.2 todoapp の HTML と CSS による骨組み

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て解説は省略します。CSS に定義したスタイルを正しく適用するには、`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、HTML の構造が同じになっているかを確認してください。

## 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つのページの中で同じクラス名を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注意が必要です。

今回の `todoapp` というクラス名を持つ要素を、CSS から `.todoapp` という CSS セレクタで指定できます。CSS セレクタとはクラス名などを使って、HTML 要素を指定できる記法です。特定の「クラス名」を持つ要素の場合は、`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` というクラス名を持つ要素の `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意味になります。

```
.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」を持つ要素の場合は `#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、その要素に対するページ内でユニークな識別子をつ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つのページの中では同じ 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 セレクタを使って要素を選択できます。

次のコードでは、`document.querySelector("CSS セレクタ")` を利用して特定の 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 属性を目印としてつけています。わかりやすくするために、JavaScript から扱う id 属性は慣習的に `js-`からはじまる名前にしています。

## 4. Todo アプリのメインとなる Todo リスト

`js-todo-list` という id 属性をつけた div 要素が、今回の Todo アプリのメインとなる Todo リス

## 第 32 章 ユースケース: Todo アプリケーション

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` 属性をつけています。

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。また `<!-->` で囲まれた範囲は HTML のコメントであるため、表示されません。

### 5. Todo アイテム数の表示

`js-todo-count` という `id` 属性をつけた `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 個となりますが、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。

### 32.2.3 セクションのまとめ

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。

### 32.2.4 このセクションのチェックリスト

- 実装する Todo アプリの構成要素を理解した
- HTML、CSS、JavaScript の役割の違いを理解した
- Todo アプリの見た目を HTML と CSS で定義した

ここまでの Todo アプリは次の URL で確認できます。

- <https://jsprimer.net/use-case/todoapp/app-structure/todo-html/>

## 32.3 Todo アイテムの追加を実装する

ここからは JavaScript で Todo アプリの機能を作成していきます。

このセクションでは、前のセクションで HTML に目印をつけた Todo リスト (`#js-todo-list`) に対して Todo アイテムを追加する処理を実装します。

### 32.3.1 Todo アイテムの追加

まず、Todo アプリではどのような操作をしたら、Todo アイテムを追加できるかを見ていきます。Todo アプリでは、ユーザーが次のような操作を行った場合に、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` キーを押して送信する
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。

- Todo アイテムのタイトルを取得するために、`input` 要素（入力欄）から内容を取得する

- `[Enter]` キーで送信されたことを知るために、form 要素の `submit` イベント（送信）を監視する
- 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はじめてみましょう。

### 32.3.2 入力内容をコンソールに表示する

form 要素で `[Enter]` キーを押して送信すると `submit` イベントが発生します。この `submit` イベントは HTML 要素の `addEventListener` メソッドを利用することで受け取れます。

次のコードでは、指定した form 要素から `submit` イベントが発生したときに呼び出されるコールバック関数を登録しています。

```
// 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ときに呼ばれるコールバック関数（イベントリスナー）
});
```

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が、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。

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す。

次のコードでは、フォーム (`#js-form`) を `[Enter]` で送信すると、input 要素 (`#js-form-input`) の内容をコンソールへ表示する処理を実装しています。

```
src/App.js
```

```
export class App {
  mount() {
    const formElement = document.querySelector("#js-form");
    const inputElement = document.querySelector("#js-form-input");
    formElement.addEventListener("submit", (event) => {
```

## 第 32 章 ユースケース: Todo アプリケーション

```
        // 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` メソッドが実行されるようになります。`submit` イベントがリ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。

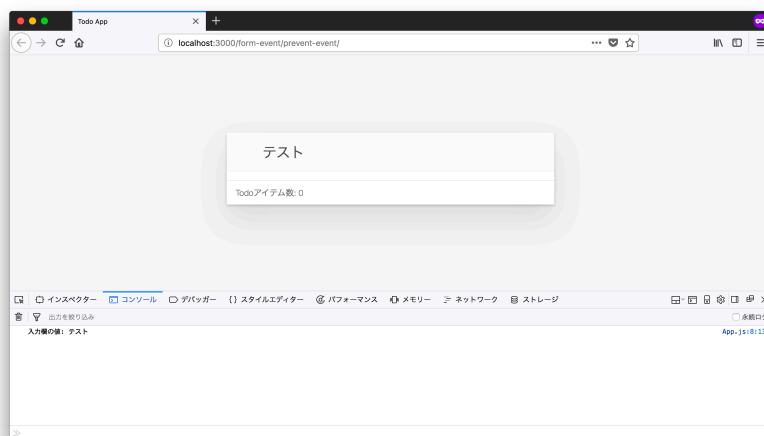


図 32.3 入力内容がコンソールに表示される

先ほどの `App#mount` メソッドでは、`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持つデフォルトの動作をキャンセルするメソッドです。

## 32.3 Todo アイテムの追加を実装する

フォームが持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 に対してフォームの内容を送信します。しかしこの動作は邪魔になるため、`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。

src/App.js より抜粋

```
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 に対してフォームの送信が行われると、結果的にページがリロードされてしまいます。そのため、`event.preventDefault()` を呼び出し、デフォルトの動作をキャンセルしていました。`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。

src/App.js から一部をコメントアウトした例

```
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` ディレクトリに、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)  
└── src  
    └── App.js (App#mount の実装)
```

ここまでの Todo アプリは次の URL で確認できます。

- <https://jsprimer.net/use-case/todoapp/form-event/prevent-event/>

### 32.3.3 入力内容を Todo リストに表示する

フォーム送信時に入力内容を取得する方法がわかったので、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` というファイルを `src/view/html-util.js` というパスに作成します。

この `html-util.js` は「[Ajax 通信](#)」の「[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。

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 HTML 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 HTML 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;");
}

/**
 * HTML 文字列から HTML 要素を作成して返す
 * @param {string} html
 */
export function htmlToElement(html) {
  const template = document.createElement("template");
  template.innerHTML = html;
  return template.content.firstElementChild;
}

/**
```



```
* HTML 文字列から DOM Node を作成して返すタグ関数
* @return {Element}
*/
export function element(strings, ...values) {
  const htmlString = strings.reduce((result, str, i) => {
    const value = values[i - 1];
    if (typeof value === "string") {
      return result + escapeSpecialChars(value) + str;
    } else {
      return result + String(value) + str;
    }
  });
  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) {
  // containerElement の中身を空にする
  containerElement.innerHTML = "";
  // containerElement 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。

#### element タグ関数のサンプルコード

```
// HTML 文字列から HTML 要素を作成
const newElement = element`<ul>
  <li>新しい要素</li>
```

## 第 32 章 ユースケース: Todo アプリケーション

```
</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なので、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。

## render 関数のサンプルコード

```
// 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
// すでに 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` タグ関数を使って、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) => {
```

## 32.3 Todo アイテムの追加を実装する

```
// 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}`;
// 入力欄を空文字列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し、フォームに入力してから **Enter** を押すと Todo リストに **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。

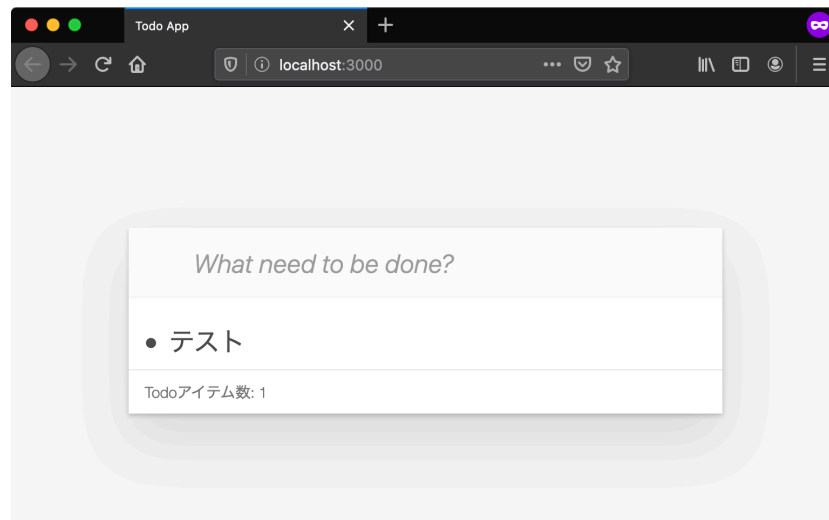


図 32.4 Todo リストへアイテムを追加

## 第 32 章 ユースケース: Todo アプリケーション

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
├── src
│   ├── App.js(Todo アイテムの表示の実装)
│   └── view
│       └── html-util.js(新規追加)
```

ここまでの Todo アプリは次の URL で確認できます。

- <https://jsprimer.net/use-case/todoapp/form-event/add-todo-item/>

### 32.3.4 セクションのまとめ

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 アイテムを作成し、これを Todo リストに追加する機能を実装しました。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、Todo リスト要素を **直接 HTML 要素として追加**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 のみにしか状態が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、実際に起きる問題やそれを解決するための仕組みを見ていきます。

### 32.3.5 このセクションのチェックリスト

- フォームの送信を `submit` イベントで受け取り、入力内容を確認した
- HTML 文字列から HTML 要素を作成する `html-util.js` を実装した
- フォームから Todo アイテムを追加した
- Todo アイテムの追加に合わせて Todo アイテム数を更新した

このセクションで、Todo アプリに Todo アイテムを追加する機能が実装できました。

- Todo アイテムを追加できる

Todo アプリに実装する残りの機能は次のとおりです。

- 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

## 32.4 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、イベントを受け取って直接 DOM を更新する方法には柔軟性がない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、Todo アイテムごとに `id` 属性などのユニークな識別子がないため、特定のアイテムを指定して更新や削除をする機能が実装できません。

このセクションでは、まずどのような点で柔軟性の問題が起きやすいのかを見ていきます。そして、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、「Todo アイテムの追加」の機能をリファクタリングしていきます。

### 32.4.1 直接 DOM を更新する問題

「**Todo アイテムの追加を実装する**」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) を直接更新していました。そのため、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。

この場合に 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 要素は文字列しか扱えないため、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、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 | フォームを入力して送信   | Todo リスト ( <code>#js-todo-list</code> ) に Todo アイテム要素を作成して子要素として追加。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     |

1 つの操作に対する表示の更新箇所が増えるほど、操作に対する処理 (リスナーの処理) 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

32.4.2 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。ここのモデルとは Todo アイテムや Todo リストなどのモノの状態や操作方法を定義したオブジェクトという意味です。クラスでは操作方法是メソッドとして実装し、状態はインスタンスのプロパティで管理できるため、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` というメソッドがあると良さそうです。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` というメソッドも必要そうです。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持つかを考えて実装します。

このようにモデルを考えた後、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。「フォームを入力して送信」という操作をした場合には、`TodoListModel` (Todo リスト) に対して `TodoItemModel` (Todo アイテム) を追加します。そして、`TodoListModel` から Todo アイテムの一覧を取得し、それを元に DOM を組み立て、表示を更新します。

先ほどの表にモデルを入れてみます。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でも表示の処理の複雑さが一定に保てる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 | フォームを入力して送信   | <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 |

この表を元に改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、Todo アイテムの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

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「フォームを入力して送信」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いいはずです。

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。

### 32.4.3 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` イベントを発生（ディスパッチ）させます。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。先ほどの表の「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、DOM Events と呼ばれるイベントの仕組みが利用できます。Node.js では、`events` と呼ばれる組み込みのモジュールで同様のイベントの仕組みが利用できます。

実行環境が提供するイベントの仕組みを利用すると簡単ですが、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持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。

### 32.4.4 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（イベントを発生させた）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` が更新され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のは大変です。

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` には作成した `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とき、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`) : 値を更新したとき、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。

## 第 32 章 ユースケース: Todo アプリケーション

- ディスパッチ側: `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
- リッスン側: `addEventListener` 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

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に類似する 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
   */
  addEventListener(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.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
 */
removeEventListener(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` では、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 `addEventListener` 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、`addEventListener` 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();
```

## 第 32 章 ユースケース: Todo アプリケーション

```
// イベントリスナー（コールバック関数）を登録
event.addEventListener("test-event", () => console.log("One!"));
event.addEventListener("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!"
```

### 32.4.5 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 リストを表現します。

- `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` というファイル名で作成します。

`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`) を持たせます。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ごと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` アイテムを指定して更新する処理のときに、アイテムを区別する識別子として利用します。

#### TodoItemModel.js を利用するサンプルコード

```
import { TodoItemModel } from "../TodoItemModel.js";  
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` というファイル名で作成します。

`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を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 {  
  /**
```

## 第 32 章 ユースケース: Todo アプリケーション

```
    * @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 の合計個数を返す
     * @returns {number}
     */
    getTotalCount() {
        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` クラスのインスタンスに対して、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ときに、`TodoListModel#onChange` で登録したイベントリスナーが呼び出されます。

#### TodoListModel.js を利用するサンプルコード

```
import { TodoItemModel } from './TodoItemModel.js';
import { TodoListModel } from './TodoListModel.js';
// 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.getTotalCount()); // => 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.getTotalCount()); // => 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って、表示の更新を試みましょう。

### 32.4.6 モデルを使って表示を更新する

先ほど作成した `TodoListModel` と `TodoItemModel` クラスを使って、「Todo アイテムの追加」を書き直してみます。

前回のコードでは、フォームを送信すると直接 DOM へ要素を追加していました。今回のコードでは、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` を見ていきます。

src/App.js

```
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.getTotalCount()}`;
    });
    // 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 つの部分に変更されているので、順番に見ていきます。

### 1. TodoList の初期化

作成した TodoListModel と TodoItemModel をインポートしています。

```
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を開始されるのに合わせるためです。

src/App.js より抜粋

```
// ... 省略...
export class App {
    constructor() {
        // 1. TodoList の初期化
        this.todoListModel = new TodoListModel();
    }
    // ... 省略...
}
```

---

## 第 32 章 ユースケース: Todo アプリケーション

## 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` 関数を使って `containerElement` の中身に上書きしています。また、アイテム数は `TodoListModel#getTotalCount` メソッドで取得できるため、アイテム数を管理していた `todoItemCount` という変数は削除できます。

src/App.js より抜粋

```
// render 関数を import に追加する
import { element, render } from './view/html-util.js';
export class App {
  // ... 省略...
  mount() {
    // ... 省略...
    this.todoListModel.onChange(() => {
      // ... 省略...
      // 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.getTotalCount()}`;
    });
    // ... 省略...
  }
}
```



### 3. フォームを送信したら、新しい TodoItem を追加する

前回のコードでは、フォームを送信（submit）すると直接 DOM へ要素を追加していました。今回のコードでは、`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` メソッドを使ってモデルを更新する処理へ置き換えるだけです。

#### 32.4.7 セクションのまとめ

今回のセクションでは、前セクションの「[Todo アイテムの追加を実装する](#)」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。コード量は増えてましたが、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、残りの機能である「Todo アイテムの更新」や「Todo アイテムの削除」を実装していきます。

#### 32.4.8 このセクションのチェックリスト

- 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
- Todo リストと Todo アイテムをモデルとして実装した
- `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

ここまでの 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/>

## 32.5 Todo アイテム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つけて完了済みに、逆に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 アイテムごとに持ち、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。

### 32.5.1 Todo アイテムの更新

現時点では Todo アイテムが完了済みかどうかの状態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">` 要素を使って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

図 32.5 input 要素の checked 属性の違い

`src/App.js` の `TodoListModel#onChange` メソッドで登録したリスナー関数内を書き換え、チェックボックスを表示しています。

Todo アイテム要素である `<li>` 要素中に次のように `<input>` 要素を追加して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>` 要素を使って打ち消し線を表示しています。

`src/App.js` から抜粋

```
this.todoListModel.onChange(() => {
  const todoListElement = element`<ul />`;
  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);
  });
  render(todoListElement, containerElement);
  todoItemCountElement.textContent = `Todo アイテム数:
    ${this.todoListModel.getTotalCount()}`;
```

```
});
```

`<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つける
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書けます。

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索していました。`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。

そして、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。

```
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
});
```

ここまですと、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。

## 第 32 章 ユースケース: Todo アプリケーション

## 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。

src/model/TodoListModel.js の変更点を抜粋

```
// =====  
// TodoListModel.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 }}  
 */  
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。

src/App.js の TodoListModel#onChange メソッドで登録したリスナー関数内を次のように書き換えます。

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。

src/App.js から変更点を抜粋

```
this.todoListModel.onChange(() => {
```

```
const todoListElement = element`<ul />`;
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);
});
render(todoListElement, containerElement);
todoItemCountElement.textContent = `Todo アイテム数:
  ${this.todoListModel.getTotalCount()}`;
});
```

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出され、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。

### 32.5.2 Todo アイテムの削除

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に指定した Todo アイテムを削除する処理を呼び出します。

## 第 32 章 ユースケース: Todo アプリケーション

## TodoListModel に指定した Todo アイテムを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。

src/model/TodoListModel.js の変更点を抜粋

```
// =====
// TodoListModel.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 に一致しない TodoItem だけを残すことで、id に一致する TodoItem を削除する
  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す処理を実装します。

`src/App.js` の `TodoListModel#onChange` メソッドで登録したリスナー関数内を次のように書き換えます。`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。

src/App.js から変更点を抜粋

```
this.todoListModel.onChange(() => {
  const todoListElement = element`<ul />`;
```

```
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>
      <s>${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) がクリックされたときに 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
    });
  });
  todoListElement.appendChild(todoItemElement);
});
render(todoListElement, containerElement);
todoItemCountElement.textContent = `Todo アイテム数:
  ${this.todoListModel.getTotalCount()}`;
});
```

---

## 第 32 章 ユースケース: Todo アプリケーション

`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。

### 32.5.3 このセクションのチェックリスト

- Todo アイテムの完了状態として `<input type="checkbox">` を表示に追加した
- チェックボックスが更新されたときの `change` イベントのリスナー関数で Todo アイテムを更新した
- Todo アイテムを削除するボタンとして `<button class="delete">x</button>` 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した

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 アイテムを削除できる

ここまでの Todo アプリは次の URL で確認できます。

- <https://jsprimer.net/use-case/todoapp/update-delete/delete-feature/>

最後のセクションでは、`App.js` のリファクタリングを行って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。

## 32.6 Todo アプリのリファクタリング

前のセクションで、予定していた Todo アプリの機能はすべて実装できました。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、コードの行数が線形的に増えていきます。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なり、メンテナンス性が低下してしまいます。

ここで、`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持っています。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管理者と言えます。

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切り出すリファクタリングを行います。

### 32.6.1 View コンポーネント

`App` クラスの大部分を占めているのは `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部品ごとのモジュールに分け、**App** クラスから作成したモジュールを使うような形にリファクタリングしていきます。ここでは、表示のための処理を扱うクラスを **View** コンポーネントと呼び、ここでは **View** をファイル名の末尾につけることで区別します。

Todo リストの表示は次の2つの部品（View コンポーネント）から成り立っています。

- Todo アイテム View 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 View 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。これらの View のモジュールは、**src/view/**ディレクトリに作成していきます。

- **TodoItemView**: Todo アイテム View コンポーネント
- **TodoListView**: Todo リスト View 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。

**src/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>
      </li>`
  }
}
```

## 第 32 章 ユースケース: Todo アプリケーション

```

        : element`<li><input type="checkbox" class="checkbox">
            ${todoItem.title}
            <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 `App` クラスでの HTML 要素を作成する部分を元になっています。`createElement` メソッドは、`TodoItemModel` のインスタンスだけではなく `onUpdateTodo` と `onDeleteTodo` という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 `View` で発生した際に呼び出されます。

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。

## TodoItemView を利用するサンプルコード

```

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呼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。

src/view/TodoListView.js ファイルを作成し、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
   */
}
```

## 第 32 章 ユースケース: Todo アプリケーション

```

    * @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 />`;
        // 各 TodoItem モデルに対応した HTML 要素を作成し、リスト要素へ追加する
        todoItems.forEach(todoItem => {
            const todoItemView = new TodoItemView();
            const todoItemElement = todoItemView.createElement(todoItem, {
                onDeleteTodo,
                onUpdateTodo
            });
            todoListElement.appendChild(todoItemElement);
        });
        return todoListElement;
    }
}

```

TodoListView#createElement メソッドは TodoItemView を使って Todo アイテムの HTML 要素を作り、<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。

### 32.6.2 App のリファクタリング

最後に、作成した TodoItemView クラスと TodoListView クラスを使って 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 クラスを使って Todo リストの HTML 要素を作るように変更します。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渡します。

- 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を発生したときに呼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を発生したときに呼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.getTotalCount()}`;
    });
    formElement.addEventListener("submit", (event) => {
      event.preventDefault();
      this.todoListModel.addTodo(new TodoItemModel({
        title: inputElement.value,
        completed: false
      }));
      inputElement.value = "";
    });
  }
}
```

第 32 章 ユースケース: Todo アプリケーション

```
    });  
  }  
}
```

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 | <code>this.todoListModel.onChange(listener)</code>            | TodoListModel が変更イベントを受け取る     |
| View → Model | <code>formElement.addEventListener("submit", listener)</code> | フォームの送信イベントを受け取る               |
| View → Model | <code>onUpdateTodo: listener</code>                           | 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 | <code>onDeleteTodo: listener</code>                           | 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知りやすくするため、リスナー関数を App クラスのメソッドとして定義し直してみましょう。

次のように、それぞれ対応するリスナー関数を `handle` メソッドとして実装して、それを呼び出すように変更しました。

```
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 を追加するとき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 の状態を更新したとき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 を削除したとき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 todoItemCountElement = document.querySelector("#js-todo-count");
  const containerElement = 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 });
      }
    });
    containerElement.appendChild(todoListElement);
    todoItemCountElement.textContent = todoItems.length;
  });
}
```

## 第 32 章 ユースケース: Todo アプリケーション

```
        }
    });
    render(todoListElement, containerElement);
    todoItemCountElement.textContent = `Todo アイテム数:
        ${this.todoListModel.getTotalCount()}`;
    });

    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。

### 32.6.3 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。

- `App` から表示に関する処理を `View` コンポーネントに分割した
- 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 アプリにはまだ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つながる場合もあります。

余力がある人は、次の機能を追加して Todo アプリ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` でのイベントリスナー登録に対応して、`App#unmount` を実装し、イベントリスナーを解除できるようにする

`App#mount` と対応する `App#unmount` を作成するという 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。ウェブページにはページ読み込みが完了したときに発生する `load` イベントと、読み込んだページを破棄したときに発生する `unload` 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 を実装したコードは、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ください。

- <https://jsprimer.net/use-case/todoapp/final/more/>

#### 32.6.4 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](#)<sup>\*3</sup> 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たものです<sup>\*4</sup>。

現実では、ライブラリをまったく使わずウェブアプリケーションを実装することはほとんどありません。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がなくなったり、最後の課題として残ったライフサイクルの問題なども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 1 部の基本文法や第 2 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、それらも基礎となる技術を使ってい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

---

<sup>\*3</sup> <http://todomvc.com/>

<sup>\*4</sup> ライブラリやフレームワークを使わずに実装した JavaScript を Vanilla JS と呼ぶことがあります。

## 第 32 章 ユースケース: Todo アプリケーション

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。知らなかっ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ったことや知らなかったことがあるのは問題ありません。知らなかったことを見つけたときにそれが何かを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。

## 付録 A

### 参考リンク集

---

ここでは、本編で取り上げられなかった JavaScript の周辺ツールやライブラリなどをいくつか紹介します。これらは時流に左右されて古くなりやすい情報であるため、本編からは独立した付録としてまとめています。

#### A.1 開発を補助するツール

JavaScript を使った開発に役立つツールをいくつか紹介します。

##### A.1.1 トランスパイラー

トランスパイラーとはソースコードからソースコードへ変換 (Transpile) する機能を持つツールです。ここでは、アプリケーション開発でも利用されることが多い JavaScript のソースコードに変換するトランスパイラーを紹介します。

###### Babel

[Babel](#) は、ECMAScript の新しい構文を古い ECMAScript の構文に変換することを主な機能にしたトランスパイラーです。たとえば、Internet Explorer など古いブラウザは ECMAScript 2015 をサポートしていないため新しい構文をパースできません。Babel で ES2015 の構文を ES5 でエミュレートするコードへ変換することで、古い構文しかサポートしていない実行環境でも動かせます。

また、「[ECMAScript](#)」の章でも紹介したように、最新のプロポーザルの機能を試すツールとしても利用されています。

- Babel: <https://babeljs.io/>

###### TypeScript

[TypeScript](#) は、JavaScript に静的型づけの構文を追加した言語とトランスパイラーです。TypeScript 言語には型注釈などの構文が用意されており、JavaScript のコードに対して型注釈をつけて静的な型チェックができます。また、TypeScript のコードは型に関する構文などを取り除いた JavaScript のコードに変換できます。

- TypeScript: <https://www.typescriptlang.org/>

### A.1.2 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 1 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どり、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。CommonJS モジュールは Node.js での実行を想定したものであったため、そのままではウェブブラウザ上で動作しません。そのため、CommonJS モジュールをブラウザでも実行できるようにモジュールの依存関係を解決したりファイルを結合する目的でモジュールバンドラーと呼ばれるツールが登場しました。

#### webpack

[webpack](#) は、JavaScript アプリケーションの作成に適したモジュールバンドラーです。webpack は、CommonJS モジュールや ECMAScript モジュールの依存関係を解決し、アプリケーション向けに最適化しながらモジュールを結合します。JavaScript のモジュール以外にも、画像や CSS などのリソースを読み込む仕組みが用意されており、さまざまな種類のファイルを扱えます。また、webpack にはプラグインで拡張できる仕組みが用意されており、柔軟な変換が可能です。

- webpack: <https://webpack.js.org/>

#### Rollup

[Rollup](#) は、JavaScript ライブラリの作成に適したモジュールバンドラーです。Rollup は、ECMAScript モジュールを主に扱い、モジュールの依存関係を解決し、ライブラリ向けに最適化しながらモジュールを結合します。CommonJS 形式や ECMAScript モジュール形式などの複数の形式のファイルを生成するといったライブラリ作成に向けた設定が柔軟にできます。また Rollup も、webpack と同様にプラグインで拡張でき、柔軟な変換が可能です。

- Rollup: <https://rollupjs.org/>

### A.1.3 コーディングスタイルの統一

複数人での開発においては、改行の位置やインデントの幅など、ソースコードのフォーマットを統一します。なぜなら、異なったスタイルのコードはレビューに余計な時間がかかってしまうからです。また、使うべきでない古いイディオムやバグを生みやすい危険なコードの混入を防ぎ、品質を保つことも重要です。

これらのコーディングスタイルの統一は、一貫性を持って持続的に行うことが重要です。そのため、ツールを使って自動化することが推奨されます。

#### Prettier

[Prettier](#) は JavaScript をはじめとする多くの言語に対応した汎用的なコードフォーマッターです。

設定ファイルがなくても利用できるため、導入しやすいのが大きな特徴です。

- Prettier: <https://prettier.io/>

### ESLint

**ESLint** は JavaScript ファイル用の Lint ツールです。**Lint** とは、ソースコードファイルを静的解析して不適切なコードやコーディングスタイルに合わないコードを検知する仕組みのことです。Lint を使うことで、チーム内でのコーディングスタイルを機械的に統一できます。

- ESLint: <https://eslint.org/>

## A.1.4 コードエディター

JavaScript や HTML、CSS などのコーディングに適したエディターを選ぶことで、開発の生産性を高められます。

### VSCode

**VSCode** は Microsoft 社がオープンソースで開発している無料のコードエディターです。JavaScript によってプラグインを書くことができ、さまざまな機能を追加できます。

- VSCode: <https://code.visualstudio.com/>

### Atom

**Atom** は GitHub 社がオープンソースで開発している無料のコードエディターです。VSCode と同様にプラグインによる拡張性が高く、GitHub と連携した機能が特徴です。

- Atom: <https://atom.io/>

## A.1.5 ブラウザの開発者ツール

多くのブラウザは開発者向けの組み込みツールを提供しており、本編で紹介したコンソールもその一部です。そのほかにも JavaScript コードをステップ実行できるデバッガーや、HTTP の通信ログなど、ブラウザごとにさまざまな機能があります。

- Firefox: [開発ツール | MDN](#)  
<https://developer.mozilla.org/ja/docs/Tools>
- Chrome: [Chrome DevTools](#)  
<https://developers.google.com/web/tools/chrome-devtools/?hl=ja>
- Safari: [Safari Developer Help](#)  
<https://support.apple.com/ja-jp/guide/safari-developer/welcome/mac>

## 付録 A 参考リンク集

### A.1.6 パフォーマンスの改善

ウェブサイトやウェブアプリケーションのパフォーマンスを計測、改善するためのツールを紹介します。

#### PageSpeed Insights

[PageSpeed Insights](#) は Google が提供するウェブパフォーマンス計測ツールです。計測したいページの URL を入力すると読み込みにかかっている時間や、改善できる項目を提示してくれます。

- PageSpeed Insights: <https://developers.google.com/speed/pagespeed/insights/>

#### WebPagetest

[WebPagetest](#) は、ブラウザを利用したウェブパフォーマンス計測ツールです。さまざまな条件下のブラウザでウェブサイトアクセスし、パフォーマンスを計測できます。BSD ライセンスの下でオープンソース化されており、任意のサーバーにインストールして実行することもできます。

- WebPagetest: <https://www.webpagetest.org/>

#### Lighthouse

[Lighthouse](#) は Google が提供するウェブページの分析ツールです。ウェブパフォーマンスだけでなく、アクセシビリティや SEO などの観点からも分析し、そのスコアを表示します。Chrome ブラウザの開発者ツールとして組み込まれていますが、npm でパッケージをインストールすれば CLI としても実行できます。

- Lighthouse: <https://developers.google.com/web/tools/lighthouse/?hl=ja>

## A.2 JavaScript の実行プラットフォーム

JavaScript はウェブサイトを作るための言語ではありません。いまでは多くのプラットフォームを超えた共通言語として、JavaScript やその周辺のエコシステムが発展しています。JavaScript を使ったプログラムを実行するためのいくつかのプラットフォームについて紹介します。

### A.2.1 ウェブサイトを公開する

ウェブサイトやウェブアプリケーションをインターネットに公開するためには、どこかのウェブサーバーでホスティング（公開）する必要があります。ここではホスティングを機能として提供し、簡単にウェブサイトを公開できるいくつかのホスティングサービスを紹介します。

#### GitHub Pages

[GitHub Pages](#) は、GitHub が提供する無料のホスティングサービスです。GitHub のリポジトリをウェブページとして公開して、リポジトリ内に配置した HTML や CSS、JavaScript などの静的ファイ

ルを配信できます。

- GitHub Pages: <https://pages.github.com/>

### Firebase Hosting

[Firebase Hosting](#) は、Google の Firebase プラットフォームが提供するホスティングサービスです。CLI を使ったシンプルなデプロイと、小規模の利用なら無料で利用できることが特徴です。

- Firebase Hosting: <https://firebase.google.com/products/hosting/?hl=ja>

### Netlify

[Netlify](#) も無料で利用できるホスティングサービスです。GitHub や BitBucket のような Git リポジトリサービスと連携していて、リモートリポジトリに push するだけで自動的にデプロイできるのが特徴です。

- Netlify: <https://www.netlify.com/>

## A.2.2 Node.js をサーバーレスに実行する

AWS Lambda や Google Cloud Functions のような [Function as a Service](#) (FaaS) と呼ばれる実行プラットフォームがあります。FaaS では Node.js のサーバーを用意しなくても関数単位で Node.js のスクリプトを実行できます。FaaS に JavaScript の関数をデプロイすると、クラウド上で管理されている Node.js サーバーにホストされ、それぞれの関数にエンドポイントが割り当てられます。

### AWS Lambda

[AWS Lambda](#) は Amazon Web Services 上で提供されるサーバーレス Node.js 実行環境です。

- AWS Lambda: <https://aws.amazon.com/jp/lambda/>

### Google Cloud Functions

[Google Cloud Functions](#) は Google Cloud Platform 上で提供されるサーバーレス Node.js 実行環境です。

- Google Cloud Functions: <https://cloud.google.com/functions/>

## A.2.3 デスクトップアプリケーションを作る

JavaScript を使って Windows や macOS、Linux などのデスクトップ環境で動作する GUI アプリケーションを作ることができます。

## 付録 A 参考リンク集

**Electron**

[Electron](#) は GitHub 社によって開発されているオープンソースのデスクトップアプリケーションフレームワークです。HTML や CSS、JavaScript を使ったウェブアプリケーションを Chromium ブラウザと一緒にパッケージ化して配布可能な実行ファイルを作成できます。

- Electron: <https://electronjs.org/>

**NW.js**

[NW.js](#) は Intel 社によって開発されているオープンソースのデスクトップアプリケーションフレームワークです。Electron と同様に Chromium ブラウザをベースにしたアプリケーションを開発できます。NW.js はブラウザの中から Node.js の開発エコシステムを直接利用できるようにしているのが特徴です。

- NW.js: <https://nwjs.io/>