

JavaScript Primer

迷わないのでの入門書

Iazu, Suguru Inatomi 著



ASCII
DWANGO

※本書の内容は、<https://jsprimer.net/> にて公開されています。

商標

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、実際にどう使われているかを知ることも目的にしています。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、「[第一部 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、「[第二部 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ためです。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^{*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

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が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っています。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。この書籍は、ECMAScript 2015 をベースにしつつ現時点の最新バージョンである ECMAScript book.esversion まで対応しています。

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 [asciidwango/js-primer](https://github.com/asciidwango/js-primer)*²で公開されています。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みます。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。

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/>

*² <https://github.com/asciidwango/js-primer>

*³ <https://jsprimer.net/>

はじめに

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。

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でも報告できます。

- 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はなく、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
- @tsin1rou
- 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。

変更点

初版からの変更点をまとめると次のようになります。

- ECMAScript の新しいバージョンである ES2020、ES2021、ES2022 に対応した
- 新しい ECMAScript の機能によって、使う必要がなくなった機能は非推奨へと変更した
- 文字では想像しにくいビット演算、非同期処理などに図を追加した
- Promise と Async Function を非同期の処理の中心として書き直した
- 一方で、エラーファーストコールバックは非同期処理としてはメインではなくなった
- `Array#includes` という表記は、Private Class Fields(`#field`) と記号が被るため廃止した
- Node.js でも ECMAScript Modules を使うようになり、CommonJS はメインではなくなった
- Node.js が 12 から 18 までアップデートし、npm は 6 から 8 までアップデート、各種ライブラリも最新にアップデート
- 読者からのフィードバックを受けて、全体をより分かりやすく読みやすくなるように書き直した

ECMAScript はアップデートにより、機能的が利用できなくなるという変更はほぼありません。その点では、初版で紹介した JavaScript は現在でも動作します。

一方で、実際の利用のされ方などの状況を見て、使われなくなっていく機能はあります。そのため、この書籍では古くなった機能は、何によって置き換えたのかも解説しています。

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オープンソース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を続けている。

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

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

1.2 JavaScript ってどのような言語？

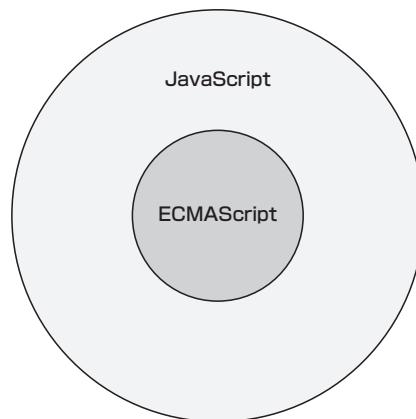


図 1.1 JavaScript と ECMAScript の範囲

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 応用編（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;
```

空白文字の置き方は人によって好みが異なるため、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いいでしょ。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";
// このコードは strict mode で実行される
```

^{*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？

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的に例外を投げることで、開発者が間違いに気づきやすくしてくれます。

たとえば、次のような `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つ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きます。

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 */
```

2.3 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}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増えるため、それを学び続けることには変わりありません。

2.4 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 Primer" という文字列であることを定義しています。

```
const bookTitle = "JavaScript Primer";
```

`const`、`let`、`var` のキーワードも共通の仕組みですが、変数同士を、（カンマ）で区切ることにより、同時に複数の変数を定義できます。

次のコードでは、`bookTitle` と `bookCategory` という変数を順番に定義しています。

```
const bookTitle = "JavaScript Primer",
      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 Primer";
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 Primer";
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ない場合は、`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 Primer" という文字列であることを定義しています。

```
let bookTitle = "JavaScript Primer";
```

`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 Primer" という文字列を代入しています。

```
let bookTitle;
bookTitle = "JavaScript Primer";
```

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 Primer";
```

`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 Primer";
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と、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、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

3.4 変数名に使える名前のルール

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、アルファベットの大文字と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くことがあるためお勧めしません。

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

```
let es2015; // OK: 数字は先頭以外なら利用できる
let 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。

```
let 1st; // NG: 数字から始まっている
let 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、`let` 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。

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

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。

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

3.5 まとめ

この章では、JavaScriptにおける変数を宣言するキーワードとして `const`、`let`、`var`があることに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

変数宣言をする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ない

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/new/>

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

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”ブラウザツール”のサブメニューから”ウェブ開発ツール”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”^{*1}を参照してください。



図 4.1 Firefox でウェブコンソールを開いた状態

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>>からはじまる欄）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 という値を入力し Enter キーを押すと、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` という入力をすると、先ほど変数に入れた "`JavaScript Primer`" という結果が次の行に表示されます。

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

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 Primer";
undefined
>> const bookTitle = "JavaScript Primer";
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 [Visual Studio Code](https://code.visualstudio.com/)*2 などの JavaScript に対応したエディターを用意しておくとスムーズです。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

*2 <https://code.visualstudio.com/>

4.2 ブラウザで JavaScript を実行する

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。

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（（と）の間に書かれたもの）の式を評価してから、その結果をコンソールに表示します。そのため、`1 + 1` の評価結果として `2` がコンソールに表示されます。

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;
```

4.4 ウェブ版の書籍でコードを実行する

```
// 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起きません。

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（`SyntaxError`）は書かれたコードの文法が間違っている場合に発生するエラーです。

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

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

index.js

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

第4章 値の評価と表示

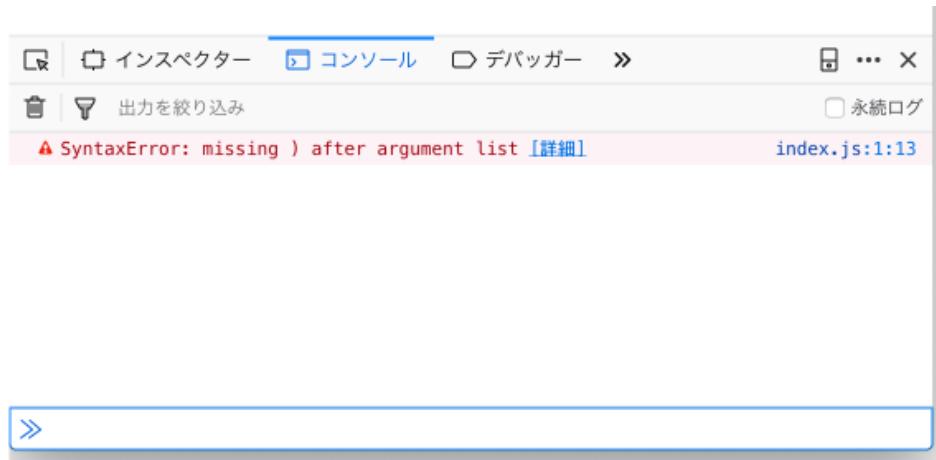


図4.2 コンソールに表示された SyntaxError

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ト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

|

| 行番号:列番号

エラー内容の説明

| ファイル名

メッセージ	意味
SyntaxError: missing) after argument list	エラーの種類は SyntaxError で、関数呼び出しの) が足りない
index.js:1:13	例外が index.js 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は表示されます。この [詳細] リンクはエラーメッセージに関する MDN の解説ページへのリンクとなっています。この例のエラーメッセージでは次の解説ページへリンクされています。

- https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Errors/Missing_parenthesis_after_argument_list

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);
```

4.5 コードの評価とエラー

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	
	<code>cosnt a = 1;</code>
<hr/>	
	<code>SyntaxError: unexpected token: identifier[詳細] index.js:1:6</code>
メッセージ	意味
<code>SyntaxError: unexpected token: identifier</code>	エラーの種類は <code>SyntaxError</code> で、予期しない識別子（変数名 <code>a</code> ）が指定されている
<code>index.js:1:6</code>	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と、`cosnt` (`const` のタイプミス) はキーワードではないため、ただの変数名として解釈されます。そのため、このコードは次のよう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是必ずしも一致しません。しかし、構文エラーの原因是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て、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があります。API に渡す値のデータ型の問題から起きた `TypeError` や存在しない変数を参照しようとして起きた `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 ReferenceError で、x 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 index.js 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、エラーが発生した行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ことが大切です。

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

4.6 まとめ

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^{*3}](#)にまとめられています。また[Google^{*4}](#)、[GitHub^{*5}](#)、[Stack Overflow^{*6}](#)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。

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

^{*3} <https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Errors>
^{*4} <https://www.google.com/>
^{*5} <https://github.com/>
^{*6} <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 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 mutable 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 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、7 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
 - 巨大な整数 (BigInt) : ES2020 から追加された `9007199254740992n` などの任意精度の整数のデータ型
 - 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变な値のデータ型
- オブジェクト（複合型）
 - プリミティブ型以外のデータ
 - オブジェクト、配列、関数、クラス、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 9007199254740992n); // => "bigint"
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"
```

プリミティブ型の値は、それぞれ `typeof` 演算子の評価結果として、その値のデータ型を返します。一方で、オブジェクトに分類される値は"object"となります。

配列 (`[]`) とオブジェクト () は、どちらも"object"という判定結果になります。そのため、`typeof` 演算子ではオブジェクトの詳細な種類を正しく判定することはできません。ただし、関数はオブジェクトの中でも特別扱いされているため、`typeof` 演算子の評価結果は"function"となります。また、`typeof null` が"object"となるのは、歴史的経緯のある仕様のバグ^{*1}です。

このことからもわかるように `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。`typeof` 演算子では、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ていきます。

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データ型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。

^{*1} JavaScript が最初に Netscape で実装された際に `typeof null === "object"` となるバグがありました。このバグを修正するとすでにこの挙動に依存しているコードが壊れるため、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

- 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。

- 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、見た目どおりの意味となります。

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

5.2.2 数値 (Number)

数値には 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。

これらのリテラルで表現できる数値は [IEEE 754](#) の倍精度浮動小数として扱われます。倍精度浮動小数では 64 ビットで数値を表現します。64 ビットのうち 52 ビットを数字の格納のために使い、11 ビットを小数点の位置に使い、残りの 1 ビットはプラスとマイナスの符号です。そのため、正確に扱える数値の最大値は $2^{53} - 1$ (2 の 53 乗から 1 引いた値) となります。

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(0b1000000000); // => 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（詳細は「[JavaScript とは](#)」の「strict mode」を参照）。

```
// 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

第5章 データ型とリテラル

浮動小数点数リテラル

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

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 2×10^8 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 BigInt ES2020

JavaScript では、1 や 3.14159 などの数値リテラルは [IEEE 754](#) で定義された倍精度浮動小数となります。倍精度浮動小数で正確に扱える数値の最大値は $2^{53} - 1$ (2 の 53 乗から 1 引いた値である 9007199254740991) です。この数値リテラルで安全に表せる最大の数値は `Number.MAX_SAFE_INTEGER` として定義されています。

```
console.log(Number.MAX_SAFE_INTEGER); // => 9007199254740991
```

数値リテラルで $2^{53} - 1$ (9007199254740991) よりも大きな値を表現したり計算すると間違った結果となる場合があります。

この問題を解決するために、ES2020 では `BigInt` という新しい整数型のデータ型とリテラルが追加されました。数値リテラルは倍精度浮動小数 (64 ビット) で数値を扱うのに対して、`BigInt` では任意の精度の整数を扱えます。そのため、`BigInt` では $2^{53} - 1$ (9007199254740991) よりも大きな整数を正しく表現できます。

`BigInt` リテラルは、数値の後ろに `n` をつけます。

```
console.log(1n); // => 1n
// 2^53-1 より大きな値も扱える
console.log(9007199254740992n); // => 9007199254740992n
```

`BigInt` は整数を扱うデータ型であるため、次のように小数点を含めた場合は構文エラーとなります。

```
1.2n; // => SyntaxError
```

5.2.4 Numeric Separators ES2021

数値が大きくなるほど、桁数の見間違いなどが発生しやすくなります。次のコードは、1兆を数値リテラルで書いていますが、桁数を読み取りにくいです。

```
1000000000000;
```

ES2021 から、数値リテラル内の区切り文字として`_`を追加できる Numeric Separators がサポートされています。Numeric Separators は、数値リテラル内では区切り文字として`_`が追加できます。次のコードも、1兆を数値リテラルで書いています。数値リテラルを評価する際に`_`は単純に無視されるため同じ意味となります。

```
1_000_000_000_000;
```

Numeric Separators は数値リテラルである整数、浮動小数点、BigInt のリテラル内でのみ利用できます。また、`_`はリテラルの先頭や数値の最後に追加することはできません。

```
_123; // 変数として評価される
3._14; // => SyntaxError
0x52_; // => SyntaxError
1234n_; // => SyntaxError
```

5.2.5 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

第5章 データ型とリテラル

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`"
```

5.2.6 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
```

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
    const 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。

5.2.7 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中の基礎となります。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 = {};
```

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できます。

第5章 データ型とリテラル

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" という文字列の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` プロパティを参照するには、. (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"
const 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。

5.2.8 配列リテラル

オブジェクトリテラルと並んで、よく使われるリテラルとして配列リテラルがあります。配列リテラルは [と] で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。

5.3 プリミティブ型とオブジェクト

```
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"
// 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。

5.2.9 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 / (スラッシュ) と / (スラッシュ) 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、+などの特定の記号や \ (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
// numberRegExp の正規表現が文字列 "123" 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
```

第5章 データ型とリテラル

```
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。

- 7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
左オペランド 演算子 右オペランド
```

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、演算子をひととおり見た後に、暗黙的な型変換と明示的な型変換について学んでいきます。

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」の章を参照）。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、2つの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します。

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。左オペランドから右オペランドを減算した値を返します。

```
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。左オペランドを右オペランドで除算した値を返します。

6.2 単項演算子（算術）

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

6.1.6 剰余演算子 (%)

2つの数値のあまりを求める演算子です。左オペランドを右オペランドで除算したあまりを返します。

```
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
```

第6章 演算子

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
```

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--;

```

第6章 演算子

```
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、オペラント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ト同士の値を比較し、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つのオペラントを比較します。同じ型で同じ値である場合に、`true` を返します。

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

また、オペラントがどちらもオブジェクトであるときは、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
```

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

第6章 演算子

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
```

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

ビット演算子では、オペランドである数値を符号付き 32 ビット整数（0 と 1 からなる 32 個のビットの集合）として扱います。

たとえば、1 という数値は符号付き 32 ビット整数のビットでは、00000000000000000000000000000001 として表現されます。わかりやすく 4 ビットごとに区切ると 0000_0000_0000_0000_0000_0001

第 6 章 演算子

のような 32 ビットの集合となります。符号付き 32 ビット整数では、先頭の最上位ビット（一番左のビット）は符号を表し、0 の場合は正の値、1 の場合は負の値であることを示しています。

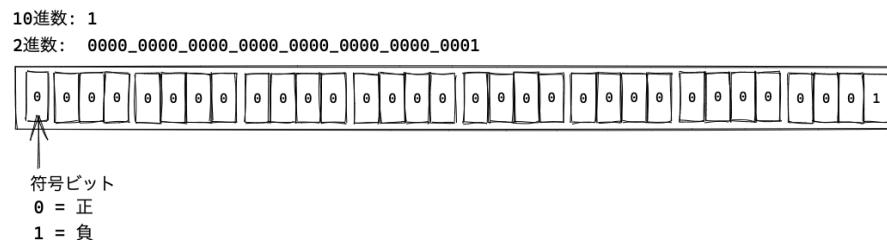


図 6.1 1 の符号付き 32bit 整数での表現

符号付き 32 ビット整数では負の数値は、2 の補数形式という形式で表現されます。2 の補数とは、それぞれのビットを反転して 1 ビットを足した値となります。

たとえば、-1 という数値の符号付き 32 ビット整数は、次のように 2 の補数で求められます。

- 10 進数の 1 は、符号付き 32 ビット整数では 0000_0000_0000_0000_0000_0000_0001 となる
- 0000_0000_0000_0000_0000_0000_0001 の各ビットを反転すると 1111_1111_1111_1111_1111_1111_1111 となる
- これに 1 ビットを足すと 1111_1111_1111_1111_1111_1111_1111 となる

これによって、-1 の符号付き 32 ビット整数は 1111_1111_1111_1111_1111_1111_1111 となります。

符号付き 32 ビット整数で表現できる数値の範囲は、1000_0000_0000_0000_0000_0000_0000 から 0111_1111_1111_1111_1111_1111_1111 までとなります。10 進数に直すと $-(2^{31})$ (2 の 31 乗の負の数) から $(2^{31}) - 1$ (2 の 31 乗から 1 引いた数) までとなります。32 ビットを超える数値については、32 ビットをはみ出るビットが最上位（一番左）から順番に捨てられます。

これから見ていくビット演算子はオペランドを符号付き 32 ビット整数として扱い、その演算結果を 10 進数の数値として返します。

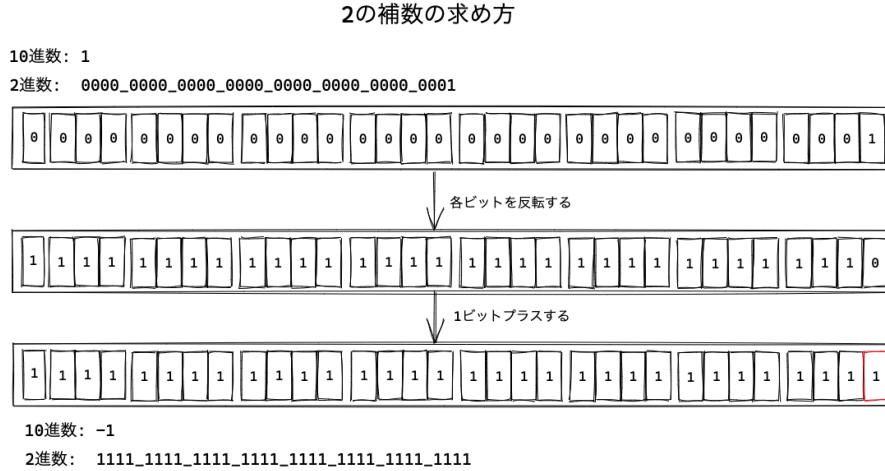


図 6.2 -1 の符号付き 32 ビット整数での表現

6.4.1 ビット論理積 (&)

ビット論理積演算子 (`&`) はビットごとの AND 演算した結果を返します。AND 演算では、オペランドの各ビットがどちらも 1 の場合は 1 となり、それ以外の場合は 0 となります。

次のコードでは、10 進数の 15 と 9 を AND 演算しています。15 は、符号付き 32 ビット整数では 0000_0000_0000_0000_0000_0000_1111 となります。9 は、符号付き 32 ビット整数では 0000_0000_0000_0000_0000_0000_1001 となります。これらを AND 演算した結果は 0000_0000_0000_0000_0000_0000_1001 となり、10 進数の値である 9 を返します。

```
console.log(15      & 9);      // => 9
// 同じ位の各ビット同士をAND演算する（上位の0は省略）
// 1111
// 1001
// -----
// 1001
console.log(0b1111 & 0b1001); // => 0b1001
```

第 6 章 演算子

6.4.2 ビット論理和 (|)

ビット論理和演算子 (|) はビットごとの **OR** 演算した結果を返します。OR 演算では、オペランドの各ビットがどちらか片方でも 1 の場合は 1 となり、両方とも 0 の場合は 0 となります。

```
console.log(15      | 9);      // => 15
// 同じ位の各ビット同士をOR演算する（上位の0は省略）
// 1111
// 1001
// -----
// 1111
console.log(0b1111 | 0b1001); // => 0b1111
```

6.4.3 ビット排他的論理和 (^)

ビット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。XOR 演算では、オペランドのビットが異なるなら 1、両方とも同じなら 0 となります。

```
console.log(15      ^ 9);      // => 6
// 同じ位の各ビット同士をXOR演算する（上位の0は省略）
// 1111
// 1001
// -----
// 0110
console.log(0b1111 ^ 0b1001); // => 0b0110
```

6.4.4 ビット否定 (~)

単項演算子の否定演算子 (~) はオペランドの各ビット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。

次のコードでは、10 進数で 15 を否定演算子 (~) で各ビットを反転させた値を得ています。15 は 0000_0000_0000_0000_0000_0000_1111 です。各ビットを反転させると 1111_1111_1111_1111_1111_1111_0000 となり、10 進数では -16 となります。

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

$\sim x$ のように x をビット否定演算子で演算した結果は、 $-(x + 1)$ となります。この性質を利用する形で、ビット否定演算子 (~) はビット演算以外で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 = "森森木森森";
// 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。次のコードのように否定演算子 (~) 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

6.4.5 左シフト演算子 (<<)

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

```
num << bit;
```

次のコードでは、9 を 2 ビット分だけ左へシフトしています。

```
console.log(      9 << 2); // => 36
console.log(0b1111 << 2); // => 0b11_1100
```

6.4.6 右シフト演算子 (>>)

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

```
num >> bit;
```

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

```
console.log((-9) >> 2); // => -3
// 1111_1111_1111_1111_1111_1111_0111 >> 2
// => 1111_1111_1111_1111_1111_1111_1101
```

6.4.7 ゼロ埋め右シフト演算子 (>>>)

ゼロ埋め右シフト演算子は、数値である num を bit の数だけ右へシフトするのは右シフト演算子(>>)と同じです。異なる点としては右にあふれたビットは破棄され、0 のビットを左から詰めます。

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

```
console.log((-9) >>> 2); // => 1073741821
// 1111_1111_1111_1111_1111_1111_0111 >>> 2
// => 0011_1111_1111_1111_1111_1111_1101
```

6.5 代入演算子 (=)

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

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

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

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

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"
```

第6章 演算子

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 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 (かつ)、OR (または)、NOT (否定) を表現できます。

6.6.1 AND 演算子 (`&&`)

AND 演算子 (`&&`) は、左辺の値の評価結果が `true` ならば、右辺の評価結果を返します。一方で、左辺の値の評価結果が `false` ならば、そのまま左辺の値を返します。

```
// 左辺は true であるため、右辺の評価結果を返す
console.log(true && "右辺の値"); // => "右辺の値"
// 左辺が false であるなら、その時点で false を返す
// 右辺は評価されない
console.log(false && "右辺の値"); // => false
```

AND 演算子 (`&&`) は、左辺の評価が `false` の場合、オペランドの右辺は評価されません。次のように、左辺が `false` の場合は、右辺に書いた `console.log` 関数自体が実行されません。

```
// 左辺が true なので、右辺は評価される
true && console.log("このコンソールログは実行されます");
// 左辺が false なので、右辺は評価されない
false && console.log("このコンソールログは実行されません");
```

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** と呼びます。

また、AND 演算子は左辺を評価する際に、左辺を真偽値へと **暗黙的な型変換** をしてから判定します。真偽値への暗黙的な型変換では、次に挙げる値は `false` へ変換されます。

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

暗黙的な型変換によって `false` に変換されるこれらの値をまとめて **falsy** な値と呼びます。`falsy` ではない値は、`true` へと変換されます。`true` へと変換される値の種類は多いため、`false` へと変換されない値は `true` となることは覚えておくとよいです。このオペランドを真偽値に変換してから評価するのは AND、OR、NOT 演算子で共通の動作です。

次のように、AND 演算子 (`&&`) は左辺を真偽値へと変換した結果が `true` の場合に、右辺の評価結果を返します。つまり、左辺が `falsy` の場合は、右辺は評価されません。

```
// 左辺は falsy ではないため、評価結果として右辺を返す
console.log("文字列" && "右辺の値"); // => "右辺の値"
console.log(42 && "右辺の値"); // => "右辺の値"
// 左辺が falsy であるため、評価結果として左辺を返す
console.log("") && "右辺の値"); // => ""
console.log(0 && "右辺の値"); // => 0
console.log(null && "右辺の値"); // => null
```

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 if 文の条件式は `false` となります。そのため、`value` が String 型ではない場合は、AND 演算子 (`&&`) の右辺は評価されずに、if 文の中身も実行されません。

AND 演算子 (`&&`) を使うと、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複雑なのと同様に、AND 演算子や OR 演算子が 3 つ 4 つ連続すると複雑で読みにくいコードとなります。その場合は抽象化ができるいかを検討するべきサインとなります。

6.6.2 OR 演算子 (||)

OR 演算子 (||) は、左辺の値の評価結果が `true` ならば、そのまま左辺の値を返します。一方で、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。

第 6 章 演算子

```
// 左辺が true のので、左辺の値が返される
console.log(true || "右辺の値"); // => true
// 左辺が false のので、右辺の値が返される
console.log(false || "右辺の値"); // => "右辺の値"
```

OR 演算子 (`||`) は、左辺の評価が `true` の場合、オペランドの右辺を評価しません。これは、AND 演算子 (`&&`) と同様の短絡評価となるためです。

```
// 左辺が true のので、右辺は評価されない
true || console.log("このコンソールログは実行されません");
// 左辺が false のので、右辺は評価される
false || console.log("このコンソールログは実行されます");
```

また、OR 演算子は左辺を評価する際に、左辺を真偽値へと暗黙的な型変換します。次のように、OR 演算子は左辺が `falsy` の場合には右辺の値を返します。

```
// 左辺が falsy のので、右辺の値が返される
console.log(0 || "左辺は falsy"); // => "左辺は falsy"
console.log("") || "左辺は falsy"; // => "左辺は falsy"
console.log(null || "左辺は falsy"); // => "左辺は falsy"
// 左辺は falsy ではないため、左辺の値が返される
console.log(42 || "右辺の値"); // => 42
console.log("文字列" || "右辺の値"); // => "文字列"
```

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。");
}
```

6.6.3 NOT 演算子 (!)

NOT 演算子 (`!`) は、オペランドの評価結果が `true` ならば、`false` を返します。一方で、オペランドの評価結果が `false` ならば、`true` を返します。つまり、オペランドの評価結果を反転した真偽値を返します。

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

NOT 演算子 (`!`) も AND 演算子 (`&&`) と OR 演算子 (`||`) と同様に真偽値へと暗黙的な型変換し

6.7 Nullish coalescing 演算子 (??) ES2020

ます。falsy である値は `true` へ変換され、falsy ではない値は `false` へと変換されます。

```
// falsy な値は true となる
console.log(!0); // => true
console.log(! ""); // => true
console.log(!null); // => true
// falsy ではない値は false となる
console.log(!42); // => false
console.log(!"文字列"); // => 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。たとえば、`!!falsy` な値のように 2 度反転すれば `false` になります。

```
const str = "";
// 空文字列は falsy であるため、true -> false へと変換される
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（長さが 0 より大きな文字列）でないことを判定
console.log(str.length > 0); // => false
```

6.7 Nullish coalescing 演算子 (??) ES2020

Nullish coalescing 演算子 (??) は、左辺の値が `nullish` であるならば、右辺の評価結果を返します。`nullish` とは、評価結果が `null` または `undefined` となる値のことです。

```
// 左辺が nullish であるため、右辺の値の評価結果を返す
console.log(null ?? "右辺の値"); // => "右辺の値"
console.log(undefined ?? "右辺の値"); // => "右辺の値"
// 左辺が nullish ではないため、左辺の値の評価結果を返す
console.log(true ?? "右辺の値"); // => true
console.log(false ?? "右辺の値"); // => false
console.log(0 ?? "右辺の値"); // => 0
console.log("文字列" ?? "右辺の値"); // => "文字列"
```

Nullish coalescing 演算子 (??) と OR 演算子 (`||`) は、値のデフォルト値を指定する場合によく利用されています。OR 演算子 (`||`) は左辺が falsy の場合に右辺を評価するため、意図しない結果となる場合が知られています。

次のコードは、`inputValue` が未定義だった場合に、`value` に対するデフォルト値を OR 演算子 (`||`)

第 6 章 演算子

で指定しています。`inputValue` が未定義 (`undefined`) の場合は、意図したように OR 演算子 (`||`) の右辺で指定した `42` が入ります。しかし、`inputValue` が `0` という値であった場合は、`0` は falsy であるため `value` には右辺の `42` が入ります。これでは `0` という値が扱えないため、意図しない動作となっています。

```
const inputValue = 任意の値または未定義;
// inputValue が falsy の場合は、value には 42 が入る
// inputValue が 0 の場合は、value に 42 が入ってしまう
const value = inputValue || 42;
console.log(value);
```

この問題を解決するために ES2020 で Nullish coalescing 演算子 (`??`) が導入されています。

Nullish coalescing 演算子 (`??`) では、左辺が nullish の場合のみ、`value` に右辺で指定した `42` が入ります。そのため、`inputValue` が `0` という値が入った場合は、`value` にはそのまま `inputValue` の値である `0` が入ります。

```
const inputValue = 任意の値または未定義;
// inputValue が nullish の場合は、value には 42 が入る
// inputValue が 0 の場合は、value に 0 が入る
const value = inputValue ?? 42;
console.log(value);
```

6.8 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。

6.8 条件（三項）演算子（?と:）

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

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

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

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

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

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

6.8.1 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^{*1}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。

^{*1} https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Operators/Operator_Precendence#Table

第6章 演算子

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するよりも、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。

```
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。

6.9 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^{*2}。

6.10 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
- 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

^{*2} 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せずに、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、比較するためです。

```
// ==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
```

7.1 暗黙的な型変換とは

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、次のことを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。

第 7 章 暗黙的な型変換

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演算子 (-) の定義はありません。そのため、マイナス演算子の対象とな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するのが難しくなります。

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。

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、次の値は `false` へ変換されます。

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

暗黙的な型変換によって `false` に変換されるこれらの値をまとめて **falsy** な値と呼びます。`falsy` ではない値は、`true` へと変換されます。

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
```

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。

7.2 明示的な型変換

```
"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

第 7 章 暗黙的な型変換

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("パースした結果 Nan になった", 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^{*1}](#) 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) {
    // Nan は自分自身と一致しない
    return x !== x;
```

^{*1} https://ja.wikipedia.org/wiki/IEEE_754

7.2 明示的な型変換

```
}
```

```
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
```

第7章 暗黙的な型変換

```

        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 型の値以外を受けつけなくする
- 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取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^{*2}](#) 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);
}

const x = 1, z = 10;
let y; // y は undefined

```

^{*2} <https://jsdoc.app/>

7.3 明示的な変換でも解決しないこと

```
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` になってしまふと `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が、`0` も空文字列となってしまい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。

第 7 章 暗黙的な型変換

この場合、空文字列とは「String 型で文字長が 0 の値」であると定義することで、`isEmptyString` 関数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、明示的な型変換をする関数を利用する
- 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` からはじまる文は **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（3章の「[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 — {と}で囲んだ関数の処理を書く場所
- 関数の返り値 — 関数を呼び出したときに、呼び出し元へ返される値

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

第8章 関数と宣言

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 `double` という関数を定義しています。`double` 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 `return` 文で返しています。

```
function double(num) {
    return num * 2;
}
// double 関数の返り値は、num に 10 を入れて return 文で返した値
console.log(double(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` 文では返り値を省略できます。`return` 文の返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() {
    // 何も返り値を指定しない場合は 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しょう。

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。

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

8.2 関数の引数

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"
```

第8章 関数と宣言

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（「暗黙的な型変換」の章を参照）。

- `false`
- `undefined`
- `null`
- `0`
- `On`
- `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ことがあります。

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

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

デフォルト引数を使って書くことで、このような挙動は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;
}
```

```
// falsy 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"
```

また、ES2020 から導入された Nullish coalescing 演算子 (??) を利用することでも、OR 演算子 (||) の問題を避けつつデフォルト値を指定できます。

```
function addPrefix(text, prefix) {
    // prefix が null または undefined の時、デフォルト値を返す
    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"
// falsy 値でも意図通りに動作する
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);
```

第8章 関数と宣言

```
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、渡された引数が入った配列
  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);
```

8.4 関数の引数と分割代入

```
// 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() {
  // 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
```

第8章 関数と宣言

```

    }
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。

```
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。

第8章 関数と宣言

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

8.5 関数はオブジェクト

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
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
```

第8章 関数と宣言

```
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つきりしています。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いても、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、関数を上書きしてしまう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。

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。一方で、`const` や `let` 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。

```
const fn = (x) => {
    return `最初の関数 x: ${x}`;
};
// const は同じ変数名を定義できないため、構文エラーとなる
```

```
const fn = (x, y) => {
  return `最後の関数 x: ${x}, y: ${y}`;
};
```

関数の上書きを避けたい場合は、`const` と関数式を使って関数を定義することで、意図しない上書きが発生しにくくなります。

8.6 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);
// 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。

8.7 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"
```

8.7.1 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に書く方法となっています。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。

8.8 まとめ

この章では、次のことを学びました。

- 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れません。

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) になります。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;
```

式は文になりますが、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。

第9章 文と式

```
{
    文;
    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);
}
```

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

9.3 function 宣言（文）と function 式

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。

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。

「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る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^{*1}](#)などの Lint ツールを使うことで、セミコロンが必要なのかをチェックできます。

セミコロンが必要か見分けるにはある程度慣れが必要ですが、ツールを使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 つのみの場合は、{}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、これらは **falsy** な値と呼ばれます（「[暗黙的な型変換](#)」の章を参照）。

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

`falsy` ではない値は、`true` へと変換されます。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 output 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;
if (num > 10) {

```

第10章 条件分岐

```
    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しまっています。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":
        console.log("ECMAScript 5");
```

第 10 章 条件分岐

```

        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値演算子 (===) 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;` は省略が可能ですが、省略した場合、後ろに続く case 節が条件に関係なく実行されます。

```
const version = "ES6";
switch (version) {
    case "ES5":
        console.log("ECMAScript 5");
    case "ES6": // 一致するケース
        console.log("ECMAScript 2015");
    case "ES7": // break されないため条件無視して実行
        console.log("ECMAScript 2016");
    default: // break されないため条件無視して実行
        console.log("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) {
    switch (version) {
```

第 10 章 条件分岐

```
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 節を組み合わせて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` となってしまうため、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に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仮引数である `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 には 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 の中に使い、処理中のループを抜けて次の文へ制御を移します。

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

switch 文で出てきたものと同様で、処理中のループ文を終了できます。

次のコードでは配列の要素に 1 つでも偶数を含んでいるかを判定しています。

```
const numbers = [1, 5, 10, 15, 20];
// 偶数があるかどうか
let isEvenIncluded = false;
for (let i = 0; i < numbers.length; i++) {
  const 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ればいいため、配列内から最初の偶数を見つけたら for 文での反復処理を終了します。このような処理は、使い回せるよう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に、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` の中に使えます。

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、新しい配列を作り返しています。偶数ではない場合、処理中の `for` 文をスキップします。

```
// 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つてコードが読みにくくなります。そのため、10 章の「[ネストした if 文](#)」のうるう年の例でも紹介したように、できるだけ早い段階でそれ以上処理を続けない宣言をして、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、反復処理を行います^{*1}。

```
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"
```

^{*1} `for...in` 文がプロパティを列挙する順番は ES2019までは実装依存でしたが、ES2020で列挙する順番が決められました。

第 11 章 ループと反復処理

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

JavaScript では、オブジェクトは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

第 11 章 ループと反復処理

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、実行環境にあらかじめ定義されている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）はクオート（"や'）を省略できます。そのため、次のように書いても同じです。

```
// プロパティ名（キー）はクオートを省略することが可能
```

第12章 オブジェクト

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

ただし、変数名として利用できないプロパティ名はクオート（"や'）で囲む必要があります。次の `my-prop` というプロパティ名は、変数名として利用できない-が含まれているため定義できません（「[変数と宣言](#)」の章の「[変数名に使える名前のルール](#)」を参照）。

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

`my-prop` というプロパティ名を定義する場合は、クオート（"や'）で囲む必要があります。

```
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。

12.2 プロパティへのアクセス

```
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
};
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
```

第12章 オブジェクト

```
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](#)」の章の「[変数名に使える名前のルール](#)」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
// プロパティ名は暗黙的に文字列に変換されているため、次も同じプロパティを参照している
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い場合はブ

12.3 オブジェクトと分割代入

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と冗長となりやすいです。そのため、短い名前で利用できるように、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

第 12 章 オブジェクト

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
- 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します。

12.4 プロパティの追加

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。

```
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" }
```

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

第 12 章 オブジェクト

の変更は防げません（3 章の「`const`」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する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（詳
細は「[JavaScript とは](#)」の `strict mode` を参照）。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純に無視されます。

```
"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
```

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 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
```

12.5 プロパティの存在を確認する

```
// 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
```

`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4つがあります。

- `undefined` との比較
- `in` 演算子
- `Object.hasOwn` 静的メソッド **ES2022**
- `Object.prototype.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 `Object.hasOwn` 静的メソッドを利用します。

第 12 章 オブジェクト

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し真偽値を返します。

```
"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: Object.hasOwn 静的メソッド ES2022

`Object.hasOwn` 静的メソッドは、対象のオブジェクト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 `Object.hasOwn` 静的メソッドの引数には、オブジェクトとオブジェクトが持っているかを確認したいプロパティ名を渡します。

```
const obj = {};
// obj が"プロパティ名"を持っているかを確認する
Object.hasOwn(obj, "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。`Object.hasOwn` 静的メソッドも、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: undefined };
// obj が key プロパティを持っているなら true となる
if (Object.hasOwn(obj, "key")) {
    console.log(`obj`は`key`プロパティを持っている`);
}
```

`in` 演算子と `Object.hasOwn` 静的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。そのため、`in` 演算子と `Object.hasOwn` 静的メソッドの違いについては、次の章の「[プロトタイプオブジェクト](#)」で詳しく解説します。

12.6 プロパティの存在確認: `Object.prototype.hasOwnProperty` メソッド

12.6 プロパティの存在確認: `Object.prototype.hasOwnProperty` メソッド

`Object.hasOwn` 静的メソッドは ES2022 で導入されたメソッドです。ES2022 より前では、`Object.prototype.hasOwnProperty` メソッドというよく似たメソッドが利用されていました。`hasOwnProperty` メソッドは、`Object.hasOwn` 静的メソッドとよく似ていますが、オブジェクトのインスタンスから呼び出す点が異なります。

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

しかし、`hasOwnProperty` メソッドには欠点があるため、`Object.hasOwn` 静的メソッドが利用できる状況では使う理由はありません。この欠点もプロトタイプオブジェクトに関するため、次の章の「[プロトタイプオブジェクト](#)」で詳しく解説します。

12.7 Optional chaining 演算子 (?.) ES2020

プロパティの存在を確認する方法として 4 つの方法を紹介しました。プロパティが存在するかが重要な場合は、基本的には `in` 演算子または `Object.hasOwn` 静的メソッドを使います。

しかし、最終的に取得したいものがプロパティの値であるならば、`if` 文で `undefined` と比較しても問題ありません。なぜなら、値を取得したい場合には、プロパティが存在するかどうかとプロパティの値が `undefined` かどうかの違いを区別する意味はないためです。

次のコードでは、`widget.window.title` プロパティに値が定義されているなら (`undefined` ではないなら)、そのプロパティの値をコンソールに表示しています。

```
function printWidgetTitle(widget) {
    // 例外を避けるために widget のプロパティの存在を順番に確認してから、値を表示している
    if (widget.window !== undefined && widget.window.title !== undefined) {
        console.log(`ウィジェットのタイトルは ${widget.window.title} です`);
    } else {
        console.log("ウィジェットのタイトルは未定義です");
    }
}
// タイトルが定義されている widget
printWidgetTitle({
    window: {
        title: "Book Viewer"
    }
})
```

第 12 章 オブジェクト

```

    }
});

// タイトルが未定義のwidget
printWidgetTitle({
    // タイトルが定義されてない空のオブジェクト
});

```

この `widget.window.title` のようなネストしたプロパティにアクセスする際には、プロパティの存在を順番に確認してからアクセスする必要があります。なぜなら、`widget` オブジェクトが `window` プロパティを持っていない場合は `undefined` という値を返すためです。このときに、さらにネストした `widget.window.title` プロパティにアクセスすると、`undefined.title` という参照となり例外が発生してしまいます。

しかし、プロパティへアクセスするたびに `undefined` との比較を AND 演算子 (`&&`) でつなげて書いていくと冗長です。

この問題を解決するために、ES2020 ではネストしたプロパティの存在確認とアクセスを簡単に行う構文として Optional chaining 演算子 (`?.`) が導入されました。Optional chaining 演算子 (`?.`) は、ドット記法 (`.`) の代わりに `?.` をプロパティアクセスに使います。

Optional chaining 演算子 (`?.`) は左辺のオペランドが nullish (`null` または `undefined`) の場合は、それ以上評価せずに `undefined` を返します。一方で、プロパティが存在する場合は、そのプロパティの評価結果を返します。

つまり、Optional chaining 演算子 (`?.`) では、存在しないプロパティへアクセスした場合でも例外ではなく、`undefined` という値を返します。

```

const obj = {
    a: {
        b: "obj の a プロパティの b プロパティ"
    }
};

// obj.a.b は存在するので、その評価結果を返す
console.log(obj?.a?.b); // => "obj の a プロパティの b プロパティ"

// 存在しないプロパティのネストもundefined を返す
// ドット記法の場合は例外が発生してしまう
console.log(obj?.notFound?.notFound); // => undefined
// undefined や null は nullish なので、undefined を返す
console.log(undefined?.notFound?.notFound); // => undefined
console.log(null?.notFound?.notFound); // => undefined

```

先ほどのウィジェットのタイトルを表示する関数も Optional chaining 演算子 (`?.`) を使うと、if 文を使わずに書けます。次のコードの `printWidgetTitle` 関数では、`widget?.window?.title` にアクセスできる場合はその評価結果が変数 `title` に入ります。プロパティにアクセスできない場合は `undefined` を返すため、Nullish coalescing 演算子 (`??`) によって右辺の"未定義"が変数 `title` のデ

12.8 `toString` メソッド

フォルト値となります。

```
function printWidgetTitle(widget) {
    const title = widget?.window?.title ?? "未定義";
    console.log(`ウィジェットのタイトルは${title}です`);
}

printWidgetTitle({
    window: {
        title: "Book Viewer"
    }
}); // "ウィジェットのタイトルは Book Viewer です" と出力される

printWidgetTitle({
    // タイトルが定義されてない空のオブジェクト
}); // "ウィジェットのタイトルは未定義です" と出力される
```

また、Optional chaining 演算子 (`?.`) はブラケット記法 (`[]`) と組み合わせることもできます。ブラケット記法の場合も、左辺のオペランドが nullish (`null` または `undefined`) の場合は、それ以上評価せずに `undefined` を返します。一方で、プロパティが存在する場合は、そのプロパティの評価結果を返します。

```
const languages = {
    ja: {
        hello: "こんにちは！"
    },
    en: {
        hello: "Hello!"
    }
};

const langJapanese = "ja";
const langKorean = "ko";
const messageKey = "hello";
// Optional chaining 演算子 (?.) とブラケット記法を組みわせた書き方
console.log(languages?[langJapanese]?.[messageKey]); // => "こんにちは！"
// languages に ko プロパティが定義されていないため、undefined を返す
console.log(languages?[langKorean]?.[messageKey]); // => undefined
```

12.8 `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

第12章 オブジェクト

か？（`String` コンストラクタ関数については「暗黙的な型変換」の章を参照）

実は `String` コンストラクタ関数は、引数に渡されたオブジェクトの `toString` メソッドを呼び出しています。そのため、`String` コンストラクタ関数と `toString` メソッドの結果はどちらも同じになります。

```
const 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"
```

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ができますが、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。

12.9 オブジェクトの静的メソッド

```
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）。そのため、オブジェクトをキーに指定したい場合は `Map` を利用します。

12.9 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。

`Object` 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、`Object.hasOwn` 静的メソッドのような静的メソッドは `Object` そのものに実装されているメソッドです。

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。

12.9.1 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,
  "three": 3
};
// Object.keys はキーを列挙した配列を返す
```

第 12 章 オブジェクト

```
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]
// 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"
```

12.9.2 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。

12.9 オブジェクトの静的メソッド

```
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

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。

第 12 章 オブジェクト

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

プロパティ名が被った場合は、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

12.9 オブジェクトの静的メソッド

(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 プロパティのオブジェクトは同じオブジェクトのままになる
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
  }
};
```

第 12 章 オブジェクト

```
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
```

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。

12.10 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
- プロパティの存在確認するには `in` 演算子か `Object.hasOwnProperty` 静的メソッドを使う
- Optional chaining 演算子 (`?.`) はネストしたプロパティの存在確認とアクセスを同時にを行う記法
- オブジェクトのインスタンスマ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。