WINGS
More Progress, More Future

速習 ECMAScript 6

次世代の標準JavaScriptを今すぐマスター！

- ☞ classで本格オブジェクト指向プログラミング
- ☞ export／importによるアプリのモジュール化
- ☞ アロー関数、分割代入、ジェネレーター etc

山田祥寛 著

目次

Part1 はじめに

ECMAScript 6とは？

ECMAScriptとは、標準化団体Ecma Internationalによって標準化された言語仕様。ブラウザー上で動作するJavaScriptは、基本的に、このECMAScriptの仕様をもとに実装されています。ECMAScript 6は、2015年6月17日（現地時間）に採択された最新バージョンです。正式名称はECMAScript 2015とされていますが、これまでの経緯からECMAScript 6、ES6などと表記されることが多いので、本書でも、あえてECMAScript 6と呼ぶものとします。

ECMAScript 6で新たに提供された主な仕様には、以下のようなものがあります。

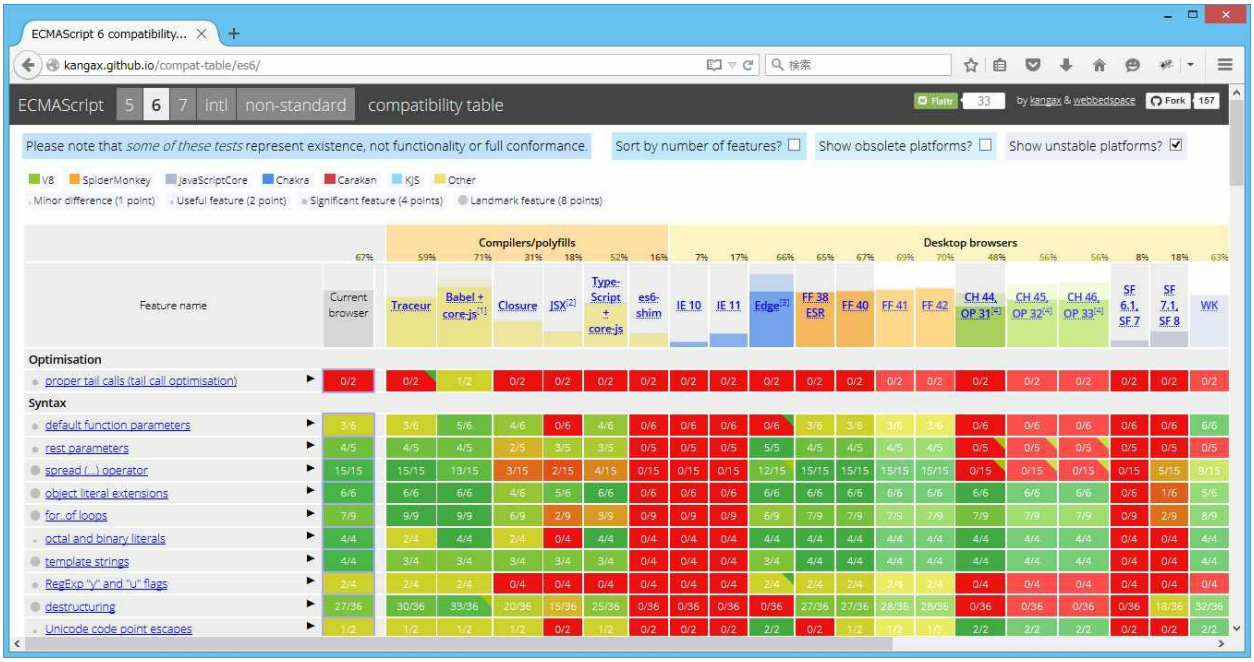
- class命令によるJava／C#ライクなクラス定義が可能に
- module命令によるコードのモジュール化に対応
- 関数構文の改善（引数のデフォルト値、可変長引数、アロー関数など）
- let／const命令によるブロックスコープの導入
- for...of命令による値の列挙
- イテレーター／ジェネレーターによる列挙可能なオブジェクトの操作が可能に
- Promise、コレクション（Map／Set）／Proxyなどの組み込みオブジェクトを追加
- String／Number／Array／Objectなどの既存組み込みオブジェクトの拡充など

さまざまな機能が追加されていますね。その中でも特にclass命令の導入は画期的です。これまでJavaScriptではなにかと不便だったオブジェクト指向プログラミングが、ようやく他の言語に近い——ということは直観的な形で行えるようになったのです。

ECMAScript 6を利用するには？

もっとも、ECMAScriptはあくまで言語仕様の取り決めにすぎません。残念ながら、現存している主なブラウザーがきちんとJavaScriptエンジンに仕様を反映させるまでには、今しばらくの時間が必要となります。

現時点での対応状況を確認するには、「ECMAScript 6 compatibility table」（http://kangax.github.io/compat-table/es6/）などのサイトが参考になるでしょう。Firefox、Chrome、Edgeなどが比較的よく対応しているのが見て取れます。



図：ECMAScript 6 compatibility table

そこで現時点でECMAScript 6を利用するためには、トランスコンパイラー／Polyfilライブラリなどのお世話になる必要があります。

種類	トランスコンパイラー	Polyfilライブラリ
概要	ECMAScript6のコードをECMAScript 5のコードに変換するためのツール	ECMAScript6のコードを実行時にECMAScript 5に変換しながら実行するためのJavaScriptライブラリ

長所	変換済みのコードを実行できるので、動作は早い	ライブラリをインポートするだけで手軽に実行できる
短所	変換する手間がある	実行時変換なので、動作は比較的低速
例	Babel／traceur-compilerなど	browser-polyfill.js／es6-shim.jsなど

表：**ECMAScript 6**を利用するための選択肢

ただし、トランスコンパイラーでもGrunt／Gulpのようなタスクランナーを利用すれば変換処理を限りなく自動化できますので、短所がそこまでネックになることはないでしょう。

［**Note**］なぜ今、**ECMAScript 6**なのか？

標準のブラウザーで利用できないならば、まだECMAScript 6を利用するのは時期尚早なのではないか、そう考える読者の方もいるでしょう。確かに、ECMAScript 6が本格的にスタンダードになるのはすこし先の話です。

しかし、JavaScript界隈の進化は、ここ数年を見ても加速度を増しており、しかも、ECMAScript 6が仕様として確定した以上、現場でスタンダードになるのは確実な未来です。既にFirefox／Chrome／Edgeなどの主要ブラウザーが主な機能に対応していますし、メジャーなJavaScriptフレームワーク／ライブラリもECMAScript 6を見据えたロードマップを示しています。伴い、関連するドキュメントにもECMAScript 6を意識した（＝前提にした）コードをちらほら見かけるようになりました。

さらに、ECMAScript 6ではクラス構文をはじめとして、基本的な構文にさまざまな手が加えられ、ECMAScript 5までのコードとは様変わりしている点も多く見受けられます。いざECMAScript 6が主流になった時に「私の知ってるJavaScriptじゃない！」とならないためにも、まずはこの時期に、少しずつでも新たな構文／仕様に慣れておくことは重要です。

Babelの導入方法

トランスコンパイラーの中でも開発が活発で、よく利用されているのがBabel（http://babeljs.io/）です。以下に、いくつかのパターンでのBabelの導入から

変換までの手順をまとめます。

コードを手動で変換する

Babelのスタンダードな利用方法です。babelコマンドを利用して、コマンドプロンプトからトランスコンパイルを実施します。

［1］Node.jsをインストールする

Babelは、Node.js（https://nodejs.org/）の上で動作します。ダウンロードしたnode-v0.12.2-x64.msiをダブルクリックすると、インストーラーが起動します。あとは、その指示に従って進めるだけなので、特に迷うところはないでしょう。

図：**Node.js**のインストーラー

［2］Babelをインストールする

Babel本体は、npmからインストールできます。コマンドプロンプトから以下のコマンドを実行してください。-gオプションは、ライブラリをグローバルにインストールしなさい、という意味です。

```
> npm install -g babel
```

[3]BabelでECMAScript 6のコードをトランスコンパイル＆実行する

Babelをインストールすることで、babel／babel-nodeなどのコマンドを利用できるようになります。ECMAScript 6のコードをコンパイルするには、babelコマンドを呼び出します。

```
> babel basic.es6.js -o basic.js
```

これでECMAScript 6で書かれたbasic.es6.jsを変換し、その結果をbasic.jsとして出力しなさい、という意味になります。basic.es6.jsの変更を監視して、変更都度に変換処理を実行するならば、以下のように-wオプションを付与してください。

```
> babel -w basic.es6.js -o basic.js
```

トランスコンパイルした結果をそのままNode.js環境で実行するならば、babel-nodeコマンドも利用できます。

```
> babel-node basic.es6.js
```

[4]Polifilライブラリを有効化する

ただし、Babelが変換対象とするのはclass／exportなどの新文法が中心です。Promiseのような新しい組み込みオブジェクト／メソッドを旧来のブラウザー環境で利用するには、Polyfilライブラリ（browser-polyfill.js）を有効化する必要があります。

browser-polyfill.jsは、以下のコマンドでインストールできます。

```
> npm install babel-core
```

あとは、インストールしたbabel-coreに含まれる browser-polyfill.jsを該当するページからインポートすることで、Polyfilが有効になります。

```
<script src="node_modules/babel-core/browser-polyfill.js">
</script>
```

Grunt経由でBabelを実行する

実際の開発では、いちいちコマンドから変換処理を実行するのは手間です。そこでGruntのようなタスクランナーに委ね、変換処理を自動化することもできます。本

書では、Grunt本体のインストール方法／設定に関する詳細は割愛しますので、詳しくは「Web作成の定形作業を自動化できるJavaScriptタスク実行環境 Grunt」（http://codezine.jp/article/detail/8556）などの専門記事も合わせて参照してください。

［1］grunt-babelをインストールする

GruntでBabelを利用するには、npmからgrunt-babelプラグインをインストールします。

```
npm install --save-dev grunt grunt-babel load-grunt-tasks
```

［2］Gruntfile.js（設定ファイル）を準備する

以下は、grunt-babelを利用するための基本的な定義ファイルです。

リスト **gruntfile.js**

```
module.exports = function(grunt) {
 require('load-grunt-tasks')(grunt);
 grunt.initConfig({
  // Babelの設定情報
  babel: {
  options: {
   // ソースマップ（変換前後の対応情報）を生成
   sourceMap: true
   },
   dist: {
    files: {
    // basic.es6.jsをbasic.jsに変換
    'lib/basic.js': 'src/basic.es6.js'
    }
   }
  }
 });
 grunt.registerTask('default', ['babel']);
};
```

［3］変換処理を実施する

あとは、gruntコマンドを実行することで、Babelを実行できます。

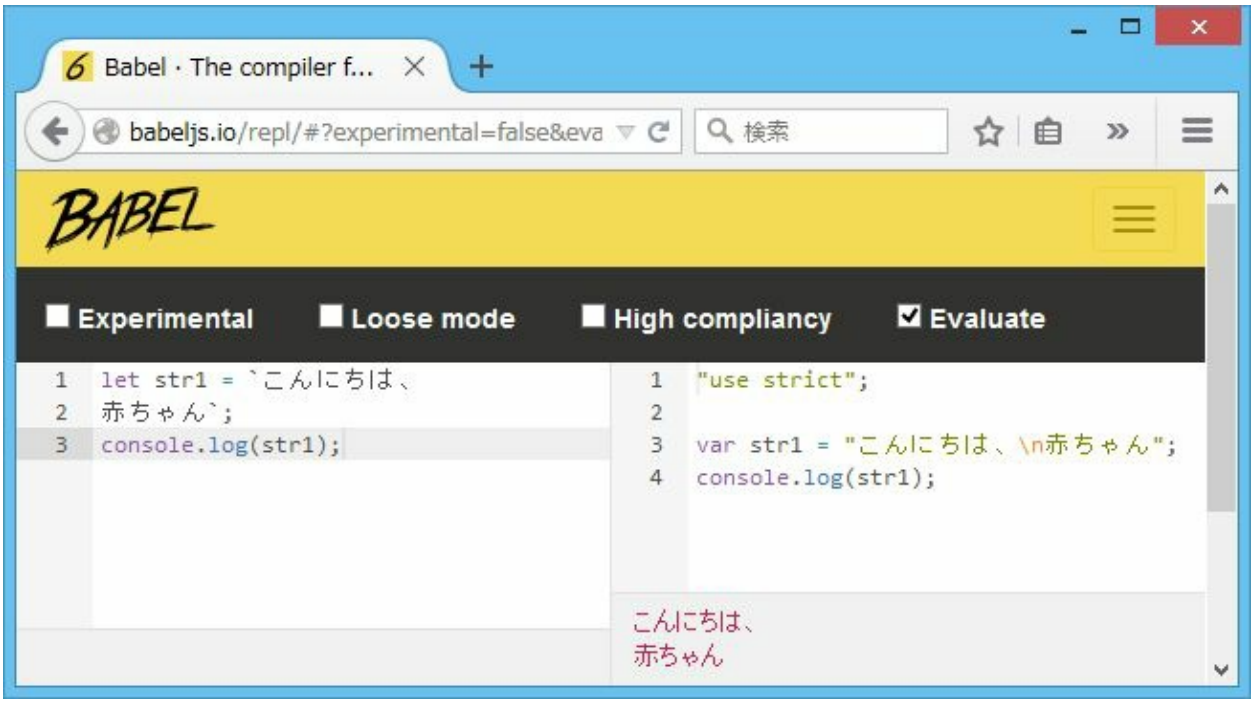
```
> grunt
Running "babel:dist" (babel) task
```

```
Done, without errors.
```

ブラウザー環境で動作する際には、先ほどと同じく、Polyfilライブラリのインポートが必要となります。

簡易インタプリターを利用する

まずは、ECMAScript 6に触れてみたい、最新の機能を学びたいという人にとって、BabelやGruntをいちいちインストールするのは手間です。とりあえず短いコードを書いて試してみたい、という人にとっては、まずは、Babel本家サイトで提供されている簡易インタプリターを利用させてもらうのが手っ取り早いでしょう。インタプリターはブラウザー上で動作しますので、特別な準備もいりません。



図：本家サイトで提供されている簡易インタプリター（**http://babeljs.io/repl/**）

ウィンドウの左枠でECMAScript 6のコードを入力すると、右枠にトランスコンパイル済みのコードが表示され、右下に出力が表示されます。

本書のサンプルも、原則として、簡易インタプリターと、ECMAScript 6のAPIに比較的対応しているという理由からFirefox 40で検証しています（先ほども触れたように、Babelが変換するのはclass／exportなど構文レベルの要素だけで、Map／SetなどAPIの動作はブラウザーに依存するためです）。

対象読者

本書は、既存のJavaScript（ECMAScript 5レベル）をある程度理解している方をターゲットにし、そこからの差分の知識を手早く習得していただくことを目的としています。

従来の構文については、原則として解説していませんので、改めてJavaScriptの基本を学びたいという方は、拙著「JavaScript本格入門」（技術評論社）、「JavaScript逆引きレシピ　jQuery対応」（翔泳社）などの専門書を合わせてお読みいただくことをお勧めいたします。

表紙		
タイトル	JavaScript本格入門	JavaScript逆引きレシピ
出版社	株式会社 技術評論社	株式会社 翔泳社
定価	2,980円（＋税）	3,000円（＋税）

版型	B5変版	A5正寸
ページ数	424ページ	632ページ
色数	2色	2色
ISBN	978-4-7741-4466-5	978-4-7981-3546-5

Part2 基本構文

ブロックスコープを有効にする - let命令

従来のJavaScriptには、ブロックレベルのスコープは存在しませんでした。ブロックスコープとは、変数の有効範囲をたとえばif／forなどのブロック配下に限定するスコープのことです。具体的には、以下のコードを見てみましょう。

```
if (true) {
 var i = 1;
}
console.log(i);
  // 結果：1
```

ブロックスコープが有効な言語では、結果は未定義エラーとなるはずです。ifブロックの中で定義された変数iは、ブロック外では無効であるからです。しかし、JavaScriptではブロックスコープの概念がありませんので、変数iはブロック外でも有効となり、結果は1です。

しかし、ECMAScript 6のlet命令を利用することで、ブロックスコープで有効な変数を宣言できるようになりました。

```
if (true) {
 let i = 1;
}
console.log(i);
  // 結果：エラー（i is not defined）
```

果たして、let命令で宣言された変数はブロック外では無効になり、結果はエラー（i is not defined）となります。

また、let命令ではスコープ内での変数の重複を認めません（var命令では許可）。ECMAScript 6以降でもvar命令は利用できますが、「スコープは最大限限定すべき」「重複チェックをコンパイラーに任せられる」などの理由から、基本はlet命令を優先して利用していくことになるでしょう。

［Note］switchブロックでのlet宣言に注意

switchブロックは、あくまで全体としてひとつのブロックです。よって、case句の単位に同名の変数をlet宣言した場合はエラーとなります。

```
switch (i) {
 case 0:
 let value = 'i:0';
 break;
 case 1:
 let value = 'i:1'; // 重複エラー
 break;
}
```

［**Note**］即時関数は利用しない

即時関数とは、関数ブロックで疑似的にスコープを形成し、グローバルスコープの汚染を防ぐテクニックです。「(function() {...}).call(this);」のように表します。従来のJavaScriptでは定石ともいえるしくみですが、ECMAScript 6では利用すべきではありません。コード全体をブロックで括り、配下の変数をlet命令で宣言すれば、即時関数と同じ効果を得られるからです。

```
{
 let data = 0; // ブロックの外からは見えない
}
console.log(data);
  // 結果：data is not defined
```

定数を宣言する - const命令

const命令で、定数――宣言時に初期値を伴い、あとから値を再代入できない変数――を宣言できます。スコープはlet命令と同じく、ブロックスコープです。

```
const data = 100;
data = 150; // エラー
```

ただし、const命令による定数は正しく「再代入できない」であって、「変更できない」ではない点に注意してください。たとえば、以下のような例を見てみましょう。

```
const data = [1, 2, 3];
data[0] = 10;
console.log(data); // 結果：[10,2,3]
```

この場合、配列オブジェクトはそのままに中身の要素だけを書き換えているため、constの制約にはかかりません（オブジェクトでプロパティを変更する場合なども同様です）。

もちろん、以下のように、配列そのものを変更した場合はエラーとなります。

```
const data = [1, 2, 3];
data = [10, 2, 3];  // エラー
```

整数リテラルの表現力を改善する - 2進数／8進数リテラル

ECMAScript 6では、従来の16進数リテラルに加えて、2進数／8進数リテラルがサポートされています。

2進数リテラルは「0b」、8進数リテラルは「0o（ゼロオー）」で始まるのが基本です。大文字小文字は区別しません（ただし、大文字のOは数字の0と区別が付きにくいため、小文字での表記がお勧めです）。

```
console.log(0o10 === 8);
  // 結果：true
console.log(0b11 === 3);
  // 結果：true
```

これに伴い、Number#toStringメソッド、Number関数でも2進数／8進数表現の変換／解析が可能になっています。

```
let num = 10;
console.log(num.toString(8));
 // 結果：12
console.log(num.toString(2));
  // 結果：1010
console.log(Number('0o12'));
  // 結果：10
console.log(Number('0b1010'));
  // 結果：10
```

ただし、Number関数と似たような機能であるNumber#parseIntメソッドでは、2進数／8進数表現は認識できません（16進数は認識）。現状では、数値リテラルの解析はNumberで統一しておくのが望ましいでしょう。

［Note］従来の8進数表現

従来のJavaScriptでも8進数リテラルが存在しましたが、標準の機能ではなく、実装によって対応がわかれるため、利用すべきではありません。従来型では「010」のように、頭に「0」（ゼロ）を付けるだけでした。

文字列リテラルへの変数／改行の埋め込みを可能にする - テンプレート文字列

テンプレート文字列（Template Strings）を利用することで、さまざまな文字列リテラル表現を利用できます。テンプレート文字列はクォート「"」「'」の代わりに、「`」（バッククォート）で文字列を括ります。

```
let str1 = `こんにちは、
赤ちゃん`;
console.log(str1);
  // 結果：こんにちは、［改行］赤ちゃん
```

これまでであれば「\n」（エスケープシーケンス）で表現していた改行文字を、文字列リテラルの中でそのまま表現できます。

また、以下のように${...}の形式で変数（式）を文字列に埋め込むことも可能です。これまでであれば、変数とリテラルとを「+」演算子で連結するしかなかったところなので、ぐんとシンプルになったことが実感できるでしょう。

```
let name = 'Yamada';
console.log(`Hello,${name}!!`);
  // 結果：Hello,Yamada!!
```

テンプレート文字列をアプリ仕様に加工する - タグ付きテンプレート文字列

タグ付けされたテンプレート（Tagged template strings）を利用することで、テンプレート文字列を関数に渡して、テンプレート文字列による出力をカスタマイズすることもできます。

```
// 2. テンプレート文字列を加工して埋め込み文字列をブラケットで括る関数
function taggedStr(formats, ...args) {
 console.log(formats);
  // 結果：["","、","さん！"]
 console.log(args);
  // 結果：["こんにちは","山田"]
 return formats[0] + '[' + args[0] + ']' + formats[1]
 + '[' + args[1] +']' + formats[2];
}
let greeting = 'こんにちは', name = '山田';
// 1. テンプレート文字列をtaggedStr関数で加工
console.log(taggedStr`${greeting}、${name}さん！`);
  // 結果：[こんにちは]、[山田]さん！
```

テンプレート文字列は、1 のように「関数名`テンプレート文字列`」の形式で、関

数に引き渡せます。関数の側では、第1引数でテンプレート文字列（分解したもの）を、第2引数（可変長引数）で文字列に埋め込まれた値のセットを、それぞれ受け取ります（2）。ここでは、埋め込まれた文字列の前後を[...]で括っているだけですが、本来は（たとえば）埋め込み文字列をエスケープ処理したり、ローカライズ処理するなどの用途で利用します。

以下に、サンプルコードのみ示しておきます。

```
// 与えられた文字列をエスケープ処理するための_e関数
function _e(str) {
 if (!str) { return ''; }
  // 変換表ESCに従って、文字列を置き換え
 return str.replace(/[<>&"']/g, function(submatch) {
 const ESC = {
  '<': '&lt;',
  '>': '&gt;',
  '&': '&amp;',
  '"': '&quot;',
  "'": '&#39;'
 };
 return ESC[submatch];
 });
}
// 分解されたtemplatesとvaluesを順に連結（valuesは_e関数でエスケープ）
function escape(templates, ...values) {
 let result = '';
 for (let i = 0; i < templates.length; i++) {
  result += templates[i] + _e(values[i]);
 }
 return result;
}
// テンプレート文字列をエスケープ処理
let name = '<Tom & Jerry>';
console.log(escape`こんにちは、${name}さん！`);
  // 結果：こんにちは、&lt;Tom &amp; Jerry&gt;さん！
```

新たなデータ型Symbolとは？

ECMAScript 6では、従来のNumber、String、Boolean、Objectなどの型に加えて、新たにSymbolという型が追加されました。Symbolとは、名前の通り、シンボル（象徴）を作成するための型です。一見して文字列とも似ていますが、文字列ではありません。まずは、具体的な挙動をみていきましょう。

まずは、シンボルを作成し、生成されたシンボルの内容を確認してみましょう。

```
let hoge = Symbol('hoge');
let hoge2 = Symbol('hoge');
console.log(typeof hoge);
  // 結果：symbol
console.log(hoge.toString());
  // 結果：Symbol(hoge)
console.log(hoge === hoge2);
  // 結果：false
```

シンボルを生成するのは、Symbol命令の役割です。引数には任意でシンボルの名前を指定できます。ただし、ここで注意していただきたいのは名前が同じでも、別々に作成されたシンボルは別物であるという点です。上の例であれば、hoge、hoge2は、いずれも名前がhogeであるシンボルですが、===演算子での比較では異なるものと見なされます。

また、シンボルでは文字列や数値への暗黙的な型変換はできません。よって、以下はいずれもエラーです。

```
console.log(hoge + '');
  // 結果：Cannot convert a Symbol value to a string
console.log(hoge - 0);
  // 結果：Cannot convert a Symbol value to a number
```

しかし、boolean型、object型への変換は可能です。

```
console.log(typeof !!hoge);
  // 結果：boolean
console.log(typeof new Object(hoge));
  // 結果：object
```

シンボルの用法

さて、このような独特の性質を持ったシンボルですが、どのような状況で利用できるのでしょうか。具体的な利用例を、以下に挙げておきます。

(1)定数の値として利用する

以下のような例を見てみましょう。

```
var JAVASCRIPT = 0;
```

```
var RUBY = 1;
var PERL = 2;
var PYTHON = 3;
var PHP = 4;
```

一般的に、このような定数は、JAVASCRIPT、RUBY、PERLなどを識別するための定数であって、割り当てられた0、1、2...といった値には意味がありません。しかし、これら定数を利用するコードでは、定数、数値いずれを利用してもエラーにはなりません。

```
if (lang === JAVASCRIPT) {...}
if (lang === 0) {...}
```

コードの可読性を考えれば「0」で比較するのは望ましい状態ではありませんし、そもそも「var HOGE = 0;」のような定数が現れた時に、同じ値の定数が混在してしまうのはバグが混入する元です（役割が似ていれば猶更です）。

そこで定数の値としてシンボルを利用するのです。

```
const JAVASCRIPT = Symbol();
const RUBY = Symbol();
const PERL = Symbol();
const PYTHON = Symbol();
const PHP = Symbol();
```

異なるSymbol命令で生成されたシンボルは、同名（引数なしも含む）であってもユニークであるのでした。生成されたシンボルの値にはどこからもわかりませんので、（たとえば）定数JAVASCRIPTと等しいのは定数JAVASCRIPTだけです。

（2）非公開なプロパティを定義する

たとえば以下は、MyClazzクラスの中で、privateなSECRETプロパティを定義する例です（class命令については後述します）。

```
// SECRETプロパティの名前でシンボルで準備
const SECRET = Symbol();
class MyClazz {
 constructor(secret) {
  this.data1 = 1;
  this.data2 = 2;
    // SECRETプロパティに値を設定
  this[SECRET] = secret;
```

```
 }
  // SECRETプロパティを利用したメソッド
 checkSecret(secret) {
  return this[SECRET] === secret;
 }
}
let c = new MyClazz(12345);
// メソッド経由ではSECRETプロパティにアクセスできる
console.log(c.checkSecret(12345));
  // 結果：true
// SECRETプロパティへの直接アクセスは不可
console.log(c.secret);
  // 結果：undefined
// オブジェクトのキー（プロパティ）を列挙
console.log(Object.keys(c));
  // 結果：["data1","data2"]
// オブジェクトのキー（プロパティ）を列挙
for (let k in c) {
 console.log(k);
}
  // 結果：data1、data2
// オブジェクトをJSON文字列に変換
console.log(JSON.stringify(c));
  // 結果：{"data1":1,"data2":2}
```

ここでは、SECRETプロパティの名前をシンボルとして準備し、「this[SECRET]=～」でプロパティとして定義しています。

シンボルSECRETの値は他からは判りませんので、SECRETプロパティに直接アクセスすることはできません。for...in命令による列挙、JSON.stringifyメソッドで生成されたJSON文字列にも、シンボルで生成されたプロパティは現れてこない点に注目してください。

[**Note**]完全に隠ぺいされるわけではない

ただし、シンボルで定義されたプロパティが完全に隠ぺいできるわけではありません。getOwnPropertySymbolsメソッドを利用すれば、シンボルプロパティにアクセスすることは可能です。

```
let idsym = Object.getOwnPropertySymbols(c)[0];
console.log(c[idsym]);
 // 結果：12345
```

配列／オブジェクトから個々の要素を抽出する - 分割代入

分割代入（destructuring assignment）とは、配列／オブジェクトを分解し、その要素／プロパティを個々の変数に展開するための構文です。

以下の例であれば、配列要素を順に変数hoge／fooに割り当てます。

```
let [hoge, foo] = [15, 21];
console.log(hoge);
  // 結果：15
console.log(foo);
  // 結果：21
```

「...」演算子を利用することで、残りの要素をまとめて配列として取り出すこともできます。たとえば以下は、配列要素の先頭2個を変数hoge、fooに、残りの要素をまとめて部分配列としてotherに、それぞれ割り当てる例です。

```
let [hoge, foo, ...other] = [10, 20, 30, 40, 50, 60];
console.log(hoge);
  // 結果：10
console.log(foo);
  // 結果：20
console.log(other);
  // 結果：[30, 40, 50, 60]
```

オブジェクトのプロパティに割り当てることもできます。

```
let {hoge, foo} = {hoge:'ほげ', foo:'ふー'};
console.log(hoge);
  // 結果：ほげ
console.log(foo);
  // 結果：ふー
```

入れ子になったプロパティを割り当てるならば、以下のようにします。

```
let data = { hoge: 'ほげ', foo: { piyo : 'ぴよ' , goo: 'ぐぅ' }};
let { hoge, foo, foo: {piyo, goo} } = data;
console.log(hoge);
  // 結果：ほげ
console.log(foo);
  // 結果：{"piyo":"ぴよ","goo":"ぐぅ"}
console.log(piyo);
  // 結果：ぴよ
console.log(goo);
  // 結果：ぐぅ
```

もしもプロパティを異なる名前の変数に割り当てたいならば、以下のようにも表現できます。たとえば以下であればhogeプロパティは変数xにセットされます。

```
let {hoge: x, foo} = {hoge:'ほげ', foo:'ふー'};
console.log(x);
  // 結果：ほげ
```

また、指定されたプロパティが存在しなかった場合のために、変数の後方に「=値」の形式でデフォルト値を用意しておくこともできます。

```
let {hoge = 'ほげほげ' , foo} = { foo:'ふー'};
console.log(hoge);
  // 結果：ほげほげ
```

[Note]宣言のない代入

本文の例では、宣言と代入をまとめて実施していますが、個々の変数宣言と代入とを切り離すこともできます。

```
let hoge, foo;
[hoge, foo] = [15, 21];
```

ただし、オブジェクトの分割代入では、前後にカッコを付けなければならない点に注意してください。左辺の{...}はブロックと見なされ、それ単体で文とすることはできないからです。

```
let hoge, foo;
({hoge, foo} = {hoge:'ほげ', foo:'ふー'});
```

分割代入の使い方

分割代入の代表的な利用局面を、以下にまとめます。

(1)関数(メソッド)から複数の値を返したい

配列(複数の値)を返す関数から、個々の変数に値を割り当てるには、以下のようにします。これで戻り値result1はhogeに、result2はfooに、それぞれ代入されます。

```
function destructure() {
 let result1 = 10;
 let result2 = 20;
 return [result1, result2];
}
let [hoge, foo] = destructure();
console.log(hoge);
  // 結果：10
```

```
console.log(foo);
  // 結果：20
```

このような状況では、一部の戻り値を無視することも可能です。たとえば以下であれば、戻り値result2だけがfooに割り当てられ、result1は切り捨てられます。

```
let [, foo] = destructure();
```

(2)変数の値を入れ替える

たとえば以下は、変数hoge、fooの内容を入れ替える例です。従来であれば、いずれかの変数をいったん一時変数に預ける必要があったものです。

```
let hoge = 1;
let foo = 15;
[hoge, foo] = [foo, hoge];
console.log(hoge);
  // 結果：15
console.log(foo);
  // 結果：1
```

(3)名前付き引数を指定する

以下は、引数としてupper(上辺)／lower(下辺)／height(高さ)を受け取り、台形の面積を求める例です。オブジェクトとして渡された引数をdestructureして、関数配下では、個々の変数としてアクセスできることが見て取れます。

```
function trapezoid({upper = 1, lower = 1, height = 1}) {
 return (upper + lower) * height / 2;
}
console.log(
 trapezoid({
 upper: 5,
 lower: 10,
 height: 2
 }));
  // 結果：15
```

同じような例で、引数に渡したオブジェクトから特定のプロパティだけを取り出すのに利用することもできます。

```
let book = {
 isbn: '978-4-7741-7568-3',
 title: 'AngularJSアプリケーションプログラミング',
```

```
 price: 3700
};
let getInfo = function({isbn}) {
 console.log(isbn);
};
getInfo(book);
  // 結果：978-4-7741-7568-3
```

この例であれば、getInfo関数は引数としてオブジェクトを受け取りますが、実際には、その中で定義されたisbnプロパティだけを分割代入によって取り出しています。複数のプロパティを必要とする場合にも、関数の呼び出し側で、いちいちプロパティを意識せず、オブジェクトをまるごと渡せるのが良いところです。

(4)正規表現でマッチした部分文字列を抽出する

RegExpオブジェクトのexecメソッドは、正規表現パターンにマッチした文字列を「マッチング文字列全体, サブマッチ文字列1,...」の配列として返します。分割代入を利用することで、得られたサブマッチ文字列を個々の変数に代入できます。

以下は電話番号を正規表現で解析し、市外局番(area)、市内局番(local)、加入者番号(privated)として取得します。

```
let tel = '000-111-2222';
let tel_pattern = /^(0\d{2,4})\-(\d{1,4})\-(\d{2,5})$/;
let [, area, local, privated] = tel_pattern.exec(tel);
console.log(area);
  // 結果：000
console.log(local);
  // 結果：111
console.log(privated);
  // 結果：2222
```

[, area, local, privated]のように、先頭を空にしているのは、この例ではマッチング文字列全体は利用していないためです。分割代入では、不要な変数はこのように無視することが可能です。

配列を個々の変数に展開する - 展開演算子

展開演算子(Spread Operator)とは、関数を呼び出す際に、列挙可能なオブジェクト(配列など)を個々の変数(引数)に展開するための演算子のことです。

たとえば以下のような例を見てみましょう。

```
console.log(Math.max(100, -10, 50, 108));
  // 結果：108
console.log(Math.max([100, -10, 50, 108]));
  // 結果：null
```

Math.maxは可変長引数（任意個数の数値）を受け取るメソッドです。よって、前者は正しく動作しますが、配列を渡した前者は動作しません。これを回避するために、従来はapplyメソッドを使って、以下のように表すのが定石でした。applyメソッドは、第1引数のオブジェクトをthis、第2引数（配列）を引数としてメソッドを実行します。この例であれば、Math.maxはthisがなんであれ、結果は変化しませんので、nullを渡しています。

```
console.log(Math.max.apply(null, [100, -10, 50, 108]));
  // 結果：108
```

しかし、ECMAScript6では、展開演算子を利用することで、より直観的に表現できます。

```
console.log(Math.max(...[100, -10, 50, 108]));
  // 結果：108
```

配列など反復可能なオブジェクトを列挙する - for...of命令

for...of命令を利用することで、列挙可能なオブジェクトからすべての要素を取り出せます。列挙可能なオブジェクトとは、配列だけでなく、Arrayライクなオブジェクト（NodeList、argumentsのような）、イテレーター／ジェネレーターなどを含みます。

```
let data = [1, 2, 4];
Array.prototype.hoge = function() {};
for(let d of data) {
 console.log(d);
}
 // 結果：1、2、4
```

従来のJavaScriptには、よく似た命令としてfor...inがありました。しかし、両者は異なるものですので、以下のコードで違いを理解しておきましょう。

```
let data = [1, 2, 4];
Array.prototype.hoge = function() {};
```

```
for(let d of data) {
 console.log(d);
}
  // 結果：1、2、4


for(let d in data) {
 console.log(d);
}
  // 結果：0、1、2、hoge
```

異なる点は2点です。

- for...of命令が値を列挙するのに対して、for...in命令はプロパティ名（インデックス）を列挙します。
- for...of命令が値だけを列挙するのに対して、for...in命令はprototypeで拡張されたメンバーも含めて列挙します。

配列などの列挙には、for...in命令ではなく、for...of命令を利用するようにしてください。

Part3 関数

引数のデフォルト値を宣言する

従来のJavaScriptでは、引数にデフォルト値を設定することができませんでした。このため、関数ブロックの中で引数が渡されているかをチェックし、未定義の場合にデフォルト値を渡す、冗長なコードを書かなければなりませんでした。

```
function show(name) {
  // 引数nameのデフォルト値を「権兵衛」に
 if (name === undefied) { name = '権兵衛'; }
 console.log('私の名前は' + name + 'です！');
}
```

しかし、ECMAScript 6では引数のデフォルト値が言語仕様として組み込まれました。仮引数の末尾に「= デフォルト値」と付与するだけです。

```
function show(name = '権兵衛') {
 console.log('私の名前は' + name + 'です！');
}
show();
  // 結果：私の名前は権兵衛です！
show('リオ');
  // 結果：私の名前はリオです！
```

また、デフォルト値には他の引数、関数（式）の結果などを指定することもできます。

```
function add(a, b = a) {
 return a + b;
}
console.log(add(1, 4));
  // 結果：5
console.log(add(1));
  // 結果：2（引数bの値はaと同じ1）
function dateFormat(date = new Date()) {
 return date.toLocaleString();
}
console.log(dateFormat(new Date(2015, 11, 4, 0, 0, 0)));
  // 結果：2015/12/4 0:00:00
console.log(dateFormat());
  // 結果：2015/8/20 16:00:48（現在の日時を表示）
```

ただし、他の引数をデフォルト値とする場合、参照できるのは自身より前に定義されたものだけである点に注意してください。

［**Note**］デフォルト値が適用されるのは？

デフォルト値が適用されるのは引数が未指定であった場合だけです（undefined値が明示的に指定された場合にも、適用されます）。その他、たとえばnull、falseなどが渡された場合には、デフォルト値は無視されます。

補足：必須パラメーターの表現

デフォルト値に関数を指定できることを利用して、必須パラメーターを表現することもできます。

```
function required() {
 throw new Error('Arguments is missing');
}
function hoge(value = required()) {
 return value;
}
hoge();
  // 結果：エラー（Arguments is missing）
```

例外をスローするだけのrequired関数を準備しておき、これを必須パラメーターのデフォルト値として指定するわけです。これによって、引数が指定されなかった場合に、required関数が実行（例外がスロー）されます。

可変長引数を利用する

仮引数の前に「...」（ピリオド3個）を付与することで、可変長引数となります（英語ではRest Parameterと表記されています）。渡された任意個数の引数を配列としてまとめて受け取る機能です。

従来のJavaScriptでは、可変長引数を扱うのにargumentsオブジェクトを介する必要がありましたが、名前がありませんので、コードの可読性は自ずと悪くなります。そもそも引数リストとは別に管理しなければならないので、直観的ではありません。しかし、標準的な可変長引数のしくみが用意されたことで、引数リストの一環として可変長引数を扱えるようになり、コードがぐんと読みやすくなります。

```
function sum(...args) {
 let result = 0;
 for(let arg of args) {
```

```
  result += arg;
 }
 return result;
}
console.log(sum(10, 20, 30));
  // 結果：60
```

なお、従来のargumentsオブジェクトはいわゆるArrayライクなオブジェクト（＝Arrayではない）でしたが、可変長引数は真正のArrayです。これは、shift、popのようなArrayの標準メソッドを直接利用できることを意味します。

関数リテラルをシンプルに記述する - アロー関数

アロー関数（Arrow Function）を利用することで、関数リテラルをシンプルに記述できます。たとえば以下の1 、2 は、いずれも同じ意味です。

```
let data = [1, 2, 3];
  // 1. 従来の関数リテラルでの表記
let formatted = data.map(function(value, index) {
 return value * value;
});
  // 2. アロー関数による表記
let formatted = data.map((value, index) => value * value);
console.log(formatted);
  // 結果：[1,4,9]
```

アロー関数では、「(引数,...) => 本体」の形式で関数を表現できます。従来のリテラル表現に比べて、function、{...}などがなくなっているというだけではありません。アロー関数（本体が1文の場合）では、本体の値がそのまま戻り値と見なされますので、returnを明記しなくても構いません。引数がひとつである場合には、以下のように引数を括るカッコを省略することもできます。

```
data.map(value => value * value);
```

ただし、引数がない場合には、カッコの省略はできません（空の丸カッコを書きます）。

```
var func = () => console.log('hoge');
```

また、アロー関数で複数の文を持ちたい場合には、{...}で本体を括ります。この場合、戻り値を表すreturnは省略できませんので、注意してください。

```
data.map((value, index) => {
 return value * value;
});
```

アロー関数はthisを固定する（レキシカルなthis）

アロー関数ではもうひとつ、thisを固定する、という機能があります。

JavaScriptでは、一般的にthisが示すものはそれぞれの関数によって決まります。たとえば、以下は従来の関数リテラルでCounterクラスを定義した例です。countプロパティはインスタンス化されてからの経過秒数を表します。

```
var Counter = function() {
  // 現在のthisを退避
 var _this = this;
 _this.count = 0;
  // 1000ミリ秒間隔でcountプロパティをインクリメント
 setInterval(function() {
  _this.count++;
 }, 1000);
};
var c = new Counter();
```

Counterコンストラクターの直下では、thisはインスタンス自身を指します。しかし、setIntervalメソッドの配下ではthisは変化してしまい、インスタンスを参照しません（ブラウザー環境であればWindowオブジェクトを指します）。そこでコンストラクターでthisを_thisに退避し、setIntervalメソッドでも_this経由でcountプロパティを参照しているのです。

しかし、アロー関数では、thisはアロー関数自身が宣言された場所によって決まります。以下の例であれば、コンストラクターが示すthis（インスタンスそのもの）を指しますので、上の例にあったような_thisへの退避が不要になります。

```
let Counter = function() {
 this.count = 0;
  // 1000ミリ秒間隔でcountプロパティをインクリメント
 setInterval(() => this.count++, 1000);
};
let c = new Counter();
```

注意：オブジェクトリテラルを返す時

Arrow Functionでオブジェクトリテラルを返すには、リテラル全体をカッコでくくりま

す。

```
let hoge = () => ({ foo: 123 });
```

以下のようにカッコなしで表された場合、{...}がブロックを、「foo:」はラベル構文と見なされてしまうからです。

```
let hoge = () => { foo: 123 };
```

Part4 組み込みオブジェクト

非同期処理を簡便に処理する - Promiseオブジェクト

非同期処理を実装するのに、古典的なアプローチとしてコールバック関数があります（setTimeout／setIntervalメソッドなどが好例です）。しかし、非同期処理が複数に連なる場合、コールバック関数は入れ子が深くなりすぎて、ひとつの関数が肥大化する傾向にあります。このような問題をコールバック地獄と言います。

このような問題を解決するのがPromiseオブジェクトです。Promiseを利用することで、非同期処理をあたかも同期処理であるかのように扱えます。jQueryの$.deferred、AngularJSの$qサービスなど、代表的なライブラリ／フレームワークで類似の機能が提供されているので、意識するとせざるとに関わらず、利用したことがある人はおおいはずです。本書では、Promiseの基本的な考え方については割愛しますので、「爆速でわかるjQuery.Deferred超入門」（http://techblog.yahoo.co.jp/programming/jquery-deferred/）などの専門記事を参照してください。jQueryのそれと構文は完全に一致しているわけではありませんが、理解する上での参考になります。

さて、具体的な例を見ていきます。以下は、文字列が渡されると、2000ミリ秒後に「入力値は●○」を、文字列が空の場合には「入力値が空です」というメッセージを、それぞれ返す非同期処理の例です。

```
function hoge(value) {
 return new Promise((resolve, reject) => {
  setTimeout(() => {
   if (value) {
   resolve(`値は${value}`);
   } else {
   reject('入力値が空です');
   }
  }, 2000);
 })
}
hoge('佐藤理央').then(
 response => { console.log(response); },
 error => { console.log(error); }
);
  // 結果：値は佐藤理央
```

Promiseオブジェクトを利用する場合、非同期処理そのものを関数として用意しておきます。この例であれば、hoge関数です。hoge関数が戻り値として返すのが、Promiseオブジェクトです。

Promiseは、非同期処理の状態を監視するためのオブジェクトで、コンストラクターには非同期処理（function）本体を記述します。

構文 **Promise**コンストラクター

new Promise((*resolve*, *reject*) => { *statements*})
resolve：処理の成功を通知するための関数
reject：処理の失敗を通知するための関数
statements：処理本体

Promiseコンストラクターは、resolve／rejectという2個の関数を受け取りますので、非同期処理ではこれらの関数を利用して、成功（resolve）／失敗（reject）を通知するわけです。この例であれば、setTimeout関数の中で、引数valueが未定義であればreject関数を、さもなければresolve関数を、それぞれ呼び出しています。resolve／reject関数の引数には、それぞれ成功した時の結果、エラーメッセージなど、任意のオブジェクトを渡すことができます。

Promiseオブジェクトでの処理結果を受け取るのは、thenメソッドです。

構文 **then**メソッド

then(*succcess*, *failure*)
success：成功コールバック関数
failure：失敗コールバック関数

引数success／failureは、それぞれresolve／reject関数で指定された引数の内容を受け取って、成功／失敗時の処理を実施します。リスト内の太字を削除して、結果が以下のように変化することを確認してみましょう。

入力値が空です

非同期処理を連結する

Promiseのありがたみは、連結してようやくイメージしやすくなります（単一の非同期処理であれば、単なる構文の違いで、コールバック関数でもさほど不便はありません）。

以下は、先ほどのhoge関数を複数回呼び出す例です。

```
// 1. 初回のhoge関数を実行
hoge('佐藤理央')
.then(
 response => {
  console.log('1. ' + response);
    // 2. 初回実行に成功したら、2回目のhoge関数を実行
  return hoge('鈴木幸助');
 }
)
.then(
 response => {
  console.log('2. ' + response);
 },
 error => {
  console.log('Error.' + error);
 }
);
  // 結果：1. 値は佐藤理央、2. 値は鈴木幸助
```

thenメソッドを連結するには、thenメソッドの配下で新たなPromiseオブジェクトを返します。この例であれば1 のhoge関数を実行し、成功したら、2 のhoge関数を呼び出します。

もしも1 の関数呼び出しで引数を空にした場合には、以下のような結果を得られます。

```
Error.入力値が空です
```

最初の成功コールバック関数が無視されて、2番目のthenメソッドで用意された失敗コールバック関数が実行されているのです。このように、非同期処理ごとに失敗コールバック関数を定義しなくても、必要な箇所でまとめてエラー処理できるのもPromiseの良いところです。

ちなみに、以下は1 では引数あり、2 の引数を省略した場合です。今度は、最初の成功コールバック関数が実行されて、その後、2番目のthenメソッドで用意された失敗コールバック関数が呼び出されます。

```
1. 値は佐藤理央
Error.入力値が空です
```

複数の非同期処理を並行して実行する

Promise.allメソッドを利用することで、複数の非同期処理を並列に実行し、そのすべてが成功した場合に、処理を実行することもできます。

```
Promise.all([
 hoge('佐藤理央'),
  hoge('腰掛奈美'),
  hoge('鈴木花子')
]).then(
 response => {
  console.log(response);
 },
 error => {
  console.log('Error.' + error);
 }
);
  // 結果：["値は佐藤理央","値は腰掛奈美","値は鈴木花子"]
```

結果は、それぞれのresolveから渡された結果を表す配列である点に注目です。また、非同期処理のいずれかが失敗した場合には、失敗コールバック関数が実行されます。

また、並行して実行された非同期処理のいずれかひとつが最初に完了したところで、後続の処理を実行するraceメソッドもあります。raceメソッドの結果は、成功／失敗に関わらず、あくまで最初に完了した非同期処理のそれによって変化します。

```
Promise.race([
 hoge('佐藤理央'),
  hoge('腰掛奈美'),
  hoge('鈴木花子')
]).then(
 response => {
  console.log(response);
 },
 error => {
  console.log('Error.' + error);
 }
);
  // 結果：値は佐藤理央（結果は変動する可能性があります）
```

オブジェクトの挙動をカスタマイズする - Proxyオブジェクト

Proxyオブジェクトは、プロパティの設定／取得／削除、for...of／for...in命令による列挙などなど、オブジェクトの標準的な操作を、アプリ独自の操作で差し替えるためのオブジェクトです。Proxyを利用することで、たとえばオブジェクト操作に対するロギング、プロパティ値の検証／変換などを、既存のオブジェクトに手を加えずに実装できます。

たとえば以下は、存在しないプロパティを取得した時に、デフォルト値として「？」を返す例です。

```
let obj = { hoge: 'ほげ', foo: 'ふー' };
var p_obj = new Proxy(obj, {
 get(target, prop) {
  return prop in target ?
   target[prop] : '?';
 }
});
console.log(p_obj.hoge);
  // 結果：ほげ
console.log(p_obj.nothing);
  // 結果：？
```

Proxyコンストラクターの構文は、以下の通りです。

構文 **Proxy**コンストラクター

new Proxy(*target*, *handler*)
target：操作を差し挟む対象のオブジェクト（ターゲット）
handler：ターゲットの操作を定義したオブジェクト（ハンドラー）

ハンドラーでは、以下のようなメソッドを定義できます。ハンドラーメソッドのことをトラップとも呼びます。

メソッド（トラップ）	ターゲットに対する操作
get function(*target*, *prop*, *receiver*) -> any	プロパティの取得
set function(*target*, *prop*, *val*, *receiver*) -> boolean	プロパティの設定

enumerate function(*target*) -> [String]	for...in命令による列挙
iterate function(*target*) -> iterator	for...of命令による列挙
deleteProperty function(*target*, *prop*) -> boolean	delete命令によるプロパティの削除

表：ハンドラーで実装できる主なメソッド

ここでは、getメソッドを実装して、ターゲット（target）のプロパティ（prop）が存在していれば、その値（target[prop]）を、さもなければ「？」を返しています。確かに元々のhogeプロパティが正しく参照できていること、存在しないnothingプロパティに対しては「？」を返すことを確認してみましょう。

なお、プロキシーに対する操作は、ターゲットにも反映されます。

```
p_obj.goo = 'ぐぅ';
console.log(obj.goo);
  // 結果：ぐぅ
console.log(p_obj.goo);
  // 結果：ぐぅ
```

コレクション関連のオブジェクトを標準で提供 - Map／Setなど

ECMAScript 6では、コレクションを管理するための専用オブジェクトとして、Map／Setが追加されました。

キー／値のセットを管理するマップ

Mapオブジェクトは、キー／値のセットでデータを管理するオブジェクトです。従来のJavaScriptではObjectオブジェクト（リテラル）をもって、マップとしての役割を担わせてきました。しかし、Mapを利用することで、以下のような利点があります。

- 任意の型でキーを設定できる（Objectは文字列キーだけ）

- マップのサイズをlengthプロパティで簡単に取得できる（Objectは不可）
- クリーンなマップを作成できる（Objectには既定のキーが存在。ただし、Object.create(null)で回避可能）

利用方法は、他の言語で類似のクラスを利用したことがある人であれば、ごく直観的です。

```
let obj = {};
// マップの生成＆値の登録
let m = new Map();
m.set('hoge', 'ほげ');
m.set('foo', 'ふぅ');
m.set('piyo', 'ぴよ');
// 1. オブジェクトをキーに値を設定
m.set(obj, 'オブジェクト');
console.log(m.get('hoge'));
  // 結果：ほげ
console.log(m.get(obj));
  // 結果：オブジェクト
// 2. オブジェクトリテラルでマップにアクセス
console.log(m.get({}));
  // 結果：undefined
console.log(m.has('hoge'));
  // 結果：true
// マップのキーを列挙
for(let key of m.keys()) {
 console.log(key);
}
  // 結果：hoge、foo、piyo、{}
// マップの値を列挙
for(let value of m.values()) {
 console.log(value);
}
  // 結果：ほげ、ふぅ、ぴよ、オブジェクト
// マップのキー／値を列挙
for(let [key, value] of m) {
 console.log(`${key}:${value}`);
}
  // 結果：hoge:ほげ、foo:ふぅ、piyo:ぴよ、[object Object]:オブジェクト
// マップを順番に処理
m.forEach((value, key) => console.log(`${key}=${value}`));
  // 結果：hoge=ほげ、foo=ふぅ、piyo=ぴよ、[object Object]=オ
```

```
ブジェクト
// hogeキーを削除
m.delete('hoge');
// すべてのキーを削除
m.clear();
```

マップのキーには、1 のように、Object型をはじめ、任意の型を指定できるのでした（function型やnullなども可能です）。ただし、参照型をキーにした場合、getメソッドでの取得には要注意です。2 の例であれば、変数objとリテラル{}の参照先は別なので、値を正しく参照することはできません。

マップの値は、コンストラクターで、以下のように配列内配列（列挙可能なオブジェクト）として初期化することも可能です。

```
let m = new Map([['hoge', 'ほげ'],['foo', 'ふぅ'],['piyo', 'ぴよ'],
[obj, 'オブジェクト']]);
```

一意な値の集合を管理するセット

Setオブジェクトを利用することで、重複しない値の集合を管理できます。重複した値が追加された場合はこれを無視します。

```
let obj = {};
// セットの生成＆値の登録
let s = new Set();
s.add(5);
s.add(10);
s.add(8);
s.add(0);
// 1. 重複した値は無視
s.add(8);
// 2. 任意のオブジェクト型を登録可能
s.add(obj);
// セットの内容を確認
console.log(s.size);
  // 結果：5
console.log(s.has(5));
  // 結果：true
// 3. 参照型に注意！
console.log(s.has({}));
  // 結果：false
console.log(s.has(obj));
  // 結果：true
// セットから値を削除
```

```
s.delete(5);
// セットの値を列挙
for(let value of s) {
 console.log(value);
}
  // 結果：10、8、0、{}
// セットの内容をクリア
s.clear();
```

重複した値を登録した場合には、確かに無視されていることが確認できます（1）。この例であれば、8が2回登録されることはありません。2、3 の参照型の登録／参照についての注意は、マップと同じです。

セットの値は、コンストラクターで、以下のように配列（列挙可能なオブジェクト）として初期化することも可能です。

```
let s = new Set([5, 10, 8, 0, 8, obj]);
```

［**Note**］**NaNの扱い**

Map／Setでは、NaNをキーとして利用することもできます。JavaScriptでは、Nan !== NaNですが、Map／Setでは、NaNはNaNであるとみなされます。他のオブジェクトは、===演算子でもって比較されます。

［**Note**］弱参照に対応したマップ／セット

Map／Setには類似オブジェクトとして、WeakHash／WeakSetと呼ばれる、キーを弱参照で管理するオブジェクトがあります。弱参照とは、このマップ以外でキーが参照されなくなると、そのままガベージコレクトの対象になるということです。標準のMap／Setでは、いわゆる強参照でキーを管理しますので、マップ／セットでキーを保持している限り、そのオブジェクトはガベージコレクトの対象にはなりません。
その性質上、WeakMap／WeakSetでは、「キーは参照型でなければならない」「列挙することはできない（必要であれば、自分で管理）」などの制限があります。

Unicode対応の改善

ECMAScript 6では、String／RegexpオブジェクトでUnicode対応が強化され、サロゲートペアをより手軽に扱えるようになりました。サロゲートペアとは、1文字4byteで表現されるUnicode文字のことです。一般的なUnicode文字は1文字2byteで扱われるため、これまでのJavaScriptではサロゲートペア文字を正しく1文字として扱うことができなかったのです。

```
//  「叱」はサロゲートペア
```

```
let str = '叱られて';
console.log(str.length); // 結果:5(「叱」を2文字と見なす)
```

具体的には、EcmaScript 6では、以下のような改善がなされています。

for...of構文でもサロゲートペアを認識

サロゲートペア文字を含んだ文字列をfor...of命令で正しく取り出すことができます。

```
let str = '叱られて';
for(let d of str) {
 console.log(d);
}
  // 結果：叱、ら、れ、て
```

Unicodeエスケープシーケンスが拡張

新たに「\u{codepoint}」の形式で、0xffffを超えるUnicode文字を表現できるようになりました。よって、以下はいずれも「叱」という文字を表します。

```
console.log('\ud842\udf9f' === '叱');
  // 結果：true
console.log('\u{20b9f}' === '叱');
  // 結果：true
```

サロゲートペアからコードポイントを取得／設定も可能に

Stringオブジェクトには新たにcodePointAt／fromCodePointメソッドが提供され、サロゲートペアからコードポイントを追加したり、コードポイントから文字を復元したりできるようになりました。

```
console.log('叱られて'.codePointAt(0).toString(16));
  // 結果：20b9f
console.log(String.fromCodePoint(0x20b9f));
  // 結果：叱
```

RegExpオブジェクトにuフラグが追加

正規表現を扱うRegExpオブジェクトにもUnicodeを扱うためのuフラグが実装されました。これを利用することで、サロゲートペアを正しく判定できます。

```
let str = '叱られて';
```

```
console.log(/^.られて$/u.test(str));
  // 結果：true
```

「.」は任意の一文字を表します。uフラグを外すと、サロゲートペアが1文字と見なされなくなりますので、結果はfalseとなります。

String／Array／Math／Objectなど組み込みオブジェクトのメソッドも拡充

既存のString／Array／Math／Number／RegExp／Objectなどの組み込みオブジェクトについても、ECMAScript 6ではさまざまなメンバーが追加されています。「ECMAScript 6 compatibility table」（http://kangax.github.io/compat-table/es6/）を見ても判るように、Internet Explorerを除く主要なブラウザーの最新版で多くの機能が実装されており、（無制限とまではいかないまでも）手元の環境でも試しやすくなっています。

なお、本書は個々のメソッドを解説するのが目的ではありませんので、以下では主な追加メソッドを列記していきます。表内で「*」の付いているものは静的メンバーです。

Stringオブジェクト

startsWith／endsWith／includesなどのメソッドが実装され、これまではindexOfメソッドのお世話にならざるを得なかった文字列の判定をよりシンプルなコードで表現できるようになりました。

メソッド	概要
*fromCodePoint	コードポイント値から文字列を生成
*raw`str`	テンプレート文字列の生の文字列を取得
codePointAt(*pos*)	UTF-16エンコードされたコードポイント値を取得（引数posは文字列内の位置）

normalize([*form*])	文字列を引数formの形式で正規化（引数formの値はNFC、NFD、NFKCなど）
repeat(*num*)	文字列を指定回数だけ繰り返したものを取得
startsWith(*search* [,*pos*])	文字列が指定された部分文字列searchで始まるか（引数posは検索開始位置）
endsWith(*search* [,*pos*])	文字列が指定された部分文字列searchで終わるか
includes(*search* [,*pos*])	文字列に指定された部分文字列searchが含まれるか

表：**String**オブジェクトに追加された主なメソッド

Arrayオブジェクト

イテレーターに対応したkeys／valuesをはじめ、要素検索のためのfind／findIndexメソッドなどが追加となっています。

メソッド	概要
*from(*alike* [,*map* [,*othis*])	配列ライクなオブジェクト、列挙可能なオブジェクトを配列に変換（引数mapは要素を変換するための関数、othisはthisとなるオブジェクト）
*of(*e1*,...)	可変長引数e1...から配列を生成

copyWithin(*target*,*start* [,*end*])	start～end-1番目の要素をtargetの位置にコピー
find(fn(*elem*, *index*, *ary*) [,othis])	コールバック関数fnが初めてtrueを返した要素を取得（コールバック関数の引数は、先頭から要素値、インデックス、配列）
findIndex(fn(*elem*, *index*, *ary*) [,*othis*])	コールバック関数fnが初めてtrueを返した要素のインデックス番号を取得
fill(*v*, *start*, *end*)	start～end-1番目の要素に値vをセット
keys()	配列のすべてのキーを含んだイテレーターを取得
values()	配列のすべての値を含んだイテレーターを取得
entries()	配列のすべてのエントリー（キー／値）を含んだイテレーターを取得

表：**Array**オブジェクトに追加された主なメソッド

Array.fromメソッドは、たとえばargumentsオブジェクトのような配列ライクなオブジェクトを配列として操作する際に用います。

```
function hoge() {
 let args = Array.from(arguments);
 console.log(args.length);
  // 結果：5
}
hoge(1, 2, 3, 4, 5);
```

これまでであれば、以下のような、やや直観的でないコードで強制的に配列に変換する必要がありました。

```
var args = Array.prototype.slice.call(arguments);
```

Mathオブジェクト

三角関数／対数に関わるメソッドが増強された他、立方根やシンプルな小数点数の切り捨てなどに対応しています。

メソッド	概要
clz32(*x*)	指定した値を32ビット符号なし整数にした時の、先頭の0の個数
sign(*x*)	指定した値が正数の場合は1、負数の場合は-1、0の場合は0を返す
trunc(*x*)	小数部分を単純に切り捨て、整数部分を取得
cbrt(*x*)	立方根を取得
hypot(*x1*, *x2*,...)	引数の二乗和の平方根を取得
log10(*x*)	底を10とする対数を取得
log2(*x*)	底を2とする対数を取得

log1p(*x*)	引数xに1を加えたものの自然対数を取得
expm1(*x*)	e^x - 1を取得
cosh(*x*)	ハイパーボリックコサインを取得
sinh(*x*)	ハイパーボリックサインを取得
tanh(*x*)	ハイパーボリックタンジェントを取得
acosh(*x*)	ハイパーボリックアークコサインを取得
asinh(*x*)	ハイパーボリックアークサインを取得
atanh(*x*)	ハイパーボリックアークタンジェントを取得
imul(*a* ,*b*)	高速な32ビット整数の乗算
fround(*x*)	指定した値に最も近い単精度float値を取得

表：**Math**オブジェクトに追加された主なメソッド

Numberオブジェクト

これまでグローバルオブジェクト（関数）として提供されてきたisNaN／isFinite／parseInt／parseFloatなどのメソッドがNumberオブジェクトのメンバーとして組み込まれ、よりわかりやすくなりました。

メンバー	概要
*EPSILON	1と、Number値で表現できる1よりも大きい最小値との差（2.220446049250313080847263336181 6E-16）
*MIN_SAFE_INTEGER	JavaScriptにおいて正確に扱える最小の整数（-9007199254740991）
*MAX_SAFE_INTEGER	JavaScriptにおいて正確に扱える最大の整数（9007199254740991）
*isNaN(*num*)	NaN（Not a Number）であるかを判定
*isFinite(*num*)	有限値であるかを判定
*isInteger(*num*)	整数値であるかを判定
*isSafeInteger(*num*)	Safe Integer（正しくIEEE-754倍精度数として表現できるか）であるかを判定
*parseFloat(*str*)	文字列を小数点数に変換
*parseInt(*str* [,*radix*])	文字列を整数に変換（引数radixは基数）

表：**Number**オブジェクトに追加された主なメンバー

isFinite／isNaNメソッドは、正しくはグローバルオブジェクトの同名のメソッドとは挙動が一致しない点に注意してください。というのも、グローバルオブジェクトのそれは引数を数値に暗黙的に変換した上で判定するのに対して、Numberオブジェクトでは数値であり、かつ、NaN／Finite（有限値）であるものだけをtrueとします。よって、以下のようなコードでは結果が変わります。

```
console.log(Number.isNaN('hoge'));
  // 結果：false
console.log(isNaN('hoge'));
  // 結果：true
```

一方、parseInt／parseFloatメソッドはグローバルオブジェクトと完全に一致した動作を提供します。

RegExpオブジェクト

新たにyフラグが追加され、lastIndexプロパティで指定された位置でマッチングを試みることができるようになりました。

```
let re = /WINGS/y;
console.log(re.flags);
  // 結果：y
console.log(re.test('WINGS'));
  // 結果：true
re.lastIndex = 2;
console.log(re.test('出版WINGS'));
  // 結果：true
```

この例では、3文字目（先頭文字は0）を先頭にマッチングを試みます。lastIndexプロパティを1、3などと変更した場合に、結果がfalse（マッチしない）になることも確認してみましょう。

なお、サンプルで利用しているflagsプロパティは、同じくECMAScript 6で追加となったプロパティで、現在のRegExpオブジェクトで有効になっているフラグを返します。

Objectオブジェクト

Objectオブジェクトでは、以下のようなメソッドが追加されています。

メソッド	概要
*assign(*target*, *src*,...)	オブジェクトtargetに対してオブジェクトsrc,...のプロパティをコピー
*is(*v1*, *v2*)	引数同士が等しいかどうかを判定
*getOwnPropertySymbols(*obj*)	オブジェクト配下に含まれるすべてのシンボルプロパティを取得（具体的な例はシンボルの項を参照）
*setPrototypeOf(*obj*, *proto*)	オブジェクトobjに新たなプロトタイプprotoを設定

表：**Object**オブジェクトに追加された主なメンバー

assignメソッドは、jQueryなどでは$.extendメソッドとして提供されていたものです。たとえば、コンストラクターで初期値をプロパティにまとめて割り当てるようなケースでも利用できます。

```
class Person {
 constructor(firstName, lastName, sex) {
  Object.assign(this, {firstName, lastName, sex });
 }
 show() {
  return `${this.lastName}${this.firstName}は${this.sex}です。`;
 }
}
let p = new Person('理央', '佐藤', '女');
console.log(p.show());
  // 結果：佐藤理央は女です。
```

コンストラクターの中では、thisが現在のインスタンスを渡しますので、これに引数をまとめたオブジェクトリテラルをまとめてマージしているわけです。

［**Note**］本項のサンプル

コンストラクター、オブジェクト初期化子の省略記法は、いずれもECMAScript 6で導入されたものです。詳しい解説は、それぞれ該当する項を参照してください。

isメソッドは、ほぼ===演算子と同じ結果を返しますが、以下の比較でのみ異なる結果を返します。

```
console.log(Object.is(+0, -0));
  // 結果：false
console.log(+0 === -0);
  // 結果：true
console.log(Object.is(NaN, NaN));
  // 結果：true
console.log(Object.is(NaN === NaN));
  // 結果：false
```

Part5 オブジェクト指向構文

オブジェクトリテラルをよりシンプルに表現する

従来からのオブジェクトリテラル――{ 名前: 値,... }の表記がよりシンプルに表現できるようになりました。

変数を同名のプロパティに設定する

プロパティの名前と、その値を表す変数とが同名である場合には、値の指定を省略できます。

```
let title = 'AngularJSアプリケーションプログラミング';
let price = 3700;
let publish = '技術評論社';
let book = { title, price, publish };
console.log(book);
  // 結果：{"title":"AngularJSアプリケーションプログラミング","price":3700,"publish":"技術評論社"}
```

これまでであれば、太字の部分は以下のように表す必要がありました。

```
{ title: title, price: price, publish: publish }
```

メソッドを定義する

従来は、メソッドも「名前: function(params,...) {...}」のように関数型のプロパティとして表記しなければなりませんでした。しかし、ECMAScript 6では「名前(params) {...}」という表記が許されています。これは、一般的なメソッド定義に沿った表記でもあり、シンプルである以上に直観的です。

```
let book = {
 title: 'AngularJSアプリケーションプログラミング',
 price: 3700,
 toString() {
    console.log(`${this.title}:${this.price}円`);
  }
};
book.toString();
  // 結果：AngularJSアプリケーションプログラミング:3700円
```

プロパティ名を動的に生成できる

プロパティ名をブラケット([...])で括ることで、式の値から動的にプロパティ名を生成できます(Computed property names)。

```
let i = 0;
let data = {
 ['hoge' + ++i]: 15,
 ['hoge' + ++i]: 20,
 ['hoge' + ++i]: 25
};
console.log(data);
  // 結果：{"hoge1":15,"hoge2":20,"hoge3":25}
```

より実践的な例については、イテレーターの項も合わせて参照してください。

クラスを定義する - class命令

ECMAScript 6では、いよいよclass命令が利用できるようになりました。たとえば以下はname(名前)／sex(性別)プロパティ、showメソッドを提供するPersonクラスの例です。

```
class Person {
 constructor(name, sex) {
  this.name = name;
  this.sex = sex;
 }
 show() {
  return `${this.name}は${this.sex}です。`;
 }
}
let p = new Person('理央', '女');
console.log(p.show());
  // 結果：理央は女です。
```

従来のprototypeプロパティやfunctionによるコンストラクター定義に比べると直観的でもあり、Java／C#などの言語を学んだことがある人であれば、すぐに理解できるはずです。ただし、public／protected／privateのようなアクセス修飾子はありません。

constructorはclassブロック配下で利用できる特殊なメソッドで、名前の通り、コンストラクターを表します。

[Note]functionコンストラクターと等価ではない

classブロックで定義されたクラスは、内部的には関数です。ただし、従来のfunctionコンストラクターと完全に等価ではない点に注意してください。

・関数として呼び出すことはできない（「let p = Person(...);」は不可）
・クラスの巻き上げ（hoisted）はない（定義前の呼び出しは不可）

匿名クラス（リテラル表現）も利用できる

「class { ... }」の形式で、クラスリテラル（表現）も指定できます。関数リテラル（function() {...}）と同じく、式の中で利用可能です。

```
const Person = class {
  ...中身は前述の通り...
}
let p = new Person('理央', '女');
console.log(p.show());
  // 結果：理央は女です。
```

静的メソッドを定義する - static修飾子

static修飾子を利用することで、静的メソッドを定義することも可能です。

```
class Figure {
 static triangle(base, height) {
  return base * height / 2;
 }
}
console.log(Figure.triangle(10, 5));
  // 結果：25
```

getter／setterも利用できる

ECMAScript 5ではオブジェクトリテラルの中で利用できたgetter／setter構文が、ECMAScript 6ではclassブロックの中で利用できます。

```
class Person {
 constructor(name, sex) {
  this.name = name;
  this.sex = sex;
 }
  // ageプロパティのgetter／setter
 get age() {
    return this._age;
  }
  set age(value) {
```

```
    this._age = value
  }
 show() {
  return `${this.name}は${this.sex}、${this.age}歳です。`;
 }
}
let p = new Person('理央', '女');
p.age = 10;
console.log(p.show());
  // 結果：理央は女、10歳です。
```

ECMAScript 6では、classブロックの中で「let age = 0;」のような、いわゆるインスタンスフィールドを定義することはできません。代わりに、getter／setterを利用してください。

既存のクラスを継承する - extendsキーワード

extendsキーワードを利用することで、既存のクラスを継承してサブクラスを定義することもできます。

```
class Person {
 constructor(name, sex) {
  this.name = name;
  this.sex = sex;
 }
 show() {
  return `${this.name}は${this.sex}です。`;
  }
}
class BusinessPerson extends Person {
 constructor(name, sex, clazz) {
  super(name, sex);
  this.clazz = clazz;
 }
 show() {
  return `${super.show()} 役職は${this.clazz}です。`;
 }
}
let bp = new BusinessPerson('理央', '女', '主任');
console.log(bp.show());
  // 結果：理央は女です。 役職は主任です。
```

サブクラスでは、superキーワードを利用することで、スーパークラスのメソッド（コンストラクター）を呼び出すこともできます（太字部分）。

［**Note**］組み込みオブジェクトも継承可能に

ECMAScript 6では、Array／Date／Errorなどの組み込みオブジェクトをextendsキーワードで継承できるようになりました。特に、Errorなどはアプリ独自の例外オブジェクトを実装したい時に、標準的な手続きで作成できるのは便利です。

列挙可能なオブジェクトを定義する - イテレーター

イテレーターとは、オブジェクトの内容を列挙するための仕組みを備えたオブジェクトです。たとえばArray、String、Map、Setなどの標準オブジェクトは、いずれも、デフォルトでイテレーターを備えているので、for...of命令で配下の要素を列挙できるわけです。

```
let array_data = [1, 2, 3];
let str_data = 'いろは';
let map_data = new Map();
map_data.set('JS', 'JavaScript');
map_data.set('PL', 'Perl');
map_data.set('PY', 'Python');
for(let tmp of array_data) {
 console.log(tmp);
}
  // 結果：1、2、3
for(let tmp of str_data) {
 console.log(tmp);
}
  // 結果：い、ろ、は
for(let [key, value] of map_data) {
 console.log(`${key}:${value}`);
}
  // 結果：JS：JavaScript、PL：Perl、PY：Python
```

配列を列挙している部分を、もう少し原始的に書くと、以下のようになります。

```
let itr = array_data.values();
let c;
while(c = itr.next()) {
 if (c.done) { break; }
 console.log(c.done);
  // 結果：false、false、false
 console.log(c.value);
  // 結果：1、2、3
}
```

valuesメソッドは配列内部のイテレーターを返します。イテレーターは、配列の次の要素を取得するためのnextメソッドを持ちます。nextメソッドの戻り値は、以下のようなプロパティを持つオブジェクトです。

- value：次の要素の値
- done：イテレーターが終端に到達したか

ここではdoneプロパティがtrueになるまでwhileループを繰り返すことで、配列の内容をすべて出力しているわけです。for...of命令は、このようなイテレーターの取得からdoneプロパティによる判定、valueプロパティからの値の取り出しまでを自動的に賄ってくれる糖衣構文であるといっても良いでしょう。

イテレーターを実装したクラスの準備

for...of命令では、内部的にはSymbol.iteratorという名前のメソッドを呼び出し、デフォルトイテレーターを取得します（ビルトインイテレーターシンボル）。よって、イテレーターを実装したクラスを定義する場合にも、Symbol.iteratorという名前でイテレーター（nextメソッドを実装したオブジェクトを返す）を用意すればよい、ということになります。

たとえば以下のMyClazzクラスは、コンストラクターで渡された配列を、for...of命令で列挙可能にしたものです。

```
class MyClazz {
  // 引数で渡された配列を保持
 constructor(data) {
  this.data = data;
 }
  // デフォルトイテレーターを取得するためのメソッドを準備
 [Symbol.iterator](){
  let current = 0;
  let that = this;
  return {
        // dataプロパティの次の要素を取得
    next(){
     return current < that.data.length ?
      {value:that.data[current++], done:false} :
      {done: true};
    }
  };
 }
```

```
}
// MyClazz内部で保持された配列を列挙
let c = new MyClazz(['ほげ', 'ふー', 'ぴよ']);
for(let d of c) {
 console.log(d);
}
  結果：ほげ、ふー、ぴよ
```

[Symbol.iterator]と、ブラケットで括っているのはComputed Propertyの構文です。

Symbol.iteratorメソッドが返すイテレーターのnextメソッドでは、配列の末尾まで到達していたら{ done: true }であるオブジェクトを、さもなくば、{ value: 配列の現在値, done: false}を返しています。

列挙可能なオブジェクトをより簡単に実装する - ジェネレーター

ジェネレーターを利用することで、列挙可能なオブジェクトをより簡単に実装できます。以下に、まずはごく基本的なジェネレーターの例を示します。

```
function* myGenerator() {
 yield 'あ';
 yield 'い';
 yield 'う';
}
for(let t of myGenerator()) {
 console.log(t);
}
  // 結果：あ、い、う
```

ジェネレーターを定義するには、function* {...}という構文を利用します。functionキーワードの後方に「*」を付与するだけです。

ジェネレーターの配下では、yield命令を呼び出すことができます。yieldは、returnとよく似た命令で関数の値を呼び出し元に返します。しかし、return命令がその場で関数の実行を終了するのに対して、yield命令は処理を一時停止します。つまり、次に呼び出された時には、その時点から処理を再開できます。

よって、定義されたジェネレーターmyGeneratorをfor...of命令に渡すことで、ループの都度、先頭から順番にyield命令による値――あ、い、うが返されるというわけです。

カウントダウンするジェネレーター

もう少しだけ実用性のありそうなジェネレーターを準備してみましょう。ジェネレーターcountdownは、引数beginの値を呼び出し都度にデクリメントします（値が0になったところで処理を終了）。

```
function* countdown(begin) {
 while (begin >= 0) {
  yield begin--;
 }
}
for(let t of countdown(10)) {
 console.log(t);
}
  // 結果：10、9、8、7、6、5、4、3、2、1、0
```

countdown関数では、beginが0以上の間だけwhileループを繰り返し、yield命令で値を返させています。よって、この例であれば、10～0の値を出力し、yield命令が値を返さなくなったところで処理を終了します。

[Note]前項サンプルの書き換え

ジェネレーターを利用することで、前項のサンプルもシンプルに記述できます。配列の内容（that.data）をすべて読み込むまでyield命令を発行します。

```
class MyClazz {
 constructor(data) {
  this.data = data;
  this[Symbol.iterator] = function*(){
      let current = 0;
      let that = this;
      while(current < that.data.length) {
        yield that.data[current++];
      }
    }
 }
}
```

アプリを機能単位にまとめる - モジュール

[Note]babel-nodeコマンド

本項の内容は複数のファイルを前提としています。簡易インタプリター（Try it out）ではなく、babel-nodeコマンドで実行してください。

アプリの規模が大きくなればなるほど、アプリを機能単位に分割／整理するモジュールの存在は重要になってきます。これまでJavaScriptではモジュールを言語としてサポートしてこなかったため、外部ライブラリのお世話になる必要がありました。

しかし、ECMAScript 6では、いよいよ言語としてモジュールがサポートされることとなりました。以下はモジュールの基本的な例で、三角形や円の面積を求めるFigureモジュールをまとめたものです。

```
// lib/Figure.js
const PI = 3.1415;
export function triangle(base, height) {
 return base * height / 2;
};
export function circle(radius) {
 return radius * radius * PI;
}
// main.js
import {triangle, circle} from './lib/Figure';
console.log(triangle(10, 5));
  // 結果：25
console.log(circle(2));
  // 結果：12.566
```

モジュールの外からアクセスできる要素は、exportキーワードで明示的に修飾します。たとえば上の例であれば、triangle／circle関数にexportキーワードが付いていますので、モジュール外部からのアクセスが可能になります。関数のほか、変数／定数／クラスなどにexportキーワードを付与することもできます。

定数PIはexportキーワードで修飾されていないので、モジュール外部からは参照できません。

以上でモジュールを準備できましたので、あとはメインのコード（ここではmain.js）からモジュールをインポートします。これには、import命令を利用します。

構文 **import命令**

```
import { name, ... } from module
  name：インポートする要素
  module：モジュール（.js拡張子を抜いたパス）
```

たとえばこの例であれば、Figureモジュールからtriangle／circle関数を利用でき

るようにします。モジュール側で明示的にexport宣言していても、利用側でインポートされなかったものにはアクセスできません。たとえば、import命令を以下のようにした場合には、circle関数にはアクセスできません。

```
import {triangle} from './lib/Figure';
```

モジュールの内容をまるごとインポートする

アスタリスク(*)でモジュール内のすべてのエクスポートをインポートすることもできます。

構文 import命令(2)

import * as *alias* from *module*
alias:モジュールの別名
module:モジュール

第2構文で、先ほどのmain.jsを書き換えてみます。

```
import * as fig from './lib/Figure';
console.log(fig.triangle(10, 5));
console.log(fig.circle(2));
```

これでFigureモジュールのすべてのエクスポートを「fig.～」の形式で参照できるようになります。

デフォルトのエクスポートを宣言する

モジュールにつきひとつだけであれば、デフォルトのエクスポートを宣言することもできます。これには、以下のようにdefaultキーワードを付与してください。デフォルトエクスポートでは、関数／クラスなどの名前は不要です。

```
export default function(base, height) {
 return base * height / 2;
};
```

これをインポートするには、以下のようにimport命令を記述します。これで、Figureモジュールのデフォルトエクスポートに対してFigという名前でアクセスできるようになります。

```
import Fig from './lib/Figure';
console.log(Fig(10, 5));
```

補足：ブラウザー環境で動作するには？

Babelでは、内部的に、import命令をrequire関数（Node.js）に変換します。よって、これをブラウザー環境で動作するには、Browserify（babelify）などツールのお世話になる必要があります。Browserify（babelify）は、require関数をブラウザーでも利用できるように変換するためのツールです。以下のコマンドでインストールできます。

```
> npm install -g browserify
> npm install --save-dev babelify
```

あとは、以下のコマンドで該当するファイルを変換＆結合してください。

```
> browserify main.js lib/Figure.js -t babelify --outfile myapp.js
```

main.js／Figure.jsをバンドルした結果がmyapp.jsとして出力されますので、あとは、これをページからインポートすることで、ブラウザー上で動作するようになります。

書籍情報

著者プロフィール

山田 祥寛（やまだ よしひろ）

Microsoft MVP for ASP.NET/IIS。執筆コミュニティ「WINGS プロジェクト」の代表でもある。主な著書に「AngularJSアプリケーションプログラミング」（技術評論社）、「ASP.NET MVC 5実践プログラミング」「はじめてのAndroidアプリ開発」（秀和システム）など。

基本情報

2015年8月発行

著　者：山田 祥寛（やまだ よしひろ）

発行者：WINGSプロジェクト

サポートサイト

http://www.wings.msn.to/

http://keijiban.msn.to/top.jsp?id=gr7638（Q&A掲示板）