書籍「改訂新版 JavaScript 本格入門　～モダンスタイルによる基礎から現場での応用まで」

- 著者：山田　祥寛（Yamada Yoshihiro）
- 2016 年 11 月 01 日　初版　第 1 刷　発行
- 2018 年 08 月 25 日　初版　第 5 刷　発行

はじめに

- この本の目的
    - Javascirpt という言語の理解を確かなものにしたい方のための書籍
- Javascript という言語は、よく言えば「柔軟な」、悪く言えば「あいまいでいいかげんな」言語
    - それだけにバグやセキュリティ上の問題をはらみやすい
    - 本格マスタするためには基礎が大事
- サンプルプログラム付属

Chapter 1 イントロダクション

1.1 JavaScript とは

- Netscape Communications 社によって開発された
- 開発当初は Live Script と呼ばれていたが当時人気のあった Java 言語にあやかって JavaScript と名前を改めた

1.1.1 JavaScript の歴史

- 動きのある動的なページをとりあえず作りたい
    - その結果、「ダサい」ページが量産された
    - 「ダサいページを作成するための言語」
    - 「プログラミング素人が使う低俗言語」
- クラスブラウザー問題
    - ブラウザ間で挙動が違う
    - 面倒さからさらにユーザの足は遠のく

1.1.2 復権のきっかけは Ajax，そして HTML5

- 2005 年、Ajax という技術が登場
    - Ajax とは：ブラウザ上で、デスクトップアプリライクなページを作成するための技術
- この頃、ブラウザベンダーによる拡張会議合戦も落着き、互換性の問題も少なくなっていました。

    - 言語の価値が見直され始めた
- Ajax 技術普及で、「Ajax 技術を支える中核」とみなされるようになった
- HTML5 が 2000 年台後半で登場したことにより更に加速
- SPA(Single Page Application)も活発

1.1.3 マイナスイメージの誤解

- JavaScript に対する誤解への言及
- JavaScript は今最も理解しておくべき言語の１つ

1.1.4 言語としての 4 つの特徴

1. スクリプト言語である：簡易であることを目的としている
2. インタプリタ型の言語である：コンパイル言語と対比する
3. 様々な環境で利用できる：node.js や Windows Script Host、Java Platform,Standard Edition
4. いくつかの機能から構成される：コア部分と DOM とブラウザーオブジェクトといろんな機能からなる

1.2 次世代 JavaScript「ECMAScript 2015」とは

- ECMAScript とは
    - 標準化団体 ECMA International によって標準化された JavaScript
    - 現在の最新版は 2015 年 6 月に採択された第 6 版　ECMAScript2015(ES2015)。通称、ECMAScript6(ES6)と呼ばれることもある
    - オブジェクト指向プログラミングが、ようやく直感的にできるように！

1.2.1 ブラウザーの対応状況

- 完全対応するには時間を要する状況
    - 参考）https://kangax.github.io/compat-table/es6/
- トランスコンパイラー
    - ES2015 のコードを従来の ES5 仕様のコードに変換するためのツール
    - 本書ではトランスコンパイラーとして「Babel」を採用している

1.3 ブラウザー付属の開発者ツール

1.3.1 開発者ツールを起動する

1.3.2 HTML／CSS のソースを確認する - ［Elements］タブ -

1.3.3 通信状況をトレースする - ［Network］タブ -

- Ajax のリクエスト内容を確認したりとか

1.3.4 スクリプトをデバッグする - ［Sources］タブ -

- 開発で最も重要！
- JavaScript ソースのデバッグに用いる
    - ブレイクポイントをおいて、処理をデバッグしてくために用いるのが一般的
- 圧縮されたコードを読みやすくする
    - [Sources]タブ下の{}(Pretty print)ボタンをクリックすることでコードを改行/インデントすることが可能

1.3.5 ストレージ／クッキーの内容を確認する - ［Application］タブ -

- Cookie の中身を確認したり設定したりできる

1.3.6 ログ確認／オブジェクト操作などの万能ツール - ［Console］タブ -

- ［Sources］タブと並んで重要！
- エラーメッセージやログを確認
- 対話的にコードを実行する
- コラム「JavaScript をまなぶ上で役立つサイト」
    - １．Mozilla Developer Network https://developer.mozilla.org/ja/docs/Web/JavaScript
    - ２．jQuery 逆引きリファレンス https://www.buildinsider.net/web/jqueryref
    - ３．HTML クイック・リファレンス http://www.htmq.com/
    - ４．ECMAScript2015 Language Specification http://www.ecma-international.org/ecma-262/6.0/
    - ５．HTML5 Experts.jp https://html5experts.jp/

Chapter 2 基本的な書き方を身につける

2.1 JavaScript の基本的な記法

2.1.1 JavaScript で「こんにちは，世界！」

2.1.2 JavaScript を HTML ファイルに組み込む - `<Script>`要素 -

- `<script>`要素を記述する位置
    - `<body>`要素の配下（閉じタグの直前）
        - ページ描画が終わったあとに動かさないと、Javascriptの処理のせいでページ表示が送れるため
    - `<head>`要素の配下
        - 関数定義等を先に読み込んでおく運用
        - ScriptでStyleを出力する際
- 基本的には「`<body>`要素の配下（閉じタグの直前）」に記載すること
- アンカータグにスクリプトを埋め込む-JavaScript 疑似プロトコル-

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>JavaScript本格入門</title>
  </head>
  <body>
    <a href="JavaScript:window.alert('こんにちは、世界！');">
      ダイアログを表示</a
    >
  </body>
</html>
```

2.1.3 文（Statement）のルール

- 文の最後に；をつける
  - つけなくても動くが、終わりを明示するため
- 複数の分を単一行で書くこともできるが、；を記載したら絶対に改行を入れる
  - デバッガは１行単位で実行されるし、読みにくい

2.1.4 コメントを挿入する

- コメントに付いては Java と同様のため割愛

2.2 変数／定数

2.2.1 変数を宣言する

- 変数宣言はvar命令で必ず宣言する！

```
// 正常に動作するが...
// 宣言の省略は原則として避けるべき
msg = "こんにちは、JavaScript！";
console.log(msg);
```

- 変数宣言
  - let 命令：より細かく変数の有効範囲を管理できる
  - var 命令
- 所感
  - 変数宣言は基本的にletかconstのみを活用する運用で差し支えないと思う
  - varとletの挙動が異なる
    - varは宣言時に初期値がなければ、無条件でundefinedが入るが、letは値が入らないため、参照しようとするとエラーを起こす
    - letをnullで初期化する運用が正しいのでは？と思う。undefinedとnull問題もあるし

2.2.2 識別子の命名規則

- 記法の使い分け
    - 変数/関数名：camelCase 記法
    - 定数名：アンダースコア記法
    - クラス（コンストラクター）名：Pascal 記法
- 所感
    - → これも Java といっしょか

2.2.3 定数を宣言する

- Const 命令
    - 定数を宣言する
    - 大文字＋アンダースコアで定数名を定義すること

2.3 データ型

- データ型を強く意識する言語とそうでない言語がある
    - Java、C++などはデータ型を意識する
    - JavaScript はデータ型について寛容
        - 数値で定義していた変数に文字列を入れる事ができる
        - 入れ物がデータに合わせて大きさや形を変えてくれる

2.3.1 JavaScript の主なデータ型

基本型と参照型の違いに注意する

2.3.2 リテラル

テンプレート文字列を利用することで、複数行に渡る文字列表現が可能

```
let name = "鈴木";
let str = `こんにちは、${name}さん
今日はとても良い天気ですね！`;
console.log(str);
```

- 連想配列とオブジェクトは同じもの

```
var obj = { x:1, y:2, z:3 };
console.log(obj.x);
console.log(obj['x']);

オブジェクト名.プロパティ名　　<= ドット演算子
```

```
オブジェクト名.['プロパティ名'] <= ブラケット演算子
```

- Note:専用の連想配列 ES2015 では連想配列を専門に扱う仕組みとして Map が追加された
- Undefined と Null の違いについて

2.4 演算子

2.4.1 算術演算子

2.4.2 代入演算子

2.4.3 比較演算子

2.4.4 論理演算子

- const 定数定義 再代入不可
- 分割配列
  - 配列の代入方法だけ理解しておく
  - 見たことあるレベルで OK

```
let data = [56, 40, 26, 82, 19, 17, 73, 99];
let [x0, x1, x2, x3, x4, x5, x6, x7] = data;

console.log(x1);
console.log(x2);
console.log(x3);
console.log(x4);
console.log(x5);
console.log(x6);
console.log(x7);

// otherの書き方
let data = [56, 40, 26, 82, 19, 17, 73, 99];
let [x0, x1, x2, ...other] = data;

console.log(x0);
console.log(x1);
console.log(x2);
console.log(other);
```

- 分割代入
  - これは nodejs でよく見る書き方
  - json オブジェクトが module 化されていて、require で分割代入しているパターン有り

```
let book = {
  tittle: "ポケットリファレンス",
```

```
  publish: "技術評論社",
  price: 2680
};
let { price, title, memo = "なし" } = book;

console.log(title);
console.log(price);
console.log(memo);
```

- 入れ子となったオブジェクトを分解する

```
let book = {
  title: "ポケットリファレンス",
  publish: "技術評論社",
  price: 2680,
  other: { keywd: "Java SE 8", logo: "logo.jpg" }
};
let {
  title,
  other,
  other: { keywd }
} = book;

console.log(title);
console.log(other);
console.log(keywd);
```

- 変数の別名を指定する

```
let book = {
  title: "ポケットリファレンス",
  publish: "技術評論社",
  price: 2680,
  other: { keywd: "Java SE 8", logo: "logo.jpg" }
};
let { title: name, publish: company } = book;

console.log(name);
console.log(company);
```

- delete 演算子について理解する
- typeof 演算子
    - オペランドに指定した変数/リテラルのデータ型を表す文字列を返します

2.4.5 ビット演算子

2.4.6 その他の演算子

2.4.7 演算子の優先順位と結合測

2.5 制御構文

2.5.1 条件によって処理を分岐する - if 命令 -

2.5.2 式の値によって処理を分岐する - switch 命令 -

2.5.3 条件氏によってループを制御する - while／do...while 命令 -

2.5.4 無限ループ

2.5.5 指定回数だけループを処理する - for 命令 -

2.5.6 連想配列の要素を順に処理する - for...in 命令 -

- 配列では for ...in 命令は使用しない

```
var data = [ 'apple','orange','banana'];
// 配列オブジェクトにhogeメソッド追加
Array.prototype.hoge = function() {}
for (var key in data) {
  // キー情報が取れるので、これで配列取得
  // が、パフォーマンス悪いし、配列との相性悪いので使わないほうがよい
  console.log(data[key]);
}
```

2.5.7 配列などを順に処理する - for...of 命令 -

- ES2015で新しく追加された命令
- 配列などを順に処理する
    - for...of 命令
- 列挙可能なオブジェクトを順番に列挙するための方法

```
// 配列を操作するケース
var data = [ 'apple','orange','banana'];
// 配列オブジェクトにhogeメソッド追加
Array.prototype.hoge = function() {}
for (var value of data) {
  // 値を直接取る
  console.log(value);
}

// オブジェクト（連想配列）を操作する場合は、
```

```
// やはり、for ... in 命令が必須
var data = {
  apple: "りんご",
  orange: "みかん",
  banana: "ばなな"
}
Array.prototype.hoge = function() {}
for (let key in data) {
  // 日本語文字列が取れる
  console.log(key);
}
```

- 所感＋結論
  - 連想配列を操作する場合は`for ... in`で決まり
  - 配列や列挙可能なオブジェクトを操作する場合は、`for ... of`で決まり

2.5.8 ループを途中でスキップする／中断する - break／continue 命令 -

- Javaと一緒なので割愛

```
for  (var i = 1; i < 10; i++) {
  for (var j = 1; j < 10; j++) {
    var k = i * j;
    if (k > 30) break;
    document.write(k + ' ');
  }
  document.write('<br />');
}
```

- 所感
  - Javascriptにおける0埋めについて考えてみた件
  - 関数としては以下を使うべきらしい

```
function zeroPadding(num,length){
    return ('0000000000' + num).slice(-length);
}
```

- コラム：`document.write()`より`textContent/innerHTML`を優先して使うようにする

2.5.9 例外を処理する - try...catch...finally 命令 -

- 例外処理はオーバーヘッドが大きいので for などのループ文の中で使用するのは避けること

2.5.10 JavaScript の危険な構文を禁止する - Strict モード -

- strict を使用する

```
"use strict";
// 任意のコード

function hoge() {
  "use strict";
  // 関数の本体
}
```

- 新規の開発ではできるだけ strict モードを有効にすることをおすすめする
- with命令の禁止
    - https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Statements/with
    - with内で省略できるらしい

Chapter 3 基本データを操作する - 組み込みオブジェクト -

3.1 オブジェクトとは

3.1.1 オブジェクト=プロパティ＋メソッド

3.1.2 オブジェクトを利用するための準備 - new 演算子 -

3.1.3 静的プロパティ／静的メソッド

3.1.4 組み込みオブジェクトとは

- ！！基本データ型は new 演算子を利用しない

```
// 本来は「var flg = false;」と書くべき
var flag = new Boolean();

if (flag) {
  console.log("flagはtrueです");
}
```

3.2 基本データを扱うためのオブジェクト

3.2.1 文字列を操作する - String オブジェクト -

- 文字列操作関数は割愛
- サロゲートペア文字の長さをカウントする

```
var msg = "丈叱る";
console.log(msg.length);
```

```
// 改めて書いてみた
var msg = '叱る';
var len = msg.length;
var num = msg.split(/[\uD800-\uDBFF][\uDC00-\uDFFF]/g).length -1;
console.log(msg.length); // 3
console.log(num); // 1
console.log(msg.length - num); // 2

/**
 * 文字列長さ取得関数
 * サロゲートペア文字列対応
 */
function getStrLength(str) {
  var len = str.length;
  var num = str.split(/[\uD800-\uDBFF][\uDC00-\uDFFF]/g).length -1;
  return len - num;
}

/**
 * 配列取得
 */
function stringToArray (str) {
    return str.match(/[\uD800-\uDBFF][\uDC00-\uDFFF]|[^\uD800-\uDFFF]/g) || [];
}
```

- 算術演算子による文字列 ⇔ 数値の変換

```
console.log(typeof (123 + "")); // 結果: string <= ①
console.log(typeof ("123" - 0)); // 結果: number <= ②
```

- 個人的に使えるなと思ったメソッド
    - startsWith
    - endsWith
    - includes

3.2.2 数値を操作する - Number オブジェクト -

3.2.3 シンボルを作成する - Symbol オブジェクト -

- 定数で利用する

```
const MONDAY = Symbol();
const TUESDAY = Symbol();
const WEDNESDAY = Symbol();
const TURTHDAY = Symbol();
const FRYDAY = Symbol();
```

```
const SATURDAY = Symbol();
const SUNDAY = Symbol();

console.log(MONDAY == TUESDAY); // false
console.log(MONDAY === TUESDAY); // false
```

3.2.4 基本的な数学演算を実行する - Math オブジェクト -

- with演算子について言及がある
  - ブロック内の処理速度が低下する
  - そもそもコードが読みにくくなる
    - →実際のアプリでは利用すべきではない

3.3 値の集合を管理／操作する - Array／Map／Set オブジェクト -

3.3.1 配列を操作する - Array オブジェクト -

- Array オブジェクトをコンストラクタで表現すると、いみが曖昧になる場合がある

```
var ary = new Array("佐藤", "高江", "長田");
var ary = new Array();
var ary = new Array(10); //指定サイズ（インデックスは0-9）で空の入れるを生成
// 10という数値の入った配列を作成したいのか、
// 長さ10の配列を作成したいのか曖昧
```

- Javascriptで配列を宣言するときは、原則として、[]を用いること！

```
// function関数の仕組みについて考察

// Http通信を受信する処理
var rcvHttp = function() {
  // リクエスト処理
  console.log("httpリクエスト受信");

  // 引数仮作成
  event = {};
  context = {};
  // ファンクション呼び出し
  myFunction(event, context, callback);

  // 締め処理
  console.log("httpレスポンス送信");
};

callback = function(err, message) {
  // 処理成功
  if (!err) {
    console.log(message);
```

```
  } else {
    console.log(err.msg);
  }
};

// 実装するFunction関数
var myFunction = function(event, context, callback) {
  console.log("--処理開始--");
  // 関数の実処理
  console.log("--処理終了--");

  // エラーオブジェクトを仮作成
  var err = { state: "error", msg: "error function!" };

  var isSuccess = true; // 処理成功!
  if (isSuccess) {
    callback(null, "success");
  } else {
    callback(err);
  }
};

// 処理開始
rcvHttp();
```

- forEachメソッド
  - 配列の要素内を指定した関数で順番に処理する
- mapメソッド
  - 配列を加工して新しい配列を返却する
- someメソッド
  - 1つの要素が条件に合致するかどうか
- everyメソッド
  - すべての要素が条件に合致するかどうか
- filterメソッド
  - 条件に合致した要素だけ取り出す

```
// forEachメソッドの例
// value:要素の値
// index:インデックス番号
// array:元の配列
var data = [2,3,4,5];
data.forEach((value,index,array)=>{
console.log(array);
console.log(index + " : " + value);
});

// mapメソッドの例
// 引数はforEachと同様
let now = new Date();
let len = 10;
```

```
let array = new Array(len).fill(null).map((_, i) => {
  // 初期値設定ロジック
  let obj = {
    date: new Date(now.setDate(now.getDate() + 1)).toISOString().split('.')[0],
    random: Math.floor( Math.random() * 11 )
  }
  return obj;
});

// someメソッドの例
var data = [2,3,4,5];
data.some((value,index,array)=>{
  return value % 3 === 0;
});

// everyメソッドの例
var data = [2,3,4,5];
data.every((value,index,array)=>{
  return value % 3 === 0;
});

// filterメソッドの例
var data = [2,3,4,5];
var result = data.filter((value,index,array)=>{
  return value % 2 === 1;
});

console.log(result);

// sortメソッドの例
// 数値の並べ替え
var ary = [5, 25, 10];
console.log(ary.sort());
console.log(ary.sort(function(x, y) {
  return x - y;
}));

// 役職の並べ変え
var classes = ['部長', '課長', '主任', '担当'];
var members = [
  { name: '鈴木清子', clazz: '主任' },
  { name: '山口久雄', clazz: '部長' },
  { name: '井上太郎', clazz: '担当' },
  { name: '和田知美', clazz: '課長' },
  { name: '小森雄太', clazz: '担当' },
];
var i = 0;
console.log(members.sort(function(x, y) {
  console.log(i++ + "回目: " + x.name + " -> " + y.name);
  return classes.indexOf(x.clazz) - classes.indexOf(y.clazz);
}))
```

3.3.2 連想配列を操作する - Map オブジェクト -

- オブジェクトリテラルとの違い
  - 任意の型でキーを指定できる
    - オブジェクトリテラルはプロパティ名をキーとして代替していたので、キーとして利用できるのは文字列だけ
    - Mapオブジェクトでは任意の型をキーとして利用できる
    - オブジェクトや、NaNすらキーになる
  - マップのサイズを取得できる
    - Mapオブジェクトでは`size`プロパティで登録されたキー/値の個数を取得できる
  - クリーンなマップを作成できる
- キーに関わる３つの中移転
  - キーは「===(厳密等価演算子)」で評価される
  - 特別なNaNは特別ではない
    - 通常、NaNは自分自身とも等しくない特別な値
    - しかし、Mapの世界では例外的にNaN === NaNとみなされます。
  - オブジェクトの比較には要注意
    - オブジェクトをキーにすることはできるが、異なる参照を持つオブジェクトは
    - 異なるキーとみなされることに注意
    - これはJavaの考え方と同様

```
var m = new Map();
m.set({}, 'hoge');
console.log(m.get({}));
// -> undefined
// {}を書いた時点で別オブジェクトとして生成されているため
```

3.3.3 重複しない値の集合を操作する - Set オブジェクト -

- 重複しない値の集合を管理するためのオブジェクト

```
let s = new Set();
s.add(10);
s.add(5);
s.add(100);
s.add(50);
s.add(5);

//let s = new Set([10, 5, 100, 50, 5]);

console.log(s.has(100));
console.log(s.size);

for (let val of s.values()) {
  console.log(val);
```

```
}

for (let val of s) {
  console.log(val);
}

s.delete(100);
console.log(s.size);

s.clear();
console.log(s.size);
```

3.4 日付／時刻データを操作する - Date オブジェクト -

3.4.1 Date オブジェクトを生成する

- Javascript標準のデータ型としてdate型は存在しません。
- 組み込みオブジェクトDateを利用すれば、日付を直感的に表現/操作できます。

```
var dat = new Date(2016, 11, 25, 11, 37, 15, 999);
console.log(dat);
console.log(dat.getFullYear());
console.log(dat.getMonth());
console.log(dat.getDate());
console.log(dat.getDay());
console.log(dat.getHours());
console.log(dat.getMinutes());
console.log(dat.getSeconds());
console.log(dat.getMilliseconds());
console.log(dat.getTime());
console.log(dat.getTimezoneOffset());

console.log(dat.getUTCFullYear());
console.log(dat.getUTCMonth());
console.log(dat.getUTCDate());
console.log(dat.getUTCDay());
console.log(dat.getUTCHours());
console.log(dat.getUTCMinutes());
console.log(dat.getUTCSeconds());
console.log(dat.getUTCMilliseconds());

var dat2 = new Date();
dat2.setFullYear(2017);
dat2.setMonth(7);
dat2.setDate(5);
dat2.setHours(11);
dat2.setMinutes(37);
dat2.setSeconds(15);
dat2.setMilliseconds(513);
```

```
console.log(dat2.toLocaleString());
console.log(dat2.toUTCString());
console.log(dat2.toDateString());
console.log(dat2.toTimeString());
console.log(dat2.toLocaleDateString());
console.log(dat2.toLocaleTimeString());
console.log(dat2.toJSON());

console.log(Date.parse('2016/11/05'));
console.log(Date.UTC(2016, 11, 5));
console.log(Date.now());
```

- 今月の最終日のとり方

```
var dat = new Date(2017, 4, 15, 11, 40);
console.log(dat.toLocaleString());
dat.setMonth(dat.getMonth() + 1);
dat.setDate(0);
console.log(dat.toLocaleString());
```

- 日付/時刻の差分を求める

```
// 月差分
var dat1 = new Date(2017, 4, 15);
var dat2 = new Date(2017, 5, 20);
var diff = (dat2.getMonth() - dat1.getMonth());
console.log(diff + '月の差があります。');

// 日付差分
var dat1 = new Date(2017, 4, 15);
var dat2 = new Date(2017, 5, 20);
var diff = (dat2.getTime() - dat1.getTime()) / (1000 * 60 * 60 * 24);
console.log(diff + '日の差があります。');

// 時間差分
var dat1 = new Date(2017, 4, 15);
var dat2 = new Date(2017, 5, 20);
var diff = (dat2.getTime() - dat1.getTime()) / (1000 * 60 * 60);
console.log(diff + '時間の差があります。');

// 分差分
var dat1 = new Date(2017, 4, 15);
var dat2 = new Date(2017, 5, 20);
var diff = (dat2.getTime() - dat1.getTime()) / (1000 * 60);
console.log(diff + '分の差があります。');
```

3.4.2 日付／時刻値を加算／減算する

3.4.3 日付／時刻の差分を求める

- 日付を求めるJSピュアライブラリー
  - https://github.com/januswel/jslib

Nodejs環境があるのであれば

moments jsを使う運用にする

3.5 正規表現で文字を自在に操作する - RegExp オブジェクト -

3.5.1 JavaScript で利用可能な正規表現

3.5.2 RegExp オブジェクトを生成する方法

- `String.match`メソッドを使うか
- `RegExp.exec`メソッドを使うか
- `RegExp.test`メソッド
- `String.Search`メソッド
- 文字列置き換え
  - `String.replace`メソッド
- 文字列分割
  - `String.split`メソッド

3.5.3 正規表現による検索の基本

3.5.4 正規表現のオプションでマッチング時の挙動を制御する

3.5.5 match メソッドと exec メソッドの挙動の違い

3.5.6 マッチングの成否を検証する

3.5.7 正規表現で文字列を置き換える

3.5.8 正規表現で文字列を分割する

3.6 すべてのオブジェクトのひな形 - Object オブジェクト -

- オブジェクトの共通的な性質/機能を提供する

- すべてのオブジェクトの基本オブジェクトである。

3.6.1 オブジェクトを基本型に変換する - toString／valueOf メソッド -

- オブジェクトを作成するのであればこの２つはそれぞれ実装する

```
var obj = new Object();
console.log(obj.toString());
console.log(obj.valueOf());

var dat = new Date();
console.log(dat.toString());
console.log(dat.valueOf());

var ary = ['prototype.js', 'jQuery', 'Yahoo! UI'];
console.log(ary.toString());
console.log(ary.valueOf());

var num = 10;
console.log(num.toString());
console.log(num.valueOf());

var reg = /[0-9]{3}-[0-9]{4}/g;
console.log(reg.toString());
console.log(reg.valueOf());
```

- 実装してみる

```
function Book(title, author) {
    this.title = title;
    this.author = author;
}

Book.prototype.toString = function() {
    return 'title=' + this.title + ', author=' + this.author;
}

var b = new Book('Yes we can', 'Maku');
alert(b);  //=> title=Yes we can, author=Maku
```

3.6.2 オブジェクトをマージする - assign メソッド -

- オブジェクトをマージする

```
let pet = {
  type: "スノーホワイトハムスター",
  name: "キラ",
```

```
  description: {
    birth: "2014-02-15"
  }
};

let pet2 = {
  name: "山田きら",
  color: "白",
  description: {
    food: "ひまわりのタネ"
  }
};

let pet3 = {
  weight: 42,
  photo: "http://www.wings.msn.to/img/ham.jpg"
};

Object.assign(pet, pet2, pet3);
console.log(pet);

// 注意!
// 先頭引数のオブジェクトが上書きされてしまうので、避けたい場合は空オブジェクトを指定して回避
// let merged = Object.assign({}, pet, pet2, pet3);
// console.log(merged);
```

- 匿名オブジェクトを生成する方法

```
var obj = new Array();
obj.name = "トクジロウ";

// この様に、Object以外のオブジェクトを使用して匿名オブジェクトを生成できるが、バグのもとになる可能性があるので、最低限の機能を持つobjectで生成するのがよい
```

3.6.3 オブジェクトを生成する - create メソッド -

```
//  完全に空のオブジェクトを生成する
var emp = Object.create(null);
console.log(emp);
// -> プロパティが設定されていないことを確認


//  デフォルトでObjectのプロパティを引き継ぐ
var obj = { x:1, y:2, z:3 };

var obj2 = new Object();
obj2.x = 1;
obj2.y = 2;
```

```
obj2.z = 3;

var obj3 = Object.create(Object.prototype, {
  x: { value: 1, writable: true, configurable: true, enumerable: true},
  y: { value: 2, writable: true, configurable: true, enumerable: true},
  z: { value: 3, writable: true, configurable: true, enumerable: true}
  }
);

for (var prop in obj3) {
   console.log(prop)
}
```

3.6.4 不変オブジェクトを定義する

- preventExtensions
- seal
- freeze

```
"use strict"; //strictモードにしないと、制約がきかない

var pet = { type: "スノーホワイトハムスター", name: "キラ" };

// コメントを外して挙動確認
//Object.preventExtensions(pet);
//Object.seal(pet);
//Object.freeze(pet);

pet.name = "山田きら";
delete pet.type;
pet.weight = 42;

console.log(pet);

// Chromeコンソールで挙動を確認
```

3.7 JavaScript プログラムでよく利用する機能を提供する - Global オブジェクト -

- グローバル変数やグローバル関数を管理するために JavaScript が自動的に生成する便宜的なオブジェクト
- Number オブエジェクトに移動したメソッドも存在する
- クエリ情報をエスケープ処理する
    - escape関数も存在するが、ブラウザやプラットフォームによって得られる結果が異なってくるので使用しないこと
- 動的に生成したスクリプトを実行する - eval 関数 -
    - 与えられた文字列を JavaScript のコードとして評価/実行する

```
var str = 'console.log("eval関数");';
eval(str);
```

- JSON (Javascript object notation) javascript object 表記法
- eval 関数を利用したくなるようなものには代替策が用意されている
- 「eval is ecvil(eval 関数は邪悪)」なのです。

3.7.1 Number オブジェクトに移動したメソッド

3.7.2 クエリ情報をエスケープ処理する - encodeURI／encodeURIComponent 関数 -

3.7.3 動的に生成したスクリプトを実行する - eval 関数 -

Chapter 4 くり返し利用するコードを 1 箇所にまとめる - 関数 -

4.1 関数とは

4.1.1 function 命令で定義する

- Function 名（関数名）について注意
    - 識別子の条件を満たす必要がある
    - その関数がどのような処理を担っているかすぐわかるような名前をつける
        - 「showMessage」のように「動詞＋名詞」の形式で命名す
- returnを明示的に関数に書かないと、undefinedが返却される

4.1.2 Function コンストラクター経由で定義する

```
var 変数名 = new Function(引数,... ,関数本体);
```

- 基本的に使用するべきではない
    - 読みづらい
    - 余計なコーディングミスも増える
    - Function コンストラクターを使用するメリットはない
- 記述するとすればどこ？
    - while/for などの繰り返しブロックの中以外
    - 頻繁に呼び出される関数の中以外

4.1.3 関数リテラル表現で定義する

```
var getTriangle = function(base, height) {
  return base * height / 2;
};
```

```
console.log('三角形の面積: ' + getTriangle(5, 2));
```

4.1.4 アロー関数で定義する

```
let getTriangle = (base, height) => {
  return base * height / 2;
};

console.log('三角形の面積: ' + getTriangle(5, 2));
```

4.2 関数定義における 4 つの注意点

4.2.1 return 命令の直後で改行しない

4.2.2 関数はデータ型の一種

```
var getTriangle = function(base, height) {
  return (base * height) / 2;
};

console.log(getTriangle(5, 2));
getTriangle = 0;
console.log(getTriangle);

// 関数gettriangleに数値型を入れては行けないト感じるが、
// javascriptではgettriangleに入っているのはあくまで関数の参照型で、型定義を持たないのであり！
```

4.2.3 function 命令は静的な構造を宣言する

```
console.log("三角形の面積: " + getTriangle(5, 2));

function getTriangle(base, height) {
  return (base * height) / 2;
}

// Function命令ははコードを解析/コンパイルするタイミングで、関数登録している
// この順序で記載しても問題なく呼び出せる
```

- Note：<script>要素は呼び出し側より先に記述する

    - 関数を定義したスクリプトブロックは呼び出し側のスクリプトブロックより前、あるいは同じスクリプトブロックに記述されなければなりません。

4.2.4 関数リテラル／Function コンストラクターは実行時に評価される

- 前述の例を関数リテラルで書き換えたらうまくいくかどうか

```
console.log("三角形の面積: " + getTriangle(5, 2));

var getTriangle = function(base, height) {
  return (base * height) / 2;
};

// 実行時エラーになりうまくいかない
```

- 改めて Funciton を定義する４つの方法を整理
    - Function 命令
        - 静的に解析される
    - Function コンストラクタ
        - できるだけ用いるべきではない！
    - 関数リテラル
        - 動的に解析される
    - アロー関数
        - 関数リテラルをよりシンプルに表記するための方法

4.3 変数はどの場所から参照できるか - スコープ -

4.3.1 グローバル変数とローカル変数の違い

```
var scope = "Global Variable";
function getValue() {
  var scope = "Local Variable";
  return scope;
}

console.log(getValue());
console.log(scope);
```

4.3.2 変数宣言に var 命令が必須な理由

```
var scope = "Global Variable";
function getValue() {
  scope = "Local Variable";
  return scope;
}
```

```
console.log(getValue());
console.log(scope);

// 以下も同じ挙動

scope = "Global Variable";
function getValue() {
  scope = "Local Variable";
  return scope;
}

console.log(getValue());
console.log(scope);

// var宣言がない場合すべてグローバル変数で扱われる！！！
```

4.3.3 ローカル変数の有効範囲はどこまで

- 以下のような場合の挙動を確認する

```
var scope = "Global Variable";
function getValue() {
  console.log(scope); //ここではグローバル変数を参照できずにundefinedとなる
  var scope = "Local Variable";
  return scope;
}

console.log(getValue());
console.log(scope);
```

- 「変数の巻き上げ」が発生するため
- Javascript では「**ローカル変数は関数の先頭で宣言する**」ことを常に意識する
    - これは、「**変数はできるだけ利用する場所の近くで宣言する**」という他の言語でのそれに反するので注意が必要です。

4.3.4 仮引数のスコープ - 基本型と参照型の違いに注意する -

- 直感的に理解できるので省略

4.3.5 ブロックレベルのスコープは存在しない（ES2015 以前）

```
if (true) {
  var i = 5;
}

console.log(i);

// この書き方はjavascriptだとうまくいく
```

```
// 即時関数 というテクニック
(function() {
  var i = 5;
  console.log(i);
}.call(this));

console.log(i);

// このような書き方であれば、iは関数の中でしか有効ではないので、最後の処理でエラーになる
// ただ、letで同じ効果を得られるので使わない
```

4.3.6 ブロックスコープに対応した let 命令

```
// 即時関数は使用しなくても同様の結果が得られるのでこちらを使用すること
{
  let i = 5;
  console.log(i);
}

console.log(i);

// 変数名の重複になってしまうので、switchの外で宣言しておくこと

switch (x) {
  case 0:
    let value = "x:0";
  case 1:
    let value = "x:1";
}
```

4.3.7 関数リテラル／Function コンストラクターにおけるスコープの違い

```
var scope = "Global Variable";

function checkScope() {
  var scope = "Local Variable";

  var f_lit = function() {
    return scope;
  };
  console.log(f_lit());

  var f_con = new Function("return scope;");
  console.log(f_con());
}
```

```
checkScope();
```

- Functionコンストラクタはグローバル変数を参照してしまう
- Functionコンストラクタは原則使用しないルールなので、このような混乱は生じにくいが覚えておく

4.4 引数のさまざまな記法

4.4.1 JavaScript は引数の数をチェックしない

```
function showMessage(value) {
  console.log(value);
}

showMessage();
showMessage("山田");
showMessage("山田", "鈴木");

// javascriptでは引数を余分に渡しても無視されることを認識する
```

```
function showMessage(value) {
  if (arguments.length !== 1) {
    throw new Error("引数の数が間違っています：" + arguments.length);
  }
  console.log(value);
}

try {
  showMessage(" 山田", " 鈴木");
} catch (e) {
  window.alert(e.message);
}

// 引数の数をチェックしたい場合はこの様に記載する
```

```
function getTriangle(base, height) {
  if (base === undefined) {
    base = 1;
  }
  if (height === undefined) {
    height = 1;
  }
  return (base * height) / 2;
}

console.log(getTriangle(5));
```

```
// 引数のデフォルトを設定する方法
// 省略できるのは２番めの引数のみ。さいしょの引数は省略できない

// このように書くことはできる
console.log(getTriangle(undefined, 5));
```

4.4.2 可変長引数の関数を定義する

```
function sum() {
  var result = 0;
  for (var i = 0, len = arguments.length; i < len; i++) {
    var tmp = arguments[i];
    if (typeof tmp !== "number") {
      throw new Error("引数が数値ではありません：" + tmp);
    }
    result += tmp;
  }
  return result;
}

try {
  console.log(sum(1, 3, 5, 7, 9));
} catch (e) {
  window.alert(e.message);
}

// 可変長として提供すればこのようなこともできる
```

4.4.3 明示的に宣言された引数と可変長引数を混在させる

- printf の例
    - 割愛

4.4.4 名前付き引数でコードを読みやすくする

- 引数に渡す値としてオブジェクトリテラルを用いることで実現している

```
function getTriangle(args) {
  if (args.base === undefined) {
    args.base = 1;
  }
  if (args.height === undefined) {
    args.height = 1;
  }
  return (args.base * args.height) / 2;
}
```

```
console.log(getTriangle({ base: 5, height: 4 }));
```

- 使用する場面
    - そもそも引数の数が多い
    - 省略可能な引数が多く、省略パターンにも様々な組み合わせがある

4.5 ES2015 における引数の記法

- 新構文によって、argumentオブジェクトがほぼ不要
- Javascript特有の冗長なコードからも解放

4.5.1 引数のデフォルト値

「仮引数＝デフォルト値」

```
function getTriangle(base = 1, height = 1) {
  return base * height / 2;
}

console.log(getTriangle(5));
//console.log(getTriangle(5, null));
//console.log(getTriangle(5, undefined));
```

以前の書き方だとこんな感じ

```
"use strict";

function getTriangle() {
  var base = arguments.length <= 0 || arguments[0] === undefined ? 1 :
arguments[0];
  var height = arguments.length <= 1 || arguments[1] === undefined ? 1 :
arguments[1];

  return base * height / 2;
}

console.log(getTriangle(5));
//console.log(getTriangle(5, null));
//console.log(getTriangle(5, undefined));
```

- デフォルト値はリテラルだけじゃなくて、他の引数や式も設定できる
- 但し自分より前に宣言されたものに限る
- 以下のような場合、undefinedになる

```
function multi(a = b, b = 5) {
  return a * b;
}

console.log(multi());
```

- デフォルト値を利用する場合の注意点
  - （1）デフォルト値が適用される場合、されない場合
    - nullだとデフォルト値が適用されない
    - undefinedだとデフォルト値が適用される

4.5.2 可変長引数の関数を定義する

- 「...」（ピリオド3個）を付与することで、可変長引数となる（英語ではRest Parameterと表記）
- 1）関数が可変長引数を受け取ることがわかりやすい
- 2）すべての配列操作が可能
  - argumentsオブジェクトは、lengthプロパティやブラケット構文を持っていることから、間違われやすいが、実態はArrayオブジェクトではない
  - 配列のように扱えるだけのArrayライクなオブジェクトに過ぎない
  - なので、可変長引数の一部を除去/追加するために、Push/shiftなどのメソッドを利用することはできない

4.5.3 「...」演算子による引数の展開

```
console.log(Math.max(...[15,-3,78,1]));
// 展開される
```

4.5.4 名前付き引数でコードを読みやすくする

```
function show({name}) {
  console.log(name);
};

let member = {
  mid: 'Y0001',
  name: '山田太郎',
  address: 't_yamada@example.com'
};

show(member);
```

4.6 関数呼び出しと戻り値

4.6.1 複数の戻り値を個別の変数に代入する

理解済み

4.6.2 関数自身を再帰的に呼び出す - 再帰関数 -

理解済み

再帰関数の実際のビジネスロジック活用シーンが微妙

フォルダの再起検索ぐらい

4.6.3 関数も引数も関数 - 高階関数 -

- 「関数を引数、戻り値として扱う関数」のことを 高階関数 と呼ぶ

4.6.4 「使い捨ての関数」は匿名関数で

理解済み

Javascriptプログラマーが好んで利用する書き方

- メリット
  - グローバルスコープで名前を定義せずに済むので意図せぬ名前の重複を回避できる
  - コード量を減らせる

4.7 高度な関数のテーマ

4.7.1 テンプレート文字列をアプリ仕様にカスタマイズする - タグ付きテンプレート文字列 -

```
'use strict';

var _templateObject = _taggedTemplateLiteral(['こんにちは、', 'さん！'], ['こんにち
は、', 'さん！']);

function _taggedTemplateLiteral(strings, raw) { return
Object.freeze(Object.defineProperties(strings, { raw: { value: Object.freeze(raw)
} })); }

function escapeHtml(str) {
  if (!str) {
    return '';
  }
  str = str.replace(/&/g, '&amp;');
  str = str.replace(/</g, '&lt;');
  str = str.replace(/>/g, '&gt;');
  str = str.replace(/"/g, '&quot;');
  str = str.replace(/'/g, '&#39;');
  return str;
}
```

```
function e(templates) {
  var result = '';

  for (var _len = arguments.length, values = Array(_len > 1 ? _len - 1 : 0), _key
= 1; _key < _len; _key++) {
    values[_key - 1] = arguments[_key];
  }

  for (var i = 0, len = templates.length; i < len; i++) {
    result += templates[i] + escapeHtml(values[i]);
  }
  return result;
}

var name = '<"Mario" & \'Luigi\'>';
console.log(e(_templateObject, name));
```

4.7.2 変数はどのような順番で解決されるか - スコープチェーン -

4.7.3 その振る舞いオブジェクトの如し - クロージャ -

```
function closure(init) {
  var counter = init;

  return function() {
    return ++counter;
  }
}

var myClosure1 = closure(1);
var myClosure2 = closure(100);

console.log(myClosure1());
console.log(myClosure2());
console.log(myClosure1());
console.log(myClosure2());
```

Chapter 5 大規模開発でも通用する書き方を身につける - オブジェクト指向構文 -

5.1 JavaScript におけるオブジェクト指向の特徴

5.1.1 「クラス」はなく「プロトタイプ」

- プロトタイプベースのオブジェクト指向
- クラスベースのオブジェクト指向ではない

5.1.2 最もシンプルなクラスを定義する

- javascriptではfunctionオブジェクトにクラスとして役割を与えている
- Note:アロー関数ではコンストラクタは定義できない

5.1.3 コンストラクターで初期化する

- 厳密にはクラスと呼ぶより、コンストラクタと呼ぶのが正しい
- コンストラクタの名前は普通の関数と区別するために大文字で始めるのが一般的

```
var Member = function(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.getName = function() {
    return this.lastName + ' ' + this.firstName;
  }
};

var mem = new Member('祥寛', '山田');
console.log(mem.getName());
```

5.1.4 動的にメソッドを追加する

- オブジェクトをNEWしてからメソッドを新しく追加することができる
- ゆるいオブジェクト指向
- オブジェクトをNEWしてから、変更を許したくない場合は、コンストラクタの最後に、`Object.seal(this);`と記載することで対応できる

5.1.5 文脈によって中身が変化する変数 - this キーワード -

- 配列ライクなものを配列に変換する`Array.from`メソッド
  - https://sbfl.net/blog/2018/07/04/javascript-array/#ArrayfromarrayLike_mapFn_undefined_thisArg_undefined
  - `arguments`や`NodeList`を、配列に変換することができる

5.1.6 コンストラクターの強制的な呼び出し

- コンストラクタの呼び出しを強制するための構文

```
// thisがGlobalを指してしまう
var Member = function(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
};

var m = Member('権兵衛', '佐藤');
```

```
console.log(m);
console.log(firstName);
console.log(m.firstName);

// コンストラクタ呼び出しを強制する方法
var Member = function(firstName, lastName) {
  if(!(this instanceof Member)) {
    return new Member(firstName, lastName);
  }
  this.firstName = firstName;
  this.lastName = lastName;
  this.getName = function(){
    return this.lastName + ' ' + this.firstName;
  }
};

var m = Member('権兵衛', '佐藤');

console.log(m);
console.log(m.firstName);
```

5.2 コンストラクターの問題点とプロトタイプ

- コンストラクタにメソッドを定義してしまうと、NEW演算子で作られたオブジェクト分それぞれ違いインスタンスができてしまい、メモリを圧迫する恐れがある
- メソッドの振る舞いがNEWされたインスタンスで同じであれば、コンストラクタにメソッドを定義するべきではない
- 上記の場合、プロトタイプにメソッドを定義することで、異なるインスタンスごとに共通のメソッドを呼び出すことができる

```
var Member = function(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;
};

Member.prototype.getName = function() {
  return this.lastName + ' ' + this.firstName;
};

var mem = new Member('祥寛', '山田');
console.log(mem.getName());

var mem2 = new Member('祥寛2', '山田2');
console.log(mem.getName());

// 同じ、getName()メソッドをプロトタイプ経由で呼び出せる
```

5.2.1 メソッドはプロトタイプで宣言する - prototype プロパティ -

5.2.2 プロトタイプオブジェクトを利用することの 2 つの利点

5.2.3 プロトタイプオブジェクトの不思議（1） - プロパティの設定 -

- プロトタイプのプロパティをオブジェクトのプロパティで隠蔽することができる説明

5.2.4 プロトタイプオブジェクトの不思議（2） - プロパティの削除 -

- `delete`演算子はプロパティを削除するだけ
- 返り値はboolean
- オブジェクトに適用しようとすると、falseが返却され、オブジェクトを削除することはできない

5.2.5 オブジェクトリテラルでプロトタイプを定義する

```
var Member = function(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
};

Member.prototype = {
  getName : function() {
    return this.lastName + ' ' + this.firstName;
  },
  toString : function() {
    return this.lastName + this.firstName;
  }
};

var mem = new Member('祥寛', '山田');
console.log(mem.getName());
console.log(mem.toString());
```

5.2.6 静的プロパティ／静的メソッドを定義する

```
var Area = function() {};

Area.version = '1.0';

Area.triangle = function(base, height) {
  return base * height / 2;
};

Area.diamond = function(width, height) {
  return width * height / 2;
};
```

```
console.log('Areaクラスのバージョン：' + Area.version);
console.log('三角形の面積：' + Area.triangle(5, 3));

var a = new Area();
console.log('菱形の面積：' + a.diamond(10, 2));
```

5.3 オブジェクト継承 - プロトタイプチェーン -

```
var Animal = function() {};

Animal.prototype = {
  walk : function() {
    console.log('トコトコ...');
  }
};

var Dog = function() {
  Animal.call(this);
};
Dog.prototype = new Animal();
Dog.prototype.bark = function() {
  console.log('ワンワン！ ');
}

var d = new Dog();
d.walk();
d.bark();

// このように書いても、あくまでプロトタイプに定義されたメソッドが継承されているだけで、コンストラクタや静的メソッドが引き継がれているわけではない
// このように書くとコンストラクタ関数がプロトタイプよりも優先される結果となった
// ドタバタ...
// ワンワン！
var Animal = function() {
  this.walk = function() {
    console.log("ドタバタ...");
}
};
Animal.walk = function() {console.log("ダンダン...")}

Animal.prototype = {
  walk : function() {
    console.log('トコトコ...');
  }
};

var Dog = function() {
  Animal.call(this);
};
```

```
Dog.prototype = new Animal();
Dog.prototype.bark = function() {
  console.log('ワンワン！ ');
}

var d = new Dog();
d.walk();
d.bark();
```

5.3.1 プロトタイプチェーンの基礎

5.3.2 継承関係は動的に変更可能

```
var Animal = function() {};

Animal.prototype = {
  walk : function() {
    console.log('トコトコ...');
  }
};

var SuperAnimal = function() {};
SuperAnimal.prototype = {
  walk : function() {
    console.log('ダダダダダ！ ');
  }
};

var Dog = function() {};
Dog.prototype = new Animal();
var d1 = new Dog();
d1.walk(); // トコトコ...

Dog.prototype = new SuperAnimal();
var d2 = new Dog();
d2.walk(); // ダダダダダ！
d1.walk(); // トコトコ... オブジェクトが生成されたタイミングで継承されているオブジェクト情報が入る

// 通常、途中で継承関係を変えるようなコードを書くことはないと思うのであまりこのコードは参考にならないが、
// このような挙動になることだけ抑えておく
```

5.3.3 オブジェクトの型を判定する

- 元となるコンストラクタを取得する
- 元となるコンストラクタを判定する instanceof 演算子

- 参照しているプロトタイプを確認する isPrototypeOf
- メンバー有無を判定する in 演算子

コンストラクタがなん何かを判定する機会がそもそもあまりない

メンバー有無の判定は使えそう

```
var obj = { hoge: function(){}, foo: function(){} };

console.log('hoge' in obj);
console.log('piyo' in obj);
```

5.4 本格的な開発に備えるために

5.4.1 プライベートメンバーを定義する

```
function Triangle() {
  // プライベートのプロパティを定義
  var _base;
  var _height;
  // プライベートのメソッドを定義
  var _checkArgs = function(val) {
    return (typeof val === 'number' && val > 0);
   }

  this.setBase = function(base) {
    if (_checkArgs(base)){ _base = base; }
  }
  this.getBase = function() { return _base; }

  this.setHeight = function(height) {
    if (_checkArgs(height)){ _height = height; }
  }
  this.getHeight = function() { return _height; }
}

Triangle.prototype.getArea = function() {
  return this.getBase() * this.getHeight() / 2;
}

var t = new Triangle();
t._base = 10;
t._height = 2;
console.log('三角形の面積: ' + t.getArea());

t.setBase(10);
t.setHeight(2);
console.log('三角形の底辺: ' + t.getBase());
console.log('三角形の高さ: ' + t.getHeight());
```

```
console.log('三角形の面積: ' + t.getArea());
```

5.4.2 Object.defineProperty メソッドによるアクセサーメソッドの実装

```
function Triangle() {
  var _base;
  var _height;

  Object.defineProperties(this, {
    base: {
      get: function() {
        return _base;
      },
      set: function(base) {
        if(typeof base === 'number' && base > 0) {
          _base = base;
        }
      }
    },
    height: {
      get: function() {
        return _height;
      },
      set: function(height) {
        if(typeof height === 'number' && height > 0) {
          _height = height;
        }
      }
    }
  });
}

Triangle.prototype.getArea = function() {
  return this.base * this.height / 2;
};

var t = new Triangle();
t.base = 10;
t.height = 2;
console.log('三角形の底辺: ' + t.base);
console.log('三角形の高さ: ' + t.height);
console.log('三角形の面積: ' + t.getArea());
```

5.4.3 名前空間／パッケージを作成する

```
function namespace(ns) {
  var names = ns.split('.');
```

```
  var parent = window;

  for (var i = 0, len = names.length; i < len; i++) {
    parent[names[i]] = parent[names[i]] || {};
    parent = parent[names[i]];
  }
  return parent;
}

var my = namespace('Wings.Gihyo.Js.MyApp');
my.Person = function() {};
var p = new my.Person();
console.log(p instanceof Wings.Gihyo.Js.MyApp.Person);
```

5.5 ES2015 のオブジェクト指向構文

5.5.1 クラスを定義する - class 命令 -

5.5.2 オブジェクトリテラルの改善

- オブジェクトの中にメソッドを定義できるようになった
- 変数を同名のプロパティに割り当てる
- プロパティを動的に生成する

```
// オブジェクトの中にfunctionを記載できる
let member = {
  name: '山田太郎',
  birth: new Date(1970, 5, 25),
  toString() {
    return this.name + '／誕生日: ' + this.birth.toLocaleDateString()
  }
};

console.log(member.toString());

// プロパティの変数名を使ってキーを表現する
let name = '山田太郎';
let birth = new Date(1970, 5, 25);
let member = { name, birth };

console.log(member);


// プロパティを動的に生成する
let i = 0;
let member = {
  name: '山田太郎',
  birth: new Date(1970, 5, 25),
  ['memo' + ++i]: '正規会員',
  ['memo' + ++i]: '支部会長',
```

```
  ['memo' + ++i]: '関東'
};

console.log(member);
```

5.5.3 アプリを機能単位にまとめる - モジュール -

import記法

5.5.4 列挙可能なオブジェクトを定義する - イテレーター -

[Symbol.iterator]

5.5.5 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 -

```
function* genPrimes() {
  let num = 2;
  while (true) {
    if (isPrime(num)) { yield num; }
    num++;
  }
}

function isPrime(value) {
  let prime = true;
  for (let i = 2; i <= Math.floor(Math.sqrt(value)); i++) {
    if (value % i === 0) {
      prime = false;
      break;
    }
  }
  return prime;
}

for(let value of genPrimes()) {
  if (value > 100) { break; }
  console.log(value);
}
```

5.5.6 オブジェクトの基本的な動作をカスタマイズする - Proxy オブジェクト -

```
let data = { red: '赤色', yellow: '黄色' };
var proxy = new Proxy(data, {
  get(target, prop) {
    return prop in target ? target[prop] : '?';
```

```
  }
});

console.log(proxy.red);
console.log(proxy.nothing);

proxy.red = 'レッド';
console.log(data.red);
console.log(proxy.red);
```

Chapter 6 HTML や XML の文書を操作する - DOM（Document Object Model） -

6.1 DOM の基本を押さえる

DOMについて概要を説明

6.1.1 マークアップ言語を操作する標準のしくみ「DOM」

```
var current = new Date();
var result = document.getElementById("result");
result.textContent = current.toLocaleString(); // textContentはよく使う
```

6.1.2 文書ツリーとノード

6.2 クライアントサイド JavaScript の前提知識

6.2.1 要素ノードを取得する

6.2.2 文書ツリー間を行き来する - ノードウォーキング -

6.2.3 イベントをトリガーにして処理を実行する - イベントドリブンモデル -

```
document.addEventListener('DOMContentLoaded', function() {
  document.getElementById('btn').addEventListener('click', function() {
    window.alert('ボタンがクリックされました。');
  }, false);
}, false);

// onloadとの違い
// →コンテンツ本体とすべての画像がロードされたところで実行
// DOMContentLoadedイベントリスナー
//  →画像のロードを待たない
// ページの初期処理は、基本的に「DOMContentLoaded」を利用する
```

6.3 属性値やテキストを取得／設定する

6.3.1 多くの属性は「要素ノードの同名のプロパティ」としてアクセスできる

```
var url = link.getAttribute("href");
link.setAttribute("href", "http://www.wings.msn.to/");
```

6.3.2 不特定の属性を取得する

6.3.3 テキストを取得／設定する

6.4 フォーム要素にアクセスする

6.4.1 入力ボックス／選択ボックスの値を取得する

6.4.2 チェックボックスの値を取得する

6.4.3 ラジオボタンの値を取得する

6.4.4 ラジオボタン／チェックボックスの値を設定する

6.4.5 複数選択できるリストボックスの値を取得する

6.4.6 アップロードされたファイルの情報を取得する

6.5 ノードを追加／置換／削除する

6.5.1 innerHTML プロパティとどのように使い分けるか

6.5.2 新規にノードを作成する

6.5.3 既存のノードを置換／削除する

6.5.4 HTMLCollection／NodeList をくり返し処理する場合の注意点

6.6 JavaScript からスタイルシートを操作する

6.6.1 インラインスタイルにアクセスする - style プロパティ -

6.6.2 外部のスタイルシートを適用する - className プロパティ -

6.6.3 スタイルクラスをより簡単に操作する - classList プロパティ -

6.7 より高度なイベント処理

6.7.1 イベントリスナー／イベントハンドラーを削除する

8.5.2 JavaScript style guide（MDN）の主な規約

8.5.3 Google 標準のコーディングスタイル