[付録] JavaScriptチートシート

著：柳井政和
Ver：1.2.0 (2021-11-29)

講義の付録の、JavaScript速習用チートシートです。他言語プログラマー、あるいはJavaScriptの仕様にそれほど詳しくない人向けの、ポイントを絞った説明です。

JavaScriptの仕様を細かく知りたい際は、MDNを見るとよいです。「MDN 検索したい内容」でGoogle検索すると、目的のページが見つかり便利です。

JavaScript | MDN
https://developer.mozilla.org/ja/docs/Web/JavaScript

■ 事前知識

● コメント

コメントは「//」の右側。
あるいは「/* ～ */」の間（こちらは複数行可能）。

```
'プログラム'    // コメント
/*
    コメント1
    コメント2
*/
```

● セミコロン

文の区切りの「;」は、付けても付けなくても構わない。どちらかに統一して記述した方がよい。

● コンソールへの出力

Webブラウザの「コンソール」は、以下の手順です。

1. Webページを右クリックして「検証」を選ぶ。開発者ツール（DevTools）が表示される。
2. 開発者ツールの「Console」タブを選ぶ。

「console.log(～)」でコンソールに情報を出力できる。「～」の部分には、「,」（カンマ）区切りで、複数の値や変数、式を書くことができる。

Chrome DevTools - Chrome Developers
https://developer.chrome.com/docs/devtools/

■ 変数と定数

● let、const とブロック スコープ

変数の宣言は「let」、定数の宣言は「const」。それぞれのスコープ（有効範囲）は「{ ～ }」（波括弧）の範囲内（ブロック スコープ）。定数は変数の一種で、再代入不可能なものを指す。

```
{ // ←これがブロック スコープの開始
    // ↑この範囲で変数や定数が有効
    let n1 = 123;    // 変数（値を差し替え可能）
    const n2 = 456;  // 定数（値を差し替え不能）

    console.log(n1, n2);  // 123, 456

    n1 = 789;  // 変数は値を差し替え可能（再代入可能）

    console.log(n1, n2);  // 789, 123

    // ↓この範囲で変数や定数が有効
} // ←これがブロック スコープの終了
```

変数には、数値でも文字列でもオブジェクトでも、何を入れてもよい。数値を入れていた変数に文字列を入れるなど、入れる内容を変更してもよい。

● var と関数スコープ

変数の宣言には「var」もある。こちらは古い宣言方法で、スコープ（有効範囲）が広い。スコープ（有効範囲）は「function() { ～ }」の範囲内（関数スコープ）。また、同じスコープ内で再宣言可能。let と const はできない。

```
function hoge() { // ←これが関数スコープの開始
    {
        var n3 = 789;  // 値を差し替え可能な変数
        console.log(n3);  // 789
    }

    // 関数スコープ単位なので
    // ブロック スコープの外でも使える
    console.log(n3);  // 789

    // 同じ階層で、同じ変数名を利用できる
    var n3 = 123;  // 再宣言可能
    console.log(n3);  // 123
```

```
} // ←これが関数スコープの終了
```

■ 数値

● 整数と小数点数

　数値は数字をそのまま書く。小数点を付ければ小数点数になる。小数点数は、丸め誤差が発生することがあるので注意が必要。

```
let n1 = 1 + 2;       // n1は 3
let n2 = 0.1 + 0.2;   // n2は 0.30000000000000004
```

● 整数化

　Math.trunc()を使うと小数点以下を切り捨てて整数にできる。

```
let n1 = Math.trunc(12.34);  // 12
```

簡便な方法として「n | 0」（有効な整数部分のビットの、OR演算を0とおこなう）の計算でも代用できる。

```
let n2 = 12.34 | 0;  // 12
```

● その他

　「MDN Number」でGoogle検索すれば、数値の仕様が分かる。

https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/Number

　「MDN Math」でGoogle検索すれば、各種の数学関数が分かる。最大値、最小値、三角関数など、さまざまな関数が用意されている。

https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/Math

■ 文字列

● 文字列の初期化

文字列は、シングルクォートやダブルクォート、バッククォートで囲む。エスケープする際は、バックスラッシュを使う。

バッククォートで囲んだ際、「${n}」のように「${ ～ }」で囲んだ領域は、変数や式など、コードを直接書くことができる。

```
let s1 = 'abc';      // abc
let s2 = "\"def\"";  // "def"
let s3 = `ABC ${s1} ${s2} DEF`;  // ABC abc "def" DEF
```

改行は「¥n」。タブ文字は「¥t」。

● 文字列の長さ

文字列の長さは「.length」で得られる。

```
let s = 'abcdefg';
let n = s.length;

console.log(s, n);  // abcdefg 7
```

● 文字列の関数

文字列の一部を得たりする各種の関数が、文字列にはある。

以下は、文字列の一部を得る関数。先頭の2文字目から、5文字目の1つ前までの文字を得ている。文字列の位置は0から数える。

```
let s1 = 'abcdefg';
let s2 = s1.substring(2, 5);

console.log(s1, s2);  // abcdefg cde
```

● その他

「MDN String」でGoogle検索すれば、文字列の仕様が分かる。

https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/String

■ 配列と for 文

● 配列の基本

配列は、Arrayオブジェクトである。多くの場合「[]」（角括弧）を利用して作る。各要素は「[数値]」という添え字を利用してアクセスする。値が入っていない要素は「undefined」（未定義）となる。配列の長さは「.length」で得る。

```
// 配列の初期化
let a1 = new Array(2);   // [empty x 2]
let a2 = ['a', 'b'];     // ['a', 'b']

// 配列の要素数
console.log(a2.length);  // 2

// ネストした配列
let a3 = ['a', ['b', 'c']];  // ['a', ['b', 'c']]
```

```
// 添え字を用いて値の変更や読み取り
let a4 = [];
a4[0] = 'abc';  // 先頭の添え字は「0」
a4[1] = 'def';
a4[4] = 'ghi';

// 各要素を出力
// 中身がない要素は「undefined」（未定義）になる
console.log(a4);      // ['abc', 'def', empty x 2, 'ghi']
console.log(a4[-1]);  // undefined
console.log(a4[0]);   // abc
console.log(a4[1]);   // def
console.log(a4[2]);   // undefined
console.log(a4[3]);   // undefined
console.log(a4[4]);   // ghi
```

● for 文

以下のように書く。

```
// 配列の要素数だけ処理をおこない、各要素を出力
let arr = ['a', 'b', 'c'];
for (let i = 0; i < arr.length; i ++) {
    console.log(i, arr[i]);
```

```
}

// 0 'a'
// 1 'b'
// 2 'c'
```

途中で処理を飛ばして制御部分に戻るには「continue」を使う。
途中で処理を中断して for 文から抜けるには「break」を使う。

● 配列の関数

配列には、他のプログラミング言語にも見られる各種メソッドがある。

```
let a = ['a', 'b', 'c', 'd', 'e'];
a.push('f');     // 末尾に追加
console.log(a);  // ['a', 'b', 'c', 'd', 'e', 'f']
```

また、反復メソッド（map, filter, forEach, reduce など）もある。

```
let arr = ['a', 'b', 'c'];
arr.forEach((x, i) => {
    console.log(i, arr[i]);
});

// 0 'a'
// 1 'b'
// 2 'c'
```

● その他

「MDN Array」でGoogle検索すれば、配列の仕様が分かる。

https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/Array

■ オブジェクト

● オブジェクトの基本

　オブジェクトは、Objectオブジェクトである。多くの場合「{}」（波括弧）を利用して作る。各要素（プロパティ）は「オブジェクト[プロパティ名]」あるいは「オブジェクト.プロパティ名」の形でアクセスする。値が入っていないプロパティは、「undefined」（未定義）となる。

```
// オブジェクトの作成
let user = {name: 'hawk', hp: 100, mp: 50, level: 1, exp: 10};

// プロパティの読み取り
console.log(user.name);   // hawk
console.log(user['hp']);  // 100

// 空のプロパティは「undefined」（未定義）になる
console.log(user.skill);  // undefined

// プロパティの変更
user.skill = 'fire';
console.log(user.skill);  // fire
```

● その他

　「MDN Object」でGoogle検索すれば、オブジェクトの仕様が分かる。

https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/Object

■ 分割代入

● 配列の分割代入

　変数への代入時に「[ ]」（角括弧）を使うことで、要素を分割して代入できる。代入の際は、要素の先頭から順に格納する。

```
// 配列の作成
const arr = ['dog', 'cat', 'monkey', 'bird'];

// 配列の要素を先頭から分割代入
const [a, b, c] = arr;

console.log(a);  // dog
console.log(b);  // cat
console.log(c);  // monkey
```

● オブジェクトの分割代入

　変数への代入時に「{ }」（波括弧）を使うことで、プロパティを分割して代入できる。代入の際は、変数名と同じプロパティの値を格納する。

```
// オブジェクトの作成
const chara = {name: 'Bob', hp: 100, mp: 8, skill: 'sword'};

// オブジェクトのプロパティから、プロパティ名の値を分割代入
const {name, hp, mp} = chara;

console.log(name);  // Bob
console.log(hp);    // 100
console.log(mp);    // 8
```

● その他

　「MDN 分割代入」でGoogle検索すれば、分割代入の仕様が分かる。分割代入は、さらに複雑な代入がおこなえる。

https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

■ 関数

● function

JavaScriptの関数の書き方は多くの種類がある。基本は「function」という予約語を使う。「function 関数名 (引数) {処理}」と書く以外に、「function (引数) {処理}」と無名関数を書く方法がある。「return 値」で戻り値（返り値）を戻せる。JavaScriptの関数はオブジェクトである。

```
// 関数の作成
function func1 (arg1, arg2) {
    // 処理
    return `result: ${arg1}, ${arg2}`;
}

// 関数の呼び出し
console.log(func1(111, 222));  // result: 111, 222

let func2 = function (arg1, arg2) {
    // 処理
    return `result: ${arg1}, ${arg2}`;
};

// 関数の呼び出し
console.log(func2(333, 444));  // result: 333, 444
```

● アロー関数

関数には、より簡便なアロー関数もある。「(引数) => {処理}」と書くことで、関数を短く書ける。

```
// アロー関数 基本
let arrow1 = (arg1, arg2) => {
    return `result: ${arg1}, ${arg2}`;
};

// アロー関数 簡略
// 処理がreturn文1行なら、波括弧とreturnを省ける
let arrow2 = (arg1, arg2) => `result: ${arg1}, ${arg2}`;

// アロー関数 簡略
// 引数が1つなら丸括弧を省ける
let arrow3 = arg1 => {
    return `result: ${arg1}`;
};
```

```
// アロー関数 簡略
// 引数が0なら丸括弧のみを書く
let arrow4 = () => {
    return `result`;
};

// アロー関数 簡略
// このように短く書く処理もある
let arrow5 = arg1 => `result: ${arg1}`;
```

アロー関数は、配列の反復メソッド（map, forEach, filter, reduce など）と一緒によく使う。

```
let arr = [123, 456, 489];
console.log(arr.map((x, i) => `${i}: ${x * 10}`));  // ['0: 1230', '1: 4560', '2: 4890']
```

配列の「map」は加工した新しい配列を作る。「forEach」は全ての要素に対して処理をおこなう。「filter」は条件を指定して要素を絞り込む。「reduce」は、要素をまとめて合計値を出したりする際に使う。

● 関数のthis

関数の this は、その関数の親に当たるオブジェクトを指す（ことが多い）。オブジェクトのプロパティになっている関数のことを、特別にメソッドと呼ぶ。

```
// 親に当たるオブジェクトが this になる。
let user1 = {
    name: 'hawk',
    getName1: function() {
        return this.name
    },
    getName2() {
        return this.name
    }
};

console.log(user1.getName1());  // hawk
console.log(user1.getName2());  // hawk
```

```
// メソッドを付け替えると、その後の親に当たるオブジェクトがthisになる。
// 「user1」オブジェクト
let user1 = {
    name: 'hawk',
```

```
    getName: function() {
        return this.name;
    }
};

// 「user2」オブジェクト
let user2 = {
    name: 'snake'
};

// メソッドの付け替え
// メソッドを付け替えると、その後の親に当たるオブジェクトが this になる
user2.getName = user1.getName;

console.log(user1.getName());  // hawk
console.log(user2.getName());  // snake
```

```
// 関数から new 演算子でオブジェクトを作る
function Enemy() {
    this.name = 'dark king';
    this.getName = function() {
        return this.name;
    }
}

// 関数「Enemy」を雛形にして、インスタンス（実体のオブジェクト）を作る
let enemy = new Enemy();

console.log(enemy.getName());  // dark king
```

アロー関数は、 function で作った関数と全く同じではない。関数スコープを作らない。「this」が指す対象は、関数スコープごとに変わる。そのため、アロー関数内では「this」が指す対象が変化しない。また、アロー関数は new 演算子を使ってインスタンス（実体）を作れない。

● その他

「MDN 関数」でGoogle検索すれば、関数の仕様が分かる。

https://developer.mozilla.org/ja/docs/Web/JavaScript/Guide/Functions

「MDN this」でGoogle検索すれば、 this の仕様が分かる。

https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Operators/this

■ 様々な値と if 文

● undefined と null

「undefined」は「未定義」を表す。変数を宣言して未定義な時、中身は「undefined」になる。

関数で、「return」で値を戻さなかった場合の戻り値も「undefined」になる。関数の引数に値を入れなかった時も、その引数は「undefined」になる。値が入っていない配列の要素や、オブジェクトのプロパティを参照した場合も「undefined」になる。

明示的に値を指定していないものは、基本的に「undefined」になる。

「null」はnull値である。明示的に「何も入っていない」ことを表す値である。

「undefined == null」は「true」である。区別を付けたい際は、より厳密な比較演算子「===」を使う。「undefined === null」は「false」である。

● '' と 0 とfalse

「''」は空文字、「0」は数値の0、「false」は真偽値の「偽」である。

「'' == 0」「'' == false」「0 == false」は「true」である。区別を付けたい際は、より厳密な比較演算子「===」を使う。「'' === 0」「'' === false」「0 === false」は「false」である。

● if 文

以下のように、条件式と処理を書く。最初に「真」の時の処理、else以降は「偽」の時の処理。

```
if (条件式) {真の時の処理}
if (条件式) {真の時の処理} else {偽の時の処理}
if (条件式) {真の時の処理} else if (条件式2) {条件式2が真の時の処理}
```

```
// if 文用の関数
const func = function(arg) {
    if (arg <= 1) {
        console.log(arg, 'は1以下');
    } else if (arg <= 5) {
        console.log(arg, 'は1より大きく、5以下');
    } else {
        console.log(arg, 'は5より大きい');
    }
}

// 0から9まで処理を実行
for (let i = 0; i < 10; i ++) {
    func(i);
```

```
}

// 0 は1以下
// 1 は1以下
// 2 は1より大きく、5以下
// 3 は1より大きく、5以下
// 4 は1より大きく、5以下
// 5 は1より大きく、5以下
// 6 は5より大きい
// 7 は5より大きい
// 8 は5より大きい
// 9 は5より大きい
```

条件式は、「false」「undefined」「null」「''」「0」は「偽」と見なす。

```
if (false)     {console.log('真')} else {console.log('偽')}  // 偽
if (undefined) {console.log('真')} else {console.log('偽')}  // 偽
if (null)      {console.log('真')} else {console.log('偽')}  // 偽
if ('')        {console.log('真')} else {console.log('偽')}  // 偽
if (0)         {console.log('真')} else {console.log('偽')}  // 偽
```

■ エラー処理

● try catch 文

「try { }」の中でエラーをキャッチして、「catch(e) { }」の中で、エラー発生時の処理を書ける。「e」にはエラーの情報が入る。

```
try {
    const str = '{name: "Ken", hp: 100}';
    const obj = JSON.parse(str);
    console.log('[Success]', obj);
} catch(e) {
    console.log('[Error]', e);
}

// [Success] { name: 'Ken', hp: 100 }
// [Error] SyntaxError: Unexpected token n in JSON at position 1
//      （詳細なエラー情報）
```

● throw 文

「throw」を使うことで、エラー情報を引き渡してエラーを発生させられる。文字列を渡す以外に、エラーオブジェクトを渡すこともできる。エラー オブジェクトを渡せば、エラーの発生行などの情報を伝達可能。

```
try {
    throw 'Oh!';
    console.log('[Success]')
} catch(e) {
    console.log('[Error]', e)
}

try {
    throw new Error('happning!');
    console.log('[Success]');
} catch(e) {
    console.log('[Error]', e);
}

// <2-2> [Error] Oh!
// <2-4> [Error] Error: happning!
//      （詳細なエラー情報）
```

● エラーのネスト

　エラーはキャッチされるまで、関数のネストをさかのぼっていく。最後までキャッチされないと、処理が停止する。

```
const func1 = function() {
    console.log('func1: start');

    const func2 = function() {
        console.log('func2: start');

        throw new Error('happning!');

        console.log('func2: end');
    };
    func2();

    console.log('func1: end');
};

try {
    console.log('try')
    func1();
    console.log('[Success]')
} catch(e) {
    console.log('[Error]', e)
}

// try
// func1: start
// func2: start
// [Error] Error: happning!
//     （詳細なエラー情報）
```

● その他

　「MDN try catch」でGoogle検索すれば、try catch 文の仕様が分かる。

https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Statements/try...catch

　「MDN throw」でGoogle検索すれば、throw 文の仕様が分かる。

https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Statements/throw

■ クラス

● クラスの宣言とインスタンスの利用

「class」を宣言することで、new演算子でインスタンス（実体）を作るクラスを作成できる。インスタンス作成時には、「constructer」メソッドが実行される。

```
// クラスの作成
class Chara {
    constructor(name) {
        this.name = name;
        this.hp = 100;
        this.mp = 50;
    }
    getStatus() {
        return `${this.name} : hp ${this.hp}, mp ${this.mp}`;
    }
}

// クラスから new 演算子でオブジェクトを作成
let enemy = new Chara('slime');

console.log(enemy.name);         // slime
console.log(enemy.getStatus());  // slime : hp 100, mp 50
```

● 静的プロパティと静的メソッド

クラス固有の静的プロパティや静的メソッドを作りたい時は「static」を使う。

```
// クラスの作成
// 静的プロパティや静的メソッドを作る
class Chara {
    static nameDefalt = 'unknown';
    static hpDefalt = 10;
    static mpDefalt = 5;
    static getStatusDefault() {
        return `${this.nameDefalt} : hp ${this.hpDefalt}, mp ${this.mpDefalt}`;
    }
}

console.log(Chara.nameDefalt);          // unknown
console.log(Chara.getStatusDefault());  // unknown : hp 10, mp 5
```

● その他

「MDN クラス」でGoogle検索すれば、クラスの仕様が分かる。

https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Classes

■ DOM操作

● DOMの準備が終わったあとに処理を実行

DOMの要素は、HTMLが読み込まれてDOMの解析が終わるまで操作することができない。そのため、DOMの準備が整うまで待ったあと、処理を実行する必要がある。
「window.addEventListener('DOMContentLoaded', <関数>)」を使うことで、処理を待ったあと関数を実行できる。

```
// DOMの準備が終わったあとに処理を実行
window.addEventListener('DOMContentLoaded', event => {
    console.log('DOMの準備が完了');
});
```

● DOM要素の取得

「document.querySelector(CSSセレクタの文字列)」で、要素を1つ選択可能。該当なしの場合は「null」が返る。

```
// 要素を選択
let element = document.querySelector('#id');
```

「document.querySelectorAll(CSSセレクタの文字列)」で要素を複数選択可能。「NodeList」（配列に似た物）が返る。該当なしの場合は、空の「NodeList」が返る。「Array.from()」と組み合わせて、配列として受け取るとよい。

```
// 要素を全て選択
let nodeList = document.querySelectorAll('.news');
let elementArray = Array.from(nodeList));
```

● 属性の取得と書き換え

「.属性」で操作できる。

```
// a 要素の target の値を取得したあと書き換え
let elementArray = Array.from(document.querySelectorAll('a'));
elementArray.map(element => {
    const target = element.target;
    console.log(target);
```

```
    element.target = '_blank';
});
```

● 内部のHTMLの書き換え

「.innerHTML」を使うと、内部のHTMLを書き換えられる。

```
// a 要素内の HTML を書き換え
let elementArray = Array.from(document.querySelectorAll('a'));
elementArray.map(element => {
    element.innerHTML = 'hogehoge';
});
```

● スタイルの書き換え

「.style.スタイルの名前」で書き換えられる。

```
// a 要素の文字色を赤に変える
let elementArray = Array.from(document.querySelectorAll('a'));
elementArray.map(element => {
    element.style.color = 'red';
});
```

● クラスの書き換え

「.classList.＜各種メソッド＞」で、クラスを追加したり削除したり、切り換えたりできる。

```
// a 要素のクラスを書き換え
let elementArray = Array.from(document.querySelectorAll('a'));
elementArray.forEach(element => {
    element.classList.add('big');
});
```

● フォームの値の操作

「.value」を使い、フォームの値を操作できる。

```
// フォームの部品の値を JavaScript! に書き換える
let elementArray = Array.from(document.querySelectorAll('input'));
elementArray.map(element => {
```

```
    element.value = 'JavaScript!';
});
```

● イベント処理

「.addEventListener()」を使い、イベントの種類と、イベント発生時に呼び出される関数を書くと、その要素でイベントが発生したときの処理を書ける。

```
// フォームの部品をクリックすると、その部品の値とイベントオブジェクトを出力する
let elementArray = Array.from(document.querySelectorAll('input'));
elementArray.map(element => {
    element.addEventListener('click', function(e) {
        console.log(this.value, e);
    });
});
```

イベント発生時の関数には、イベントの詳細が入ったオブジェクトが引数として渡される。イベント発生時の関数のthisは、そのイベントが起きたオブジェクトになる。

■ 非同期処理と Promise

● 非同期処理

JavaScript は基本的に、シングル スレッドで処理をおこなう。そのため時間の掛かる処理を待たずに処理を進める「非同期処理」が用いることが多い。そうすることで、 Web ブラウザの UI を止めることなく、処理をおこなえる。

JavaScript では関数を実行する場合に、「処理終了後に実行する関数」を引数にして渡すことが多い。「関数(引数1, 引数2, ..., 終了後に実行する関数)」のように書く。こうした引数の関数のことをコールバック関数と呼ぶ。待ち時間がある処理では、コールバック関数は時間差で実行される。

時間が掛かる処理が終わってから次の時間が掛かる処理をおこなう。そうしたことを複数回おこなうと、JavaScript では関数のネストが深くなる。

```
時間が掛かる処理(引数1, 引数2, ..., 終了後に実行する関数() {
    時間が掛かる処理(引数1, 引数2, ..., 終了後に実行する関数() {
        時間が掛かる処理(引数1, 引数2, ..., 終了後に実行する関数() {
            時間が掛かる処理(引数1, 引数2, ..., 終了後に実行する関数() {
            })
        })
    })
})
```

```
// ネストが深い非同期処理
setTimeout(() => {
    console.log('1つめの処理');
    setTimeout(() => {
        console.log('2つめの処理');
        setTimeout(() => {
            console.log('3つめの処理');
        }, 1000);
    }, 1000);
}, 1000);

// 1つめの処理
// 2つめの処理
// 3つめの処理
```

こうした問題を解決するために、 Promise という文法が JavaScript に追加された。

● Promise resolve then

Promise オブジェクトは、関数を引数に取る。引数の関数内で、「resolve()」（解決）を実行すると、Promise オブジェクトの後続の「then()」内の関数を実行する。

「then()」内の関数で Promise オブジェクトを返せば、同じように「resole()」実行時に、後続の「then()」や「catch()」内の関数を実行する。

この仕組みは、文章で書くと分かり難いので、コードの形で書く。この段階では、まだあまり便利なようには見えない。

```
// Promise を利用した非同期処理
new Promise(function(resolve, reject) {
    setTimeout(() => {
        console.log('1つめの処理');
        resolve();
    }, 1000);
})
.then(function() {
    return new Promise(function(resolve, reject) {
        setTimeout(() => {
            console.log('2つめの処理');
            resolve();
        }, 1000);
    });
})
.then(function() {
    return new Promise(function(resolve, reject) {
        setTimeout(() => {
            console.log('3つめの処理');
            resolve();
        }, 1000);
    });
});

// 1つめの処理
// 2つめの処理
// 3つめの処理
```

「resoleve()」を実行すると、後続の「then()」の第1引数の関数が実行される。

● Promise resolve then 2

「resoleve()」や「reject()」に引数を付けると、その値を次の処理に送ることができる。

また、多くの場合、 Promise オブジェクトを返す関数を用意しておき、処理を簡素に書く。

```
let wait = function(time) {
    return new Promise(function(resolve, reject) {
```

```
        setTimeout(() => {
            console.log(`${time}ミリ秒`);
            resolve(`end: ${time}`);
        }, time);
    });
}

wait(2000)
.then(res => {
    console.log(res);
    return wait(1500);
})
.then(res => {
    console.log(res);
    return wait(1000);
})
.then(res => {
    console.log(res);
    return wait(500);
})
.then(res => {
    console.log(res);
});

// 2000ミリ秒
// end: 2000
// 1500ミリ秒
// end: 1500
// 1000ミリ秒
// end: 1000
// 500ミリ秒
// end: 500
```

● async と await

Promiseの処理は、関数ブロックの先頭に「async」を書き、 Promise オブジェクトを返す関数の前に「await」を書くことで、シンプルに書くことができる。

```
let wait = function(time) {
    return new Promise(function(resolve, reject) {
        setTimeout(() => {
            console.log(`${time}ミリ秒`);
            resolve(`end: ${time}`);
        }, time);
    });
```

```
};

(async function() {
    await wait(2000);
    await wait(1500);
    await wait(1000);
    await wait(500);
})();

// 2000ミリ秒
// 1500ミリ秒
// 1000ミリ秒
// 500ミリ秒
```

● Promise reject catch

Promise オブジェクトの引数の関数内で「reject()」（拒否）を実行すると、 Promise オブジェクトの後続の「then()」の第2引数の関数が実行される。あるいは、後続の最初の「catch()」まで処理が移動する。

```
// then の2つめの関数を設置
new Promise(function(resolve, reject) {
    setTimeout(() => {
        console.log('1つめの処理');
        reject();
    }, 1000);
})
.then(function() {
    return new Promise(function(resolve, reject) {
        console.log('resolveである');
        setTimeout(() => {
            resolve();
        }, 1000);
    });
}, function() {
    return new Promise(function(resolve, reject) {
        console.log('rejectである');
        setTimeout(() => {
            resolve();
        }, 1000);
    });
});

// 1つめの処理
// rejectである
```

```
// catch の使用。最初の reject で一気に catch まで飛ぶ。
new Promise(function(resolve, reject) {
    console.log('1つめの処理');
    setTimeout(() => {
        reject();
    }, 1000);
})
.then(function() {
    console.log('2つめの処理');
    return new Promise(function(resolve, reject) {
        setTimeout(() => {
            resolve();
        }, 1000);
    });
})
.then(function() {
    console.log('3つめの処理');
    return new Promise(function(resolve, reject) {
        setTimeout(() => {
            resolve();
        }, 1000);
    });
})
.catch(function() {
    console.log('catchした');
});

// 1つめの処理
// catchした
```

● Promise の静的メソッド

Promiseには、いくつかの静的メソッドがある。、全て待ってから進む「Promise.all()」は、よく使う。

```
let wait = function(time) {
    return new Promise(function(resolve, reject) {
        setTimeout(() => {
            console.log(`${time}ミリ秒`);
            resolve(`end: ${time}`);
        }, time);
    });
}

// Promise オブジェクトの配列を作る
let arr = [];
```

```
arr.push(wait(2000));
arr.push(wait(1500));
arr.push(wait(1000));
arr.push(wait(500));

// 全て解決するまで待つ
Promise.all(arr)
.then(res => {
    console.log('終了', res);
});

// 500ミリ秒
// 1000ミリ秒
// 1500ミリ秒
// 2000ミリ秒
// 終了 (4) ['end: 2000', 'end: 1500', 'end: 1000', 'end: 500']
```

● その他

「MDN Promise」でGoogle検索すれば、Promiseの仕様が分かる。

https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Global_Objects/Promise