

JavaScript 速習

これだけは覚えておこう

まずは、プログラムを書き始める前に、以下のことを頭に入れていきましょう。

変数の使い方

変数を宣言するときは、**var** を使います。変数名は、半角英数字を利用できます。_(アンダーバー)は利用できますが、ハイフンは利用できません。小文字と大文字を区別します (**eXample** と **example** は違う変数です) ので、注意しましょう。

```
var sample = "example";
```

sample という変数を宣言して、その中に文字列の **example** を入れます。

```
var sample;
```

空の変数を宣言できます。

```
var sample, sample1, sample2;
```

いくつかの変数を連続して宣言することもできます。

ポイント

プログラムには、“型”という概念があります。それは、その代入された値が、文字列か数字かなど判別します。**JavaScript** では、以下のものが代表的な型として決められています。

文字列(**string**)、数字(**number**, **int**, **float**)、真偽(**boolean**, **true** | **false**)、オブジェクト(**object**)、配列(**array**)

この 5 つを覚えておけば、基本的には大丈夫です。使い方は以下の通りです。
※真偽、オブジェクト、配列に関してはあとで説明します。

文字列 “sample”

”か’で囲んだものが文字列になります。

私たちは、メッセージや表示する値などの場合は、”を、プログラムで利用する値は、’で囲むようにしています。

数字 1000, 1.02

数字は、そのままかきます。

整数もかけますし、小数点を含んだ値も書けます。マイナスにする場合は、
-100
というような形で書きます。

プログラムをかく準備

JavaScript 速習

フロントエンド **JavaScript** は、ブラウザを使ってかいていきます。

文字コードで **UTF-8** を編集できるテキストエディタであれば、どのエディタでも利用することができます。

HTML、**JavaScript** のハイライティングをしてくれるエディタを使うと、間違いが少なくなります。

デバッグの方法

フロントエンド **JavaScript** のデバッグは、ブラウザの開発ツールを利用していきます。

開発メニューから **JavaScript** デバッガを見ていきます。

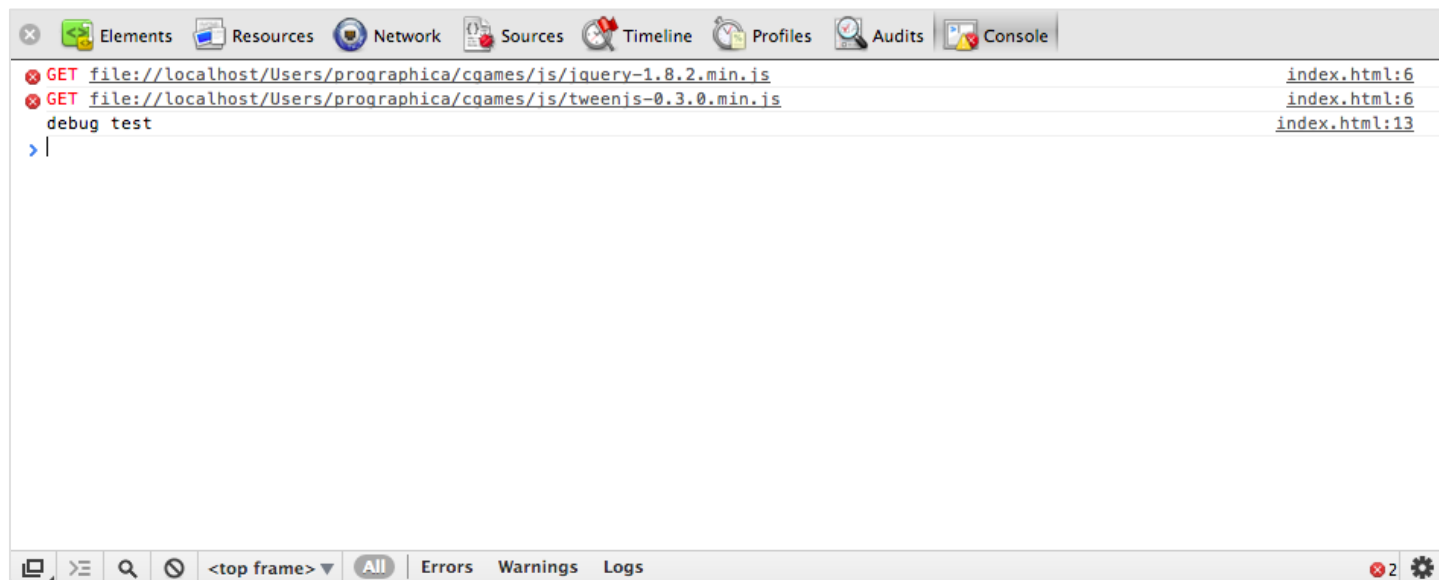
開発画面は下記のように使っていきます。

1) コンソール画面 (基本)

`console.info`, `console.debug` で、中身の値をコンソール画面に出力していくことが可能です。

自分が書いたプログラムで、エラーは発生しないが、希望した結果が返ってこない場合などは、1つ1つの値を `console.debug` で出力しながら、オブジェクトの中身の値や、配列の値などが正しいか見ていくことが肝心です。迷ったら、`console.debug` で値を出力しましょう。

また、**JavaScript** のエラーは、このコンソール画面に表示されます。



例)

```
console.info(a);
```

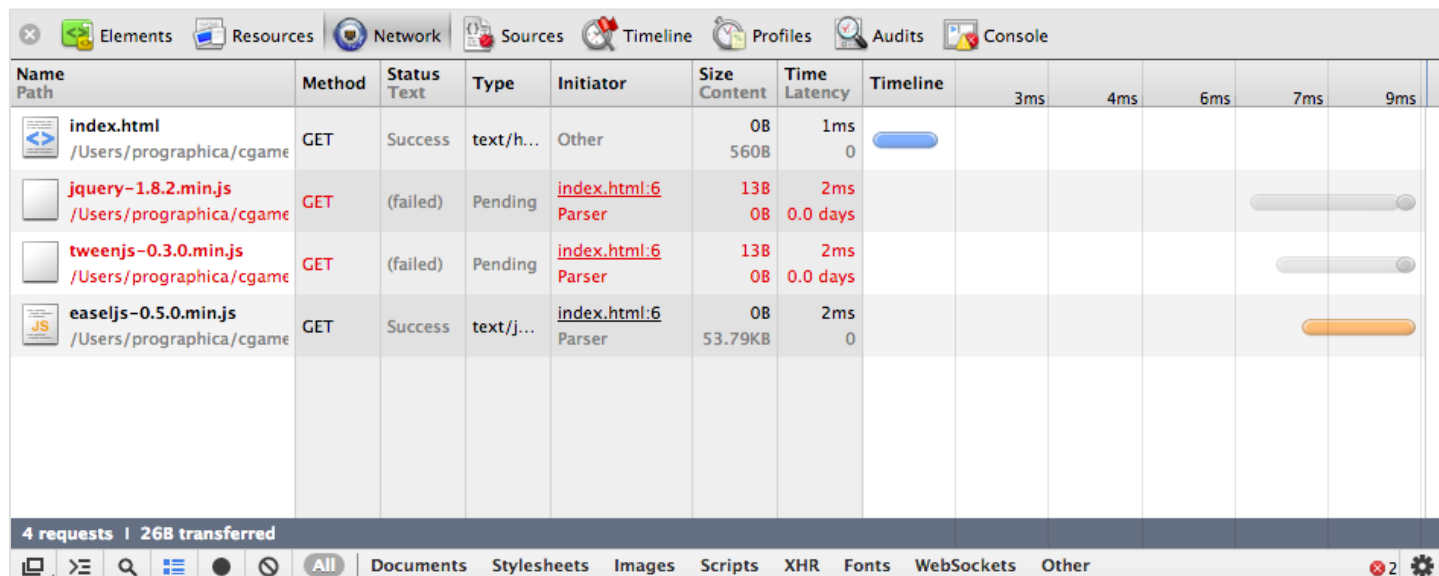
```
console.debug(a);
```

2) ネットワーク (基本)

ネットワークへのアクセス状況や、ネットワークから取得したファイルの一覧・中身を確認することができます。

JSON や **XML** でやりとりしているプログラムなど、**JSON** の中身を確認したい場合に非常に有効です。

JavaScript 速習



Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency	Timeline	3ms	4ms	6ms	7ms	9ms
index.html /Users/prographica/cgame	GET	Success	text/h...	Other	0B 560B	1ms 0						
jquery-1.8.2.min.js /Users/prographica/cgame	GET	(failed)	Pending	index.html:6 Parser	13B 0B	2ms 0.0 days						
tweenjs-0.3.0.min.js /Users/prographica/cgame	GET	(failed)	Pending	index.html:6 Parser	13B 0B	2ms 0.0 days						
easeljs-0.5.0.min.js /Users/prographica/cgame	GET	Success	text/j...	index.html:6 Parser	0B 53.79KB	2ms 0						

4 requests | 268 transferred

All Documents Stylesheets Images Scripts XHR Fonts WebSockets Other

JSON の中身を確認する

中身を確認する場合は、ネットワークタブで、**JSON** を取得した **URL** をクリックします。サブタブから、プレビューをクリックすると、**JSON** が整形された状態で確認することが可能です。

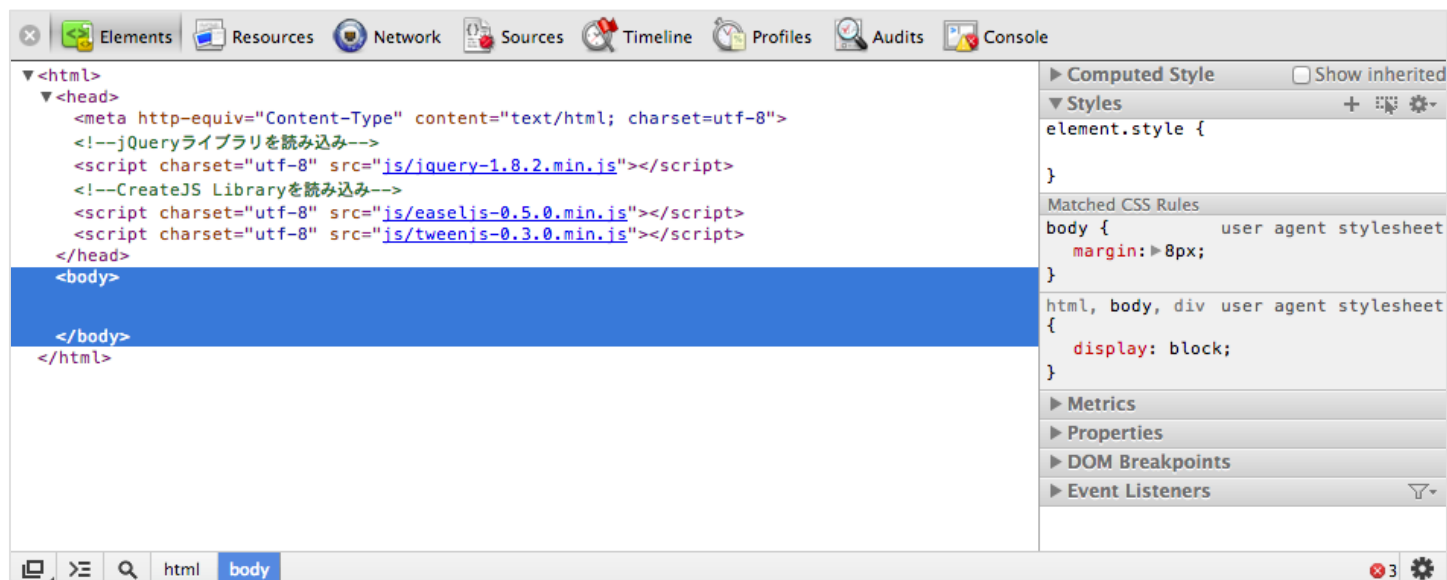
ポイント

URL が赤くなっている場合は、取得に失敗しています。

その場合は、取得している **URL** をよく確認し、送信するパラメータが間違っていないかも同時に確認してください。右クリックで、新しくその **URL** を開くことで原因がわかる場合もあります。

4) エレメント (基本)

そのときの **HTML** の状態を知ることができます。



<pre><html> <head> <meta http-equiv="Content-Type" content="text/html; charset=utf-8"> <!--jQueryライブラリを読み込み--> <script charset="utf-8" src="js/jquery-1.8.2.min.js"></script> <!--CreateJS Libraryを読み込み--> <script charset="utf-8" src="js/easeljs-0.5.0.min.js"></script> <script charset="utf-8" src="js/tweenjs-0.3.0.min.js"></script> </head> <body> </body> </html></pre>	<p>Computed Style <input type="checkbox"/> Show inherited</p> <p>Styles</p> <pre>element.style { }</pre> <p>Matched CSS Rules</p> <pre>body { margin: 8px; } html, body, div { display: block; }</pre> <p>Metrics</p> <p>Properties</p> <p>DOM Breakpoints</p> <p>Event Listeners</p>
---	---

JavaScript で、**HTML** の一部を書き換えたり、スタイルを変えたりしていきませんが、その結果を随時確認しながら、プログラムを進めていくことができます。

3) タイミング (アドバンスト)

JavaScript で実行しているプログラムなどの状態を知ることができます。

かいたアプリのパフォーマンスを上げたい場合など、タイミングを使っていきます。

Visual Studio のコンソールツールも基本的に同じような機能が提供されています。

ただし、**console.info** は、文字列のみを出力するので注意が必要です。オブジェクトや配列の中身を確認したい場合は、**console.dir** を利用します。

プログラムを制御

では、早速プログラムを書いていきましょう。

```
<script type="text/javascript">
```

```
var a = 1;
```

```
var b = 2;
```

```
alert(a + b); => 答えは何？
```

```
//if で比較することができます
```

```
if(a > b){
```

```
    alert("aの方が大きい");
```

```
}else{
```

```
    alert("bの方が大きい");
```

```
}
```

```
//分岐を下記のように書くことができます
```

```
switch(a){
```

```
    case 1:
```

```
        alert("a は 1 である");
```

```
        break;
```

```
    case 2:
```

```
        alert("a は 2 である");
```

```
        break;
```

```
    default:
```

```
        alert("a はそれ以外である");
```

```
        break;
```

```
}
```

```
//ループは以下のようにできます
```

```
//配列を順に呼び出して、アラートで配列の中身を表示します。
```

```
var arr = ["apple", "orange", "pine"];
```

```
for(var i = 0; i < arr.length; i++){
```

```
    alert(arr[i]);
```

```
}
```

```
var obj = {a: 1, b: 2, c: 3};
```

```
for(var key in obj){
```

```
    alert();
```

```
}
```

```
//文字列と数字を足す場合
```

```
var a = "1";
```

```
var b = 2;
```

```
alert(a + b); => 答えは何？
```

</script>

先ほどのデバッグ方法を試しながら、正しく動作するように、プログラムを書いてみてください。

オブジェクトと配列

JavaScript の中で、より効率的なプログラムを書いて行くには、オブジェクト・配列の使い方を知る必要があります。

オブジェクトは、名前と値を対に持つことができる箱です（連想配列とも呼ばれます）。

オブジェクトは特に、後に出てくる関数の引数に初期コンフィグとしてそのまま渡されて、画面を生成するような形で利用します。

配列は、値を順序よく並べて、保管しておく箱です。

また、JSON 形式を利用して、通信を行った場合、受け取った JSON データ（ただのテキスト）を変換した場合には、このオブジェクトと配列に変換されます。

オブジェクトは、`{}`（中カッコ）で囲ってかいていきます。

配列は、`[]`（カギカッコ）で囲ってかいていきます。

では、早速オブジェクトと配列を使ってプログラムを書いてみましょう。

```
<script type="text/javascript">
```

```
//空のオブジェクトを宣言する
```

```
var obj = {};
```

```
//オブジェクトに値を追加していきます
```

```
//.firstname の部分をプロパティまたはキーと呼びます。
```

```
//実際の値は、バリューと呼ばれます。
```

```
obj.firstname = “山田”;
```

```
obj.lastname = “太郎”;
```

```
obj.age = 23;
```

```
//配列をオブジェクトに入れることもできます。もちろんさらにオブジェクトを入れることができます。
```

```
obj.tag = [“タグ 1”, “タグ 2”];
```

```
//山田 太郎と出力されます。各値には、.（ドット）でアクセスすることができます。
```

```
console.debug(obj.lastname + ” ” + obj.firstname);
```

```
//まとめて宣言する場合は、 下記のように宣言できます。
```

```
//複数のプロパティを設定する場合は、,（カンマ）で区切ってきます。
```

```
var obj = {
```

```
    lastname: “山田”,
```

```
    firstname: “太郎”,
```

```
age: 23,
```

```
tag: [“タグ 1”, “タグ 2”]
```

```
};
```

</script>

ポイント

最後にカンマを加えてしまう場合がありますが、モダンブラウザではエラーになりません。
ただし、IE の古いバージョン（IE6, 7, 8）ではエラーとなりますので、注意が必要です。

関数を使う

プログラムは、通常上から実行されてしまいますが、実行したいときに呼び出せるようにするために、関数を使っていきます。モダン JavaScript では、通信が成功したときや、なにかボタンを押したときに、実行するプログラムを呼び出すために、関数が使われます。

関数本来の意味は、少し違います。

関数には、引数（その関数を実行するために外から持ってきた値）と戻り値（その関数が実行された結果を返す）という仕組みがあります。

工場を考えてみるとわかりやすいのですが、引数が、工場に持ってくる素材、関数自体が生産ライン、戻り値ができた製品となります。もう少し具体的なイメージとしては、

リンゴを素材として、生産ラインにのせると、生産ラインで切って絞って、製品としてリンゴジュースができるイメージ。ミカンをあげると、ミカンジュースができます。

その他必要な素材（水など）は、あらかじめ工場内に持っているので、外から与える必要はありません。

概念としては、基本的に、共通項目をまとめ上げて、プログラムをきれいに書く方法だと考えてください。

それでは、簡単に JavaScript で利用するシーンをあげてみます。

```
<script type="text/javascript">
```

```
//mat が引数です。()の中に書きます。
```

```
//中括弧の中に、実行するプログラムを書いていきます。
```

```
//return でそのプログラムを終了して、戻り値を返し、次のプログラムに移っていきます。
```

```
var factory = function(mat){  
    var juice = mat + “ジュース”;  
    return juice;  
};
```

```
console.debug(factory(“りんご”));
```

```
console.debug(factory(“みかん”));
```

```
</script>
```

モダン JavaScript での利用例（jQuery での使われ方）

```
<script type="text/javascript">
```

```
//url で指定された
```

```
$.ajax({  
    url: “sample.json”,  
    success: function(res, success, xhr){  
        console.debug(res);  
    return;  
    }  
});
```

JavaScript 速習

```
});  
</script>
```

さらに使いこなすために

イベントとリスナー

直訳すると聞く人ですが、モダン **JavaScript** におけるリスナーは、なにかアクション（たとえば、クリックやダブルクリックしたとき、他にも **AJAX** 通信が完了したときなど、に実行される関数をリスナーと言います。

たとえば先ほどの

モダン **JavaScript** での利用例（jQuery での使われ方）

```
<script type="text/javascript">
//url で指定された
$.ajax({
    url: "sample.json",
    success: function(res, success, xhr){
        console.debug(res);
    }
    return;
});
</script>
```

も、一種のリスナーと呼ぶことができます。

通常、たとえば jQuery の場合は（bind と言いますが）、下記のように書くことが可能です。

HTML

```
<div id="sample">クリックして</div>
```

JavaScript

```
$("#sample").bind('click', function(e){
    console.debug(e);
    return;
});
```

というように、クリックされたときに実行するプログラムを追加する行為を、リスナーを追加すると言います。ですので、リスナーは、待っている人、そのイベント（クリックやダブルクリック）がアクションされるのを待っている人という意味となります。

モダン **JavaScript** では、常に、通信や、アクションの発生が非同期で行われるので、このイベントを発火（アクションさせる）概念と、リスナーの概念が非常に重要になります。

仕事で使うために

スコープ

JavaScript には、グローバル領域と、ローカル領域という考え方があります。

先ほどの関数の中（たとえるなら工場の中）が、ローカル領域で、それ以外が<script>の中に直接書き込んでいるところなどが、グローバル領域といえます。

少し具体的な説明をすると、

関数内で宣言（**var**をつかったもの）された変数は、その関数内でしか利用できません。外に出すには、戻り値としてその変数を出す以外方法はありません。

グローバル変数（グローバル領域に宣言された変数のことを言います）は、グローバル領域または、ローカル領域の両方どちらからも、アクセスできます。

まとめると、

グローバル変数は、どこからでもアクセスできるもの

ローカル変数は、その宣言された関数内でのみようできる使い捨てのもの
ということです。

ポイント

後に説明しますが、グローバル変数は、よほどでない限り使用しないようにするのが、うまくプログラミングをするコツです。

ローカル変数をきっちりと利用することにより、変数名を簡単なものにできます（重なっても問題ないため）。また、ローカル変数はその関数の実行が終了すると破棄されますので、メモリの無駄遣いを減らすことができます。

無名関数（高度な話題）

前述した、スコープを最大限利用しながら、プログラム全体をクラス化、カプセル化（機能を分割して、パッキングして使いやすくする）するときに、無名関数という考え方が出てきます。

ちょうど、下のような書き方です。

```
var app = app || {};
```

```
(function(){
```

```
    var state = null;
```

```
    //開始時に実行
```

```
    app.start = function(){
        state = "start";
        console.debug(state);
    }
```

```
    //終了時に実行
```

```
    app.end = function(){
```

```
        if(state == "start"){
            console.debug("スタートされていきました");
        }else{
            console.debug("スタートされていませんでした");
        }

        return;
    }
})();
```

というような書き方をしていきます。

グローバル領域に、**app** を宣言し、そのプロパティとして、**start**, **end** を実行できるようになっています。また、無名関数の中に、**state** が 1 つ宣言されていますが、**app.start** と **app.end** で共有できるのが理解できると思います。ちょうど、クラス変数の様な形です。

この場合、**state** を直接グローバル領域から書き換えることはできないですが、**app.start** と **app.end** からアクセスすることができます。なので、**app.start**, **app.end** はグローバル領域または、他のローカル領域から実行をしたとしても、この **state** は、その 2 つの関数で共有する形で利用することができるわけです。

この様に無名関数を利用して、機能ごとに関数（ここではメソッドと言います）をパッキングして、必要な部分のみをグローバル領域からアクセスできるようにすること（**app** がまさにそれです）、それぞれで共有したい変数は、無名関数直下を書くということで、オブジェクト指向に近い形で、プログラムを書いていくことができます。

UI システムが大規模になればなるほど、関わる人数やコード量も増えるので、このようにプログラムを分割し、効率的に管理していくことにより、メンテナンス性の高いすっきりとしたコードを書いていくことができます。

プロトタイプ（高度な話題）

JavaScript は、プロトタイプ型言語と言われています。

プロトタイプを語るには、**new**, **prototype** といった言葉が出てきますので、先に下のプログラムで解説をしていきます。

```
<script type="text/javascript">

var class = function(name){
    this.name = name;
};

class.prototype.getName = function(){
    return this.name;
}
</script>
```

すごく簡単なプログラムを書いてみましたが、これがプロトタイプ型言語の神髄です。これは、クラスを作るような形で、まさにプロトタイプ（原型）を作るということです。

実行する場合には、

```
var obj = new class("山田太郎");
```

```
console.debug(obj.getName());
```

これは、山田太郎がコンソールに出力されます。

new は、作成したプロトタイプを初期化する方法です。複製され、新しいオブジェクトが作成されます。仮にしたの **obj** に入っているオブジェクトを変更したとしても、上のプロトタイプには影響を与えません。

this

また、**this** というのが気になると思いますが、これは、自分自身を参照するために **JavaScript** が用意している名前です。その関数によって、**this** が指す先は違いますが、上の場合は、**this** はそのオブジェクト自体 (**class**) を指します。

this がどこを指しているかわからない場合は、**this** を **console.debug** で出力してください。自分の位置がわかります。

みんなで書くために（高度な話題）

モバイルアプリ全盛の時代になり、サーバー側とクライアント側がより明確に、役割分担がされてきました。よりインタラクティブな **UI** を求められるようになりフロントエンドのコード量が増大し、また、**UI** がよりアプリやサービスの成功を左右するファクターとなりつつある現在のトレンドで、**UI** の仕様変更・改善が多くなってきています。

その中で、フロントエンドエンジニアは、よりメンテナンス性がよく、変更に強いコードの書き方を要求されます。それをうまくやりこなし、長く生存できるコードを書いていくためのいくつかのエッセンスをみていきます。

コードの統一

この話はずっと言われていることですが、コードに自分の癖を出さないというのが、最大の論点になります。そのために、プロジェクトを開始する前に、簡単なコーディング規約や約束事を決めておくのが重要です。単純な話からしていくと、インデントの付け方（タブでつけるのか、スペースでつけるのか）、**function** の後に、スペースをつけるのか、**}** は下げて作っていくのかという思想の統一まで。0 からスタートさせる場合は、特に気をつけるべきです。

私たちがよく行うことは、アーキテクト（設計者）にその部分を司ってもらうという明確なルールを定めます。エンジニアには、アーキテクトが示す設計を指針として書くことを求めます。評価指標に、そのような指針を受けて、同じようなコードが書けたかどうかを入れ込むと気合いが入るかもしれません。

また、アーキテクトは、特にウェブサービスにおけるフロントエンド周りを実装する場合に、以下の指針を示すべきです。

- **JavaScript** の場合は、クラス化の方法（**CommonJS**、**prototype** を利用？などの指針）
- クラス化する単位とファイルシステムへの配置の定義
- フロントエンドにおける画面の書き方（**DOM** の生成の仕方や画面を度の単位で分割して書かせるか）
- バックエンドとの通信方法（通信用モジュールの提供・必ず一度ラップすること）
- グローバル変数を扱う場合のルール・設定ファイル（設定変数）のルール

ある程度は、経験により共有できますが、理想は、本格的にプロジェクトを始める前に、上記の指針をまとめたモックアップをアーキテクトが作成し、進めるべきです。

エンジニアは、このコードの指針に従って、できるだけ自我を出さずに新しい画面を作り込んでいくというのが、要求されるスキルとなります。

コメントをチームで共有

複数人で仕事をする場合に、最も重要なのが、コード上でコミュニケーションをとれるかということです。不思議な書き方をしてしまいましたが、複数人で同じプロジェクトを扱う場合、または長くメンテナンスや改修を続けていくコードを書いていく場合に非常に重要になります。

コメントは、いくつかの種類に分けられます。

以下は、Google が定義する JSDoc コメントを参考にしています。

参考：<http://www38.atwiki.jp/aias-jsstyleguide2/pages/14.html>

最上位・ファイルレベルのコメント

ファイル1つか、クラス1つに対して、頭の部分に書いていくコメントです。ブロックコメントとして書いていきます。

```
// Copyright 2009 Google Inc. All Rights Reserved.
```

```
/**
 * @fileoverview ファイルの説明、使い方や依存関係に
 * ついての情報。
 * @author user@google.com(名 姓)
 */
```

クラスの作成者、ライセンスとクラス自体の説明を加えておけば十分です。

メソッドヘッダーコメント

関数1つ1つに記述していくコメントです。通常は、ブロックコメントとして書いていきます。

メソッドヘッダーは、一番大事ですが、割と抜けていることが多いので、癖をつけて記述していくようにしましょう。

```
/**
 * 長い param/return アノテーションの説明文の折り返し方を示します。
 * @param {string} foo これは1行でおさめるには長すぎるパラメータの
 *   説明文です。
 * @return {number} この戻り値の説明文は長すぎて、とても1行の中には
 *   入りきりません。
 */
```

```
project.MyClass.prototype.method = function(foo) {
  return 5;
};
```

@param には、引数として指定される項目についての説明を書いていきます。複数の引数がある場合には、@param を複数書いていきます。

@param {引数の型} 引数の名前 引数の説明

※ 半角スペースで3つの要素を書いていきます。

@return には、戻り値の型と説明を書いていきます。引数が複数ある場合はないので、1 つで大丈夫ですね。

@には、他にもいろいろな内容を記述することができますが、運用で無理のない範囲で記述をしてもらうことをおすすめします。

通常コメント

if や、**switch**、**for** などの制御文を書いていくときに、それ自身が何をしているのかを書いていくことが後でコードを見直すときに有効になります。また、ローカル変数（関数の中で **var** で宣言されている変数）も、それ自体が何を管理している変数なのかをコメントとして残していくのは、コード上でコミュニケーションをとっていくときに有効な手段となります。

メソッド内は、特にこだわりがない場合、**//** のコメント記法を利用して書いていきます。1 行で収まるように簡素に書いていくのが重要です。

CreateJS 速習

CreateJS は、HTML5 の **canvas** 機能を利用して（オールド IE の場合は **SVG**）、**Flash** のようなインタラクティブなムービーやゲームを制作していくためのフレームワークです。

通常、HTML5 でインタラクティブなアニメーションやゲームを制作して行くには、複雑なプログラムを書いていく必要がありますが、このフレームワークを利用することにより、簡単にキャラクターを動かしたり、衝突判定をしたり、音を鳴らしたりすることができます。

また、**JavaScript** で書かれていますので、基本的にモバイル系デバイス（iPhone, Android のウェブブラウザ）でも動作します。

jQuery・CreateJS をつかってみる

早速、jQuery・CreateJS を使ってゲームを作ってみましょう。

まずは、ゲームを動かすためのステージを作っていきます。

名称を **index.html** か **default.html** (Windows ストアアプリの場合)として、作成してください。

HTML を記述する

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<!-- 1 -->

<!-- 2 -->
</head>
<body>
    <canvas id="stage" width="500" height="500" />
</body>
</html>
```

ここでは、**canvas** タグを書きました。

1. フレームワークを読み込む

jQuery・CreateJS の機能を利用するには、フレームワークが記述してある JS ファイルを読み込ませる必要があります。次に、今回利用するフレームワーク（CreateJS）を配置して読み込みます。

js というフォルダを作成し、

その中にダウンロードしてきた

jQuery のフレームワークファイル

jquery-1.8.2.min.js

※Windows8 の場合は、FIX されたものを導入。

CreateJS のフレームワークファイル

easeljs-0.5.0.min.js

tweenjs-0.3.0.min.js

を読み込ませます。

<!-- 1 -->の部分に以下のコードを追加していきます。

```
<script charset="utf-8" src="js/jquery-1.8.2.min.js"></script>
<script charset="utf-8" src="js/easeljs-0.5.0.min.js"></script>
<script charset="utf-8" src="js/tweenjs-0.3.0.min.js"></script>
```

これで、このアプリで **jQuery・CreateJS** が利用できるようになります。

2. アプリエントリーポイント（アプリ起動部分）を作っていく

アプリのエントリーポイントというのは、実際に **HTML** 上でアプリケーションが動いていくために、はじめに実行されるプログラムだと思ってください。ですので、アプリを使うための準備をしていくというイメージです。たとえば、画面にボタンをおいたり、初期画面を表示したり、そんな感じのものです。

```
<script type="text/javascript">
$(document).ready(function(){
    console.debug('Started');

    /* 3 */

    /* 4 */

    return;
});
</script>
```

先ほど関数を使うで、上記の関数の使い方を説明しました。

この場合は、**HTML** が準備されたら、**ready** の中の関数（メソッド）を実行してくださいという **jQuery** であらかじめ用意された方法です。

高度な話題

これを書かずに、直接 **script** タグの中に書いた場合は、正常にプログラムが動かない場合があります。なぜなら、**HTML** がすべて用意される前に、メインのアプリプログラムを走らせてしまうため、**HTML** 中の要素が取得できない可能性があるからです。

3. CreateJS を使うために、ステージを作成する

次に、**CreateJS** を実際に使っていくためにステージを作成します。**CreateJS** はステージの中でしか動かないので注意してください。ステージは **canvas** タグの中に作られます。

3 の部分に以下のコードを書いていきましょう。

```
var stage = new createjs.Stage("stage");
```

たったこれだけです。これでステージの準備ができました。

createjs.Stage の引数（先ほど学びましたね）は、**canvas** タグの ID で示しているとおりで。ですので、HTML が初期化された後に、**canvas** タグで ID が **stage** とつけられているものがここまで、**CreateJS** で使えるように初期化されたということです。

見た目上変わっていませんが、では、コンソール画面でみていきましょう。

デバッグ部分で説明したエレメントタブで、**body** タグの下に、**canvas** タグが存在しています。エラーが発生せず、ここが表示されていれば、初期化されている証拠です。

そのタグを触ると青枠で、画面上に表示されたと思いますが、それがわかりますでしょうか？

青枠が全画面に表示されず、**500px x 500px** で表示されています。では、それを全画面で表示できるように変更していきましょう。

3 の前にプログラムを書いていきます。

```
var width = $(window).width();  
var height = $(window).height();
```

```
$('#stage').attr('width', width);  
$('#stage').attr('height', height);
```

先ほど、でてきたローカル変数を宣言しています。わかりますか？

また、**jQuery** を使って、ブラウザの幅・たかさを取得して、ステージを作成する **canvas** タグの幅・高さを設定しているわけです。

もう一度実行して、コンソールのエレメントタブでみてください。

全画面に青枠が表示されていますよね？

キャラクターを配置してみる

先ほど作成したステージにゲームの主人公キャラクターを配置します。**CreateJS** のスプライトという機能を使っていきます。スプライトとはその中に、画像や、同じようなスプライトを配置していける箱のようなものだと考えてください。

たとえば、キャラクターが、頭、手、足がそれぞれ動く（パンチさせる・走るなどのアクションをさせるため）としましょう。ステージの中に、それぞれのパーツを並列でおいて、キャラクター自身を上下左右に動かしたい場合、頭、手、足をそれぞれ動かさないといけませんか？それをやっていると、時間がいくらあっても足りませんよね？

スプライト機能は、それを解決するために、頭、手、足を 1 つのグループに登録する様な機能です。それをまとめたスプライトの名前を主人公だとすれば、主人公を動かすプログラムを作るだけで、その中にあるパーツも一緒に動いてくれるわけです。

まずは今まで書いたコードに、付け加える形で行っていきましょう。**/* 4 */**の部分に、以下のコードを付け加えていきます。

```
//スプライト
```

```
var obj = new createjs.Container ();
```

```
//頭、体、手、足を付け加えてパーツの位置を調整
```


JavaScript 速習

```
var body = new createjs.Bitmap();
obj.addChild(body);

var handLeft = new createjs.Bitmap();
obj.addChild(handLeft);

var handRight = new createjs.Bitmap();
obj.addChild(handRight);

var legLeft = new createjs.Bitmap();
obj.addChild(legLeft);

var legRight = new createjs.Bitmap();
obj.addChild(legRight);

/* 5 */
```

キャラクターを動かしてみる（リスナーの追加・イベントを発火）

キャラクターが作成できたら次に、それを動かしていきます。

先ほど作成したキャラクターのスプライトに、リスナーを追加していきます。たとえばパンチをする機能を追加します。

先ほどのプログラムにリスナーを追加します。

```
var obj = new createjs.Container();
```

...(省略)

//本来は、`addEventListener` などカスタムイベントを追加する形で

```
obj.punch = function(){
    handRight.rotate = 100;
    stage.update();
return;
};
```

このように、そのオブジェクトに対して新しい動作を付け加えます。

リファレンスをみて、名前が重ならないようにしましょう。

実行するには、下記のようにします。

```
obj.punch();
```

他のフレームワークには、明確なカスタムリスナーとイベントを発火させるような仕組みが組み込まれているものもありますが、ない場合には、オブジェクトを拡張して上記のように自身のメソッドをコール（実行）できるような形にしておく、見通しよく、便利に利用することができます。

マウスの操作に反応させる

マウス（タッチ）に反応させるようにしていきましょう。

`Container` は、マウスまたはタッチに反応するリスナーをあらかじめ持っています。

JavaScript 速習

```
obj.onClick = function(e){
    obj.punch();
    return;
};
```

これで、キャラクター自身をタップしたときに、パンチを繰り出すようになります。

では、画面をタップしたときにパンチを繰り出すようにするにはどうしたらいいでしょうか？現在、画面を支配しているのは、前に説明した **Stage** ですね。 **Stage** にもマウス操作に反応するリスナーを持っています。

```
stage.onClick = function(e){
    obj.punch();
    return;
};
```

これで、画面をタップしたときに、キャラクターがパンチをするようになります。

主人公を操作するコントローラをつける

もう少し発展させて、画面上の特定のボタンをタップすると、パンチを繰り出すようにしましょう。まずは、ボタンを配置していきます。

```
//コンテナーを作成して、その下にビットマップを追加
var bt = new createjs.Container();
var b = new createjs.Bitmap(URL を指定);
bt.addChild(b);
```

```
//コンテナーがクリックされたときに punch を実行
bt.onClick = function(){
    obj.punch();
    return;
}
```

```
//ステージにボタンを配置
stage.addChild(bt);
```

キーボードの操作に反応させる

コントローラができれば、今度はキーボードでも操作できるようにしていきます。スペースをおしたら、パンチをするようにします。

```
//jQuery の記述方法
$(window).on('keydown', function(e){
    //ここに実行したいプログラムを書いていきます
    /* 7 */
});
```

keydown で実行されるコールバックの引数 **e** には、**keyCode** が入っています。
console.debug(e.keyCode)で実際に出力してみてください。

実際にキーを押下すると、そのキーに対応した **keyCode** が返却されます。

7の部分にかきのとおり記述します。

```
switch(e.keyCode){
    //スペースが押下されたときに実行
    case 32:
        //実行したい処理を記述
        break;
    //右キーが押下されたときに実行
    case 39:
        break;
    //左キーが押下されたときに実行
    case 37:
        break;
}
```

キャラクターを動かす

ここまでできたら、主人公を動かすことを考えてみます。
十字コントローラを作って、それぞれ左右上下に動くようにします。
まずは、先ほど作成したキャラクターを拡張していきます。

```
obj.move = function(dir){
    switch(dir){
        case 'up':
            obj.y = -20;
            break;
    }
    return;
}
```

つづいて、キーボードの矢印キーに反応する様にします。

敵を作ろう（キャラクターの応用と機能のカプセル化）（高度な話題）

敵は、主人公を参考に動くようにしていきます。敵は複数いるといいですね。1つのコードを複製して、たくさんの敵を自動的に動かしていく効率的な書き方を学びましょう。
ここから、JavaScriptの少し難しい話題である **prototype** の考え方が出てきます。これを理解すればあなたも、かなりの JavaScript マスターになれます。

プロトタイプとは

先ほどの説明で、プロトタイプとは、クラスを作っていくような感覚で、原型を作っていくというお話をしました。実際に敵の原型を作っていきましょう。

```
/**
 * コンストラクタ
 * new されると、これが一番はじめに実行される
 */
var enemy = function(){
    this.base = this.create();
```

JavaScript 速習

```
        return;  
};
```

上記 **enemy** という関数の原型を作成するために以下のコードを書いています。

//敵の体、手、足などを作る

```
enemy.prototype.create = function(){  
    //基本的に主人公を作ったときと同じコードをここに記述する  
    //親のコンテナを戻り値として返すことにより、その後にアクセスできるようにする  
  
    //スプライト  
    var obj = new createjs.Container ();  
  
    //頭、体、手、足を付け加えてパーツの位置を調整  
    var body = new createjs.Bitmap();  
    obj.addChild(body);  
  
    var handLeft = new createjs.Bitmap();  
    obj.addChild(handleft);  
  
    var handRight = new createjs.Bitmap();  
    obj.addChild(handRight);  
  
    var legLeft = new createjs.Bitmap();  
    obj.addChild(legLeft);  
  
    var legRight = new createjs.Bitmap();  
    obj.addChild(legRight);  
  
    return obj;  
};
```

//動くためのメソッド

```
enemy.prototype.move = function(dir){  
    switch(dir){  
        case 'up':  
            obj.y = -20;  
            break;  
    }  
    return;  
};
```

このように、変数名.**prototype** の中に、プロパティを作り、関数を追加していくことにより、その変数自体を拡張していくことができます。実際に利用するときには

```
var en = new enemy();  
en.move('up');
```

というような形で、**enemy** 自体を複製し(new を行くと複製されて、**prototype** の中に設定されているプロパティを引き継ぐことができる)、利用することができます。

敵を倒す武器を作ろう（衝突判定）

先ほどの主人公が、武器を投げて敵をやっつけるというルールを考えてみましょう。
武器を発射して、当たったと判定されたときに、敵を削除します。

今回は、当たったという判定をどのように行うかにフォーカスを当てます。
まずは、主人公を作成したときにも記述した方法で、武器を作ります。

```
var c = new createjs.Container();
var b = new createjs.Bitmap(画像の URL);
```

```
c.x = 100;
c.y = 100;
```

武器を表示して、飛ばす方法は、下記の通りです。

```
createjs.Tween.get(c).to({x: 0}, 1000).call(function(){
    //このアニメーションが終了したら実行される
});
```

上記のプログラムは、コンテナの **X** 座標が **0** になるまで、動かし続けるというプログラムです。
X 座標が **0** になったら、**call** でしてされた関数が実行されます。

動いている間に敵にヒットしたかどうかを判別するには、下記のように記述します。

```
//動いている間、一定の間隔で実行される onTick というイベント
c.onTick = function(){
    /* 8 */
}
```

実際に、敵とヒットしているかどうかをみるには、敵の位置座標を武器のローカル座標に変更し、**hitTest** で検査していきます。

```
//enemy が敵のオブジェクト
//c が武器の座標
```

```
var co = enemy.globalToLocal(c.x , c.y);
```

```
//武器が、敵に当たっているかどうかを検査
if(c.hitTest(co.x, co.y)){
    //当たったときに実行をするプログラムをここに記述します
    return;
}
```

これで、動いている間に一定間隔で、敵とヒットしているかどうかを判別することができます。

点数をつけよう（機能のカプセル化）

点数をつけていきます。

さて、ゲームの中で、点数がつく瞬間はいつでしょうか？

ゴールをした時は当然ですが、敵を倒したとき、なにかアイテムをゲットしたときなどいくつかのシーンが考えられます。点数をつけるのは、いくつかのプログラムを書かなくてはならないので、共通化できるのが望ましいです。そこで、以下の書き方を覚えていきましょう。

```
var app = app || {};  
app.point = {};  
  
(function(){  
  
    var point = 0;  
  
    //ポイントを初期化  
    app.point.init = function(){  
        point = 0;  
        return;  
    }  
  
    //現在の point を取得する  
    app.point.get = function(){  
        return point;  
    }  
  
    //ポイントを追加する  
    app.point.set = function(n){  
        point += n;  
        return;  
    }  
  
})();
```

これは、さきほど説明した無名関数を利用して、カプセル化をしていく方法ですね。

変数 **point** を直接変更せずに、**app.point.get**, **app.point.set** で設定していくのがポイントです。さらに、何でポイントを取得したなどを取りたい場合（クリアしたときに表示したいですね？）は、下記のように変更することで、対応することができます。

```
var app = app || {};  
app.point = {};  
  
(function(){  
  
    var point = 0;  
    var pkind = {};  
  
    //ポイントを初期化  
    app.point.init = function(){
```

```
        point = 0;
        pkind.bonus = 0;
        pkind.block = 0;
        pkind.beat = 0;
        return;
    }

    //現在の point を取得する
    app.point.get = function(){
        return point;
    }

    //現在の point を取得する
    app.point.getDetail = function(){
        pkind.total = point;
        //オブジェクトとして返すように変更
        return pkind;
    }

    //ポイントを追加する
    app.point.set = function(n, k){
        //総合ポイントを保持
        point += n;

        //種類で振り分ける
        switch(k){
            case 'block':
                pkind.block += n;
                break;
            case 'beat':
                point.beat += n;
                break;
            case 'bonus':
                pkind.bonus += n;
                break;
        }
        return;
    }
}());
```

さて、ここで大事なことがあります。
一度書いたコードを壊さずに拡張しましたよね？

たとえば、最初からランキングを詳細に分けて表示したいと思っていたら、特に気にする必要はないのですが、最初は、トータルポイントで表示したいという風にチームで決まっていた、あとから、詳細のポイントを表示したいということになったらどうでしょうか？

もし、最初のように書いておかず、同じようなコードをばらばらに書いていたら、その箇所を全部修正しなければいけないですよ？これが、共通化する意味です。

JavaScript 速習

最初にコードを書くときは、見通しが悪かったり、なんでこんなに分割して書くんだろう？と思ったりするものですが、このように、最初から細分化（細分化しすぎるとただめですが（笑））をしておくことによって、あとから上がってくる仕様変更に強いプログラムができあがります。

これが当たり前のできるエンジニアが、いいエンジニアと呼ばれます。

さて、ここまでポイントを計算するところを書いてきましたが、実際にポイント进行操作する側を書いていきましょう。

たとえば、敵を倒したとき（武器が敵に当たったとき）にポイントがゲットできることとします。

```
app.point.set(100, 'beat');
```

と書くことにより、100 ポイントを倒したというアクションで登録できます。

時間制限を作ろう（定期的にプログラムを動かす）

JavaScript で時間を制御するには、以下の方法を使います。この関数は、JavaScript 標準のものです。

実行時間に到達したら、1 度だけ実行するのが、**setTimeout** です。

```
setTimeout(コールバック, 実行時間);
```

それに対して、一定期間繰り返して実行するのが、**setInterval** です。

```
setInterval(コールバック, 繰返時間);
```

実行時間・繰返し時間は、ミリ秒で設定します。ミリ秒といわれてもぴんときないので、下記を参考にしてください。

5 秒後に実行する場合は、**5000**

5 分後に実行したい場合は、**300000**

わかりますか？秒 x 1000 をすればいいわけです。

では、実際に書いてみましょう。

```
<script type="text/javascript">
```

```
//セットした 1 秒後にコールバックが実行されます
```

```
var inte = setInterval(function(){
    console.debug("5 分たちました");
}, 1000);
```

```
//途中でそれを中断したい場合は、以下のようにします
```

```
clearTimeout(inte);
```

```
//セットした時間から 5 分ごとにコールバックが実行されます
```

```
var inte = setInterval(function(){
    console.debug("5 分たちました");
}, 300000);
```


JavaScript 速習

```
//途中でそれを中断したい場合は、以下のようにします
clearInterval(inte);
</script>
```

今回の場合、5分でゲームを終了させるので、スタート時に **setTimeout** を実行し、5分後に、コールバック関数を実行できるようにします。また、途中でゲームをやめる場合もありますので、やめるときには、**setTimeout** を実行したときに戻り値として取得した値を **clearTimeout** の引数として与えて実行します。

高度な話題 ランキングを作ろう（点数を保存する） - Local Storage 編

ゲーム内で獲得した点数を保存するには、**HTML5** の機能を利用していきます。
HTML5 ではローカルストレージ という、ブラウザまたはアプリ側に値を保存しておく簡単なデータベースの機能が搭載されていますので、特になにかを新しくインストールしなくても、そのままデータベースを利用することができます。
簡単に、**Local Storage** の例を見ていきましょう。

```
//ローカルストレージを使えるように
var db = window.localStorage;
```

次に、プロパティ名を指定して、値を挿入していきます。

```
//setItem(プロパティ名, 値)を指定する
db.setItem("ranking", 1000);
```

値を取得する場合は、**getItem** を利用します。

```
var t = db.getItem("ranking");
//これで先ほど挿入した 1000 が取得できる
//ブラウザで、再読み込みをしても、値が取得できることを確認してください。
```

Local Storage をクリア（全部のデータを削除）する場合は、下記のとおり実行します。

```
//すべてのデータを削除
db.clear();
```

高度な話題 ランキングを作ろう（点数を保存する） - openDatabase 編

HTML5 では、さらに高度なデータベースを利用することが可能です。
Local Storage とは違い、本格的な **SQL** を利用して、データをローカルで管理することが可能です。

ローカルストレージに値を保存する

特定の ID と値を **app** というデータベースの **ranking** という名前のテーブルに保存します。

```
var db = window.openDatabase("sampledb", "1.0", "Sample Database", "1048576");
```

第 1 引数から、データベース名、データベースのバージョン、表示名、見積りサイズ（バイト数）を指定します。次に、テーブルが存在しない場合、テーブルを作成します。

JavaScript 速習

//SQL で、テーブルを作成します。

```
db.executeSql("CREATE TABLE IF NOT EXISTS ranking (id integer primary key autoincrement, name, val)");
```

//SQL で、値を挿入していきます

```
db.executeSql ("INSERT INTO ranking (id, value) VALUES (1, 300);");
```

保存された値の確認方法は、以下のようにおこなっていきます。

今までデバッグに利用してきたコンソール機能のリソースタブを確認します。

左側のツリーの中に **Local Storage** という項目があります。それをクリックすると、そのアプリ内で利用しているデータベースの一覧が表示されます。表示されている項目の中身をみていくことにより、実際にプログラムからレコードが挿入できたか確認することができます。

自分が想定したとおり、値が挿入されていることが確認できたでしょうか？

高度な話題

Local Storage の機能は、**SQLite** の仕様にに基づいています。

さらに機能を知りたい場合には、**SQLite** について調べてみてください。

ローカルストレージから値を取得する

次に値を取り出してみます。

値を取り出して、実際にランキング表を表示するところまで書いてみましょう。

まずは、保存した値を取得してみます。

```
var db = window.openDatabase("sampledb", "1.0", "Sample Database", "1048576");
```

//SQL で、値を取得していきます

```
db.query("SELECT FROM ranking ORDER BY value DESC;");
```

SQL は基本的に英語の文法のように書いてきます。上の SQL は **ranking** テーブルから、**value** を降順に並べて、すべて取得するという構文です。このように英語で書けるのは、簡単そうに見えますね。

高度な話題

たとえば、特定の ID の値のみを取得したい場合は、以下のように SQL を書くことができます。

```
db.query("SELECT FROM ranking WHERE id = 1 ORDER BY value DESC;");
```

WHERE 構文を使うことにより、特定のレコードを取得できます。