HTML + CSS
&
javascript

実践

Chapter.1

動きのあるサイト

動きの表現（アニメーション）

JavaScript では、いわゆるパラパラ漫画のような要領で、少しずつ要素の表示される位置や色を変化させアニメーションを実現する。

そんなアニメーションの処理を実現するには、いくつかの選択肢が考えられる。

しかし実際には、JavaScript で実装する場合に限っては、ほぼ「これを使っておけば間違いない」というやり方が確立している。ここでは一応、そのスタンダードな方法以外の部分も簡単に解説。

setInterval
setTimeout

setInterval と setTimeout は、名前の雰囲気が何となく似ているが、動作内容は少々違っている。詳細は省くが利用するなら　setTimeout のほうを使う。setTimeout は、一定時間経過後、指定された関数を時間差で呼び出すという機能があり、かつてはこれがアニメーションの実現に利用されていた。

やってみよう

ボタンを押してから、3 秒経過してからアラートが出るような仕組みを作ってみよう。

```
function timerFunc () {
    console.log('timer functin!');
    setTimeout(timerFunc, 1000); //  １秒後に自分自身を再帰呼び出し
}
```

requestAnimationFrame

setTimeout は次の呼び出しまでの時間をミリ秒単位で指定するものだったのに対して、requestAnimationFrame は呼び出しのタイミングを、ブラウザ側で最適に制御してくれる。
特に理由がない限り、アニメーション処理にはこちらを使う。

```
function timerFunc () {
    console.log('timer functin!');
    requestAnimationFrame(timerFunc);
}
```

※ setTimeout の時とは異なり経過時間を指定していない点に注目

やってみよう

簡単なアニメーションループを作り、DOM を使ってエレメントの座標を変化させるアニメーションを記述してみよう。

Chapter.2

組み込み関数

JavaScript 組み込みの関数

JavaScript では、あらかじめ言語仕様に組み込まれているオブジェクトや関数というものがある。いわゆる、ビルトインと呼ばれるもの。

先に解説したとおり、JavaScript ではあらゆるものがオブジェクトなので、ビルトイン関数とは呼ばず、組み込みのオブジェクトと呼ぶのが本来は正しい……が、わかりにくいので、ビルトイン関数という呼び方をするのが一般的。

ここでは代表的な、よく使うものだけ解説。

Date

Date 関数はその名の通り時間や日付に関する多数の処理を行うことができる。

new 演算子を使ってインスタンス化したあと、目的のメソッドなどを呼び出して機能を利用する。

```
var d = new Date();

console.log(d.getYear() + 1900);
```

※ 上記は現在の年の数値を取得しコンソールに出力する例

やってみよう

年の他に、月や日、時間や分など、様々な単位で時間を取得して、表示してみよう。
また、タイムスタンプがいったい何を表しているのか、もしわからないようなら理解しておこう。

Array

Array 関数を用いれば、JavaScript で配列を利用することが可能になる。JavaScript における配列には、連想配列という概念がなく、原則として添字はすべて数値になっている。（連想配列的なものとしてはオブジェクトがあるが詳しくは後述）

```
var a = new Array();

a.push(1, 2, 3); // push メソッドは配列に要素を追加する
console.log(a);
```

やってみよう

配列は割りと奥深いので、小さな課題をこなしながら徐々になれていくほうがいいでしょう。

まずは 100 個の要素を持つ配列のすべてに値を格納するには、どんな書き方ができるか考えてみましょう。

Math

Math は先述の Date や Array とは異なり、完全に静的なオブジェクトとして定義されており、インスタンス化することができない。
直接、プロパティとなっている関数、及びプロパティそのものを呼び出したり参照したりしつつ利用する。
三角関数やランダムなど、利用する頻度は意外にも高い。

```
var p = Math.PI; // 円周率

var r = Math.random(); // ランダムな数値の取得

var q = Math.sqrt(25); // 平方根を求める
```

やってみよう

Math には今後お世話になることが多いです。まずは乱数を使った簡単なプログラムなどを記述してみるのがいいでしょう。

random でどのような値が返却されてくるのか、何度か実行して調べてみましょう。

Chapter.3

変数・配列・オブジェクト

変数・配列・オブジェクト

JavaScript では、どんなものでも実体としては何らかのオブジェクトなので、関数ですら、変数に代入できてしまうという話は以前にもした。

こういった他の言語には見られない特徴がある JavaScript では、変数や配列、オブジェクトの扱いが微妙に紛らわしく、わかりにくい。

ここで再度、復習する。

配列

配列は、Array 関数をインスタンス化することで利用できる。

ただし、それ以外にも配列を定義する方法があり、これには配列リテラルと呼ばれるものを使う。

配列リテラルを使おうと、あるいは Array 関数を使おうと、生成される配列の性質は変わらない。しかし一般には、可能な限り配列の生成には配列リテラルを使うべきとされている。どうしてだろう？

```
var a = new Array();
var b = [];
```

※ 両者は全く同じ意味になり、後者が配列リテラルを使った配列の生成の例

配列

JavaScript では、ビルトインの関数を、オーバーライドして書き換えることができてしまう。つまり、Array 関数からインスタンス化する方法の場合は、万が一には別のものに置き換わっている可能性がゼロとは言い切れない。

```
Array = function(){ alert('array?'); };

※ ビルトイン関数を別の関数で上書きしている例（ふつうはやらない）
```

このような理由から、可能な限り配列の初期化には配列リテラルを用いるべきだという風潮が生まれた。ただ、これについて過度に捉えるよりも「わかりやすく記述すること」により注力したほうがよい。

オブジェクト

オブジェクトは、配列と少し似ている部分もあるが、まったく別の概念。

配列と同じように、関数からインスタンス化することもできるが、オブジェクトリテラルを利用して初期化するのが一般的。むしろ、オブジェクトの場合は原則オブジェクトリテラルを使ってできないこと、というのが無いので、リテラルによる生成だけ理解できていればよい。

```
var a = new Object();
var b = {};
```

※ 両者は全く同じ意味になり、後者が配列リテラルを使った配列の生成の例

オブジェクトへの値の格納

オブジェクトは、キーと、その値がセットになった構造をしている。
配列は添字に数値しか使えなかったが、オブジェクトならキー部分に文字を指定して名前をつけることが可能。
JavaScript ではオブジェクトや配列が様々なところで頻繁に利用されるので、まずはしっかり慣れておくことが重要。

```
var obj = {}; // 空のオブジェクトを生成する場合
var obj = {
    prop1: "property", // 文字列を値として格納
    prop2: 123,        // 数値を値として格納
    prop3: []          // 値には配列も当然指定できる
};
```

やってみよう

独自のプロパティを持たせたオブジェクトを生成し、それをコンソールに出力してみよう。オブジェクトを出力するとコンソールにはどのように表示されるのか、確かめておこう。

オブジェクトの参照方法

オブジェクトの中身を参照する際には、ピリオド記法か括弧記法でアクセスする。

これはどちらが優れているということは一概には言えないが、あまりそれぞれがゴチャゴチャに混ざっているのは良くない。書きやすいほうに統一しておくことが望ましい。

```
var obj = {
    prop1: "property",
    prop2: 123,
};

console.log(obj.prop1);
console.log(obj["prop1"]);
```

※ 上記はアクセス方法は違うが、参照している要素はどちらも同じ

やってみよう

このようにふたつのアクセス方法が用意されているが、括弧記法でないとできないこと、というのもある。

それはどんな処理だろう。

オブジェクトへのプロパティの追加

オブジェクトは、初期化時以外にも、任意のタイミングで自由にプロパティを追加することができる。
追加するの場合も、ピリオド記法でも括弧記法でもどちらも利用できる。

オブジェクトのプロパティの書き方はここまで見てきたように、異なる方法で、同じ効果を得られる書き方が複数存在するので、極力統一して記述するようにしたほうがよい。

```
var obj = {}; // 空のオブジェクトを生成する場合
obj.prop1 = "property";
obj["prop2"] = 123;

※ 上記はいずれも正しく期待通りに動作する
```

Chapter.3.5

やってみよう

配列に慣れよう

配列を使った処理に慣れるために、簡単なゲームを作ってみる。

ここではサンプル用の雛形を用意したので、それを改造していき、シンプルな「まるばつゲーム」を作ってみる。

（別名では三目並べとも言う）

- 3x3 のマス目（テーブルタグで作る）があり、プレイヤーはふたり
- プレイヤーは交互に、お互いのマークとなる ○ と × を空いているマスに入れる
- いずれかのプレイヤーの模様が一直線に並ぶように配置されれば勝利
- ゲームの性質上、引き分けとなることがありえる

参考：http://jsdo.it/tef_tef/dKcp

※ なお、オブジェクトは無理に使わなくてもよいが配列は使ってみよう

Chapter.4

スコープと無名関数

スコープ

JavaScript のスコープは、とても重要なもののひとつ。

スコープの概念を正しく理解していないと、簡単にグローバル汚染などが起こるので注意が必要。また、他の言語と比較すると、少し独特な仕様になっているようにも思えるので注意。

※ スコープとは、変数や関数を参照できる範囲のこと。

※ 普通、スコープの外側に出てしまうと、どのような変数や関数があるのかはわからなくなり、同時に、仮に名前がわかっていてもその中身を参照することはできない状態になる。

ブロックスコープ？

C 言語などの場合、スコープは波括弧で作られたブロックの中に収められるのが一般的。

一方の JavaScript には、ブロックスコープという概念はない。

それではどのような判断基準でスコープが決まるのかというと、意外なほど単純に「 関数単位で区切られる 」という認識でいればよい。

```
var A = 0;

var myFunc = function(){
     var A = 10;
}

myFunc();        // myFunc を実行する
console.log(A); // ここで何がコンソールに出力される？
```

グローバルスコープ

先ほどの例にあったように、JavaScript では関数の中にスコープが限定される。
どんな関数にも属していない箇所で宣言された関数や変数は、すべてグローバルスコープになる。一般に、グローバルスコープに定義を行うのはマナー違反とされており、なにかしらの関数で包み込んで、限られたスコープしか適用されないように実装するのが普通。
また関数の内部であっても、var を使って宣言していない変数は暗黙のグローバル変数になってしまうため、注意が必要。

```
var A = 100; // グローバルスコープで宣言は極力しない

var myFunc = function(arg){
     var A = 10;
     var B = 10;
     var C = 10;  // 極力限られたスコープで定義するように
}
```

グローバルスコープ

先ほどの補足になるが、例えば次のように記述した場合、 for 文で使っている変数 i はグローバルスコープになってしまう。

```
var myFunc = function () {
    for(i = 0; i < 10; ++i){
        console.log(i);
    }
}
```

これは、変数 i を「 var 」を使って宣言していないので、宣言されていない変数＝グローバル変数と処理系が判断してしまうから。もし同じスコープ内で i という名前の変数を使っている処理があれば、この関数が動作するたびに影響を受けてしまう。

グローバルスコープ

また、C 言語などのようなブロックスコープという概念が JavaScript にはないので、次のように記述しても、スコープを限定することはできない。

```
var i = 0;
{
    var i = 2;
    console.log(i);
}
```

※ 上記のようなコードを実行しようとすると、同じ変数名が重複しているためエラーになってしまう

無名関数(即時関数)

このように、JavaScript では、関数をうまく利用することでスコープを限定するが、これに関連する方法としてよく利用されるのが無名関数、あるいは即時関数と呼ばれるもの。

これは、即席で作った名前のない関数のことで、名前がないため、再度呼び出すことができず、一度だけ実行される即席の処理などに利用される。

```
var myFunc = function(){
    console.log('my function');
};
```

※ 変数に代入するために名前のない関数を即席で作っている例

無名関数によるカプセル化

無名関数を利用すれば、関数内部に変数などのスコープを閉じ込めることができる。
いわゆるカプセル化を行うことができ、グローバル汚染などを抑止することができる。非常に一般的によく利用されている概念なので、最初は難しいかもしれないが感覚的に理解できるようになっておいたほうがよい。

```
(function(arg){
    var i, j, k;
    i = 2;
    j = 5;
    k = 10;
    console.log(arg * i * j * k);
})(10000);

console.log(i);
```

※ さて、このコードを実行すると二回あるログ出力でどんな結果が出力されるだろうか

無名関数の即時実行

先ほどの例のように、無名関数を即時実行するには括弧をうまく使って関数を定義する。

```
(function(){
    処理;
})();

// 一見すると括弧がいっぱいあって意味不明な感じもするが、以下を見てイメージするとわかりやすいかも

①
var myFunc = function(){処理;};
myFunc();

 ↓↓↓

②
(function(){処理;})();
// myFunc があるべきところにそのまま無名関数を置いて、丸括弧で囲んだだけ！
```

Chapter.5

prototype

prototype

prototype は、JavaScript において非常に重要な概念ではあるものの、若干難しく理解に苦労することも多い。

ここでは簡単に概要だけ説明するに留める。

JavaScript の仕組みとしての prototype の他に、同じ名前の「prototype.js」というフレームワークも存在する。

両者は全くの別物なので、検索して調べる場合など気をつけること。

prototype

JavaScript には、クラスという概念が言語仕様としては備わっていない。

ただ、ここで解説している prototype を用いると、似たようなことが実現できる。高度な JavaScript 実装を行うためには欠かせない概念だと言える。

一言で prototype を表現するなら「オブジェクトの規定値を定義する」と考えると良い。簡単な例を示す。

```
function myFunc(){}                          // 空っぽの関数の入れ物を定義
myFunc.prototype.prop1 = 'my prop 1'; // その関数の prototype にプロパティを生やす

var m = new myFunc();                        // 関数をインスタンス化する
console.log(m.prop1);                        // インスタンス化した直後にプロパティを参照してみると？
```

※ さてコンソールには何が出力されるでしょうか

prototype

prototype は、単にプロパティの値だけでなく関数でも同じことができ、いうなればこれはメソッドのように振る舞う。

規定値だけでなく、デフォルトで備えているメソッドを定義することもできるというわけ。

```
function myFunc(){}
myFunc.prototype.prop1 = 'my prop 1';
myFunc.prototype.func1 = function(){
    alert('my function 1');
}

var m = new myFunc();
m.func1();
```

※ func1 を実行すると何が起こる？

prototype

prototype の概念を理解するときにわかりにくいのは、入れ物が「関数」になっているところ。

ただ、Date や Array をインスタンス化するときのことを思い出すと、そのあたりも覚えやすい。

```
var d = new Date();
var a = new Array();
```

いずれも、最後に括弧がつくこと、そして new 演算子とともに呼び出すこと、このふたつの条件が合わさってインスタンス化が成されている。括弧が付いているということは、それはすなわち関数の呼び出しを行っているということなので、prototype の入れ物になるのもオブジェクトではなく関数だ ……と考えると覚えやすいかもしれない。

prototype を使ったほうがいい場面とは

JavaScript は先述の通りクラスという仕組みを持たない言語なので、オブジェクト指向っぽいことをやろうとすると、どうしても prototype などの仕組みが必要になる。

たとえば他の言語でも、クラスを使わなくてもプログラムは書けるが、拡張性や汎用性の高いものにしようと思えば、クラスを使ったりモジュール化したりといった工夫が必要になる。

prototype もこれと同じで、利用しなくても JavaScript 自体は処理を記述できる。しかし、汎用性を高めたり抽象化を行う場合には、prototype などの知識が必要になる。
いずれは、しっかりと理解して使いこなすことができるようになることを目指したほうがよいが、無理に使えばいいというものではないので、汎用的な処理などを手掛けるようになったら、自分で記述できるようになっていることが望ましい。

Chapter.6

Canvas2D

Canvas2D

HTML5 という最新の規格で、新しく HTML タグとして登場した Canvas というエレメントがある。

この Canvas はグラフィクスを描画するための領域として使われるタグで、JavaScript を利用して、Canvas タグの領域上に色を塗ったり、図形を描画したり、あるいは文字を描いたりといったことができるように設計されている。

ここからは Canvas の使い方について見ていくが、JavaScript に慣れるために Canvas2D を使うだけで、あまり難しいことはやらない。

Canvas2D コンテキスト

Canvas エレメントは、通常の DOM エレメントとして、属性を設定したり CSS でスタイルを当てたりといったことが普通にできる。

しかしその真価は、Canvas エレメントから取得できる「Canvas2D コンテキスト」にある。これは、Canvas 上に図形や文字、色付けなどを行うための命令が詰めこまれたオブジェクトである。

```
var canvas = document.getElementById('canvas');
var context = canvas.getContext('2d');
```

上記のように、Canvas エレメントの「getContext メソッド」を使うことでコンテキストを取得することができる。

Canvas2D コンテキスト

Canvas2D コンテキストは、多数のプロパティやメソッドを持つオブジェクトで、実装する際は、適切にプロパティに設定を行いつつ、適切にメソッドを呼び出してやる。

たとえば、最も簡単な例として Canvas を赤く塗りつぶすには次のようにする。

```
var canvas = document.getElementById('canvas');
var context = canvas.getContext('2d');

// まずプロパティに対して設定する
context.fillStyle = 'red';

// 次に描画のための命令などを呼び出す
context.fillRect(0, 0, canvas.width, canvas.height);
```

Canvas2D のポイント

- fill モードと stroke モードのふたつがある
- それぞれのモードごとに、CSS と同じように色が設定できる
- CSS に類似するプロパティが多くあるので CSS に対する理解が深いと Canvas2D コンテキストのプロパティの設定もしやすい

- 図形を描画する命令では、縦横の位置を表す XY と、幅と高さを表す WH が頻繁に登場する
- 引数が多いメソッドが多いので、落ち着いて考えるが吉

Canvas2D コンテキスト実践

Canvas2D コンテキストを使って、マウスに連動するインタラクティブな動作を実現させる。

Canvas 上にマウスカーソルを持って行くと、その部分に図形を描画するようなプログラム（つまりカーソルを置き換えているような感じ）を書く。

図形の形は、丸でも四角でもなんでもよい。

もし余裕があったら、ドラッグ開始と終了をイベントとして検知する仕組みを取り入れ、簡易的なペイントアプリケーションを作ってみるのもおもしろいかも。

Chapter.7

その他の HTML5 関連技術

HTML5 関連

勧告されたばかりの新しい HTML の規格が HTML5（2014年10月勧告）。

またそれに連動するようにして登場してきた、次世代の新しいウェブ関連技術を広く含めて、広義の HTML5 と呼ぶことがある。

Canvas は HTML5 の正式な仕様に含まれる概念だが、WebGL や CSS3 などの新しい技術も、広い意味で HTML5 の一部として扱われていた時期がある。

ここからは簡単に HTML5 関連について説明する。

Audio、Video

Canvas と同じように、タグとして HTML に埋め込むことができるオーディオとビデオのエレメント。

もちろん JavaScript からも制御が可能だが、Audio タグよりも WebAudio と呼ばれるよりオーディオの処理に特化した API があったりして、あんまり使われていない。

ただ、タグを埋め込むだけでいいので非常に簡単に使うことができる。

File API

ファイル API は、クライアント（ユーザーが操作しているデバイス）側にあるローカルファイルを、ブラウザから JavaScript によって操作するための API 。

FileAPI の登場により、JavaScript でアップロードするファイルを事前にチェックするなどの、ファイル関連の操作が行えるようになった。

JavaScript でローカルファイルを扱う場合には必須の API となる。

XMLHttpRequest

XMLHttpRequest 自体は HTML5 が提唱される前から存在したもの。

HTML5 では新しい規格に合わせて、バイナリ形式のファイルなどを扱うことができるように拡張された。

XMLHttpRequest は、サーバーと JavaScript が通信を行い、データをやり取りするためのインターフェースで、これを利用するとページ遷移をせずにリアルタイムにサーバーからのレスポンスを受けることができる。

Twitter などで、ページをリロードしていないのに次々に情報が流れてくるのはこういった技術を利用しているからである。

CSS3

CSS も、HTML5 の議論が白熱する中で次のステージへと向かった。それが CSS3 と呼ばれる新しい CSS の仕様。

CSS の書き方の基本的な部分は特に変わっていないが、新しい属性や文法が多数追加されている。主に、アニメーションなどのこれまで CSS では表現できなかった部分や、モバイル端末のようなデバイスの登場に対応するための、解像度に応じたスタイルの指定方法などが盛り込まれている。

最近では、ちょっとしたアニメーションなら JavaScript を用いるよりも CSS を用いたほうが効率が良いことが多い。

WebRTC

WebRTC は、ブラウザと JavaScript を用いて、リアルタイムにコミュニケーションを取るための便利な機能を定義する。

たとえば、マイクやカメラといったデバイスに装備された様々なデバイスに JavaScript からアクセスすることができる。

ウェブカメラからの入力を JavaScript で解析しビデオチャットする、といったことができる。

WebGL

HTML5 で Canvas や Video などの新しいタグが採用され、ブラウザ上での表現力は飛躍的に向上した。

WebGL はそのさらに先を行くもので、ブラウザ上でリアルタイムに 3DCG の描画を行うことができる API の規格。

OpenGL ES と呼ばれるモバイル機器向けの 3D API をブラウザから叩けるようにしたもので、ネイティブの OpenGL や DirectX といった、3D 描画用の API と同様の高い表現力が得られる。

TypedArray

JavaScript には、周知の通り変数の型がない。これにより、JavaScript で厳密な数値を扱った計算や、バイナリ単位でのデータ処理を行うのは非常に難しかった。

そこで生まれたのがこの TypedArray。

直訳して「型付き配列」と呼ばれることもある TypedArray は、先述の WebGL を扱う際や、XMLHttpRequest のバイナリデータを扱う際などに活躍する。