JavaScript によるイベントドリブン

アジェンダ

- JavaScript の言語について
- DOM について
- イベントについて
- jQuery について
- MVC アーキテクチャについて

まずはじめに

Just moment!

- Web で JS を使うとき、HTML の知識が前提となることが多い
  - http://www.kanzaki.com/docs/htminfo.html
  - 少なくとも「簡単なHTMLの説明」は押さえておきたい。

目的

講義時間は限られているので

- JS を学ぶ上でとっかかりをつかめること
  - リファレンスを提示します。概念を理解
- 言語的部分、DOM 及びイベントドリブンなプログラミング

覚えようとするとクソ多いのでリファレンスひける部分は覚えない

JS とはどんな言語であるか?

- 基本クライアントサイド=ブラウザで動く
  - 実装がたくさんある
- はてなのエンジニアはみんな誰もがある程度書けます
  - ウェブアプリを書く上で必須なため
- フロントエンドの重要性

先に JS のデバッグ方法

- デバッガ
  - Firefox なら Firebug、Google Chrome ならデベロッパー ツール
  - スクリプトを中断する場所 (ブレークポイント) を指定でき、そこから1行ずつ実行できる
  - スクリプト中に `debugger;` と書いておけばそこで中断する
- `console.log()`
  - ブロックせずにコンソール領域へ直接出力
  - 他言語の print デバッグに相当する
  - 最近のブラウザでは標準でサポートされつつある
- `document.title`
  - view-source:http://ma.la/files/shibuya.js/dec.html
  - `console.log()` が一般的でなかったころ
- `alert()`
  - 古典的だが今でも役に立つこともある
  - その時点で処理がブロックするので、ステップをひとつずつ確認するとき便利
- これらを仕込んで、リロードしまくることで開発します
  - エラーコンソールをどうやって開くかからはじめましょう

クロスブラウザについて

ここでクロスブラウザについて覚えても仕方ないので、Firefox と Firebug で開発することを前提にします。

JavaScript 言語について

言語的特徴

- 変数に型なし
  - プリミティブな型 + オブジェクティブな型
  - ただしプリミティブ型も自動的にオブジェクティブ型へ変換される
- 関数はデータとして扱える (オブジェクト)

- 文法は C に似てる
- 言語的な部分はECMAScript として標準化されている
    - ActionScript も同じ。AS やったことあるなら JS は DOM さえ覚えれば良い
- プロトタイプ指向なためクラスというものはない
    - クラス指向を模倣することはできる

ひとつずつ説明していきます

変数に型なし

Perl と一緒で、Java や C などと違う点

```
var foo = "";
foo = 1;
foo = {};
```

文字列を入れた変数にあとから数値を入れたりオブジェクトを入れられる

値自体には当然ちゃんと型があります

JSの型

- `undefined`
- `null`
- `number`
- `boolean`
- `string`
- `object`
    - `({})`
    - `[]`

`number`, `boolean`, `string` とかはプリミティブ値と呼ばれます

オブジェクト以外はプリミティブです。

JSの型 - `typeof`

`typeof` 演算子で調べることができます

```
typeof undefined;
typeof 0;
typeof true;
typeof {};
typeof [];
typeof null;
typeof "";
typeof new String("");
typeof alert;
```

それぞれどれになるでしょう?

`"undefined"`, `"number"`, `"boolean"`, `"string"`, `"object"`

JSの型 - `typeof`

```
typeof undefined //=> "undefined"
typeof 0;        //=> "number"
typeof true;     //=> "boolean"
typeof "";       //=> "string"
typeof {};       //=> "object"
typeof [];       //=> "object"
typeof null;     //=> "object"
typeof new String(""); //=> "object"
typeof alert;    //=>"object"
```

JS の型、object 型

- `Array` や `RegExp` など
- `string`, `number` をラップする `String`, `Number`
- `Function`

プリミティブ型からオブジェクト型へは自動変換がかかります (`null`, `undefined` 以外)

```
"foobar".toLowerCase();
```

としたとき、`"foobar"` は `string` プリミティブであるにも関わらず、メソッドを呼べるのはメソッドを呼ぼうとしたときに自動的に `String` オブジェクトへ変換されるからです。

```
new String("foobar").toLowerCase();
```

とはいえ大抵はオブジェクトへの変換を気にする必要はありません。

`object` 型

そもそもオブジェクト型って何かというと、名前と値のセット (プロパティ) を複数持ったものです。

連想配列とかハッシュとか言えばだいたい想像できると思います。

```
var obj = {}; // オブジェクトリテラル記法。new Object() と同じ
obj['foo'] = 'bar'; // foo という名前に 'bar' という値をセットしている。
obj.foo = 'bar'; // これも上と全く同じ意味
```

```
var obj = {
   foo : 'bar'
};
```

配列もJSではオブジェクトとして扱えます。

`undefined` と `null`

```
var foo;
typeof foo; //=> undefined
```

未定義値。

```
var foo = null
typeof foo; //=> object
```

何も入ってないことを示す

(`object` が入っていることを示したいが空にしときたいとか)

関数はデータ

JS では関数もデータになれ、`Function` オブジェクトのインスタンス扱いです。(第一級のオブジェクトというやつです)

変数に代入でき、引数として関数を渡すことが可能です。

```
var fun = function (msg) {
 alert(arguments.callee.bar + msg);
};
// object なので
fun.bar = 'foobar';

fun();
```

関数はデータ (2)

JS では関数もデータになれ、`Function` オブジェクトのインスタンス扱いです。(第一級のオブジェクトというやつです)

変数に代入でき、引数として関数を渡すことが可能です。

```
var fun = function (callback) {
 alert('1');
 callback();
 alert('3');
};
fun(function () {
 alert('2');
});
```

関数の定義

```
function foobar() {
}
```

と

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

はほぼ同じです (呼べるようになるタイミングが違います)

関数: `arguments`

```
function foobar() {
 alert(arguments.callee === foobar); //=> true
 alert(arguments.length); //=> 2
 alert(arguments[0]);     //=> 1
 alert(arguments[1]);     //=> 2
}
foobar(1, 2);
```

変数のスコープ

関数スコープです。`for` などのループでスコープを作らないことに注意

```
function (list) {
 for (var i = 0, len = list.length; i < len; i++) {
  var foo = list[i];
 }
}
```

は以下と同じ

```
function (list) {
 var i, len, foo;
 for (i = 0, len = list.length; i < len; i++) {
  foo = list[i];
 }
}
```

変数のスコープ：ハマりポイント

```
var foo = 1;
```

```
(function () {
 alert(foo); //=> undefined
 var foo = 2;
 alert(foo); //=> 2
})();
```

は以下と同じ

```
 var foo = 1;
(function () {
 var foo = undefined;
 alert(foo); //=> undefined
 foo = 2;
 alert(foo); //=> 2
})();
```

小ネタ: 配列的なオブジェクトを配列にする

```
 var arrayLike = {
  '0' : 'aaa',
  '1' : 'bbb',
  '2' : 'ccc',
  '3' : 'ddd',
};
arrayLike.length = 4;
var array = Array.prototype.slice.call(arrayLike, 0);
```

`Array` が持つ関数の多くはレシーバに `Array` コンストラクタを使って作られたことを要求しない

`Array` の基本的要件は

- `length` プロパティを持っていること
- 数字 (の文字列) を配列の要素のプロパティとして持っていること

`arguments` や `NodeList` なども `Array` ではない `Array`-like なもの

プロトタイプ指向

他のあるオブジェクトを元にして新規にオブジェクトをクローン (インスタンス化) していくオブジェクト指向技術

クラス指向はクラスからしかインスタンス化できないが、プロトタイプ指向ではあらゆるオブジェクトを基に新たなオブジェクトを生成できる

メリット

- 柔軟
- HTML のように個々の要素がほぼ同じだけど微妙に違う場合に便利

デメリット

- 自由すぎる

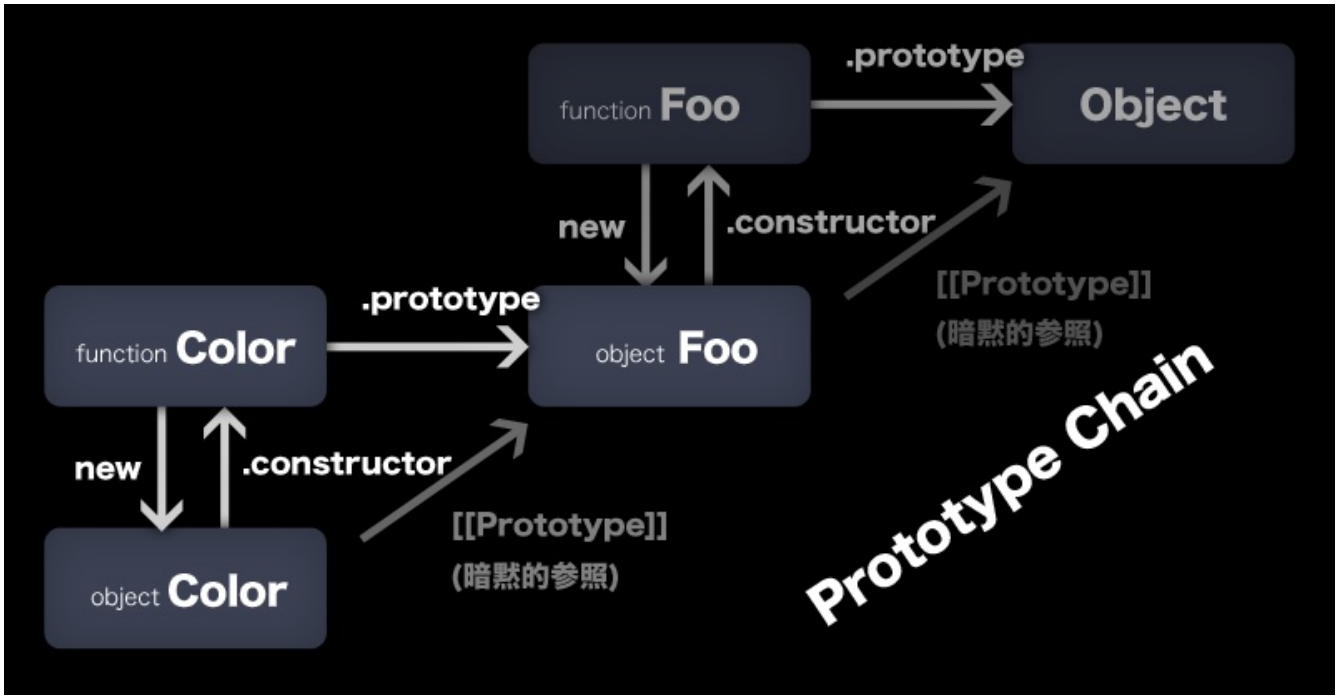
JS におけるプロトタイプ

プロトタイプにしたいオブジェクトに初期化関数を組み合せることでオブジェクトをクローンできる

```
 function Foo() { }
Foo.prototype = {
   foo : function () { alert('hello!') }
};

var instanceOfFoo = new Foo();
instanceOfFoo.foo(); //=> 'hello!'
```

JS におけるプロトタイプ (プロトタイプチェーン)




```
 function Foo() { }
Foo.prototype = {
   foo : function () { alert('hello!') }
};

function Bar() { }
Bar.prototype = new Foo();
Bar.prototype.bar = function () { this.foo() };

var instanceOfBar = new Bar();
instanceOfBar.bar(); //=> 'hello!'
```

- 新規に作られたオブジェクトは自身のプロトタイプへの暗黙的な参照を持っている
- プロトタイプにしたいオブジェクトに初期化関数を組み合せることでオブジェクトをクローンできる
- 暗黙的参照は既定オブジェクトである `Object` まで暗黙的な参照を持っている
    - => プロトタイプチェイン

プロトタイプチェーンは長くなると意味不明になりがちなので、あんまりやらないことが多い気がします。(継承やりすぎると意味不明なのがひどくなった感じです)

`this` について

- `this` という暗黙的に渡される引数のようなものがあります
- Perl の `$self` みたいなやつです
- 普通はレシーバーが渡されます
    - `foo.bar()` の `foo` のことをレシーバといいます
- 明示的に渡せすこともできる (`apply` , `call`)

```
 var a = { foo : function () { alert( a === this ) } };
a.foo(); //=> true
a.foo.call({}); //=> false
```

JS の使われかた

- HTML の
- `script` 要素で

```
 <!DOCTYPE html>
<title>JS test</title>
<script type="text/javascript" src="script.js"></script>
```

JS については以上です

質問など

- 変数に型なし
- 関数はデータとして扱える (オブジェクト)
- プロトタイプ指向
- 文法的に
- そもそも?

(あとで書いてみると疑問になることが多いと思うので聞いてください)

DOM について

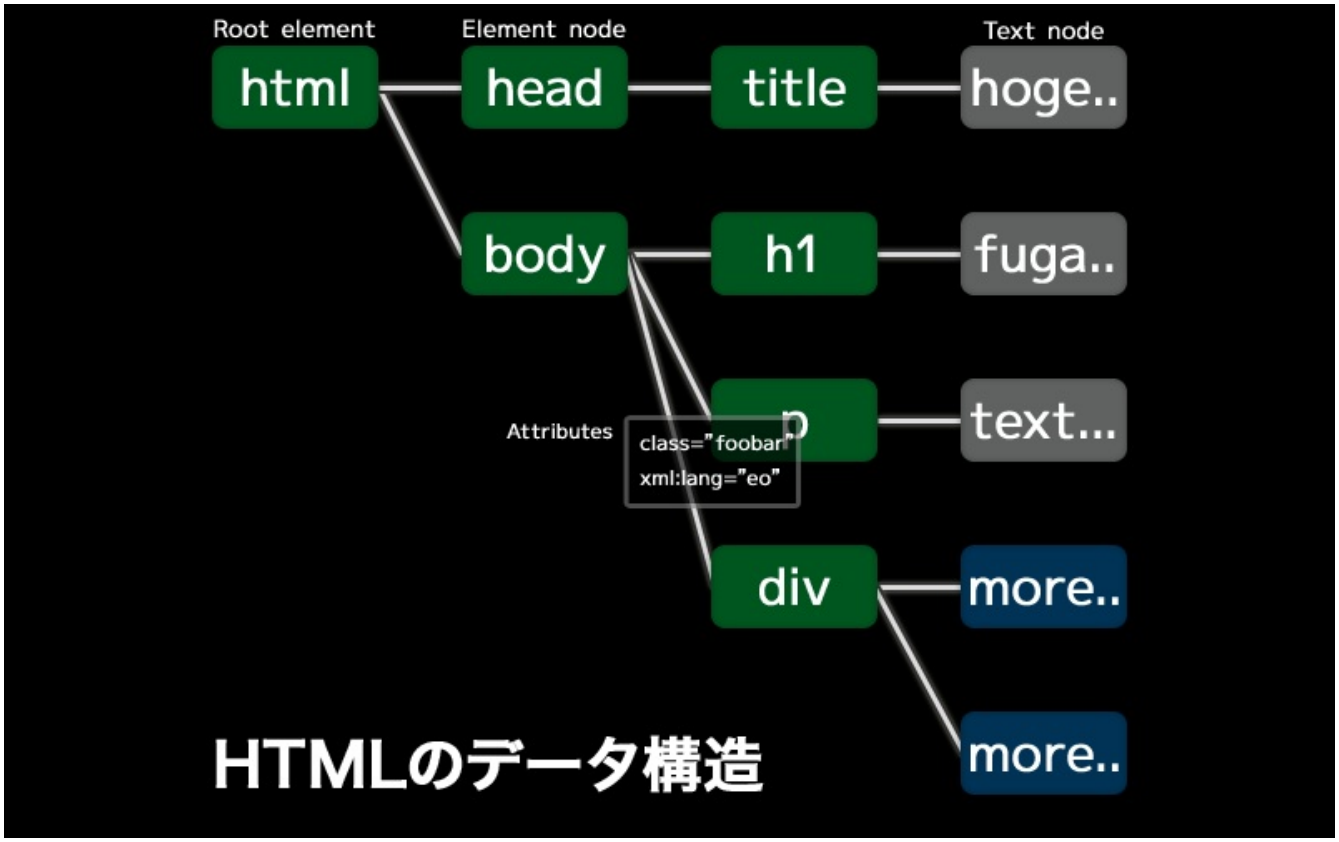
DOM とは

- Document Object Model の略です
- HTML とか CSS を扱うときの API を定めたものです
- ブラウザはDOM APIをJSで扱えるようにしています
- DOM にはレベルというものがあります
    - DOM Level 0 = 標準化されてなかったものの総称
    - DOM Level 1 = とても基本的な部分 (Element がどーとか)
    - DOM Level 2 = まともに使える DOM (Events とか)
    - DOM Level 3 = いろいろあるが実装されてない

今は高レベルの DOM といえそうなものでは、HTML5 として仕様が策定されていたりする

DOM Level 0 の多くも HTML5 で標準化されている

DOM の基本的な考えかた



一番上にはドキュメントノード (文書ノード)

DOM の構成要素

- `Node`
  - 全ての DOM の構成要素のベースクラス
- `Element`
  - HTML の要素を表現する
- `Attr`
  - HTML の属性を表現する
- `Text`
  - HTML の地のテキストを表現する
- `Document`
  - HTML のドキュメントを表現する
- `DocumentFragment`
  - 文書木に属さない木の根を表現する

`Text` も `Attribute` も `Node` のうちです。これらがツリー構造 (文書木) になっています。

よく使うメソッド

- `document.createElement('div')`
  - 要素ノードをつくる
- `document.createTextNode('text')`
  - テキストノードをつくる
- `element.appendChild(node)`
  - 要素に子ノードを追加する
- `element.removeChild(node)`
  - 要素の子ノードを削除する
- `element.getElementsByTagName('div')`
  - 指定した名前をもつ要素を列挙
- `document.getElementById('foo')`
  - 指定したIDをもつ要素を取得
- `node.cloneNode(true);`
  - 指定したノードを子孫ノード込みで複製
- https://developer.mozilla.org/en/DOM/element
- https://developer.mozilla.org/en/DOM/document

例えばテキストノードを要素に追加する場合

```
<div id="container"></div>
```

+

```
var elementNode = document.createElement('div');
var textNode    = document.createTextNode('foobar');
elementNode.appendChild(textNode);

var containerNode = document.getElementById('container');
containerNode.appendChild(elementNode);
```

↓

```
<div id="container"><div>foobar</div></div>
```

- ブラウザの画面に表示されるのは文書木に属するノード
- ノードを作った段階では、そのノードはまだ文書木に属していない

空白もノードです

```
<div id="sample">
<span>foobar</span>
</div>
```

```
alert(document.getElementById('sample').firstChild); //=> ?
```

続きはリファレンスで

言語的な部分は仕様を読むのが正確でてっとりばやい(和訳あるし)

- http://www2u.biglobe.ne.jp/~oz-07ams/prog/ecma262r3/

ちょい読んでみましょう

続きはリファレンスで (2)

DOM 的な部分は Mozilla Developer Center がよくまとまっている(部分的に和訳ある)

- https://developer.mozilla.org/En

DOM も仕様を読むのは参考になる (和訳あり)

- http://www2u.biglobe.ne.jp/~oz-07ams/prog/

イベント

並列性

- JS に並列性はない・組み込みのスレッドはない
- 同時に処理されるコードは常に1つ
  - 1つのコードが実行中だと他の処理は全て**止まる**ということ
- 多くのブラウザはループ中スクロールさえできない
  - Opera 10.60 では止まらないようになっている

非同期プログラミング

待たない/待てないプログラミングのこと

- 待てないのでコールバックを渡す
- 待てないのでイベントを設定する (方法としてはコールバック)
- 待たないので他の処理を実行できる

イベント

JS で重要なのは「イベント」の処理方法です。

JS では非同期プログラミングをしなければなりません。

イベントドリブン

- JS ではブラウザからのイベントをハンドリングします

メリット

- 同時に2つのコードが実行されないので同期とかがいりません
  - 変数代入で変に悩まなくてよい
- イベントが発火するまで JS レベルでは一切 CPU を食わない

デメリット

- 1つ1つの処理を最小限にしないと全部止まります
  - JS関係は全て止まります
  - ブラウザUIまで止まることが多いです
- コールバックを多用するので場合によっては読みにくい
  - あっちいったりこっちいったり

イベントの例

- `setTimeout(callback, time)`
  - 一定時間後にコールバックをよばせる
  - (非同期。DOM Events ではない)
- `element.addEventListener(eventName, callback, useCapture)`
  - あるイベントに対してコールバックを設定する
  - イベントはあらかじめ定められている

`setTimeout`

```
```

```
setTimeout(function () {
    alert('2');
}, 100);
alert('1');
```

https://developer.mozilla.org/ja/DOM/window.setTimeout

`addEventListener`

```
document.body.addEventListener('click', function (e) {
    alert('clicked!');
}, false);
```

https://developer.mozilla.org/ja/DOM/element.addEventListener

イベントの例

- `mousedown`
- `mousemove`
- `mouseup`
- `click`
- `dblclick`
- `load`

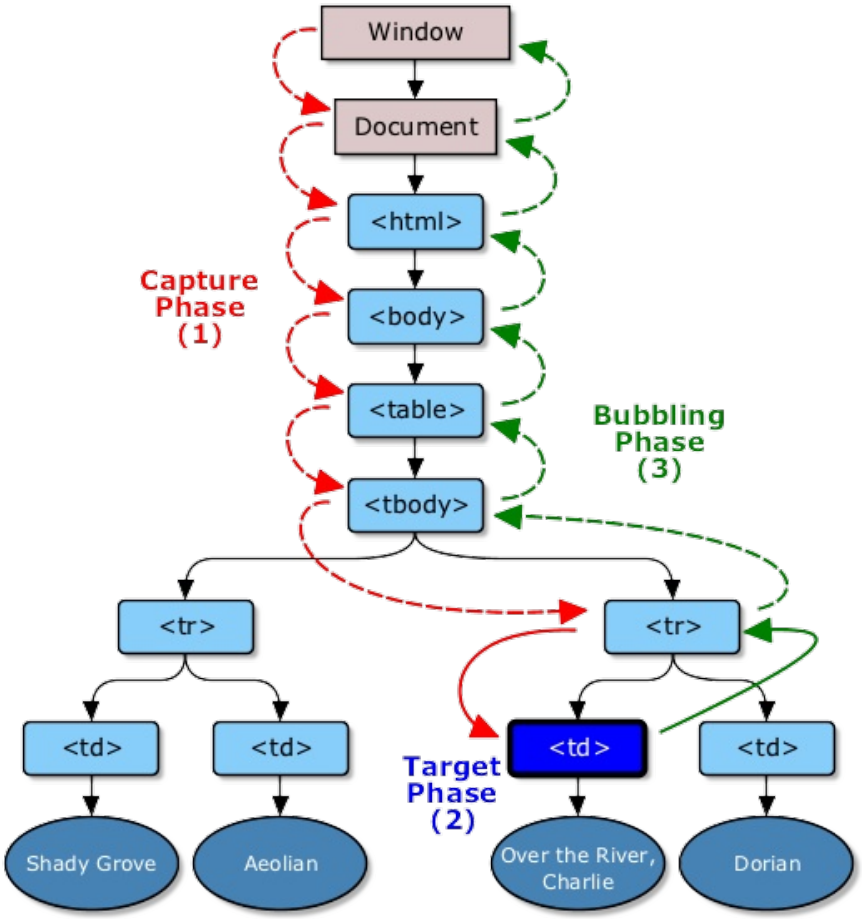
などなど。いっぱいあります。http://esw.w3.org/List*of*events

イベントバブリング

```
<p id="outer">Hello, <span id="inner">world</span>!</p>
```

- `inner` をクリックしたというのは、 `outer` をクリックしたということでもある
- イベントは実際に発生したノードから親に向かって浮上 (バブル) していく
- バブルしないイベントもある (`focus` 、 `load` 、etc.)

http://www.w3.org/TR/2011/WD-DOM-Level-3-Events-20110531/



`load` イベントについて

- DOM の構築
- 画像のロード

などが終わった直後に発生するイベント

基本的に `load` イベントが発生しないと、要素に触れません。

```
window.addEventListener('load', function (e) {
  alert('load');
}, false);
```

みたいに書くのが普通

イベントオブジェクトの構成要素

```
document.body.addEventListener('click', function (e) {
    alert(e.target);
}, false);
```

コールバックに渡されるオブジェクト

- `target` : イベントのターゲット (クリックされた要素)
- `clientX`, `clientY` : クリックされた場所の座標
- `stopPropagation()` : イベントの伝播 (含むバブリング) をとめる
- `preventDefault()` : イベントのデフォルトアクションをキャンセルする
  - デフォルトアクション : リンクのクリックイベントなら、「リンク先のページへ移動」

https://developer.mozilla.org/en/DOM/event をみるといいです

オブジェクトのメソッドをイベントハンドラとして使う

```
function Notifier(element, message) {
    this.message = message;

    var self = this;
    element.addEventListener('click', function (event) {
        self.notify();
    }, false);
}

Notifier.prototype.notify = function () {
    alert(this.message);
};

new Notifier(document.body, 'Clicked!');
```

* `addEventListener('click', this.notify, false)` では `notify` 中の `this` が何を指すかわからない * 最近のブラウザなら `this.notify.bind(this)` とも書ける

質問

- 非同期プログラミング
- 並列性なし
- イベントドリブン
- イベントオブジェクト

XMLHttpRequest

`XMLHttpRequest`

- 所謂 AJAX というやつのキモ
- JS から HTTP リクエストを出せる

生 **`XMLHttpRequest`** の使いかた

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/api/foo', true);
req.onreadystatechange = function (e) {
  if (xhr.readyState == 4) {
    if (xhr.status == 200) {
      alert(xhr.responseText);
    } else {
      alert('error');
    }
  }
};
req.send(null);
```

通常どんなJSフレームワークもラッパーが実装されてます

とはいえ一回は生で使ってみましょう

XMLHttpRequest での POST

POST するリクエスト bodyを自力で 作ります

```
var xhr = new XMLHttpRequest();
xhr.open('POST', '/api/foo', true);
req.onreadystatechange = function (e) { };
var params = { foo : 'bar', baz : 'Hello World' };
var data = ''
for (var name in params) if (params.hasOwnProperty(name)) {
```

```
  data += encodeURIComponent(name) + "=" + encodeURIComponent(params[name]) +
"&";
}
// data //=> 'foo=bar&baz=Hello%20World&'

req.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
req.send(data);
```

みたいなのが普通。multipartも送れるけどまず使わない

JSON をリモートから読みこむ

- JSON : オブジェクトのシリアライズ形式の一種。JS のオブジェクトリテラル表記と一部互換性がある 最近のブラウザなら `JSON` オブジェクトがありますが、古いブラウザにも対応するときは自分で `eval` します https://developer.mozilla.org/ja/Using*native*JSON

```
 var xhr = new XMLHttpRequest();
xhr.open('GET', '/api/status.json', true);
req.onreadystatechange = function (e) {
  if (xhr.readyState == 4) {
    if (xhr.status == 200) {
      var json = eval('(' + xhr.responseText + ')');
    } else {
      alert('error');
    }
  }
};
req.send(null);
```

質問

- `XMLHttpRequest`

ハマりポイント

- http 経由じゃないと XHR うまくうごかない

jQuery

jQuery

- 世界的によく使われているライブラリ
- はてなでも採用事例が増えてきている
- 書き方がちょっと独特

```
 // ページが読み込まれたときに
$(function ($) {
    // 文書中のすべての p 要素の背景色と文字色を変える
    $('p').css({ backgroundColor: '#ff0', color: '#000' });
});
```

jQuery を使う

```
 <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jque
ry/1.7/jquery.min.js"></script>

<!-- 開発中はこちらのほうがデバッグが楽かも？ -->
<script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquer
y/1.7/jquery.js"></script>
```

- http://docs.jquery.com/
- http://api.jquery.com/

jQuery の使い方

```
jQuery === $ //=> どちらも同じ jQuery 関数を指す

$(function ($) { ... });
  //=> 文書読み込み完了時に関数を実行

$('css selector');
  //=> CSS セレクタで要素を選択し、
  //   それらの要素が含まれる jQuery オブジェクトを作成

$('<p>HTML fragment</p>');
  //=> HTML 要素を内容込みで作成し、
  //   その要素が含まれる jQuery オブジェクトを作成
```

jQuery の使い方、イベント編

```
$('.foo').on('click', function (event) { ... });
  //=> foo クラスを持つ要素の click イベントを指定
  //=> イベントを登録する要素は実行時点で存在したもののみ

$('.foo').click(function (event) { ... });
  //=> 上と同じ

$(document).on('click', '.foo', function (event) { ... });
  //=> foo クラスを持つ要素の click イベントを指定
  //=> 実行時点で存在したかに関わらず、文書中のすべての
  //   foo クラスを持つ要素の click イベントを監視
```

jQuery の使い方、リクエスト編

```
$.get(url, { foo: 42 }).done(function (res) {
    alert(res);
});

$.post(url, { foo: 42 }).done(function (res) {
    alert(res);
});

$.ajax({ url: url, ... });
```

質問

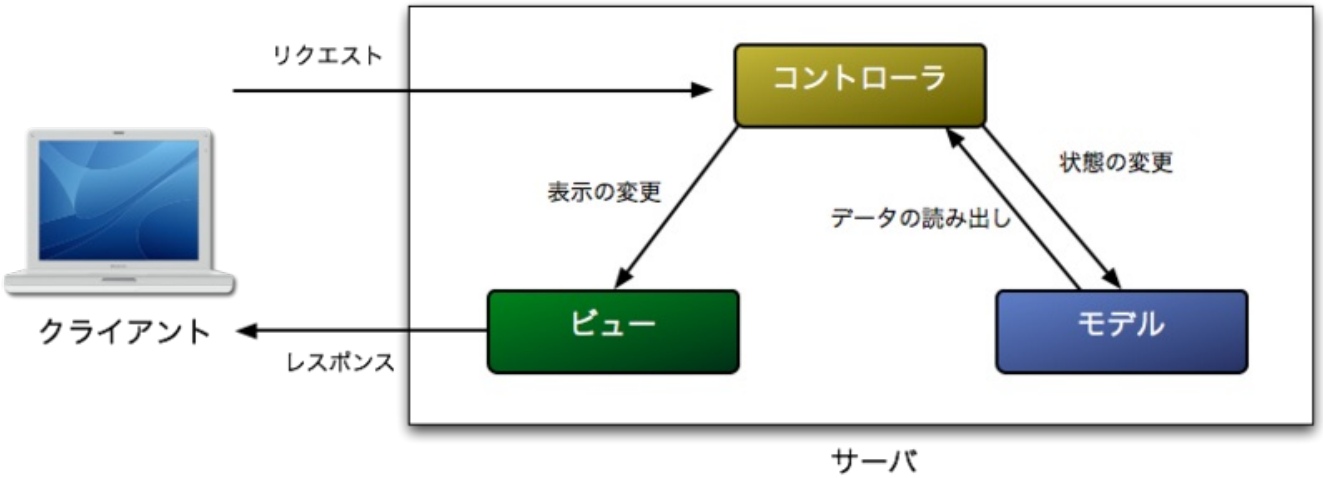
- jQuery

MVC アーキテクチャ

MVC アーキテクチャ

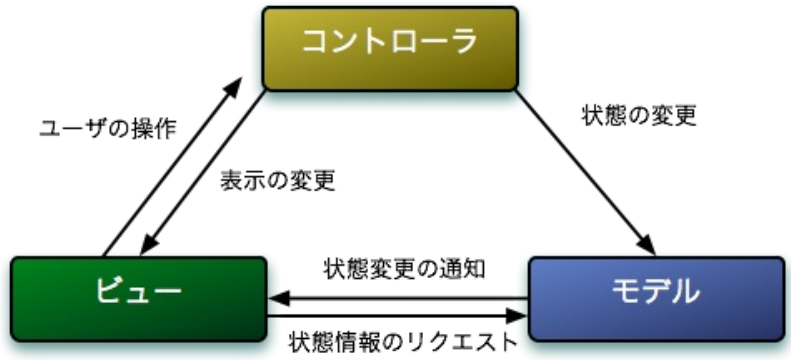
- Model-View-Controller
- これまでの課題でやった Web アプリケーションの MVC とはちょっと違う
    - 「Web アプリケーションの MVC」を「MVC2」と呼ぶこともある

Web アプリケーションの MVC



* Model は一方的に操作され、またはデータを読み取られるのみ

クライアントサイドプログラミングの MVC



* Model はしばしば observer パターンを実装し、自身の状態の変更を View に**通知する**

クライアントサイドプログラミング

- Web API (HTTP リクエストを送り、テキストや JSON で結果を受け取るような API) をそのまま Model として使うこともある
- 規模が小さい場合、View と Controller は一緒くたに実装することもある

例

- `/api/time.json` にアクセスすると `{ "time": "2012-04-01 12:00:00" }` というJSON 形式で現在時刻を返す API があるとする
- ボタンを押すと時刻表示を更新するようなウィジェットを作る

```
 <script>

var TimeWidget = function (button, output) {
    this.button = $(button);
    this.output = $(output);
    this.button.click(this.onClick.bind(this));
};

TimeWidget.prototype = {
    onClick: function (event) {
        $.get('/api/time.json').done(this.onGetTime.bind(this));
    },
    onGetTime: function (res) {
        this.output.text(res.time);
    }
```

```
};

$(function () {
    new TimeWidget($('#time-update-button'), $('#time-output'));
});

</script>

<p>
  <input id="time-update-button" type="button" value="表示を更新">
  <span id="time-output"></span>
</p>
```

Observer パターン

- jQuery を使えば大抵のオブジェクト上で observer パターンを実装できる

```
 <script>

// Model に相当
function Toggler() {
    this.enabled = true;
}

Toggler.prototype.toggle = function () {
    this.enabled = !this.enabled;
    // jQuery の機能を使って自身の状態が変化したことを通知する
    $(this).triggerHandler('update');
};

// View-Controller に相当
function ToggleDisplay(toggler, output, messages) {
    this.toggler = toggler;
    this.output = output;
    this.messages = messages;
    // jQuery の機能を使って Model の状態を監視する
    $(this.toggler).on('update', this.onUpdate.bind(this));
}

ToggleDisplay.prototype.onUpdate = function () {
    this.output.text(this.messages[this.toggler.enabled ? 0 : 1]);
};

$(function () {
    var toggler = new Toggler();
    new ToggleDisplay(toggler, $('#display-ja'), ['有効', '無効']);
    new ToggleDisplay(toggler, $('#display-en'), ['Enabled', 'Disabled']);
    setInterval(function () {
        toggler.toggle();
    }, 1000);
});

</script>

<p id="display-ja" lang="ja"></p>
<p id="display-en" lang="en"></p>
```

* 1 秒ごとに「有効」「無効」(または "Enabled", "Disabled") の表示が切り替わる

質問

- MVC アーキテクチャ

課題

課題1 (3.5点)

- ページ継ぎ足し機構を作れ (ダイアリー、グループにあるようなもの)
  - DOM を理解する
  - XHR を使える
  - イベントを理解する (`click` , `load`) jQuery、Ten といったフレームワークを使ってもよい

ヒント

手順

- 継ぎ足しを実行するトリガーとなる要素にイベントを設定 ++ 次のページの識別子をどうにかして取得
- サーバサイドに API を作る (`/api/page?id=...`) みたいな ++ ページの識別子を受け取ってそのページの内容を返す
- `XMLHttpRequest` で API を叩く
- 表示を更新する ++ `createElement` , `removeChild` , `appendChild` ...

スクリプトファイル置き場

- `static/js/diary.js` などに JS ファイルを設置すると、
- HTML からは `<script type="text/javsacript" src="/js/diary.js"></script>` でその JS ファイルを参照できます

配点

- 動くこと (1.5)
- 設計 (1)
- UIへの配慮 (1)

課題2 (3.5点)

タイマーを管理する `Timer` クラスをつくれ。

- コールバックを概念を理解する jQuery、Ten といったフレームワークを使っては**いけない**

仕様

```
 var timer = new Timer(time);
    //=> time ミリ秒のタイマーを作る
timer.addListener(callback1);
    //=> タイマーが完了したときに呼ばれる関数を追加する
    //=>  callback : Function => タイマーが完了したときに呼ばれる関数
timer.addListener(callback2);
    //=> タイマーが完了したときに呼ばれる関数は、複数指定できる
timer.start();
    //=> タイマーをスタートさせる。
    //=> start() してからコンストラクタに指定したミリ秒後に addListener に指定したコールバックが呼ばれる
timer.stop();
    //=> タイマーをストップさせる。
```

例

```
 var timer = new Timer(1000);
timer.addListener(function (e) {
  alert(e.realElapsed); //=> start() 時から実際に経過した時間
  timer.start(); //=> 再度スタートできる
});
timer.start();
```

```
document.body.addEventListener('click', function () {
  timer.stop(); //=> クリックで止まる。
}, false);
```

以上のようなインターフェイスの `Timer` クラスを作れ。(ライブラリを使わずに)

ヒント

- `Timer` の `addListener` は自分で実装しろということです (DOM のメソッドではない)
- `setTimeout()` を使うことになると思います

配点

- 動くこと (1.5)
- 設計 (1)
- `removeListener` を実装 (1)

課題3 (3点)

- その場編集機能を作れ
  - 要素の作成、追加、削除
  - こまめなフィードバックでストレスを感じさせないように
  - その場編集用 API の設計 jQuery、Ten といったフレームワークを使ってもよい

配点

- 動くこと (1)
- 設計 (1)
- UIへの配慮 (1)