LPI-Japan主催

HTML5プロフェッショナル認定試験 レベル2　ポイント解説無料セミナー

2015年7月25日

アシアル株式会社　生形可奈子

アシアルについて

Asial
Engineering the future

事業内容
HTML5アプリ開発環境（Monaca）、ネイティブアプリ開発、
サーバーサイド開発、インフラ、教育事業など

自己紹介

生形　可奈子（うぶかた・かなこ）

- 講師・Monacaエバンジェリスト
- 著書：「スラスラわかるJavaScript」（翔泳社）

目次

- HTML5プロフェッショナル認定試験とは
  - 概要
  - 試験範囲
- 頻出・難解ポイント解説
  - JavaScriptのデータ型
  - オブジェクト
  - 関数とスコープ
  - prototype
  - イベント

HTML5プロフェッショナル認定試験

HTML5プロフェッショナル認定試験とは

- 特定非営利活動法人LPI-Japanが実施する、HTML5および周辺技術の知識レベルを測る認定制度です。

- 試験の難易度を示す2種類のレベルがあり、段階的に受験します。
  - Level1

    マルチデバイスに対応した静的なWebコンテンツを HTML5を使ってデザイン、作成できるレベル

  - Level2

    システム間連携や最新のマルチメディア術に対応したWebアプリケーションや動的Webコンテンツの開発・設計ができるレベル

Level2の出題範囲 （重要度の高いもの）

出題範囲	重要度
JavaScript	
JavaScript文法	★★★★★★★★★★ 10
WebブラウザにおけるJavaScript API	
イベント	★★★★★★★★ 8
ドキュメントオブジェクト／DOM	★★★★★★ 6
ウィンドウオブジェクト	★★★★★★★★ 8
グラフィックス	
Canvas(2D)	★★★★★★ 6

受験について

- **試験方式はコンピュータベーストテスト（CBT）です。試験配信会社の「ピアソンVUE」を通して受験します。**

問題数	40～45問
試験時間	90分
合格ライン	約7割
回答方式	殆どが選択式（複数回答あり） 記述式も１問程度 コードリーディング問題が多い
受験料	¥15,000（税抜）

EcmaScript6

EcmaScript6

JavaScriptの標準仕様

- JavaScriptは、各ブラウザに独自に実装されており、それぞれ 独自仕様が含まれている場合がある
- ActionScriptやJScript等、JavaScript以外の言語も存在する
- 上記言語を標準化するために作られた仕様が「EcmaScript」
- 言語仕様のみを定義しており、HTML5 APIやDOMは含まれない

最新バージョンは6

- 2015年6月17日に承認

JavaScriptのデータ型

JavaScriptのデータ型

- 代表的なデータ型

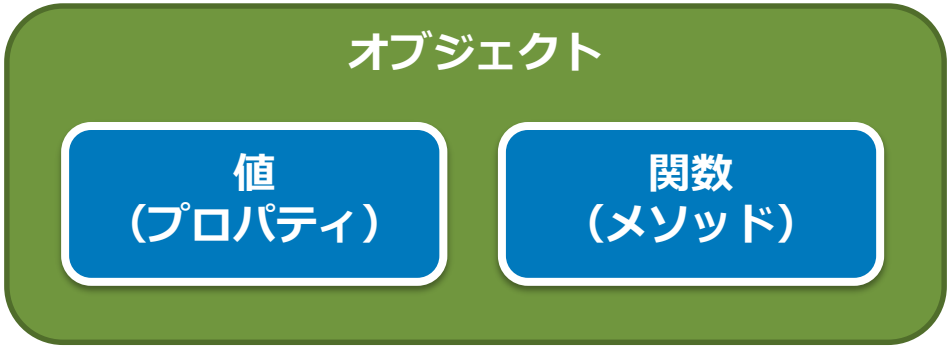
型の分類	代表的な型
プリミティブ型 （１つの値のみを持つ）	数値、文字列、論理値
オブジェクト型 （プロパティの集合体）	オブジェクト、関数、配列など
特殊な型	null（値が存在しない） undefined（値が定義されていない） Symbol（ユニークな値を持つ）※ES6

オブジェクトの基本

オブジェクトの基本

プロパティとメソッド

- オブジェクトの要素には、数値や文字列などの値だけではなく、関数も代入することができる
- 値のことを「プロパティ」、関数のことを「メソッド」という



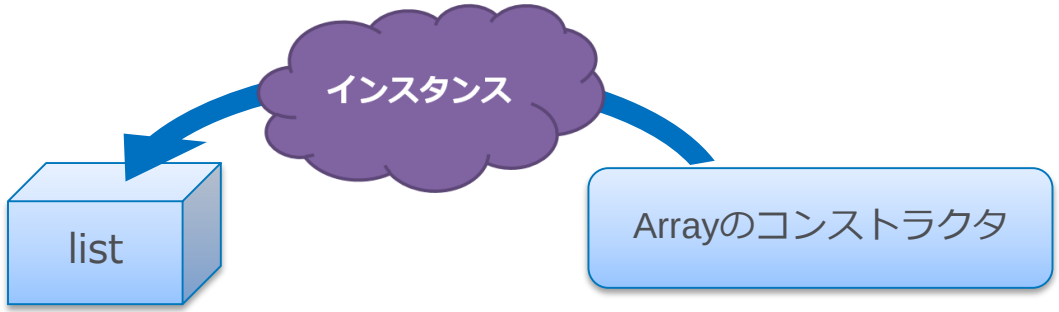
オブジェクト生成の仕組み

オブジェクト生成時の流れ

```
var list = new Array(1,2,3);
```

① オブジェクトの生成時に、コンストラクタという関数が実行される

② コンストラクタは、新しくオブジェクトを生成し、それを戻り値として返却する（このオブジェクトをインスタンスと呼ぶ）



オブジェクトの基本文法

オブジェクトの作成

- newを使用

```
var オブジェクト名 = new Object();
```

- オブジェクトリテラルを使用

```
var オブジェクト名 = {};
```

オブジェクトの例

- オブジェクトの生成とプロパティへのアクセス

```
var obj = {
    name: "hoge"
    show: function() {
        console.log(this.name);  // hoge
    }
};

// メソッドの呼び出し
obj.show();
```

プロパティの削除

- delete演算子により、定義済みのプロパティをオブジェクトから削除することができる

```
var obj = {
        name: "hoge"
};
delete obj.name;
```

問題：プロパティの削除

- 以下の実行結果は？

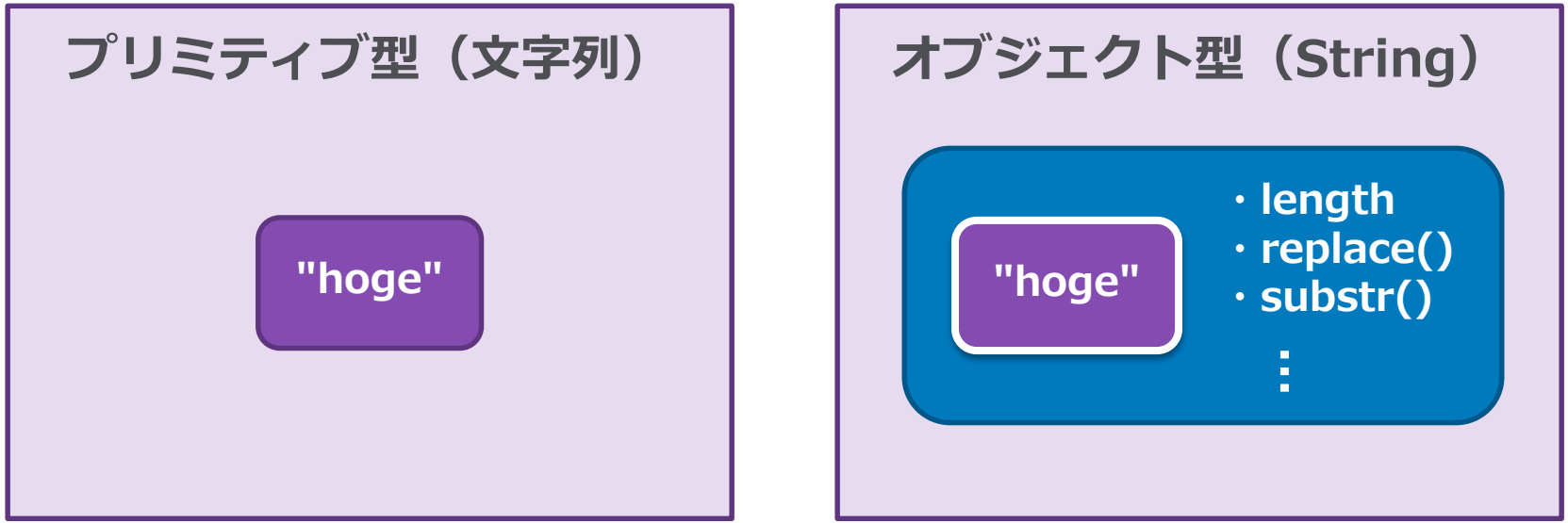
```
var obj = {
        name: "hoge"
};
delete obj.name;
console.log(obj.name);
```

- null
- undefined
- 空文字
- エラー

プリミティブ型とオブジェクト型

プリミティブ型とオブジェクト型

- プリミティブ型は１つの値だけ、オブジェクト型は複数のメソッドやプロパティを持つ



問題：プリミティブ型に対してプロパティを指定

- 以下の実行結果は？

```
console.log("hoge".length);
```

- 4
- null
- undefined
- エラー

ラッパーオブジェクト

- ラッパーオブジェクト … プリミティブ型に対応するオブジェクト

プリミティブ型	ラッパーオブジェクト
数値	Number
文字列	String
論理値	Boolean

```
"hoge".length
```

上記のように記述すると、プリミティブ型の文字列がラッパーオブジェクトであるString型に暗黙的に変換される

問題：プリミティブ型とラッパーオブジェクトの比較

- 以下の実行結果は？（true / false）

```
var a = "hoge";
var b = new String("hoge");
console.log(a == b);
```

- 以下の実行結果は？（true / false）

```
var a = "hoge";
var b = new String("hoge");
console.log(a === b);
```

問題：値の比較、参照の比較

- 以下の実行結果は？（true / false）

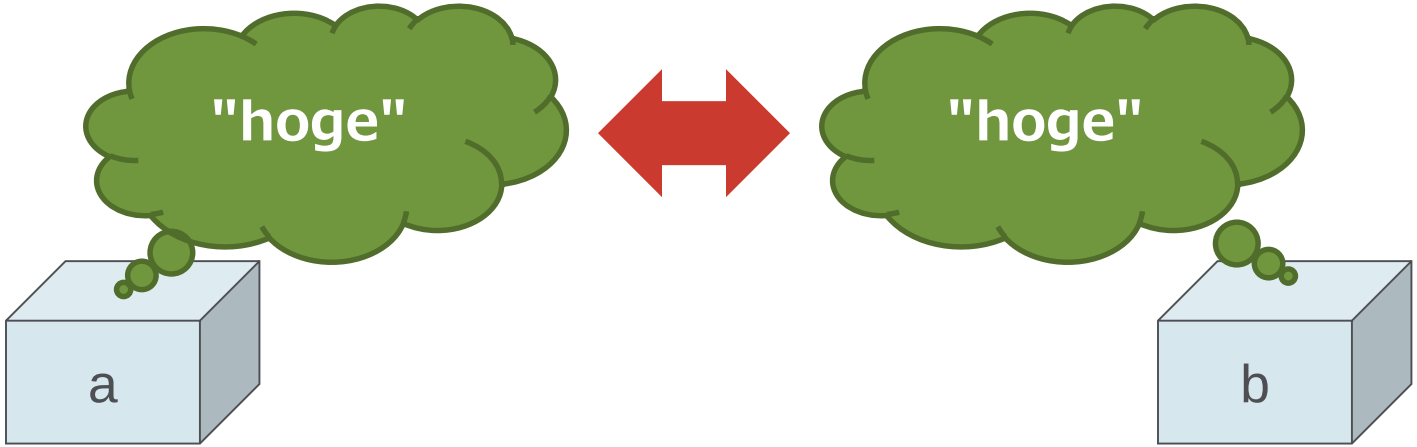
```
var a = "hoge";
var b = "hoge";
console.log(a == b);
```

- 以下の実行結果は？（true / false）

```
var a = new String("hoge");
var b = new String("hoge");
console.log(a == b);
```

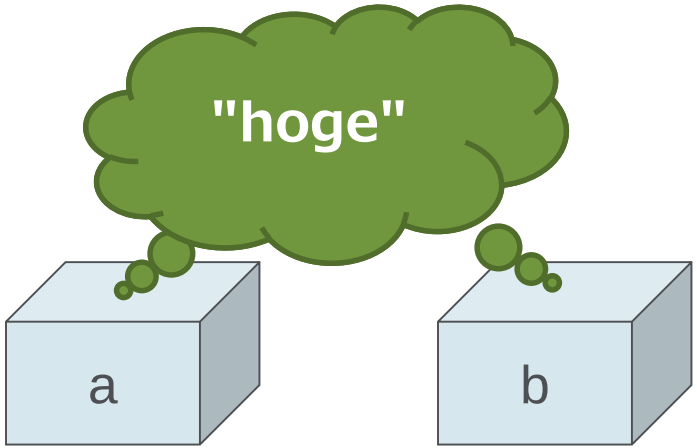
値の比較、参照の比較

- プリミティブ型は値そのものを比較する
- オブジェクト型は参照しているオブジェクトが同じものかどうかを比較する



同じオブジェクトを参照している例

```
var a = new String("hoge");
var b = a;
console.log(a == b);
```



この場合は同じオブジェクトを
参照しているのでtrue

特殊数値

JavaScriptの特殊な値

特殊数値

種類	代表的な型
Number.MAX_VALUE Number.MIN_VALUE	正の最大値 正の最小値（負の最大値ではない）
Infinity	無限大（ゼロ除算、オーバーフロー）
NaN	数値ではない値（計算失敗など）

問題：isNaN関数

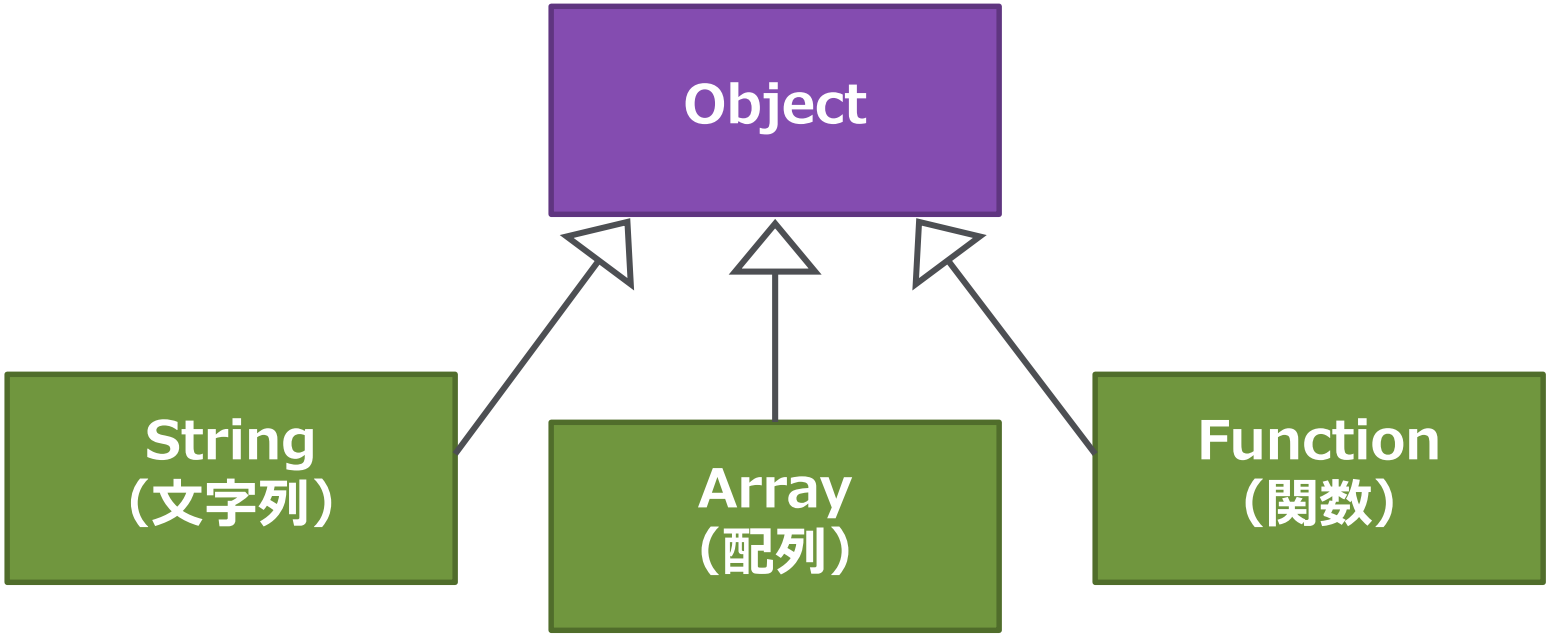
- isNaN関数は、数値として解釈できない値の場合にtrueを返す。以下のうち、trueを返すのはどれか？

  - isNaN(null)
  - isNaN(undefined)
  - isNaN("10")
  - isNaN(true)
  - isNaN(Infinity)
  - isNaN(Number.MAX_VALUE)

様々なオブジェクト

オブジェクトの種類

- Objectオブジェクトはすべてのオブジェクトのベースとなっている



typeof演算子

typeof演算子

- データの型を調べる演算子なので、StringやArrayなどのオブジェクトはほぼすべて「object」と判定される

  ※ 関数は「function」と判定される

```
var list = new Array(1,2,3);
console.log(typeof list);  // object
```

instanceof演算子

instanceof演算子

- どのコンストラクタによって生成されたインスタンスなのかを判定するので、オブジェクトの種類を調べることができる

```
var list = new Array(1,2,3);
console.log(list instanceof Array);  // true
```

配列

配列の基本文法

配列の作成

- newを使用

```
var 配列名 = new Array(1, 2, 3);
```

- 配列リテラルを使用

```
var 配列名 = [1, 2, 3];
```

問題：配列のインデックス

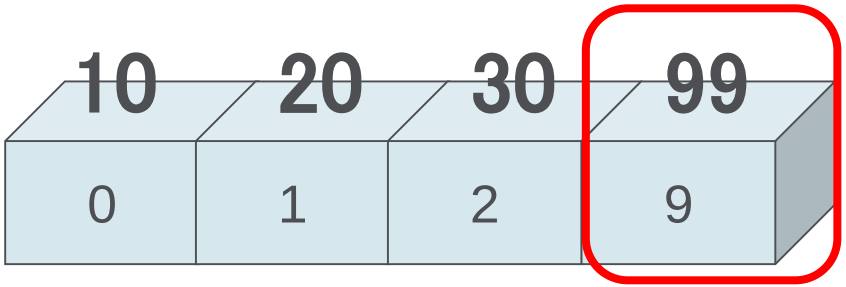
- 以下の実行結果は？

```
var array = [10, 20, 30];
array[9] = 99;
console.log(array.length);
```

- 3
- 4
- 10
- undefined

配列のインデックス

- インデックスは連番でなくても割り振ることができる



- lengthプロパティは最大インデックス+1を返す

関数

関数の定義いろいろ①

- function命令文

```
func("hello");
function func(message) {
        alert(message);
}
```

- 関数にfuncという名前をつけている
- 関数は静的に作成される（スクリプト実行前に評価される）

関数の定義いろいろ②

関数リテラル

```
var func = function(message) {
        alert(message);
};
func("hello");
```

- funcという変数に関数を代入している
- この場合、関数名は省略できる（無名関数や匿名関数と呼ばれる）

関数の定義いろいろ③

即時関数

```
(function(message) {
        alert(message);
})("hello");
```

- (関数の定義)(引数); とすることで定義と呼び出しを同時に行う
- 主にスコープを限定する用途などで利用される

EcmaScript6から追加される機能

- デフォルト引数
- 可変長引数
- アロー記法

デフォルト引数

■デフォルト引数

```
function f(x, y=20){
    console.log(x);  // 10
    console.log(y);  // 20
}

f(10);
```

- 引数に値が渡されなかった場合に代入する値を指定することができる

可変長引数

可変長引数

```
function f(...args) {
      for(var i=0; i<args.length; i++) {
            console.log(args[i]);
      }
}

f(10, 20);
f(10, 20, 30, 40);
```

- 引数の数を動的に変更することができる

アロー記法

従来の記法

```
var f = function(x) { return x * 2; };
```

アロー記法

```
var f = x => x * 2;
```

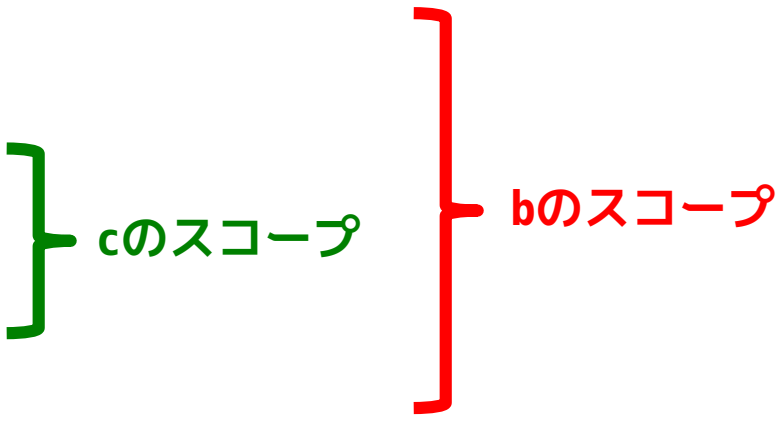
- functionキーワードを省略できる
- 引数が１つの場合は、()を省略できる
- 処理がreturn文のみの場合は、{}とreturnキーワードを省略できる

変数のスコープ

スコープとは

- 変数はスコープ（有効範囲）内でのみ参照できる

```
var a = 10;  // aはグローバル変数なのでどこからでも参照可
function func1() {
    var b = 20;
    function func2() {
        var c = 30;
    }
}
```



- スコープは関数内でのみ作られる

スコープとは

- スコープ外で変数を参照した場合、ReferenceErrorが発生する

```
var a = 10;
function func1() {
    var b = 20;
    function func2() {
        var c = 30;
    }
}
console.log(a);  // 10
console.log(b);  // ReferenceError
```

スコープとは

- varをつけていない場合、その変数はグローバルスコープとなる

```
var a = 10;
function func1() {
    b = 20;
    function func2() {
        var c = 30;
    }
}
console.log(a);  // 10
console.log(b);  // ReferenceError（？）
```

varを付け忘れているので、
aと同じグローバル変数になる

スコープとは

- varをつけていない場合、その変数はグローバルスコープとなる

```
var a = 10;
function func1() {
    b = 20;
    function func2() {
        var c = 30;
    }
}
func1();  // 関数の実行
console.log(b);  // 20
```

func1関数が実行されると、
bがグローバル変数として生成される

use strict

- use strict（厳格モード）を宣言すると、varの付け忘れなどの、推奨されない記述に対して警告が表示される

```
function func() {
    "use strict";
    b = 20;  // エラー
}
```

  - 厳格モードはスクリプト全体に適用することもできるが、関数内でのみの適用が推奨される

問題：JavaScriptのスコープ①

- 以下の実行結果は？

```
for(var i = 0; i < 3; i++) {
    var a = "hoge";
}
console.log(a);
```

- エラー
- undefined
- null
- "hoge"

JavaScriptのスコープ①

- JavaScriptにはブロックスコープがない

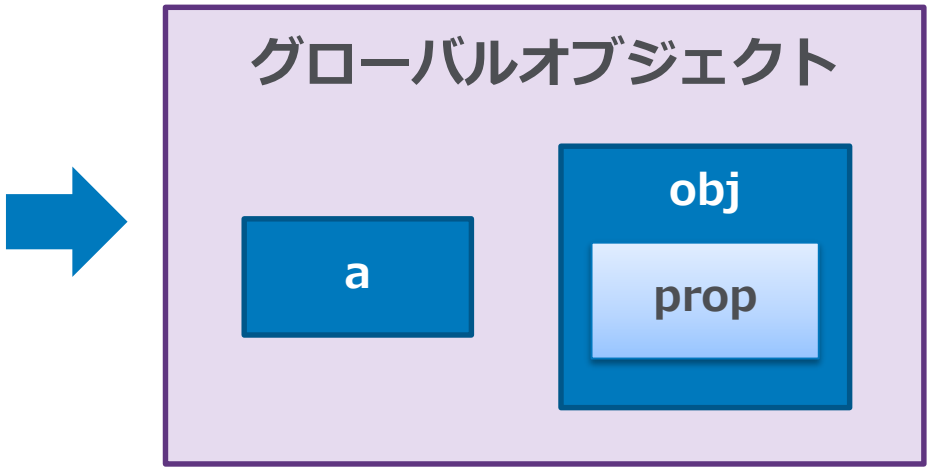
```
if(true) {
    let a = "hoge";
}

console.log(a);  // a is not defined
```

- スコープが発生するのは関数内のみ
- ただし、ES6のletキーワードを使えばブロックスコープが利用できる

グローバル変数とは何か？

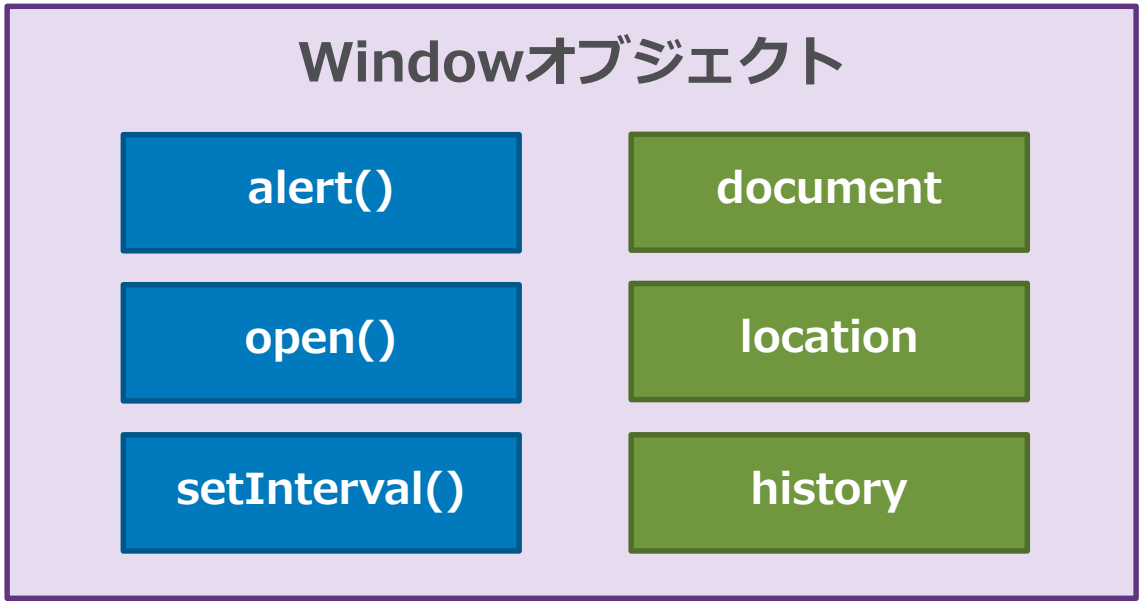
- スクリプト内の値やオブジェクトは全て、暗黙的に一番外側に存在しているグローバルオブジェクトに含まれている

```
<script>
    var a = 10;
    var obj = {
        prop: "hoge"
    };
</script>
```



グローバルオブジェクト

- ブラウザ環境におけるグローバルオブジェクトは、「Windowオブジェクト」である



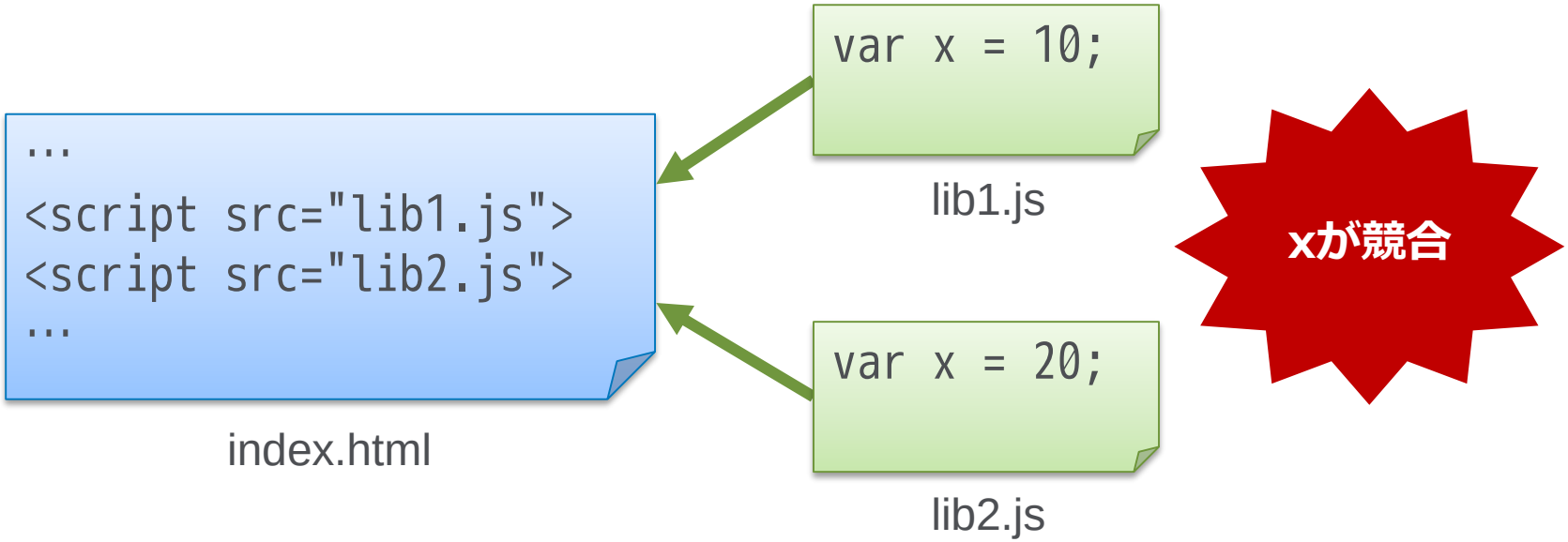
グローバル変数の正体

- グローバル変数＝グローバルオブジェクトのプロパティ

```
<script>
var a = 10;

console.log(a);          // 10
console.log(window.a);   // 10
</script>
```

グローバル変数利用の注意

- 複数のJSファイルを読み込んでいる場合に、グローバル変数が競合してしまう場合があるので利用は必要最小限に



問題：JavaScriptのスコープ②

- 以下の実行結果は？

```
var a = "global";
function func() {
    console.log(a);
    var a = "func";
}
func();
```

- エラー
- undefined
- "global"
- "func"

JavaScriptのスコープ②

ホイスティング（変数の巻き上げ）とは

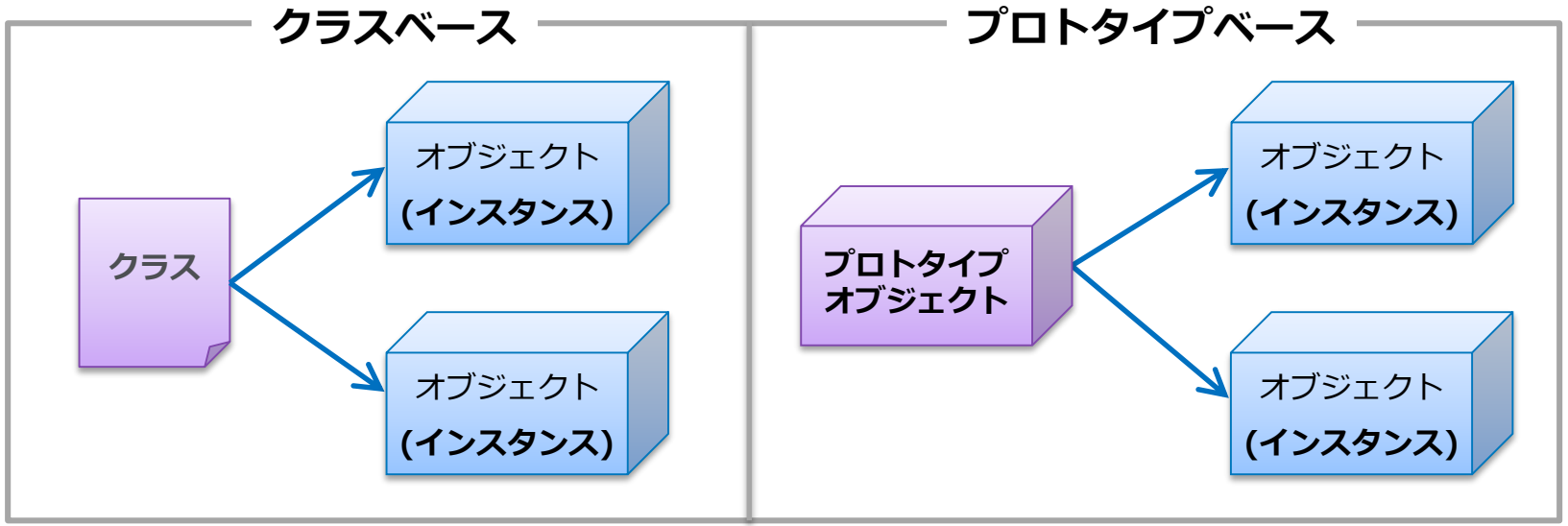
- 関数の途中で宣言された変数は、関数の先頭に巻き上げて宣言される
- ただし、初期値として代入されている値は巻き上げない
- 先ほどのコードは以下のように解釈される

```
var a = "global";
function func() {
    var a;
    console.log(a);
    a = "func";
}
func();
```

prototype

プロトタイプベースオブジェクト指向

- JavaScriptはプロトタイプベースのオブジェクト指向言語
  - あるオブジェクトを基礎として、新しいオブジェクトを生成する仕組み
  - 新しく作られたオブジェクトは、「インスタンス」と呼ばれる



プロトタイプベース指向に則ったオブジェクト定義

- プロトタイプオブジェクト＝関数
  - 基礎となるオブジェクト（クラスベース言語におけるクラスの役割）は、関数オブジェクトである

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

  - 上記のオブジェクトを元にインスタンスを生成するには、new演算子を使う

```
var dog1 = new Dog();
```

プロトタイプベース指向に則ったオブジェクト定義

コンストラクタ＝関数の処理

- 関数内の処理はインスタンスを生成（new）した時に、コンストラクタとして実行される

```
var Dog = function(_name) {
    this.name = _name;
    this.show = function() {
        alert(this.name);
    };
};

var dog1 = new Dog("ポチ");  // コンストラクタを実行
```

プロトタイプベース指向に則ったオブジェクト定義

- インスタンス＝Objectオブジェクト
  - プロトタイプオブジェクトはFunction型だが、インスタンスはObject型となる

```
var Dog = function(_name) {
    // 省略
};
var dog1 = new Dog("ポチ");

console.log(typeof Dog);   // function
console.log(typeof dog1);  // object
```

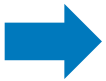
プロトタイプベース指向に則ったオブジェクト定義

コンストラクタの暗黙的処理

- インスタンスがObject型になるのは、コンストラクタが暗黙的に新しいObjectを生成しているためである。

ソースコード

```
var Dog = function(_name) {
  this.name = _name;
  this.show = function() {
    alert(this.name);
  };
};
```

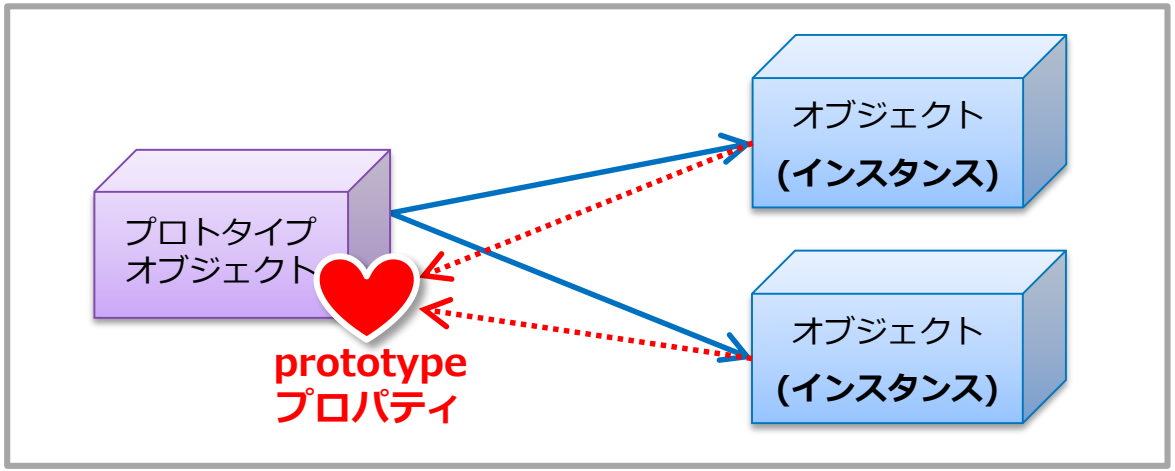


暗黙的処理

```
var Dog = function(_name) {
  var this = {};
  this.name = _name;
  this.show = function() {
    alert(this.name);
  };
  return this;
};
```

prototypeプロパティとは

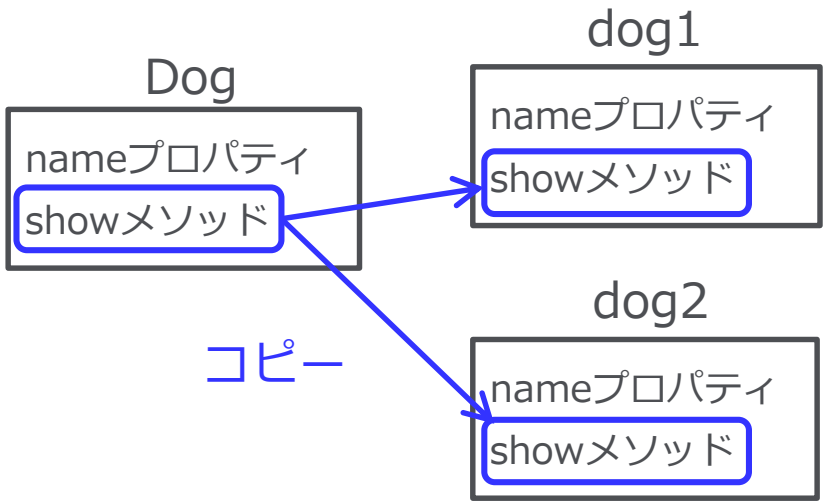
- オブジェクトのコア
  - プロトタイプオブジェクトが持つプロパティ
  - 各インスタンスは、プロトタイプオブジェクトのprototypeを参照する
  - prototypeプロパティには、各インスタンスで共通の機能を格納する



共通化されていない例

- オブジェクトをインスタンス化すると、元のオブジェクトで定義されたプロパティとメソッドが各インスタンスにコピーされる

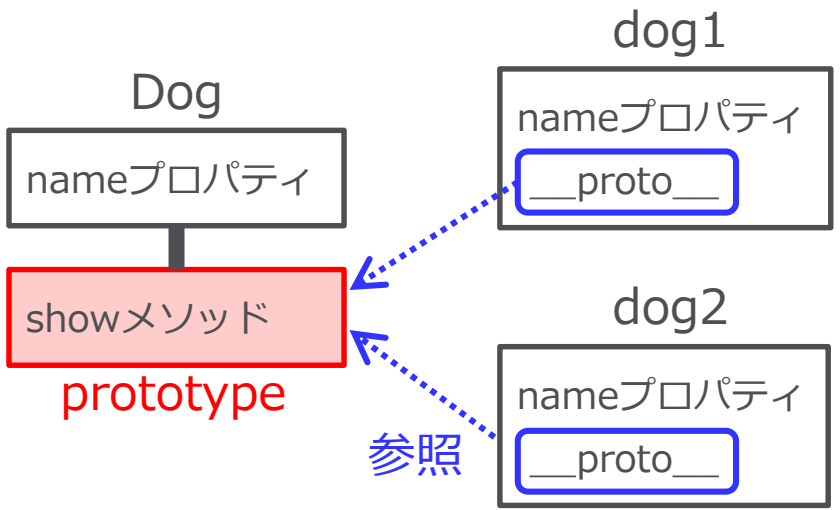
```
var Dog = function(_name) {
    this.name = _name;
    this.show = function() {
        alert(this.name);
    };
};
var dog1 = new Dog("ポチ");
var dog2 = new Dog("シロ");
```



共通化されている例

- 各インスタンスが内部的に持つ__proto__プロパティから元のオブジェクトのprototypeプロパティを参照する

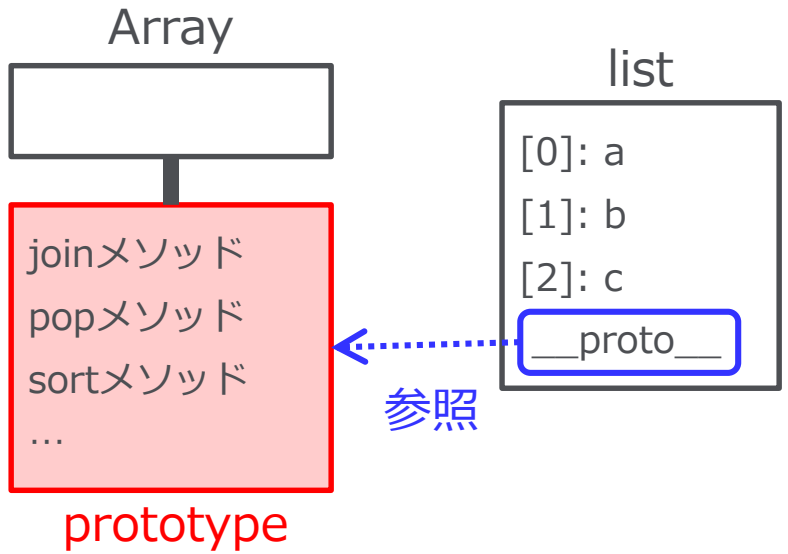
```
var Dog = function(_name) {
    this.name = _name;
};
Dog.prototype.show =
function() {
    alert(this.name);
};
var dog1 = new Dog("ポチ");
var dog2 = new Dog("シロ");
```



組み込みオブジェクトの例

- 以下は、Arrayのインスタンスからprototypeに定義されているjoinメソッドを呼び出している例

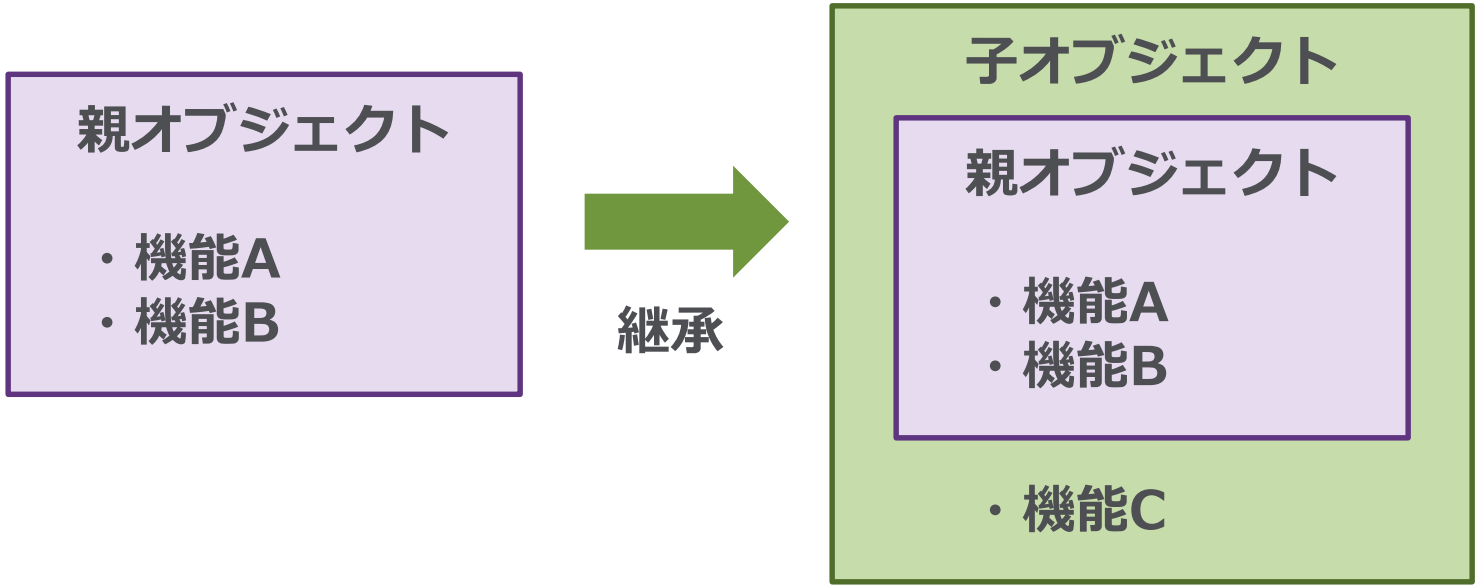
```
var list =
  new Array('a', 'b', 'c');
var str = list.join('+');
console.log(str);  //a+b+c
```



継承

継承

- 継承とは
  - あるオブジェクトを拡張して、別のオブジェクトを作る仕組み



JavaScriptにおける継承の実現

最も単純な継承関係

- 子のprototypeプロパティに、親のインスタンスを代入する

```
// 親
var Parent = function() {
    this.x = 10;
};
Parent.prototype.show =
function() {
  console.log(this.x);
};
```

```
// 子
var Child = function() {
    this.y = 20;
};

// 継承
Child.prototype = new Parent();

// インスタンス化
var child = new Child();
```

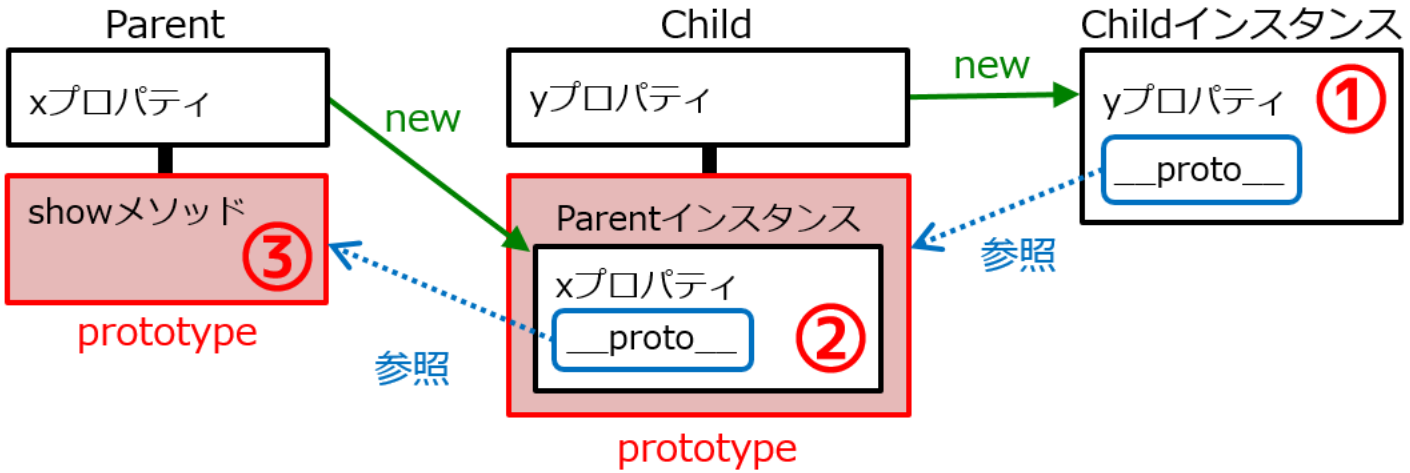
プロトタイプチェーン

- **プロパティを探索する順序**

  ①自身のインスタンス内に該当プロパティがあるかどうかを調べる

  ②プロトタイプオブジェクトのprototype内を調べる

  ③親オブジェクトのprototype内を調べる



ES6：class構文

- class構文により、クラスベース言語のように扱うことができる

```
class Parent {
    show(text) {
      console.log(text);
    }
};

class Child extends Parent {
    show() {
      super.show("child");
    }
};
```

メソッドはprototype内に定義される

キーワード１つで継承関係を実現

親オブジェクトの参照も容易

this

this

- thisとは、所属しているオブジェクトを指すキーワード

```
var obj1 = {
        name: "hoge",
        func: function() {
                console.log(this.name);    // hoge
        }
};
obj1.func();
```

this = obj1

問題：オブジェクトの外側に記述されたthisは？

- 以下のthisは何を指す？

```
<script>
    console.log(this);
</script>
```

- null
- undefined
- Functionオブジェクト
- Windowオブジェクト

thisのパターン

1. 関数呼び出し
2. メソッド呼び出し
3. コールバック関数内のthis

1. 関数呼び出し

- 関数を呼び出した場合はWindowオブジェクトになる

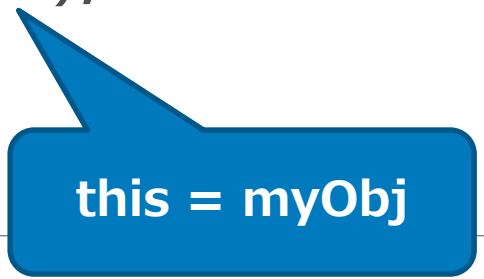
```
function func() {
    console.log(this);
}
func();
```

this = Windowオブジェクト

2. メソッド呼び出し

- オブジェクトのメソッドを呼び出した場合は自分自身のオブジェクトになる

```
var myObj = {
  show: function() {
    console.log(this);
  }
};
myObj.show();
```



3. コールバック関数内のthis

- 以下のようにメソッドをコールバック関数として渡した場合は、
  thisはコールバック関数を実行するオブジェクトに依存する

```
var myObj = {
  show: function() {
    console.log(this);
  }
};
window.setInterval(myObj.show, 1000);
```

this = Windowオブジェクト

setIntervalのコールバック関数

3. コールバック関数内のthis

- 以下のthisは、イベント発生元のHTML要素を指す

```
<button id="btn">ボタン</button>
<script>
    function func() {
        console.log( this );
    }
    var btn = document.getElementById("btn");
    btn.addEventListener("click", func);
</script>
```

clickイベントのコールバック関数

thisが指すオブジェクトの見分け方

- thisが指すオブジェクトは、文脈によって異なる

「どこに記述されているか」ではなく、
「誰が実行しているか」がポイント

call/applyメソッド

- call/applyによって、thisが指すオブジェクトを変更できる

```
var obj1 = {
        name: "わたし",
        show: function() {
                console.log(this.name);    // ぼく
        }
};
var obj2 = {
        name: "ぼく"
};
obj1.show.call(obj2);  // obj2をthisとして実行
```

callとapplyの違い

- call()はカンマ区切り、apply()は配列でメソッドの引数を渡す

```
var obj1 = {
    name: "わたし",
    showName: function(msg1, msg2) {
                console.log(this.name + msg1 + msg2);      }
};
var obj2 = {
    name: "ぼく"
};
obj1.showName.call(obj2, "こんにちは", "よろしくね");
obj1.showName.apply(obj2, ["こんにちは", "よろしくね"]);
```

call/applyの活用

- 配列のような構造だがArrayオブジェクトではないオブジェクトから、Arrayオブジェクトが持つjoinメソッドを利用する例

```
var obj = {
    "0": "aaa",
    "1": "bbb",
    "2": "ccc",
    length: 3
};

// Arrayオブジェクトのjoinメソッドを借りる
var str = Array.prototype.join.call(obj, "+");
```

イベント

キー入力イベント

イベント	発生タイミング
input	文字が入力されたとき
keydown	キーが押されたとき
keypress	キーが押されたのち、離れたとき
keyup	キーが離れたとき

マウスイベント

イベント	発生タイミング
mousedown	ポインティングデバイスのボタンを押したとき
mouseup	ポインティングデバイスのボタンを離したとき
mousemove	カーソルが移動したとき
mouseover	カーソルが要素の上に乗ったとき
mouseout	カーソルが要素から外れたとき
mousewheel	マウスホイールを回転させたとき
scroll	ページがスクロールしたとき

ドラッグ&ドロップイベント

イベント	発生タイミング
drag	ドラッグしている間
dragstart	ドラッグが開始されたとき
dragend	ドラッグが終了したとき
dragenter	ドラッグ中の要素がドロップ可能領域に入ったとき
dragleave	ドラッグ中の要素がドロップ可能領域から外れたとき
dragover	ドラッグ中の要素がドロップ可能領域上にいるとき
drop	ドロップしたとき

タッチイベント（スマートデバイス用）

イベント	発生タイミング
touchstart	画面に指が触れたとき
touchmove	画面上で指を動かしている間
touchend	画面から指が離れたとき

バリデーション属性（HTML5で追加）

- <input>要素に付与すると、 submit時にチェックを行ってくれる
  - required　必須
  - pattern　正規表現
  - min　最小値
  - max　最大値
  - maxlength　最長文字数

```
名前：<input type="text" required>
郵便番号：<input type="text" pattern="^[0-9]{3}-[0-9]{4}$">
年齢：<input type="number" min="18" max="99">
ID：<input type="text" maxlength="6">
```

invalidイベント

- バリデーション属性とinvalidイベントの組み合わせ

```
<form>
    郵便番号：
    <input id="zipcode" type="text" pattern="^[0-9]{3}-[0-9]{4}$">
    <input type="submit" value="送信">
</form>
<script>
    var zipcode = document.getElementById("zipcode");
    zipcode.addEventListener("invalid", changeStyle);
    function changeStyle(e) {
        e.target.style.color = "red";
    }
</script>
```

受験対策

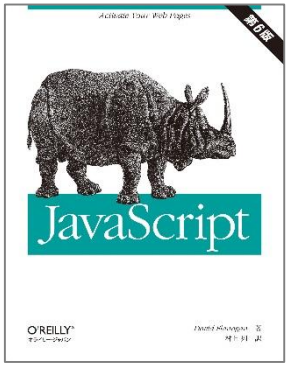
参考書籍

レベル２対策本

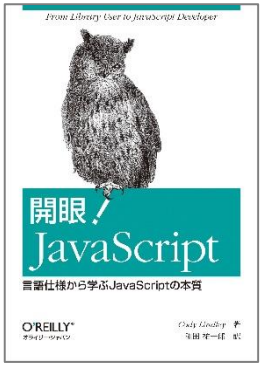
HTML5プロフェッショナル
認定試験レベル2攻略テキスト

コア・JavaScript

JavaScript



開眼！JavaScript



HTML5 API

HTML5 Web標準API バイブル

EcmaScript6の実習

https://babeljs.io/

LPI-JAPAN HTML5 Professional Certification

Open the Future with HTML5.