ななちゃんのIT教室

関数について詳しくなろう　の巻

by nara.yasuhiro@gmail.com

ななちゃんが、JavaScript　の
関数の使い方を詳しく学ぶという　お話

第 0.1 版　2017 年 6 月 18 日

フリー素材
http://freeillustration.net

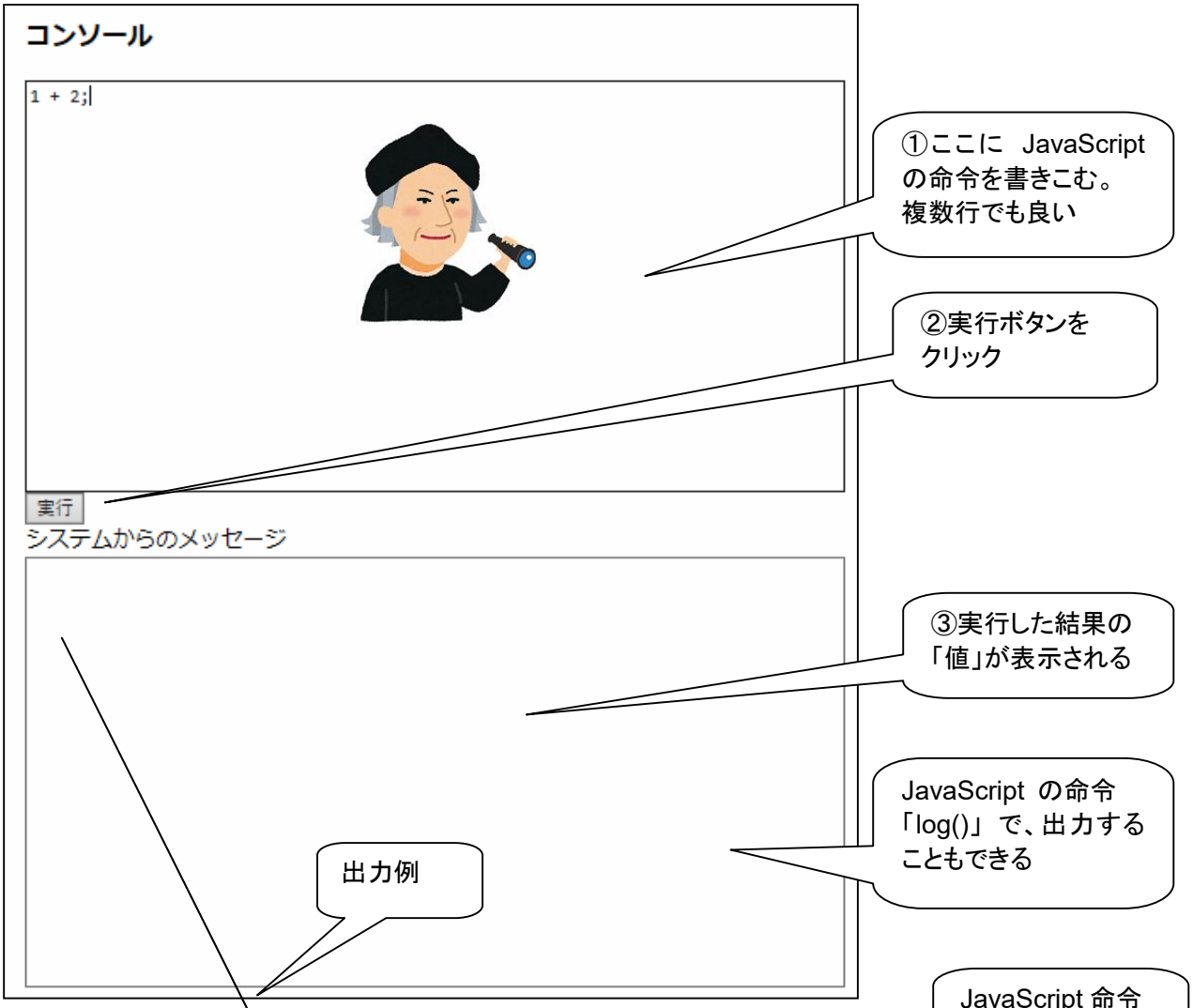
いらすとやフリー素材
http://www.irasutoya.com/

もくじ

第1回　秘密道具：マイ・コンソール

なな：クリじい、「関数の使い方」の勉強をするんだけど、便利な秘密道具はない？

クリ：あるぞ、あるぞ。定番秘密道具の「マイ・コンソール」。他の巻を読んでない読者のために、説明しよう。




```
<= 1 + 2;
=> Number number 3
<= "1" + "2"
=> String string "12"
<= 1;2;
=> Number number 2
<= var x = 1;
=> Undefined undefined undefined
<= x
=> Number number 1
<= var x; x= 1;
=> Number number 1
```

JavaScript 命令「1+2」を入力した

```
<= 1 + 2;
=> Number number 3
```

実行結果の「値」は 3

注意：var x = 1; の値は「undefined」

実行結果の「型」は Number

本　教材ではこのように圧縮表示しています

```
1 + 2;          // Number number 3
"1" + "2"       // String string "12"
1;2;            // Number number 2
var x = 1;      // Undefined undefined undefined
x               // Number number 1
var x; x= 1;    // Number number 1
```

「型」の判定方法は 2 種類 r

JavaScript 命令

実行結果の「型」と「値」

「○；○」のように、複数の JavaScript 命令がある場合、一番右の命令の型、値だけ表示される

```
<!DOCTYPE html>
<html>
 <head>
    <meta charset="utf-8">
    <title>コンソール</title>
 </head>
 <body>
    <h3>コンソール</h3>
    <textarea rows="19" cols="80" id=pg autofocus>1 + 2;</textarea>
    <br><input type=button onClick=go() value="実行">
    <br>システムからのメッセージ
    <br><textarea rows="20" cols="80" id=log></textarea>
    <script>
var geval = eval;

var logp = document.getElementById("log");
var pgp    = document.getElementById("pg");
var logd;

function clog(s) { logp.value += s; }
function log(s) { logd += s; }
function typeIs(obj) {
      return(Object.prototype.toString.call(obj).slice(8, -1)); }
function isPrimitive(x) {
    return (typeof x)!="object";
}
function toLiteral(x) {
    if (typeIs(x)=="Number" && isNaN(x)) return "NaN";
    if (x === Infinity)                  return "Infinity";
    if ((typeIs(x)!="Symbol")&&(-x === Infinity)) return "-Infinity";
    if (typeIs(x)=="Set") return "Set("+JSON.stringify([...x])+")";
    if (typeIs(x)=="Map") return "Map("+JSON.stringify([...x])+")";
    return JSON.stringify(x);
}
function type(x) { return "" + (typeof x); }
function isInteger(n) { return n%1 === 0; }
function keys(obj) { return Object.keys(obj); }
function go() {
 logd = "";
 try {
            var v = geval(pgp.value);
            clog("<= " + pgp.value + "\n=> "
               + typeIs(v) + " " + type(v) + " " + toLiteral(v) + "\n");
            pgp.value = "";
            logp.scrollTop = logp.scrollHeight;
            pgp.focus();
 }
 catch(e) { clog("<= " + pgp.value + "\n=>! " + e + "\n");
                pgp.value = "";
                logp.scrollTop = logp.scrollHeight;
                pgp.focus();
 }
 if (logd != "") clog(logd + "\n");
}
   </script>
 </body>
</html>
```

第2回　関数とは

なな：関数って何？

先生：関数はプログラムのかたまりのようなもの。JavaScript では Function クラスのオブジェクトとして記憶します。下記のようなイメージです。

```
function add(x, y) {
    var ans = x + y;
    return ans;
}

add(1, 2)          // Number number 3
```

値が数値型の 3 ということ。
alert(add(1,2)) と書けば、
「3」と表示するアラート画面が出る
ということ。

なな：メソッドというのを聞いたような気がするけど。

先生：関数でも、オブジェクトのプロパティ値として登録したものは、とくにメソッドとよんでいます。

なな：関数の特徴は？

先生：関数の中で定義された変数は、ローカル変数となり、関数外には見えません。関数外にあるグローバル変数を、関数の中からアクセスできますが、グローバル変数と同じ名前のローカル変数を宣言すると、グローバル変数はその名前でアクセスできなくなります。「window.グローバル変数名」という形で、グローバル変数にアクセスすることはできますが。

なな：関数は、どうやって使うの？

先生：関数を使うことを、「関数を実行する」とか、「関数を呼び出す」とか言います。関数呼び出し演算子 ( ) を使います。実引数は、三つ以上あっても、ひとつもなくてもかまいません。関数の用途によります。

```
戻り値 = 関数名 (実引数, 実引数);
```

なな：関数は、どうやって作るの？

先生：関数を作ることを、「関数を定義する」といいます。関数定義方法に 3 種類あります。

なな：関数定義方法の一つ目は？

先生：「関数定義文」です。下記引数の個数は、用途により、三つ以上でも、ひとつもなくてもかまいません。

```
function 関数名 (仮引数, 仮引数) {
    :
    return 戻り値;
}
```

なな：関数定義文は、プログラムのどこに書けば良いの？　プログラムの先頭？　末尾？

先生：先頭でも、末尾でも、プログラム中でも構いません。関数を、グローバルな環境（すべてのスコープの外側）で宣言した場合、グローバルな関数宣言となります。グローバルな関数宣言は、静的な関数として機能します。つまり、ファイルがブラウザに読み込まれた時点で関数宣言が評価されます。ですから、プログラムファイル中

で、後ろのほうで宣言される関数を、手前にあるプログラムから呼び出すことが可能です。 このようなことを「巻き上げ」といいます。

```
var x = add(2,3);
function add(x,y) { return x + y; }
x;          // Number number 5
```

関数 add() を、定義より前に利用している

関数 add() の定義

ローカルな環境（関数内）で子関数を宣言した場合、ローカルな関数宣言となります。ローカルな関数宣言は、動的な関数として機能します。つまり、関数スコープに入った瞬間（親関数が実行開始した時）に、スコープ内の子関数宣言が評価されます。関数スコープに入るたびに、新しい子関数オブジェクトが生成されるということです。ここでも、実行に先立ち、関数宣言が評価されるので、巻き上げが可能です。

```
var x = f(1,2);

function f(x,y) {
    return add(x,y);
    function add(a,b) { return a + b; }
}
x          // Number number 3
```

関数 add() を、定義より前に利用している

関数 add() の定義

なな：関数定義方法の二つ目は？

先生：「関数リテラル」です。function 演算子を使用して、関数を動的に生成します。

```
var 変数 = function 関数名(引数, 引数) {
    :
    return 戻り値;
};
```

なな：具体例のほうが分かりやすいかも。

先生：下記のようになります。関数名（ここでは f）は省略可能です。作られる関数オブジェクトを変数（ここではadd）に記憶することができるので、この例では、関数名を記述する必要がありません。

```
var add = function f(x,y) { return x + y; };

add(2,3);          // Number number 5
```

関数名を省略した場合、関数リテラルは、匿名関数とか、無名関数とも呼ばれます。

```
var add = function (x,y) { return x + y; };

add(2,3);          // Number number 5
```

関数リテラル

本当は不正確ですが、この形を、「関数 add を定義している」と言う場合も多々あります。匿名関数は、

```
function f () { alert("!"); };
setinterval(f,1000);
```

を、下記のように縮めて書く時に利用できます。

```
setinterval(function() { alert("!"); },1000);
```

それから、onclick = function() { alert("clicked!";) みたいな使い方もできます。

なな：「この例では、関数名を記述する必要がありません」ということは、関数名が必要な場合もあるの？

先生： 関数内から、自身の関数オブジェクトを取得したい場合なんかです。自分自身を呼び出すとか。この関数名は、関数内でのみ利用可能で、プログラムの他の部分からは見えません。でも、自身の関数オブジェクトは、関数定義を記憶した変数（この例では add）や、arguments.callee プロパティから取得することもできます。

（arguments.callee プロパティからの取得は、Internet Explorer 8 以前では利用不可）

クリ： 実験してみよう！ 同一比較演算子で true になる。

```
var s = "";
var f = function g(){
    s += (f === g) + "/";
    s += (f === arguments.callee);
};
f();
s;                    // String string "true/true"
```

なな： この定義方法でも、書く場所は自由で、巻き上げがあるの？

先生： No。関数宣言文の場合と異なり、関数リテラルが、先に評価される事や、巻き上げはありません。プログラムが上から順番に実行され、関数リテラルまで到達した時点で、初めて関数オブジェクトが生成されます。関数リテラルは、動的な関数として機能します。

なな： 関数が定義されただけでは、中身は動作しないのね。プログラムから呼び出されるまで、動作しないのね。

先生： そうです。 でも、プログラムの書き方を工夫すると、定義と同時に実行することも可能です。 そういうのを「即実行する」と言います。

```
var add;
(add = function f(x,y) { return x + y; })(1,2);       // Number number 3
add(2,3);                                             // Number number 5
```

先生： この例では、定義の時に、(1,2) という引数で「即実行」し、「3」という結果を得て、その後に、あらためて、(2,3) という引数で呼び出し実行し、「5」という結果を得ています。

```
(function f(x,y) { return x + y; })(1,2);     // Number number 3
```

```
(function (x,y) { return x + y; })(1,2);      // Number number 3
```

このふたつは、ほとんど同じに見えますが、上には「f」があり、下には「f」がありません。どちらも、関数の宣言内容は残らないので、プログラムの他の部分から呼び出して使うことはできません。定義して、1 回だけ実行して、おしまいです。したのほうの形は、特に、匿名即実行関数と呼ばれます。

なな： 宣言内容が残らないんだったら、無意味では？

先生： 下記のように、結果だけ利用するとか、

```
var x; x = (function (x,y) { return x + y; })(1,2);   // Number number 3
```

グローバル関数に影響を与えない、ということを利用した特別な用途があります。

先生：下記は、説明用の簡単なプログラムですが、変数 x が外に見えず、set()、get() という関数名も、m.set()、m.get() という形でしか使えません。このため、関数外で、x、set、get という変数や関数を自由に定義しても、名前の衝突が起きません。「m」という名前だけ避ければ大丈夫です。

```
var m = (function(){
   var x;
   return {
        set: function(d) { x = d; },
        get: function()   { return x; }
             }
})()
m.set(7)
m.get()                    // Number number 7
```

```
var m = (function(){
   var x;
   return [
        function(d) { x = d; },
        function()   { return x; }
             ]
})()
m[0](8);
m[1]();                    // Number number 8
```

な：関数定義方法の三つ目は？

先生：「Function コンストラクタ」です。下記のような形になります。

```
new Function ( 引数 , "関数内の文" ) :Function
```

引数	String	引数で使用する変数名を、必要な数だけ順番に指定。
関数内の文	String	関数内の文を、文字列で指定。
戻り値	Function	Function オブジェクト

先生：具体的には、下記のような使い方をします。

```
var add = new Function("x","y",
    "var v = x + y;" +
    "return v;"
);
add(1,2);          // Number number 3
```

```
var add = new Function("x, y", "return x + y");
add(1,2);          // Number number 3
```

な：一つ目、二つ目にくらべて、メリットはあるの？

先生：あまり使われません。でも、プログラム本体、この例では、"return x + y" の部分を、文字列で与えられるので、宣言前に編集できるというメリットがあります。下記の例では、"-" の部分を、動的に変更できます。

```
var ope = "-";
var calc = new Function("x", "y", "return x" + ope + "y;");
calc(2, 3);          // Number number -1
```

第3回 Function クラス オブジェクトのプロパティとメソッド

先生: 関数宣言文、関数リテラル、Function オブジェクトの 3 方法で宣言した関数は、いずれも、Function クラスのオブジェクトという形で記憶されます。関数内部で、下記のような情報を利用できます。

まず、「Arguments オブジェクト」。arguments 変数が記憶しています。

```
(function (){
    return arguments.length + ":" + arguments.callee;
})(1,2);
// String string "2:function (){\n\treturn arguments.length + \":\" + arguments.callee;\n}"
```

Arguments オブジェクトのプロパティ

プロパティ名	型	説明
length	Number	引数から渡されたデータの総数を取得。
callee	Function	自身の関数オブジェクトを取得。strict モードでは、利用できない。

先生: 上記のプログラム例では、「1,2」という、ふたつの引数で呼び出しているので、length が 2 になっています。callee は自分自身なので、プログラムの内容が表示されています。Arguments オブジェクトの中には、引数から渡されたデータも格納されています。

```
(function (){
    return arguments[0] + ":" + arguments[1];
})(1,2); // String string "1:2"
```

Function クラスのプロパティ

プロパティ名	型	説明
prototype	Object	「新しく生成するオブジェクト」のプロトタイプを設定。
name	String	関数名。匿名関数では "" や、undefined。IE 11 非対応。
length	Number	サポートしている引数の数。

さらに、関数の実行途中に、取得できるプロパティ。

caller	Function	呼出元関数のFunction オブジェクト。トップならnull。Strictモード不可。
arguments	Arguments	Arguments オブジェクトを取得する。(arguments 変数と同等)

Function クラスのメソッド

メソッド	説明
toString()	関数のソースコード。文字列。ネイティブ実装の組み込み関数などでは不可。
apply()	関数を実行する。(引数データは配列で指定) 。
call()	関数を実行する。(引数データは可変引数で指定)
bind()	束縛効果のある、新しい関数オブジェクトを生成する。
isGenerator()	自身がジェネレータ関数であるか調べる。

先生: apply() メソッドの使い方は、「関数名.apply (this , [引数])」

this (略可) 関数内での this を指定。null を指定した場合デフォルトの動作。
引数 (略可) 引数として渡すデータを配列に格納して指定。
戻り値 実行した関数の戻り値が得られる

例: add.apply(this,[1,2]) // Number number 3

先生：call() メソッドの使い方は、「関数名.call (this , 引数)」

this (略可)　関数内での this を指定。null を指定した場合デフォルトの動作。
引数 (略可)　引数から渡すデータを順番に指定。
戻り値　実行した関数の戻り値が得られる

例：add.call(this,1,2)　// Number number 3

なな：bind() の使い方は？

先生：「bind (this , 引数)」です。InternetExplorer 8 以前では非対応なので要注意。

this (略可)　関数内での this を指定。null を指定した場合デフォルトの動作。
引数 (略可)　引数配列に対して、0 番地から定数を挿入したい場合に指定。（引数の上書きではない）
戻り値　束縛効果（this の設定）のある、新しい関数オブジェクト。

```
var x = 9;
var module = {
   x: 81,
   getX: function() { return this.x; }
};

module.getX();                  // Number number 81

var outerGetX = module.getX;
outerGetX();                    // Number number 9

var boundGetX = outerGetX.bind(module);
boundGetX();                    // Number number 81

outerGetX.bind(module)();       // Number number 81

var getX2 = function() { return this.x; }
getX2();                        // Number number 9

getX2.bind(module)();           // Number number 81
```

```
function add(x,y) { return x + y; }
add(1,2);                   // Number number 3

var add1 = add.bind(this,1);
add1(2);                    // Number number 3

var add2 = add.bind(undefined,2);
add2(2);                    // Number number 4
```

なな：isGenerator() メソッドの使い方は？

先生：「関数. .isGenerator( )」です。ECMAScript 2015 からの機能です。
引数は、なし。戻り値は、自身がジェネレーター関数であれば true、そうでなかれば false です。

第4回　JavaScriptの識別子

なな：識別子って？

先生：変数や関数の名前のことよ。命名規則は以下のとおり。

・先頭文字は英字、アンダースコア（_）、ドル記号（$）のいずれか
・2文字目以降は英数字、アンダースコア、ドル記号のいずれか

JavaScriptで固有の意味を持つ予約語は、識別子として利用することはできない。予約語は、

break case catch continue default delete do else finally for function if in instanceof new return switch this throw try typeof var void while with

JavaScriptですでに定義されているオブジェクトやそのメンバ名（例えば、ArrayやNumber、Objectなど）も、特別な理由がない限り、識別子として利用するのは避けるべき。もともとの機能が利用できなくなる。

将来的に予約語として採用される可能性があるキーワード。具体的には、

abstract boolean byte char class const debugger double enum export extends final float goto implements import int interface long native package private protected public short static super synchronized throws transient volatile

第5回　再帰

なな：再帰って？

先生：関数がその関数自身を呼び出すことよ。自身を呼び出す関数のことを再帰関数と言います。例、階乗計算。

```
function factorial(n){
   if ((n == 0) || (n == 1))
      return 1;
   else
      return (n * factorial(n - 1));
}

factorial(1); // Number number 1
factorial(2); // Number number 2
factorial(3); // Number number 6
factorial(4); // Number number 24
factorial(5); // Number number 120
```

先生：関数が、自分自身のオブジェクトを参照する方法は 3 種類あります。

```
var foo = function bar() { ... };
```

の場合、「 … 」の部分で、

1.関数名：	bar()
2.arguments.callee：	arguments.callee()
3.関数を参照したスコープ内変数	foo()

先生：無限再帰を防ぐための条件（終了条件）が必要です。

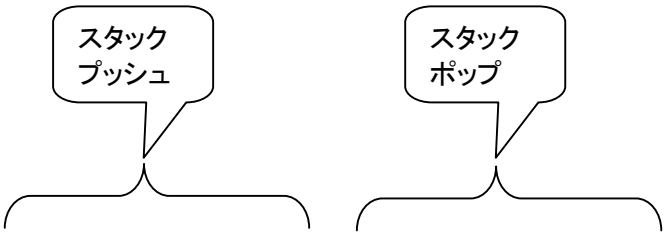
```
function loop(x) {
   if (x >= 10) return;     // 終了条件
     :
   loop(x + 1);             // 再帰呼出し
}
loop(0);
```

先生：単純な反復ループでは行えないアルゴリズムもあります。 例えば、 ツリー構造（例えば DOM）のすべてのノードを取得するのに、再帰を使うと簡単になります。

```
function walkTree(node) {
   if (node == null) return;
   // ノードに対し処理を行う
   for (var i = 0; i < node.childNodes.length; i++) {
      walkTree(node.childNodes[i]);
   }
}
```

先生：再帰は関数スタックを使っている。下記の例で、スタックのふるまいを見ることができます。

```
var s = "";
function foo(i) {
   if (i < 0) return;
   s += 'begin:' + i + "/";
   foo(i - 1);
   s += 'end:' + i + "/";
}
foo(3);
s;        // String string "begin:3/begin:2/begin:1/begin:0/end:0/end:1/end:2/end:3/"
```



第6回 クロージャ

ののの：**クロージャ**って？

先生：関数は、**スコープ**として機能します。通常であれば、ローカル変数は、スコープの外に出ると消滅します。関数の中では、関数を宣言したり、動的に関数を作成することができます。任意のスコープ内で作られた関数を、**ローカル関数**と言います。ローカル関数の中からは、同じスコープ内とスコープの外側に存在する環境にアクセスする事ができます。ローカル関数は、オブジェクトの一種なので、スコープの外に持ち出すことができます。スコープの内側で作成した関数オブジェクトを、スコープの外側から実行できます。この場合、スコープ内の環境は消滅しません。関数オブジェクトが消滅しない限り、スコープ内の環境も生存し続けます。関数が生存していれば、その**関数内からアクセスできるすべての環境も生存し続けます**。この仕様を、**クロージャ**といいます。

```
function f() {
        var local_a = 123;
        function g() {
            return local_a;
        }
        return g;
}

var outerG = f();
outerG();       // Number number 123
```

関数の外側から見えない変数

関数の外側から見えない変数のデータを得た

```
function f() {
        var count = 0;
        function g() {
            return ++count;
        }
        return g;
}

var outerG = f();
outerG();       // Number number 1
outerG();       // Number number 2
outerG();       // Number number 3
```

関数の外側から見えない変数

関数の外側から見えない変数のデータを得た

関数の外側から見えない変数のデータを得る仕組みがクロージャ

先生：引数の変数も、ローカル変数として扱われるため、生存し続けます。

```
function f(name) {
        var count = 0;
        return function () {
                count++;
                return(name + ":" + count);
        };
}

var g1 = f("obj1");
var g2 = f("obj2");

g1();   // String string "obj1:1"
g1();   // String string "obj1:2"
g1();   // String string "obj1:3"
g2();   // String string "obj2:1"
g2();   // String string "obj2:2"
g1();   // String string "obj1:4"
```

name が引数

オブジェクト g1 と g2 は、引数の値 “obj1” と “obj2” を記憶している

第7回　関数の仮引数（デフォルト仮引数と残余仮引数）

先生：これらは、ECMAScript 2015 からの機能です。

な：デフォルト仮引数は？

先生：JavaScript では、実引数が与えられなかった仮引数は、undefined となります。以前、デフォルト値を設定するには、仮引数の値が undefined だった場合にデフォルト値を代入する必要がありました。

```
function multiply(a, b) {
   b = (typeof b === 'undefined') ? 1 : b;
   return a*b;
}
multiply(5);       // Number number 5
```

```
function say(msg) {
   msg = msg || "Hi!";
   return msg;
}
say("Hello!");    // String string "Hello!"
say();            // String string "Hi!"
```

先生：新しい、デフォルト仮引数を使うと、下記のように、とても簡潔に書けます。

```
function multiply(a, b = 1) {
   return a*b;
}
```

な：残余仮引数は？

先生：不特定多数の引数を配列のように表すことができます。「…resr」の部分が残余仮引数。

```
function mul(m, ...rest) {
    for (var i=0; i<rest.length; i++) m *= rest[i];
    return m;
}

mul(2, 1, 2, 3);   // Number number 12
```

先頭→m、残余→rest

rest = [1, 2, 3]

2 * 1 * 2 * 3 = 12

先生：さらに、ECMAScript 2015 からの、reduce を使うと、もっとコンパクトになります。

```
function mul(m, ...rest) {
    return rest.reduce(function(a,b){return a*b;}, m);
}

mul(2, 1, 2, 3);   // Number number 12
```

初期値を m として、rest[] の各要素を順に掛けてゆく

先生：さらに、後述のアロー関数を使えば、こんなにコンパクトになります。

```
function mul(m, ...rest) {
    return rest.reduce((a,b)=>a*b, m);
}

mul(2, 1, 2, 3);   // Number number 12
```

function(a,b){return a*b;
→ (a,b) => a*b

第8回 アロー関数

なな：アロー関数って？

先生：関数定義の短縮形と、レキシカルな this が特徴です。ECMAScript 2015 からの機能です。

```
var show = function(text) { return text; }
```

→

```
var show = (text) => { return text; };
show("hello");  // String string "hello"
```

引数が一つの時には()を省略可能

```
var show = text => { return text; };
```

また一文の場合、{}とreturnの省略も可能。

```
var show = (text) => text;
```

なな：レキシカルな this は？

先生：これまでの関数定義では、関数の中に子関数があると、グローバルスコープに戻るという問題がありました。

```
data = 456;
function cls() {
    this.data = 123;
    this.func = function(){return this.data;};
}
(new cls()).func(); // Number number 123
```

```
data = 456;
function cls() {
    this.data = 123;
    this.func = function(){
        function g() { return this.data;};
        return g();
    }
}
(new cls()).func();        // Number number 456
```

先生：これに対処するため、self などのローカル変数を宣言し、this の値をコピーしたりしていました。

```
data = 456;
function cls() {
    this.data = 123;
    var self = this
    this.func = function(){
        function g() { return self.data;};
        return g();
    }
}
```

先生：アロー関数ではこれを解決しました。レキシカルな this とは文法的に常識的なthisに治ったということです。

```
data = 456;
function cls() {
    this.data = 123;
    this.func = () => {return this.data;};
}
(new cls()).func();        // Number number 123
```

親関数　従来通り

```
data = 456;
function cls() {
    this.data = 123;
    this.func = function(){
        g = () => { return this.data;};
        return g();
    }
}
(new cls()).func();        // Number number 123
```

子関数　治った！

```
data = 456;
function cls() {
    this.data = 123;
    this.func = function(){
        g = () => this.data;
        return g();
    }
}
(new cls()).func();        // Number number 123
```

中身が 1 文なので、{ } 省略

```
data = 456;
function cls() {
    this.data = 123;
    this.func = () => {
        g = () => { return this.data;};
        return g();
    }
}
(new cls()).func();        // Number number 123
```

親関数もアローに

先生：従来の function も直すと、これまでのプログラムが誤動作する可能性があるので、新しいアロー関数だけ直したということです。

第9回　コンストラクタ関数

先生：Function オブジェクトは、コンストラクタ関数としても機能します。コンストラクタ関数は、新しいオブジェクトを初期化する為に利用することができるものです。コンストラクタ関数をインスタンス化（実体化）するには、new 演算子を使用します。

```
var 変数 = new コンストラクタ関数 (引数);
```

なな：コンストラクタ関数？　インスタンス化？

先生：コンストラクタ関数はたいやきの型、インスタンスはたいやきみたいなものね。「new 演算子」を使った場合、生成される新しいオブジェクト（インスタンス）内の関数の this キーワードの値は、インスタンス自身を指すように設定されます。

なな：具体例は？

先生：コンストラクタ関数で、プロパティを初期化設定します。コンストラクタ関数内では、return 文は使いません。

```
function MyClass () {
    this.aaa = "a";
    this.bbb = "b";
    this.ccc = "c";
}

var MyInstance = new MyClass();
MyInstance;     // Object object {"aaa":"a","bbb":"b","ccc":"c"}
```

先生：コンストラクタ関数内で、変数を宣言（var 変数名）した場合、ローカル変数となります。関数スコープの外側から、ローカル変数に直接アクセスする事はできません。通常であれば、コンストラクタ関数を抜けると、ローカル変数は消滅します。でも、パブリックなメソッドを用意すればローカル変数を生存させ続ける事ができます。

```
function MyClass (name, value) {
    var _name;
    var _value;
    this.getName = function ()   { return _name; };
    this.setName = function (v) { _name = v; };
    this.getValue = function () { return _value; };
    this.setValue = function (v){ _value = v; };
    _name   = (name ===   undefined)?"元の名前":name;
    _value = (value === undefined)?123:value;
}

var obj = new MyClass();
obj.getName();                  // String string "元の名前"
obj.getValue();                 // Number number 123
obj.setName("新しい名前");
obj.setValue(456);
obj.getName();                  // String string "新しい名前"
obj.getValue();                 // Number number 456
var obj2 = new MyClass("新しい名前2",789);
obj2.getName();                 // String string "新しい名前2"
obj2.getValue();                // Number number 789
```

先生:コンストラクタ関数は、プロトタイプと呼ばれるオブジェクトを、最初から持っています。関数からプロトタイプを取得するには、prototype プロパティを使用します。コンストラクタ関数が所有していたプロトタイプは、新しいインスタンスと関連付けられます。新しいインスタンスにとって、プロトタイプは、原型(親)となります。インスタンスから、プロトタイプとなるオブジェクトを取得するには、Object.getPrototypeOf() メソッドを使用します。__proto__ プロパティからアクセスする事もできます。変更も可能です。

クリ: 実験で確かめてみよう。

```
var obj = new MyClass();
Object.getPrototypeOf(obj)===MyClass.prototype; // Boolean boolean true
```

```
var obj = new MyClass();
obj.__proto__===MyClass.prototype; // Boolean boolean true
```

先生: コンストラクタ関数のプロトタイプは、すべての派生インスタンス間で共有されます。もしプロトタイプの内容を変更した場合、すべての派生オブジェクトに影響があります。プロトタイプは、別のオブジェクトに**取り換える**事ができます。prototype プロパティの設定は、実体化よりも以前に、済ませておく必要があります。実体化後に変更しても、生成済みのインスタンスには、反映されません。 組み込みクラス関数の prototype プロパティは、変更できません。

```
function MyClass () { this.xxx = "x"; }
MyClass.prototype = { aaa:"a", bbb:"b", ccc:"c" };

var obj = new MyClass();

MyClass.prototype = { ddd:"d", eee:"e", fff:"f" };

obj.aaa;          // String string "a"
obj.xxx;// String string "x"
obj.ddd;          // Undefined undefined undefined

var obj2 = new MyClass();
obj2.aaa;         // Undefined undefined undefined
obj2.xxx;         // String string "x"
obj2.ddd;         // String string "d"

obj               // Object object {"xxx":"x"}
obj2              // Object object {"xxx":"x"}
Object.getPrototypeOf(obj);     // Object object {"aaa":"a","bbb":"b","ccc":"c"}
Object.getPrototypeOf(obj2);    // Object object {"ddd":"d","eee":"e","fff":"f"}
```

prototype =
はプロトタイプ
取り換え

先生: デフォルトのプロトタイプを**拡張**してみましょう。最初から存在するプロトタイプに対して、機能を追加する事ができます。組み込み関数(組み込みクラス)のプロトタイプも、拡張できるけど、プログラムが読みにくくなるので、しない方が良いです。プロトタイプの「拡張」は、「取り換え」とは異なります。

```
function MyClass () { this.xxx = "x"; }

MyClass.prototype.aaa = "a";
MyClass.prototype.bbb = "b";
MyClass.prototype.ccc = "c";

var obj = new MyClass();

obj.aaa;          // String string "a"
obj.xxx;          // String string "x"
```

prototype.xxx =
はプロトタイプ**拡張**

先生：プロトタイプを使って、オブジェクトを数珠つなぎのように連結する事ができます。 このような連結構造を、「**プロトタイプチェーン**」といいます。

```
function MyFunc_A (){ }
function MyFunc_B (){ }
function MyFunc_C (){ }

var obj_a = new MyFunc_A();

  MyFunc_B.prototype = obj_a;
var obj_b = new MyFunc_B();

  MyFunc_C.prototype = obj_b;
var obj_c = new MyFunc_C();
```

なな：プロトタイプチェーン？　どんな働きをするの？

先生：オブジェクトのプロパティに、読み取りアクセスを試みたとします。オブジェクトにプロパティが存在する場合、オブジェクトからデータを取得。存在しなかった場合、次に、プロトタイプにアクセス。プロトタイプにプロパティが存在する場合、プロトタイプからデータを取得。存在しなかった場合、さらに次のプロトタイプにアクセス。最終的に、プロトタイプチェーン内に存在しなかった場合、未定義となります。

```
function MyFunc () { this.bbb = "b"; }

MyFunc.prototype.aaa = "a";

var obj = new MyFunc();

console.log(obj.bbb); // "b"
console.log(obj.ccc); // undefined
```

```
function MyFunc_A () { this.aaa = "a"; }
function MyFunc_B () { this.bbb = "b"; }
function MyFunc_C () { this.ccc = "c"; }

MyFunc_B.prototype = new MyFunc_A();
MyFunc_C.prototype = new MyFunc_B();

var obj_c = new MyFunc_C();

obj_c.ccc;      // String string "c"
obj_c.bbb;      // String string "b"
obj_c.aaa;      // String string "a"
obj_c.xxx;      // Undefined undefined undefined
```

先生： オブジェクトのプロパティに、書き込みアクセスを試みたとします。プロパティが存在しなかった場合、自身のオブジェクトにプロパティが追加されます。存在する場合、自身のプロパティにデータがセットされます。書き込みアクセスによって変化するのは、自身のオブジェクトなので、プロトタイプ内が、汚染する事はありません。

```
function MyFunc () { }
MyFunc.prototype.aaa = "a";
MyFunc.prototype.bbb = "b";

var obj0 = new MyFunc();
var obj1 = new MyFunc();

obj0.aaa = "A";
obj1.bbb = "B";

MyFunc.prototype.aaa;  // String string "a"
MyFunc.prototype.bbb;  // String string "b"

obj0.aaa;              // String string "A"
obj1.aaa;              // String string "a"
obj0.bbb;              // String string "b"
obj1.bbb;              // String string "B"
```

先生： メソッドは、クラス本体（Function定義）に入れることも、prototype に入れることもできます。クラス本体に入れたメソッド定義は、インスタンスにコピーされます。prototype に入れたメソッドは、インスタンスにはコピーされず、メソッドが呼び出された時にクラス本体の prototype が参照されます。メソッドの数が多い場合、メソッドを prototype に入れたほうが、インスタンスが小さくて済みます。

```
function MyClass (name, value) {
    this._name;
    this._value;
    this.getName = function ()   { return this._name; };
    this.setName = function (v) { this._name = v; };
    this.getValue = function () { return this._value; };
    this.setValue = function (v){ this._value = v; };
    this._name   = (name ===   undefined)?"元の名前":name;
    this._value = (value === undefined)?123:value;
}
var obj = new MyClass();
obj.getName();              // String string "元の名前"
obj.getName+"";             // String string "function ()   { return this._name; }"
MyClass.prototype.getName+""; // String string "undefined"

function MyClass2 (name, value) {
    this._name;
    this._value;
    this._name   = (name ===   undefined)?"元の名前":name;
    this._value = (value === undefined)?123:value;
}
MyClass2.prototype.getName = function ()   { return this._name; };
MyClass2.prototype.setName = function (v) { this._name = v; };
MyClass2.prototype.getValue = function () { return this._value; };
MyClass2.prototype.setValue = function (v){ this._value = v; };

var obj2 = new MyClass2();
obj2.getName();             // String string "元の名前"
obj2.getName+"";            // String string "function ()   { return this._name; }"
MyClass2.prototype.getName+""; // String string "function ()   { return this._name; }"
```

prototype を参照している

先生：組み込みクラスを継承した拡張クラスを作って使うこともできます。元のクラスは影響を受けません。

```
var MyError = function(message) {
	this.name = "myError";
	this.message = message || "my error";
};
MyError.prototype = new Error();

var s = "";
try {
	throw new MyError();
} catch (e) {
	if (e instanceof MyError) {
		s += e.message;
	}
}
s;		// String string "my error"
```