ソフトウェア開発
第3回目授業

平野 照比古

2018/10/12

問題 1

- 単に実行結果だけでは満点ではない。
- 必ず考察を付けること。
- 今回の場合、`substring()` と `slice()` の比較が欲しい。

問題 2

- 単に実行結果しか書いていないものが多かった。必ず、意味も含めた考察が欲しい。
- 2. では 2 つの変数の値の交換が簡単にできることを指摘してほしい。

問題 3

- コンソールに表示された結果をすべて書くこと
- 最後の `undefined` は `console.log()` の戻り値なので不要
- `pop()` と `push()` を組み合わせればスタックが容易に構成できる。
- `shift()` と `push()` を組み合わせればキューが容易に構成できる。
- `slice()` と `splice()` の違いを書いてないものが多かった。

問題 4(1)

- 与えられた日時から 1 週間後の日時

```
>theDay = new Date();
Wed Oct 04 2017 23:50:32 GMT+0900 (東京 (標準時))
>new Date(theDay.getFullYear(),theDay.getMonth(),
  theDay.getDate()+7);
Wed Oct 11 2017 00:00:00 GMT+0900 (東京 (標準時))
```

- 与えられた日時の翌月の 1 日

```
>new Date(theDay.getFullYear(),theDay.getMonth()+1,1);
Wed Nov 01 2017 00:00:00 GMT+0900 (東京 (標準時))
```

- 与えられた日時の前の月の最終日

```
>new Date(theDay.getFullYear(),theDay.getMonth(),0);
Sat Sep 30 2017 00:00:00 GMT+0900 (東京 (標準時))
```

問題 4(2)–与えられた日時の月の第 1 月曜日

```
>new Date(theDay.getFullYear(),theDay.getMonth(),
  (theDay.getDate()-theDay.getDay()+7+1)%7||7);
Mon Oct 02 2017 00:00:00 GMT+0900 (東京 (標準時))
```

- 第 1 月曜日を求めることに関しては正しい答えが多かった。
- その他の曜日に変更したときに簡単に変更できない解答がすべて

```
>for(m=0;m<12;m++){
    theDay = new Date(2018,m,12);
    console.log(new Date(theDay.getFullYear(),theDay.getMonth(),
                (theDay.getDate()-theDay.getDay()+1 +7)%7||7));
}
 Mon Jan 02 2017 00:00:00 GMT+0900 (東京 (標準時))
 Mon Feb 06 2017 00:00:00 GMT+0900 (東京 (標準時))
 Mon Mar 06 2017 00:00:00 GMT+0900 (東京 (標準時))
 Mon Apr 03 2017 00:00:00 GMT+0900 (東京 (標準時))
 Mon May 01 2017 00:00:00 GMT+0900 (東京 (標準時))
 Mon Jun 05 2017 00:00:00 GMT+0900 (東京 (標準時))
 Mon Jul 03 2017 00:00:00 GMT+0900 (東京 (標準時))
 Mon Aug 07 2017 00:00:00 GMT+0900 (東京 (標準時))
 Mon Sep 04 2017 00:00:00 GMT+0900 (東京 (標準時))
 Mon Oct 02 2017 00:00:00 GMT+0900 (東京 (標準時))
 Mon Nov 06 2017 00:00:00 GMT+0900 (東京 (標準時))
 Mon Dec 04 2017 00:00:00 GMT+0900 (東京 (標準時))
```

前回の演習問題の解答 (1)

次の式の評価結果を求めなさい。

式	結果	理由
`4+"5"`	`"45"`	右のオペランドが文字列なので左の数は文字列に変換され、それらが連接される。
`4-"5"`	-1	演算子が-なので右の文字列が数に変換される。
`4+"ff"`	`"4ff"`	前と同様
`4+"0xff"`	`"40xff"`	前と同様
`4-"0xff"`	`"-251"`	前と同様

前回の演習問題の解答 (2)

次の式の評価結果を求めなさい。

式	結果	理由
`4+parseInt("ff")`	`NaN`	文字列内に数として正しく変換されるものがないので `parseInt()` の戻り値が `NaN` となり、これ以降の数の演算は `NaN` となる。
`4+parseInt("0xff")`	259	`parseInt()` は正しく 16 進数として解釈するので $4 + 255 = 259$ となる。
`4+parseInt("ff",16)`	259	基数を 16 と指定しているので、正しく 255 と解釈される。

前回の演習問題の解答 (3)

次の式の評価結果を求めなさい。

4+"1e1"	41e1	文字列の連接
4+parseInt("1e1")	5	"1e1" は数値リテラルとしては $1 \times 10^1 = 10$ を表すが、parseInt() は整数リテラル表記しか扱わない．数の変換は e の前で終わるので 1 の値が戻り値となる。
4+parseFloat("1e1")	14	parseInt() と異なり、parseFloat() の戻り値は 10 となる。

前回の演習問題の解答 (4)

次の式の評価結果を求めなさい。

"4"*"5"	20	文字列の間では*の演算が定義されていないので両方とも数に変換されて計算される。
"4"/"5"	0.8	上と同様

前回の演習問題の解答 (5)

次の式の評価結果を求めなさい。

`[].length`	0	配列の要素がないので長さは 0 となる。
`[[]].length`	1	長さを求める配列は空の配列一つを要素に持つ。
`0 == "0"`	`true`	文字列`"0"`が数 0 に変換されて比較される
`0 == []`	`true`	空の配列が空文字`""`に変換されたのち、数 0 に変換される。
`"0" == []`	`false`	空の配列が空文字`""`に変換されたのち、文字列同士の比較が行われる。
`![]`	`false`	空の配列はオブジェクトとして存在するので `true` と解釈され、否定演算子で `false` になる。

前回の演習問題の解答 (6)

次の式の評価結果を求めなさい。

`false == []`	`true`	空の配列が空文字""に変換されたのち、数 0 に変換される。
`false == undefined`	`false`	`false` が数に 0 に変換され、数と+undefined+は等しくない。
`[] == []`	`false`	配列はオブジェクトであり、二つの空の配列は別物とみなされる。
`typeof []`	"object"	配列はオブジェクトである。
`null == undefined`	`true`	この比較は `true` と定義されている。
`a=[], b=a, a==b;`	`true`	同じメモリーにあるオブジェクトを参照している。

簡単な関数の例

sum() という関数を定義している例

```
function sum(a,b) {
  var c = a + b;
  return c;  // return a + b;  でもよい。
}
```

関数の定義の構成

- `function` キーワード
  戻り値の型を記す必要はない。
- 関数の名前
  `function` の後にある識別名が関数の名前になる。この場合は `sum` が関数の名前になる。
- 引数のリスト
  関数名の後に () 内にカンマで区切られた引数を記述する。この場合は変数 `a` と `b` が与えられている。引数はなくてもよい。
- 関数の本体であるコードブロック
  {}で囲まれた部分に関数の内容を記述する。
- `return` キーワード
  関数の戻り値をこの後に記述する。戻り値がない場合には戻り値として `undefined` が返される。

関数の実行例

```
>sum(1,2)
3
>sum(1)
NaN
>sum(1,2,3)
3
```

関数の実行例の解説

- 引数に 1 と 2 を与えれば期待通りの結果が得られる。
- 引数に 1 だけを与えた場合、エラーが起こらず、NaN となる。これは、不足している引数 (この場合には b) には undefined が渡されるためである。1+undefined の結果は NaN になる。
- 引数を多く渡してもエラーが発生しない。無視されるだけである。

これらのことから JavaScript の関数はオブジェクト指向で使われるポリモーフィズムをサポートしていない。

関数の再定義

同じ関数を定義してもエラーにならない。後の関数の定義が優先される。

```
function sum(a, b){
  return a+b;
}
function sum(a, b, c){
  return a+b+c;
}
```

仮引数への代入

- 仮引数に値を代入してもエラーとはならない。
- 仮引数の値がプリミティブなときとそうでないときとでは呼び出し元における変数の値が異なる。

呼び出した関数の中で仮引数の値を変化させたときの例

```
function func1(a){
  a = a*2;
  return 0;
}
function func2(a){
  a[0] *=2;
  return 0;
}
```

呼び出した関数の中で仮引数の値を変化させたときの例の解説

- func1() では仮引数 a の値を 2 倍している。
- 次のように実行すると、呼び出し元の変数の値には変化がない
- プリミティブな値を仮引数で渡すと値そのものが渡される

```
>a = 4;
4
>func1(a); // 戻り値は 0
0
>a;          //a の値は変化しない
4
```

呼び出した関数の中で仮引数の値を変化させたときの例

- func2() の仮引数は配列が想定してある。この先頭の値だけ 2 倍される関数である
- これに配列を渡すと、戻ってきたとき配列の先頭の値が変化している
- プライミティブ型以外では仮引数の渡し方が参照渡しである

```
>a = [1,2,3];
[1, 2, 3]
>func2(a);
0
>a;
[2, 2, 3]   //配列の先頭の値が 2 倍されている
```

可変引数をとる関数

引数の数を固定しないようにするためには仮引数に展開演算子をつける[1]。

```
function sumN(...args) {
    let S = 0;
    for(let i=0;i<args.length;i++) {
        S += args[i];
    }
    return S;
}
```

- 1 行目で関数の仮引数 args に展開演算子 ... をつける。
- args は配列となのでその大きさは args.length
- これを用いて総和のプログラムが 2 行目から 5 行目に記述

[1] これまでの JavaScript では、関数の引数を表す配列のような性質を持つ arguments オブジェクトが用意されていたが、ES2015 では非推奨となった。

実行例

実行例は次のとおりである。

```
>sumN(1,2,3,4);   //戻り値は 10
>sumN(1,2,3,4,5);//戻り値は 15
```

JavaScript における変数のスコープ

変数のスコープとはある場所で使われている変数がどこから参照できるかという概念

1. 非 strict モードでは変数は宣言しなくても使用できる。
2. let により明示的に定義された変数はそのブロック内で有効
3. let で宣言された変数はそのブロック内で宣言より前で使用不可
4. 関数内で var により宣言された変数は関数の先頭で宣言したと同じ (変数の巻き上げ)。初期化は宣言の位置で行われる。
5. 関数の外で宣言された変数や宣言されないで使用された変数はすべてグローバルとなる。

スコープチェイン

関数の中で関数を定義すると、その内側の関数内で `var` や `let` で宣言された変数のほかに、一つ上の関数で利用できる (スコープにある) 変数が利用できる。これがスコープチェインである。

スコープチェインの解説

```
let G1, G2;
function func1(a) {
  let b, c;
  function func2() [
    let G2, c;
    ...
  }
}
```

スコープチェインの解説

- 関数 `func1()` ではグローバル変数である `G1` と `G2`、仮引数の `a` とローカル変数 `b` と `c` が利用できる。
- 関数 `func2()` ではグローバル変数である `G1`、`func1()` の仮引数の `a` と `func1()` のローカル変数 `b`、`func2()` のローカル変数 `G2` と `c` が利用できる。

このように内側で定義された関数は自分自身の中で定義されたローカル変数があるかを探し、見つからない場合には一つ上のレベルでの変数を探す。これがスコープチェインである。
JavaScript の関数のスコープは関数が定義されたときのスコープチェインが適用される。これをレキシカルスコープと呼ぶ。
レキシカルスコープは静的スコープとも呼ばれる。これに対して実行時にスコープが決まるものは動的スコープと呼ばれる。

変数のスコープの確認 (1)

```
let S = "global";
function func1(){
  console.log(S);
  return 0;
}
```

変数のスコープの確認 (1) 解説

- グローバル変数 S が宣言されていて"global" という文字列の値に初期化されている。
- func1() が定義されている。
- S の値をコンソールに出力している。
- この関数内で変数 S は宣言されていないので 2 行目で定義したグローバル変数が参照される。

```
>func1();
global
0
```

変数のスコープの確認 (2)

```
let S = "global";
function func2(){
  console.log(S);
  let S = "local";
  console.log(S);
  return 0;
}
```

変数のスコープの確認 (2) 解説

- S の値をコンソールに 2 回出力している。
- この関数内で変数 S はローカル変数として宣言されている。
- そのしたの変数 S はローカル変数となる
- この後、let で宣言されるので文法上エラーとなる。
- var で宣言すると、この段階ではローカル変数 S には値が代入されていないのでその値は undefined となり、後の出力はその前で定義された値となる。

var で宣言した場合の出力

```
>func2();
undefined
local
0
```

変数のスコープの確認 (3)

```
let S = "global";
function func3(){
  let S = "local";
  func1();
  return 0;
}
```

- ローカル変数 S を定義して、初期値を"local"としている。
- 初めに定義した関数 func1() を呼び出している。
- func1() の実行の際は、もともとこの関数が定義された時の変数 S(1 行目) が参照される。

```
>func3();
global
0
```

変数のスコープの確認 (4)

```
let S = "global";
function func4(){
  let S = "local in func4";
  func5 = function() {
    console.log(S);
    return 0;
  };
  console.log(S);
  return 0;
}
```

変数のスコープの確認 (4) 解説

- ローカル変数 s の値を設定している。
- その後の出力は設定した値となる。

```
>func4();
local in func4
0
```

変数のスコープの確認 (5)

- 関数オブジェクトを変数 func5 に代入している。これにより func5() という関数が定義される。
- 変数宣言がないので変数 func5 はグローバル変数となる
- func5() 内では変数 S の内容を出力が定義されている。
- func4() を実行した後では func5() が実行できる。
- 関数 func5() が定義された段階での変数 S はこの関数が func4() の中で定義されているので、18 行目の変数 S が参照される。

```
func5();
local in func4
0
```

関数もデータ

- 関数もデータ型のひとつなので、関数の定義を変数に代入することができる。
- 代入はいつでもできるので、実行時に関数の定義を変えることも可能
- 関数の戻り値として関数自体を返すことも可能

無名関数とコールバック関数

- 関数オブジェクトは関数の引数として直接渡すこともできる。その関数には名がなくてもよい (無名関数)。
- イベント (マウスがクリックされた、一定の時間が経過した) が発生したときに、その処理を行う関数を登録する必要がある。このように関数に引数として渡される関数のことをコールバック関数という。

コールバック関数の例

一定の経過時間後にある関数を呼び出す window オブジェクトの setTimeout() メソッドの使用例

```
1  let T = new Date();
2  window.setTimeout(
3    function callMe(){
4      let NT = new Date();
5      if(NT.getTime()-T.getTime()<10000) {
6        console.log(Math.floor((NT.getTime()-T.getTime())/100
7        window.setTimeout(callMe,1000);
8      }
9    },1000);
```

無名関数の例の解説

- １行目では実行開始時の時間を変数 T に格納。単位はミリ秒。
- このメソッドは一定時間経過後に呼び出される関数と、実行される経過時間 (単位はミリ秒) を引数に取る。
- 実行する関数は 3 行目から 9 行目で定義されている。
- この関数内で一定の条件のときはこの関数を呼び出す。この関数の名前は `callMe`(3 行目)。
- 4 行目で呼び出されたときの時間を求め、経過時間が 10000 ミリ秒以下であれば (5 行目)、経過時間を秒単位で表示する (6 行目)。
- さらに、自分自身を 1 秒後に呼び出す (7 行目)。

自己実行関数の必要性

次のコード考える。

```
let i;
for(i=1;i<10;i++) {
   console.log(i+" "+i*i);
}
```

このプログラムを実行すると 1 から 9 までの値とそれの 2 乗の値がコンソールに出力される。実行後に、コンソールに i と入力すれば 10 が出力される。
関数を定義すれば、その関数はグローバル空間に残る。

自己実行関数の例

定義した関数をその場で実行できる機能

```
(function(){
  let i;
  for(i=1;i<10;i++) {
    console.log(i+" "+i*i);
  })();
```

関数の定義を全体で () で囲み、そのあとに関数の呼び出しを示すための () を付ける。
この技法は、初期化の段階で 1 回しか実行しない事柄を記述し、かつグローバルな空間を汚さない (余計な変数などを残さない) 手段として用いられる。
let で変数 i を for 内で宣言すると、その変数は for 文内でしか存在しない。また、スコープの規則が少し異なる。

クロージャとは

- 関数内部で宣言された変数は、その外側から参照することができない。
- その関数は関数内のローカル変数を閉じ込めている。
- 関数内部で定義された関数を外部に持ち出す (グローバルな関数にする) と、持ち出された関数のスコープチェイン内に定義された親の関数のスコープを引き継いでいることから、親の関数のローカル変数の参照が可能

関数に対して依存する環境 (変数や呼び出せる関数などのリスト) を合わせたものをその関数のクロージャと呼ぶ。

クロージャの例 – 変数を隠す

```
function f1() {
  let n=0;
  return function() {
    return n++;
  };
}
```

実行例 (変数 n が参照できない)

```
>ff= f1();
() {
 return n++;
 }
>n;
VM351:2 Uncaught ReferenceError: n is not defined(…)
```

クロージャの例 – 変数を隠す (実行例その 2)

変数 n は存在している

```
>ff();
0
>ff();
1
>ff();
2
>ff2=f1();
() {
    return n++;
  }
>ff();
2
>ff2();
0
```

クロージャの例 – 変数を隠す (解説)

- 変数 `ff` に関数 `f1()` で返される関数オブジェクトを代入する。
- `f1()` 内のローカル変数は参照できない。
- 関数 `f1()` を実行して、戻り値の関数を実行すると、`f1()` 内のローカル変数が参照できている。
- 何回か実行すると戻り値が順に増加していることがわかる。つまり、ローカルには変数 `n` が存在している。

もう一度関数 f1() を実行すると新しい関数が得られる。

```
>ff2=f1();
() {
    return n++;
  }
>ff();
2
>ff2();
0
```

クロージャの例 – 無名関数をその場で実行する

関数 f1 を一度だけ実行して、それがこれ以上実行されないようにするためにはこの関数を無名関数としてその場で実行すればよい。

```
let foo = (function () {
  let n=0;
  return function() {
    return n++;
  };
})();
```

無名関数をその場で実行する (解説)

- 無名関数を定義した部分を () でくくり、引数リストをその後の () に記述する。ここでは、引数がないので中はない。
- 戻された関数オブジェクトを変数 foo に代入する。
- 前と同じように実行できる。

```
>foo();
0
>foo();
1
>foo();
2
```

ローカル変数の値を単純に返すと、不都合が起こる例

```
function f2() {
  let a = [];
  let i;
  for(i=0; i<3; i++) {
    a[i] = function() {
      return i;
    };
  }
  return a;
}

>funcs=f2()
[function a.(anonymous function)(), function a.(anonymous fun
    function a.(anonymous function)()]
>funcs[1]()
3
```

すべて同じ値を返す関数になってしまっている。

改良

この不具合は、for の制御変数 i を for 文の初期化のところで宣言すれば発生しない。

```
function f3() {
  let a = [];
  for(let i=0; i<3; i++) {
    a[i] = function() {
        return i;
      };
  };
  return a;
}
```

改良–従来の方法

```
function f3() {
  let a = [], i;
  for(i=0; i<3; i++) {
    a[i] = (function(x){
      return function() {
        return x;
      }
    })(i);
  };
  return a;
}
```

解説と実行

- 引数を取る無名関数を用意し、その場で与えられた引数を返す無名関数を返す関数を実行している。
- 仮引数には、実行されたときの i のコピーが渡されるので、その後変数の値が変わっても呼び出された時の値が仮引数に保持される。