JavaScript 講習会 #1

4/26 (土) @OUCC部室

一応, 自己紹介

- すしす
- Twitter: @susisu2413
- Flash (ActionScript 3)
- JavaScript
- Haskell (最近始めた)
- ベクターグラフィックス

予定

今日

- キソー

次回 (5月中旬以降?)

- 発展?
- 応用?

JavaScript #とは

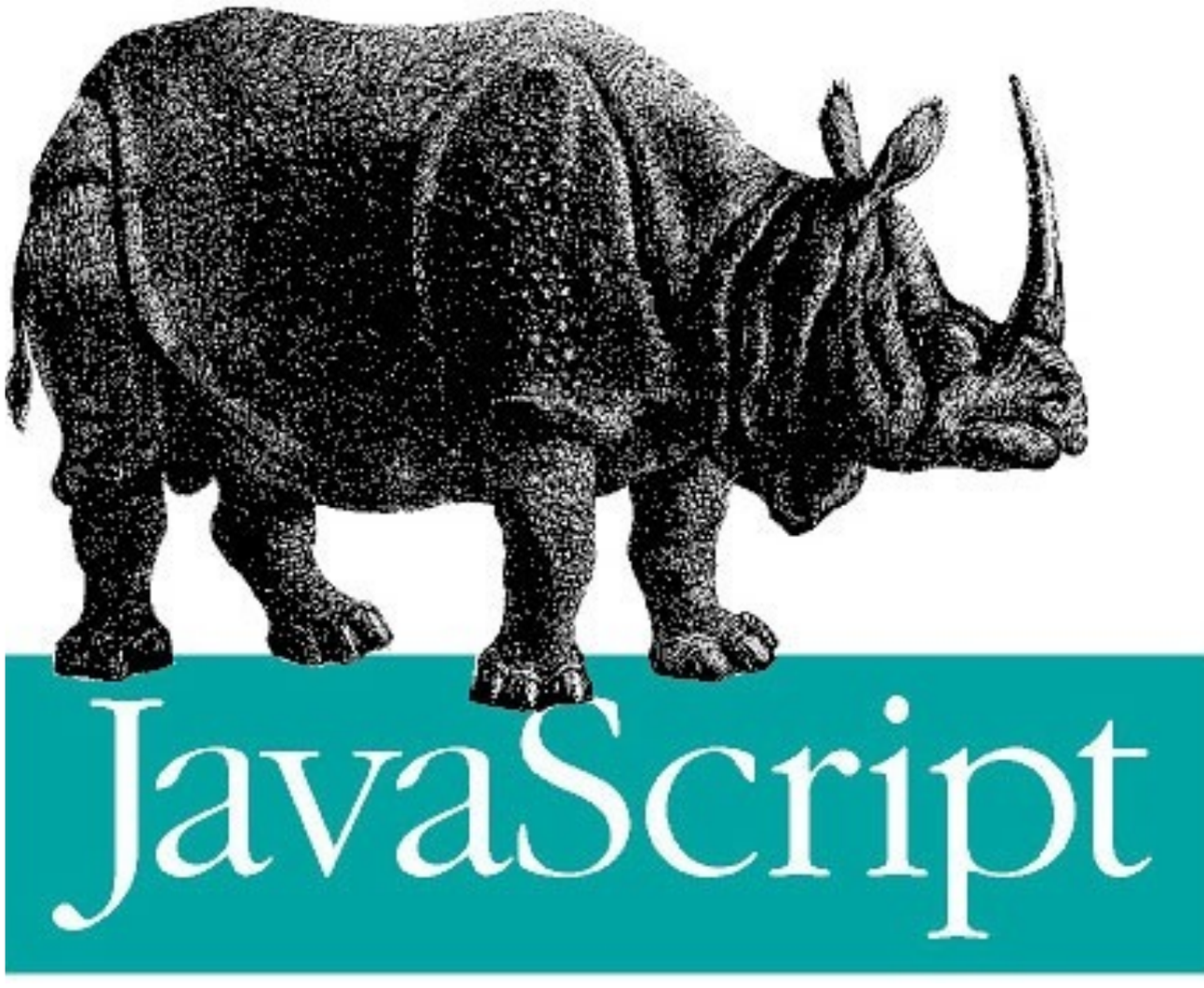
JavaScript #とは

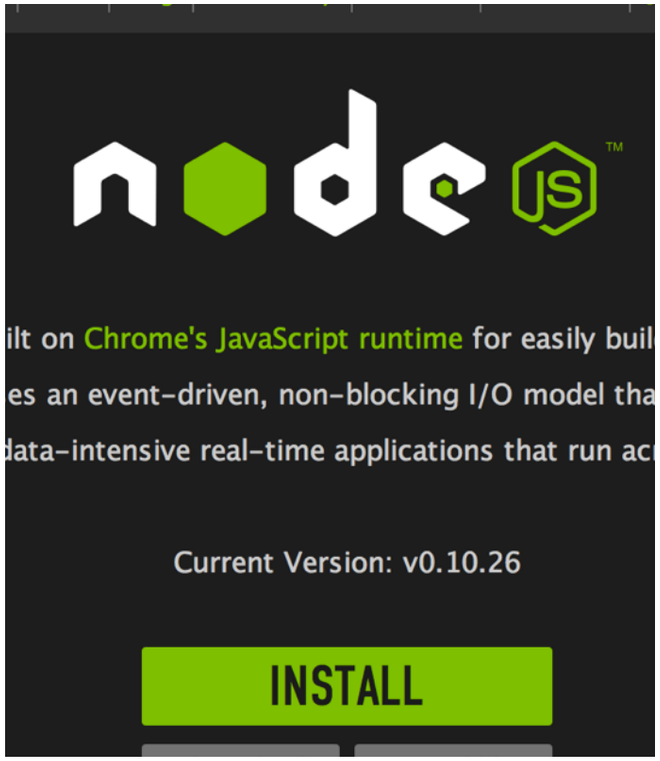
元は動的なWebページを作成するための言語

主な実行環境はWebブラウザ

Web ページのアニメーションから本格的なアプリケーションまで

最近ではアプリケーションの拡張やサーバーサイド開発でも





JavaScript を探せ! (探さなくても見つかる)

JavaScript #とは

Javaっぽい文法

“Java と JavaScript はインドとインドネシアくらい違う”

プロトタイプベースのオブジェクト指向

クラスベース (Java, C# など) とは少し扱いが異なる

第一級関数をサポート

関数型言語的な特徴

JavaScript #とは

- ECMAScript

  JavaScript の標準化規格, Ecma International によって策定

  JavaScript は ECMAScript の "方言" のひとつ

- この講習会では ECMAScript 5th を解説します

  最近の JavaScript 実装は大体 ES 5th に準拠しています

準備

1.Web ブラウザを起動

2.about:blank を開く

3.開発コンソールを開く

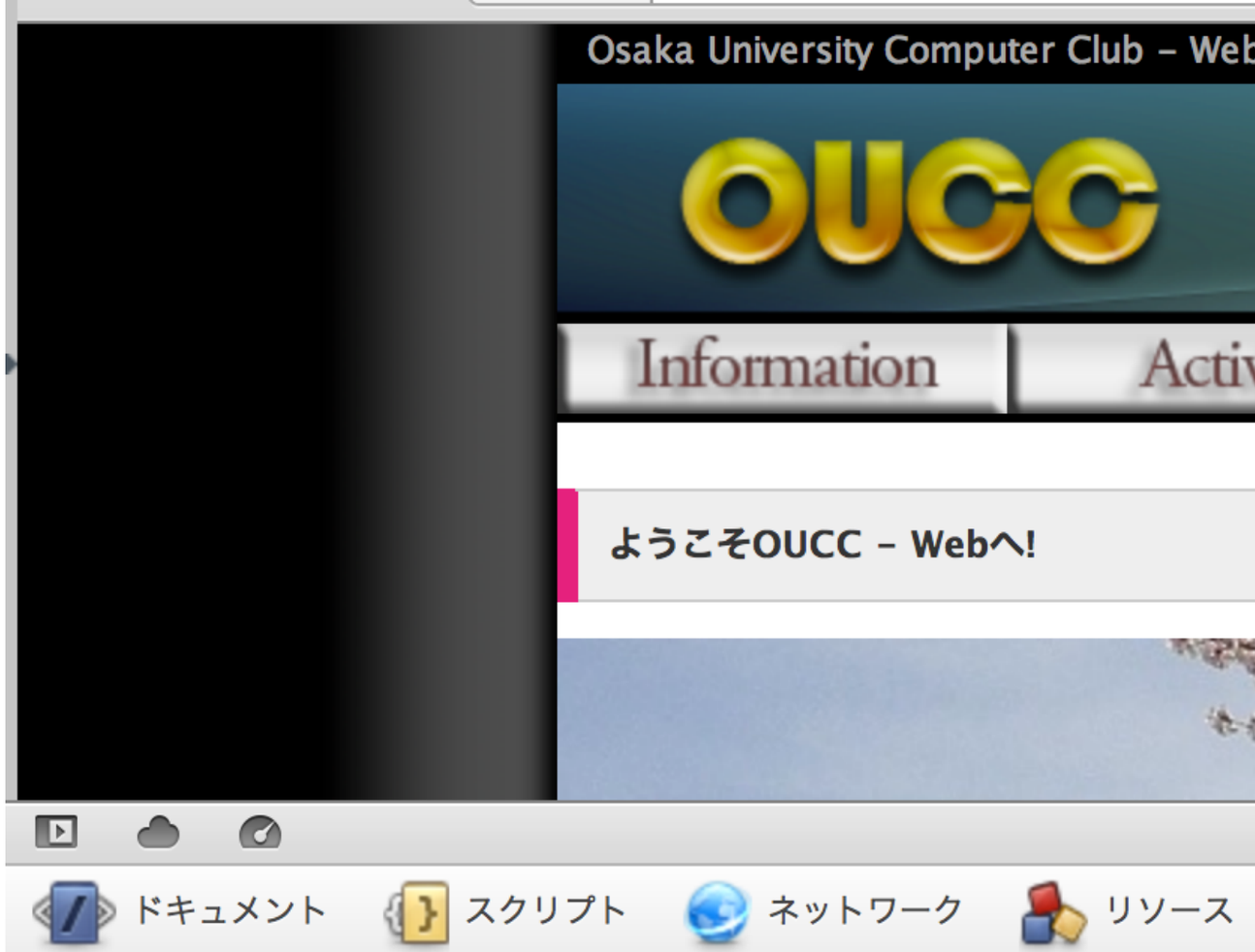
IE: F12

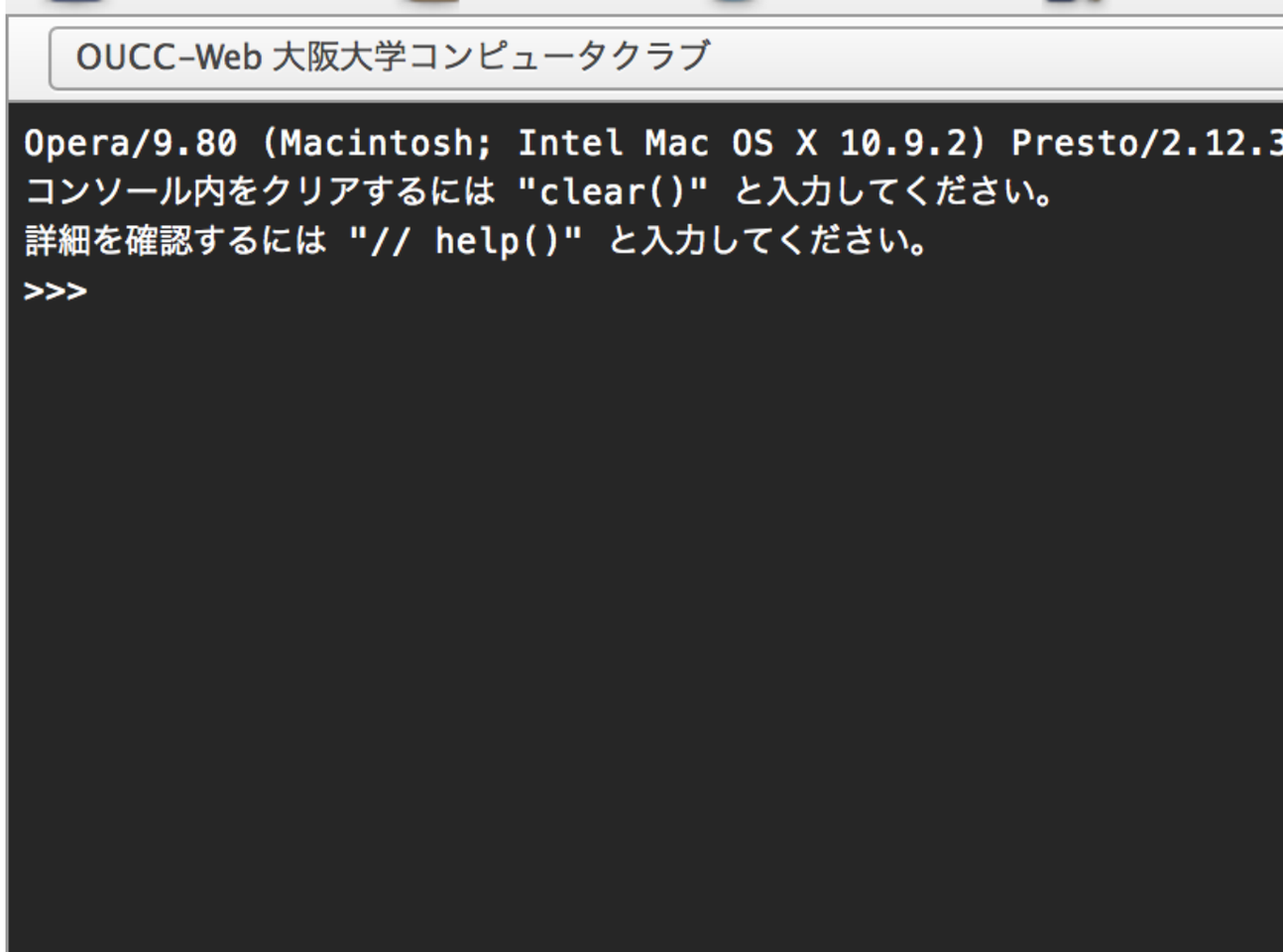
その他: Ctrl + Shift + I (⌥⌘I)

コンソール (Console) を選択

4.または Node.js

```
$ node
```





データ型と式

data types and expressions

数値 (Number)

- 実数値 (Infinity (無限大), NaN (非数) を含む)
- 四則演算 (+, -, *, /), 剰余 (%)
- ビット演算 (&, |, ^, ~ など)

C言語などと違い, 整数と浮動小数の扱いは曖昧です

```
>> 1
1
>> 2 + 3
5
>> 2.5 * 4 + 1
11
```

文字列 (String)

- "ダブルクォーテーション" や 'シングル (略)' で囲まれた文字列
- 連結は +
- "文字列"[n] で n文字目を取得できる (開始は 0)

```
>> "hoge"
"hoge"
>> "foo" + "bar"
"foobar"
>> "piyopiyo"[3]
"o"
```

真偽値 (Boolean)

- true (真) または false (偽)
- 論理演算 and (&&), or (||), not (!)

```
>> true && false
false
>> true || false
true
>> !true
false
```

配列 (Array)

- データを並べたもの

  [ ] 内にカンマ (,) 区切りで記述

- 数値, 文字列, 真偽値など, 何でも入れられる

- n番目の要素を取得するには 配列[n] (開始は 0)

```
>> [1, 2, 3, 4, 5]
>> [1, "nya", true]
```

オブジェクト (Object)

- ハッシュテーブル (名前 - 値 の対応)

  { 名前1: 値1, 名前2: 値2, ...} のように記述

- 値の取得は オブジェクト.名前

```
>> ({ name: "Meu Meu", chikuwa: 1000 })
>> ({ "foo": "bar" })
```

※ ( ) で括らないとブロック (後述) と判断される場合があります

その他のデータ型

- 関数 (Function)

  関数も1つのデータとして扱う (第一級関数) (後述)

- Null

  値は null のみ　空のオブジェクトを表す ({ } とは違うよ!)

- 未定義値 (undefined)

  undefined

- etc.

余談: JSON

- JavaScript Object Notation

  JavaScript のデータ記法のサブセット

  数値, 文字列, 真偽値, 配列, オブジェクト, Null

- データのやりとりで使用

  XML (冗長) を置き換えつつある

  Twitterとか


```
{
    "contributors_enabled": false,
    "created_at": "Wed Sep 16 11:18:30 +0000 2009",
    "default_profile": false,
    "default_profile_image": false,
    "description": "勉強教えてください",
    "entities": {
        "description": {
            "urls": []
        },
        "url": {
            "urls": [
                {
                    "display_url": "susisu.ktkr.net",
                    "expanded_url": "http://susisu.ktkr.net/",
                    "indices": [
                        0,
                        22
                    ],
                    "url": "http://t.co/CCL0co8h3R"
                }
            ]
        }
    },
    "favourites_count": 5782,
    "follow_request_sent": false,
    "followers_count": 497,
    "following": false,
    "friends_count": 560,
    "geo_enabled": false,
    "id": 74706872,
    "id_str": "74706872",
    "is_translation_enabled": false,
    "is_translator": false,
    "lang": "ja",
    "listed_count": 53,
    "location": "明後日",
    "name": "すしす",
    "needs_phone_verification": false,
    "notifications": false,
    "profile_background_color": "00BFFF",
```

関数の使用

- 関数名(引数1, 引数2, ...)
- 関数の例
  - 絶対値 Math.abs, 冪乗 Math.pow, 平方根 Math.sqrt
  - 小数点以下切り捨て Math.floor, 切り上げ Math.ceil
  - 四捨五入 Math.round
  - コンソール出力 console.log
  - etc.

関数の使用

```
>> Math.abs(-1)
1
>> Math.sqrt(Math.pow(3, 2) + Math.pow(4, 2))
5
>> console.log("avocado")
"avocado"
undefined
```

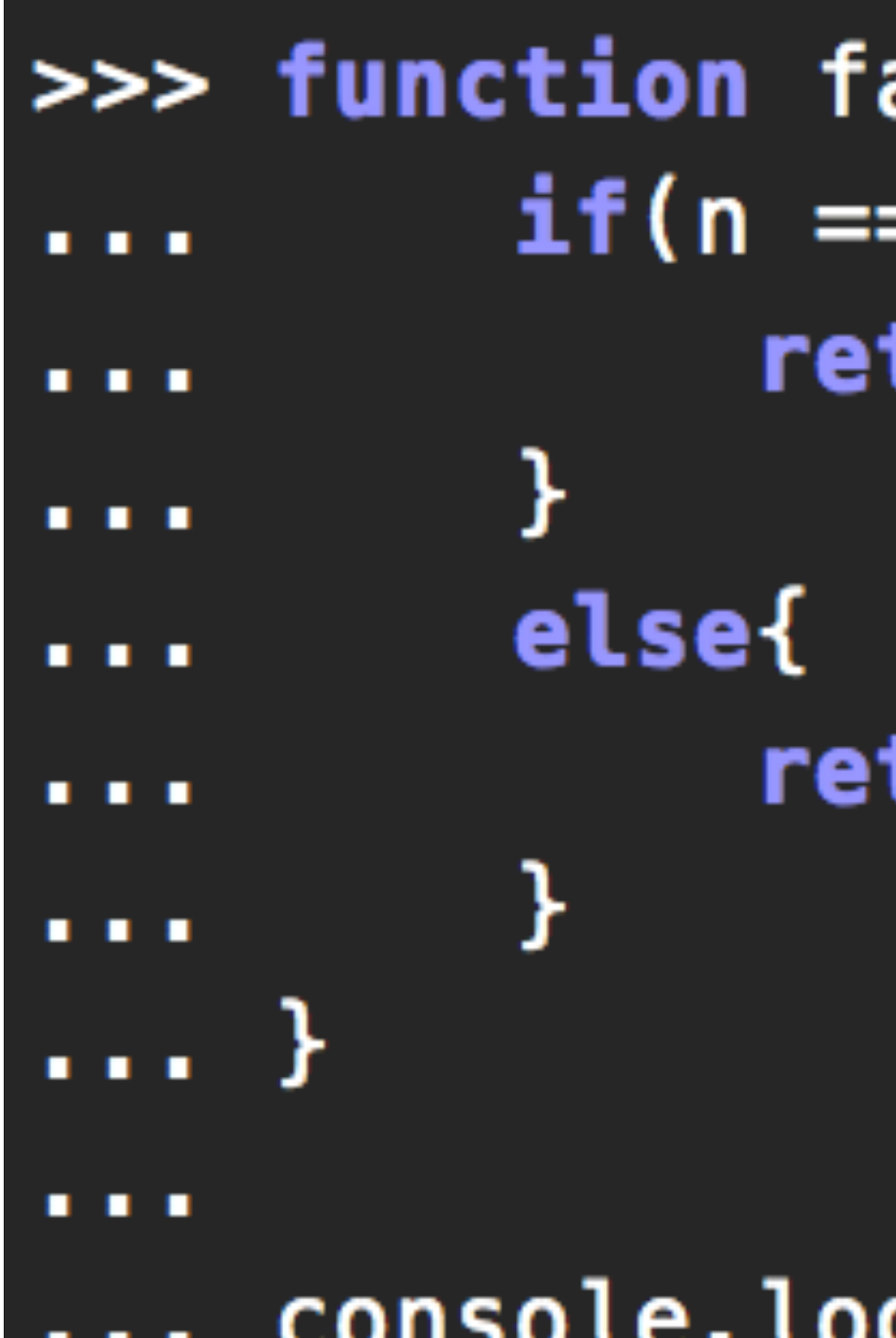
※ console.log は 引数をコンソールに出力して, undefined を返す

変数と文

variables and statements

複数行入力

- Shift + Enter で改行 (もしくは複数行入力モードになる) (多分)
- 今までどおり Enter または Ctrl + Enter (⌘Enter) で実行
- 今後複数行やインデントでコード例を示すことがありますが, 別に一行で入力しても問題無いです (見づらいけど)



用語の解説

- 式

  大体ここまでに解説したもの

  データとその演算結果, 関数の適用結果など

- 文

  式, 今から解説する文, もしくはそれらを複数個並べたもの

  式からなる文や変数宣言は後ろに ; (セミコロン) をつける (べき)

コメント

- プログラムとして実行されない部分
- // から改行まではコメントになります
- /* 〜 */ の間はコメントになります

変数

- var 変数名 = 値;

  変数に型はない (どんなデータでも代入可能)

- 値の更新は 変数名 = 新しい値

```
>> var x = 1;
>> x
1
>> x = 2
2
>> x + 3
5
```

変数

```
>> var arr = [1, 2, 3, 4, 5];
>> arr[0]
1
>> arr[1] = -2
-2
>> arr
[1, -2, 3, 4, 5]
>> arr[5] = 6
6
>> arr
[1, -2, 3, 4, 5, 6]
```

変数

```
>> var obj = { foo: 100, bar: { baz: 200} };
>> obj.foo
100
>> obj.bar.baz
200
>> obj.foo = 10
10
>> obj.foo
10
>> obj.bar.hoge = 20
20
>> obj.bar.hoge
20
```

条件分岐 (if, else)

- 条件が成り立つときに文を実行する

  if(条件){ 文 }

- 条件が成り立つときに文1, 成り立たない時に文2を実行

  if(条件){ 文1 } else{ 文2 }

条件分岐 (if, else)

- 条件1が成り立てば文1

- 条件1が成り立たず, 条件2が成り立てば文2

- 条件が共に成り立たなければ文3

  を実行

if(条件1){ 文1 } else if(条件2){ 文2 } else{ 文3 }

条件に便利な演算子

- 比較演算子

- 等価比較とその否定

    ==, !=

- 大小比較

    <, <=, >, >=

もしもし, 5ですか

```
var x = 10;
if(x == 5){
    console.log("x is 5");
}
else{
    console.log("x is not 5");
}

// => "x is not 5"
```

もしもし, どこにいますか

```
var x = 1;
if(x < 0){
    console.log("x is negative");
}
else if(x > 0){
    console.log("x is positive");
}
else{
    console.log("x is 0 (or not a number)");
}

// => "x is positive"
```

while ループ

- 条件が成り立つ間, 何回も文を実行
- while(条件){ 文 }
- ループ内で途中で抜け出すには break;

while ループの例

```
var x = 0;
while(x < 10){
    console.log(x);
    x = x + 1;
}
```

```
// => 0, 1, 2, ..., 9 を順に出力
// (最後に 10 が出力されるかもしれませんが,
// それは console.log によるものではありません)
```

for ループ

- while ループに加えて, 初期化と進行部分が存在
- for(初期化; 条件; 進行){ 文 }
- 初期化部分が1度だけ実行された後, 条件の評価, 文, 進行が繰り返される
- while と同じく, 途中で抜けるには break;

for ループの例

```
for(var x = 0; x < 10; x = x + 1){
    console.log(x);
}
// => 0, 1, 2, ..., 9 を順に出力
```

```
for(var y = 0; y < 100; y = y + 1){
    if(y * y >= 1000){
        break;
    }
    console.log(y);
}
// => 100 以下の非負整数 y を順に出力
// y * y が 1000 を超えた時点で終了
```

ありきたりな練習コーナー

---

1. 自然数 n の階乗を求めてみよう

2. n 番目のフィボナッチ数を求めてみよう

1の解答例

```
var n = 10; // 好きな自然数
var factorial = 1; // 階乗の結果が入る変数
for(var i = 1; i <= n; i = i + 1){
    factorial = factorial * i; // 順番に掛けていく
}
console.log(factorial);
// => 3628800
```

2の解答例

```
var n = 10; // 好きな自然数
var a = 1; // フィボナッチ数列の第1項
var b = 1; // フィボナッチ数列の第2項
if(n == 1){
    console.log(a);
}
else{
    for(var i = 2; i < n; i = i + 1){
        var t = b; // b の元の値を確保
        b = a + b;
        a = t;
    }
    console.log(b);
}
// => 55
```

関数

functions

関数の定義

- function 文

- function 関数名(引数1, 引数2, ..., 引数N){ 文 }

- 関数の返り値は return 式; (同時に関数の処理を終了)

- 標準の関数と同じように使用可能

  関数名(引数1, 引数2, ...)

関数定義の例

- 1つ引数をとって, 2倍した値を返す関数

```
function double(x){
    return x * 2;
}

// 使用してみる
console.log(double(5)); // => 10
```

関数定義の例

- 2つ引数をとって, それらを足した値を返す関数

```
function addTwo(x, y){
    return x + y;
}

// 使用してみる
console.log(addTwo(2, 3)); // => 5
```

関数定義の例

- 引数が偶数なら2で割り，奇数なら3倍して1を足す関数

```
function collatz(x){
    if(x % 2 == 0){
        return x / 2;
    }
    else{
        return 3 * x + 1;
    }
    console.log("return で処理が終わるので，これは出力されない");
}
```

関数の特徴

- 関数内で定義した変数は, その関数内でしか使用できない (スコープ)

```
function doNothing(){
    var localX = 100;
}

console.log(localX); // => エラー
```

関数の特徴

- 関数は function 文の前後どちらでも使用可能

```
console.log(addTwo(1, 2)); // => 3

function addTwo(x, y){
    return x + y;
}

console.log(addTwo(3, 4)); // => 7
```

関数定義の例

- 階乗

- 無駄な変数が関数の外に出現しない

```
function factorial(n){
    var result = 1;
    for(var i = 1; i <= n; i = i + 1){
        result = result * i;
    }
    return result;
}
console.log(factorial(10)); // => 3628800
```

再帰関数

- 関数の内部で, その関数自身を呼び出す

- 例えば階乗は 0! = 1, n! = n * (n - 1)! と定義されている

⇒ 再帰関数で定義可能

関数定義の例

- 階乗 (再帰関数版)

- 数学的な定義そのまま

```
function factorial(n){
    if(n == 0){
        return 1;
    }
    else{
        return n * factorial(n - 1);
    }
}
console.log(factorial(10)); // => 3628800
```

例題コーナー

- n番目のフィボナッチ数を求める関数を定義してみよう

フィボナッチ数は $a_1 = 1$, $a_2 = 1$, $a_n = a_{n-1} + a_{n-2}$ で定義される

オーキドのことば「こういうものには

- ただしnが大きくなると実行速度がひどく遅い

```
function fibonacci(n){
    if(n == 1){
        return 1;
    }
    else if(n == 2){
        return 1;
    }
    else{
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
console.log(fibonacci(10)); // => 55
```

オブジェクト指向

object-oriented programming

JavaScript はオブジェクト指向言語

- データはすべてオブジェクト (として扱える)
  - 数値も
  - 文字列も
  - 配列も
  - 当然オブジェクトも
  - (null と undefined はちょっと扱いが異なるけど)

プロパティとメソッド

- オブジェクト (ハッシュテーブル) に格納された値にはオブジェクト.名前 でアクセスできた

- 同じように全てのデータ型について値を格納・アクセスできる (ただし数値, 文字列, 真偽値には格納は不可)

- あるデータ型について、そのデータ型ならばアクセスできる値を持つことがある

  もし関数でなければプロパティ、関数ならばメソッドと呼ぶ

配列のプロパティとメソッド

- length プロパティ

  配列の長さ (正確には最後の要素のインデックス + 1)

```
>> [1, 2, 3, 4].length
4
```

- concat メソッド

  配列同士を結合

```
>> [1, 2, 3, 4].concat([5, 6, 7])
[1, 2, 3, 4, 5, 6, 7]
```

- etc.

文字列のプロパティとメソッド

- length プロパティ

  文字列の長さ

```
>> "this is a pen".length
13
```

- substr メソッド

  部分文字列を取り出す

```
>> "this is a cat".substr(2, 5)
"is is"
```

- etc.

で、何が嬉しいの

- 例えば「長さを測れるもの」の長さの2倍を計算したい
- 長さを測れるものはいくつもある
  - 配列
  - 文字列
  - 関数の長さは引数の数?
  - 他にも長さを測れるものがあるかも?

で、何が嬉しいの

- もしオブジェクト指向じゃなければ

```
function lengthDoubled(something){
    if(something instanceof Array){
        return arrayLength(something) * 2;
    }
    else if(something instanceof String){
        return stringLength(something) * 2;
    }
    else if(something instanceof Function){
        return functionLength(something) * 2;
    }
    else if(something instanceof ...){ ...
```

で、何が嬉しいの

- オブジェクト指向ならば

```
function lengthDoubled(something){
    return something.length * 2;
}
```

- データ (長さを測れるもの) と特有の値もしくは手続き (長さ) がセットになっている

余談: オブジェクト指向じゃない言語でもこのへんを解決する仕組みは存在します

ぼくらはみんな文字になる　文字になるから

- toString メソッド

  全てのデータ (Object) は toString メソッドを持つ

```
>> "string".toString()
"string"
>> (243).toString()
"243"
>> true.toString()
"true"
>> [1, 2, 3, 4].toString()
"1, 2, 3, 4"
```


らきすたはここまで