



# JAVASCRIPT 入門

鄭祥飛

2021/04/14

2021©ROCOCO



# 文 ( statement )

```
Let test = " aa" ;
```

1. セミコロンの区切り。1行に複数「文」がある場合、セミコロンが必須。
2. 大文字と小文字区別。
3. Unicodeを使用

ソースコードテストを左から右にスキャンされ、制御文字、改行文字、コメント解析して実行します。



# 特別コメント

```
#!/usr/bin/env node
```

スクリプトの実行に使用したい特定の  
JavaScript エンジンへのパスを指定する  
のに使用される特殊なコメントです。



# 変数宣言

変数の名前は識別子とも呼ばれ、下記のルールが有る

- 1．先頭文字：文字、アンダースコア、ドル
- 2．続く文字に、数字が可能
- 3．大文字小文字区別

宣言方法3種類がある：

VAR：ローカル或いはグローバル変数

LET：ローカル変数

CONST：読み取り専用。例外：`const myArray=[]; const myObj = {};`

未指定：例、`x = 4 2 ;` 未宣言のグローバル変数を生成する。おすすめしない。

# 変数評価

- 1 . 変数に初期値を設定する。
- 2 . 未設定場合、直接使用されると、undefinedとなる

```
let myArray = [];
```

```
console.log(myArray[0]);
```


- 3 . 異常の回避ために、if(test !== undefined) { doSomething();}
4. undefined === false; 結果 : TRUE
5. Null === false ;結果 : TRUE
6. null===0 ;結果 : TRUE
7. 0===false ;結果 : TRUE



# 変数のスコープ

```
if (true) { var x = 5; }  
console.log(x); // x は 5
```

```
if (true) { let y = 5; }  
console.log(y);  
// ReferenceError: y が定義されていない
```



# データ型一覧

- プリミティブ型のデータ型が7つあります。
  - Boolean (真偽値)。 `true` または `false`。
  - null。 null 値を意味する特殊なキーワードです。(JavaScript は大文字・小文字を区別するため、 `null` は `Null` や `NULL` などと同じではありません。)
  - undefined (未定義)。値が未定義の最上位プロパティです。
  - Number (数値)。整数または浮動小数点数。例えば `42` や `3.14159` など。
  - BigInt (長整数)。精度が自由な整数値。例えば `9007199254740992n` など。
  - String (文字列)。テキストの値を表す連続した文字。"Howdy" など。
  - Symbol (シンボル) (ECMAScript 2015 の新機能)。インスタンスが固有で不変となるデータ型。
- そして Object (オブジェクト)。

# BigInt

1 . NUMBER:-9007199254740991 から 9007199254740991までの数字

```
console.log(999999999999999999);
```

Number.MAX\_SAFE\_INTEGERとNumber.MIN\_SAFE\_INTEGER

2 . BigIntの作成

```
9007199254740995n ;
```

```
BigInt("9007199254740995");
```

```
console.log(10n === 10); // → false
```

```
console.log(10n == 10); // → true
```

3 . BigIntの計算

```
+10n; // → TypeError: Cannot convert a BigInt value to a number
```

```
BigInt(10) + 10n; // → 20n
```

```
10 + Number(10n); // → 20
```

```
BigInt(10.2); // → RangeError
```

```
BigInt(null); // → TypeError
```

```
BigInt("abc"); // → SyntaxError
```

```
BigInt(true); // → 1n
```





# 数字


0, 117, -345, 123456789123456789n (10進数)

015, 0001, -0o77, 0o777777777777777n (8進数)

0x1123, 0x00111, -0xF1A7, 0x123456789ABCDEFn (16進数)

0b11, 0b0011, -0b11, 0b11101001010101010101n (2進数)

16進数：大文字・小文字の違いは値には影響しません。





# データ型の変換

- 1 . 文字列から数字への変換  
parseInt(): 小数点を切り捨てる  
parseFloat()  
(+ ' 1.1' )



# 文字列

- 1 . 文字列の長さ
- 2 . 文字列に変数を挿入。

```
var name = 'Bob', time = 'today' ;  
`Hello ${name}, how are you ${time}?`
```





# 配列

- 1 . 配列の長さ
- 2 . [ , 'aaa' , ' ' ]
- 3 . [ , 'aaa' , ' ' ]



# 対象（オブジェクト）

プロパティ名とそれに関連付けられたオブジェクトの値  
との 0 個以上の組が波括弧 ({} ) で囲まれたもので作られたリストです

**例：**

```
var car = { myCar: 'Saturn', getCar: carTypes('Honda'), special: sales };
```

**対象の使用：**

```
car.myCar  
Car[' myCar' ]
```



# 算術演算子

演算子	説明	例
<a href="#">剰余</a> (%)	二項演算子です。2 つの被演算子で除算したときの、整数の余りを返します。	12 % 5 は 2 を返します。
<a href="#">インクリメント</a> (++)	単項演算子です。被演算子に 1 を加えます。前置演算子 (++x) として用いると、被演算子に 1 を加えた後にその値を返します。後置演算子 (x++) として用いると、被演算子に 1 を加える前にその値を返します。	x が 3 の場合、++x は x に 4 を設定して 4 を返します。一方、x++ は 3 を返したあと x に 4 を設定します。
<a href="#">デクリメント</a> (--)	単項演算子です。被演算子から 1 を引きます。戻り値はインクリメント演算子のものと同様です。	x が 3 の場合、--x は x に 2 を設定して 2 を返します。一方、x-- は 3 を返したあと x に 2 を設定します。
<a href="#">単項符号反転</a> (-)	単項演算子です。被演算子の符号を反転して、その値を返します。	x が 3 のとき、-x は -3 を返します。
<a href="#">単項プラス</a> (+)	単項演算子です。数値でない被演算子の数値への変換を試みます。	+ "3" は 3 を返します。 + true は 1 を返します。
<a href="#">べき乗演算子</a> (**) 	基数部を指数部乗したものを計算します、つまり、 <b>基数部</b> <sup>指数部</sup> となります。	2 ** 3 は 8 を返します。 10 ** -1 は 0.1 を返します。

# 比較演算子

<a href="#">等価 (==)</a>	被演算子が等しい場合に true を返します。	<pre>3 == var1 "3" == var1  3 == '3'</pre>
<a href="#">不等価 (!=)</a>	被演算子が等しくない場合に true を返します。	<pre>var1 != 4 var2 != "3"</pre>
<a href="#">厳密等価 (===)</a>	被演算子が等しく、かつ同じ型である場合に true を返します。 <a href="#">Object.is</a> や <a href="#">JavaScript での等価</a> も参照してください。	<pre>3 === var1</pre>
<a href="#">厳密不等価 (!==)</a>	被演算子が等しくなく、かつ/または同じ型でない場合に true を返します。	<pre>var1 !== "3" 3 !== '3'</pre>
<a href="#">より大きい (&gt;)</a>	左の被演算子が右の被演算子よりも大きい場合に true を返します。	<pre>var2 &gt; var1 "12" &gt; 2</pre>
<a href="#">以上 (&gt;=)</a>	左の被演算子が右の被演算子以上である場合に true を返します。	<pre>var2 &gt;= var1 var1 &gt;= 3</pre>
<a href="#">より小さい (&lt;)</a>	左の被演算子が右の被演算子よりも小さい場合に true を返します。	<pre>var1 &lt; var2 "2" &lt; 12</pre>
<a href="#">以下 (&lt;=)</a>	左の被演算子が右の被演算子以下である場合に true を返します。	<pre>var1 &lt;= var2 var2 &lt;= 5</pre>

# 論理演算子

MDN Web Docs

演算子	使用法	説明
<a href="#">論理積 (AND)</a> ( <code>&amp;&amp;</code> )	<code>expr1 &amp;&amp; expr2</code>	<code>expr1</code> を <code>false</code> と見ることができる場合は、 <code>expr1</code> を返します。そうでない場合は <code>expr2</code> を返します。したがってブール値を用いた場合、両被演算子が <code>true</code> であれば <code>&amp;&amp;</code> は <code>true</code> を返し、そうでなければ <code>false</code> を返します。
<a href="#">論理和 (OR)</a> ( <code>  </code> )	<code>expr1    expr2</code>	<code>expr1</code> を <code>true</code> と見ることができる場合は、 <code>expr1</code> を返します。そうでない場合は <code>expr2</code> を返します。したがってブール値を用いた場合、どちらかの被演算子が <code>true</code> であれば <code>  </code> は <code>true</code> を返し、両方とも <code>false</code> であれば <code>false</code> を返します。
<a href="#">論理否定 (NOT)</a> ( <code>!</code> )	<code>!expr</code>	単一の被演算子を <code>true</code> と見ることができる場合は、 <code>false</code> を返します。そうでない場合は <code>true</code> を返します。





# 条件演算子

書き方：条件？ 値1： 値2

例：var status = (age >= 18) ? 'adult' : 'minor';



# ビット演算子

演算子	使用法	説明
<a href="#">ビット論理積 (AND)</a>	$a \ \& \ b$	被演算子の対応するビットがともに 1 である各ビットについて 1 を返します。
<a href="#">ビット論理和 (OR)</a>	$a \   \ b$	被演算子の対応するビットがともに 0 である各ビットについて 0 を返します。
<a href="#">ビット排他的論理和 (XOR)</a>	$a \ ^ \ b$	被演算子の対応するビットが同じ各ビットについて 0 を返します。 [被演算子の対応するビットが異なる各ビットについて 1 を返します。]
<a href="#">ビット否定 (NOT)</a>	$\sim a$	被演算子の各ビットを反転します。
<a href="#">左シフト</a>	$a \ << \ b$	2進表現の $a$ を $b$ ビット分だけ左にシフトします。右から 0 で詰めます。
<a href="#">符号維持右シフト</a>	$a \ >> \ b$	2進表現の $a$ を $b$ ビット分だけ右にシフトします。溢れたビットは破棄します。
<a href="#">ゼロ埋め右シフト</a>	$a \ >>> \ b$	2進表現の $a$ を $b$ ビット分だけ右にシフトします。溢れたビットは破棄し、左から 0 で詰めます。

# ビット演算子

演算子	説明	例
<a href="#">左シフト (&lt;&lt;)</a>	この演算子は、第1被演算子を指定したビット数分だけ左にシフトします。左に溢れたビットは破棄されます。0のビットを右から詰めます。	$9 \ll 2$ の結果は 36 になります。1001 を 2 ビット左にシフトすると 100100 になり、これは 36 となるからです。
<a href="#">符号維持右シフト (&gt;&gt;)</a>	この演算子は、第1被演算子を指定したビット数分だけ右にシフトします。右に溢れたビットは破棄されます。左端のビットのコピーを左から詰めます。	$9 \gg 2$ の結果は 2 になります。1001 を 2 ビット右にシフトすると 10 であり、これは 2 となるからです。同様に、 $-9 \gg 2$ の結果は、符号が維持されるため -3 になります。
<a href="#">ゼロ埋め右シフト (&gt;&gt;&gt;)</a>	この演算子は、第1被演算子を指定したビット数分だけ右にシフトします。右に溢れたビットは破棄されます。0のビットを左から詰めます。	$19 \ggg 2$ の結果は 4 になります。10011 を 2 ビット右にシフトすると 100 になり、これは 4 となるからです。非負数では、0埋め右シフトと符号を維持した右シフトは同じ結果になります。

# 代入演算子

代入	<code>x = y</code>	<code>x = y</code>
加算代入	<code>x += y</code>	<code>x = x + y</code>
減算代入	<code>x -= y</code>	<code>x = x - y</code>
乗算代入	<code>x *= y</code>	<code>x = x * y</code>
除算代入	<code>x /= y</code>	<code>x = x / y</code>
剰余代入	<code>x %= y</code>	<code>x = x % y</code>
べき乗代入	<code>x **= y</code>	<code>x = x ** y</code>
左シフト代入	<code>x &lt;&lt;= y</code>	<code>x = x &lt;&lt; y</code>
右シフト代入	<code>x &gt;&gt;= y</code>	<code>x = x &gt;&gt; y</code>
符号なし右シフト代入	<code>x &gt;&gt;&gt;= y</code>	<code>x = x &gt;&gt;&gt; y</code>
ビット論理積 (AND) 代入	<code>x &amp;= y</code>	<code>x = x &amp; y</code>
ビット排他的論理和 (XOR) 代入	<code>x ^= y</code>	<code>x = x ^ y</code>
ビット論理和 (OR) 代入	<code>x  = y</code>	<code>x = x   y</code>
論理積代入	<code>x &amp;&amp;= y</code>	<code>x &amp;&amp; (x = y)</code>
論理和代入	<code>x   = y</code>	<code>x    (x = y)</code>
Null 合体代入	<code>x ??= y</code>	<code>x ?? (x = y)</code>



# 特別

## Delete:

演算子はオブジェクトやオブジェクトのプロパティ、配列の指定されたインデックスの要素を削除します。

## typeof


演算子は、未評価の被演算子の型を指す文字列を返します

## IN

指定したプロパティが指定のオブジェクトにある場合に **true** を返します。構文は以下のとおりです。

## instanceof

指定されたオブジェクトが指定されたオブジェクトの種類である場合に **true** を返します。





# グループ化演算子

グループ化演算子 ( ) は式内での評価の優先順位を制御します。  
例えば、加算が最初に評価されるよう、最初に行われる演算を  
乗算と除算から加算と減算へと上書きすることができます。



# 優先順位

次の表では演算子の優先順位を、高いものから低い順に並べています。

メンバー	<code>· []</code>
インスタンスの呼び出し/作成	<code>() new</code>
否定/インクリメント	<code>! ~ - + ++ -- typeof void delete</code>
乗算/除算	<code>* / %</code>
加算/減算	<code>+ -</code>
ビットシフト	<code>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</code>
関係	<code>&lt; &lt;= &gt; &gt;= in instanceof</code>
等値	<code>== != === !==</code>
ビット論理積	<code>&amp;</code>
ビット排他的論理和	<code>^</code>
ビット論理和	<code> </code>
論理積	<code>&amp;&amp;</code>
論理和	<code>  </code>
条件	<code>?:</code>
代入	<code>= += -= *= /= %= &lt;&lt;= &gt;&gt;= &gt;&gt;&gt;= &amp;= ^=  = &amp;&amp;=   = ??=</code>



# 宿題

1. 果物で値段を検索する。

