第J章 JavaScript 超簡単入門

JavaScript (ECMAScript) の基本をごくごく簡単に説明する。

J.1 JavaScript の基本

変数

JavaScript には型チェックはないので、変数の宣言に型の情報は必要なく、 `var` というキーワードで変数を宣言する。

```
var i = 0;
```

2015 年の ECMAScript 6 (ES6) からはさらに `let`, **`const`** という 2 つのキーワードが追加された。このうち、 **`let`** はブロックスコープの局所変数を宣言する。つまり、**`let`** というキーワードで宣言された変数は、その周りのブレース ({ 〜 }) の間でのみ有効である。

一方、**`const`** はブロックスコープの代入不可の変数を宣言する。つまり、代入して値を変えようとすると、実行時にエラーになる。

演算子

次のような演算子 +, −, *, /, %, ++, −−, =, +=, == の意味は C 言語や Java とほぼ同じである。また「+」演算子は文字列の連接にも使用できる。

`console.log` メソッド

コンソール画面（後述）にメッセージを出力するためには `console.log` というメソッドを使う。引数の数は決まっておらず、与えられた引数を順に出力する。

```
1  console.log("hello, world", 1024);
```

テンプレートリテラル

テンプレートリテラル (template literal) は、途中に式を挿入することができる文字列定数である。途中で改行して複数行にまたがることもできる。文字列をバッククォート「`」で囲む。

```
1  console.log(`Hello!
2 World!
3 Good-Bye!`);
```

なお、バッククォート自体を含めるにはバックスラッシュ「\」に続けてバッククォートを書く（「\`」）。

```
  console.log(`
   ∧ ∧
/(´Д\`) ノ
        |
`);
```

テンプレートリテラルの中には「${式}」という形を含むことができる。この式の値が文字列に変換されて、この場所に埋込まれる。

```
  const x = 3;
  console.log(`2 × x = ${2 * x}`);
```

制御構造

条件判断 （**if** 文）, 繰返し （**while** 文, **for** 文, **do** ～ **while** 文）はほとんど C 言語や Java と同じである。ただし **for** 文は C や Java と違う形もあるので後で紹介する。

関数の定義

関数の定義も C 言語と良く似ているが、JavaScript では戻り値の型を書く必要がないので、C 言語で関数の戻り値の型を書く部分に、キーワード _____ を用いるところだけが異なる。また、仮引数の型を宣言する必要もない。return 文の書き方も C 言語と同じである。

```
// JavaScript
function cube(n) {
  return n * n * n;
}
```

```
/* （参考）C 言語 */
double cube(double n) {
  return n * n * n;
}
```

もう一つ C 言語と異なるところとして、JavaScript では、関数定義のなかに別の関数を定義することができる。

匿名関数

JavaScript でも無名の関数を定義することができる。JavaScript では次のような形を用いる。

function （変数$_1$, …, 変数$_n$） { 宣言・文の並び }

つまり、**function** というキーワードと括弧の間に関数名がない。

ES 6 から匿名関数をさらに短く書けるアロー関数という構文が用意された。（厳密に言うと、単なる構文の違いではなく、this というキーワードの指すオブジェクトが異なる、など意味の違いもある。）

分類	一般形	補足説明
アロー関数	（変数$_1$, …, 変数$_m$） => { 文$_1$ … 文$_n$ }	
	（変数$_1$, …, 変数$_m$） => 式	

記号「=>」をアロー (arrow) と読んでいる。下の段の形は `(変数 1, …, 変数 m) => { return 式; }` の省略形である。また、仮引数が一つだけのときは周りの丸括弧を省略することができる `変数 => …`。逆に無引数のときは `() => …` のように中が空の丸括弧を書かなければいけない。

分割代入

ES6 から、代入の左辺に要素が変数になっている配列を書くことができるようになった。これを_____(destructuring assignment) という。対応する位置にある値が、変数に代入される。

```
  const cs = ['red', 'yellow', 'green', 'blue'];
  let [x, y, ...rest] = cs;  // x が 'red'、
  // y が 'yellow'、rest が [ 'green', 'blue' ]
  [y, x] = [x, y];           // x と y の入れ替え
```

ここで `...` は_____(rest parameters) という構文で、上の例では、`x`, `y` に割り当てられなかった、残りの配列の要素が `rest` という配列にまとめられる。

スプレッド構文

逆に `...` を配列の初期値が並ぶ中などで使う形を_____(spread syntax) と呼ぶ。 `...` の後に式を書いて、配列などの中に別の配列などを展開することができる。

```
  const fruits = ["apple", "banana", "orange"];
  const veges  = ["tomato", "lettuce"];
  const foods  = [...veges, "tofu", ...fruits];
```

この例では、`foods` の値は `["tomato", "lettuce", "tofu", "apple", "banana", "orange"]` になっている。

J.2 ジェネレーター

ECMAScript 6（2015 年）よりジェネレーター (generator) 関数という機構が導入された。ジェネレーター関数はコルーチンの一種 (stackless coroutine) を提供する。

ジェネレーター関数は `function*` というキーワードで定義される。

```
function* gfib() {
  let a = 1, b = 1;
  while (true) {
    yield a;
    [a, b] = [b, a + b];
  }
}
```

ジェネレーター関数の内部では `yield` というキーワードを使って値を生成する。

ジェネレーター関数を呼び出すと、すぐに関数内部のコードが実行されるのではなく、一旦、ジェネレーター (generator) オブジェクトが作られて返される。このジェネレーターオブジェクトの `next` メソッドを呼び出すと、ジェネレーター関数内部のコードが実行され、`yield` された値を `value` プロパティーとして持つオブジェクトを返す。さらに `next` メソッドを呼び出すと yield 式のつづきから実行が再開され、やはり、次の `yield` された値を `value` プロパティーとして持つオブジェクトを返す。例えば、次のようになる。

```
const gen = gfib();
console.log(gen.next().value);  // 1 を出力する
console.log(gen.next().value);  // 1 を出力する
console.log(gen.next().value);  // 2 を出力する
console.log(gen.next().value);  // 3 を出力する
```

ジェネレーターオブジェクトは `for` ～ `of` 文の中でも使うことができる。

```
for （変数 of 式） {
    文のならび
}
```

この `for` ～ `of` 文 はジェネレーターオブジェクト（より一般的には iterable オブジェクト）の `next` メソッドによって返されるオブジェクトの `value` プロパティーを変数に代入してループする。ジェネレーター関数の中のコードの実行が return するか、関数を抜けるとループを終了する。

```
for (const v of gfib()) {
    if (v > 100) break;
    console.log(v); // 1, 1, 2, 3, 5, 8 を出力する。
}
```

Q J.2.1 次の漸化式で定義される数列（Pell 数列というらしい）

$$
\begin{aligned}
a_1 &= 1 \\
a_2 &= 2 \\
a_n &= a_{n-2} + 2a_{n-1} \quad (n \geq 3)
\end{aligned}
$$

の項を順に yield するジェネレーター関数 `pell` を定義せよ。

次のコードで, 1, 2, 5, 12, 29, 70, 169, 408, 985, 2378, 5741, を順に出力することを確認せよ。

```
function* pell() {
  // ここを考える
}

for (const v of pell()) {
    if (v > 10000) break;
    console.log(v);
}
```

J.3 HTML タグ

HTML 文書の全体はだいたい次のような構造を持っている。

```
<!Doctype HTML>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8">
<title>…</title>
    ⋮
</head>
<body>
    ⋮
</body>
</html>
```

JavaScript は HTML の `<script type="text/javascript">` 〜 `</script>` というタグの間に書く。

```
<script type="text/javascript">
// ここに JavaScript のプログラムを書く。
</script>
```

あるいは、

```
<script type="text/javascript" src="nantoka.js">
</script>
```

で、別ファイル (*nantoka*`.js`) の JavaScript ソースを読み込める。なお `type="text/javascript"` は省略可能である。

この例でわかるように JavaScript ソースファイルの拡張子は通常 ＿＿ である。

この script タグは HTML 文書の `<head>` 〜 `</head>` の間、 `<body>` 〜 `</body>` の間のどちらにも書くことができる。

J.4 JavaScript プログラムのデバッグ

JavaScript はプログラムの実行前にエラーを見つけてくれないので、気軽に試せる反面、デバッグがしにくい、という短所がある。

問題があったときでも、実行が止まってしまって画面に何も表示されず、そのままでは手がかりが何も得られないことが多い。そういう場合はブラウザーの「コンソール」という画面を表示しておけば、実行時に表示されるエラーメッセージを見ることができる。「コンソール」の表示の仕方はブラウザーによって異なるが、Chrome の場合は、「⋮」－「その他のツール」－「デベロッパーツール」、 Firefox の場合は、「≡」－「その他のツール」 －「ウェブ開発ツール」、 Edge の場合は「…」－「その他のツール」－「開発者ツール」、で表示されるようである。

コンソール画面にメッセージを出力するためには `console.log` という関数を使う。また、`alert` という関数を呼び出すと、画面に警告ダイアログを開くことができる。（`alert` の場合、警告ダイアログを閉じるまで、次の処理に進まない。）

```

```

```
1 console.log("Hello!");
```

```
1 alert("Hello!");
```

これらの関数呼び出しをプログラム中に適宜挿入しておいて実行すると、どこまで実行されたか、どこで実行が止まっているか、などを確認することができる。

J.5 さらに詳しく知りたい人のために ...

文献 (ECMA 262) は、JavaScript (ECMAScript) の仕様書である。

> **(ECMA 262)** ECMA International “ECMAScript Language Specification” https://www.ecma-international.org/publications-and-standards/standards/ecma-262/
