付録 J　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、____ というキーワードで変数を宣言する。

```
var i = 0;
```

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。

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。

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。

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(n) {
  var a = 1, b = 1;
  while (a < n) {
    yield a;
    var tmp = b;
    b += a;
    a = tmp;
  }
}
```

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。

```
var gen = gfib(100);
console.log(g.next().value);  // 1 を出力する
console.log(g.next().value);  // 1 を出力する
console.log(g.next().value);  // 2 を出力する
console.log(g.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 (v of gfib(10)) {
   console.log(v); // 1, 1, 2, 3, 5, 8 を出力する。
}
```

J.3 HTML タグ

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。

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、Firefox の場合は、「ツール」－「Web 開発」－「Web コンソール」、Internet Explorer の場合は「ツール」－「F12 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。）

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

```
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 [1] は、JavaScript (ECMAScript) の仕様書である。

この章の参考文献

[1] ECMA International 「ECMAScript Language Specification」
http://www.ecma-international.org/publications/standards/Ecma-262.htm