JavaScript マニュアル
JavaScript 関数 から プロトタイプオブジェクト指向 まで
基礎 と 応用 編

1.JavaScript の関数

・関数を定義する3つの方法
・function 文による定義
・Function コンストラクタによる定義
・関数リテラルによる定義
・関数の引数として関数を引き渡す（高階関数）
・高階関数と匿名関数
・関数内部で利用できる特殊なオブジェクト（arguments オブジェクト）
・JavaScript は引数をチェックしない
・arguments オブジェクトの実体
・arguments オブジェクトによる可変個引数の関数宣言
・現在実行中の関数を参照する（callee プロパティ）

2.JavaScript のデータ型

・扱えるデータ型
・データ型の明示的な変換
・組み込みオブジェクトのコンストラクタ
・「＋演算子」「－演算子」

3.グローバル・スコープとローカル・スコープ

・ローカル変数は var キーワードで宣言する
・ローカル変数は宣言された関数全体で有効
・ブロック・レベルのスコープは存在しない
・匿名オブジェクトによる疑似ブロック・スコープ
・関数リテラルと Function コンストラクタにおけるスコープの違い
・仮引数のスコープと参照型
・グローバル変数とローカル変数の実体
・クロージャの仕組みを理解する
・オブジェクトのように振る舞うクロージャ
・イベント・ハンドラにクロージャを適用する

4.JavaScript でオブジェクト指向プログラミング

・JavaScript における“クラス”の定義
・コンストラクタとプロパティ
・メソッド － コンストラクタによる定義 －
・メソッド － インスタンスへの追加 －
・プロトタイプ・ベースのオブジェクト指向
・プロトタイプ・オブジェクトを介する利点
・undefined 値によるプロトタイプ・オブジェクトのメンバの無効化
・プロトタイプをオブジェクト・リテラルで定義する
・プロトタイプ・チェーン － JavaScript の継承機構 －

1.JavaScript の関数

JavaScript における関数とはそれ自体が「オブジェクト」であり、変数やオブジェクトのメンバとして格納したり、あるいは、引数としてほかの関数に引き渡したりすることも可能です。オブジェクトといって分かりにくければ、関数は JavaScript における「データ型」の一種であるといってもいいです。関数のこの特徴は、JavaScript プログラミングの柔軟性を支えるものであり、同時に、(JavaScript に不慣れな開発者にとっては)まず初めのとっつきにくさを感じさせる一因でもあります。

➢ 関数を定義する3つの方法

JavaScript で関数を定義するには、大きく3つの方法があります。

- **function 文による定義**
- **Function コンストラクタによる定義**
- **関数リテラルによる定義**

- **function 文による定義**

JavaScript で関数を定義する場合の最も基本的な構文。function 文による関数定義は以下のとおりです。

```
function 関数名([引数 1 [, 引数 2 [, ……]]) {
  [関数内で実行される任意の命令……]
}
```

キーワード「function」の後方に、関数名、そして引数を丸カッコでくくって指定する必要がある。なお、関数名は一般的な文字列や式ではなく、識別子(名前)である必要があります。また、関数の本体は、必ず中カッコ({ })でくくります。if／while などの命令ブロックでは、その配下に文が 1 つしかない場合、

```
if (x == 1) alert('変数 x は 1 です。'); // 中カッコの省略
```

のように中カッコを略記することが可能ですが、関数定義では配下の文が 1 つであっても「中カッコは省略できない」ので注意する必要があります(ただし、if／while などの命令ブロックでも、ブロックの範囲を明確にするという意味で、中カッコは常に記述するのが好ましい)。以下に、具体的な関数の例を記述します。以下の add 関数は、引数 x、y で与えられた数値を加算し、その結果を戻り値として返すものです。なお、関数から呼び出し元に戻り値を返すのは return 文の役割です。関数が戻り値を持たない場合、return 文は省略することもできます。

```
window.alert(add(5, 7)); // 「12」を表示

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

これはごく基本的な関数の例でもあり、特筆すべき点はないように見えるかもしれません。しかし、次のような例ではどうでしょうか。

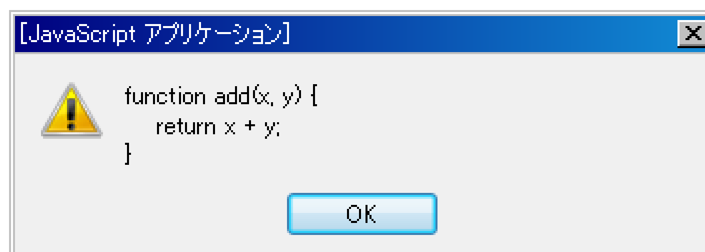
```
function add(x, y) {
  return x + y;
}

window.alert(add(5, 7)); // 「12」を表示

add = 0;  A
window.alert(add); // 「0」を表示  B

window.alert(add(5, 7)); // 「関数を指定してください」エラーが発生  C
```

恐らく、このコードを見た多くの方が違和感を持つはずです。add 関数と同名の変数が定義されたことに問題があるならば A でエラーが発生するはずだし、変数を評価する際に関数と変数が識別できないことに問題があるならば B でエラーが発生するはずです。しかし、エラーが発生するのは C です。この挙動は、JavaScript における関数の重要な性質を示しています。結論から言うと、JavaScript における関数とはオブジェクトであり、関数を定義するのは「＜関数名＞という名前（ここでは add）の変数に関数オブジェクトを格納している」のと同じ意味というわけです。

この視点でもう一度コードを見直してみると、JavaScript の動作が明確になると思います。最初の段階では、変数 add には関数オブジェクトがセットされます。これが A の代入式で上書きされ、数値 0 がセットされます。従って、B の時点では変数 add の最新の情報である 0 が表示されるようになりますが、その次の C では数値を「add(5, 7)」のような式で評価しようとしたためにエラーが発生した、とまあ、こういうわけです。これをもっとあからさまに確認しているのが、以下のコードです。

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

window.alert(add);
```

ここでは、以下の画面のように、関数定義がそのままダイアログ表示されることが確認できるはずです。ここからも、function 文によって変数に関数定義が格納されていることが理解できます。



ただし、function 文による関数定義が、いわゆる「=」演算子による代入式とは異なる点もあるので注意が必要です。例えば次のようなコードを見てみましょう。

```
window.alert(add(5, 7)); // 「12」を表示  A

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

「関数定義とは変数定義である」という理解と照らし合わせてみると、これまた、直感的には理解しがたいコードです。function 文がそのまま変数を定義しているならば、Aの時点ではまだ add 関数は定義されていないのでエラーが発生しなければならない。しかし、実際にはそうはならない。これは、厳密には、function 文は動的に実行される文ではなく、静的な構造を宣言するためのキーワードであるためです。静的な構造を宣言」とはどういうことかというと、コードが解析／コンパイルされるタイミングで、function 文は関数を定義してしまうということです。従って、実行時にはすでにコード内の構造として add 関数が登録されているものとして、どこからでも add 関数を呼び出すことができるのです。

● Function コンストラクタによる定義

関数がオブジェクトであるというならば、もしかしたら new 演算子で関数を宣言できるのではないのか。まさにそのとおりで、JavaScript では、確かに Function コンストラクタと new 演算子を用いることで関数を定義することが可能です。前述の add 関数を、Function コンストラクタで定義すると、以下のように記述できます。

```
var add = new Function(″x″, ″y″, ″return x + y″);
```

Function コンストラクタでは、任意の数の引数を指定することが可能です。末尾の引数を除くすべての引数は、関数に引き渡すべき引数を表し、そして、末尾の引数が関数の本体を表しています。ここでは関数本体が単一の文から構成される例を示していますが、文が複数存在する場合には、通常の関数定義と同様、セミコロン(;)で文を区切れば可能となっています。また、引数(ここでは x と y)は、以下のようにカンマ区切りでまとめて指定することも可能です。

```
var add = new Function(″x, y″, ″return x + y″);
```

もっとも、先の function 文を使わずに、あえて Function コンストラクタを利用するメリットは見えにくいかもしれません。シンプルに function 文を利用した方がコードもすっきりと見やすいのではないかと考えられると思います。しかし、Function コンストラクタには、標準の function 文にはない重要な特徴があります。というのも、Function コンストラクタでは関数の本体部分を文字列として指定できるという点です。このため、スクリプト内で文字列操作を行うことで、実行時に動的な関数の挙動を変更することができます。例えば、以下のコードは calc 関数を Function コンストラクタで定義する、ごくシンプルなコードです。

```
var ope = ″-″;
var calc = new Function(″x″, ″y″, ″return x″ + ope + ″y;″)

window.alert(calc(2, 3)); // 「-1」を表示
```

ここで注目してほしいのは、Function コンストラクタの末尾の引数と関数本体が変数 ope の値によって動的に生成されている点です。ここでは変数 ope の値を固定的に指定しているだけですが、条件に応じて変数 ope の値を切り替えることも可能です。ただし、ここで 1 つ注意しなければならないポイントとして気をつけなければならないことが出てきます。それは、「実行時に動的に関数の挙動を変更できる」ということは、Function コンストラクタでは function 文とは異なり、「関数が静的な構造として組み込まれるわけではない」ということです。例えば、以下のコードは、宣言と呼び出しを逆にした場合です。

```
window.alert(add(5, 7));  A
var add = new Function(″x″, ″y″, ″return x + y″);
```

一見、同じ処理をしているように見えますが、このコードはエラーとなります。A の段階では、まだ変数 add に関数オブジェクトがセットされていないためです。このことからも Function コンストラクタが実行時に評価されていることが分かると思います。ちなみに、同じ理由から、Function コンストラクタは for／while などのループ内、または頻繁に呼び出される関数内で使用するべきではありません。Function コンストラクタは、実行時に呼び出されるたびに新たな関数オブジェクトを生成するため、実行速度低下の原因となるためです。

● 関数リテラルによる定義

そして、関数を定義する 3 番目の手段が「関数リテラル」です。関数リテラルを使って、先ほどの add 関数定義を書き直してみると、以下のように記述できます。

```
var add = function(x, y) { return x + y; };
```

function 文を利用した最初の記法に似ていると思われるかもしれないが、いくつかの違いもあるので注目してください。まず、function 文では関数 add を直接に定義しているのに対して、関数リテラルでは「function(x, y) ……」と名前のない関数を定義したうえで、これを変数 add に格納している。関数リテラルは、宣言する時点では名前を持たないことから「無名関数」、または「匿名関数」と呼ばれる場合もあります。また、function 文は静的な構造を宣言するものであるのに対して、関数リテラルは式として使用される。つまり、function 文よりも柔軟性のある記述が可能になるわけです。

✓ 匿名関数定義における関数リテラルと Function コンストラクタの違い

宣言時に名前が必要ないという意味では、前述した Function コンストラクタも同様です。しかし、Function コンストラクタが文字列で関数本体を定義しなければならないことから記述が冗長になりやすいのに対して、関数リテラルは JavaScript の標準的な構文で記述でき、コードが読みやすいというメリットがあります。通常、匿名関数を定義するには、(Function コンストラクタではなく)関数リテラルを使用するのが好ましいのです。また、厳密には Function コンストラクタと関数リテラルとでは、関数解釈の挙動が異なる場合もあるので、注意が必要です。

➢ 関数の引数として関数を引き渡す（高階関数）

以上、JavaScript の関数がオブジェクトであり、変数にも自由に代入できることが理解できました。では、関数の引数として関数を引き渡すこともできるのではないかという点について話を進めます。JavaScript では、文字列や数値を引数としてセットするのとまったく同じ要領で、ある関数をそのほかの関数の引数としてセットすることが可能になっています。そして、このような関数のことを「高階（こうかい）関数」と呼びます。

● 高階関数の具体的な例

以下のコードで定義する arrayReduce 関数は、引数に与えられた配列（ary）の内容を、指定されたユーザー定義関数 func の規則に従って順番に処理し、最終的な結果を戻り値として返します。

```
function arrayReduce(ary, func) {

  var result = ary[0];

  for (i = 1; i < ary.length; i++) {
    result = func(result, ary[i]);
  }
  return result;
}

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

var myAry = [5, 3, 4, 7];
var result = arrayReduce(myAry, elementSum);

window.alert(result); // 「19」を表示
```

arrayReduce 関数の引数 func として引き渡している、配列処理のためのユーザー定義関数は elementSum 関数です。引数 func で引き渡される関数は、第 1 引数と第 2 引数とで何らかの演算を行い、その結果値を返さなければならない。

ここでは、elementSum 関数は与えられた 2 つの値を加算する処理を行うので、arrayReduce 関数はそれ全体として、配列要素すべての合計を算出するはずです。与えられた配列 myAry の要素合計 19（＝5＋3＋4＋7）が結果として返されることが確認できます。

もちろん、ユーザー定義関数は自由に入れ替えることが可能です。例えば、以下のコードは arrayReduce 関数に引き渡すユーザー定義関数を、elementSum 関数から elementMultiply 関数に置き換えたものです。

```
function arrayReduce(ary, func) {

  var result = ary[0];

  for (i = 1; i < ary.length; i++) {
    result = func(result, ary[i]);
  }
  return result;
}

function elementMultiply(x, y) {
  return x * y;
}

var myAry = [5, 3, 4, 7];
var result = arrayReduce(myAry, elementMultiply);

window.alert(result); // 「420」を表示
```

elementMultiply 関数では積算を行うため、arrayReduce 関数全体として今度は配列要素すべてを掛け合わせたものを算出することになり、結果を確認してみると、確かに配列 myAry の要素合計 420（＝5×3×4×7）が返されることが確認できる。

このように高階関数を利用することで、より汎用性の高い（機能の差し替えが容易な）関数を定義できることがお分かりいただけると思います。

- **高階関数と匿名関数**

実は、高階関数は先ほど登場した匿名関数と密接な関係を持っています。というのも、高階関数においては、引数として引き渡す関数が「その場限り」でしか利用されないことがあるためです。例えば、先ほどの例でもユーザー定義関数 elementSum／elementMultiply は arrayReduce 関数の処理を規定するために用意した関数であり、これをほかの個所で再利用する予定がないならば、あえて関数に名前を付ける必要がないことが直感的に理解できると思います。そのような場合には匿名関数を用いて、以下のように記述するのがいいでしょう。

```
function arrayReduce(ary, func) {

  var result = ary[0];

  for (i = 1; i < ary.length; i++) {
    result = func(result, ary[i]);
  }
  return result;
}

var myAry = [5, 3, 4, 7];

var result = arrayReduce(
  myAry,
  function(x, y) { return x + y; }
);

window.alert(result); // 「19」を表示
```

これが、先ほど関数リテラルは「function 文よりも柔軟性のある記述が可能」であると記述した意味です。匿名関数を用いることで、いわゆる「使い捨て」の関数定義を、高階関数を呼び出すコードにそのまま組み込むことができるので、よりシンプルにコードを記述できることがお分かりになるはずです。また、関連する処理が 1 カ所で記述できることから、コードの可読性も向上するというメリットもあります。

➢ 関数内部で利用できる特殊なオブジェクト（arguments オブジェクト）

JavaScript の関数を利用する場合に、もう 1 つ忘れてはならないトピックとして、arguments オブジェクトがあります。arguments オブジェクトは、関数の内部でのみ利用可能なオブジェクトで、関数に渡された引数値を管理することができます。

● JavaScript は引数をチェックしない

arguments オブジェクトは、具体的にどのような局面で利用すればよいのか。それを解説する前に、まずは以下のコードを見てください。

```
function display(msg) {
  window.alert(msg);
}

display();              // undefined  A
display('山田');         // 「山田」と表示  B
display('山田', '掛谷'); // 「山田」と表示  C
```

display 関数自体はごく単純なもので、引数 msg に与えられた文字列をそのままダイアログ表示するための関数です。この display 関数を、それぞれ引数 0、1、2 個を指定した形で呼び出してみる。通常の Visual Basic や C#のような言語に慣れている方にとっては、関数のシグニチャと正しく合致した呼び出し以外はエラーとなるのが、直感的に正しい挙動です。しかし、JavaScript ではそうはならない。いずれも正しく呼び出せてしまうのです。B が正しく動作するのは当然として、A では引数が与えられないため「undefined」（未定義）が、C では、多かった引数は（取りあえず）無視されて B と同じ結果を得ることができます。この結果から分かることは、JavaScript が「シグニチャ（データ型の概念がないので、要は引数の数）をチェックしない」という点です。とすると、想定していた数以上の引数が与えられた C のようなケースでは、「多すぎた引数」はただ切り捨てられてしまうだけなのでしょうか。

そこで、登場するのが、arguments オブジェクトです。JavaScript では関数が呼び出されたタイミングで内部的に arguments オブジェクトが生成され、呼び出し元から渡された変数を格納します。この arguments オブジェクトを利用することで、（例えば）関数に与えられた引数の数をチェックすることも可能になります。

- **arguments オブジェクトの実体**

厳密には、JavaScript は関数呼び出しのタイミングで、ローカル変数や引数情報を、Activation Object（通称、「Call オブジェクト」とも呼ばれる）のプロパティとして格納しています。arguments オブジェクトも、その実体は Call オブジェクトの arguments プロパティです。

Call オブジェクトは、アプリケーション側から明示的に生成したり呼び出したりすることはできないし、通常は意識することすらない存在であるので、本稿ではただ単に「arguments オブジェクト」と呼ぶものとします。以下のコードは、先ほどの display 関数に引数の個数チェックを加えたものです。

```
function display(msg) {
  if (arguments.length == 1) {
    window.alert(msg);
  } else {
    window.alert('引数の数が正しくありません。');
  }
}

display();            //「引数の数が正しくありません。」を表示
display('山田');       //「山田」を表示
display('山田','掛谷'); //「引数の数が正しくありません。」を表示
```

arguments オブジェクトは、通常の配列オブジェクトと同様に length プロパティを公開しており、これにより配列に含まれる要素数（ここでは関数に実際に渡された引数の数）を取得することができます。

ここでは length プロパティを確認して、その値が 1 以外である場合（引数の数が 1 個でない場合）、エラー・ダイアログを表示するようにしています。同様に、arguments オブジェクトの中身を確認することで、引数のデータ型や値の妥当性などを確認し、関数内部の処理で予期せぬエラーを未然に防ぐような処理も可能になります。

もっとも、このような arguments オブジェクトによる引数の妥当性チェックは、Visual Basic や C#などの世界ではコンパイラが自動で行ってくれるもので、JavaScript では自分で実装しなければならないのは、むしろデメリットにも感じられます。

- **arguments オブジェクトによる可変個引数の関数宣言**

しかし、arguments オブジェクトにはもう 1 つ便利な（そして重要な）使い方があります。それが可変個引数の関数宣言です。可変個引数の関数とは、（あらかじめ指定された個数の引数ではなく）任意の数の引数を与えられる関数のこと。宣言時に引数の個数を特定できないような関数を定義したい場合に、利用することができます。例えば、以下では引数に与えられた任意の数の数値の平均値を求める average 関数を定義してみることにします。

```
function average() {

  var sum=0;

  for (i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }
  return sum / arguments.length;
}

window.alert(average(10, 20, 30, 40)); // 「25」を表示
```

ここでは for ループの中で arguments オブジェクトからすべての要素を取り出し、その合計値を要素数で除算しています。arguments オブジェクトから個々の要素にアクセスする場合も、通常の配列と同様に「arguments[i]」のように記述すればアクセスが可能となります。

ここでは、すべての引数を名前を持たない引数として扱っているが、もちろん、一部の引数は通常の名前付き引数として明示的に宣言しておき、その後方に名前なし引数を持たせるようなことも可能です。

```
function format(fmt) {
  for (i = 1; i < arguments.length; i++) {
    var reg = new RegExp("¥¥{" + (i - 1) + "¥¥}", "g")
    fmt = fmt.replace(reg,arguments[i]);
  }
  return fmt;
}

var dat = new Date();

window.alert(
  format("今日は{0}年{1}月{2}日です", dat.getFullYear(), dat.getMonth() + 1, dat.getDate() )
); // （例えば）「2007 年 7 月 5 日」を表示
```

format 関数は、第 1 引数に与えられた書式に含まれるプレイスホルダ（{0}、{1}、{2}……）を第 2 引数以降の対応する文字列で置き換えた結果を返すための関数です（ .NET Framework ライブラリの String.Format メソッドに相当するもの ）。

ここで注目していただきたいのは、format 関数では第 1 引数 fmt を名前付き引数として指定しておき、第 2 引数以降を名前なし引数として定義している点です。前述したように、JavaScript では一部の引数を明示的に名前付き引数として宣言することも可能になっています。ただしこの場合も、arguments オブジェクトには、名前付き引数／名前なし引数の双方が「すべて」含まれているという点に注意してください。

従って、この場合、引数 fmt は「arguments[0]」としてもアクセスすることが可能です。そして、名前なし引数の値は「arguments[1]」以降に格納されることになります。ここでは、for ループで 2 番目の要素（インデックスは 1）から末尾の要素までを順番に取り出し、対応するプレイスホルダ（{0}、{1}、{2}……）との置き換え処理を行っているというわけです。

つまり、format 関数のような例でも、すべての引数を名前なし引数としてしまうことが構文上は可能です。しかし、コードの可読性を考慮した場合、すべてを arguments オブジェクトに委ねるのではなく、固定（名前付き）で宣言できる引数は明示的に宣言しておくのがいいでしょう。

● 現在実行中の関数を参照する（callee プロパティ）

最後に、arguments オブジェクトが公開している重要なプロパティとして、callee プロパティを紹介します。callee プロパティは現在実行中の関数を参照するためのプロパティです。具体的なコード例を見てみましょう。以下のコードは、再帰処理によって、与えられた数値の階乗（例えば 4 の階乗は 4 X 3 X 2X 1 ＝ 24）を求める factorial 関数を定義したものです。

```
function factorial(n) {
  if (n == 0) {
    return 1;
  } else {
    return n * arguments.callee(n – 1);
  }
}

alert(factorial(4)); // 24 を表示
```

再帰処理に際して、arguments.callee プロパティで自分自身を呼び出しているのが確認できるはずです。もっとも、この例であれば、あえて callee プロパティを使わなくても、シンプルに、

```
return n * factorial(n – 1);
```

と関数名を指定すればよいと思われるかもしれない。確かに、この場合は問題ない。しかし、関数が匿名関数の場合はどうだろう。再帰処理に際して呼び出すべき名前がないので、この場合は、callee プロパティを利用する必要があるというわけだ。

```
var factorial = function(n) {

  if (n == 0) {
    return 1;
  } else {
    return n * arguments.callee(n – 1);
  }
}
```

再帰処理は繰り返し構造を表現するうえで欠かせない初歩的な技術でもあり、callee プロパティはこの再帰処理を匿名関数とともに表現するうえで欠かせない機能です。以上、関数の基本的な構文と、関連する匿名関数や高階関数、arguments オブジェクトの概念とその使い方について紹介しました。

2.JavaScript のデータ型

➢ 扱えるデータ型

JavaScript は、変数の型を意識する必要のない言語ですが、これは変数を使用する際にデータ型を指定する必要がない（指定できない）というだけで、JavaScript がデータ型そのものを持たないというわけではありません。JavaScript の世界では、プログラマがそれと指定しなくても、代入された値に応じて、適切なデータ型が自動的に割り当てられるというだけなのです。そのため、基本的には開発者がデータ型を意識しなければならない局面というのはそれほど多くはないのですが、それでもまったく意識しなくてもよいというわけではありません。JavaScript で利用可能なデータ型は以下のとおりです。

データ型	基本型	数値型（number）
		文字列型（string）
		真偽型（boolean）
		特殊型（null／undefined）
	参照型	配列（array）
		オブジェクト（object）
		関数（function）

この表を見ても分かるように、JavaScript のデータ型は大きく「基本型」と「参照型」に分類でき、値がいずれの型に属するかによって振る舞いも異なるので、注意が必要です。基本型が変数に対して直接に値を格納するのに対して、参照型ではその参照値が格納されます。参照値とは、値を実際に格納しているメモリ上のアドレス（ポインタ）のことです。

・非配列型

```
var x = 10;
var y = x;

x = 20;
window.alert(y); // 10(コピーされた値を表示)
```

・配列型

```
var x = [0, 1, 2];
var y = x;

x[0] = 10; // 配列の先頭の要素を書き換え
window.alert(y); // 10,1,2(参照元の値を表示)
```

上記の JavaScript を実行するとわかると思いますが、配列型を指定するとデータが入っているアドレスだけが格納されるので配列の値を変更すると、それを参照している箇所も自動的に書き換わることになります（var キーワードは、変数を宣言するための命令です。var キーワードを使わず直接記述することも可能ですが、特別な理由がない限り変数の宣言では var をつけてください。詳しくはスコープを参照）。

➢ データ型の明示的な変換

繰り返しですが、JavaScript という言語は型についてとかく寛容な言語です。例えば、次のようなコードも何ら問題なく動作してしまいます。

```
var x = 10;  // 数値型
var y = "2"; // 文字列型

window.alert(x * y); // 20
```

このようなコードでも「＊演算子」の前後は数値であろうと推測し、JavaScript が文字列（変数 y）を自動的に数値に変換したうえで演算を行ってくれています。これは、JavaScript のスクリプト言語としてのシンプルさを支える特徴です。しかし、このような寛容さは時として思わぬ不具合をもたらす一因にもなります。次に、以下のようなコードを見てみましょう。

```
var x = 10;  // 数値型
var y = "2"; // 文字列型

window.alert(x + y); // 表示結果は？
```

ここで、多くの方は結果として「12」が返されることを期待すると思いますが、結果は「102」になります。オペランドの片方が文字列である場合、「＋演算子」は（加算演算子ではなく）文字列連結演算子と見なされ、変数 x と y とが文字列として連結されることになるためです。このように、型に対する寛容さは、時として予期せぬ結果を得ることにもなるわけです。そこで、JavaScript ではデータ型を明示的に変換する方法も提供しています。ここでは代表的な例として、組み込みオブジェクトのコンストラクタを利用する方法と、「＋演算子」「－演算子」を利用する方法について紹介します。

✓ 組み込みオブジェクトのコンストラクタ

JavaScript では String、Number、Boolean のような組み込みオブジェクトが用意されています。これらのコンストラクタを呼び出すことで、明示的に対応する型に値を変換することができます。例えば、次は変数 y を Number オブジェクトに格納することで、明示的に数値に変換している例です。

```
var x = 10;
var y = "2";
var yy = new Number(y);

window.alert(x + yy); // 12
```

ちなみに、これらコンストラクタは関数として呼び出すことも可能です。

```
var yy = Number(y);
```

のように書き換えれます。

✓ 「＋演算子」「－演算子」

数値⇔文字列間の変換を行うならば、＋演算子あるいは「－演算子」を利用した明示的な変換を行うことも可能です。以下のコードは、変数 x（数値）、変数 y（文字列）をそれぞれ文字列、数値に変換する例です。なお、コード内で使用している typeof 演算子は、指定された変数の内部データ型を文字列で返すためのものです。

```
var x = 10;
window.alert(typeof x); // number

var xx = x + "";
window.alert(typeof xx); // string

var y = "2";
window.alert(typeof y); // string

var yy = y - 0;
window.alert(typeof yy); // number
```

前述したように、＋演算子は与えられたオペランドのいずれかが文字列である場合に、もう片方も文字列に自動変換したうえで文字列連結を行うという性質を持っています。逆に、その性質を利用して、数値 x を強制的に文字列に変換することも可能というわけです。また、－演算子を利用することで「文字列→数値」の変換も可能となります。

ちなみに、ここで「0」を減算して変換できるならば、「0」を加算しても「文字列→数値」変換を行えるのではないかと思っているならば、それは不可能です。JavaScript で、＋演算子は加算演算子と文字列連結演算子という 2 つの意味を持っているということを考えれば、オペランドの片方が文字列である場合、＋演算子は（加算演算子ではなく）文字列連結演算子として解釈されてしまう点は、やや混乱しやすいポイントでもあるので、注意してください。

3.グローバル・スコープとローカル・スコープ

変数を利用する場合、その変数のスコープ（Scope）を意識することは重要です。スコープとは、プログラム中での変数の有効範囲のことを意味します。JavaScript のスコープは、大きく「グローバル・スコープ」と「ローカル・スコープ」とに分類できます。グローバル・スコープとはプログラム中のどこからでも有効な変数の範囲を、ローカル・スコープとはその変数が宣言された関数の中でのみ有効な変数の範囲を表すスコープのことです。また、グローバル・スコープに属する変数のことを「グローバル変数」、ローカル・スコープに属する変数のことを「ローカル変数」と呼びます。JavaScript だけではなく、多くのプログラミング言語で共通する考え方です。

```
var num = 1;

function localFunc() {
  var num = 0;
  return num;
}

window.alert(localFunc()); // 0
window.alert(num);         // 1
```

上のコードを実行すると、グローバル・スコープとローカル・スコープの違いがわかると思います。JavaScript では、関数の外で定義された変数はグローバル変数、関数内部で定義された変数はローカル変数と見なされます。同名であっても、スコープの異なる変数は異なるものと見なされる点に注目してください。グローバル変数とローカル変数の基本的な挙動について、いくつか注意するべき点があります。以下は、JavaScript で変数を扱う場合に知っておくべきスコープにかかわるポイントをまとめたものです。

✓ローカル変数は var キーワードで宣言する

```
num = 1;

function localFunc(){
  num = 0;
  return num;
}

window.alert(localFunc()); // 0
window.alert(num);         // 0
```

JavaScript では変数宣言における var キーワードは省略可能です。しかし、**JavaScript では var キーワードが省略された場合、その変数はグローバル変数と見なされる**ということに気をつけてください。

この例の場合、関数内で宣言された変数 num もグローバル変数と見なされ、最初に定義されたグローバル変数 num の値を上書きしてしまうというわけです。結果として、上書きされたグローバル変数 num の値「0」が出力されることになるわけです。もっとも、関数内部で意図せずにグローバル変数を書き換えてしまうのは（当然）好ましいことではありません。**関数内では原則として、すべての変数を var キーワード付きで定義する**ようにしておかなければいけません。

var キーワードを付けておくことで、それが変数の宣言であることが視覚的にも確認しやすくなるし、そもそもグローバル変数では var キーワードを付けない、ローカル変数では var キーワードを付ける、というのもかえって間違いのモトとなるので、**基本的には「すべての変数宣言は var キーワード付きで行う」**ことを心がけてください。

✓ローカル変数は宣言された関数全体で有効

先ほど、ローカル変数は「宣言された関数内で有効である」と述べましたが、より厳密には「宣言された関数＜全体＞で有効である」というのが正確です。このやや不可思議な特徴を理解するために、以下のコードを見てみましょう。

```
var num = 1;

function localFunc() {
  window.alert(num); // 表示結果は？  A
  var num = 0; B
  return num;
}

window.alert(localFunc()); // 0
window.alert(num);         // 1
```

このコードは先ほどのものに A の部分を追加したものです。さて、ここで問題です。A の段階で出力される値はいくつになっているでしょうか。

まだ、ローカル変数 num の宣言 B が実行される前なので、グローバル変数 num の値である「1」が表示されると考えた方、残念ながらハズレです。

繰り返しますが、JavaScript でのローカル変数は「宣言された関数＜全体＞」で有効となります。つまり、この場合、ユーザー定義関数 localFunc の中で定義されたローカル変数 num が、関数の先頭から有効になることになります。しかし、A の時点ではまだ var 命令が実行されていないので、ローカル変数 num は初期化されていないことになります。

ということで、先ほどの問題の答えは「undefined（未定義）」となります。これは JavaScript のやや分かりにくい挙動でもあり、時として思わぬ不具合の原因となる可能性もあります。これを避けるという意味で、ローカル変数は関数の先頭で宣言するということを心がけてください。

✓ブロック･レベルのスコープは存在しない

Visual Basic や C#、C++のような言語とは異なり、JavaScript ではブロック･レベルのスコープが存在しない点にも要注意です。例えば、以下に C#によるごく基本的なコードを挙げてみます。

```
// C#
for (int i = 0; i < 10; i++) {
  // ブロック内の処理
}
Response.Write(i); // エラー  A
```

JavaScript 以外の言語では、for ブロック内で定義された変数 i はブロック配下でのみ有効であるので（ブロック･レベルのスコープを持つため）、ブロック外（ここでは A）で変数 i を参照しようとすると、コンパイル時にエラーが発生します。 しかし、JavaScript で同様のコードを記述すると結果が異なります。以下は、コードを JavaScript で書き直したものです。

```
for (var i = 0; i < 10; i++) {
  // ブロック内の処理
}
window.alert(i); // 表示結果は？  A
```

Visual Basic や C#、C++など、ブロック･レベルのスコープを持つ言語に慣れた方にとっては、直感的に A の部分で実行時エラーが返されると思われるかもしれませんが、JavaScript での実行結果は「10」が出力されます。繰り返しになりますが、JavaScript ではブロック･レベルのスコープが存在しないために、for ループを抜けた後も変数 i の値はそのまま保持されるのです。

もっとも、意図せぬ変数の競合を防ぐという意味でも、スコープは一般的にできるだけ必要最小限にとどめるのが好ましいです。そのような場合には、JavaScript でも、匿名関数というものを利用することで、ブロック・レベルのスコープを疑似的に表現することが可能になっています。

```
( function() {
  for (var i  =0; i < 10; i++) {
    // ブロック内の処理
  }
} ) () ;

window.alert(i); A
```

太字の部分はやや見慣れない構文であるかもしれないが、要は以下のコードと同じ意味です。

```
var f = function() {
  for (var i = 0; i < 10; i++) {
    // ブロック内の処理
  }
};
f();
```

JavaScript の関数とはそれ自体がオブジェクトであるので、このように匿名関数自体を式の一部として利用するような書き方もできてしまうのです。ちなみに、匿名関数の記述は開発時に関数単体の動作を確認したいという場合にも有効な機能です。関数定義全体をカッコでくくり、その末尾に「();」や「(引数);」のように動作に必要な情報を付与すればいいので、よりシンプルに記述できます。匿名関数の中で定義された変数は、ローカル変数として認識されるので、外部とは隔離することができます。意味のない匿名関数の定義が直感的に分かりにくいという難点こそあるものの、確かにAの個所で「'i'は宣言されていません」という実行時エラーが発生し、スコープが隔離されていることが確認できます。

✓ 匿名オブジェクトによる疑似ブロック・スコープ

匿名関数を用いた疑似ブロック・スコープの記法を紹介したが、そのほかにも with ブロックと匿名オブジェクトを利用して、疑似ブロック・スコープを定義することも可能です。

```
with ({i:0}) {
  for (i = 0; i < 10; i++) {
    // ブロック内の処理
  }
}
window.alert(i);
```

with 命令は、ブロック内で共通して利用するオブジェクトを指定するものです。with 命令を利用することで、ブロック内で指定されたオブジェクトのメンバにアクセスする場合、「オブジェクト名.メンバ名」ではなく「メンバ名」で直接記述することが可能になります。この例では、「i = 0」は実際には「obj.i = 0」（変数 obj はオブジェクト変数とする）を表しているわけだ。この違いは、「function」が「with」に置き換わっただけだが、意味的にはより直感的に理解しやすいコードとなっていると思います。通常のスクリプトであればこちらの匿名オブジェクト構文を、パフォーマンスを強く意識したい局面では匿名関数構文を、というような使い分けをするといいと思います。

✓ 関数リテラルと Function コンストラクタにおけるスコープの違い

関数リテラル、Function コンストラクタは構文的な違いこそあるものの、いずれも匿名関数を定義するという意味でほぼ同様の機能を提供します。ただし、関数内で入れ子に関数リテラル／Function コンストラクタを使用した場合には、スコープの解釈が互いに異なる点に要注意です。

```
var num = 0;

function scope() {
  var num = 1;

  // Function コンストラクタ
  var myScope1 = new Function("", "window.alert(num);");

  // 関数リテラル
  var myScope2 = function() { window.alert(num); };

  myScope1(); // 0
  myScope2(); // 1
}
scope();
```

結果を見ても分かるように、Function コンストラクタはグローバル変数 num を参照しているのに対して、関数リテラルがローカル変数 num を参照しています。直感的には混乱しやすい挙動です。

✓ 仮引数のスコープと参照型

仮引数とは、関数に引き渡される変数のことです。仮引数に指定した変数は、通常、ローカル変数として処理されます。

```
var x = 1; A

function ref(x) {
  x++;
  return x;
}

window.alert(ref(10)); // 11 B
window.alert(x);      // 1  C
```

ここでは、まずAでグローバル変数 x に「1」が代入され、次にBによって関数 ref が呼び出されるわけであるが、その内部で使用されている仮引数 x はローカル変数の扱いであるので、これをいくら操作してもグローバル変数 x には 影響を及ぼさない。実際、ここではローカル変数 x にまず「10」を与え、その後、1 インクリメントしているわけであるが、Cの結果を見ても、もともとのグローバル変数 x には影響が及んでいないことが確認できる。

ここまでは比較的容易に理解できるところであるが、これに先ほども紹介した参照型が絡んでくると、(難しくはないが)やや混乱するおそれがあるので注意してもらいたい。
例えば、次のようなコードをみてみよう。

```
var x = [0, 1, 2]; // 配列 A

function ref(x) {
  x[0] = 10;
  return x;
}

window.alert(ref(x)); // [10,1,2] B
window.alert(x);      // [10,1,2] C
```

前述したように、JavaScript において配列は参照型であり、参照型の値は参照渡しされる。つまり、この場合、まずAでグローバル変数 x に配列が代入される。次にBで関数 ref が呼び出され、仮引数 x が生成される。繰り返しであるが、仮引数は自動的にローカル変数と見なされるので、ここでAのグローバル変数 x と仮引数(ローカル変数)x は別物である。しかし、値自体は参照渡しされているので、結果的に同一の値を参照しているわけだ。従って、関数の中で配列に対して操作を行った場合、その結果はグローバル変数にも反映されることになる(C)。

このような挙動は、参照型の概念を理解してさえいれば、難なく理解できるものであるが、グローバル変数/ローカル変数とも絡んでくると混乱のモトとなるので、理解のうえ整理しておきたいことだ。

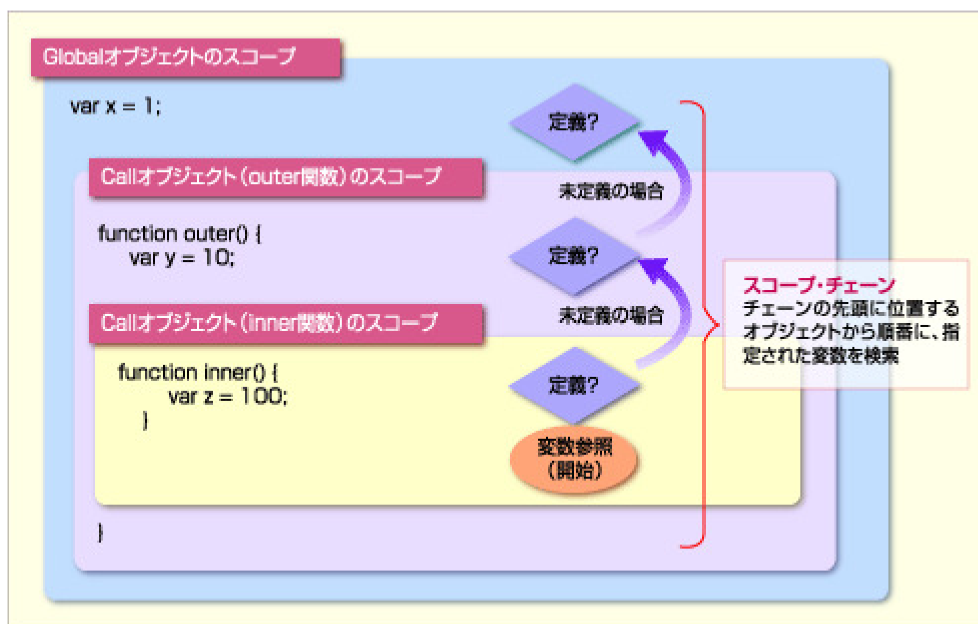
✓ グローバル変数とローカル変数の実体

JavaScript は関数呼び出しのタイミングで、ローカル変数（仮引数を含む）を、Activation Object（通称「Call オブジェクト」）のプロパティとして格納しています。とすると、グローバル変数も何らかのオブジェクトのプロパティなのではないかと思えてきた方はご明察。JavaScript は、JavaScript コードを実行するタイミングで、グローバル変数を「Global オブジェクト」のプロパティとして格納しています（同様に、JavaScript で提供されている eval や isNaN、isFinite のような関数も、その実体は Global オブジェクトのメソッドであると解釈できます）。

もっとも、Global オブジェクトは（先の Call オブジェクトがそうであったように）アプリケーション側から明示的に生成したり呼び出したりすることはできないし、通常は意識すらすることのない存在であるのだが、Global／Call オブジェクトを意識してみると見えてくることもあります。

繰り返しであるが、JavaScript はコードの実行時、関数の呼び出し時に、それぞれ新しい Global／Call オブジェクトを生成する。これらのオブジェクトを、呼び出しの順に連結したリストを「スコープ・チェーン」と呼びます。

JavaScript では、変数を解決する際に、このスコープ・チェーンの先頭に位置するオブジェクトから順にプロパティを検索して、合致したプロパティが見つかればその値を、見つからなければ次のオブジェクトを検索する仕組みになっています。



例えば、入れ子の関数内で記述された変数の場合、スコープ・チェーンには、先頭から、内部の Call オブジェクト、外部の Call オブジェクト、そして、Global オブジェクトが含まれているはずです。そこで、この順番でプロパティの有無が確認され、最初に見つかったところの値が返されるというわけです。また、最末尾の Global オブジェクトでも合致するプロパティが見つからなかった場合、その変数は未定義であると見なされます。

なお、スコープ・チェーンは、それぞれの実行コンテキスト（個々の関数の実行）ごとに形成されこれによって、JavaScript は実行コンテキストごとにローカル変数の独立性を保証しているわけです。

✓ **クロージャの仕組みを理解する**

スコープについて理解したところで、「クロージャ」について触れておくことにしよう。クロージャとは、ひと言でいうならば、「ローカル変数を参照している関数内関数」のことです。もっとも、これだけの説明ではなかなかイメージがわきにくいと思うので、具体的にクロージャを利用したコードを 1 つ挙げてみましょう。

```
function myClosure(init) {
  var cnt = init;

  return function() {
    return ++cnt;
  }
}

var result = myClosure(10); A

window.alert(result()); // 11 B
window.alert(result()); // 12 C
window.alert(result()); // 13 D
```

一見すると、myClosure 関数は引数 init を受け取り、これをインクリメントした結果を返しているように見えるかもしれない。しかし、ここで注目していただきたいのは、myClosure 関数が数値ではなく、関数を戻り値として返す高階関数であるという点です。

通常、myClosure 関数の中で定義されたローカル変数 cnt は、myClosure 関数の処理が終了した時点で破棄されるはずですが、この場合、myClosure 関数の戻り値である匿名関数がローカル変数 cnt を参照している。このため、「myClosure 関数の終了後もそのままローカル変数が保持される」というわけです（このような動作も、先ほど紹介した「スコープ・チェーン」の概念を理解していると、大いに納得できるはずです。ここでは、匿名関数内の Call オブジェクトを先頭に、myClosure 関数の Call オブジェクト、Global オブジェクトというスコープ・チェーンが形成されている）。

これが理解できれば、コードの後半の B C D の挙動についても納得できるはずです。A で返された匿名関数は、ローカル変数 cnt は維持しつつも、これをくくっている関数（ここでは myClosure）とは独立して動作することができます。つまり、B では変数 result に格納された匿名関数が呼び出されて、変数 cnt をインクリメントした結果として「11」を、その後、C、D でも保持された変数 cnt を 1 ずつインクリメントした「12」、「13」を返すことになるわけです。このように、クロージャは一種の記憶域を提供する仕組みなのです。

✓ オブジェクトのように振る舞うクロージャ

クロージャを「一種の記憶域を提供する仕組み」として注目してみると、以下のようなコードを記述することもできます。クロージャを定義した myClosure 関数を複数の個所から呼び出し、戻り値を異なる変数に格納した例です。

```
function myClosure(init) {
  var cnt = init;

  return function() {
    return ++cnt;
  }
}

var result1 = myClosure(1); A
var result2 = myClosure(10); B

window.alert(result1()); // 2
window.alert(result1()); // 3
window.alert(result2()); // 11
window.alert(result2()); // 12
window.alert(result1()); // 4
```

一見不可思議な挙動に思われるかもしれないが、スコープ・チェーンの考え方からすれば、当然の挙動でもあります。繰り返しであるが、JavaScript では関数呼び出しのタイミングで新しい Call オブジェクトを生成します。ここでは、A、B でそれぞれ myClosure 関数が呼び出されたタイミングで、

- 匿名関数内の Call オブジェクト（先頭）
- myClosure 関数の Call オブジェクト
- Global オブジェクト

というスコープ・チェーンが形成されているわけであるが、このスコープ・チェーンがいずれも「実行コンテキストの単位に独立したものである」という点が重要なことです。つまり、これらのスコープ・チェーンに属するローカル変数もまた相互に独立したものである（別物である）ということなのです。

これを念頭に置いて、いま一度、見てみるとコードの流れは明快です。A、B での myClosure 関数呼び出しでそれぞれ独立したクロージャとローカル変数 cnt（値は「1」と「10」）を生成する。そして、後続の result1、result2 のクロージャそれぞれの呼び出しでも、独立した値「1」、「10」の変数 cnt を、それぞれ別物としてインクリメントしていることになるわけです。

このようなクロージャの仕組みは、オブジェクトにおけるプロパティ／メソッドにもよく似ていると思われるかもしれない。実際、A、B での関数呼び出しはオブジェクトのインスタンス化、クロージャから参照されるローカル変数はプロパティ、クロージャを構成する関数はメソッド、クロージャをくくっている親関数はコンストラクタとなぞらえることができます。

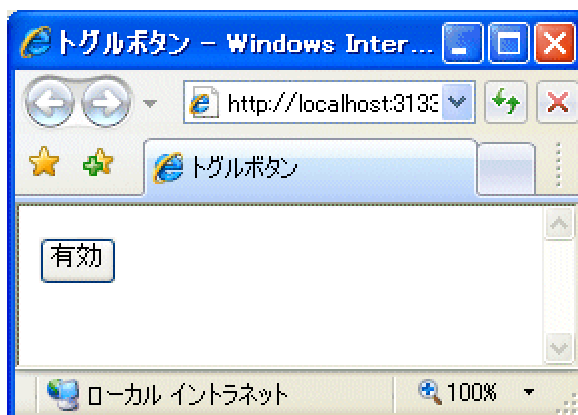
もちろん、クロージャとオブジェクトとが常に置き換え可能であるわけではなく、クロージャはその構造上、1 つの関数として記述する必要があることから、複雑な処理の定義には不向きです。一方、関数内関数というその構造からシンプルな処理をシンプルに記述するには、オブジェクトよりもクロージャが向いているといえます。

✓ イベント・ハンドラにクロージャを適用する

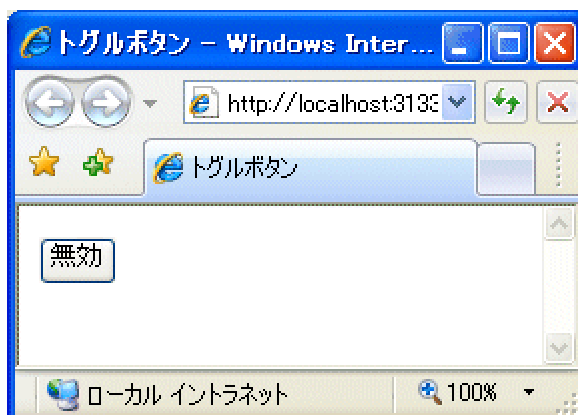
クロージャの仕組みと基本的な挙動については以上です。今後もクロージャという言葉を何度も目にすること

になるはずですが、いまいちクロージャという概念が心に染み入ってこないのは、概念そのものの分かりにくさというよりも、クロージャを使用するメリットが見えてこないせいかもしれません。

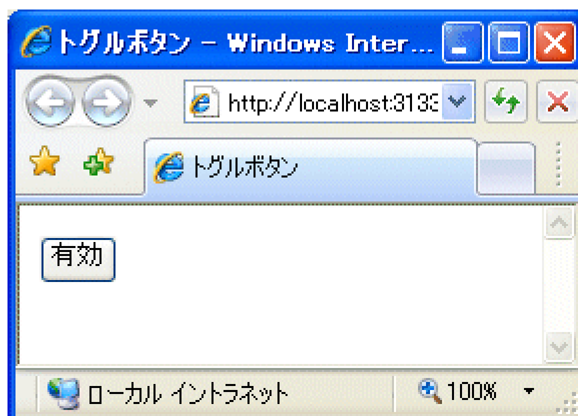
そこでここでは、クロージャを利用する例として、イベント・ハンドラにクロージャを適用するサンプルを見てみることにしよう。ここで紹介するのは、クリックのたびにキャプションを有効／無効に切り替えるトグル・ボタンを実装するコードです。



有効ボタンをクリック



無効ボタンをクリック



クロージャを利用したトグル・ボタンの動作例
ボタンのクリックのたびに、キャプションが「有効」「無効」と切り替わる

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head><title>トグルボタン</title></head>
<body>

<form id="form1" runat="server">
 <div>
  <input id="btn" type="button" value="無効" />
 </div>
</form>

<script type="text/javascript">

document.getElementById('btn').onclick = setButtonState();

function setButtonState() {
    var flag = false;
    var btn = document.getElementById('btn');

    return function() {
        flag = !flag;
        this.innerText = flag ? "有効" : "無効";
  };
}
</script>

</body>
</html>
```

ここではクロージャを利用して、トグル・ボタンの状態（変数 flag）を維持し、これをクリックのたびに反転させています。クロージャを利用しない場合、状態はグローバル変数として別個に管理しなければならないが、クロージャを利用することでイベント・ハンドラの中でまとめて記述できるので、すっきりと可読性の高いコードを記述できることが分かると思います（ここでは、状態を管理する変数が flag1 つであるからそうでもないが、管理すべき情報が複数に及んだ場合を想像してほしい）。また、グローバル変数を利用していないので、もしも同様の機能を持つボタンを設置したいという場合にも、ただ単に、

```
document.getElementById('btn2').onclick = setButtonState();
```

のようなハンドラ定義のコードを記述しさえすればよい。

4.JavaScript でオブジェクト指向プログラミング

JavaScript がこれだけの注目を浴びた理由の 1 つとして、Ajax 技術の登場とも相まって、JavaScript に対する確かな理解の必要性が高まったという事情は否定できない。しかし、それだけでは説明できない急速な注目の理由として、もう 1 つ、JavaScript という言語そのものが持つユニークさが開発者の目を引いたという点は看過できない。

もっとも、このユニークさは同時に、多くの開発者が感じている JavaScript に対する苦手意識と同義でもある。これまで Visual Basic や C#、Java といった言語でオブジェクト指向構文になじんできた開発者にとって、JavaScript のオブジェクト指向構文はいかにも奇異なものに映るのだ。ようやくクラスという概念を理解した開発者が、JavaScript という言語の背後にたびたび見え隠れする「プロトタイプ・ベースのオブジェクト指向」というキーワードに、そもそもの敬遠感を抱いているという事情もあるだろう。

しかし、JavaScript における「プロトタイプ・ベースのオブジェクト指向」は何ら目新しい概念ではない。ごくごく単純化していってしまえば、プロトタイプとは「より縛りの弱いクラス」のようなものと思っていただければよいだろう。次はいよいよ JavaScript に対する苦手意識の根幹でもある（と思われる）「プロトタイプ・ベースのオブジェクト指向」構文について解説する。

なお、本論に入るに当たって、1 点のみ注意していただきたい点がある。JavaScript では厳密な意味でのクラスという概念は存在しない。しかし、本稿を読まれている読者の多くはクラス・ベースのオブジェクト指向構文になじんでいると思われることから、ここでは JavaScript において「クラス的な役割を持つ存在」を、便宜上、「クラス」と呼ぶものとする。

➢ JavaScript における"クラス"の定義

JavaScript におけるオブジェクト指向構文について理解するには、抽象的な解説を重ねるよりも、まずは具体的なクラスを実際に作成してみた方が話が早いだろう。以下は、JavaScript で中身を持たない、最も単純なクラスを定義した例です。

```
var Animal = function() { };
```

変数 Animal に対して、空の関数リテラルを代入しているだけのコードです。「これがクラス？」と思われた方もいるかもしれないが、これが JavaScript におけるクラスです。れっきとしたクラスである証拠に、

```
var anim = new Animal();
```

と、クラス・ベースのオブジェクト指向構文でもおなじみの new 演算子を利用して、実際にインスタンス化を行ってみると、確かに正しく（エラーなども出ずに）実行できることが確認できます。ここではまず、JavaScript では関数オブジェクトにクラスとしての役割を与えているという点を覚えておきましょう。

✓コンストラクタとプロパティ

コンストラクタとは、オブジェクトを生成する際に自動的に呼び出される関数（メソッド）のこと。クラス・ベースのオブジェクト指向構文を少しでもかじったことのある方ならば、これはごく耳になじんだキーワードの 1 つです。

前述した、Animal 関数は new 演算子によって呼び出され、オブジェクトを生成するという意味で厳密には「クラス」そのものというよりも「コンストラクタ」と呼ぶのがより正しいです。

コードを見ても分かるように、JavaScript においては、コンストラクタと（普通の）関数との間に本質的な違いはありません。関数として呼び出すのか、それとも new 演算子によって呼び出すかによって、関数の振る舞いが変わるだけです。もっとも、（構文規則ではないが）実際のコードで、通常の関数とコンストラクタとが区別できないのは不便であるので、一般的にはコンストラクタ（クラス）名は大文字で始めるのが好ましいとされています。

さて、先ほどの空のコンストラクタを定義したわけですが、通常、コンストラクタの中では生成するオブジェクトを初期化するためのコードを記述するのが一般的です。具体的には、オブジェクト共通で利用するプロパティ（メンバ変数）などを定義するのがコンストラクタの役割とされています。以下は、Animal クラスに name／sex プロパティを追加した例です。

```
var Animal = function(name, sex) {
  this.name = name;
  this.sex = sex;
}
var anim = new Animal("トクジロウ", "オス");
window.alert(anim.name + ":" + anim.sex); //「トクジロウ:オス」
```

ここで注目していただきたいのは、コンストラクタの中の this キーワードです。コンストラクタとして関数オブジェクトを呼び出した場合、this キーワードは新たに生成されるオブジェクトを表すことになります。そして、

```
this.プロパティ名 = 値
```

のように記述することで、オブジェクトにプロパティを追加することができるというわけです。実際、コンストラクタで設定された name／sex プロパティが、作成したオブジェクトから正しく参照できることが確認できると思います。

もう 1 点、コンストラクタを定義する場合のささやかな注意点があります。というのも、コンストラクタでは「戻り値を返す必要がない」という点です。クラス・ベースのオブジェクト指向構文に慣れていればごく当たり前のポイントではありますが、プロトタイプ・ベースのオブジェクト指向構文でもこの点は同様ですのであらためて注意してください。コンストラクタ関数の役割は、あくまでこれから生成するオブジェクトの初期化を行うことであって、オブジェクトそのものを返すことではないのです。

✓ メソッド ― コンストラクタによる定義 ―

いまさらいうまでもなく、オブジェクトの構成は大きく「データ」と「手続き」とに分類できる。先ほどは、プロパティ（メンバ変数）を定義することでオブジェクトの「データ」を定義したので、当然の流れとして、次は「手続き」の部分――「メソッド（メンバ関数）」を定義してみることにしましょう。以下は、先ほどの Animal クラスに toString という名前でメソッドを追加した例です。

```
var Animal = function(name, sex) {
  this.name = name;
  this.sex = sex;
  this.toString = function() {
    window.alert(this.name + " " + this.sex);
  };
}
var anim = new Animal("トクジロウ", "オス");
anim.toString(); //「トクジロウ オス」
```

JavaScript には厳密な意味でのメソッドという概念はない。このコードを見ても分かるように、「値として関数オブジェクトが渡されたプロパティがメソッドと見なされる」わけです。ここでは toString という名前のプロパティに匿名関数を引き渡すことで、toString というメソッドを追加したことになります。

✓ メソッド ― インスタンスへの追加 ―

もっとも、JavaScript ではメソッドをあらかじめコンストラクタで定義できるばかりではありません。インスタンス化されたオブジェクトに対しても、後からメンバを追加できるのが JavaScript の特徴です（これを JavaScript の「動的性質」と呼ぶ）。その JavaScript の動的性質を利用することで、先ほどのコードを以下のように書き換えることができます。

```
var Animal = function(name, sex) {
  this.name = name;
  this.sex = sex;
}
var anim = new Animal("トクジロウ", "オス");
anim.toString = function() {
  window.alert(this.name + " " + this.sex);
};
anim.toString(); //「トクジロウ オス」
```

先ほど同様、toString メソッドが正しく動作していることが確認できるはずです。ただし、インスタンスに対して動的にメソッドを追加した場合には、注意も必要となります。以下に、Animal クラスに対するインスタンスを 2 つ登場させた例を見てみましょう。

```
var Animal = function(name, sex) {
  this.name = name;
  this.sex = sex;
}
var anim = new Animal("トクジロウ", "オス");
anim.toString = function() {
  window.alert(this.name + " " + this.sex);
};
anim.toString(); //「トクジロウ オス」

var anim2 = new Animal("リンリン", "メス");
anim2.toString(); // エラー発生
```

インスタンス anim2 から toString メソッドを呼び出そうとした時点で、エラーが発生することが確認できます。Internet Explorer では、恐らく「オブジェクトでサポートされていないプロパティまたはメソッドです。」というエラーメッセージが出力されるはずです。

このことから、インスタンスへのメソッドの追加はあくまでインスタンス（ここでは anim）への追加であって、「本体あるクラスへの追加ではない」ことが理解できると思います。つまり、クラス・ベースのオブジェクト指向言語においては、あるクラスを基に作成されたインスタンスは必ず同じメンバを持つはずであるが、JavaScript では同一の“クラス”を基に作成されたインスタンスでも異なるメンバを持つ可能性がある、ということなのです。これが、先ほどプロトタイプ・ベースのオブジェクト指向が、クラス・ベースのそれよりも「縛りが弱い」と述べた理由のひとつです。ちなみに、ここではメンバを追加する例について述べたが、インスタンス、あるいはコンストラクタ経由で追加したメンバは delete 演算子（後述）で削除することも可能です。いずれにせよ、ここでは、インスタンスに追加したメンバは、（オブジェクト共通ではなく）そのインスタンスのみで有効であるという点を押さえておきましょう。

✓ プロトタイプ・ベースのオブジェクト指向

ということで、インスタンス共通のメソッドを定義するには、インスタンスに対してではなく、（少なくとも）コンストラクタによって定義する必要があることが分かった。しかし、「少なくとも」とただし書きが付いたことからも予想できるように、コンストラクタでメソッドを追加するのは好ましいことではない。

というのも、クラス（コンストラクタ）はインスタンスを生成する都度、それぞれのインスタンスのためにメモリを確保する。Animal クラスに属する name、sex、toString という 3 つのメンバを設定するわけです。ところが、toString「メソッド」については、すべてのインスタンスでそれぞれまったく同じ値を設定しているにすぎない。ここでは、Animal クラスでメソッドが 1 つ登録されているだけなので、さほど問題にはならないかもしれないが、メソッドが 10 も 20 も登録されているクラスだとしたらどうだろう。インスタンスごとに 10 も 20 ものメソッドを「無駄に」コピーしなければならなくなってしまう。これは当然、好ましい挙動ではない。そこで登場するのが、「プロトタイプ」という考え方です。JavaScript におけるすべてのオブジェクトは「prototype」という名前のプロパティを公開している。prototype プロパティは、デフォルトで何らプロパティを持たない空のオブジェクト（プロトタイプ・オブジェクト）を参照しているが、必要に応じてメンバを追加することが可能になっている。そして、ここで追加されたメンバは、そのままインスタンス化された先のオブジェクトに引き継がれる――もっといえば、prototype プロパティに対して追加されたメンバは、そのクラス（コンストラクタ）を基に生成されたすべてのインスタンスから利用できるというわけだ。やや難しげないい方をするならば、

「関数オブジェクトをインスタンス化した場合、インスタンスは基となる関数オブジェクトに属する prototype オブジェクトに対して、暗黙的な参照を持つことになる」といい換えてもよいかもしれない。

そろそろ分かりにくくなってきたという方のために、ここで具体的なコードを見てみることにしましょう。以下は、コンストラクタ経由により追加した toString メソッドを、prototype オブジェクト経由で追加するように書き換えた例です。

```
var Animal = function(name, sex) {
  this.name = name;
  this.sex = sex;
}

Animal.prototype.toString = function() {
  window.alert(this.name + " " + this.sex);
};

var anim = new Animal("トクジロウ", "オス");
anim.toString(); // 「トクジロウ オス」
```

このように Animal クラスから生成されたインスタンス（ここでは anim）は、Animal.prototype プロパティによって参照されるオブジェクトを暗黙的に参照するようになる。「プロトタイプ・ベースのオブジェクト指向」というと、（特に「クラス・ベースのオブジェクト指向」に慣れたエンジニアにとっては）なじみにくい概念にも感じられるかもしれない。しかし、要は「単にクラスという抽象化された設計図が存在しない」のが JavaScript の世界と考え直せばいいかもしれない。

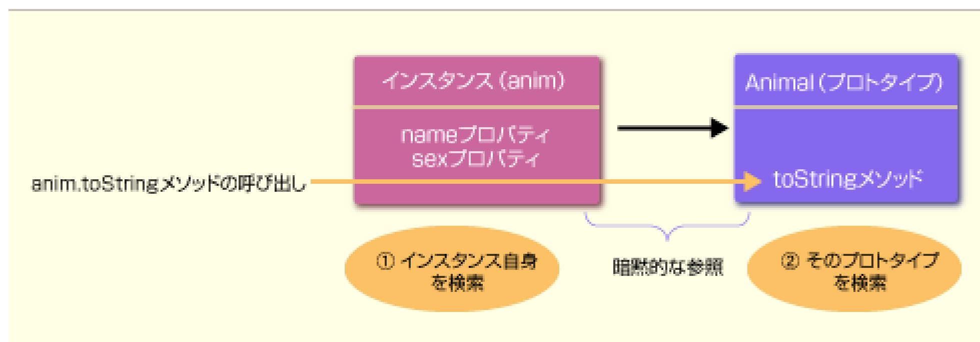
JavaScript の世界で存在するのは、常に実体化されたオブジェクトであり、新しいオブジェクトを作成するにも（クラスではなく）オブジェクトをベースにしているというだけだ。そして、新しいオブジェクトを作成するための原型を表すのが、それぞれのオブジェクトに属するプロトタイプ・オブジェクト（prototype プロパティ）なのである。クラスという抽象的な概念を間に差し挟まない分、より直感的な世界に思えてこないだろうか。

✓プロトタイプ･オブジェクトを介する利点

プロトタイプの概念が理解できたところで、話を戻そう。そもそも、コンストラクタでメソッドを追加するのは好ましくないという話から、プロトタイプが登場したわけであるが、プロトタイプを介することで何が変わるのだろうか。ポイントは 2 点です。

- 必要なメモリ量を節約できる

繰り返しであるが、プロトタイプ･オブジェクトの内容はそれぞれのインスタンスから暗黙的に参照されるものです。具体的には、アプリケーションからオブジェクトのメンバを参照する場合、内部的には以下のような順序で検索が行われています。

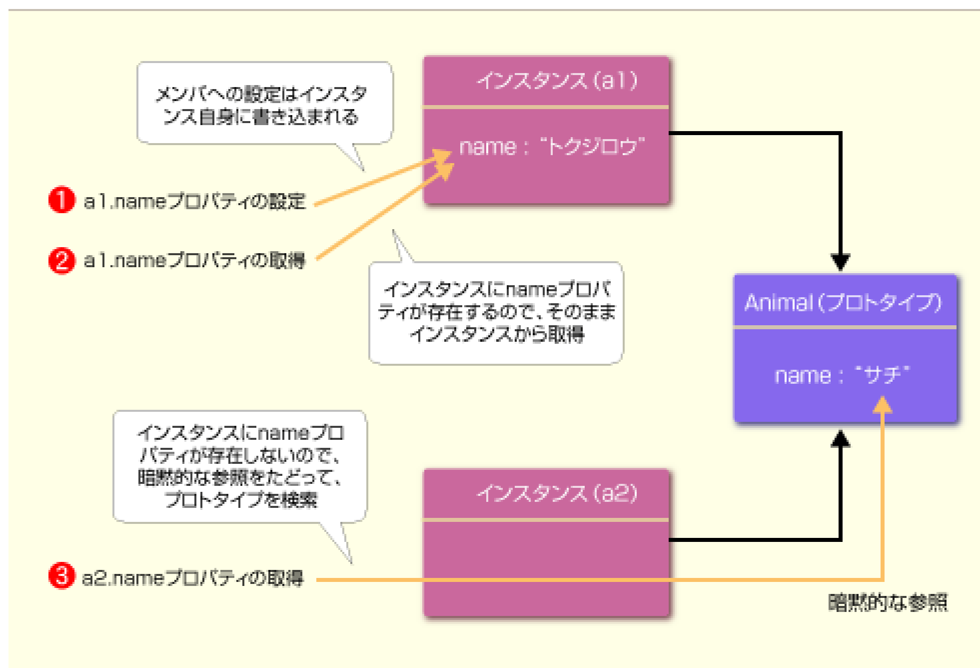


最初にインスタンス側（ここでは anim）に toString という名前のメンバが存在しないかを検索します。しかし、ここではインスタンス自身が toString というメンバを持たないので、暗黙的な参照をたどってプロトタイプ･オブジェクトを取得し、その toString メソッドを取得するのです。

つまり、インスタンス化に際して、プロトタイプ･オブジェクト配下のメンバが個々のオブジェクトにコピーされるわけではないので、それぞれのオブジェクトで消費するメモリを節約できるというわけです。もっとも、ここでふと疑問がわき上がってきます。すべてのインスタンスが基となるオブジェクト（プロトタイプ）に対して暗黙的な参照を持つとすると、プロトタイプで提供されるメンバに対する変更は（いわゆるクラス変数やインスタンス変数のように）すべてのインスタンスで共有されてしまうのだろうか。具体的なコードで確認してみましょう。

```
var Animal = function() {};

Animal.prototype.name = ″サチ″;
var a1 = new Animal();
var a2 = new Animal();

window.alert(a1.name + ″ | ″ + a2.name); //「サチ | サチ」

a1.name = ″トクジロウ″;
window.alert(a1.name + ″ | ″ + a2.name); //「トクジロウ | サチ」
```

name プロパティはプロトタイプ･オブジェクト（Animal.prototype）で宣言されたプロパティであるが、結果を見ても分かるように、あるインスタンス（ここでは a1）に対して施された変更は異なるインスタンス（ここでは a2）には反映されていないことが確認できる。これはどうしたことだろう。結論からいってしまうと、プロトタイプに対する暗黙的な参照が利用されるのは、読み込みの場合だけであるのだ。書き込みはあくまでインスタンス自身に対して行われるため、プロトタイプに対して変更が影響することはない。内部的な挙動については、以下の図を見てみるとわかりやすい。

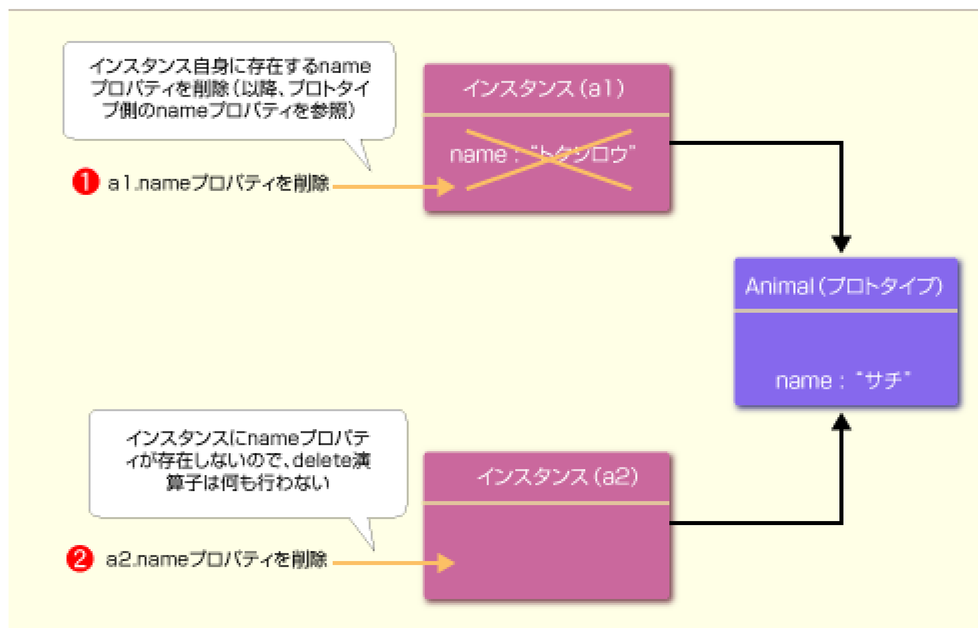


初期状態では、インスタンス a1、a2 ともにプロトタイプ･オブジェクトを参照しているわけであるが、a1.name プロパティに対して新たな値が設定されたところで、インスタンス a1 の側ではプロトタイプ･オブジェクトの name プロパティを参照する必要がなくなる。よって、インスタンス側で用意されている name プロパティが取得されるというわけだ。もちろん、この時点でインスタンス a2 は name プロパティを持たないので、そのまま暗黙の参照をたどってプロトタイプ･オブジェクトの name プロパティを参照することになる。

ちなみに、この考え方は delete 演算子の場合でも同様である。delete 演算子は、オペランドとして指定された配列要素やプロパティ／メソッドを削除するための演算子です。例えば、末尾に以下のようなコードを追加してみます。

```
delete a1.name; // インスタンス a1 の name プロパティを削除
delete a2.name; // インスタンス a2 の name プロパティを削除

window.alert(a1.name + " | " + a2.name); //「サチ | サチ」
```

インスタンス a1 には独自の（プロトタイプ参照によって取得したのではない）name プロパティが存在するので、delete 演算子はこの値を削除します。一方、インスタンス a2 には独自のプロパティは存在しないので、delete 演算子は何も行わないというわけです（暗黙的な参照をたどって、プロトタイプ･オブジェクトが操作されることはない）。結果、それぞれのオブジェクトの状態は以下の図のようになります。



インスタンス a1 の側では、独自のプロパティが存在しなくなったので、再び暗黙的な参照をたどって、プロトタイプ・オブジェクトの値が有効になるというわけです。繰り返しではあるが、インスタンス側でのメンバの追加／削除が、プロトタイプ・オブジェクトに対して影響を及ぼすことはないのです。

✓ undefined 値によるプロトタイプ・オブジェクトのメンバの無効化

delete 演算子ではなく、インスタンス側のプロパティに undefined（未定義）値を設定することで、疑似的にインスタンス側で（ほかのインスタンスに影響を及ぼすことなく）プロトタイプ・オブジェクトが提供するメンバを無効化することも可能です。ただし、delete 演算子がプロパティそのものを削除するのに対して、undefined キーワードはあくまでプロパティそのものの存在はそのままに、値を未定義に設定するだけである点に注意してください（厳密にはこの場合、インスタンスに対して値が undefined である name プロパティを追加している）。つまり、for…in ループでオブジェクト内のメンバを列挙した場合などには、undefined キーワードで未定義となったプロパティは依然として表示されることになります。

```
var Animal = function() {};

Animal.prototype.name = "サチ";
var a1 = new Animal();

a1.name = undefined;

for (key in a1) {
  window.alert(key + ":" + a1[key]);
} // 「name:undefined」
```

- プロトタイプ・オブジェクトの変更はリアルタイムに認識

プロトタイプ・オブジェクト配下のメンバが（インスタンスにコピーされるわけではなく）暗黙的な参照を通じて、必要都度にアクセスされるという事実には、もう１つ大きなメリットがある。それは、インスタンスを生成した「後」に、基となるプロトタイプ・オブジェクトにメンバを追加した場合にも、これを認識できるという点である。例えば以下のような例を見てみよう。

```
var Animal = function() {};
Animal.prototype.name = "サチ";
var anim = new Animal();

Animal.prototype.sex = "メス"; // インスタンスの生成後にメンバを追加

window.alert(anim.sex); // 「メス」
```

もっとも、この性質は、先ほどの「暗黙的な参照」を理解していれば、さほど驚くには当たらないだろう（むしろ当然ともいえる性質である）。以上が、プロトタイプ・オブジェクトの基本であるが、ここでもう１つ、プロトタイプ・オブジェクトをよりシンプルに定義するための記法を紹介しておく。

✓ **プロトタイプをオブジェクト・リテラルで定義する**

ここまでのコード例では、プロトタイプ・オブジェクトに対して、ドット演算子で個々のメンバを追加してきた。もちろん、これはこれで正しい記法なのだが、メンバの数が多くなってきた場合、どうしてもドット演算子による記法ではコードが冗長になりがちだ。ささいなことであるかもしれないが、毎回、「Animal.prototype.メンバ名 = ～」のように記述しなければならないのはタイプ量という観点でもうれしくないし、そもそもクラス名（本稿ではAnimal）が変更になった場合に、すべてのメンバ定義について変更しなければならないという点も好ましくない。

そこで登場するのが、オブジェクト・リテラル表現である。リテラルとは、任意の式内に直接に記述可能なデータ値（表現）のこと。本連載でも、関数リテラルや配列リテラルについて紹介してきたが、リテラル表現を利用することで、より記述上の制約を受けずに柔軟なコードを記述できるというメリットがあることは、すでに実感いただけているのではないだろうか。オブジェクトにもリテラル表現があるというならば、これを使わない手はない。以下は、その具体的なコード――Animal クラス（Animal.prototype）に対して、リテラル表現を使って、getVoice／toString メソッドを追加する例です。

```
var Animal = function(name, sex){
  this.name = name;
  this.sex = sex;
}

Animal.prototype = {
  getVoice : function() {
    window.alert(this.name + "「チュウ！」");
  },
  toString : function() {
    window.alert(this.name + " " + this.sex);
  }
};

var anim = new Animal("トクジロウ", "オス");
anim.toString(); // 「トクジロウ　オス」
```

リテラル表現を利用することで、「Animal.prototype.～ =」のような式を記述する必要がなくなった分だけ、コードがすっきりと見やすくなったのがわかると思います。ちなみに、オブジェクト・リテラルの「{名前 : 値, ……}」という表記は、JavaScript における連想配列（ハッシュ）の記法でもあります（後述）。通常、プロトタイプ・オブジェクトに対して複数のメンバをまとめて追加する場合には、このオブジェクト・リテラルの記法を用いるのが好ましいです。

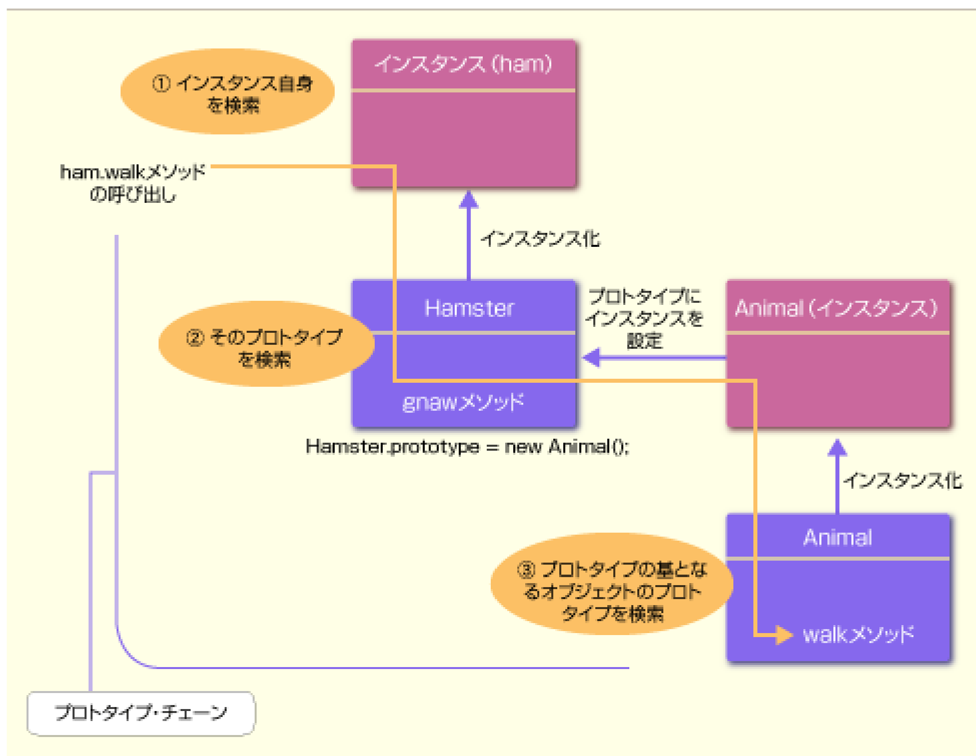
✓ **プロトタイプ・チェーン ― JavaScript の継承機構 ―**

プロトタイプ・ベースのオブジェクト指向を理解するうえで、もう 1 つ、忘れてはならない重要なキーワードとして「プロトタイプ・チェーン」があります。プロトタイプ・チェーンとは、プロトタイプ・ベースのオブジェクト指向における継承機構であるといってもよいでしょう。

```
var Animal = function() {}
Animal.prototype = {
  walk : function() {
    window.alert("トコトコ");
  }
};

var Hamster = function() {};
Hamster.prototype = new Animal();

Hamster.prototype.gnaw = function() {
  window.alert("ガジガジ…");
};

var ham = new Hamster();
ham.walk(); //「トコトコ」 A
ham.gnaw(); //「ガジガジ…」
```

注目していただきたいのは、リスト内の太字の部分――プロトタイプ・オブジェクト（Hamster.prototype）に Animal クラスのインスタンスを格納しているという点です。これによって、Hamster クラスのインスタンスでは、Animal クラスの walk メソッドを呼び出すことが可能になります。この動作は、先ほどの「暗黙的な参照」を理解していれば、その延長線上の概念として理解は容易であるはずです。

JavaScript では、まず現在のインスタンスからメンバの検索を行いそこで該当するメンバが存在しない場合には、次にインスタンスの基となるオブジェクトのプロトタイプから適合するメンバを検索します。そして、それでも該当するメンバが検出されなかった場合には、さらに、そのプロトタイプに格納されたオブジェクトの基となるオブジェクトのプロトタイプから適合するメンバを検索するというわけです。



つまり、Aで walk メソッドを呼び出すと、まず(1)ham インスタンス自身のメンバを、次に(2)Hamster クラスの prototype プロパティ内を、そして、(3)Animal クラスの prototype プロパティ内を、順に検索していくことになります。結果、Animal.prototype で定義された walk メソッドを検出し、これを実行するというわけです。

このように、JavaScript ではプロトタイプにインスタンスを設定することで、インスタンス間の継承関係を形成することができます。もちろん、この継承関係はさらに多階層にすることも可能で、その場合にも順に階層をさかのぼって、最上位の Object.prototype に行き当たるまでメンバの検索が行われることになります。そして、このようなプロトタイプの連なりを称して、「プロトタイプ・チェーン」と呼びます。

さて、このプロトタイプ・チェーン。JavaScript における継承機構であると述べたが、C#や Visual Basic のようなクラス・ベースのオブジェクト指向とは大きく異なるポイントがあります。というのも、クラス・ベースのオブジェクト指向では継承関係が静的に決まるのに対して、JavaScript(プロトタイプ・チェーン)では継承関係を自由に変更可能であるという点です。例えば、以下の例を見てみましょう。

```
var Animal = function() {}
Animal.prototype = {
  walk : function() {
    window.alert("トコトコ");
  }
};

var SuperAnimal = function() {}
SuperAnimal.prototype = {
  walk : function() {
    window.alert("ダーッ！");
  }
};

var Hamster = function() {};
Hamster.prototype = new Animal(); // Animal を関連付け
var ham1 = new Hamster();
ham1.walk(); // 「トコトコ」A

Hamster.prototype = new SuperAnimal (); // SuperAnimal を関連付け
var ham2 = new Hamster();
ham2.walk(); // 「ダーッ！」B
ham1.walk(); // ？？？C
```

ここでは、Hamster.prototype に Animal オブジェクトを関連付けた状態でインスタンス ham1 を、その後、SuperAnimal オブジェクトに切り替えた状態でインスタンス ham2 を生成している。このため、それぞれ A、B ではプロトタイプ・チェーンをたどって、Animal／SuperAnimal クラスの walk メソッドを実行しているわけだ。ここまでは、ごく直感的に理解できる挙動だと思う。では、C の結果はどうなるだろう。インスタンス ham1 を生成したタイミングでは、Hamster.prototype には Animal オブジェクトが関連付いていた。しかし、C の時点ではすでに Hamster.prototype にセットされているのは SuperAnimal オブジェクトである。とすると、C では現在のプロトタイプである SuperAnimal オブジェクトの walk メソッドが呼び出され、「ダーッ！」が出力されるのだろうか。

しかし、結果は「トコトコ」となる。このことから、JavaScript において、いったん形成されたプロトタイプ・チェーンはその後の変更にかかわらず保存されることが理解できる。この点は間違えやすいポイントの 1 つでもあるので、注意していただきたい。

JavaScript のオブジェクト指向は「プロトタイプ・ベースのオブジェクト指向」と呼ばれ、従来の「クラス・ベースのオブジェクト指向」に精通した開発者であればあるほど、なじみにくく感じる概念かもしれない。しかし、あまり難しく考える必要はない。

繰り返しではあるが、JavaScript ではクラスという抽象的な概念の代わりに、すべての概念が実体を持ったオブジェクト（インスタンス）で表されるというだけだ。クラス・ベースのオブジェクト指向言語では、クラスを基にオブジェクトを作成していたところが、JavaScript ではオブジェクトを基に異なるオブジェクトを作成することになっただけなのだ。この 1 点さえ分かってしまえば、実は、プロトタイプ・ベースのオブジェクト指向も恐れるに足らない。