Effective JavaScript

A Few Ways to Improve Your JavaScript

Effective JavaScript は JScript 、JavaScript 、ECMAScript による効果的なスクリプトの構築方法を解説するものです

目次

0.1 はじめに

Dynamic Scripting の JScript リファレンス では JScript 、JavaScript 、ECMAScript の仕様、標準を解説しています。しかし仕様書やリファレンスだけでスクリプトを書くプログラマなどいません。**Effective JavaScript** は効果的 (effective) で強力 (dynamic) なスクリプトの作成方法を紹介することにより、より高度な技術の解説を行います

考え方

基礎的な技術で強力なスクリプトを書くことを目的としているので、純粋に言語だけの解説に集中しています。私は色々考えた末 HTML と DOM をチュートリアルから外しました。既存の JavaScript のチュートリアルには HTML や DOM-0 と絡められているものもありますが、言語と周辺技術の区別が付きにくくなる傾向があります。**Effective JavaScript** で解説するテクニックはこれらの周辺技術と独立に存在するもので、Web ページで JavaScript を使用する場合にも役立ちます

使用する JavaScript

例外を扱うため JScript 5.6 、JavaScript 1.5 (Mozilla 1) 、ECMAScript 3rd Edition を前提にしています

対象読者

あなたは JScript 、JavaScript について知っている必要があります。ここでは「ブラウザのステータスバーに文字列を流す方法」や「配列オブジェクトの扱い方」といった解説はありません。ECMA-262 やスクリプトエンジンの実装について精通している必要はありませんが、ある程度の知識はあるものとします

もしあなたが「自分は中級者だ」と思っているなら問題ありません

0.2 表記法

Effective JavaScript で使用する表記法と約束事について

表記法

限定子

例えば `String` というクラスに `concat` というメンバがある場合に、このメンバを指すつもりで単に `concat` としても、他のクラスに同名のメンバがあればどれを指しているのか分からなくなります。他の言語で一般に使用され

ている *限定子* (ドキュメントなどに使用する "表記" に過ぎないので、コード中では使えない場合がある) としては C/C++ や Perl では `String::concat` 、Java では `String.concat` などどします

JavaScript のメンバは元々配置箇所があいまいで、あるオブジェクトのメンバを他のオブジェクトのメンバとして使用することもできます。そのため JavaScript には限定子のある程度定着した表記法がありません。このチュートリアルでは C/C++ や Perl と同じように2つのコロン (`::`) をメンバの限定に使用します。Java と同じ表記を避けたのは JavaScript にプロトタイプという厄介なものがあるからです

プログラマ、クライアント、ユーザ、ゲスト

プログラマ とは職種ではなく、単にスクリプトを書く人間を指します。*クライアント* はプログラマが記述したコードをスクリプトレベルで利用する側です。人間とは限らず、オブジェクトや関数である場合もあります。*ユーザ* は出来上がったアプリケーションの使用者です。最後に *ゲスト* は Web ページ (XML 含む) の訪問者を意味します

クラス、オブジェクト、インスタンス、インターフェイス

JavaScript には *クラス* がありません (内部原理としては存在します)。クラスと (特に組み込みの) オブジェクトを混同しないようにして下さい。クラスはオブジェクトのテンプレート (雛形) のようなもので、オブジェクトの共通の性質や挙動を定義します。ただ、クラスをコードレベルで定義、参照することはできません

Effective JavaScript で *オブジェクト* と書いてあるのは、他のオブジェクト指向言語での意味と同じです。オブジェクト事態の意味については該当項を参照して下さい

インスタンス はオブジェクトとほとんど同じ意味です。ただ、インスタンスの方がより具象的です

インターフェイス は JavaScript の仕様にはありません。インターフェイスはオブジェクトの持つ (実装を含まない) メソッド定義の集合のようなものです。JavaScript においてインターフェイスは重要であるため、この語を使用することがあります

静的メンバ、インスタンスメンバ、クラスメンバ

該当項参照。*インスタンスメンバ* は通常のメンバで、個々のインスタンスが独立に保持するメンバを指します。対して *クラスメンバ* はコンストラクタが直接保持しているメンバです (JavaScript はオブジェクト生成機構という意味でコンストラクタをクラス定義として捉えることがある) 。**Effective JavaScript** ではクラスメンバを *静的メンバ* とも記述します

汎用メソッド、非汎用メソッド

JavaScript ではメソッドとオブジェクトから抽出したり、他のオブジェクトに結び付けたりできます。ECMAScript の仕様ではこのようなほかのオブジェクトを解して呼び出すことが可能なメソッドを *汎用* (generic; 訳語は私が訳出) メソッドとしています。逆に呼び出すことのできないものを *非汎用* メソッドと記述します。これらのメソッドはコードレベルで定義することができます

1.1 引数

最初の話題が関数であることについて変に感じられるかもしれません。基本的なことを論じるにしても、高度で変態的なテクニックを扱うにもしても、最初のトピックが関数だと拍子抜けかもしれません。しかし関数 (と呼ぶべきもの) は JavaScript の中核となっている要素なのです

JavaScript にはスコープが2つしか (実際は3つですが) ありません。関数の外側と内側です。これらのスコープは入れ子にでき、外側のスコープからは内側がどうなっているのか分かりません。しかし、JavaScript のスコープはたったこれだけなのです。JavaScript でクラスもどきを定義する方法に代表される隠しテクニックはこの単純な仕組みを利用したものが多いのです。そのため、何が可能で何が不可能なのかを見極めるためには、JavaScript の奇妙な関数の仕組みについて理解しておかなければならないのです。関数を使ったテクニックは後々紹介するとして、本節では関数の基本事項について述べます。まずは引数です

値渡しと参照渡し

関数の呼び出し側がどのように引数を渡すか考えて下さい。引数が関数に渡されるとき、あなたが引数にセットした値は *概念的には* スコープを飛び越えることになります。しかし実引数がプリミティブ値の場合、実際にはそのようなことは起こりません。関数側が受け取るのは渡されたもののコピーです。これが *値渡し* です

```
function hoge(v) {
  v = 0;
}
```

```
var n = 100;
hoge(n);
n; // 100 のまま
```

この例で v は呼び出し側の n のコピーになります。つまり値が同じになるだけで、因果関係は全くありません。関数側で値が変更されても呼び出し側の変数には何の影響もありません

オブジェクトを渡すときには *参照渡し* が行われます。関数側が受け取るオブジェクトは呼び出し側のオブジェクトを参照します。この場合は関数側でオブジェクトを変更すると呼び出し側のオブジェクトも影響を受けます。このとき「いや、やっぱり呼び出し側のオブジェクトは変更されないじゃないか」と言って次のようなコードを書く人がいるかもしれません

```
function hoge(o) {
  o = new Object();
}

var o = new Object();
o.p = 100;
hoge(o);
o.p; // o.p は 100 のままだ...
```

これは、関数 hoge はオブジェクトを変更して *いない* のです。hoge の1行目で何が行われるか考えましょう。まず new で全く新しいオブジェクトを作成します。この時点では o には呼び出し側のオブジェクトを参照しています。しかし = 演算子が使われていることにより関数内の o の参照先が変わってしまうのです。元のオブジェクトは呼び出し元の o から依然として参照されているのでガベージコレクトが起こることはありませんが、これ以降の o に対する変更は呼び出し元には影響しなくなるのです。結果として外の o.p は値が変化しなかったのです。次のように書き直すと参照渡しが行われていることを確認できます。関数側はオブジェクトを *変更している* のです

```
function hoge(o) {
  o.p = 0;
}

var o = new Object();
o.p = 100;
hoge(o);
o.p; // o.p は 0 に変更されている!
```

参照渡しが存在する理由の1つは効率です。プリミティブでない値 (複雑な値; オブジェクト) は複数のプロパティを持っており、それを全てコピーしていたら非常に効率が悪くなります。そのためオブジェクトの存在する位置だけを関数に教え、関数はその位置情報を使って必要に応じてオブジェクトにアクセスします。2つ目の理由は呼び出し先での変更を呼び出し元に影響させることです。JavaScript のネイティブオブジェクトが持つメソッドには引数を変更するものはほとんどありませんが、これは非常に有用な手段です

JavaScript では実引数が単純であれば値渡しが、複雑であれば参照渡しが使用されます。VBScript では関数定義がこれを指定することができますが、JavaScript では実引数の値をスクリプトエンジンが調べて勝手に決定します。JavaScript の仮引数には型を指定することができないため、関数定義側はその引数がどちらの方法で渡されているか静的には分かりません。これを解決するのが次項の実引数の扱いに関するテクニックです

仮引数と実引数

JavaScript では、仮引数リストの長さと異なる数の実引数を関数に渡してもスクリプトエンジンは文句を言いません。関数が呼び出されると実引数はその位置に応じて当該仮引数に割り当てられます (引数の渡し方はさっき説明しました)。実引数が足りないと割り当てられなかった各仮引数には undefined が設定され、逆に実引数係引数に比べて多いと、余りは無視されてしまいます

無視されるといっても値が完全に棄てられる訳ではありません。このような状況でも実引数を参照できるように ECMAScript は arguments を関数の局所変数として定義するようにしています。実引数はこの配列のようなオブジェクトのプロパティとして保持されます。仮引数 (parameters) が静的なものであるのに対し、実引数 (arguments) は関数の呼び出しが行われるまでどのようになるか分かりません。この特性を利用すると非常に興味深い関数を記述できるようになります

引数の省略

ECMAScript では仮引数に対応する実引数が与えられなかった場合は、その仮引数が undefined にセットされることになっています。このことから、実引数を調べてそれが省略されているかどうか確認することができ、他の幾つかの言語で採用されている *引数の省略* がエミュレートできるのです。引数の省略とは指定した引数を関数呼び出し時に省略することを認め、その仮引数にあらかじめ決めておいた値をセットするするというものです

例えば文字列をディスプレイに出力する関数を考えましょう。あなたはロケールのことも考えて第2引数に文字列を左から右に書くのか、右から左に書くのかを指定できるブール値を設けることにしました

```
function outputStringToDisplay(str, bLtr) { // bLtr が true であれば左から右
  // ...
}
```

実際にこの関数を使っていると、あることに気が付きます。bLtr に true をセットすることの方が圧倒的に多いのです。呼び出しの度に第2引数を書くのは面倒です。しかし、右から左に読むロケールが存在することを考えると仮引数を削ってしまうのも気が進まないことです。このような場合に第2引数を省略したらそれを true とみなす、ようなことが実現できたら便利だと思いませんか? これをやってみたのが以下に示すコードです

```
function outputStringToDisplay(str, bLtr) { // bLtr が true であれば左から右
  if(bLtr == undefined) // bLtr が省略されていたら
    bLtr = true;        // true として処理を進める
  // ...
}

outputStringToDisplay("Effective JavaScript", true); // 左から右
outputStringToDisplay("Effective JavaScript");       // これも同じ
```

省略されたかどうかを調べるときに if(!bLtr) のような書き方はできません. これだと false や 0 を指定したときも省略されたと勘違いしてしまいます. 逆に undefined が引数として正当である場合にも注意して下さい

この方法は中々役立つものですが、クライアントは引数が省略を許すかどうか、そのデフォルト値が何なのかをどうやって知るのでしょうか. これにはコメントを使用します

```
function outputStringToDisplay(str, bLtr /* = true */)
```

デフォルト値には普通定数を使用します

省略可能な引数は *後方から連続しているものだけ* です. 例えば次の内1番目のものは問題ありませんが、2番目のものは実現できません (エラーになるわけではありませんがあなたの思った通りには動きません)

```
function foo(a, b, c /* = 1 */, d /* = 2 */) {}
function bar(a, b /* = 1 */, c, d /* = 2 */) {}

foo(100, 200); ok。省略可能な c と d を省略している
bar(100, 200); 駄目。省略不可能な c を省略している!
```

省略可能な引数のテクニックは ECMAScript のネイティブメソッドにも使用されています

引数チェック

前項でクライアントが渡してきた実引数が undefined であるかどうか調べました。このようなテクニックを使っていると実引数の正当性が重要になってきます。JavaScript の緩いセマンティクスにより、クライアントは仮引数リストとは異なる実引数リストを使って関数を呼び出すことができます。更に引数の型について関数は何も指定できません。前項の outputStringToDisplay 関数は第1引数を文字列、第2引数をブール値として処理を行おうとしますが、実際にはこれと異なる型の引数が渡される可能性があるのです。string と boolean オブジェクトはオブジェクトではないため、この関数はそれ程深刻な状況を生み出さないでしょう。第1引数にオブジェクトを渡すと、第1引数を文字列として扱うコンテキストでは toString メソッドが呼び出され、大事に至ることはありません。実際にデ

ィスプレイに出力された文字列を見てクライアントは「変な文字列がプリントされてしまった。あの関数の第1引数を間違えたかな?」と思うぐらいで特に深刻なエラーが発生するわけではありません

しかし引数がオブジェクトを要求していたら大変です。関数はそのオブジェクトのメソッドを呼び出そうとしたときに、オブジェクトが実は違うオブジェクトだったら、或いはプリミティブ値だったら一発でエラーになります。ここで問題なのはエラーになることではなく、間違った引数が与えられたということなのです。引数が正しいか調べるには前項と同じように関数の先頭で処理を行う必要があります

現実問題として、関数内でエラーが発生したときにこの種の引数チェックを使用しておくと、エラーの責任をクライアントに付き返したり、関数のセマンティクスをクライアントに通知できるようになります. 関数の引数チェックに失敗したときにクライアントに対して「そりゃ、君の渡してきた実引数に変なものが混じっているからだよ。」と言うことができるのです。これは契約の理論ですが、契約についてはまた後で述べることにしましょう

引数の正当性を構成するのは以下の3つです

- 数が合っているか
- 各引数の型が正しいか
- 各引数の値が有効なものか

この3つの正当性を評価するのは大して難しくありません。まず引数リストの長さですが、これは1行で片が付きます

```
function foo(a, b) {
  if(arguments.callee.length != arguments.length)
    // そりゃ、君の渡してきた...
}

function bar(a, b, c /* = 0 */) {
  if(arguments.callee.length > arguments.length
      || arguments.callee.length - 1 < arguments.length)
    // そりゃ、君の渡してきた...
}
```

次に引数の型のチェックですがこれも簡単です。typeof 演算子と instanceof 演算子を使うだけです。ただし、目的の型とそのセマンティクスによりチェックの仕方が少し異なります

まずプリミティブ型を要求する引数の型チェックをやってみましょう. この場合は typeof 演算子を使うだけです。outputStringToDisplay 関数では次のようになるでしょう

```
function outputStringToDisplay(str, bLtr /* = true */) {
  if(typeof(str) != "string")
    // 第1引数は文字列でないと駄目 -> エラー!
  if(typeof(bLtr) != "boolean")
    // 第2引数はブール値でないと駄目 -> エラー!

  // 本処理...
}
```

しかしこのままでチェックが厳しすぎると思うかもしれません。「String とか Boolean オブジェクトを許容してもいいんじゃないか。JavaScript のオブジェクトは全部 toString メソッドを持っているし、これも文字列として扱えるだろう。」その通りです。JavaScript のオブジェクトは設計上の理由からプリミティブ値への変換も持つものが多いのです。しかし大抵の変換は自動的に行われるので、予期しない変換が起こることがあります. 対応するプリミティブ型が存在するオブジェクト (Boolean 、Number 、String) はコンストラクタを関数として呼び出すことによりプリミティブ値を取り出すことができます

```
function outputStringToDisplay(str, bLtr /* = true */) {
  str = String(str);     // str が何であろうとプリミティブな string 型に変換
  bLtr = Boolean(bLtr); // bLtr が何であろうとプリミティブな boolean 型に変換

  // 本処理...
}
```

これはシンプルな方法で、何をやろうとしているかすぐに分かります。いずれも ECMAScript で定められた型変換規則に従って変換が行われるのです。ただし何でも変換してしまうのでエラーを出す場所がありません。1番目の方法と2番目の方法のどちらを使うかは、その関数の意味で変わってきます。より厳密ながたチェックが必要な場合は前者を、そうでない場合は後者を選んで下さい

引数がオブジェクトを要求してくる場合は `instanceof` 演算子を使用します。事前に `typeof` でオブジェクトかどうか調べる必要はありません。次の例は `Date` オブジェクトを使って何かする関数です

```
function useDate(oDate) {
  if(!(oDate instanceof Date))
    // 第1引数は Date オブジェクトでないと駄目 -> エラー!

  // 本処理...
}
```

「ん? `Date` の派生オブジェクトでもこの関数は通してくれるのかな?」通ります。`Date` の派生オブジェクトは `Date` インスタンスでもあります。`instanceof` 演算子はオブジェクトのプロトタイプチェインを辿り、`Date` を見つけ出すのです

逆に `Date` オブジェクトしか認めない場合 (無意味な気がしますが) は `constructor` プロパティでそのオブジェクトのコンストラクタを調べます

```
function useDate(oDate) {
  if(!oDate.constructor || oDate.constructor != Date)
    // 第1引数は Date オブジェクトでないと駄目 -> エラー!

  // 本処理...
}
```

引数がオブジェクトである場合の問題については他の章でもう一度議論します

引数の型をコメントとして書き込むには以下の形式を使用すると良いでしょう

```
function useDate(/* Date */ oDate)
```

最後は引数の値の正当性です。これは厄介というか面倒です。引数の値として何が正しいかは関数によって異なり、決まった方法がないのです。ここでは例を挙げるだけにしておきましょう

文字列から部分文字列を検索する関数を考えましょう。JavaScript のネイティブオブジェクトには、既に文字列検索のための関数がありますが、まあ、あなたはこの関数を作ったのです。引数リストは次のようになるでしょう

```
function searchString(/* string */ strTarget,     // 検索対象
                      /* string */ strSubstring,  // 検索する文字列
                      /* number */ iBegin         // 検索開始位置
) {
  // 本処理...
}

// 使う
searchString("Effective JavaScript", "Script", 7);
```

仮引数リストを眺めていると、引数として不当な値がすぐに思いつきます

- strTarget 空文字列だったら検索する意味は無い
- strSubstring が空文字列だったら検索する意味は無い
- strTarget の文字列長が strSubstring の文字列長より短かったら検索する意味は無い
- strTarget の文字列長が iBegin 以下だったら検索する意味は無い

同時に次のようなことも思いつきます

- iBegin が負数だったらどう扱えばよいか

- iBegin が小数だったらどう扱えばよいか
- 「検索する意味が無い」とはエラーを意味するか

後で出てきた3つの疑問については決まった正解というものがありません。これらを決定するのは関数の作者なのです。String オブジェクトの文字列検索メソッドは変な引数を渡してもエラーになりませんが、この関数はもっと厳密なものにしてみましょう。上で挙げた事例を全てエラーとして処理するものとします

しかしその前に、「関数がエラーを出す」とはどういうことでしょうか。いままで幾つかの項で「エラーを出す」と言ってきましたが、コード中ではコメントを書いただけでした。実はこのエラー定義も関数作者の仕事なのです

一般的には例外を使用します。ただ例外は最近採り入れられたもので、対応していないスクリプトエンジンは例外処理を見つけると、逆にエラーを発生させてしまうのです。この **Effective JavaScript** は ECMAScript 3rd Edition を前提としているので、堂々と例外を使ったコードが書けますが、クロスブラウザを考慮したコードには使用できないことに注意して下さい (例外については該当章を参照)

では例外を使って searchString を書いてみましょう。ここでは Error オブジェクトを使いますが、エラーに応じた NativeError オブジェクトでも構いません

```
function searchString(/* string */ strTarget,     // 検索対象
                      /* string */ strSubstring,  // 検索する文字列
                      /* number */ iBegin         // 検索開始位置
) {
  // 実引数の数のチェック
  if(arguments.callee.length != arguments.length)
    throw Error("引数の数が合っていません");

  // 型チェックはプリミティブ値につき手抜き
  strTarget = String(strTarget);
  strSubstring = String(strSubstring);
  iBegin = Number(iBegin); // NaN になる可能性もあるが...

  // 空文字列は駄目
  if(strTarget.length == 0 || strSubstring.length == 0)
    throw Error("空文字列は検索に使用できません");

  // 小数や負数は使えない
  if(iBegin < 0 || iBegin % 1 != 0)
    throw Error("検索開始位置に負数や小数は使用できません");

  // strTarget が短い
  if(strTarget.length < strSubstring || strTarget.length <= iBegin)
    throw Error("検索文字列長か開始位置が検索対象文字列長に比べて大きすぎます");

  // 本処理...
}
```

冗長な気もしますが、こういう方法があるということです

NativeError オブジェクトの話が出ましたが、ECMAScript には関数に不適切な引数を渡したときに使用する専用の例外オブジェクトが用意されていません。つまり ECMAScript では引数チェックはあまり問題ではないというのが普通の考え方なのです。ECMAScript の仕様を鵜呑みにするならこのままで構いませんが、折角引数チェックの議論を行ったので例外オブジェクトを作成することにしましょう

```
function InvalidArgumentError(message) {
  var e = new Error(message);
  e.name = "InvalidArgumentError";
  return e;
}
```

「例外オブジェクトとか言ってこりゃただの関数だ」と思われるかもしれません。関数はオブジェクトですがそういう意味ではないですね。この奇妙なアプローチについては例外の章を見て下さい。何はともあれ、実際に例外を投げるには以下のようにします

```
if(/* 引数チェック */)
  throw InvalidArgumentError(/* エラーの説明 */);
```

テクニック — 残余引数

残余引数 (rest parameter) とは対応する仮引数リストが無く、残りものになった実引数を扱う機能で、幾つかの言語で使用されています。これにより可変長引数が実現できますが、JavaScript では引数リストは元々可変長であり、簡単に模倣できます。役に立つかはともかく方法を紹介しましょう

実引数が `arguments` に格納されことは前に述べたとおりです。残余引数 (と勝手に思っているもの) にアクセスするにはこの `arguments` オブジェクトに添え字を使ってアクセスするだけです。これでは簡単すぎるので残余引数だけを格納したリスト (実際はこのリスト全体を残余引数という) を作成するとしましょう。これも簡単です

```
function foo(a, b /*, ... */) {
  var restParameter = Array.prototype.slice.call(
    arguments, arguments.length - arguments.callee.length - 1, arguments.length);

  // ...
}
```

`Array::slice` メソッドは配列から部分配列を取り出すメソッドですが、`arguments` は `Array` オブジェクトでないことに注意して下さい。`Object::call` メソッドを使って汎用メソッド (該当項参照) である `Array::slice` を `arguments` を介して呼び出しています (ただしこれは実装依存です)。ただし `Object::call` は ECMAScript 3rd Edition で採用されたものなので、確実に restParameter を作成したい場合は `for` 文を使って下さい

可変長引数も ECMAScript ネイティブオブジェクトのメソッドに使用されています

テクニック - 関数の多重定義

同じ名前で、意味が少しだけ異なる関数を複数定義したいことがあります。例えば `String::replace` メソッドは第1引数が文字列か正規表現パターンかで挙動が異なります。しかし実際にはメソッドが複数用意されているわけではありません。JavaScript では同じ名前で関数を定義すると最後に定義したものだけが有効になります。関数もオブジェクトに過ぎないためこれは当然のことです

```
function foo() {
  return 1;
}
function foo() {
  return 2;
}
foo(); // 2
```

ここでも引数を調べることにより同名の関数を複数定義したように見せることができます (勿論呼び出しに関しての話です)。調べることは実引数の型と数です

例えばドキュメントが書かれた日付を出力する関数を考えましょう。この関数には月と日が必要ですからこれらの値を引数として要求することにしましょう。まず思いつくのは以下のような引数リストです

```
function outputTimeStamp(/* number */ nMonth, /* number */ nDay) {
  // 月と日から文字列を作成...
}
```

しかしすぐに別の形式を思いつきます。ECMAScript には日付を表すオブジェクトがあるのです

```
function outputTimeStamp(/* Date */ oDate) {
  // Date オブジェクトから文字列を作成...
```

```
}
```

最初から文字列として与えたい場合もあるでしょう

```
function outputTimeStamp(/* string */ strTimeStamp) {
  // 引数をそのまま使う...
}
```

3種類の形式を考えることができました。次はこれら3つの関数を1つにまとめなければいけません。そのためには関数の先頭で引数チェックを行います

```
function outputTimeStamp(/* ... */) {
  if(arguments.length == 2) {
    var nMonth = Number(arguments[0]);
    var nDay = Number(arguments[1]);
    // 月と日から文字列を作成...

  } else if(arguments.length == 1 && typeof(arguments[0]) == "string") {
    var strTimeStamp = String(arguments[0]);
    // 文字列をそのまま使う...

  } else if(arguments.length == 1 && arguments[0] instanceof Date) {
    var oDate = arguments[0];
    // Date オブジェクトから文字列を作成...

  } else
    throw InvalidArgumentError("実引数の数が合っていません");

  // ...
}
```

このテクニックは簡単ですがコードの見通しが多少悪くなります。コメントをうまく使って下さい。また1つの関数に押し込まずに引数に応じた別の関数を用意し、内部で呼び出すという方法もあります

```
function outputTimeStamp(/* ... */)
  if(arguments.length == 2)
    return __outputTimeStampByNumberNumber(arguments[0], arguments[1]);
  else if(arguments.length == 1 && typeof(arguments[0]) == "string")
    return __outputTimeStampByString(arguments[0]);
  else if(arguments.length == 1 && arguments[0] instanceof Date)
    return __outputTimeStampByDate(arguments[0]);
  else
    throw InvalidArgumentError("実引数の数が合っていません");
}
```

1.2 関数とオブジェクト

関数という型はありません。JavaScript では関数は呼び出し可能なオブジェクトに過ぎません。逆にオブジェクトであることを利用して、関数に奇妙な振る舞いをさせることができます

この節では関数の基本的な定義からスタートし、関数をオブジェクトとして使用する例を幾つか紹介します

無名関数

まず復習から始めましょう。関数はその定義方法から4つに分けることができます

1. *FunctionDeclaration* により静的に定義された関数
2. *FunctionExpression* により動的に作成された関数
3. Function コンストラクタにより動的に作成された関数
4. 実装から提供される組み込み関数

とりあえず 4. は無視しましょう。ここで重要なのは 1. から 3. です。最も一般的なのは 1. で、関数の名前、引数リスト、本体で構成される関数の宣言です

```
function plus(a, b) {
  return a + b;
}
```

これだけで十分な気がします。実際、あらゆる静的関数はこれで事足ります。しかし関数を動的に定義したい場合は 2. や 3. の方法が必要になります。1.1 引数 の初めで関数はスコープを持つと説明しました。動的に関数が作成できれば、動的にスコープを形成する実行単位を生み出すことができるのです。関数とスコープの関係は 1.3 クロージャ で解説するとして、この節では関数のオブジェクトとしての側面について述べます

2. や 3. の方法で作成された関数は無名関数になることがあります。無名関数とはその名の通り名前の無い関数です。「関数に名前が無かったらどうやって呼び出すのか分からん」と思われるかもしれませんが、そうではありません。以下は 2. の方法で無名関数を作成する例です

```
var plus = function(a, b) {
  return a + b;
};
```

これは先程 1. の方法で作成した関数と同じ引数リスト、本体を持つ関数です。「plus という名前が付けてある」と思われるかもしれませんが、"plus" は関数の名前ではありません。"plus" は無名関数 (オブジェクト) を参照するオブジェクトに過ぎません。「えー、証拠を見せて」という方はこのオブジェクトの文字列表現を見てみましょう。`Function::toString` は関数のシグニチャと本体を出力するのでしたね

```
// 続き
plus.toString();  // function (a, b) {
                  // return a + b;
                  // }
```

確かに名前がありませんね。理論家のあなたには次の説明はどうでしょうか。次のコードが実行されたとき何が起こるか考えてみて下さい

```
var plus1, plus2;
plus1 = plus2 = function(a, b) {
  return a + b;
};
plus1 == plus2; // true
```

2行目で1つの無名関数に2つの名前を付けようとしています (この方法で名前が付けられるとしたら)。しかし1つの関数に複数の名前を付けることはできません。別々の関数が作成されたと考えるしかありませんが、そうすると5行目は `false` になるはずです (うーん、変な説明)

しかし "plus" という識別子を使ってこの関数を呼び出すことはできるのです。無名関数は JavaScript コード単位の中でも複雑な部類に入ります。例えば Mozilla 付属の JavaScript デバッガ Venkman は無名関数を参照するオブジェクトまでは参照してくれません (これは言語、デバッガ双方の仕様のためで、Venkman に落ち度はありません。またソースを「読ませる」ことである程度は推測してくれます)

2. で作成された無名関数に名前をつける方法は `function` の後に名前を書いてしまうことです

```
var plus = function _plus(a, b) {
  return a + b;
};
plus(2, 3);   // 5
_plus(2, 3); // JScript 以外はエラー! 理由は後述
plus == _plus; // JScript では false
```

"_plus" を使って関数呼び出しが成功するのは JScript だけです。結論から言うと JScript の実装が間違っているのです。詳細を述べる前に関数に名前を付けるとはどういうことが考えてみましょう。ECMAScript 3rd Edition には以下のように記述されています

> *FunctionExpression* の関数名は再帰呼び出しを可能にするために *FunctionExpression* の本体から参照できるようになっています。しかし *FunctionDeclaration* とは異なり、*FunctionExpression* を囲っているスコープからは参照できず影響を与えることもありません — ECMAScript 3rd Edition 13 Function Definition (文章は exeal が訳出)

FunctionDeclaration による静的定義では、関数名は変数オブジェクト (variable object) と呼ばれるオブジェクトのプロパティ名になり、実際の `Function` オブジェクトはそのプロパティの値になります。これに対して *FunctionExpression* の場合は変数オブジェクトをその場で作成してスコープチェインの先頭に追加し、このオブジェクトのプロパティとして関数名を与えます。そしてこのオブジェクトの寿命は関数定義の終わりまでです。つまりその関数名は関数の内部からは参照可能ですが、関数定義の外では無効なのです。無名関数に名前を付ける場合はこの点に注意して下さい

3. の方法は少々特殊で、仮引数リスト本体を *文字列を使って* 定義できます。引数はカンマで区切ります

```
var plus = new Function("a, b", "return a + b;");
plus.toString(); // function anonymous(a, b) {
                 // return a + b;
                 // }
```

この場合も "plus" は関数名ではなく、コンストラクタが返したオブジェクトを参照するだけです。面白いことにこの方法で作成された関数の文字列表現には "anonymous" が使用されます。当たり前ですが "anonymous" は関数名でも何でもありません

関数オブジェクト

JavaScript の関数を単なる手続き (procedure) と考えないことが重要です。関数が呼び出し可能なオブジェクトに過ぎないことは初めに述べました。オブジェクトであるからには変数に代入したり、実引数として関数に渡したり、関数から返すことができます。或いはオブジェクトのプロパティとすることでメソッドにもなり得ます (3章を見て下さい)。ここでは関数をオブジェクトとして使用したテクニックを幾つか紹介します

よく用いられるのは `Array::sort` メソッドで使用されているテクニックです。このメソッドは省略可能な引数として引数が2つの関数をとります。メソッドは配列の要素を比較する際にこのユーザ定義関数を使って順序を決定するのです。ちょっと使ってみましょう。

```
// Array::sort メソッドは、デフォルトでは各要素を文字列とみなして
// 辞書順にソートする。この関数は大文字小文字を区別せずに辞書順に
// ソートするために用意した、引数が2つのユーザ定義関数
function fn(a, b) {
  if(a.toLowerCase() == b.toLowerCase())
    return 0;
  return (a.toLowerCase() > b.toLowerCase()) ? 1 : -1;
}

var arr = ["JavaScript", "javascript", "JAVASCRIPT"];
arr.sort();   // "JAVASCRIPT","JavaScript","javascript"
arr.sort(fn); // "JavaScript","javascript","JAVASCRIPT"

// 無名関数を使うと以下のようにも書ける
arr.sort(function(a, b) {
  if(a.toLowerCase() == b.toLowerCase())
    return 0;
  return (a.toLowerCase() > b.toLowerCase()) ? 1 : -1;
});
```

このようなソートを行う関数では要素間の大小 (どちらが先になるか) の定義が必要です。しかしそのルールは無限に存在するため、ルールを表す引数を単一の数値などの単純なもので構成するのは不可能です。その点関数はルールを細かく表現できるので、`Array::sort` はソート処理の内大小関係を求める部分をユーザ定義関数に委託できるように設計されているのです

では例として二分検索を行う関数を考えましょう。この関数は要素の比較をユーザ定義関数に委託します

```
// 二分検索 (arr はソート済みとする)
// 見つかったら true を返す
function binarySearch(/* Array */ arr,   // 検索対象配列
                      /* object */ obj, // 検索する値
                      /* function */ fn // ユーザ定義関数
) {
  var iBottom = 0;
  var iTop = arr.length - 1;
  while(iBottom <= iTop) {
    var iCenter = Math.floor((iBottom + iTop) / 2);
    if(fn(obj, arr[iCenter]) == 0)
      return true;
    else if(fn(obj, arr[iCenter]) < 0)
      iBottom = iCenter + 1;
    else
      iTop = iCenter - 1;
  }
  return false;
}

function fn(a, b) {
  if(a == b)
    return 0;
  return (a > b) ? 1 : -1;
}
var arr = new Array(-70, 6, 113, 202);

binarySearch(arr, 113, fn);  // true
binarySearch(arr, -113, fn); // false
```

二分検索は値の大小を使って検索する範囲を徐々に狭め、検索値を探すアルゴリズムです。この内、大小関係の定義を第3引数で与えられたユーザ定義関数に委ねています

このような関数を要求するときにはその関数のセマンティクスを正確にドキュメントする必要があります。引数の数、戻り値、再帰関数にしてもよいかなどをどこかに書いておく必要があります

以上の例のように関数をパラメータとすることで、処理の一部をクライアントがカスタマイズできるようになります。その利用方法は引数だけではありません。JavaScript で最もよく用いられるのは HTML のイベントハンドラです

```
document.body.onload = function() {
  //ドキュメント読み込み完了時の処理
};
```

他にも特別に関数を要求してくるオブジェクトもあります。典型的なのは Observer パターンです。Observer パターンではオブジェクトの状態が変化したときに、オブザーバと呼ばれる関数やメソッドが呼び出されます。JavaScript は関数をそのまま渡すことができるので簡単に Observer パターンを実装できます (Dynamic Scripting では DSMenu にこのパターンが使われています。ただし単一の関数ではなくインターフェイスを使って実装しています。<リソース>を参照)

関数アダプタ

前項で C の関数ポインタのようなことをやりました。となると次は C++ の *関数アダプタ (adapter)* について考えるとしましょうか

関数アダプタは関数ポインタによく似ていますが、関数ではありません。関数アダプタは多重定義された ( ) 演算子を使って、あたかも関数呼び出しをしているかのようにアクセスできるクラス (オブジェクト) です。関数ポインタと違うところは関数ポインタが単なる関数であったのに対して、関数アダプタはオブジェクトです。つまりカプセル化された内部表現を伴っているのです

以上は C++ のお話で JavaScript プログラマには分かりにくかったかもしれませんが、心配することはありません。JavaScript の関数は全て C++ の関数アダプタのように機能します。これは JavaScript の関数が全てオブジェク

トであることに基づいています。要するにユーザ定義関数の実行中に、その関数のプロパティを参照することができるのです

まだ少し分かりにくいですか? では例を見て下さい。以下の関数は先ほど Array::sort メソッドに使用したユーザ定義関数の変形バージョンです。前回は大文字小文字を区別するかどうかという条件をハードコーディングしていましたが、今回はプロパティを使って条件を決定できるようになっています。関数の bIgnoreCase プロパティが真のときは大文字小文字は区別されません。逆に偽のときは区別されます

```
function CompareString(a, b) {
  if(CompareString.bIgnoreCase) { // 大文字小文字を区別しない
    a = a.toLowerCase();
    b = b.toLowerCase();
  }
  if(a == b)
    return 0
  return (a > b) ? 1 : -1;
}

var arr = new Array("b", "a", "C");
CompareString.bIgnoreCase = false;
arr.sort(CompareString); // "C","a","b"
CompareString.bIgnoreCase = true;
arr.sort(CompareString); // "a","b","C"
```

今回は簡単な例だったので少々インチキっぽいですが、それなりに役立つテクニックです。特にオブジェクト内で関数部分の方が小さくなったときに、この関数は ( ) 演算子の多重定義のように振舞います。また、これにより COM の高級クライアントが持つデフォルトメソッドを実現することもできます

```
// デフォルトメソッドの簡単な例
function Foo() {
  return Foo.getValue(); // getValue をデフォルトメソッドにする
}

// Foo オブジェクトの本体はこっち
Foo.getValue = function() {/* ... */};
Foo.setValue = function() {/* ... */};
// オブジェクトの定義が続く...
```

関数ポインタと Function コンストラクタの限界

なるほど、*関数ポインタ* のテクニックは便利なもので、JavaScript プログラマには必須の技術と言えます。この関数ポインタと関数の動的生成を組み合わせると非常に強力であることは誰でも分かります。例えばこのようなものはいかがでしょうか

```
// 配列の要素の内、条件に適合したものだけを抽出して
// 新しい配列を作成する関数
function filterArray(/* Array */ arr,   // 元の配列
                     /* function */ fn  // ユーザ定義関数
                                        // 引数は1つで配列の要素。戻り値は
                                        // その要素を新しい配列に加えるかどうか
) {
  var arrNew = [];
  for(var i = 0; i < arr.length; ++i) {
    if(fn(arr[i]))
      arrNew.push(arr[i]);
  }
  return arrNew;
}
```

関数 filterArray は条件に適合した要素だけを新しい配列に追加し、その条件はユーザ定義関数により決められます。使い方は次のようになるでしょう

```
var arrOriginal = [45, -29, 213, 99, 118];

// 50 より大きい要素だけを取り出す
function greaterThan50(n) {
  return n > 50;
}
filterArray(arrOriginal, greaterThan50);  // 213,99,118

// 100 より大きい要素だけを取り出す
function greaterThan100(n) {
  return n > 100;
}
filterArray(arrOriginal, greaterThan100); // 213,118

// 200 より大きい要素だけを取り出す
function greaterThan200(n) {
  return n > 200;
}
filterArray(arrOriginal, greaterThan200); // 213
```

まあ、何が言いたいのかすぐに分かると思いますが、条件を変更するたびに関数を作り直すのが面倒なのです。無名関数をそのまま引数に渡せば少し見通しが良くなりますが、根本的には変わりません

ここでは3つのユーザ定義関数の性質に共通点があることに注意しましょう。これらの関数はいずれも引数がある数値より大きいかどうかを調べています。この「ある数値」をパラメータとする関数が1つあれば事足りると考えられます。そこで登場するのが Function オブジェクトを返す関数、関数を返す関数です

```
function greaterThan(/* number */ u) {
  return new Function("n", "return n > " + u + ";");
}
```

この関数があれば先ほどのコードは非常に簡単になります

```
var arrOriginal = [45, -29, 213, 99, 118];

// 50 より大きい要素だけを取り出す
filterArray(arrOriginal, greaterThan(50));  // 213,99,118

// 100 より大きい要素だけを取り出す
filterArray(arrOriginal, greaterThan(100)); // 213,118

// 200 より大きい要素だけを取り出す
filterArray(arrOriginal, greaterThan(200)); // 213
```

これは素晴らしいテクニックです。この方法を使えば他にも lessThan (より小さい) 、equalTo (等しい) など汎用的な関数をその場で簡単に作成できます

しかしこの方法には問題があるのです。まず Function コンストラクタが引数に文字列を要求することです。上の greaterThan 関数を見てもあまり綺麗ではありません。今は単純な例ですが、複雑な条件を記述する関数だとこのままでは厳しいのは明らかです。普通の関数定義と同様に記述できたら良いと思いませんか?

「関数を返す関数」という考え方自体は上策と言えます。Function コンストラクタの弱点が明らかになったところで、次の節へ進むとしましょう。次節で議論する「クロージャ」は JavaScript 屈指の強力なテクニックです。クロージャはここでの問題を解決するだけでなく、新しいスクリプト記述とさらに汎用的なプログラミングを可能にします

1.3 クロージャ

関数を返す関数

以下が前節の greaterThan 関数を書き直したものです。お察しの通り Function コンストラクタの代わりに関数式を使っています

```
function greaterThan(/* number */ u) {
  return function(n) {
    return n > u;
  };
}

// 使う
var arrOriginal = [45, -29, 213, 99, 118];
filterArray(arrOriginal, greaterThan(50)); // 213,99,118
```

さて、このコードを見てどうでしょうか。「何だか訳が分からない」と思うかもしれません。それもそのはずで、実際に奇妙なことが起こっているのです。関数呼び出しについて次のような手順を考えるとこのコードは動かないような気がします

1. greaterThan(50) とやると引数が1つで、その引数が u より大きければ真を返す関数が作成され、返される
2. filterArray で実際に 1. の戻り値である無名関数が呼び出される
3. 無名関数が実行され、n と u が比較される。でも関数 greaterThan のスコープはもう終わっているから u は見つからない???

では Function コンストラクタのときはなぜうまく動き、違和感が無かったのでしょうか。異なる点は外側の引数の埋め込み方です。もう一度前回の greaterThan を見てみましょう

```
function greaterThan(/* number */ u) {
  return new Function("n", "return n > " + u + ";");
}
```

ここでは u は文字列に組み込まれ、完成した文字列 (例えば "return n > 50;") が無名関数の本体になっています。しかし今回作成した関数から返される無名関数はそうではありません。以下のコードを実行すれば分かります

```
var f = greaterThan(50);
f.toString(); // function(n) {
              //   return n > u;
              // }
```

返された無名関数には確かに u がそのまま残っています。しかし関数のコードそのものには "u" は見当たりません。なぜこれで参照エラーにならないのでしょうか

関数+スコープ

実はこのようにして作成された関数は自身を作成した実行フレーム (activation frame) への参照を保持することで、u を可視のまま保っているのです。実行フレームはスコープに入るたびに作成され、そのスコープ中で定義されている全ての限定名とその値というセットの集合を持ちます。そしてスコープが終了しても実行フレームは *自身を参照するものがある限りガベージコレクトされません*。そしてこの無名関数を *クロージャ (closure; 囲まれたもの)* というのです (ECMAScript の仕様にはこの言葉は使われていない)

他のプログラム言語 (スクリプト以外) から移行してきた人にとっては奇怪千万ですが、JavaScript の強みの一つにこのクロージャが挙げられます。以下は JavaScript 1.5 で private プロパティを実現しようとする、よく見られる例ですがクロージャが理解できれば局所変数がコンストラクタの外でも有効な理由が説明できるでしょう (ここで示す

private メンバ実現のアプローチはよく知られているものですが、これは不完全です。私のアプローチは第3章で紹介しましょう)

```
function Person(/* string */ strName) {
  var m_strName = strName; // private メンバ
  // ゲッタ
  Person.prototype.getName = function() {
    return m_strName;
  };
}
```

テクニック — テンプレート

前項まででクロージャを使って本体の一部だけが異なる関数を作成する方法を説明しました。「一部」とはクロージャから可視な限定子です。一部だけが異なっていてあとは同じというと 1.1 引数 で紹介した引数型チェックを自動化できるのではないでしょうか

```
// 第1引数が T のインスタンスでないと例外を投げる関数を返す関数
function foo(/* constructor */ T) {
  return function(/* T */ n) {
    if(!(n instanceof T))
      throw InvalidArgumentError("引数の型が合っていません");
    // 本処理...
  };
}

// 使う
var doSomethingWithDate = foo(Date);
var doSomethingWithNumber = foo(Number);

doSomethingWithDate(new Number(7)); // エラー! "引数の型が合っていません"
doSomethingWithDate(new Date());    // ok

doSomethingWithNumber(new Number(7)); // ok
doSomethingWithNumber(new Date());    // エラー! "引数の型が合っていません"
```

このように汎用的な関数が書けるようになります。説明していませんでしたが doSomethingWithNumber を作成した後でも doSomethingWithDate は健在です。上書きされる心配はありません。スコープに入るたびに新しい実行フレームが作成されるのでしたね

ところでこのようなテクニックは役に立つのでしょうか。確かに上のコードは引数チェックという意味では正しく動作します。しかし "本処理..." の部分には何が書けるか考えてみて下さい。使用例では T が `Number` と `Date` となる関数を返しましたが、この関数が数値と日付に対して行いたい汎用的な処理とは何でしょうか。或いはそのような汎用的な処理が可能なのでしょうか

私は関数内部の特定の型だけが異なる関数を作成するこのテクニックを勝手に *テンプレート (template)* と呼んでいますが、JavaScript でテンプレートが有効な場合は非常に限られています。テンプレートは元々型に厳密な C++ の機能ですが JavaScript の変数には型が無く、一度宣言された変数にはどのような型のオブジェクトも格納できます。このため汎用的なプログラミングあまり意味が無いのです

しかし型をパラメータとしたクロージャは有用なものです。このテクニックは JavaScript のオブジェクト指向について解説する第3章でもう一度議論します

1.4 関数の変形

前節で一部だけ動作の異なる関数を幾つでも作成する方法を示しました。更に柔軟な関数の生成方法があれば素晴らしいと思いませんか? ここでは既存の (基本的な) 関数同士を合成したり、引数を固定したりして全く別の関数を生成する方法を紹介します

復習

その前に復習というか再認識しておいて頂きたいことがあります。関数の**意味**についてです。例えば以下のようなクロージャを使った簡単なものがあるとしましょう

```
// 足し算をする関数を返す
function plus(x, y) {
  return function() {
    return x + y;
  };
}

var f = plus(3, 4);
f(); // 7
```

この関数 (`plus`) は加算を行う関数を返しますが、この関数は普通の関数 (例えば `function plus(x, y) {return x + y;}`) と明らかに違っています。それは *引数を指定しただけでは加算演算はまだ行われない* という点です。つまり:

- 演算を遅らせることが出来る
- 後で何度でも同じ引数の組み合わせで呼び出すことが出来る

これらの特徴は関数は呼び出し時に自分のコードを実行するという基本的な性格も表しています。上記の例では関数 `f` は外側から見れば中身がどうなっているのか分かりませんが、`f()` とすることで関数を呼び出すことが、つまり `f` に込められた意味のあるコードが実行されるということです。そしてそれはいつでもよいし、呼び出し側が何であっても構わないのです。この当たり前のように思える関数の性質が本節で最も重要なことです

関数の合成

まず関数同士の合成から始めましょう。例えば数学で以下のような2つの関数があるとします

- f(x) = x * 2
- g(x) = x + 2

この2つは引数の数に注意すれば、組み合わせて使うことが出来ます。例えば g の結果を引数として f に適用する場合は f(g(x)) のようになります。そしてこれを新たな関数 h(x) = f(g(x)) として考えることが出来るでしょう。2つの関数から違う意味の関数 h(x) = (x + 2) * 2 が得られたわけです

この程度の数式であれば JavaScript でも表現できそうです。各関数を定義してみましょう

```
function f(x) { // f(x) = x * 2
  return x * 2;
}
function g(x) { // g(x) = x + 2
  return x + 2;
}
function h(x) { // h(x) = f(g(x))
  return f(g(x));
}

h(3); // 10
```

しかしこのような合成の組み合わせは幾らでも考えられるので、`h` のような関数をいちいち定義するのは面倒です。必要なときだけ生成できるようにしたいものです。例えばこんな風に使えたら便利です:

```
compose(f, g)(3); // f と g を合成して得られた関数を引数を付けて呼び出す
```

このような合成を行うコードを書くのは以外に簡単ですが、関数の引数リストに注意して下さい。上述の例では全て引数は1つでしたが、数によっては合成できない可能性があります (考え方として正しくないという意味で)。ここでは以下の5つの組み合わせを考えます

関数合成パターン

組み合わせ	実装
f(g(x))	compose_f_gx
f(g(x), h(x))	compose_f_gx_hx
f(g(x), h(y))	compose_f_gx_hy
f(g(x, y))	compose_f_gxy
f(g())	compose_f_g

まず compose_f_gx の実装を示します。クロージャを使えば簡単ですがよく分からない場合は前節、前々節に戻ってみてください

```
function compose_f_gx(f, g) {
  // 引数チェックはお好みで…
  if(!(f instanceof Function) || f.length != 1
      || !(g instanceof Function) || g.length != 1)
    throw "引数は1つの引数をとる関数でなければなりません。";

  return function(x) {
    return f(g(x));
  };
}

// 使う
compose_f_gx(f, g)(3);
```

どうってことないですね? では残りの4つも書いてみましょう (簡単のため引数チェックは省いてあります)

```
function compose_f_gx_hx(f, g, h) {
  return function(x) {
    return f(g(x), h(x));
  };
}

function compose_f_gx_hy(f, g, h) {
  return function(x, y) {
    return f(g(x), h(y));
  };
}

function compose_f_gxy(f, g) {
  return function(x, y) {
    return f(g(x, y));
  };
}

function compose_f_g(f, g) {
  return function() {
    return f(g());
  };
}
```

使用例が少なかったので関数合成の威力は分からなかったかもしれません。しかしこれらは次の引数束縛と組み合わせて使うことで非常に強力なツールとなります

引数束縛の概要

次は引数の束縛 (固定) です。概要を理解していただくために、また数学の関数に登場してもらいましょう

- f(x, y) = x + y

和を返す関数ですね。ここで *x* を3とかに固定すると、g(y) = 3 + y となり新しい関数が得られます。第1引数を固定したことで引数の数が *減少しています。*これが引数の束縛の簡単なバージョンです

引数を減らすだけでなく順序を入れ替えたりも出来ます。例えば h(x, y) = f(y, x) とすると第1引数と第2引数を入れ替えた関数が得られます。引数の束縛により、使いにくい関数を使い易いものに書き直すことが出来るのです

2項関数の引数束縛

引数の束縛はかなり強力ですが、束縛を自動化するための関数の実装も結構大変です。まずは簡単なものとして、束縛対象の関数が2引数、束縛は固定のみとして話を進めましょう。これだけでもそれなりに役に立つものです。まず以下のような、引数を束縛して欲しそうな、わざとらしい関数があるものとします

```
function plus(lhs, rhs) {        // 加算
  return lhs + rhs;
}
function minus(lhs, rhs) {       // 減算
  return lhs - rhs;
}
function multiplies(lhs, rhs) { // 乗算
  return lhs * rhs;
}
function divides(lhs, rhs) {     // 除算
  return lhs / rhs;
}
```

四則演算を行う関数群ですね。全て引数が2つであることに注意して下さい

さて、引数を束縛する関数を用意します。第1引数を束縛するものを `bind1st` 、第2引数を束縛するものを `bind2nd` とすると実装は以下のようになります。これもクロージャを使えば簡単です

```
function bind1st(f, x) {
  // 引数チェック
  if(!(f instanceof Function) || f.length != 2)
    throw "第1引数は二項関数でなければなりません。";

  return function(y) {
    return f(x, y);
  };
}

function bind2nd(f, y) {
  // 引数チェック
  if(!(f instanceof Function) || f.length != 2)
    throw "第1引数は二項関数でなければなりません。";

  return function(x) {
    return f(x, y);
  };
}
```

では実際に使ってみましょう。四則演算であれば何でも出来るはずです

```
bind1st(plus, 3)(4);  // 7 = 3 + 4
bind2nd(minus, 3)(4); // 1 = 4 - 3

var f = bind2nd(multiplies, 3); // f(x) = x * 3
f(2);  // 6 = 2 * 3
f(42); // 126 = 42 * 3
```

ここで関数合成も一緒に使ってみましょう

```
compose_f_gx(
  bind1st(divides, 30),
  bind2nd(plus, 3)
```

```
)(2);  // 6 = 30 / (2 + 3)

compose_f_gx(
  bind1st(plus, 5),
  compose_f_gx(
    bind1st(plus, 3),
    bind2nd(plus, 8)
  )
)(2);  // 18 = (5 + (3 + 8)) + 2
```

関数合成を使うことで数式における括弧を表現することが出来ます。これにより3項以上の計算も出来ます (3項以上の計算に2項演算の組み合わせを使うのは貧相な感じがしますか? 大抵のプログラミング言語では2項の演算しか出来ませんよ)。簡単のために数値計算の例ばかり挙げましたが、引数が2つの関数であれば何にでも適用できます

引数リストに非依存の束縛

上で示した実装は引数が2つでないと使えません。これはつまらない制限なので、是非引数総数に依存しないバージョンを考えましょう

引数の束縛には色々ありますが、以下に示す4パターンに分けて考えます。見れば分かるとおりこれらは結局1つの同じ機能を言い表しているに過ぎません。しかし全ての機能をまとめた実装をいきなり紹介するのは難しいので敢えて分解しました。ここで _X という識別子は X 番目の引数を表すオブジェクトです

引数束縛パターン (func(a, b, c) という関数があるとして)

パターン	式	意味
引数の固定 (他引数以外による)	bind(func, _1, _2, 42)	第1引数、第2引数はそのままで、第3引数を数値 42 で固定。引数は1つ減少
引数の固定 (他引数による)	bind(func, _1, _2, _2)	第1引数、第2引数はそのままで、第3引数に第2引数と同じ値を渡す。引数は1つ減少
	bind(func, _1, _3, _2)	第1引数はそのままで、第2引数と第3引数を交換。引数の減少は無し
	bind(func, _1, _2 + 42, _2 + _3)	第1引数はそのままで、第2引数は常に数値 42 を足した値、第3引数は第2引数と足した値を使う。引数の減少は無し
引数の加工	bind(func, _1.method(arg), _2._1)	第1引数をオブジェクトとみなし、その *method* メソッドを引数付きで呼び出した結果を第1引数として渡す。第2引数もオブジェクトとみなし、第1引数で表現される名前を持つプロパティの値を第2引数として渡す

後ろに行くほど難しくなりますが非常に役立ちます (後で使用例を紹介します)。まず簡単な最初の3つを実装します。簡単といっても色々注意しなければならないことがあります

最初に引数を表現する _X の定義ですが、各オブジェクトは自分が何番目の引数であるかを知っておけばいいでしょう。オブジェクトの概要やコンストラクタの書き方については3章以降を見てください。ただしここでは簡単なものしか出てきません。定義は以下のようになります:

```
function Parameter(i) {
  this.i = i;
}

var /* const */ _1 = new Parameter(1); // 第1引数
var /* const */ _2 = new Parameter(2); // 第2引数
var /* const */ _3 = new Parameter(3); // 第3引数
...
```

引数を幾つまで定義するかは設計者の自由ですが、少な過ぎると実用に耐えられません。しかし多過ぎると実装者が泣くことになります (後で分かります)

limitedBind の実装

準備ができたので上の表の最初の3つを実現する limitedBind 関数を書いてみましょう。焦らずにまずは引数チェックまでです

```
// limitedBind(f : function, ...) : function
function limitedBind(f) {
  if(!(f instanceof Function))
    throw "第1引数は関数でなければなりません。";
  else if(arguments.length - 1 != f.length)
    throw "引数の数が正しくありません。";

  var arr = [];
  for(var i = 1; i < arguments.length; ++i) {
    if(arguments[i] instanceof Parameter) {
      if(arguments[i].i > f.length)
        throw "束縛対象関数に適合しない _" + arguments[i].i + "が見つかりました。";
      arr[arguments[i].i - 1] = true;
    }
  }
  for(var i = 0; i < arr.length; ++i) {
    if(arr[i] != true)
      throw "_X は _1 から連続していなければなりません。";
  }
```

最初に第1引数が関数であることを確認し、次に関数と残余引数の数が一致しているかどうかを調べています。その後のループは残余引数中に不正な `_X` が無いかを調べています。例えば3引数の関数に対して `_4` とか `_5` とかがあるとまずいわけです。ただしこのテストは高速化のためで、実際は配列 `arr` を使ったチェックの方が重要です。それが最後のループの部分で、`_X` が `_1` から連続して使われているかを調べています。例えば `limitedBind(func, _1, _2, _4)` とか `limitedBind(func, _2)` とかはまずいわけです

引数チェックが終わるといよいよ本題ですが、よく考えて書き始めないと途中で行き詰ってしまいます。呼び出し側の束縛の仕方で場合分けするのがよいでしょう。しかしその呼び出し側の**意図**を読み取るのが結構大変です。幾つか明らかなことがあるのでそれを整理しましょう

- 渡された引数が `Parameter` オブジェクトでなければ、他の値による引数の固定 → 引数が減少する
- 渡された引数が `Parameter` オブジェクトであれば評価の先送りを期待している。ただし引数が減少するかどうかはこれだけでは分からない
- 束縛後の関数の引数の個数は、渡された `Parameter` オブジェクト中最大の添え字に一致する

他にもありますがこれでいいでしょう。場合分けは3番目、つまり束縛後の関数の引数の個数で行うといいでしょう。あとはクロージャを使って関数呼び出しを行う関数を返します。`limitedBind` 後半部分のアウトラインは以下のようになります:

```
  // 続き
  var outerArgs = arguments;

  switch(arr.length) { // 結果の引数の個数で場合分け
  case 0:
    return function() {...};
  case 1:
    return function(a1) {...};
  case 2:
    return function(a1, a2) {...};
  ...
  }
  // ここに来ることはありません
}
```

`outerArgs` は `limitedBind` の実引数を参照します。クロージャ内では `arguments` 識別子は無名関数の実引数を指すので、名前を退避させているのです

各無名関数の引数リストに注目してください。当たり前ですがこの引数長は束縛済み関数の引数の総数と同じです。引数の上限をあまり大きな数に設定するとこの辺りで嫌な感じがしてきます。例えば上限が15であれば case 15 まで書かなければいけません。しかもこの後すぐ分かりますが、束縛済み関数の引数リストが長くなるほど case 文は長大になります

ここでは全ての場合の実装を示すのはやめて、引数の少ない最初の方だけ書いてみます。まず結果の引数長が0の場合ですが、この場合は _X は1つも使われていないはずです。全ての引数が Parameter 以外の値で固定されるような状況です。簡単ですがもう1段階場合分けが必要です。元の関数の引数長に基づく場合分けです

```
// outerArgs.length は limitedBind の実引数長だが
// f.length + 1 と同値であることは最初に確認した
switch(outerArgs.length - 1) {
case 0: return f();
case 1: return f(outerArgs[1]);
case 2: return f(outerArgs[1], outerArgs[2]);
case 3: return f(outerArgs[1], outerArgs[2], outerArgs[3]);
...
}
```

無名関数の中に switch 文を置くのは嫌だという場合は工夫してください。またこの程度の場合分けであれば eval と Function コンストラクタで何とかできるかもしれません

次は引数が1つになる場合です。この場合、_1 が (必ず) 登場します。_1 が現れた箇所は束縛済み関数の第1引数になる部分ですから、無名関数の引数 (a1) をそのまま使います。ここでも元の関数の引数長に基づく場合分けが必要になります

```
switch(outerArgs.length - 1) {
case 1: // limitedBind(func, _1) とかあまり意味の無い場合
  return f(a1);
case 2: // limitedBind(func, _1, n) とか limitedBind(func, n, _1)
  return f(
    (outerArgs[1] == _1) ? a1 : outerArgs[1],
    (outerArgs[2] == _1) ? a1 : outerArgs[2]);
case 3: // limitedBind(func, _1, _1, n) とか limitedBind(func, _1, n, _1)
  return f(
    (outerArgs[1] == _1) ? a1 : outerArgs[1],
    (outerArgs[2] == _1) ? a1 : outerArgs[2]
    (outerArgs[3] == _1) ? a1 : outerArgs[3]);
...
}
```

ちょっとややこしくなってきましたね。次は引数2つになるパターンで、_1 と _2 が出てきます。前と同じで a1 が _1 に、a2 が _2 に対応します

```
switch(outerArgs.length - 1) {
case 2: // limitedBind(func, _1, _2) とか limitedBind(func, _2, _1)
  return f(
    (outerArgs[1] == _1) ? a1 : a2,
    (outerArgs[2] == _1) ? a1 : a2);
case 3: // limitedBind(func, _1, _2, n) とか limitedBind(func, _2, n, _1)
  return f(
    (outerArgs[1] instanceof Parameter) ?
      arguments[outerArgs[1].i - 1] : outerArgs[1],
    (outerArgs[2] instanceof Parameter) ?
      arguments[outerArgs[2].i - 1] : outerArgs[2],
    (outerArgs[3] instanceof Parameter) ?
      arguments[outerArgs[3].i - 1] : outerArgs[3]);
...
}
```

case 3 で Parameter::i プロパティを使った式が出てきました。これが一般形で、これ以降はこのケースと同じ要領でコードを書くことになります

ある程度実装できたものとして使ってみましょう。limitedBind の使用には少し慣れが必要ですが、うまく使えるようになればかなり強力です。またわざとらしい例ですが:

```
function one(a) {
  return a;
}
function ten(a, b) {
  return a + 10 * b;
}
function hundread(a, b, c) {
  return a + 10 * b + 100 * c;
}

limitedBind(one, 42)();   // 42
limitedBind(one, _1)(42); // 42

limitedBind(ten, 6, 3)();       // 36
limitedBind(ten, _2, _1)(6, 3); // 63

limitedBind(hundread, _3, _2, _1)(3, 8, 6); // 386
limitedBind(hundread, _1, _1, _1)(6);       // 666

limitedBind(
  limitedBind(
    limitedBind(
      hundread, _3, _2, _1 // 1 <-> 3
    ), _2, _1, _3          // 1 <-> 2
  ), _1, _3, _2            // 2 <-> 3
)(3, 8, 6);                // 638
```

limitedBind と合成関数を組み合わせて使えば無限に関数を作成することができます

引数の加工

これで話が終われば気分がいいのですがまだ先があります。引数束縛の4番目と5番目のパターンの実装が残っています。他の3つのパターンでは引数をそのまま渡すことしかしませんでしたが、このパターンでは引数の束縛ではなく、引数に手を加えてから束縛対象の関数に渡します。例えば4番目のパターンは `bind(func, _1 + 5)` のような使い方を想定しています。これは第1引数に数値5を足してから `func` を呼び出して値を返すような関数を生成します。しかしこの式はコンパイルできません。`_1 + 5` がどうにもならないからです。`_1` は勝手に作ったオブジェクトですから、加算のための二項演算について JavaScript インタープリタは面倒を見てくれません。5番目のパターンについても同じことが言えます

4番目のパターンについてこの問題を解決するには非直感的な式を導入せざるを得ません。例えば足し算の例であれば `bind(func, _1.plus(5))` とか `bind(func, Lambda.plus(_1, 5))` といった感じです。しかしこの解決策は結局は使い物にならないことが分かるでしょう。前の3パターンでは引数はどんなオブジェクトでも使えましたが (`Parameter` のインスタンス以外)、加算演算などは数値や文字列でしか使わないのです。そんな限られた状況のために骨を折るのは嫌でしょう。ただ5番目のパターンについては解決策がそれほど適用範囲の狭いものではありません。例えば次のような使い方に落ち着くことができます

```
// print は文字列を出力する関数
bind(print, _1.toUpperCase()); // 常に大文字で出力する関数

// これを以下のように書く
bind(print, _1.invoke(String.prototype.toUpperCase, []));
```

この書き方もやはり自然とは言えませんが、算術演算子よりも一般的であることは確かです。ここで注目すべきは `_1.invoke()` の部分です。`bind` はこの呼び出しの戻り値を引数として受け取るので、「第1引数の String.prototype.toUpperCase メソッドを引数無しで呼んだ結果」ということを表現するためのオブジェクトを `_1.invoke` が返すようにすればいいでしょう。そのようなオブジェクトのコンストラクタは例えば以下のようになります:

```
function Invocation(i, method, args) {
  this.i_ = i;           // 何番目の引数か
```

```
    this.method_ = method; // 呼び出すメソッド
    this.args_ = args;     // その引数
}
```

Invocation オブジェクトが引数として渡されると、bind はその部分の引数が加工するものとして扱います。しかし bind のクライアントは実際には Invocation オブジェクトが生成されていることを知りません (ソースを読めば分かりますが)。勿論それでいいわけです。Parameter オブジェクトもそうですがクライアントがこれらの内部事情について知る必要は全くありません

この5番目のパターンの実装は非常に複雑であるので、本稿ではここまでとし、別の機会に紹介しようと思います。興味のある方は他の処理系や関数型言語を見て挑戦してください

関数の変形の応用例

例えば JavaScript で絵を描くアプリケーションを作っているとしましょう。あなたは各操作の実装を終えており、以下のような関数が揃えてあります

```
// ドットを打つ
function setPixel(x, y, color) {...}
// 直線を引く
function drawLine(x1, y1, x2, y2, color) {...}
// 長方形を描く
function drawRectangle(x1, y1, x2, y2, color) {...}
// 正円を描く
function drawCircle(x, y, r, color) {...}
// テキストを描く
function drawText(x, y, text, color) {...}
...
```

これらの関数はユーザの操作と対応しており、ユーザはコマンドという概念で各操作を捉えています。ここであなたはユーザの作業を自動化するマクロを実装しようとします。マクロは一連のコマンドの実行ですから、これらの関数を順番に呼び出すことで実装できます。この連続呼び出しを単純なループで実現するために、あなたは各コマンドに数値を割り当てて、配列に記憶するようにするかもしれません

```
for(var i = 0; i < commandList.length; ++i) {
  executeCommand(commandList[i]);
}
```

しかしこのままではコマンドの順番が分かるだけで、各コマンドの詳細が分かりません。例えばドットを打つコマンドなら、点の座標と色が必要になるはずです。つまり各コマンドは種類と付加情報で構成されるわけです。あなたはそのようなデータ構造で各コマンドを表現し、配列に押し込もうとしますがうまくいきません。付加情報の量や与え方は*コマンドによって異なるのです*。これは各関数の引数がバラバラであることが原因ですが、無理に引数の数を統一することはあまり良いことではありません。JavaScript 1.5 では引数の数はチェックされないので、コマンド ID と引数の配列を1セットにしたデータ構造というのもアリですが、色々大変です

ここで引数束縛の出番です。引数リストがバラバラであれば、必要な情報で束縛して引数を全部削ってしまえばいいのです。つまり必要な情報は全て関数に覚えておいて貰うのです

```
commandList.push(bind(setPixel, 50, 100, red));
commandList.push(bind(drawLine, 70, 100, 110, 100, blue));
...
```

コマンドリストには引数を受け取らない関数が収められることになります。マクロを実行するときには、このリストの各要素に対して**引数無しの関数呼び出し**をするだけです。付加情報のためのデータ構造やコマンド ID も必要ありません

引数が0の関数

関数の合成と引数の束縛について説明してきましたが、本節の最初で確認した関数の持つ意味を思い出してください。「呼び出しにより自分コードを実行する」のが関数の役目であり本質 (の1つ) です。変形によって新たに生み出された関数の持つ意味はオリジナルのものとどう違うのでしょうか。はじめに出てきた和を返す関数の意味は「2つの値を足す」でした。片方の引数を n で固定したものは「n ともう1つの値を足す」で、両方を n1 、n2 で固定したものは「n1 と n2 を足す」であると考えられます。引数が減少するにつれ、関数の意味に含まれる曖昧さが除去されていくのが分かると思います。引数の減少が限界まで進んだもの、つまり引数が0になった関数の意味は非常にはっきりしており、そこには曖昧さなどありません

抽象的なことを書きましたが、要するに引数の個数を0 (またはそれに近い値) にまで落とした関数は単一の命令とみなすことができます。それが先に述べたコマンドになるわけです

this 引数の束縛

本節で紹介したテクニックは関数に対して適用するものですが、勿論メソッドにも使えます。ところでメソッドには普通の関数とは異なり、適用オブジェクトを指す `this` 引数というものがあります。これは引数リストには現れませんが、メソッド中であれば `this` で参照でき、束縛も可能です。ただしここで考察した `limitedBind` はそのままでは使えません

`this` 引数がオブジェクトを指すのはそのオブジェクトに対してメソッド呼び出しを行う場合ですが、それ以外の場合、メソッドはどのオブジェクトにも結び付けられていない宙ぶらりの状態です。コンストラクタ `C` で生成したオブジェクトに対して呼び出すメソッドは `C.prototype` などに保持されているのが普通ですが (3章を読んでください)、別に他のオブジェクトについて呼び出しても構わないのです。これには ECMAScript 3rd Edition で導入された `Function::apply` 、`Function::call` メソッドを使います。汎用的な関数として `limitedBind` などに機能を追加しても良いのですが、ここでは別の関数として定義します。ここまで読んでこられた方なら実装は明らかでしょう

```
// bindThis(f : function, obj : object) : function
function bindThis(f, obj) {
  return function() {
    return f.apply(obj, arguments);
  };
}

// 使う
var n = 42;
bindThis(Object.prototype.toString, n)(); // "[object Number]"
bindThis(Number.prototype.toString, n)(); // 42
bindThis(String.prototype.toString, n)(); // エラー!
```

ライブラリの置き場所

本節では幾つか関数を作成しましたが、グローバルな名前空間ではちょっと格好が付かないと思われるかもしれません。実はいい場所があるのです。ここです:

```
Function.prototype.compose_f_gx = ...;
...
Function.prototype.bind = ...;
Function.prototype.limitedBind = ...;
Function.prototype.bindThis = ...;
```

私は全て第1引数が関数になるようにこれらの関数を設計しました (とはいえそれ以外にあまり無いのですが)。どうせならあらゆる関数オブジェクトからメソッドとして呼び出せるようにしてみると良いかもしれません

```
func.bind(_2, _1)();
func.bind(_1, _1, _1).compose_f_gx(otherFunc)(n);
String.prototype.toString.bindThis(text)();
```

3.1 プロパティとメソッド

オブジェクト指向が世に現れてから、プログラミングは大きく変わりました。今までのデータ中心アプローチからオブジェクト (object) と振る舞い (behavior) を念頭に置いた設計がなされるようになり、最近新しく登場した言語はオブジェクト指向として設計されています。オブジェクト指向プログラミングはコードの再利用、抽象化、分散を可能にし、現代の高速な開発サイクルにおいてよく使用されます

JavaScript はオブジェクト指向言語です。また、スクリプトであること、仕様が緩いことから他の言語には見られない高級さを達成しています。JavaScript に取り組んでいる間はオブジェクトから逃れることはできません。あなたがオブジェクトに接していないと思っていても、幾つものオブジェクトがスクリプトエンジン、ホストで活動しています。しかしそれらを「オブジェクト」として扱うことができれば何も難しいことは無いのです

この章では JavaScript におけるオブジェクト指向、特にどのようにすれば他の言語のようにオブジェクトを扱うことができるのかについて解説します。オブジェクト指向についての (言語非依存の) 理論や仕組みは既知のものとします

プロパティ

あなたが JavaScript にかじりついて1週間以上経っていれば *プロパティ (property)* がどういうものか、少なくとも JavaScript でどんな役割を持っているかご存知でしょう。プロパティはその名の通り、「特性」、「特質」といったもので、「オブジェクト O のプロパティ P」というと、「`O.P` は O というオブジェクトの特性 P を表す」となります。プロパティは常にオブジェクトに関連付けられており、普通はオブジェクトが無い状態でプロパティは存在できません

他の言語のプログラマは「でもプロパティなんて聞いたこと無い。データメンバ (メンバ変数、フィールド) じゃないの?」と言われるかもしれませんね。どちらもオブジェクトの内部表現を表すものですが、違いはどこにあるのでしょうか

実は「プロパティ」は JavaScript や VBScript などの超高級言語にしか無い概念なのです。データメンバやメンバ変数、フィールドが実データそのものであるのに対し、*プロパティはデータではありません*。プロパティの実体はゲッタやセッタといったデータメンバを直接操作するメソッド呼び出しであり、オブジェクトの「状態」ではなく「操作」なのです。超高級言語はこれらの公開演算を隠蔽し、スクリプトからはあたかも変数のように見えるようにしているのです。例えば以下のコードはプロパティに値を設定するものですが、内部では見た目以上に多くのことが行われています

```
var arr = new Array();
arr.length = 0; // セッタが呼び出され、
                // 新しい値 (0) がオブジェクトに通知され、
                // 値の正当性がチェックされ、
                // 配列長さが変化するので各要素のメモリを開放し、
                // ... 以下続く
```

まぁ、色々処理が行われますがゲッタとセッタというものがあることだけ覚えておいて下さい。読み取り専用のプロパティはセッタを用意しない (或いは用意しても何もしない) ことで実現できます。ECMAScript はゲッタとセッタをソースレベルで定義する方法を用意していませんが、JavaScript 1.5 や Windows スクリプトレットは独自の構文を使用してゲッタとセッタを定義できるようにしています。これらについては 3.4 アクセスレベル で解説します

以上のことからプロパティは純粋にオブジェクトの内部表現を表すものではないことが分かりますが、以降第4節までプロパティを単純にオブジェクトのデータとして扱うことにします。また、「読み取り専用」などのプロパティの挙動は言語レベルでは *属性 (attribute)* と呼ばれる概念で表現されます。コードレベルでこれらの属性を調べたり、設定したりすることはできません

Table 3.1 プロパティ属性 (*ECMAScript 3rd Edition 8.6.1 Property Attributes* より作成)

属性	説明
ReadOnly	プロパティは読み取り専用です。*コードレベルでの* 値の設定は無視されます。ただし「定数」や「永久に変更されない」ことを意味するのではありません。ホストなどがこのプロパティの値を変更することもあります
DontEnum	`for...in` ステートメントで列挙できません
DontDelete	`delete` 演算子でオブジェクトから削除できません
Internal	内部的に使用されるプロパティで名前がありません。また、コードレベルで直接アクセスすることはできません。この属性の付いた内部プロパティ、内部メソッドは **Table 3.2** の通りです

Table 3.2 内部プロパティ及び内部メソッド (*ECMAScript 3rd Edition 8.6.2 Internal Properties and Methods* より作成)

プロパティ	説明
[[Prototype]]	このオブジェクトのプロトタイプです
[[Class]]	このオブジェクトの種類を表す文字列です
[[Value]]	このオブジェクトの内部表現です
[[Get]](*PropertyName*)	*PropertyName* で表される名前のプロパティの値を返します
[[Put]](*PropertyName* , *Value*)	*PropertyName* で表される名前のプロパティに *Value* に設定します
[[CanPut]] (*PropertyName*)	*PropertyName* で表される名前のプロパティに [[Put]] 操作が可能であるかを返します
[[HasProperty]] (*PropertyName*)	このオブジェクトが *PropertyName* で表される名前のプロパティを持っているかどうかを返します
[[Delete]] (*PropertyName*)	*PropertyName* で表される名前のプロパティをオブジェクトから削除します
[[DefaultValue]](*Hint*)	オブジェクトのデフォルト値を返します
[[Construct]](...)	オブジェクトを作成します。この内部メソッドは new 演算子により呼び出されます。各コンストラクタがこのメソッドを実装します
[[Call]](...)	オブジェクトに設定されているコードを実行します。この内部メソッドは関数呼び出し式により呼び出されます。各関数がこのメソッドを実装します
[[HasInstance]](*Value*)	*Value* がその動作をこのオブジェクトに委譲しているかどうかを返します。ネイティブオブジェクトの中では Function オブジェクトだけがこのメソッドを実装しています
[[Scope]]	スコープです
[[Match]](*String* , *Index*)	正規表現マッチをテストし、結果を返します

内部プロパティ、内部メソッドは実際のコードには全く登場しないものですが、これらを意識するとスクリプトの表面上の動作も説明できるようになります。例えばスコープチェインやプロトタイプチェイン、コンストラクタなどは根本が分かっていないとなかなか理解の難しい要素です。以降の解説ではこれらのプロパティを使うことがあります

メソッド

JavaScript の *メソッド(method)* は呼び出し可能なプロパティに過ぎません。メソッドが普通の関数と異なるのは特定のオブジェクトを介して呼び出される点です。関数がメソッドとして呼び出された場合は関数中の this はそのオブジェクトを指すようになります

```
function foo() {
  return this;
}

var obj = new String();
obj.bar = foo;

foo();          // 最上位オブジェクト
obj.bar();      // obj
foo.call(obj); // obj
```

Function::apply 、Function::call メソッドを使うと好きなオブジェクトを介してメソッドを呼び出すことができます。これらのメソッドは ECMAScript 3rd Edition で採り入れられたもので、スクリプトエンジンがサポートしていない場合も考えられます。しかしこれらのメソッドは比較的簡単にエミュレートできます。Function::apply メソッドの簡単な実装は以下のようになるでしょう

```
Function.prototype.__apply = function(
    /* object */ oThis /* = null */, /* Array */ arrArgs /* = null */) {
  if(oThis == null || oThis == undefined) { // グローバルオブジェクトに適用
    if(arrArgs == null || arrArgs == undefined)
      return this();
    return this(arrArgs);
  }
  if(!(oThis instanceof Object))
    return undefined;   // 実際は throw TypeError()

  oThis.$00000000 = this;
```

```
  var oReturn;
  if(arrArgs == null || arrArgs == undefined)
    oReturn = oThis.$00000000();
  else if(!(arrArgs instanceof Array))
    return undefined;   // 実際は throw TypeError()
  else {
    // 関数呼び出し式を作る
    var strInvokeExpression = "oThis.$00000000(";
    for(var i = 0; i < arrArgs.length; ++i) {
      if(i != 0)
        strInvokeExpression += ",";
      strInvokeExpression += "arrArgs[" + i + "]";
    }
    strInvokeExpression += ")";
    var oReturn = eval(strInvokeExpression);
  }

  delete oThis.$00000000;
  return oReturn;
};
```

本節以降では `Function::call` メソッドが度々登場するので動作を覚えておいて下さい

コンストラクタ

HTML に JavaScript を埋め込む場合、`window` や `document` といったオブジェクトが最初から与えられます。プロパティやメソッドはオブジェクトがあって初めて使うことできます。DOM のようにトップレベルのオブジェクトが幾つか与えられ、そのオブジェクトから他の非公開オブジェクトが取得できるようになっているものを *オブジェクトモデル* といいます。オブジェクトモデルにおいてトップレベルオブジェクト以下にある各オブジェクト (`document` とか `alert`) は特別に作成する必要はありません

しかし、オブジェクトモデルに属さ**ない**オブジェクトも存在します。HTML ウェブページのために JavaScript でスクリプトを作成する大抵の人は、ホストオブジェクトとそのメソッド呼び出しだけでスクリプトを書きます。しかし JavaScript には ECMAScript で定義されたネイティブオブジェクトも用意されています。これらのオブジェクトを作成するには *コンストラクタ (ctor/constructor)* を呼び出して自分でオブジェクトを作成する必要があります

オブジェクトを利用するクライアントはオブジェクトを作成し、メソッドを呼び出すことでスクリプトを完成させます。ただ、HTML/DOM のオブジェクトは作成する必要が無いため、JavaScript の初心者はオブジェクトの作成を行う機会が少なくなっているのです。ECMAScript オブジェクトや自前で定義したオブジェクトを使うと実行環境の差を埋めやすくなります。これは HTML/DOM の実装がバラバラだからです。ECMAScript オブジェクトの仕様は比較的小さなもので、解釈の余地もあまりありません。自作オブジェクトについては仕様と実装を切り離すことができないため、正しくオブジェクトを定義すれば実行環境間の問題を完全に回避できます

JavaScript のオブジェクト指向における3番目の基本事項は「オブジェクトの作成 (インスタンス作成)」です。そしてオブジェクト作成のメカニズムは非常に簡潔なものです。JavaScript でオブジェクトを作成するのはコンストラクタであり、コンストラクタ自体は見た目はただの関数 (内部メソッド[[Construct]]を実装したオブジェクト) です。コンストラクタを呼び出すには普通の関数呼び出し式ではなく `new` 演算子を使います。例えば `String` コンストラクタを呼び出すには次のようにします

```
var obj = new String();
```

コンストラクタはオブジェクトの作成に成功すると新しく作成したインスタンスを返します。クライアントはこのインスタンスを使って仕事をするだけです

基本事項はこんなところです。次節からはオブジェクトを使うだけではなく、オブジェクトを定義する方法について述べます

3.2 コンストラクタとプロトタイプ

前節ではオブジェクトを作成して使う方法について述べました。あるオブジェクトについてこのような立場をとる人間、オブジェクトをクライアントといいます。クライアントは「顧客」とか「依頼人」という意味です。つまりオブジェクトか

ら見れば「お客」なのです。それ故オブジェクトはクライアントを中心に振舞わなければなりませんし、クライアントとの契約が守れないときはそれなりの対処をしなければいけません (「2. 例外」を参照)

この節で解説するのはオブジェクトを定義する方法です。JavaScript についてよく知らない多言語プログラマは「JavaScript はオブジェクトのクライアントとしてはまあまあだが、クラスを定義できないから駄目だ」と言うかもしれません。確かに JavaScript にはクラスを定義するための構文が存在しません。しかしそれは言語が貧弱だからではなく、JavaScript がクラスベースではなくプロトタイプベースの設計を選んだからです (JavaScript 2.0 ではクラスが使えますが)。プロトタイプ理論に従えば「オブジェクトを継承」させることも可能なのです。クライアントの前で仕事をするのはクラスではなくオブジェクトであることを忘れないで下さい

コンストラクタの定義

`new` 演算子を使ってコンストラクタ (とクライアントが考えているもの) を呼び出そうとするとコンストラクタの内部メソッド[[Construct]]が呼び出されます。このメソッドが呼び出される以下のように処理が行われます (*ECMAScript 3rd Edition 13.2.2 [[Construct]]* より作成)

1. オブジェクトを作成し、その内部プロパティ[[Class]]に `"Object"` を設定する
2. コンストラクタの `prototype` プロパティがオブジェクトであれば、作成されたオブジェクトの内部プロパティ[[Prototype]]に設定する。オブジェクトでなければ `Object::prototype` の値が使用される
3. コンストラクタの内部メソッド[[Call]]を呼び出す。引数には[[Construct]]呼び出しに使用されたものが使用される。また `this` 引数には新しく作成したオブジェクトを使う
4. [[Call]]呼び出しがオブジェクトを返した場合はそのオブジェクトを返す。そうでなければ最初に作成したオブジェクトを返す

これからコンストラクタの定義について述べるわけですが、そのコンストラクタのコードが実行されるのは上記の第3ステップです。つまり制御がコンストラクタ関数に入ったときには既にオブジェクトは作成されています。このオブジェクトは `this` 引数としてコンストラクタに渡されます。では実際にコンストラクタを記述してみましょう。別に難しいことはありません

```
function MyObject() {
  this; // 作成されたオブジェクト
}
```

普通は値を返す必要はありません (上記第4ステップ参照)。呼び出し側は次のようになるでしょう

```
var obj = new MyObject;
obj; // 作成されたオブジェクト
```

引数の括弧がありませんがこれは正しいものです。このようにするとコンストラクタに引数は渡されません

これだけではあまり面白くありません。コンストラクタがしなければならないことは「オブジェクトを作成すること」(概念的な話)、「オブジェクトのメンバを初期化すること」、「オブジェクトが作成できないときに何か対処すること」の3つです。素直な設計では作成はコンストラクタが呼び出されたときに終了しているので、ここでは後の2つについてコードを書くことにしましょう。メンバの初期化は `this` 引数にプロパティをくっつけるだけです

```
function MyObject(n) {
  this.myProperty = n; // プロパティ
}

var obj = new MyObject(7);
obj.myProperty; // 7
```

コンストラクタは関数でもあるので「1. 関数」で述べたテクニックが全て使えます。エラーハンドリングについては省略します

メソッドの定義

プロパティと同じ方法でメソッドも定義できるのではないでしょうか。例えば次のようなコードが考えられます

```
function MyObject() {
  this.myMethod = function() {
  };
}

var obj = new MyObject();
obj.myMethod(); // 呼び出す
```

このメソッド呼び出しは正しく動作しますが JavaScript のメソッド定義からは逸脱しています。例えば以下のコードを見てみましょう

```
var str = new String();
str.concat;                    // concat オブジェクト
str.hasOwnProperty("concat"); // false
```

このことから分かるように `String` インスタンスの `concat` メンバはインスタンスの直接のプロパティ (メソッド) ではないのです。しかしそうは見えません。では本当は何処にプロパティがあるのかというと `String::prototype` オブジェクトのプロパティになっています

```
String.prototype.hasOwnProperty("concat"); // true
```

つまり MyObject コンストラクタは次のようにする必要があったのです

```
function MyObject() {
  MyObject.prototype.myMethod = function() {
  };
  // 或いは次でも可
  this.constructor.prototype.myMethod = function() {
  };
}
```

ただ、1つのコンストラクタにプロトタイプは1つしか無いのでコンストラクタが呼び出される度にメソッドをセットするのは無駄です。次のようにコンストラクタから外してしまうのが一般的です

```
function MyObject() {
}

MyObject.prototype.myMethod = function() {
};
```

プロトタイプ

プロトタイプ (prototype) はコンストラクタオブジェクト毎に1つずつしか存在しないプロパティです。プロトタイプを使うと同じコンストラクタから作成されたインスタンス (厳密には同じ `prototype` プロパティを持つインスタンス) に共通のメンバを与えることができます。つまりプロトタイプは静的オブジェクト指向言語のクラスのような役割をしているのです。しかしプロトタイプの仕事はこれだけではありません

コンストラクタとプロトタイプは密接に結びついており、コンストラクタから作成されたオブジェクトはそのコンストラクタに固有なプロトタイプ (例えば `MyObject.prototype`) を参照します。この参照は内部プロパティ [[Prototype]]で普通はコード上には現れないものです (JavaScript 1.3 は `__proto__` プロパティとしてこの参照を公開しています)。そして上記の myMethod のようにコンストラクタのプロトタイプにセットされたメンバはこの参照を介して探索されます。具体的には以下のようなことが起こります

1. オブジェクトのメンバにアクセスしようとする。実装は与えられた名前のプロパティがオブジェクトにあるか調べる (上の例の場合は**無い**)
2. 見つからなかった場合はプロトタイプオブジェクトから与えられた名前のメンバを探す (上記の場合は MyObject.prototype.myMethod が見つかる)

3. それでも見つからない場合はプロトタイプのプロトタイプから探す
4. 以下、プロトタイプが無くなるまで続ける。プロトタイプが無い場合は内部プロパティ[[Prototype]]には `null` がセットされている

このようなプロトタイプの繋がりを勿体付けて *プロトタイプチェイン (prototype chain)* と呼びます。プロトタイプチェインを使うと同じプロトタイプチェインを辿ることのできるオブジェクトの間に階層的な関係を持ち込むことができます

では簡単な例で少し復習してみましょう。以下は人間オブジェクトの例です

```
// コンストラクタ
function Person(nAge) {
  this.m_nAge = nAge;
}

// 年齢を返すメソッド
Person.prototype.getAge = function() {
  return this.m_nAge;
};

var Don = new Person(22);    // 勿論これは冗談
var Exeal = new Person(21);

Don.getAge == Exeal.getAge; // true
Don.getAge();               // 22
```

最後の2行では Don インスタンスから getAge プロパティを探しますが、失敗します。次に Don のプロトタイプ Person::prototype (= Don.__proto__) から探し、プロパティを見つけます

クラスメンバ

クラスメンバ(静的メンバ) はインスタンス毎に与えられるメンバではなくクラス毎に存在するメンバです。JavaScript にはクラスが無いのでコンストラクタメンバとでも呼ぶことにしましょう。コンストラクタメンバは特定のインスタンスが無くても存在可能なメンバで `constructor` プロパティもこれに該当します

コンストラクタメンバの定義方法は非常に簡単です。コンストラクタにプロパティを設定するだけです

```
// コンストラクタ
function Person(nAge) {
  this.m_nAge = nAge;
  if(Person.m_nPopulation != undefined)
    ++Person.m_nPopulation;
}

Person.m_nPopulation = 0;

var Don = new Person(22);
var Exeal = new Person(21);

Person.m_nPopulation; // 2
Don.m_nPopulation;    // 残念ながらこれは駄目
```

継承

何度か書きましたが JavaScript にはクラスがありません。代わりにあるのはプロトタイプです。静的なオブジェクト指向言語ではクラス定義を継承しますが、プロトタイプベース言語ではプロトタイプベースの継承を行います

クラス *D* がクラス *B* のメンバを *継承 (inheritance)* すると、クラス *D* でメンバを定義しなくてもクラス *D* のインスタンスはクラス *B* のインスタンスが持つメンバを持つようになります。継承はオブジェクト指向における再利用性を実現している概念で、クラスベース言語ではクラスを *派生* させることでメンバの継承を行います

ではプロトタイプベース言語でメンバを継承するにはどうすればいいのでしょうか。例えばクライアントが次のようなコードを書けるようにしたいのです

```
function Person(nAge) {
  this.m_nAge = nAge;
}

Person.prototype.getAge = function() {
  return this.m_nAge;
};

function Programmer(nAge, strProject) {
  /* Person メンバの継承を実現するコード */
}

var Exeal = new Programmer(21, "EJS");
Exeal.getAge();
```

まずはプロパティの継承を考えましょう。以下のようにすれば巧くいくような気がします

```
function Person(nAge) {
  this.m_nAge = nAge;
}

function Programmer(nAge, strProject) {
  this.m_nAge = nAge;              // 真似する
  this.m_strProject = strProject; // 新しく追加するプロパティ
}
```

しかしこれでは2つのコンストラクタの間には何の関係もありません。また Person が変更されると Programmer も変更しなければなりません

(JavaScript の) コンストラクタの仕事にはプロパティの追加以外にインスタンスの初期化も含まれます。上のような簡単なコンストラクタはただ引数を代入しているだけですが、プロパティの数が増えたり複雑なものになると Programmer コンストラクタで同じコードを書くのは馬鹿らしくなります。それにこのままでは Person コンストラクタのクライアントである Programmer コンストラクタが Person の内部事情について先天的な知識を持つ必要があるので、オブジェクト指向の持つ再利用性を著しく欠いてしまいます。ちょっと嫌な方法ですがここでは Programmer コンストラクタから Person コンストラクタを呼び出してインスタンス初期化の一部委譲することにしましょう。新しく作成されたインスタンスに Person コンストラクタを適用するには以下の2つの方法があります

```
// 1つ目
function Programmer(nAge, strProject) {
  this.__super = Person;          // 新インスタンスを介して
  this.__super(nAge);             // 継承元コンストラクタを呼ぶ
  this.constructor = Programmer; // コンストラクタが Person にセットされるので元に戻す
  delete this.__super;
  /* Programmer コンストラクタの処理 */
}

// 2つ目
function Programmer(nAge, strProject) {
  Person.call(this, nAge);
  this.constructor = Programmer;
  /* Programmer コンストラクタの処理 */
}
```

2つ目の方法は ECMAScript 3rd Edition でのみ使用できます

さて、少々安直ですがプロパティの継承はできました。次はメソッドの継承です。メソッドの実体は Person.prototype に集められているのでこれを Programmer.prototype にコピーすれば良いような気がしますが、次のようにしても巧くいきません

```
/*
 * コンストラクタの定義...
```

```
 */

Programmer.prototype = Person.prototype;

Programmer.prototype.getProjectName = function() {
  return this.m_strProject;
};

var Exeal = new Person(21);
Exeal.getProjectName; // undefined でない!
```

プロトタイプをコピーしようとしても実際には参照されるだけです。Programmer.prototype は Person.prototype の中身を参照しているので、ここにメソッドを追加すると Person にも影響が及んでしまいます

勿論参照でなく純粋に複製できたとしても正解ではありません。プロトタイプが継承に使われるのはメンバ探索がプロトタイプチェインを辿るからです。プロトタイプを単純にメソッドの置き場所と考えてはいけません。関連するコンストラクタとプロトタイプは階層的であり、Programmer プロトタイプでの探索に失敗した後、Person プロトタイプを探索するようにしなければいけません。そのためには Programmer プロトタイプのプロトタイプが Person プロトタイプである必要があります。JavaScript なら次のようにすれば良いでしょう

```
Programmer.prototype.__proto__ = Person.prototype;
```

「でも ECMAScript 仕様外の機能は使いたくない」という声が聞こえてきそうですが、これ以外にも方法はあります。しかしこれもちょっと嫌な方法ですが次のようにします

```
Programmer.prototype = new Person();
```

直感的には分かりにくいかもしれないので順に見ていきましょう

1. Person インスタンスを作成し、Programmer コンストラクタのプロトタイプにセットする
2. Person インスタンスのプロトタイプは Person のプロトタイプである
3. Programmer.prototype のプロトタイプは Person のプロトタイプである
4. Programmer のプロトタイプのプロトタイプは Person のプロトタイプである

嘘みたいな方法ですがちゃんと動きます

```
Programmer.prototype = new Person();

var Exeal = new Programmer(21, "EJS");
Exeal.getAge();           // 21
Exeal.getProjectName();   // "EJS"
Exeal instanceof Person; // true
```

「ちょっと嫌な」と書いたのはコンストラクタ呼び出しが新インスタンスの作成以上のことをするからです。例えばコンストラクタメンバで使った m_nPopulation はメンバの継承で1になってしまいます。オブジェクトの構成が簡単であれば色々と対処できますが、継承元のクライアントは継承元のコンストラクタに対して、実際に実用インスタンスを作成するために呼び出したのかメソッドを継承したいだけなのかを通知する手段がありません。この継承の方法に問題が無いのは継承元のコンストラクタがそのように設計されているときだけです。そしてそのようなスペックは継承元が提示しない限りクライアントからは分かりません

この方法ではコンストラクタメンバまでは継承されませんがこれは妥当なことと思えます。コンストラクタメンバはコンストラクタごとに存在するのです

このように JavaScript における継承は継承元コンストラクタの呼び出しによるプロパティのセットと、プロトタイプによるメソッドの継承の2段階で構成されます

多重継承

いくつかのオブジェクト指向言語では多重継承が認められています。多重継承は1度に複数のクラスのメンバを継承します。例えばプログラマのクラスと弁護士のクラスからメンバを継承したクラスのインスタンスは、プログラマとしても弁護士としても活躍することができます

多重継承は便利なものですが使い方を間違えると (ほとんど正解が無い場合もある) 混乱を引き起こします。メンバ名の衝突やダイヤモンド継承などの問題から Java には多重継承がありません。そして JavaScript にも多重継承は *ありません*。これは多重継承にまつわる問題を懸念して仕様から外されたのではなく、プロトタイプチェインが1本の鎖状であることから必然的に不可能になっているものだと思います

オーバーライド

メソッドを継承するときに一部のメソッドの動作を変更したいことがあります。例えば人間を表す Person コンストラクタが以下のように定義されているとしましょう

```
function Person() {
}

// 眠る
Person.prototype.sleep = function() {
  this.goToBed();
};
```

この Person からメンバを継承して Programmer を作ると Programmer インスタンスでも sleep メソッドが使えます。しかしプログラマの寝床はベッドではありません。Programmer インスタンスの場合、「ベッドに行く」という振る舞いを変更しなければいけません。このためには Progammer.prototype にメソッドを設定し直します

```
function Person() {
}

Person.prototype = new Person();

// 眠る
Programmer.prototype.sleep = function() {
  this.relaxOnYourChair();
};
```

このようにすると Programmer インスタンスの sleep メソッドの呼び出しでは Programmer.prototype.sleep が呼び出されるようになります。このように継承先でメソッドの定義を変更することを *オーバーライド* といいます

オーバーライドされた元のメソッドを呼び出すには、つまり基底プロトタイプのメソッドを呼び出すには `Function::apply` か `Function::call` メソッドを使います

```
var Exeal = new Programmer();

// 今日はベッドで眠れそう
Person.prototype.sleep.call(Exeal);
```

ただしこの呼び出し自体は Programmer が Person を継承しているかどうかを何も考慮していません。Java には `super` キーワードがあり、基底クラスにアクセスできますが ECMAScript の仕様には基底コンストラクタをポイントするための仕組みがありません

プロパティはオーバーライドできません。継承先でプロパティを定義し直すと単純に上書きになります

残りの話題

JScript 、JavaScript 、ECMAScript におけるオブジェクト指向のオフィシャルな (?) 話題は以上のようなものです。JavaScript のメンバの種類、コンストラクタとプロトタイプ、プロトタイプチェインによる継承が可能になりました。これ

らを習得したことで (習得して頂けましたね?) あなたは JavaScript のオブジェクトを自由に操り、仕事をさせることができます。次節からは更にオブジェクトの実装側のテクニックを取り上げていくことにします

3.3 アクセスレベル

世の中には知らない方が良いことがあります。自動販売機ではお金を入れてボタンを押せば目的の商品は得られます。自動販売機の内部事情について、あなたの知識が皆無でも商品は手に入ります

オブジェクトの実装者が望むことはオブジェクトの機能をクライアントに提示することであり、クライアントが知りたいことはオブジェクトが持っている機能です。クライアントに公開されているオブジェクトであればあらゆるオブジェクトはこの原則に基づきます。オブジェクトとクライアントの間には (普通) これ以上の情報交換は必要ありません。この種の情報交換量を抑えるために実装側ができることは、オブジェクトの内部事情を隠してしまうことです。オブジェクトは自分が提供できる機能を含む最小限の情報だけを公開し、公開する必要の無い細かい実装はその存在自体をクライアントから隠蔽するのです

public メンバ、private メンバ

オブジェクトのメンバにはクライアントに公開されているものやクライアントに対して非公開になっているものがあります。*公開 (public) メンバ* についてクライアントは制限なしにアクセスできます。メソッドであれば自由に呼び出すことができ、プロパティであれば読み取り、或いは書き込みが可能です。これに対して *非公開 (private) メンバ* はクライアントからは一切アクセスできません。クライアントに提示する必要の無いメンバをこの private メンバにすることでクライアントから隠すことができます

JavaScript には非公開メンバを定義する公式の方法がありません。私が考えた代替案には以下のものがあります:

- 名前付け規約
- 専用の名前空間を使う
- コンストラクタ内で非公開メンバを var 宣言し、ゲッタ、セッタをオブジェクトにセットする
- コンストラクタ内で非公開メンバの集合を var 宣言し、参照テーブルを使う

以下、順に見ていくことにしましょう

名前付け規約

これは最もよく用いられる方法です。例えば非公開メンバの名前をアンダースコアで始めることにします

```
function Person(nAge) {
  this._nAge = Number(nAge);
}
Person.prototype.getAge = function() {
  return this._nAge;
};
```

安直な方法ですが意外に説得力があります。「アンダースコアで始まる名前のメンバはむやみに変更しないで下さい」というようなコメントが書かれていることもあります。ここまでされるとコードを見る側は private (とされている) メンバに直接アクセスしようなどと思わなくなります

また、ECMAScript の文法から逸脱したことは何もしていないので問題は全く起こりません

専用の名前空間を使う

私がよく使う方法です。仕組みは非常に簡単です。オブジェクトに _private という名前のプロパティを追加し、このオブジェクトに非公開メンバを定義します

```
function Person(nAge) {
  this._private = {
    m_nAge : nAge
  };
}
Person.prototype.getAge = function() {
```

```
  return this._private.m_nAge;
};
```

オブジェクトに直接含まれるプロパティが公開メンバというわけです。文法上の抑止力はありませんが、意図していることはすぐに分かるでしょう。メソッドを private にする場合は以下のようにするしかありません

```
Person.prototype._private = {
  getAge : function() {
    return this._private.m_nAge;
  }
};
Person.prototype.doSomethingWithAge = function() {
  Person.prototype._private.getAge.call(this);
  // ...
};
```

prototype と _private を逆にするのもありですが

コンストラクタ内で非公開メンバを var 宣言し、ゲッタ、セッタをオブジェクトにセットする

これはよく紹介される方法ですが、実際に使用されている例は多くありません。コードを見た方が早いでしょう

```
function Person(nAge) {
  var m_nAge = Number(nAge);
  this.getAge = function() {
    return m_nAge;
  };
}
```

コンストラクタ内で宣言された局所変数は同じく局所宣言された無名関数から参照されます。このためコンストラクタのスコープが終了しても m_nAge は生き続けます (そしてコンストラクタ呼出し毎に新しい局所変数が用意されます)

この方法の問題点はメソッドをプロトタイプではなく、オブジェクト自身にセットしていることです。これでは継承できませんし、ECMAScript のプロトタイプベースのセマンティクスから外れてしまいます。しかしこれらに目を瞑れば完璧に動作します

コンストラクタ内で非公開メンバの集合を var 宣言し、参照テーブルを使う

これはあまり見かけない例で、実際に使われているコードもあまりありません。この方法では private メンバの集合をコンストラクタ内で局所宣言します。この集合は同じコンストラクタから作成される全てのオブジェクトの非公開メンバをまとめたものです。この集合の永続性はクロージャを使えば保証できるでしょう (1つ前の方法と同じ)。あとはこの集合に含まれる private メンバとオブジェクトを対応させるだけです。具体的には以下のようになります

```
function Person() {
  var m_cInstances = 0;     // インスタンスの通し番号
  var m_arrInstances = []; // 通し番号 -> インスタンスの変換配列
  var m_arrPrivates = [];  // 通し番号 -> 非公開メンバの変換配列
  function findPrivate(oThis) { // インスタンス -> 非公開メンバの変換関数
    for(var i = 0; i < m_arrInstances.length; ++i) {
      if(oThis == m_arrInstances[i])
        return m_arrPrivates[i];
    }
    return null;
  }

  // コンストラクタを書き換える
  Person = function(nAge) {
    m_arrInstances[m_cInstances] = this;
    m_arrPrivates[m_cInstances++] = {
      m_nAge : Number(nAge)
```

```
    };
  };

  // メソッドの定義
  Person.prototype.getAge = function() {
    return findPrivate(this).m_nAge;
  };

} Person(); // コンストラクタの書き換えを実行
```

最初に4つの局所定義があります。コンストラクタ内にはこれらを参照するメンバがあるので、コンストラクタ終了後もこれらは有効です。ただしこの局所定義は1つあれば十分なもので、コンストラクタが呼び出される度に作成されるべきではありません。最初の呼び出しでのみこれらの局所宣言を評価させるには1回目の呼び出しでコンストラクタ自身を書き換えます。勿論コンストラクタを書き換えても局所定義は有効なままです。これでコンストラクタに1つしか存在しない変数が作成できました

実際に呼び出されるコンストラクタでは局所配列 m_arrInstances に自身を追加しています。そしてもう一方の配列 m_arrPrivates に非公開メンバのコレクションを追加しています。このときの添字はインスタンスの通し番号で、インスタンスが作成される度に1増加されます

メソッド内で非公開メンバを参照場合は局所関数 findPrivate を使います。この関数はインスタンスを非公開メンバコレクションに変換します。findPrivate の内部ではインスタンスを m_arrInstances から探し、その添字を使って m_arrPrivates からコレクションを取得します

コンストラクタ定義の後にコンストラクタを1回以上呼び出すのを忘れないで下さい。またメソッド定義の場所にも注意が必要です。メソッド定義は findPrivate か可視である位置、古いコンストラクタのスコープでなければなりません。且つコンストラクタ書き換えの後ろである必要があります

ただ非公開メンバを探すのに少し時間がかかるので効率はあまりよくありません (インスタンスの数に比例して悪くなります)。また、この方法や1つ前の方法では非公開メンバは実際にはインスタンスから離れた場所にあるため、`Function::call` メソッドなど他のオブジェクトを介してメソッドを呼び出すと不具合が生じます (ただし他のオブジェクトを適用できるかどうかはメソッドの設計者に決定権があります)

protected メンバ

1つ事がうまくいくと欲が出るものです。次は *限定公開 (protected) メンバ* について考えてみましょう。静的オブジェクト指向 (?) において限定公開メンバはそのクラスと派生クラスに対して公開されているメンバです。継承関係を持たないクライアントはアクセスできません

限定公開メンバについても正式に定義方法があるわけではありません。私が考えた代替案は次の3つです

- 名前付け規約
- 専用の名前空間を使う
- private 参照テーブルを継承先メソッドからも可視にする

以下、順に見ていくことにしましょう

名前付け規約

非公開メンバのときと同じです。例えばメンバ名を2つのアンダースコアで始めます

```
// 基底コンストラクタ
function Person(nAge) {
  this.__nAge = nAge;    // 限定公開メンバ
  this._oSpouse = null; // 非公開メンバ
}

// 派生コンストラクタ
function Programmer(nAge) {
  /* 基底コンストラクタの呼び出しは省略 */
}
Programmer.prototype = new Person;
Programmer.prototype.doSomethingWithMembers = function() {
  this.__nAge;    // ok
  this._oSpouse; // エラーが起こるわけではないが危険
```

```
};
```

この方法はあまり効果がありません。どっちがどっちか分からなくなります

専用の名前空間を使う

これは簡単で、見た目にも分かり易いものです

```
// 基底コンストラクタ
function Person(nAge_) {
  this._protected = { // 限定公開メンバ
    nAge : nAge_
  };
  this._private = {   // 非公開メンバ
    oSpouse : null
  };
}

// 派生コンストラクタ
function Programmer(nAge_) {
  /* 基底コンストラクタの呼び出しは省略 */
}
Programmer.prototype = new Person;
Programmer.prototype.doSomethingWithMembers = function() {
  this._protected.nAge;  // ok
  this._private.oSpouse; // エラーが起こるわけではないが危険
};
```

ただし限定公開メソッドを定義するのが困難になります

private 参照テーブルを継承先メソッドからも可視にする

非公開メンバのときに使ったテクニックはクロージャを使って、非公開メンバにアクセス可能なメンバを限定するというものでした。派生コンストラクタのプロトタイプにもメンバを公開すれば限定公開メンバが実現できそうです

考え方は非公開メンバと同じで、限定公開メンバを集めたテーブルを基底コンストラクタの古いコンストラクタに局所定義します。あとはこのテーブルを参照する局所関数 findProtected を継承先コンストラクタに公開するだけです。findProtected への参照を受け取った継承先コンストラクタはこの参照を含むメソッドを定義し、この参照自体も外から隠蔽するためにコンストラクタを書き換えます。以下に非公開メンバと限定公開メンバを使ったコードとモデル図を示します

```
function Person() {
  var m_cInstances = 0;       // インスタンスの通し番号
  var m_arrInstances = [];    // 通し番号 -> インスタンスの変換配列
  var m_arrPrivates = [];     // 通し番号 -> 非公開メンバの変換配列
  var m_arrProtecteds = [];   // 通し番号 -> 限定公開メンバの変換配列
  function findPrivate(oThis) { // インスタンス -> 非公開メンバの変換関数
    for(var i = 0; i < m_arrInstances.length; ++i) {
      if(oThis == m_arrInstances[i])
        return m_arrPrivates[i];
    }
    return null;
  }
  function findProtected(oThis) { // インスタンス -> 限定公開メンバの変換関数
    for(var i = 0; i < m_arrInstances.length; ++i) {
      if(oThis == m_arrInstances[i])
        return m_arrProtecteds[i];
    }
    return null;
  }

  // コンストラクタを書き換える
  Person = function(nAge, nSalary) {
```

```
    m_arrInstances[m_cInstances] = this;
    m_arrPrivates[m_cInstances] = {
      m_nAge : Number(nAge)
    };
    m_arrProtecteds[m_cInstances] = {
      m_nSalary : Number(nSalary)
    };
    this.findProtected = findProtected; // 継承先コンストラクタに参照を渡す
  };

  // メソッドの定義
  Person.prototype.getAge = function() {
    return findPrivate(this).m_nAge;
  };

} Person(); // Person コンストラクタの書き換えを実行

// 継承先コンストラクタ
function Programmer() {
  Person.call(this); // findProtected への参照を受け取る
  var findProtected = this.findProtected;

  // コンストラクタを書き換える
  Programmer = function(nAge, nSalary) {
    Person.call(this, nAge, nSalary); // 普段通りのプロパティの継承
    this.constructor = Programmer;
    this.findProtected = undefined;
  };

  // プロトタイプ継承もここで行う
  Programmer.prototype = new Person;

  // メソッドの定義
  Programmer.prototype.getSalary = function() {
    return findProtected(this).m_nSalary;
  };
} Programmer(); // Programmer コンストラクタの書き換えを実行
```

Fig. 3.1 非公開メンバと限定公開メンバのモデル

ここでも「クロージャ様様」なわけですが、少し複雑すぎるような気がしますね。少々目的がぼやけてきましたが、順に説明しましょう。この例では m_nSalary を限定公開メンバにすることを考えています。非公開メンバの場合と同様に参照テーブルに格納し、局所宣言された findProtected でのみアクセスできるようにしています。findProtected 関数はインスタンスを引数にとり、そのインスタンスの限定公開メンバの集合を返します

あとはこの関数を継承先から見えるようにすればいいのですが、グローバル変数に参照させると他からもアクセスされてしまいます。コンストラクタ外からこの関数を使おうとしても、実際には目的のインスタンスが必要なのですが、ここで使われている方法は関数自体を外に漏らさないようにしています。こういう場合は単純にソースコード上で静的にスコープの関係を考えても駄目です。重要なのは継承先コンストラクタのメソッドが定義されるコンテキストで、この局所変数への参照が存在していれば良いという点です。メソッドを定義するタイミングを一瞬のものとするテクニックは前にも出てきました。コンストラクタの書き換えです

書き換える前のコンストラクタに findProtected への参照を落とし、この一時的に有効なスコープでメソッドを定義してしまいます。復習ですがメソッド内の `findProtected(this)...` という記述はコンストラクタが書き換えられた後でも有効です

次に findProtected の参照を継承先の古いコンストラクタに渡す方法ですが、ここで Programmer と Person の間の関係をもう一度考える必要があります。私は Programmer の古いコンストラクタから Person の新しいコンストラク

タを呼び出し、findProtected への参照を得ています。この方法自体はトリッキーなものですが、Programmer が Person のメンバを継承していることとはあまり関係ありません。他の全く関係無いコンストラクタが同じ方法を使って findProtected への参照を獲得できてしまうわけです (この関数の起動には引数に Person コンストラクタを通過したインスタンスが必要ですが)。これに対処する方法は色々ありますが、ここでは何もしていません

今回はマイナーなテクニックばかりになってしまいました。クロージャの使い方はよく理解しておいて下さい。次節でもクロージャを使ったテクニックを紹介します