JavaScript におけるオブジェクト指向の特徴

「クラス」ではなくて「プロトタイプ」

- JavaScript には Java や C# などにおける「クラス」がない
- 「プロトタイプ」とは「あるオブジェクトの元となるオブジェクト」
- 「縛りのゆるいクラス」のようなもの

もっともシンプルなクラス

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


//インスタンス化
var mem = new Member();
```

JavaScript では関数（Function オブジェクト) に
クラスとしての役割を与えている

コンストラクタで初期化

```
var Member = function (firstName, lastName){
 this.firstName = firstName;
 this.lastName = lastName;
 this.getName = function(){
  return this.lastName + ' ' + this.firstName;
 }
};

var mem = new Member('太郎', '山田');
console.log(mem.getName()); // 山田 太郎
```

ここで重要なこと

- this
  - コンストラクタによって生成されるインスタンス（自分自身）を表すもの
  - this に対して変数を指定することでプロパティを指定
- メソッド
  - 厳密にはメソッドという概念はない
  - 値が関数オブジェクトであるプロパティがメソッドとみなされる

動的にメソッドを追加

```
var Member = function (firstName, lastName){
 this.firstName = firstName;
 this.lastName = lastName;
};

var mem = new Member('太郎', '山田');
mem.getName = function(){
  return this.lastName + ' ' + this.firstName;
 }
console.log(mem.getName()); // 山田 太郎
```

インスタンスに対して直接メンバを追加

```
var Member = function (firstName, lastName){
 this.firstName = firstName;
 this.lastName = lastName;
};

var mem = new Member('太郎', '山田');
mem.getName = function(){
  return this.lastName + ' ' + this.firstName;
 }
console.log(mem.getName()); // 山田 太郎

var mem2 = new Member('花子', '鈴木')
console.log(mem2.getName()); // ?
```

注意点

- 同一クラスを元に生成されたインスタンスであっても、それぞれが持つメンバは同一であるとは限らない
- これが「縛りの弱いクラスのようなもの」の理由

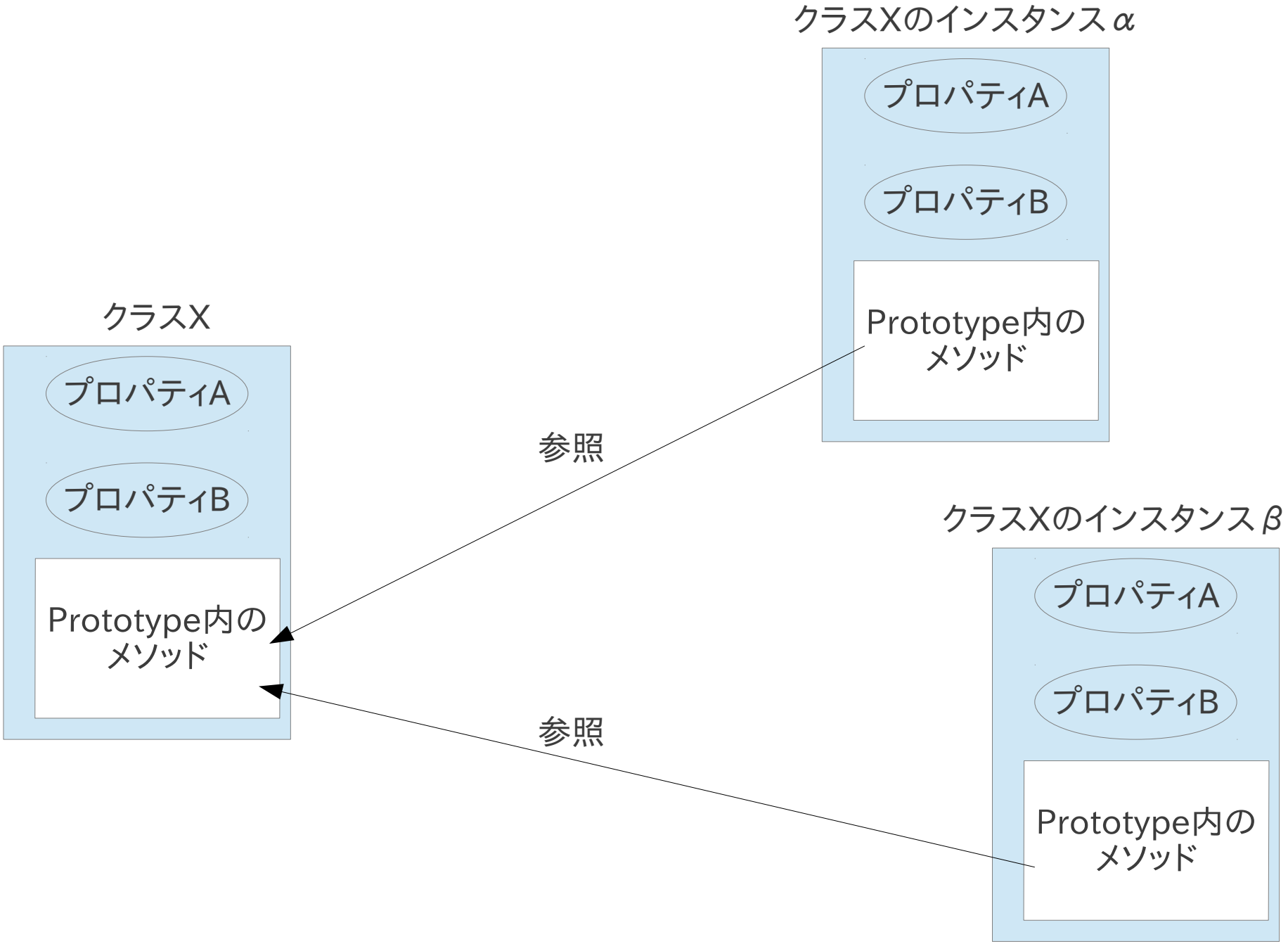
コンストラクタの問題点と
プロトタイプ

メソッドはプロトタイプで宣言する - prototype プロパティ-

- コンストラクタによるメソッドの追加には
メソッドの数に比例して、「無駄なメモリ」を消費する
- コンストラクタはインスタンスの生成のたびに
それぞれのインスタンスのためにメモリを確保
- メソッドはすべてのインスタンスで中身が同じ
→その分のメモリ確保は無駄

メソッドはプロトタイプで宣言する - prototype プロパティ-

- オブジェクトにメンバを追加するために、prototype というプロパティを用意
- Prototype に格納されたメンバは、インスタンス化されたオブジェクトに引き継がれる
- 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


クラスXのインスタンスα
プロパティA
プロパティB
Prototype内の
メソッド
クラスX
プロパティA
プロパティB
Prototype内の
メソッド
参照
クラスXのインスタンスβ
プロパティA
プロパティB
Prototype内の
メソッド
参照

コード例

```
var Member = function (firstName, lastName){
 this.firstName = firstName;
 this.lastName = lastName;
};
Member.prototype.getName = function(){
  return this.lastName + ' ' + this.firstName;
 }
var mem = new Member('太郎', '山田');
console.log(mem.getName()); // 山田 太郎
```

プロトタイプオブジェクトを利用することの2つの利点

- メモリの使用料を節減できる
  - メンバが呼び出された際の処理
    - インスタンス側に要求されたメンバが存在しないか確認
    - 存在しない場合は、暗黙参照をたどって検索
- メンバの追加や変更をインスタンスがリアルタイムに認識できる

コード例

```
var Member = function (firstName, lastName){
 this.firstName = firstName;
 this.lastName = lastName;
};
var mem = new Member('太郎', '山田');
Member.prototype.getName = function(){
  return this lastName + ' ' + this.firstName;
 }
console.log(mem.getName()); // 山田 太郎
```

インスタンス化したあとに、メソッドを追加している点に注目

プロトタイプオブジェクトの不思議(1)
- プロパティの設定 -

- プロトタイプオブジェクトでプロパティを宣言したら？

```
var Member = function (){};
Member.prototype.sex = '男';

var mem1 = new Member();
var mem2 = new Member();
console.log(mem1.sex + '|' + mem2.sex); //男|男
mem2.sex = '女';
console.log(mem1.sex + '|' + mem2.sex); //男|女
```

Prototype オブジェクトが利用されるのは「値の参照時だけ」

静的プロパティ/メソッドの定義

- オブジェクト名.プロパティ名 = 値;
- オブジェクト名.メソッド名 = function(){
  /*メソッドの定義*/
  }

```
var Area = function() {} ; //コンストラクタ

Area.version = '1.0';

//静的メソッド triangle の定義
Area.triangle = function(base, height) {
 Return base * height /2 ;
}
//静的メソッド diamond の定義
Area.diamond = function (width, height ){
 return width * height /2
}

console.log('Areaクラスのバージョン： + Area.version); // 1.0
console.log('三角形の面積 : '+ Area.triangle(5, 3)); //7.5
var a = new Area();
console.log('菱形の面積: ' + a.diamond(10, 2) ) //エラー
```

静的プロパティ/メソッドを定義するときの2つの注意点

- 静的プロパティは基本的に読み取り専用
- 静的メソッドの中では this キーワードは使えない
  - インスタンスメソッド内では this はインスタンス自身を指す
  - 静的メソッド内では this はコンストラクタを表す
    - 静的メソッド内からインスタンスプロパティにはアクセスできない

プロトタイプオブジェクトの不思議 プロパティの削除

- 削除する場合にも「インスタンスの単位で行われる」ことに注目

```
var Member = function (){};
Member.prototype.sex = '男';

var mem1 = new Member();
var mem2 = new Member();
console.log(mem1.sex + '|' + mem2.sex); //男|男
mem2.sex = '女';
console.log(mem1.sex + '|' + mem2.sex); //男|女
delete mem1.sex
delete mem2.sex
console.log(mem1.sex + '|' + mem2.sex); //男|男
```

オブジェクトリテラルでプロトタイプを定義

- 利点
  - Member.prototype. ～　のような記述を最小限に抑えられる
  - オブジェクト名に変更があった場合にも影響が少ない
  - 同一オブジェクトのメンバ定義がひとつのブロックに収まっているので可読性が上がる

コード例

```
var Member = function (firstName, lastName){
 this.firstName = firstName;
 this.lastName = lastName;
}

Member.prototype.getName = function(){
  return this lastName + ' ' + this.firstName;
 }
Member.prototype.toString = function(){
  return this lastName +  this.firstName;
 }
```

```
 var Member = function (firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;
 }

 Member.prototype = {
   getName : function(){return this lastName + ' ' + this.firstName;}
   toString   : function(){return this lastName +  this.firstName;}
 }
```