ソフトウェア開発
第5回目授業

平野 照比古

2015/10/23

レポート課題 3 について

単純に付け加えただけでは失敗する。

```
var Person2 = (function (){
  var ID = 0;
  return function(){
    this.ID = ID++;
    this.name = "foo";
    Object.defineProperty(this, "ID",
      {get : function(){return this.ID},
       enumerable:true,
       configurable:false
      });
  }
})();

>p = new Person2();
Object {name: "foo"}
>p.ID
Uncaught RangeError: Maximum call stack size exceeded(…)
```

レポート課題 3 について–エラーの理由

スタックオーバーフローが起きている。
原因は `Object.defineProperty()` 内の関数の戻り値 `this.ID` が `this.ID = ID++`を指しているのではなく、この関数自体を指しているため

レポート課題 3 について–解決方法

オブジェクト内に変数を用意する

```
var Person2 = (function (){
  var ID = 0;
  return function(){
    var mID = ID++;
    this.name = "foo";
    Object.defineProperty(this, "ID",
      {get : function(){return mID},
       enumerable:true,
       configurable:false
      });
  }
})();
```

prototype、class と extensible という３つの属性がある。

class 属性

- class 属性はオブジェクトの型情報を表す文字列
- 最新の ECMAScript 5 でもこの属性を設定する方法はない。
- 直接値を取得する方法もない。
- クラス属性は間接的にしか得られない。
- 組み込みのコンストラクタで生成されたオブジェクトではそのクラス名が間接的に得られる
- 独自のコンストラクタ関数では、"Object"しか得られない。

class 属性–実行例

```
function Person(){
  this.name = "foo";
  this.birthday = {
    year : 2001,
    month : 4,
    day : 1
  };
  this["hometown"] = "神奈川";
}
```

class 属性–toString

Object から継承した toString() を直接呼び出すと次のような結果になる。

```
>p = new Person();
Person {name: "foo", birthday: Object, hometown: "神奈川"}
>p.toString();
"[object Object]"
>p +"";
"[object Object]"
```

Person() コンストラクタには toString() メソッドが定義されていないので、もともとの Object で定義されている toString() が呼び出されている。

他のオブジェクトのメソッドを自分のメソッドのように呼び出す

Object の prototype 属性に定義されている toString を引数のメソッドとして利用する

```
>Object.prototype.toString.call([]);
"[object Array]"
>Object.prototype.toString.call(null);
"[object Null]"
>Object.prototype.toString.call(undefined);
"[object Undefined]"
>Object.prototype.toString.call(NaN);
"[object Number]"
>Object.prototype.toString.call(window);
"[object global]"
>window+""
"[object Window]"
```

extensible 属性

extensible はオブジェクトに対してプロパティの追加ができるかどうかを指定

- ECMAScript 5 ではこの属性の取得や設定ができる関数が用意されている。
- この属性の取得は Object.isExtensible() に調べたいオブジェクトを引数にして渡す。
- オブジェクトのプロパティを拡張できなくするためには Object.preventExtension() に引数として設定したいオブジェクトを渡す。

extensible 属性の変更—実行例

```
>p = new Person();
Person {name: "foo", birthday: Object, hometown: "神奈川"}
>p.mother = "aaa"
"aaa"
>p.mother
"aaa"
```

Object.preventExtensions(p) を実行する前では存在しない属性の追加ができる。

extensible 属性の変更—実行例 (続き)

```
>Object.preventExtensions(p);
Person {name: "foo", birthday: Object, hometown: "神奈川", mot
>p.grandmother = "AAA";
"AAA"
>p.grandmother;
undefined
```

設定後は、新しい属性が定義できていない。

extensible 属性の変更—実行例 (つづき)

```
>p.mother = "bbb";
"bbb"
>p.mother;
"bbb"
>delete p.mother;
true
>p.mother
undefined
```

存在するプロパティは値の変更や削除が可能

属性の削除の禁止

属性の削除まで禁止したい場合には `Object.seal()` を用いる。

- 一度この関数を実行されたオブジェクトは解除できない。
- すでにその状態になっているかどうかは、`Object.isSealed()` を用いる。
- 書き込み可の属性の値は変えることができる。

属性の削除の禁止—実行例

```
>p = new Person();
Person {name: "foo", birthday: Object, hometown: "神奈川"}
>Object.isSealed(p);
false
>Object.seal(p);
Person {name: "foo", birthday: Object, hometown: "神奈川"}
>Object.isSealed(p);
true
>p.mother = "aaa";
"aaa"
>p.mother;
undefined
```

Object.seal() を実行した後、存在しない属性 p.mother は定義されていない

属性の削除の禁止—実行例 (続き)

```
>p.hometown
"神奈川"
>p.hometown = "Japan"
"Japan"
>p.hometown
"Japan"
```

既存の属性の値は変更可能

オブジェクトの固定化

オブジェクトを拘束するためには、Object.freeze() を用いる。
この状態を確認するためには Object.isFrozen() を用いる。

オブジェクトの固定化—(実行例)

```
>Object.isFrozen(p);
false
>Object.freeze(p);
Person {name: "foo", birthday: Object, hometown: "Japan"}
>p.hometown = "Tokyo"
"Tokyo"
>p.hometown;
"Japan"
```

- p.hometown の値が設定できていない
- この状態で属性値を変えたい場合には、属性にたいするセッターメソッドを定義する
- Object.seal() や Object.freeze() の影響は、渡されたオブジェクト自身の属性にしか影響を及ぼさない。
- 継承元のオブジェクトには影響を及ぼさない。

prototye 属性

- オブジェクトの prototype 属性の値は、同じコンストラクタ関数で生成された間で共通のもの
- オブジェクトリテラルで生成されたオブジェクトは Object.prototype で参照できる
- new を用いて生成されたオブジェクトはそのコンストラクタ関数の prototype を参照
- コンストラクタ関数の prototype もオブジェクトであるから、それの prototype も存在

この一連の prototype オブジェクトをプロトタイプチェーンとよぶ。

Object.create()

- ECMAScript 5 で定義されている Object.create() メソッドは引数で与えられたオブジェクトの prototype を prototype に持つオブジェクトを生成
- このメソッドに対して 2 番目の引数を与えて、新たに生成されたオブジェクトのプロパティを指定できる。

基本のオブジェクト

```
function Person(){
  this.name = "foo";
  this.year = 2001;
  this.month = 4;
  this.day = 1;
  this.toString = function(){
    return "私の名前は"+this.name+"です";
  }
  this.showBirthday = function() {
    return this.name+"の誕生日は"+
      this.year+"年"+this.month+"月"+this.day+"日です";
  }
}
```

基本のオブジェクト—解説

- 6 行目から 8 行目ではオブジェクトを文字列に変換する必要ができたときに呼び出される `toString()` メソッドを定義
- 9 行目から 12 行目では誕生日をわかりやすい形で表示する `showBirthday()` メソッドを定義

基本のオブジェクト—実行例

```
>p=new Person();
Person {name: "foo", year: 2001, month: 4, day: 1}
>p +"";
"私の名前は foo です"
>p.showBirthday();
"foo の誕生日は 2001 年 4 月 1 日です"
```

基本のオブジェクト—問題点

- このコードをみるとコンストラクタ関数が呼ばれるごとに、2 つのメソッドのコードが繰り返し使用されている
- 同じコンストラクタ関数から生成さるオブジェクトに対して共通するプロパティやメソッドはコンストラクタ関数の prototype に移動した方が効率が良い。

基本のオブジェクト—改良

```
function Person2(name, y, m, d){
  this.name = name;
  this.year = y,
  this.month = m,
  this.day = d
}
```

基本のオブジェクト—改良 (続き)

```
Person2.prototype = {
  toString : function(){
      return "私の名前は"+this.name+"です";
    },
  get age(){
      var Now = new Date();
      var Age = Now.getFullYear() - this.year;
      if((Now.getMonth()+1) < this.month) {
        Age--;
      } else {
        if((Now.getMonth()+1) == this.month &&
           Now.getDate() < this.day) Age--;
      }
      return Age;
    },
  get birthday() {
      return this.year+"年"+this.month+"月"+this.day+"日";
    }
```

基本のオブジェクト—改良 (解説)

- prototype はオブジェクトリテラルの形式で定義
- 8 行目から 10 行目では toString() メソッドを定義
- 11 行目から 21 行目では、age プロパティを定義
  function キーワードの代わりに get を用いることで、ゲッターとして定義している
  この関数は実行時におけるオブジェクトの満年齢を得る
  - 12 行目で実行時の日時を求めている。
  - 13 行目で実行時の年から、誕生日の年の差を求めている。
  - 14 行目では実行時の月と誕生日の月を比較し、誕生日の月を過ぎていなければ、年齢を 1 だけ減らす。
    Date.getMonth() は 0(1 月) から 11(12 月) の値を返すことに注意
  - 17 行目では実行時の月と誕生日の月が同じ時に、両者の日を比較して、誕生日前か判定
- 22 行目から 24 行目では birthday プロパティをゲッターとして定義

基本のオブジェクト—改良 (実行結果)

```
>p2 = new Person2("me",1995,4,1);
Person2 {name: "me", year: 1995, month: 4, day: 1}
>p2.birthday;
"1995 年 4 月 1 日"
>p2.age;
20
>p.age=30;
30
>p2.age;
20
>Person2.prototype.constructor
Object() { [native code] }
```

基本のオブジェクト—改良 (実行結果解説)

- p.age にはセッターが定義されていないので見かけ上の代入を行っても無効
- Person2.prototype.constructor の結果が Person2 になっていないのは、7 行目で Person2.prototype に代入しているいオブジェクトに constructor プロパティがないからである。
- 通常はこれには問題があるので代入するオブジェクト内に次のような記述を追加

  ```
  constructor : Person2,
  ```

- 別の方法としては、Person2.prototype.constructor = Person2; の行を追加

get キーワードの代わりに set キーワードを用いるとセッターを定義できる。

オブジェクトの継承

- オブジェクトの継承とは既に存在するオブジェクト (クラス) に対して機能の追加や修正を行って新しいオブジェクト (クラス) を構成すること
- JavaScript ではクラスをサポートしていないので厳密な意味での継承はできない
- `prototype` を用いることでメソッドや共通プロパティの継承が可能

継承の例

Person2 を継承して、学籍番号を追加のプロパティとする Student オブジェクトを構成

```
function Student(n, id, y, m, d){
  this.name   = n;
  this.year   = y;
  this.month  = m;
  this.day    = d;
  this.id     = id;
}
Student.prototype = new Person2();
Student.prototype.constructor = Student;
```

継承の例—解説

- name などのプロパティはオブジェクトごとに違う値をとるので Person2.rototype 内には置くことができない。したがって、それぞれを this のプロパティに格納 (2 行目から 6 行目)
- Person2 の prototype を利用するために、Student.prototype にオブジェクトを新規に作成して代入 (8 行目)。
- これにより、この後で Person2 のプロパティが変更されても、Student オブジェクトには影響がでない。
- Student.prototype.constructor を Student に戻しておく (9 行目)

継承の例—実行結果

```
>s = new Student("me",1323300,1995,4,1)
Student {name: "me", year: 1995, month: 4, day: 1, id: 132330(
>s.age;
20
>s.name
"me"
>s+"";
"私の名前は me です"
>s.constructor;
Student(n, id, y, m, d){
  this.name   = n;
  this.year   = y;
  this.month  = m;
  this.day    = d;
  this.id     = id;
}
```

エラーオブジェクトとは

- エラーが発生したことを知らせるオブジェクト
- 通常は計算の継続ができなくなったときにエラーオブジェクトをシステムに送る操作が必要
- これをエラーを投げる (throw する) という

エラーオブジェクトには表 1のようなプロパティがある。

Table: エラーオブジェクトのプロパティ

プロパティ	説明
message	エラーに関する詳細なメッセージ。コンストラクタで渡された文字列か、デフォルトの文字列
name	エラーの名前。エラーを作成したコンストラクタ名になる

エラー処理の例

コンストラクタに与えられた引数をチェックして不正な値の場合にはエラーを投げるように書き直したもの

```
function Person(name, y, m, d){
  if(name === "") throw new Error("名前がありません");
  this.name = name;
  this.year = y;
  if(m<1 || m>12) throw new Error("月が不正です");
  var date = new Date(y,m,0);
  if(d<1 || d>date.getDate()) throw new Error("日が不正です");
  this.month = m,
  this.day = d
}
```

エラー処理の例—解説

- 2 行目で、name が空文字であればエラーを発生
- 5 行目では月の値の範囲をチェック
- 6 行目では、与えられた年と月からその月の最終の日を求めている。Date.getMonth() の戻り値が 0(1 月) から 11(12 月) になっているので、new Date(y,m,0) により翌月の 1 日の 1 日前、つまり、問題としている月の最終日が設定できる
- 7 行目で与えられた範囲に日が含まれていなければエラーを発生

エラー処理の例—実行例

- 通常の日時ならば問題なく、オブジェクトが構成される。

```
>p = new Person("foo",1995,4,1);
Person {name: "foo", year: 1995, month: 4, day: 1}
```

- 1996 年はうるう年なので 2 月 29 日が存在する。したがって、エラーは起こらず正しくオブジェクトが作成できる。

```
>p = new Person("foo",1996,2,29);
Person {name: "foo", year: 1996, month: 2, day: 29}
```

- 1995 年はうるう年ではないので 2 月 29 日がない。したがって、エラーが起きる。

エラー処理の例—実行例 (続き)

不正な月や日では当然、エラーが起こる。

```
>p = new Person("foo",1995,2,29);
 Uncaught Error: 日が不正です (…)
>p = new Person("foo",1995,13,29);
 Uncaught Error: 月が不正です (…)
>p = new Person("foo",1995,12,0);
 Uncaught Error: 日が不正です (…)
```

エラー処理の例—問題点

オブジェクトを継承するオブジェクトに対して、継承先のオブジェクトに対してエラーチェックの部分を再び書く必要がある。
エラーチェックの部分を関数化して、継承先でも同じようなチェックができるようにする。

エラー処理の例 (改良版)

```
function Person2(name, y, m, d){
  this.checkDate(y, m, d);
  this.checkName(name);
  this.name = name;
  this.year = y;
  this.month = m,
  this.day = d
}
```

エラー処理の例 (改良版)—続き

```
Person2.prototype = {
  toString : function(){
      return "私の名前は"+this.name+"です";
    },
  checkDate : function(y, m, d) {
      var date = new Date(y,m,0);
      if(m<1 || m>12) throw new Error("月が不正です");
      if(d<1 || d>date.getDate()) throw new Error("日が不正です");
    },
  checkName : function(name) {
      if(name === "") throw new Error("名前がありません");
    },
 get age(){
   ... // 内容は省略
    },
 get birthday() {
      return this.year+"年"+this.month+"月"+this.day+"日";
    }
```

エラー処理の例 (改良版)—解説

エラーチェックをする関数を作成して、それを Person2.prototype のなかに置いている。

- 2 行目と 3 行目でエラーチェックをする関数を呼び出している。この関数はエラーが起きたときはエラーを投げるので、コンストラクタ関数の制御から離れる
- それぞれのエラーチェックをする関数は 13 行目から 17 行目と 18 行目から 20 行目に記述

エラー処理の例 (改良版)—続き

Person2 継承した Student オブジェクト

```
function Student(n, id, y, m, d){
  this.checkDate(y, m, d);
  this.checkName(name);
  this.name   = n;
  this.year   = y;
  this.month  = m;
  this.day    = d;
  this.id     = id;
}
Student.prototype = new Person2();
Student.prototype.constructor = Student;
```

エラー処理の例 (改良版)—実行例

```
>s=new Student("foo",1223300,1995,12,1);
Student {name: "foo", year: 1995, month: 12, day: 1, id: 12233
>s=new Student("foo",1223300,1995,4,0);
 Uncaught Error: 日が不正です (…)
>s=new Student("foo",1223300,1995,13,1);
 Uncaught Error: 月が不正です (…)
```

エラー処理の問題点

- 前節の例ではエラーが発生するとそこでプログラムの実行が止まる
- エラーが発生したときに、投げられた (throw された) エラーを捕まえる (catch する) ことが必要
- try{...}catch{...} 構文を使用

エラー発生でプログラムを停止させない方法

try{...}catch{...} 構文を用いてオブジェクトが正しくできるまで繰り返す。

```
function test() {
  var y, m, d;
  for(;;) {
    try {
      y = Number(prompt("生まれた年を西暦で入力してください"));
      m = Number(prompt("生まれた月を入力してください"));
      d = Number(prompt("生まれた日を入力してください"));
      return new Person2("foo", y, m, d);
    } catch(e) {
      console.log(e.name+":"+e.message);
    }
  }
}
```

解説

- テストを繰り返す関数 test() が定義
- 3 行目では無限ループが定義されている。正しいパラメータが与えられたときに 8 行目で作成されたオブジェクトを戻り値にして関数の実行が終了
- try{} 内にはエラーが発生するかもしれないコードを中に含める。
  - ここでは年、月、日の入力を prompt 用いてダイアログボックスを表示させ、そこに入力させている。
  - 戻り値は文字列なので、Number で数に直している。
- 与えられた入力が正しくなければエラーが投げられ、catch(e) の中に制御が移る。
- catch(e) における e には発生したエラーオブジェクトが渡されるので、コンソールにその情報を出力する (10 行目)。

実行はブラウザで

補足

try{...}catch{} 構文について

- finally{} を付けることもできます。try{...}catch{} 内の部分は try や catch の部分が実行された後必ず呼び出される
- try{...}catch{...} 構文は入れ子にできる。投げられたエラーに一番近い catch にエラーがつかまる。