ソフトウェア開発
第 6 回目授業

平野 照比古

2018/11/9

レポートの形式

問題 1 の解答で実行結果のキャプチャ画面の内容についての解説がない、または見にくいものが多かった。

- 個々の実行結果が何を目的としているかの説明がない。
- キャプチャ画面に重複部分がある。
- これを避けるために、個々の実行結果の脇に手書きで何をしているかを書いてほしい。

クラスの extensible 属性

- class Person を freeze していない解答が目立った。
- クラスのほうを freeze してもインスタンスは freeze されないことを確認してほしい。

Person クラスの prototype

- 関数で定義したクラスとクラスによる定義の違いを認識してほしい。
- 両者の比較をしてある解答はなかった。
- prototype 内にメソッドが存在する
- インスタンス内の__proto__はコンストラクタの prototype への参照

```
>p = new Person("foo",2001,4,1);
{}
>p.__proto__=== Person.prototype;
true
```

WeakMap を利用したクラス (1)

- p.name の型は"string"なので、インスタンスの値がコピーされて戻ってくる。その値を delete しているのでメソッドは消去されない。
- p.birthday の戻り値はオブジェクトなので、その参照が返る。したがって、参照先のデータは書き直せてしまう。

WeakMap を利用したクラス (2)

p.birthday のプロパティを書き直せなくするには次の方法がある。

- コンストラクタ内でオブジェクトを freeze する。
  この方法だとオブジェクトの内容の変更ができない。
- birthday オブジェクトの内容をコピーして返す。

```
  get birthday2(){
    return {
      year:properties.get(this).birthday.year,
      month:properties.get(this).birthday.month,
      day:properties.get(this).birthday.day};
 }
```

for(... in ...) では次のように記述する。

```
  get birthday3(){
    let res = {};
    let obj = properties.get(this).birthday;
    for(let key in obj) {
      res[key] = obj[key];
    }
    return res;
  }
```

オブジェクトが変更になっても限定的ではあるが利用可能

別解

一般に、ある値がまたオブジェクトになっていたらその中もコピーする必要がある。それをしない方法として、一旦、JSON オブジェクトで文字列に変換し、再びオブジェクトに戻すという方法がある。

```
get birthday4(){
  return JSON.parse(JSON.stringify(properties.get(this).birthday))
}
```

クラスの継承におけるエラーチェック

Person クラスで行っている例が目立った。

エラーチェックの厳密化

- 小数のチェック
- 名前が空
- 年が負

などいろいろな工夫があってよい解答がいくつがあった。
エラーチェックはなるべく早い段階で行うほうが手間がかからないので、その手段を本日の授業で解説する。

正規表現とは

- 文字列のパターンを表すオブジェクト
- JavaScript では RegExp クラスが正規表現を表す。
- JavaScript の正規表現は Perl 5 の書式に近いもの

正規表現のオブジェクトの生成

- RegExp() コンストラクタで生成できる
- 通常は特殊なリテラルを使って記述する
- 正規表現リテラルは文字列をスラッシュ(/) で囲んで記述

正規表現のオブジェクトの生成の例

```
let pattern = new RegExp("^s");
let pattern = /^s/;
```

- これらの正規表現はどちらも s で始まる文字列を表す
- これを s で始まる文字列にマッチするという。
- 正規表現内の文字には通常の文字 (ここでは s) を表すものと、特別な意味を持つ文字 (ここでは^) がある。

メタ文字

正規表現では次の文字は特別な意味を持つ。

```
^ $ . * + ? = ! | \ / ( ) [ ] { }
```

- これらの文字をそのまま文字として使いたい場合はその前にバックスラッシュ(\) を付ける。
- 英数字の前にバックスラッシュを付けると別な意味になる場合がある

正規表現のリテラル文字

文字	意味
英数字	通常の文字
\0	NULL 文字 (\u0000)
\t	タブ (\u0009)
\n	改行 (\u000A)
\v	垂直タブ (\u000B)
\f	改ページ (\u000C)
\r	復帰 (\u000D)
\xnn	16 進数 nn で指定された ASCII 文字 (\x0A は \n と同じ)
\uxxxx	16 進数 xxxx で指定された Unicode 文字 (\u000D は\r と同じ)
\cX	制御文字 (\cC は\u0003 と同じ)

文字クラス

文字	意味
[...]	[] 内の任意の 1 文字
[^...]	[] 内以外の任意の 1 文字
.	改行 (Unicode の行末文字) 以外の任意の 1 文字 [^\n] と同じ
\w	任意の単語文字。[A-Za-z0-9] と同じ
\W	任意の単語文字以外の文字。[^A-Za-z0-9] と同じ
\s	任意の Unicode 空白文字
\S	任意の Unicode 空白文字以外の文字
\d	任意の数字。[0-9] と同じ
\D	任意の数字以外の文字。[^0-9] と同じ
[\b]	リテラルバックスペース

文字クラスの例

- \d\d は 2 桁の 10 進数にマッチ
- 先頭に 0 が来てもよい。
- 先頭に 0 が来る場合を除くのであれば [1-9]\d となる。

一般には、同じパターンの繰り返しが必要になることが多い。

繰り返しの指定

文字	意味
{m,n}	直前の項目の m 回から n 回までの繰り返し
{m,}	直前の項目の m 回以上の繰り返し
{m}	直前の項目の m 回の繰り返し
?	直前の項目の 0 回 (なし) か 1 回の繰り返し。{0,1}と同じ
+	直前の項目の 1 回以上の繰り返し。{1,}と同じ
*	直前の項目の 0 回以上の繰り返し。{0,}と同じ

繰り返しの指定の例

- 4 桁の 10 進数のパターンは\d\d\d\d
- 繰り返しを使うと\d{4}
- 1 桁以上の 10 進数は\d+
- 先頭が 0 でないようにすると、[1-9]\d*

貪欲な繰り返し

- 通常、正規表現において繰り返しはできるだけ長く一致するように繰り返しが行われる。
- これを貪欲な繰り返しという。
- "aaaaab"という文字列に対し、正規表現 /\a+/ がマッチする部分は aaaaa の長さ 5 の文字列

非貪欲な繰り返し

- できるだけ短い文字列のマッチで済ませる繰り返しを、非貪欲な繰り返しという
- これを指定するには繰り返し指定の後に?を付ける。
- ??、+?、*?、{1,5}? などのように記述する。
- "aaaaab"という文字列に対し、正規表現 /\a+?/ がマッチする部分は a の長さ 1 の文字列
- 正規表現 /\a+?b/ がマッチする部分は全体の"aaaaab"
- これはマッチが開始する位置が、この文字列の先頭から始まるためである。

選択、グループ化、参照

文字	意味
|	この記号の左右のどちらかを選択する
(...)	正規表現のグループ化をする。これにより、*,+,| などの対象がグループ化されたものになる。また、グループに一致した文字列を記憶して後で参照できる。
(?:...)	グループ化しか行わない。一致した文字列を記憶しない。
\n	グループ番号 n で指定された部分表現に一致する。グループ番号は左から数えた (の数である。ただし、(? は数えない。)

選択、グループ化の例

- (J|j)ava(S|s)cript は JavaScript, Javascript, javaScript, javascript の 4 つにマッチ
- [+-]?\d+ は符号つき (なくてもよい)10 進数とマッチする。
- [+-]?により、符号がなくてもよい

一致位置の指定

文字	意味
^	文字列の先頭
$	文字列の最後
\b	単語境界。\w と\W の間の位置。[\b] との違いに注意
\B	単語境界以外
(?=p)	後に続く文字列が p に一致することが必要。
(?!p)	後に続く文字列が p に一致しないことが必要。

最後の 2 つは Java にはマッチさせたいが JavaScript にはマッチさせたくないときなどに使用できる。

フラグ

文字	意味
i	大文字と小文字を区別しない
g	グローバル検索をする。初めに一致したものだけでなくすべてを検索する。
m	複数行モードにする。^は文字列の先頭だけでなく、行の先頭に,、$は文字列の末尾と行の末尾に一致する。

- g のフラグはパターンマッチした部分文字列を置き換えるメソッド内でしか意味を持たない。
- フラグを書く位置は正規表現リテラルを表す/..../ の後に書く。
- たとえば、/javascript/ig である。これは JaVaSCRipt などにマッチする。

String オブジェクトのメソッド

文字	引数	意味
match()	正規表現	引数の正規表現にマッチした部分を文字列の配列で返す。見つからない場合は null が変える。g フラグがない場合の補足は下の例を参照。
replace()	正規表現、 置換テキスト	g フラグがあれば一致した部分すべてを、ない場合は、はじめのところだけ置換した文字列を返す。 置換文字列の中でグループ化した部分文字列を$1,$2,... で参照できる。
search()	正規表現	正規表現に一致した位置を返す。見つからない場合は -1 を返す。g フラグは無視される。
split()	正規表現、 分割最大数	正規表現のある位置で文字列を分割する。2 番目の引数はオプション

メソッドの実行例

４桁の数字にマッチする正規表現オブジェクトを作成する。

```
let ex = /\d\d\d\d/;
undefined
```

この正規表現に対し、それぞれのメソッドを適用させる。

```
>"20144567".search(ex);
0
>"20144567".match(ex);
["2014"]
>"20144567".replace(ex,"AA");
"AA4567"
```

実行例の解説

- 検索対象の文字列はすべて数字からなるのでこの正規表現にマッチする位置は 0 である。
- マッチした文字列は先頭から 4 文字
- "AA"で置き換えると先頭の 4 文字が置き換えられる。

メソッドの実行例–g フラグ

正規表現に g フラグをつけて同じようなことをする。

```
>let exg = /\d\d\d\d/g;
undefined
>"20144567".search(exg);
0
>"20144567".match(exg);
["2014", "4567"]
>"20144567".replace(exg,"AA");
"AAAA"
```

g フラグがあるので match() や replace() が複数回実行されていることがわかる。

マッチした文字列の利用

```
>"aaa bbb".replace(/(\w*)\s*(\w*)/,"$2,$1");
"bbb,aaa"
```

- 英数字からなる 2 つの文字列 (`(\w*)`) の順序を入れ替えて、その間に , を挿入する。
- `$1` は文字列 `aaa` に、`$2` は文字列 `bbb` にマッチしている。

文字列のあとを参照

文中の Java を JavaScript に変えるものである。
JavaScript を JavaScriptScript にしないために (?!p) を用いる。

```
>"Java と JavaScript は全く違う言語です。".
  replace(/Java(?!Script)/,"JavaScript");
"JavaScript と JavaScript は全く違う言語です。"
```

貪欲さと非貪欲さの確認

```
>"aaaaab".match(/a+/);
["aaaaa"]
>"aaaaab".match(/a+?/);
["a"]
```

- 上の例は貪欲なので a の繰り返しの部分を最大限の位置でマッチ
- 下の例は非貪欲なので最小限の長さの部分にしかマッチしていない

```
>"aaaaab".match(/a+?b/);
["aaaaab"]
```

- ab にマッチしていない
- 初めに先頭の a にマッチしたので b が来るところまでマッチ

前方参照

```
>"abcdbcc".search(/((.)\2).*\1/);
-1
```

- (.) の部分は左かっこが２番目にあるので、\2 で参照できる。したがって、(.)\2 は同じ文字が２つ続いていることを意味する。この部分全体が再び () でくくられているので、その部分は\1 で参照できる。
- したがってこの正規表現は、同じ文字の繰り返しが２回現れる文字列にマッチする。
- 文字列"abcdbcc"には同じ文字が連続して現れるのが末尾の"cc"しかないので、この文字列にはマッチしない (戻り値が -1 である)。

前方参照 (2)

```
>"abccbcc".search(/((.)\2).*\1/);
2
>"abccbcc".match(/((.)\2).*\1/);
["ccbcc", "cc", "c"]
```

- この文字列では cc という同じ文字を繰り返した部分が 2 か所あるのでマッチする。
- はじめの cc の位置が先頭から 2 番目なので戻り値が 2
- match() を行うと、cc ではさまれた部分文字列が戻り値の配列の先頭に、以下、\1 と\2 にマッチした部分文字列が配列に入っている。

前方参照 (3)

```
>"abccbcckkccaaMMaa".match(/((.)\2).*\1/);
["ccbcckkcc", "cc", "c"]
```

- この例では cc が部分文字列に 3 か所現れている。
- マッチした部分は 1 番はじめと 3 番目の cc にはさまれた部分である。これは貪欲なマッチのためである。
- この文字列には aa ではさまれた部分文字列も存在するが、g フラグが付いていないのではじめにマッチしたものしか戻ってこない。
- そのあとの配列には\1 と\2 が入っている。

前方参照 (4)

```
>"abccbcckkccaaMMaa".match(/((.)\2).*\1/g);
["ccbcckkcc", "aaMMaa"]
```

g フラグを付けるとマッチした部分文字列の配列が戻ってくるが、フラグがなかったときのように\1 などの情報は得られない。

```
>"abccbcckkccaaMMaa".match(/((.)\2).*?\1/g);
["ccbcc", "aaMMaa"]
```

この例は非貪欲でグローバルなマッチである。非貪欲にすると一番目と 2 番目の cc にはさまれた部分と aa ではさまれた部分がそれぞれマッチする。

split() における正規表現の利用 (1)

```
>"␣1,␣2␣,␣3␣␣␣,␣␣4".split(/\s*,\s*/);
["␣1", "2", "3", "4"]
```

- 0 個以上の空白、,0 個以上の空白で分割
- 1 の前にある空白が除去できていない

split() における正規表現の利用 (2)

```
>" 1, 2 , 3    ,  4".split(/\W+/);
["", "1", "2", "3", "4"]
```

- 非単語文字の 1 個以上の並びで分割
- 先頭の空白で分割されているので、分割された初めの文字列はから文字列""

split() における正規表現の利用 (3)

```
>" 1, 2 , 3    ,  4".replace(/\s/,"").split(/\W+/);
["1", "2", "3", "4"]
```

先頭の分割文字列が空文字になるのを防ぐために、初めに空白文字を空文字に置き換えて (取り除いて) いる。その文字列に対し非単語文字列で分割しているので空文字が分割結果に表れない。