Wybrane elementy praktyki projektowania oprogramowania
Zestaw 3

Javascript - obiekty, tablice, programowanie funkcyjne

2020-10-27

Liczba punktów do zdobycia: **10/30**
Zestaw ważny do: 2020-11-10

1. (**1p**) Pokazać jak zdefiniować nowy obiekt zawierający co najmniej jedno pole, jedną metodę oraz właściwości z akcesorami `get` i `set`. Pokazać jak do istniejącego obiektu dodać nowe pole, nową metodę i nową właściwość z akcesorami `get` i `set`.

   Uwaga. Do dodawania nowych składowych do istniejących obiektów można użyć metody `Object.defineProperty`. Które z w/w rodzajów składowych (pole, metoda, właściwość) **mogą** być dodawane w ten sposób, a które **muszą** być dodawane w ten sposób (bo inaczej się nie da)?

2. (**1p**) Napisać jeszcze raz rekurencyjną implementację funkcji `fib(n)` ale tym razem użyć pokazanej na wykładzie techniki memoizacji. Porównać czasy obliczeń z poprzednimi implementacjami.

3. (**1p**) Napisać własne implementacje metod `map`, `forEach` i `filter` dla tablic. Zademonstrować przygotowane implementacje w praktyce, przekazując funkcje zwrotne w postaci zwykłej i jako lambda wyrażenia.

```
function forEach( a, f ) {
   // ?
}

function map( a, f ) {
   // ?
}

function filter( a, f ) {
   // ?
}

var a = [1,2,3,4];

forEach( a, _ => { console.log( _ ); } );
// [1,2,3,4]

filter( a, _ => _ < 3 );
// [1,2]

map( a, _ => _ * 2 );
// [2,4,6,8]
```

   Uwaga. Napis `_` jest poprawnym identyfikatorem, a to znaczy że `_ => _ < 3` jest zwykłym lambda wyrażeniem które inaczej można zapisać jako `x => x < 3` albo nawet `function(x) { return x < 3; }`.

4. (**1p**) Poniżej pokazano przykład funkcji, która w swoim dokmnięciu "łapie" zmienną lokalną w sposób niekoniecznie zgodny z oczekiwaniami.

```
function createFs(n) { // tworzy tablicę n funkcji
   var fs = []; // i-ta funkcja z tablicy ma zwrócić i
   for ( var i=0; i<n; i++ ) {
     fs[i] =
        function() {
           return i;
        };
   };
   return fs;
}

var myfs = createFs(10);

console.log( myfs[0]() ); // zerowa funkcja miała zwrócić 0
console.log( myfs[2]() ); // druga miała zwrócić 2
console.log( myfs[7]() );

// 10 10 10
```

   Jednym ze sposobów skorygowania tego nieoczekiwanego zachowania jest zastąpienie `var` przez `let`. Wyjaśnić dlaczego tak jest.

   Istnieje inny sposób, polegający na dodaniu dodatkowego zagnieżdżenia funkcji w funkcji, który dla każdej iteracji pętli `for` utworzy nowy kontekst wiązania domknięcia.

   Zademonstrować ten sposób. Formalnie - tak zmodyfikować powyższy kod żeby pozostawić definicję `var i` przy pętli `for` ale zmienić sposób przypisania funkcji w instrukcji `fs[i] = ....`

   Wskazówka (znacznie ułatwiająca rozwiązanie): rozszerzenie języka o `let` pojawiło się stosunkowo niedawno i nie jest obsługiwane przez starsze przeglądarki. Dlatego w praktyce do zamiany współczesnego dialektu (ES6) na jego starszą wersję (ES5) rozumianą przez przeglądarki używa się technologii Babel, pokazanej na wykładzie. Co zrobi Babel kiedy w powyższym kodzie spróbuje się zastosować pierwszą z metod, czyli zamianę `var` na `let`?

5. (**1p**) Napisać funkcję przyjmującą dowolną liczbę argumentów, która policzy ich sumę.

```
function sum(?) {
   // ?
}

sum(1,2,3);
// 6

sum(1,2,3,4,5);
// 15
```

6. (**1p**) Poniżej pokazano definicję prostego iteratora

```
function createGenerator() {
    var _state = 0;
    return {
        next : function() {
            return {
                value : _state,
                done : _state++ >= 10
            }
        }
    }
}
```

i jej użycie w obiekcie umożliwiające iterowanie jego zawartości za pomocą `for-of`

```
var foo = {
    [Symbol.iterator] : createGenerator
};

for ( var f of foo )
    console.log(f);
```

Pokazać jak sparametryzować definicję tego generatora czyli formalnie - zastąpić stałą `10`, która pojawia się w ciele metody `createGenerator` przez parametr. Zdefiniować kilka różnych obiektów `foo1`, `foo2` z generatorami zainicjowanymi różnymi wartościami argumentów.

7. (**2p**) Zarówno iteratory jak i generatory mogą być "nieskończone", czyli zawsze zwracać kolejną wartość. Zaimplementować takie nieskończone generatory dla liczb fibonacciego: zwykły iterator (zwracający obiekt z funkcją `next`) oraz generator (czyli funkcję wewnętrznie używającą `yield` do zwracania kolejnych wartości).

```
function fib() {
   ...
   return {
      next : function() {
         ...
         return {
            value : ...,
            done  : ...
         }
      }
   }
}

function *fib() {
  ...
  ... yield ...
}
```

W obu przypadkach możliwe jest iterowanie się po kolejnych wartościach za pomocą pokazanej na wykładzie konstrukcji

```
var _it = fib();
for ( var _result; _result = _it.next(), !_result.done; ) {
    console.log( _result.value );
}
```

Czy w którymś z przypadków możliwe jest iterowanie się po kolejnych wartościach za pomocą `for-of`:

```
for ( var i of fib() ) {
    console.log( i );
}
```

8. (**2p**) Próba iterowania nieskończonych iteratorów/generatorów takich jak w poprzednim zadaniu powoduje problem - taki nieskończony iterator/generator zawsze zwraca kolejną wartość i naiwne iterowanie nigdy się nie kończy.

   Pokazać jak rozwiązać ten problem za pomocą dodatkowej funkcji generującej, która jako argumenty przyjmuje iterator/generator oraz liczbę elementów które powinna zwrócić i zwraca dokładnie taką, skończoną liczbę elementów:

```
function* take(it, top) {
   ... yield ...
}

// zwróć dokładnie 10 wartości z potencjalnie
// "nieskończonego" iteratora/generatora
for (let num of take( fib(), 10 ) ) {
  console.log(num);
}
```

Wiktor Zychla