Wybrane elementy praktyki projektowania oprogramowania
Zestaw 4

Javascript - dziedziczenie prototypowe, programowanie asynchroniczne

2022-11-22

Liczba punktów do zdobycia: **10/37**
Zestaw ważny do: 2022-12-06

1. (**2p**) Węzeł drzewa binarnego (`Tree`) to obiekt który przechowuje referencje do swojego lewego i prawego syna oraz wartość (np. liczba lub napis). Proszę w notatkach do wykładu odnaleźć implementację takiego węzła i nauczyć się używać zdefiniowanej tam funkcji konstruktorowej.

   Na wykładzie pokazano jak do prototypu funkcji konstruktorowej dodać generator, który powoduje enumerowanie zawartości drzewa "wgłąb":

```
function Tree(val, left, right) {
  this.left = left;
  this.right = right;
  this.val = val;
}

Tree.prototype[Symbol.iterator] = function*() {
  yield this.val;
  if ( this.left )  yield* this.left;
  if ( this.right ) yield* this.right;
}

var root = new Tree( 1,
  new Tree( 2, new Tree( 3 ) ), new Tree( 4 ));

for ( var e of root ) {
  console.log( e );
}

// 1 2 3 4
```

   Państwa zadaniem jest zaproponować definicję iteratora (zaimplementowanego wprost jako funkcja zwracająca obiekt zawierający `next`... lub jako generator z `yield`), który enumeruje zawartość drzewa "wszerz". Dla podanego wyżej przykładowego drzewa wynikiem enumeracji powinno być `1 4 2 3` i oczywiście w ogólnym przypadku, wyniki enumerowania "wgłąb" i "wszerz" są różne, poza tym że w obu przypadkach pierwszym zwróconym wynikiem jest wartość z korzenia drzewa.

   Wskazówka: iterator nie będzie rekursywny, tak jak ten "wgłąb". Zamiast tego, iterator wykorzystuje pomocniczą strukturę danych - **kolejkę** (jak za pomocą tablicy i funkcji `splice` w Javascript uzyskać efekt kolejki, czyli dodawanie elementów na koniec kolejki, ściąganie elementów z początku kolejki?) - oraz następujący algorytm

   (a) do kolejki włoż korzeń drzewa

(b) powtarzaj dopóki kolejka jest niepusta
   i. wyjmij węzeł z kolejki
   ii. do kolejki włóż lewy i prawy podwęzeł (jeśli istnieją)
   iii. zwróć (`value` lub `yield`) wartość węzła

Pytanie dodatkowe, niepunktowane: co by się stało, gdyby zamiast kolejki użyć stosu (dodawanie elementów na koniec stosu, ściągane elementów z końca stosu)?

2. (**1p**) Na dowolnym przykładzie zademonstrować jak można uzyskać efekt "składowych prywatnych" znany z wielu języków obiektowych.

   Formalnie: zdefiniować funkcję konstruktorową `Foo`, do jej prototypu dodać metodę publiczną `Bar`, którą można zawołać na nowo tworzonych instancjach obiektów, ale w ciele funkcji `Bar` zawołać funkcję `Qux` która jest funkcją prywatną dla instancji tworzonych przez `Foo` (czyli że funkcji `Qux` nie da się zawołać ani wprost na instancjach `Foo` ani w żaden inny sposób niż tylko z wewnątrz metody publicznej `Bar`).

3. (**1p**) Zademonstrować w praktyce tworzenie własnych modułów oraz ich włączanie do kodu za pomocą `require`. Czy możliwa jest sytuacja w której dwa moduły tworzą cykl (odwołują się do siebie nawzajem)? Jeśli nie - wytłumaczyć dlaczego, jeśli tak - pokazać przykład implementacji.

4. (**1p**) Napisać program, który wypisze na ekranie zapytanie o imię użytkownika, odczyta z konsoli wprowadzony tekst, a następnie wypisze `Witaj ***` gdzie puste miejsce zostanie wypełnione wprowadzonym przez użytkownika napisem. Użyć dowolnej techniki do spełnienia tego wymagania, ale nie korzystać z zewnętrznych modułów z `npm` a wyłącznie z obiektów z biblioteki standardowej (wszystkie te techniki sprowadzają się do jakiejś formy dojścia do strumienia `process.stdin`).

   Wskazówka: `https://stackoverflow.com/q/8128578/941240`, jak zwykle w takim przypadku kiedy korzysta się ze SO, proszę przejrzeć odpowiedzi, nie tylko tę najwyżej punktowaną.

5. (**1p**) Napisać program używający modułu (`fs`), który przeczyta w całości plik tekstowy a następnie wypisze jego zawartość na konsoli.

6. (**2p**) Pokazać w jaki sposób odczytywać duże pliki linia po linii za pomocą modułu `readline`. Działanie zademonstrować na przykładowym kodzie analizującym duży plik logów hipotetycznego serwera WWW, w którym każda linia ma postać

```
08:55:36 192.168.0.1 GET /TheApplication/WebResource.axd 200
```

   gdzie poszczególne wartości oznaczają czas, adres klienta, rodzaj żądania HTTP, nazwę zasobu oraz status odpowiedzi.

   W przykładowej aplikacji wydobyć listę adresów IP trzech klientów, którzy skierowali do serwera aplikacji największą liczbę żądań. Przykładowe dane dla aplikacji proszę sobie wygenerować we własnym zakresie (niekoniecznie ściągać logi z rzeczywistego serwera!)

   Wynikiem działania programu powinien być przykładowy raport postaci:

```
12.34.56.78 143
23.45.67.89 113
123.245.167.289 89
```

7. (**2p**) Wybrać jeden z modułów i funkcję asynchroniczną do odczytu danych (`fs::readFile`) i pokazać klasyczny interfejs programowania asynchronicznego, w którym asynchroniczny wynik wywołania funkcji jest dostarczany jako argument do funkcji zwrotnej (callback).

   Następnie pokazać jak taki klasyczny interfejs można zmienić na `Promise` na trzy sposoby:

   - za pomocą "ręcznie" napisanej funkcji przyjmującej te same argumenty co `fs::readFile` ale zwracającej `Promise`
   - za pomocą `util.promisify` z biblioteki standardowej
   - za pomocą `fs.promises` z biblioteki standardowej

   Na zademonstrowanym przykładzie pokazać dwa sposoby obsługi funkcji zwracającej `Promise`

   - "po staremu" - wywołanie z kontynuacją (`Promise::then`)
   - "po nowemu" - wywołanie przez `async/await`

   Wiktor Zychla