Wybrane elementy praktyki projektowania oprogramowania
Zestaw 4

Javascript - dziedziczenie prototypowe, biblioteka standardowa

07-11-2017

Liczba punktów do zdobycia: **10/40**
Zestaw ważny do: 21-11-2017

1. (**1p**) Węzeł drzewa binarnego (`Tree`) to obiekt który przechowuje referencje do swojego lewego i prawego syna oraz wartość (np. liczba lub napis). Pokazać jak zdefiniować funkcję konstruktorową tworzącą taki węzeł (argumentami funkcji powinny być właśnie: lewe i prawe poddrzewo oraz wartość węzła). Pokazać przykład użycia tej funkcji do utworzenia instancji drzewa o głębokości 3 (nie musi być to drzewo pełne).

2. (**1p**) Do prototypu funkcji drzewa binarnego z poprzedniego przykładu dodać iterator (zaimplementowany jako generator przy użyciu `yield`), który pozwoli napisać

```
var root = new Tree( ..... );

// enumeracja wartości z węzłów
for ( var e of root ){
    console.log( e );
}
```

3. (**1p**) Na dowolnym przykładzie zademonstrować jak za pomocą techniki IIFE (*immediately invoked function expression*) 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`).

4. (**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.

5. (**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`).

6. (**1p**) Napisać program używający modułu (`fs`), który przeczyta w całości plik tekstowy a następnie wypisze jego zawartość na konsoli.

7. (**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.

   Wynikiem działania programu powinien być przykładowy raport postaci:

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

8. (**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 dwa sposoby:

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