ZROZUMIEĆ JAVASCRIPT

Wprowadzenie do programowania

Marijn Haverbeke

Helion

no starch press

Tytuł oryginału: Eloquent JavaScript, Second Edition

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-0969-2

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/zrojsc.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/zrojsc
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.
Printed in Poland.

Spis treści

5

Funkcje wyższego rzędu

Duży program to drogi program i to nie tylko z powodu ilości czasu, jaką trzeba poświęcić na jego napisanie. Rozmiar prawie zawsze wiąże się z poziomem złożoności, a ta z kolei mąci w głowie programistom. Natomiast zdezorientowany programista może popełniać błędy. Ponadto w dużym programie jest wiele miejsc do ukrycia błędów.

Wróćmy na chwilę do ostatnich dwóch przykładów z wprowadzenia. Pierwszy jest samodzielny i zawiera sześć wierszy kodu źródłowego.

```
var total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
```

Drugi wykorzystuje dwie funkcje zewnętrzne i zajmuje tylko jedną linijkę.

```
console.log(sum(range(1, 10)));
```

W którym jest większe ryzyko popełnienia błędu?

Jeśli weźmiemy pod uwagę rozmiary definicji funkcji `sum` i `range`, to drugi program też jest duży — nawet większy od pierwszego. Ale i tak jestem gotów się spierać, że w tym programie jest mniejsze ryzyko wystąpienia błędu.

Swoje twierdzenie opieram na tym, że rozwiązanie w tym programie jest przedstawione za pomocą słów, które dość dokładnie opisują problem. Sumowanie przedziałów liczb nie polega na używaniu pętli i liczników, tylko na wykorzystaniu przedziałów liczbowych i sum.

Definicje tych słów (funkcji `sum` i `range`) oczywiście muszą zawierać pętle, liczniki i inne szczegóły implementacyjne. Ale jako że wyrażają prostsze pojęcia niż program jako całość, łatwiej jest napisać je bez błędów.

Abstrakcja

W programowaniu tego rodzaju słowniki nazywa się **abstrakcjami**. Ukrywają one szczegóły i umożliwiają programiście wyrażanie problemów na nieco wyższym (bardziej abstrakcyjnym) poziomie.

Porównaj dwa poniższe przepisy na zupę grochową:

> Wsyp szklankę suszonego grochu na osobę do pojemnika. Dodaj tyle wody, aby przykryć groch. Odstaw pojemnik na przynajmniej 12 godzin. Wyjmij groch z wody i przełóż go do garnka. Wlej cztery szklanki wody na osobę. Przykryj garnek i gotuj na wolnym ogniu dwie godziny. Obierz pół cebuli na osobę. Pokrój cebulę nożem na kawałki i dodaj ją do garnka. Weź po jednym selerze na osobę i pokrój go nożem, a następnie też dodaj do garnka. Weź po jednej marchewce na osobę. Pokrój ją na kawałki. Użyj noża! Wrzuć warzywo do garnka. Gotuj jeszcze dziesięć minut.

I drugi przepis:

> Dla każdej osoby: jedna szklanka suszonego łuskanego grochu, pół pokrojonej cebuli, seler i marchewka.
>
> Namoczyć groch przez 12 godzin, gotować na wolnym ogniu w czterech szklankach wody (na osobę). Pokroić i dodać warzywa. Gotować jeszcze 10 minut.

Drugi przepis jest krótszy i czytelniejszy. Ale trzeba rozumieć parę słów odnoszących się do gotowania — *namoczyć*, *gotować na wolnym ogniu*, *pokroić* i może *warzywo*.

Programista jednak nie może oczekiwać, że wszystkie potrzebne mu słowa będą czekać na niego w słowniku. Przez to może dać się wciągnąć w postępowanie zilustrowane przez pierwszy przykład — dokładny opis wszystkich kroków do wykonania przez komputer bez zwracania uwagi na pojęcia wyższego rzędu, które wyrażają poszczególne słowa.

Programista z doświadczeniem powinien nabyć umiejętność rozpoznawania sytuacji, w których aż się prosi, żeby wyabstrahować jakiś koncept w postaci nowego słowa.

Abstrakcja operacji przeglądania tablicy

Zwykłe funkcje, takie jak oglądane do tej pory, są bardzo dobrym narzędziem abstrakcji. Ale czasami ich możliwości okazują się niewystarczające.

W poprzednim rozdziale kilka razy wystąpiła pętla `for` tego typu:

```
var array = [1, 2, 3];
for (var i = 0; i < array.length; i++) {
  var current = array[i];
  console.log(current);
}
```

Pętlę tę można przeczytać tak: „Dla każdego elementu w tablicy dokonaj zapisu w konsoli". Ale jest to dość zawiły sposób działania polegający na użyciu zmiennej licznikowej `i`, testu porównawczego z długością tablicy oraz dodatkowej deklaracji zmiennej do pobrania bieżącego elementu. Oprócz zwykłej brzydoty kod ten ma jeszcze tę cechę, że łatwo w nim się pomylić. Można przypadkowo po raz kolejny użyć zmiennej `i`, źle napisać nazwę `length` jako `lenght`, pomylić zmienną `i` z `current` itd.

W świetle tych faktów można zdecydować się na napisanie funkcji. Wiesz, jak to zrobić?

W sumie nie ma nic trudnego w napisaniu funkcji przeglądającej tablicę i wywołującej funkcję `console.log` na każdym elemencie tej struktury.

```
function logEach(array) {
  for (var i = 0; i < array.length; i++)
    console.log(array[i]);
}
```

A co, jeśli zechcemy każdy element poddać innej operacji? Jako że „robienie czegoś" reprezentuje się jako funkcję, a funkcje to wartości, możemy przekazać naszą czynność jako wartość funkcyjną.

```
function forEach(array, action) {
  for (var i = 0; i < array.length; i++)
    action(array[i]);
}

forEach(["Wampeter", "Foma", "Granfalloon"], console.log);
// → Wampeter
// → Foma
// → Granfalloon
```

Zamiast przekazywać gotową funkcję do `forEach`, można utworzyć wartość funkcyjną na miejscu.

```
var numbers = [1, 2, 3, 4, 5], sum = 0;
forEach(numbers, function(number) {
  sum += number;
});
console.log(sum);
// → 15
```

To wygląda jak klasyczna pętla `for` z treścią główną wpisaną jako blok pod spodem. Ale teraz treść ta znajduje się w wartości funkcyjnej, jak również w nawiasie wywołania funkcji `forEach`. Dlatego właśnie potrzebne jest zamknięcie klamry *i* nawiasu.

Stosując tę technikę, można określić nazwę zmiennej dla bieżącego elementu (`number`), zamiast wybierać ją ręcznie z tablicy.

Tak naprawdę to nie musimy pisać funkcji `forEach`, ponieważ jest to standardowa metoda tablic. A ponieważ tablica jest już dostarczona jako obiekt, na którym działa metoda, do `forEach` przekazuje się tylko jeden wymagany argument — funkcję do wykonania na każdym elemencie.

Aby pokazać, jak bardzo jest to przydatne, wrócę do funkcji z poprzedniego rozdziału. Zawiera ona dwie pętle przeglądające tablice.

```
function gatherCorrelations(journal) {
  var phis = {};
  for (var entry = 0; entry < journal.length; entry++) {
    var events = journal[entry].events;
    for (var i = 0; i < events.length; i++) {
      var event = events[i];
      if (!(event in phis))
        phis[event] = phi(tableFor(event, journal));
    }
  }
  return phis;
}
```

Przy użyciu funkcji `forEach` można ten kod trochę skrócić i uprościć.

```
function gatherCorrelations(journal) {
  var phis = {};
  journal.forEach(function(entry) {
    entry.events.forEach(function(event) {
      if (!(event in phis))
        phis[event] = phi(tableFor(event, journal));
    });
  });
  return phis;
}
```

Funkcje wyższego rzędu

Funkcje działające na innych funkcjach, pobierające je jako argument lub je zwracające, nazywają się **funkcjami wyższego rzędu**. Jeśli już przyjmujesz do wiadomości fakt, że funkcje to zwykłe wartości, to istnienie takich funkcji nie wyda Ci się ani trochę dziwne. Określenie to pochodzi z matematyki, w której bardziej precyzyjnie odróżnia się funkcje od innych wartości.

Funkcje wyższego rzędu umożliwiają abstrahowanie *czynności*, nie wartości. Występują w paru postaciach. Na przykład można utworzyć funkcję tworzącą nowe funkcje.

```
function greaterThan(n) {
  return function(m) { return m > n; };
}
var greaterThan10 = greaterThan(10);
console.log(greaterThan10(11));
// → true
```

Można też tworzyć funkcje zmieniające inne funkcje.

```
function noisy(f) {
  return function(arg) {
    console.log("wywoływanie z ", arg);
    var val = f(arg);
    console.log("wywołano z ", arg, "- otrzymano", val);
    return val;
  };
}
noisy(Boolean)(0);
// → wywoływanie z 0
// → wywołano z 0 – otrzymano false
```

Można nawet pisać funkcje implementujące nowe typy kontroli przepływu sterowania.

```
function unless(test, then) {
  if (!test) then();
}
function repeat(times, body) {
  for (var i = 0; i < times; i++) body(i);
}

repeat(3, function(n) {
  unless(n % 2, function() {
    console.log(n, "to liczba parzysta");
  });
```

```
});
// → 0 to liczba parzysta
// → 2 to liczba parzysta
```

Opisany w rozdziale 3. zakres leksykalny działa w tym przypadku na naszą korzyść. W poprzednim przykładzie zmienna `n` jest parametrem funkcji zewnętrznej. A ponieważ funkcja wewnętrzna znajduje się w środowisku funkcji zewnętrznej, to również ma dostęp do `n`. Funkcja wewnętrzna ma dostęp do zmiennych z otaczającego ją środowiska. Blok funkcji wewnętrznej odgrywa rolę podobną do bloku {} w zwykłej pętli i instrukcji warunkowej. Ważna różnica polega na tym, że zmienne zadeklarowane w funkcji wewnętrznej są niedostępne w funkcji zewnętrznej. I zazwyczaj jest to bardzo korzystne.

Przekazywanie argumentów

Zdefiniowana wcześniej funkcja `noisy`, zawijająca swój argument w inną funkcję, ma pewien poważny defekt.

```
function noisy(f) {
  return function(arg) {
    console.log("wywoływanie z ", arg);
    var val = f(arg);
    console.log("wywołano z ", arg, "- otrzymano", val);
    return val;
  };
}
```

Jeśli `f` przyjmie więcej niż jeden argument, to użyje tylko pierwszego. Wprawdzie można by było dodać więcej argumentów do funkcji wewnętrznej (`arg1`, `arg2` itd.) i przekazać je do `f`, ale nie wiadomo, ile dokładnie byłoby ich potrzeba. Poza tym takie rozwiązanie pozbawiłoby nas możliwości wykorzystania w `f` informacji z `arguments.length`. Jako że zawsze byłaby przekazywana taka sama liczba argumentów, nie byłoby wiadomo, ile w rzeczywistości zostało ich oryginalnie podanych.

Rozwiązaniem takich problemów w języku JavaScript jest użycie metody `apply`. Przekazuje się jej tablicę (lub obiekt podobny do tablicy) argumentów i wywołuje ona funkcję z tymi argumentami.

```
function transparentWrapping(f) {
  return function() {
    return f.apply(null, arguments);
  };
}
```

Jest to bezużyteczna funkcja, ale ilustruje interesującą nas technikę — zwracana przez nią funkcja przekazuje wszystkie podane argumenty, i tylko te, do `f`. Robi to, przekazując swój własny obiekt `arguments` do metody `apply`. Pierwszy argument metody `apply`, dla którego w tym przypadku przekazujemy `null`, może zostać wykorzystany do symulowania wywołania metody. Wrócę do tego w następnym rozdziale.

JSON

Funkcje wyższego rzędu, które w jakiś sposób stosują funkcję do elementów tablicy, są powszechnie wykorzystywane w języku JavaScript. Jedną z najprostszych z nich jest metoda `forEach`. Istnieje też parę innych podobnych metod tablicowych. Aby się z nimi zaznajomić, poznamy nowy sposób przechowywania danych.

Kilka lat temu pewna osoba przeczesała mnóstwo archiwów i napisała książkę o historii mojego nazwiska (*Haverbeke* znaczy „owsiany potok"). Otwierając ją, liczyłem, że znajdę opowieści o rycerzach, piratach, alchemikach… ale okazało się, że pełno w niej historii flamandzkich wieśniaków. Dla zabawy zapisałem informacje o bezpośrednich przodkach w formacie komputerowym.

Zawartość utworzonego przeze mnie pliku wygląda mniej więcej tak:

```
[
  {"name": "Emma de Milliano", "sex": "f",
   "born": 1876, "died": 1956,
   "father": "Petrus de Milliano",
   "mother": "Sophia van Damme"},
  {"name": "Carolus Haverbeke", "sex": "m",
   "born": 1832, "died": 1905,
   "father": "Carel Haverbeke",
   "mother": "Maria van Brussel"},
  … itd.
]
```

Format ten nazywa się JSON (wym. jak ang. imię Jason), czyli JavaScript Object Notation. Jest on powszechnie wykorzystywany do przechowywania danych i przesyłania ich przez internet.

JSON jest podobny do tego, jak w języku JavaScript zapisywane są tablice i obiekty, tylko ma pewne ograniczenia. Wszystkie nazwy własności muszą znajdować się w podwójnych cudzysłowach i można zapisywać tylko proste dane — żadnych funkcji, zmiennych ani niczego, co wymaga wykonania obliczeń. Także komentarze w formacie JSON są niedozwolone.

W języku JavaScript dostępne są funkcje `JSON.stringify` i `JSON.parse` służące do konwersji danych z tego formatu i na ten format. Pierwsza pobiera wartość JavaScript i zwraca łańcuch w formacie JSON. Druga pobiera taki łańcuch i zamienia go na wartość.

```
var string = JSON.stringify({name: "X", born: 1980});
console.log(string);
// → {"name":"X","born":1980}
console.log(JSON.parse(string).born);
// → 1980
```

Zmienna `ANCESTRY_FILE`, dostępna w środowisku testowym do tego rozdziału i w pliku pobranym pod adresem *ftp://ftp.helion.pl/przyklady/zrojsc.zip*, zawiera treść mojego pliku JSON w postaci łańcucha. Zdekodujemy ją i zobaczymy, ile osób zawiera.

```
var ancestry = JSON.parse(ANCESTRY_FILE);
console.log(ancestry.length);
// → 39
```

Filtrowanie tablicy

Aby znaleźć w zbiorze danych o przodkach osoby, które były młode w 1924 roku, można użyć poniższej funkcji. Odfiltrowuje ona elementy z tablicy, które nie przejdą testu.

```
function filter(array, test) {
  var passed = [];
  for (var i = 0; i < array.length; i++) {
    if (test(array[i]))
      passed.push(array[i]);
  }
  return passed;
}

console.log(filter(ancestry, function(person) {
  return person.born > 1900 && person.born < 1925;
}));
// → [{name: "Philibert Haverbeke", …}, …]
```

W funkcji tej został użyty argument o nazwie `test`. Jest to wartość funkcyjna mająca za zadanie zapełnić „lukę" w funkcjonalności. Funkcja `test` jest wywoływana dla każdego elementu, a jej wartość zwrotna określa, czy element ma zostać dodany do tablicy zwrotnej.

Trzy osoby z pliku żyły i były młode w 1924 roku — moja babcia, mój dziadek i moja cioteczna babka.

Zwróć uwagę, że funkcja `filter` nie usuwa elementów z istniejącej tablicy, tylko tworzy nową tablicę zawierającą tylko te elementy, które przejdą test. Jest to funkcja *czysta*, ponieważ nie modyfikuje przekazanej jej tablicy.

Podobnie jak `forEach` funkcja `filter` też jest standardową metodą tablic. W powyższym przykładzie przedstawiłem jej definicję tylko po to, by pokazać, jak dokładnie działa. Ale od tej pory będę jej używać w następujący sposób:

```
console.log(ancestry.filter(function(person) {
  return person.father == "Carel Haverbeke";
}));
// → [{name: "Carolus Haverbeke", …}]
```

Przekształcanie tablic za pomocą metody map

Powiedzmy, że mamy tablicę obiektów reprezentujących ludzi utworzoną poprzez przefiltrowanie w jakiś sposób tablicy `ancestry`. Wolelibyśmy jednak mieć tablicę nazwisk, ponieważ jest dla nasz czytelniejsza.

Metoda `map` przekształca tablicę, stosując funkcję do wszystkich jej elementów i tworząc nową tablicę ze zwróconych wartości. Ta nowa tablica ma taką samą długość jak poprzednia, ale jej zawartość będzie „zmapowana" na nową formę przez funkcję.

```
function map(array, transform) {
  var mapped = [];
  for (var i = 0; i < array.length; i++)
    mapped.push(transform(array[i]));
  return mapped;
}

var overNinety = ancestry.filter(function(person) {
  return person.died - person.born > 90;
});
console.log(map(overNinety, function(person) {
  return person.name;
}));
// → ["Clara Aernoudts", "Emile Haverbeke", "Maria Haverbeke"]
```

Co ciekawe, osoby, które żyły nie mniej niż 90 lat, to te same, które widzieliśmy wcześniej — były młode w latach 1920. Jest to zarazem najmłodsza generacja w moim zbiorze danych. Podejrzewam, że ma to związek z postępem medycyny.

Podobnie jak `forEach` i `filter`, również `map` jest standardową metodą tablic.

Podsumowywanie przy użyciu metody reduce

Kolejną często wykonywaną czynnością na tablicach jest obliczanie z ich treści pojedynczej wartości. Jako przykład może posłużyć znany nam już przypadek sumowania kolekcji liczb. Innym przykładem może być znajdowanie najstarszej osoby w zbiorze danych.

Operacja wyższego rzędu reprezentująca tę technikę nazywa się **redukcją** (a czasami **zwijaniem**). Można ją sobie wyobrażać jak zwijanie tablicy po jednym elemencie. Przy sumowaniu liczb należy zacząć od zera i każdy kolejny element dodawać do aktualnej sumy.

Parametrami funkcji redukcyjnej `reduce` są, oprócz tablicy, funkcja łącząca i wartość początkowa. Funkcja ta jest nieco bardziej skomplikowana od `filter` i `map`, więc dobrze się jej przyjrzyj.

```
function reduce(array, combine, start) {
  var current = start;
  for (var i = 0; i < array.length; i++)
    current = combine(current, array[i]);
  return current;
}

console.log(reduce([1, 2, 3, 4], function(a, b) {
  return a + b;
}, 0));
// → 10
```

Standardowa tablicowa metoda `reduce`, która oczywiście jest podobna do powyższej, ma jeszcze jedno udogodnienie. Jeżeli tablica zawiera przynajmniej jeden element, można opuścić argument `start`. Wówczas metoda potraktuje pierwszy element tablicy jako wartość startową i rozpocznie redukowanie od drugiego elementu.

Aby za pomocą metody `reduce` znaleźć najstarszego z moich przodków, można napisać taki kod:

```
console.log(ancestry.reduce(function(min, cur) {
  if (cur.born < min.born) return cur;
  else return min;
}));
// → {name: "Pauwels van Haverbeke", born: 1535, …}
```

Składalność

Zastanów się, jak można by było napisać poprzedni przykład (znajdujący najstarszą osobę) bez użycia funkcji wyższego rzędu. Odpowiedni kod nie byłby aż taki zły.

```
var min = ancestry[0];
for (var i = 1; i < ancestry.length; i++) {
  var cur = ancestry[i];
  if (cur.born < min.born)
    min = cur;
}
console.log(min);
// → {name: "Pauwels van Haverbeke", born: 1535, …}
```

Znajduje się w nim parę zmiennych więcej i program ten jest o dwa wiersze dłuższy od poprzedniego, ale też jest zrozumiały.

Zalety funkcji wyższego rzędu stają się ewidentne, gdy trzeba **składać** funkcje. W ramach przykładu napiszemy kod znajdujący średni wiek mężczyzn i kobiet w zbiorze danych.

```
function average(array) {
  function plus(a, b) { return a + b; }
  return array.reduce(plus) / array.length;
}
function age(p) { return p.died - p.born; }
function male(p) { return p.sex == "m"; }
function female(p) { return p.sex == "f"; }

console.log(average(ancestry.filter(male).map(age)));
// → 61.67
console.log(average(ancestry.filter(female).map(age)));
// → 54.56
```

(To trochę dziwne, że trzeba zdefiniować funkcję `plus`, ale w języku JavaScript operatory, w odróżnieniu od funkcji, nie są wartościami, więc nie mogą być przekazywane jako argumenty).

Zamiast wielkiej pętli jest tu logika programu starannie złożona z interesujących nas koncepcji — określanie płci, obliczanie wieku i uśrednianie wartości. Możemy je zastosować jedna po drugiej, aby otrzymać szukany wynik.

Jest to *doskonały* sposób pisania przejrzystego kodu. Niestety nie ma nic za darmo.

Koszty

W szczęśliwej krainie eleganckiego kodu i pięknych tęczy mieszka złośliwy potwór o nazwie **niewydajność**.

Program przetwarzający tablicę w najelegantszej formie składa się z sekwencji osobnych kroków, z których każdy robi coś z tą tablicą i tworzy nową tablicę. Ale tworzenie takiej ilości pośrednich tablic jest kosztowne.

Analogicznie przekazanie funkcji do metody `forEach`, aby wykonała iterację za nas, jest wygodnym i czytelnym rozwiązaniem. Ale wywołania funkcji w JavaScripcie są bardzo kosztowne w porównaniu z wykonywaniem zwykłych pętli.

To samo dotyczy wielu innych technik pozwalających zwiększyć czytelność programu. Abstrakcje dokładają warstwę między to, co robi komputer, a koncepcje, z którymi pracuje programista, i w ten sposób dodają maszynie pracy. Nie jest to oczywiście żelazna zasada — istnieją języki programowania z lepszym wsparciem dla abstrakcji, w których nie wiążą się one z pogorszeniem wydajności, a nawet w JavaScripcie doświadczony programista potrafi znaleźć sposoby na pisanie bardziej abstrakcyjnego, choć nadal szybkiego kodu. Niemniej jednak problem ten istnieje.

Na szczęście większość komputerów jest niesamowicie szybka. Jeśli więc masz do przetworzenia w miarę nieduży zbiór danych albo robisz coś, co musi mieścić się w ludzkiej skali czasu (np. po każdym kliknięciu przycisku), to *nie ma znaczenia*, czy napiszesz piękne rozwiązanie działające w pół milisekundy, czy doskonale zoptymalizowane zajmujące jedną dziesiątą milisekundy.

Dobrym pomysłem jest oszacowanie, jak często ma być wykonywany dany program. Jeśli wstawisz pętlę do pętli (bezpośrednio albo przez wywołanie funkcji wykonującej pętlę), to kod wewnętrznej pętli zostanie wykonany $N \times M$ razy, gdzie N jest liczbą powtórzeń pętli zewnętrznej, a M liczbą powtórzeń pętli wewnętrznej w każdej iteracji pętli zewnętrznej. Jeśli w tej wewnętrznej pętli umieścisz jeszcze kolejną wykonującą P iteracji, to jej instrukcje zostaną wykonane $N \times M \times P$ razy itd. W sumie może wyjść bardzo dużo obliczeń i jeśli program będzie powolny, problem może sprowadzać się do jednego miejsca w pętli wewnętrznej.

Praprapra…

W pliku z danymi znajdują się informacje o moim dziadku Philibercie Haverbeke. Wychodząc od niego, mogę prześledzić swój rodowód, aby dowiedzieć się, czy pochodzę od swojego najstarszego przodka Pauwelsa van Haverbeke w linii prostej. Jeśli tak, to chciałbym wiedzieć, ile teoretycznie mam jego kodu DNA.

Aby móc przechodzić od imienia i nazwiska rodzica do obiektu reprezentującego tę osobę, najpierw zbudujemy obiekt wiążący imiona z ludźmi.

```
var byName = {};
ancestry.forEach(function(person) {
  byName[person.name] = person;
});

console.log(byName["Philibert Haverbeke"]);
// → {name: "Philibert Haverbeke", …}
```

Jednak problem nie jest tak prosty, że wystarczy przejść po własnościach `father` i policzyć, ile stopni się minęło przed dotarciem do Pauwelsa. W rodzinie było parę przypadków małżeństw między dalszymi krewnymi (małe woski i te sprawy). To powoduje, że niektóre gałęzie łączą się z powrotem w paru miejscach, co z kolei oznacza, że mam więcej niż $^1/_2{}^G$ wspólnych genów z daną osobą (G oznacza liczbę pokoleń między Pauwelsem a mną). Wzór ten oznacza, że każde pokolenie dzieli pulę genów na pół.

Rozsądnym podejściem do tego problemu jest potraktowanie go w sposób analogiczny do redukcji, czyli operacji polegającej na sprowadzeniu tablicy do pojedynczej wartości poprzez obliczenie kombinacji jej wartości od lewej do prawej. W tym przypadku też chcemy skondensować strukturę danych do pojedynczej wartości, ale poprzez podążanie śladem więzów rodzinnych. *Kształt* tych danych przypomina drzewo, a nie płaską listę.

Redukcję tej struktury przeprowadzimy w ten sposób, że wartość dla danej osoby obliczymy przez połączenie wartości jej przodków. Można to zrobić rekurencyjnie — jeżeli interesuje nas osoba A, musimy obliczyć wartości dla jej rodziców, co z kolei wymaga obliczenia wartości dla dziadków osoby A itd. Teoretycznie mogłoby to doprowadzić do konieczności wykonania obliczeń dla nieskończonej liczby osób, ale ponieważ nasz zbiór danych jest skończony, musimy znaleźć jakiś sposób na zatrzymanie się. Dlatego dodamy możliwość przekazania do naszej funkcji redukcyjnej wartości domyślnej reprezentującej osoby, których nie ma w danych. W naszym przypadku wartością tą będzie po prostu zero, ponieważ wychodzimy z założenia, że osoby, których nie ma na liście, nie mają wspólnego DNA z interesującym nas przodkiem.

Funkcja `reduceAncestors` pobiera osobę, funkcję łączącą wartości z dwóch rodziców i wartość domyślną i kondensuje wartość z drzewa rodzinnego.

```
function reduceAncestors(person, f, defaultValue) {
  function valueFor(person) {
    if (person == null)
      return defaultValue;
    else
      return f(person, valueFor(byName[person.mother]),
                       valueFor(byName[person.father]));
  }
  return valueFor(person);
}
```

Funkcja wewnętrzna `valueFor` obsługuje pojedynczą osobę. Dzięki magii rekurencji może wywołać samą siebie, aby obsłużyć ojca i matkę tej osoby. Wyniki, wraz z samym obiektem osoby, są przekazywane do funkcji `f`, która zwraca rzeczywistą wartość dla tej osoby.

Później przy użyciu tej wartości można obliczyć, ile wspólnego DNA miała moja babcia z Pauwelsem van Haverbeke, i podzielić to przez cztery.

```
function sharedDNA(person, fromMother, fromFather) {
  if (person.name == "Pauwels van Haverbeke")
    return 1;
  else
    return (fromMother + fromFather) / 2;
}
var ph = byName["Philibert Haverbeke"];
console.log(reduceAncestors(ph, sharedDNA, 0) / 4);
// → 0.00049
```

Osoba nazywająca się Pauwels van Haverbeke oczywiście ma 100 procent wspólnych genów z Pauwelsem van Haverbeke (w zbiorze danych nie ma osób, które nazywały się dokładnie tak samo), więc dla niego funkcja zwraca 1. Wszyscy pozostali ludzie mają z nim wspólną średnią wartość z wartości dzielonych przez ich rodziców.

Statystycznie rzecz ujmując, mam około 0,05% wspólnego DNA z przodkiem, który żył w XVI wieku. Oczywiście to tylko statystyczne przybliżenie, a nie dokładna liczba. Nie jest to wiele, ale biorąc pod uwagę ilość materiału genetycznego w człowieku (około 3 miliardów par zasad), możliwe, że w moim organizmie wciąż istnieje jakaś cząstka pochodząca od Pauwelsa.

Wartość tę można by było też obliczyć bez posługiwania się funkcją `reduceAncestors`. Ale oddzielenie ogólnego podejścia (kondensacja drzewa rodzinnego) od specyficznego przypadku (obliczenie ilości wspólnego DNA) może poprawić klarowność kodu i pozwala na wielokrotne wykorzystanie abstrakcyjnej części programu w innych przypadkach. Na przykład poniższy kod znajduje odsetek znanych przodków wybranej osoby, którzy żyli dłużej niż 70 lat.

```
function countAncestors(person, test) {
  function combine(person, fromMother, fromFather) {
    var thisOneCounts = test(person);
    return fromMother + fromFather + (thisOneCounts ? 1 : 0);
  }
  return reduceAncestors(person, combine, 0);
}
function longLivingPercentage(person) {
  var all = countAncestors(person, function(person) {
    return true;
  });
  var longLiving = countAncestors(person, function(person) {
```

```
    return (person.died - person.born) >= 70;
  });
  return longLiving / all;
}
console.log(longLivingPercentage(byName["Emile Haverbeke"]));
// → 0.145
```

Zważywszy na to, że nasz zbiór danych zawiera dość przypadkową kolekcję osób, liczb takich nie należy traktować zbyt poważnie. Ale przedstawiony kod ilustruje fakt, że funkcja `reduceAncestors` stanowi dla nas przydatne słowo w pracy z drzewami rodzinnymi.

Wiązanie

Metoda `bind`, którą mają wszystkie funkcje, tworzy nową funkcję wywołującą oryginalną funkcję, ale z ustawionymi niektórymi argumentami.

Poniżej znajduje się przykład użycia metody `bind`. Jest to definicja funkcji o nazwie `isInSet`, sprawdzającej, czy dana osoba znajduje się w podanym zbiorze łańcuchów. Aby wywołać metodę `filter` w celu zebrania obiektów tych osób, których imienia i nazwiska znajdują się w określonym zbiorze, możemy napisać wyrażenie funkcyjne wywołujące funkcję `isInSet` z naszym zbiorem jako pierwszym argumentem albo *częściowo zastosować* funkcję `isInSet`.

```
var theSet = ["Carel Haverbeke", "Maria van Brussel", "Donald Duck"];
function isInSet(set, person) {
  return set.indexOf(person.name) > -1;
}
console.log(ancestry.filter(function(person) {
  return isInSet(theSet, person);
}));
// → [{name: "Maria van Brussel", …},
//    {name: "Carel Haverbeke", …}]
console.log(ancestry.filter(isInSet.bind(null, theSet)));
// → … taki sam wynik
```

Wywołanie funkcji `bind` zwróci funkcję, która wywoła funkcję `isInSet` ze zbiorem `theSet` jako pierwszym argumentem i pozostałymi argumentami przekazanymi do powiązanej funkcji.

Pierwszy argument, w którego miejscu w przykładzie przekazano wartość `null`, jest używany dla wywołań metody, podobnie jak pierwszy argument funkcji `apply`. Bardziej szczegółowo opisuję to w następnym rozdziale.

Podsumowanie

Możliwość przekazywania funkcji jako wartości do innych funkcji to nie tylko sztuczka, ale bardzo przydatna właściwość języka JavaScript. Pozwala na pisanie obliczeń jako funkcji z „lukami", które kod wywołujący musi wypełnić, dostarczając wartości funkcyjne implementujące braki.

Tablice udostępniają parę metod wyższego rzędu — `forEach` do wykonywania działań na każdym elemencie tablicy po kolei, `filter` do tworzenia nowych tablic z odfiltrowaną częścią elementów, `map` do tworzenia nowych tablic, których każdy element został przepuszczony przez jakąś funkcję, oraz `reduce` do łączenia wszystkich elementów tablicy w jedną wartość.

Funkcje mają metodę `apply`, przy użyciu której można je wywoływać z tablicą argumentów. Mają też metodę `bind` służącą do tworzenia częściowo zastosowanych wersji funkcji.

Ćwiczenia

Spłaszczanie

Za pomocą metod `reduce` i `concat` „spłaszcz" tablicę tablic do postaci pojedynczej tablicy zawierającej wszystkie elementy tablic wejściowych.

Różnica wieku między matką i dzieckiem

Przy użyciu przykładowego zbioru danych z tego rozdziału oblicz średnią różnicę wieku między matkami i dziećmi (wiek matki w chwili urodzenia dziecka). Możesz użyć zdefiniowanej w tym rozdziale funkcji `average`.

Zwróć uwagę, że nie wszystkie matki znajdujące się w danych są obecne w tablicy. W sytuacji tej może być przydatny obiekt `byName`, który ułatwia znajdowanie obiektu osoby po jej imieniu i nazwisku.

Historyczna średnia długość życia

Gdy poszukaliśmy w naszym zbiorze danych osób, które żyły dłużej niż 90 lat, znaleźliśmy tylko osoby z ostatniego pokolenia. Przyjrzyjmy się bliżej temu zjawisku.

Oblicz i wyświetl średni wiek osób ze zbioru danych dla każdego stulecia. Wiek, w którym żyła dana osoba, można obliczyć, dzieląc datę jej śmierci przez 100 i zaokrąglając otrzymany wynik w górę, np. `Math.ceil(person.died / 100)`.

Zadanie dla chętnych: napisz funkcję o nazwie `groupBy` abstrahującą operację grupowania. Funkcja ta powinna przyjmować jako argumenty tablicę i funkcję obliczającą grupę dla elementu w tej tablicy oraz zwracającą obiekt odwzorowujący nazwy grup na tablice numerów grup.

Wszystko i trochę

Tablice mają też standardowe metody o nazwach `every` i `some`. Obie one przyjmują funkcję predykatywną, która dla argumentu w postaci elementu tablicy zwraca prawdę albo fałsz. Podobnie jak operator `&&` zwraca prawdę tylko wtedy, gdy wyrażenia po jego obu stronach są prawdziwe, tak funkcja `every` zwraca prawdę tylko wtedy, gdy predykat zwraca prawdę dla *wszystkich* elementów tablicy. Natomiast funkcja `some` zwraca prawdę, gdy predykat zwróci prawdę dla *któregokolwiek* z elementów. Funkcje te nie przetwarzają więcej argumentów, niż jest konieczne — na przykład jeśli funkcja `some` znajdzie predykat zwracający prawdę dla pierwszego elementu tablicy, to nie sprawdza już pozostałych wartości.

Napisz dwie funkcje, `every` i `some`, działające jak te metody, tylko niebędące metodami, a pobierające tablicę jako swój pierwszy argument.

Problem z językami obiektowymi polega na tym, że wszystkie działają
w niejawnie powiązanych z nimi środowiskach, których nie da się pominąć.
Chcesz banana, ale w efekcie otrzymujesz goryla trzymającego banana
i całą dżunglę.

— Joe Armstrong, w wywiadzie dla *Sztuki kodowania*

Skorowidz

A

B

C

Ć

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

X

Z

Ż

POZNAJ MOŻLIWOŚCI JĘZYKA JAVASCRIPT!

Współczesne aplikacje i strony internetowe nie byłyby takie, jakie są obecnie, gdyby nie JavaScript. Ten język programowania jeszcze kilka lat temu przeżywał kryzys, lecz w końcu został doceniony. Obecnie jest stosowany wszędzie tam, gdzie użytkownicy wymagają najwyższej interaktywności, szybkości działania oraz wygody korzystania z aplikacji internetowej. Jeżeli chcesz poznać JavaScript i użyć go już w najbliższym projekcie, to trafiłeś na doskonałą książkę.

Otwórz ją i przekonaj się, jak wygląda składnia JavaScriptu oraz typowe konstrukcje w tym języku. W trakcie lektury kolejnych rozdziałów nauczysz się budować przejrzystą strukturę programu, korzystać z obiektów i tablic oraz wyrażeń regularnych. Ponadto poznasz tajniki programowania obiektowego i najlepsze techniki obsługi błędów. Gdy opanujesz już podstawy związane z językiem, przyjdzie czas na drugą część książki, poświęconą możliwościom JavaScriptu w środowisku przeglądarki. Na kolejnych stronach znajdziesz informacje o modelu DOM, korzystaniu z elementu canvas oraz obsłudze formularzy. Na koniec poznasz tajniki optymalizacji kodu, żeby w jeszcze większym stopniu wykorzystać potencjał JavaScriptu. Książka ta jest doskonałą lekturą dla czytelników, którzy chcą bezproblemowo wkroczyć w świat tego języka!

Dzięki tej książce:

- poznasz składnię języka JavaScript
- wykorzystasz jego konstrukcje
- poznasz techniki programowania obiektowego
- zmodyfikujesz model DOM strony WWW
- w pełni wykorzystasz potencjał JavaScriptu

Marijn Haverbeke — niezależny programista i pisarz. W centrum jego zainteresowań znajdują się języki programowania oraz narzędzia dla programistów. Jest autorem edytora CodeMirror, a także narzędzi Tern oraz Acorn.

Helion
33823 numer katalogowy
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900
Informatyka w najlepszym wydaniu

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

no starch press

ISBN 978-83-283-0969-2
9 788328 309692
cena: 79,00 zł