ZROZUMIEĆ JAVASCRIPT

WYDANIE III

Wprowadzenie
do programowania

Marijn Haverbeke

Helion

no starch press

Tytuł oryginału: Eloquent JavaScript, 3rd Edition: A Modern Introduction to Programming

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-6362-5

Helion SA
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/zrojs3.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/zrojs3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

11

Programowanie asynchroniczne

Sercem komputera, częścią, która wykonuje poszczególne kroki wszystkich operacji, jest **procesor**. Programy, które pokazywałem do tej pory, dają procesorowi pracę do wykonania. Prędkość, z jaką zostanie wykonana na przykład pętla, w bardzo dużym stopniu zależy właśnie od prędkości procesora.

Jednak wiele części programów korzysta także z innych zasobów niż procesor. Mogą na przykład komunikować się przez sieć lub pobierać dane z dysku twardego, który jest znacznie wolniejszy niż pamięć główna.

Gdy wykonywane są takie operacje, szkoda by było marnować czas procesora na bezczynne oczekiwanie, bo przecież mógłby robić wiele innych rzeczy. Częściowo zajmuje się tym system operacyjny, który przydziela procesor różnym działającym akurat programom. To jednak nie pomaga, gdy chcemy, aby **jeden** program wykonywał postęp, oczekując na odpowiedź na żądanie sieciowe.

Asynchroniczność

W modelu **programowania synchronicznego** działania są wykonywane jedno po drugim. Kiedy wywołamy czasochłonną funkcję, to musimy czekać, aż skończy działanie i zwróci wynik. W międzyczasie reszta programu stoi w miejscu.

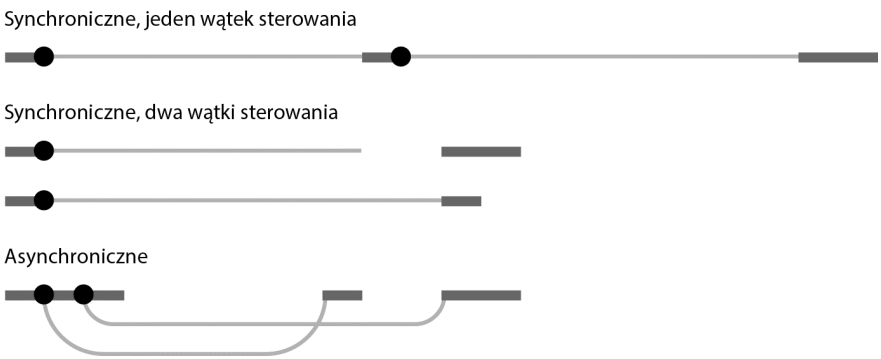
Model **asynchroniczny** umożliwia wykonywanie wielu działań jednocześnie. Kiedy zaczynamy wykonywać jakieś działanie, reszta programu nadal działa. Gdy czynność ta zostanie zakończona, program zostaje o tym poinformowany i uzyskuje dostęp do wyniku (może na przykład pobrać dane z dysku).

Aby porównać programowanie synchroniczne i asynchroniczne, możemy posłużyć się przykładem prostego programu pobierającego dwa zasoby z sieci i łączącego wyniki.

W środowisku synchronicznym, w którym funkcja żądająca zwraca wynik dopiero po zakończeniu pracy, najprostszym rozwiązaniem jest wysyłanie żądań jedno po drugim. Jego wadą jest to, że drugie żądanie rozpocznie się dopiero po zakończeniu pierwszego. Całkowity czas wykonywania wyniesie nie mniej niż suma czasów wykonywania obu funkcji.

Z kolei w systemie asynchronicznym ten problem można rozwiązać przez utworzenie dodatkowego wątku sterowania. **Wątek** to dodatkowy program, którego wykonywanie system operacyjny może przeplatać z wykonywaniem innych programów — a dzięki temu, że nowoczesne komputery zawierają wiele procesorów, wątki mogą być wykonywane nawet jednocześnie, każdy przez inny procesor. Drugi wątek mógłby rozpocząć drugie żądanie i wówczas oba wątki czekają na pojawienie się wyników, po czym ponownie się synchronizują, aby je połączyć.

Na poniższym schemacie grube linie reprezentują czas potrzebny na normalne wykonanie programu, a cienkie — czas oczekiwania na sieć. W modelu synchronicznym czas oczekiwania na sieć *wlicza* się w oś czasu wykonywania określonego wątku. W modelu asynchronicznym rozpoczęcie operacji związanej z wykorzystaniem sieci powoduje *rozdzielenie* osi czasu. Program, który zainicjował tę akcję, nadal działa obok dodatkowej operacji i zostaje powiadomiony o jej ukończeniu.



Patrząc z innej strony, oczekiwanie na zakończenie operacji jest **wewnętrzną** cechą modelu synchronicznego, ale **zewnętrzną**, czyli znajdującą się pod naszą kontrolą, cechą modelu asynchronicznego.

Asynchroniczność to miecz obosieczny. Z jednej strony ułatwia pisanie programów, które nie pasują do prostolinijnego modelu wykonywania, a z drugiej strony sprawia, że programy typowe dla tego liniowego modelu są bardziej niezgrabne. W dalszej części rozdziału pokazuję, jak to rozwiązać.

Obie najważniejsze platformy wykonawcze języka JavaScript — przeglądarki i Node.js — wykonują czasochłonne operacje asynchronicznie, zamiast wykorzystywać wątki. Choć programowanie z użyciem wątków jest bardzo trudne (gdy program wykonuje wiele operacji naraz, jeszcze trudniej jest go zrozumieć), ogólnie uważa się je za bardzo dobre rozwiązanie.

Wronia technologia

Większość ludzi wie, że wrony to bardzo mądre ptaki. Posługują się narzędziami, umieją planować, zapamiętują fakty, a nawet potrafią przekazywać sobie informacje.

Mało kto jednak wie, że te ptaki potrafią o wiele więcej, tylko się z tym przed nami kryją. Pewien ceniony (choć odrobinę ekscentryczny) ekspert od krukowatych powiedział mi, że ich technologia wcale nie jest tak bardzo w tyle za ludzką i na dodatek nas gonią.

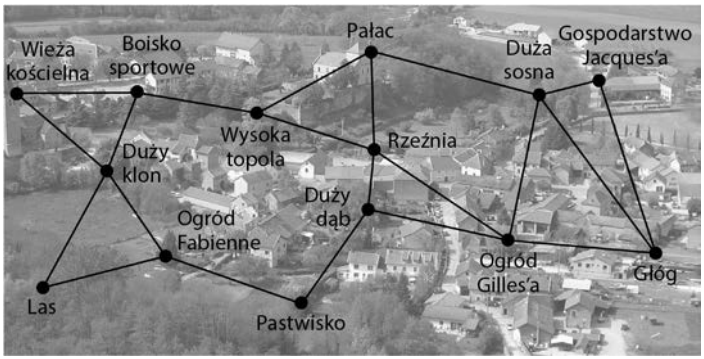
Na przykład wiele wronich kultur potrafi konstruować maszyny liczące. Nie są to urządzenia elektroniczne podobne do naszych komputerów, tylko maszyny wykorzystujące małe owady blisko spokrewnione z termitami, które wykształciły symbiotyczne relacje z wronami. Ptaki dostarczają pożywienia, a owady w zamian tworzą i prowadzą skomplikowane kolonie, które przy pomocy żyjących w nich stworzeń wykonują obliczenia.

Kolonie takie zazwyczaj znajdują się w dużych, długo istniejących gniazdach. Ptaki wspólnie z owadami tworzą sieć kulistych struktur z gliny ukrytych między częściami gniazda, w których żyją i pracują owady.

Z innymi urządzeniami maszyny te komunikują się przy użyciu sygnałów świetlnych. Wrony umieszczają kawałki odblaskowego materiału w specjalnych słupkach komunikacyjnych, a owady kierują nimi w taki sposób, aby odbijały światło na inne gniazdo, kodując dane w postaci sekwencji szybkich błysków. To oznacza, że komunikacja jest możliwa tylko między gniazdami mającymi niezaburzony kontakt wizualny.

Mój znajomy ekspert od krukowatych sporządził mapę sieci wronich gniazd we wsi Hières-sur-Amby nad Rodanem. Na następnej stronie pokazana jest ta mapa z zaznaczonymi gniazdami i połączeniami między nimi.

W zdumiewającym efekcie zbieżnej ewolucji wronie komputery zaczęły wykonywać programy w języku JavaScript. W tym rozdziale napiszemy dla nich kilka podstawowych funkcji sieciowych.



Wywołania zwrotne

Jednym z podejść do programowania asynchronicznego jest dodanie do funkcji wykonujących czasochłonne operacje dodatkowego argumentu — **funkcji zwrotnej**. Operacja zostaje rozpoczęta i gdy dobiegnie końca, zostaje wywołana funkcja zwrotna z wynikiem.

Na przykład funkcja `setTimeout`, dostępna zarówno w Node.js, jak i przeglądarkach, czeka określoną liczbę milisekund (sekunda to tysiąc milisekund), po czym wywołuje określoną funkcję.

```
setTimeout(() => console.log("Tick"), 500);
```

Oczekiwanie to nie jest przesadnie ważna czynność, ale może się przydać na przykład do aktualizacji animacji albo sprawdzania, czy wykonywanie jakiejś czynności trwa dłużej niż określoną ilość czasu.

Wykonywanie kilku asynchronicznych operacji po kolei oznacza konieczność przekazywania nowych funkcji do realizacji obliczeń.

Większość wronich komputerów w gniazdach ma pamięć długoterminową, w której dane są przechowywane w gałązkach i mogą być pobierane później. Żłobienie, lub wyszukiwanie danych, trochę trwa, a więc interfejs do pamięci długoterminowej jest asynchroniczny i wykorzystuje funkcje zwrotne.

Kulki pamięciowe przechowują dane możliwe do zapisania w formacie JSON pod nazwami. Wrona może zapisać informacje o miejscach, gdzie ukryła jedzenie, pod nazwą „schowki na jedzenie". Może to być tablica nazw wskazujących inne dane, opisujących rzeczywisty schowek. Aby znaleźć schowek na jedzenie w kulkach pamięciowych gniazda na dużym dębie, wrona może wykonać następujący kod:

```
import {bigOak} from "./crow-tech";
bigOak.readStorage("schowki na jedzenie", caches => {
  let firstCache = caches[0];
```

```
  bigOak.readStorage(firstCache, info => {
    onsole.log(info);
  });
});
```

(Nazwy wszystkich wiązań i łańcuchy zostały przetłumaczone z wroniego na polski).

Ten styl programowania jest znośny, ale każda kolejna asynchroniczna operacja zwiększa głębokość wcięcia, ponieważ potrzebna jest kolejna funkcja. Przy bardziej skomplikowanych zadaniach wymagających wykonania kilku czynności może zacząć robić się niezręcznie.

Wronie komputery do komunikacji wykorzystują pary żądanie-odpowiedź. To znaczy, że jedno gniazdo wysyła wiadomość do innego gniazda, a ono natychmiast wysyła potwierdzenie odbioru i ewentualnie odpowiedź na zadane pytanie.

Każda wiadomość ma określony typ, który decyduje o sposobie jej obsługi. Możemy zdefiniować procedury obsługi żądań różnego typu i wywoływać je, aby udzielić odpowiedzi na określone żądania.

Interfejs eksportowany przez moduł `"./crow-tech"` udostępnia funkcje zwrotne służące do komunikacji. Gniazda mają metodę `send`, która wysyła żądanie. W trzech pierwszych argumentach należy jej przekazać nazwę gniazda docelowego, typ żądania oraz jego treść. W czwartym i ostatnim argumencie należy podać funkcję, którą ma wywołać, gdy nadejdzie odpowiedź.

```
bigOak.send("Pastwisko", "note", "Kraczmy głośno o 19:00.",
() => console.log("Notka dostarczona."));
```

Aby gniazda mogły odbierać żądania, najpierw musimy zdefiniować typ żądania o nazwie `"note"`. Kod obsługujący te żądania musi działać nie tylko w jednym komputerze-gnieździe, ale we wszystkich gniazdach, które mogą odbierać wiadomości tego typu. Załóżmy zatem, że jedna z wron przeleci się po gniazdach i we wszystkich zainstaluje naszą procedurę obsługi.

```
import {defineRequestType} from "./crow-tech";
defineRequestType("note", (nest, content, source, done) => {
  console.log(`${nest.name} received note: ${content}`);
  done();
});
```

Funkcja `defineRequestType` definiuje nowy typ żądania. Przedstawiony przykład dodaje obsługę typu żądań `"note"`, które umożliwiają wysłanie notki do wybranego gniazda. Nasza implementacja wywołuje `console.log`, abyśmy mieli możliwość sprawdzić, czy żądanie zostało odebrane. Gniazda mają własność `name` zawierającą ich nazwę.

Czwarty argument procedury obsługi, `done`, jest funkcją zwrotną, która ma zostać wywołana przez tę procedurę po zakończeniu realizacji żądania. Gdybyśmy użyli wartości zwrotnej procedury jako wartości odpowiedzi, to oznaczałoby to, że procedura ta nie jest w stanie wykonywać operacji asynchronicznych. Funkcja wykonująca pracę asynchroniczną zwraca sterowanie, zanim praca zostanie wykonana, pozostawiając w planach wywołanie funkcji zwrotnej. Potrzebujemy więc mechanizmu asynchronicznego — w tym przypadku kolejnej funkcji zwrotnej — do sygnalizowania dostępności odpowiedzi.

Asynchroniczność jest w pewnym sensie zaraźliwa. Każda funkcja wywołująca funkcję działającą asynchronicznie sama musi być asynchroniczna, czyli musi zwracać wynik przy użyciu funkcji zwrotnej lub innego podobnego mechanizmu. Wywołanie funkcji zwrotnej jest odrobinę bardziej skomplikowane i zagrożone błędem niż zwykłe zwrócenie wartości, dlatego nie jest dobrze, gdy w taki sposób trzeba skonstruować dużą część programu.

Obietnice

Praca z abstrakcyjnymi koncepcjami jest łatwiejsza, gdy mogą być reprezentowane przez wartości. W przypadku operacji asynchronicznych można zwrócić obiekt reprezentujący to przyszłe zdarzenie, zamiast planować wywołanie funkcji w pewnym momencie w przyszłości.

Do tego właśnie służy standardowa klasa `Promise`. **Obietnica** (ang. *promise*) to asynchroniczna operacja, która może zakończyć się w pewnym momencie, wytwarzając pewną wartość. Może powiadomić wszystkie zainteresowane strony o udostępnieniu tej wartości.

Najprostszym sposobem na utworzenie obietnicy jest wywołanie funkcji `Promise.resolve`. Opakowuje ona przekazaną jej wartość w obietnicę. Jeśli to już jest obietnica, to tylko ją zwraca — w przeciwnym razie otrzymujemy nową obietnicę, która natychmiast kończy działanie, zwracając naszą wartość w wyniku.

```
let fifteen = Promise.resolve(15);
fifteen.then(value => console.log(`Otrzymałam ${value}`));
// → Otrzymałam 15
```

Wynik obietnicy można pobrać za pomocą metody `then`. Rejestruje ona funkcję zwrotną, która zostanie wywołana, gdy obietnica zostanie rozwiązana i wytworzy wartość. Do jednej obietnicy można dodać kilka funkcji zwrotnych i zostaną one wywołane, nawet jeśli dodamy je już po rozwiązaniu (zakończeniu) obietnicy.

To jednak nie wszystko, co robi metoda `then`. Zwraca ona kolejną obietnicę, która oddaje wartość zwróconą przez procedurę obsługi, a jeśli ta zwróci obietnicę, czeka na tę obietnicę, po czym oddaje jej wartość.

Obietnice można sobie wyobrazić jako narzędzie do wprowadzania wartości do rzeczywistości asynchronicznej. Normalna własność po prostu jest. Obiecana

wartość to taka, która może już być lub może pojawić się w przyszłości. Operacje wykorzystujące obietnice operują na takich opakowanych wartościach i są wykonywane asynchronicznie, gdy pojawią się potrzebne im wartości.

Aby utworzyć obietnicę, można użyć konstruktora `Promise`. Ma on dość niezwykły interfejs, ponieważ jako argument przyjmuje funkcję, którą natychmiast wywołuje, przekazując jej funkcję, której może użyć do rozwiązania obietnicy. Działa właśnie w taki sposób, a nie na przykład przez wykorzystanie metody `resolve`, aby obietnicę mógł rozwiązać tylko ten kod, który ją utworzył.

Poniżej znajduje się przykład utworzenia interfejsu wykorzystującego obietnicę dla funkcji `readStorage`:

```
function storage(nest, name) {
  return new Promise(resolve => {
    nest.readStorage(name, result => resolve(result));
  });
}
storage(bigOak, "enemies")
.then(value => console.log("Otrzymałam", value));
```

Ta funkcja asynchroniczna zwraca konkretną wartość. Taka jest podstawowa zaleta obietnic — ułatwiają posługiwanie się funkcjami asynchronicznymi. Funkcje wykorzystujące obietnice nie muszą przekazywać funkcji zwrotnych, dzięki czemu wyglądają jak zwykłe funkcje — pobierają dane w argumentach i zwracają ich wyniki. Jedyna różnica polega na tym, że wynik może być jeszcze niedostępny.

Błędy

Zwykłe operacje JavaScript mogą zaznaczyć błąd wykonywania przez zgłoszenie wyjątku. Operacje asynchroniczne często muszą korzystać z tego typu mechanizmów, gdy na przykład nie uda się nawiązać połączenia sieciowego albo któraś część kodu asynchronicznego zgłosi wyjątek.

Jednym z największych problemów związanych z programowaniem asynchronicznym przy użyciu funkcji zwrotnych jest trudność dopilnowania, aby błędy były prawidłowo zgłaszane do funkcji zwrotnych.

Zgodnie z powszechnie przyjętą konwencją pierwszy argument funkcji zwrotnej określa operację, która się nie powiodła, a drugi zawiera wartość zwróconą przez tę operację, jeśli zakończy się bez błędu. Takie funkcje zwrotne zawsze muszą sprawdzać, czy otrzymały wyjątek, oraz dopilnować, by wszelkie problemy, włącznie z wyjątkami zgłaszanymi przez funkcje, które wywołują, były wychwytywane i przekazywane do odpowiedniej funkcji.

Obietnice to ułatwiają. Mogą być rozwiązywane (operacja zakończona powodzeniem) lub odrzucane (niepowodzenie). Procedury obsługi rozwiązania (rejestrowane za pomocą słowa kluczowego `then`) są wywoływane tylko wtedy,

gdy operacja zostanie pomyślnie wykonana, z kolei odrzucenia są automatycznie propagowane do nowej obietnicy zwróconej przez `then`. A kiedy procedura obsługi zgłosi wyjątek, powoduje automatyczne odrzucenie obietnicy utworzonej przez wywołanie `then`. Jeśli więc którekolwiek ogniwo asynchronicznego łańcucha operacji zawiedzie, wynik całego łańcucha zostaje odrzucony i nie zostaje wywołana żadna procedura obsługi powodzenia znajdująca się za miejscem wystąpienia błędu.

Wartość jest zwracana zarówno po pomyślnym rozwiązaniu, jak i odrzuceniu obietnicy. W tym drugim przypadku nazywa się ona **powodem** odrzucenia. Gdy odrzucenie jest spowodowane wyjątkiem w funkcji obsługowej, jako powód zwracana jest wartość tego wyjątku. Analogicznie, kiedy procedura obsługi zwraca obietnicę, która zostaje odrzucona, to odrzucenie przepływa do następnej obietnicy. Istnieje funkcja `Promise.reject`, która tworzy nową, natychmiast odrzucaną obietnicę.

Do bezpośredniej obsługi takich odrzuceń służy metoda `catch` obietnic, która rejestruje procedurę obsługi wywoływaną w momencie odrzucenia obietnicy, podobnie jak procedury `then` są wywoływane w celu obsługi normalnego rozwiązania. Ponadto, podobnie jak `then`, zwraca nową obietnicę, która zamienia się w wartość oryginalnej obietnicy w przypadku normalnego rozwiązania i w wynik procedury `catch` w przeciwnym razie. Jeśli procedura `catch` zgłosi błąd, nowa obietnica także zostaje odrzucona.

Metoda `then` także przyjmuje procedurę obsługi odrzucenia w drugim argumencie, dzięki czemu można zainstalować dwa rodzaje procedur obsługi w jednym wywołaniu metody.

Funkcja przekazywana do konstruktora `Promise` wraz z funkcją rozwiązania otrzymuje drugi argument, którego może użyć do odrzucenia nowej obietnicy.

Łańcuchy wartości obietnic tworzone przez wywołania metod `then` i `catch` można sobie wyobrażać jako rurociąg, przez który płyną asynchroniczne wartości lub informacje o błędach. Jako że takie łańcuchy powstają dzięki rejestracji procedur obsługi, każde ogniwo ma powiązaną procedurę obsługi powodzenia lub niepowodzenia (albo obie). Procedury nieodpowiadające typowi wyniku (powodzenie lub niepowodzenie) są ignorowane. Natomiast pasujące są wywoływane, a ich wynik określa rodzaj następnej wartości — powodzenie w przypadku zwrotu wartości niebędącej obietnicą, odrzucenie w przypadku zgłoszenia wyjątku i wynik obietnicy w przypadku, gdy zostanie zwrócony.

```
new Promise((_, reject) => reject(new Error("Niepowodzenie")))
  .then(value => console.log("Procedura obsługi 1"))
  .catch(reason => {
    console.log("Znaleziono błąd " + reason);
    return "nic";
  })
  .then(value => console.log("Procedura obsługi 2", value));
// → Znaleziono błąd Error: Niepowodzenie
// → Procedura obsługi 2 nic
```

Środowisko JavaScript wykrywa nieobsłużone odrzucenia obietnic i tak samo jak w przypadku nieobsłużonych wyjątków zgłasza je jako błędy.

Sieci są problematyczne

Czasami jest za mało światła dla wronich systemów transmisji sygnałów za pomocą luster albo coś stanie na drodze sygnału i wówczas można go wysłać, ale nie dociera do adresata.

W takim wypadku funkcja zwrotna, do której miał zostać wysłany sygnał, nie zostaje wywołana, przez co program zatrzyma się, nawet nie zauważając, że był jakiś problem. Dobrze by było, aby w razie braku odpowiedzi przez pewien czas jakiś mechanizm ogłaszał **przekroczenie czasu** i zgłaszał błąd.

Błędy transmisji często mają charakter losowy, na przykład reflektory samochodu zakłócą przesyłanie sygnału świetlnego, i wówczas wystarczy ponowić próbę, aby się udało. Skoro tak, to wbudujemy naszej funkcji wysyłającej żądania mechanizm automatycznego ponawiania próby kilka razy przed zgłoszeniem błędu.

A ponieważ już ustaliliśmy, że obietnice to dobre rozwiązanie, napiszemy naszą funkcję wysyłającą żądania tak, aby zwracała właśnie obietnicę. Funkcje zwrotne i obietnice nie różnią się pod względem tego, co mogą wyrażać. Funkcje zwrotne można opakowywać, aby udostępniały interfejs obietnicowy i odwrotnie.

Nawet jeśli żądanie zostanie wykonane i nadejdzie odpowiedź, ta może oznaczać niepowodzenie — gdy na przykład żądanie próbuje użyć niezdefiniowanego typu żądania lub procedura obsługi zgłosi błąd. Dlatego metody `send` i `defineRequestType` są zbudowane zgodnie z opisaną wcześniej zasadą, tzn. w pierwszym argumencie przyjmują powód niepowodzenia, a w drugim — rzeczywisty wynik.

W naszym opakowaniu będą to rozwiązanie i odrzucenie obietnicy.

```
class Timeout extends Error {}
  function request(nest, target, type, content) {
    return new Promise((resolve, reject) => {
      let done = false;
      function attempt(n) {
        nest.send(target, type, content, (failed, value) => {
          done = true;
          if (failed) reject(failed);
          else resolve(value);
        });
        setTimeout(() => {
          if (done) return;
          else if (n < 3) attempt(n + 1);
          else reject(new Timeout("Czas minął."));
        }, 250);
      }
```

```
    attempt(1);
  });
}
```

To rozwiązanie się sprawdzi, ponieważ obietnice mogą być rozwiązywane i odrzucane tylko raz. Pierwsze wywołanie `resolve` lub `reject` określa wynik obietnicy, a każde kolejne, takie jak przekroczenie czasu nadchodzące po zakończeniu żądania lub żądanie wracające po zakończeniu innego żądania, są ignorowane.

Do utworzenia pętli asynchronicznej służącej do obsługi ponawiania prób potrzebujemy funkcji rekurencyjnej — zwykła funkcja nie pozwala na wstrzymanie pracy, aby poczekać na zakończenie asynchronicznej operacji. Funkcja `attempt` wykonuje jedną próbę wysłania żądania. Ustawia też limit czasu 250 milisekund, po upływie którego, jeśli nie nadejdzie odpowiedź, wykonuje kolejną próbę, a jeśli wykonała już cztery próby, odrzuca obietnicę, zwracając obiekt `Timeout` jako przyczynę.

Ponawianie próby co ćwierć sekundy i rezygnacja z dalszych prób po upływie sekundy to arbitralnie dobrane działania. Istnieje nawet możliwość — jeśli żądanie dotrze do adresata, tylko procedura obsługi potrzebuje odrobinę więcej czasu — że żądania zostaną dostarczone więcej razy. Będziemy mieli to na uwadze podczas pisania naszych procedur, dzięki czemu powielone komunikaty powinny być nieszkodliwe.

Generalnie dziś nie stworzymy światowej klasy niezawodnej sieci, ale to nie szkodzi, ponieważ na razie wrony nie mają zbyt wysokich wymagań.

Aby całkowicie odizolować się od funkcji zwrotnych, zdefiniujemy opakowanie dla funkcji `defineRequestType`, które pozwoli funkcji obsługowej zwrócić obietnicę lub zwykłą wartość i przekaże ją do funkcji zwrotnej za nas.

```
function requestType(name, handler) {
  defineRequestType(name, (nest, content, source,
                           callback) => {
    try {
      Promise.resolve(handler(nest, content, source))
      .then(response => callback(null, response),
            failure => callback(failure));
    } catch (exception) {
      callback(exception);
    }
  });
}
```

Metoda `Promise.resolve` konwertuje wartość zwróconą przez procedurę obsługi na obietnicę, jeśli jeszcze to nie zostało zrobione.

Zwróć uwagę, że wywołanie procedury obsługi musiało zostać opakowane w blok `try`, aby każdy zgłoszony bezpośrednio przez nią wyjątek był przekazywany do funkcji zwrotnej. Jest to dobra ilustracja trudności, jakie

sprawia prawidłowa obsługa błędów przy użyciu samych funkcji zwrotnych — łatwo zapomnieć o poprawnym poprowadzeniu takich wyjątków, a jeśli się tego nie zrobi, błędy będą zgłaszane do niewłaściwych funkcji zwrotnych. Obietnice w dużym stopniu to automatyzują, dzięki czemu pozwalają łatwiej unikać błędów.

Kolekcje obietnic

Każdy gniazdowy komputer ma własność `neighbors`, w której przechowuje tablicę innych gniazd w zasięgu transmisji. Aby sprawdzić, które gniazda są w danej chwili dostępne, możemy napisać funkcję wysyłającą żądanie „ping" (proste żądanie z prośbą o odpowiedź) do każdego z nich i sprawdzającą, które odpowiedziały.

Podczas pracy z kolekcjami obietnic działających jednocześnie przydatna jest funkcja `Promise.all`. Zwraca ona obietnicę, która czeka na wszystkie obietnice w tablicy, po czym zostaje rozwiązana do tablicy wartości zwróconych przez te obietnice (w takiej samej kolejności jak w oryginalnej tablicy). Jeśli którakolwiek obietnica zostanie odrzucona, cały wynik funkcji `Promise.all` zostaje odrzucony.

```
requestType("ping", () => "pong");
function availableNeighbors(nest) {
  let requests = nest.neighbors.map(neighbor => {
    return request(nest, neighbor, "ping")
      .then(() => true, () => false);
  });
  return Promise.all(requests).then(result => {
    return nest.neighbors.filter((_, i) => result[i]);
  });
}
```

Wszystkie obietnice nie mogą zostać odrzucone tylko dlatego, że jedno z sąsiednich gniazd jest niedostępne, ponieważ wówczas niczego byśmy się nie dowiedzieli. Dlatego funkcja mapująca sąsiednie gniazda na obietnice żądań wiąże procedury obsługi, które w przypadku pozytywnej odpowiedzi na żądanie zwracają `true`, a w przypadku negatywnej — `false`.

W procedurze obsługi połączonej obietnicy funkcja `filter` usuwa z tablicy `neighbors` elementy o wartości `false`. Wykorzystujemy tu fakt, że funkcja `filter` przekazuje indeks tablicowy bieżącego elementu jako drugi argument do swojej funkcji filtrującej (podobnie działają metody `map`, `some` i inne tego rodzaju metody tablicowe wyższego rzędu).

Zalewanie sieci

Możliwość komunikacji tylko między sąsiednimi gniazdami znacznie ogranicza przydatność tej sieci.

Jeśli chcemy rozsyłać informacje do całej sieci, to jedną z możliwości jest stworzenie typu żądania, które jest automatycznie przekazywane do sąsiadów. Sąsiedzi przekazują je do swoich sąsiadów i tak dalej, aż informacja obiegnie całą sieć.

```
import {everywhere} from "./crow-tech";
everywhere(nest => {
  nest.state.gossip = [];
});

function sendGossip(nest, message, exceptFor = null) {
  nest.state.gossip.push(message);
  for (let neighbor of nest.neighbors) {
    if (neighbor == exceptFor) continue;
    request(nest, neighbor, "gossip", message);
  }
}

requestType("gossip", (nest, message, source) => {
  if (nest.state.gossip.includes(message)) return;
  console.log(`${nest.name} received gossip '${
               message}' from ${source}`);
  sendGossip(nest, message, source);
});
```

Aby uniknąć wysyłania w nieskończoność jednej wiadomości po sieci, każde gniazdo przechowuje tablicę łańcuchów, które już widziało. Tablicę tę definiujemy za pomocą funkcji `everywhere` — która wykonuje kod na każdym gnieździe — dodającej własność do obiektu `state` gniazda, w której będziemy przechowywać stan lokalny gniazda.

Gdy gniazdo odbierze drugi raz tę samą wiadomość, co jest bardzo prawdopodobne, zważywszy, że są one na ślepo rozsyłane na wszystkie strony, ignoruje ją. Kiedy natomiast odbierze nową wiadomość, natychmiast rozsyła ją do wszystkich sąsiadów z wyjątkiem tego, od którego ona nadeszła.

W efekcie nowa plotka rozprzestrzenia się w sieci jak plama atramentu na wodzie. Nawet jeśli niektóre połączenia nie będą działać, plotka dotrze do miejsca przeznaczenia alternatywnymi drogami.

Ten rodzaj komunikacji sieciowej nazywa się **zalewaniem** (ang. *flooding*) — sieć zalewa się informacją, aż dotrze do wszystkich węzłów.

Rozsyłanie komunikatów

Jeśli dany węzeł chce przekazać informację tylko jednemu innemu węzłowi, to technika zalewania jest mało efektywna, szczególnie w przypadku dużych sieci, w których konieczne byłoby przesyłanie dużej ilości bezużytecznych danych.

Inne podejście polega na przesyłaniu wiadomości od węzła do węzła, aż uda się dotrzeć do celu. Trudność w tym podejściu stanowi to, że trzeba znać układ sieci. Aby wysłać żądanie w kierunku odległego gniazda, należy wiedzieć, które sąsiednie gniazdo leży na drodze do celu. Wysyłka w niewłaściwym kierunku na niewiele się zda.

Ponieważ każde gniazdo wie tylko o istnieniu bezpośrednich sąsiadów, nie ma informacji potrzebnych do obliczenia trasy. Musimy w jakiś sposób rozesłać informację o tych połączeniach do wszystkich gniazd, najlepiej w sposób umożliwiający wprowadzanie zmian z czasem, gdy na przykład gniazdo zostanie porzucone lub zostanie zbudowane nowe gniazdo.

Ponownie możemy zastosować zalewanie, tylko zamiast sprawdzać, czy dana wiadomość została odebrana, teraz możemy sprawdzać, czy nowy zestaw sąsiadów danego gniazda pasuje do bieżącego zestawu, który mamy.

```
requestType("connections", (nest, {name, neighbors},
                            source) => {
  let connections = nest.state.connections;
  if (JSON.stringify(connections.get(name)) ==
      JSON.stringify(neighbors)) return;
  connections.set(name, neighbors);
  broadcastConnections(nest, name, source);
});

function broadcastConnections(nest, name, exceptFor = null) {
  for (let neighbor of nest.neighbors) {
    if (neighbor == exceptFor) continue;
    request(nest, neighbor, "connections", {
      name,
      neighbors: nest.state.connections.get(name)
    });
  }
}

everywhere(nest => {
  nest.state.connections = new Map;
  nest.state.connections.set(nest.name, nest.neighbors);
  broadcastConnections(nest, nest.name);
});
```

W operacji porównywania użyłem `JSON.stringify`, ponieważ operator `==` w odniesieniu do obiektów i tablic zwraca prawdę tylko wtedy, gdy dwie wartości są dokładnie identyczne, a nie o to nam w tym przypadku chodzi. Porównywanie łańcuchów JSON to prymitywny, ale bardzo skuteczny sposób na porównanie ich treści.

Węzły natychmiast zaczynają rozsyłać swoje połączenia, dzięki czemu, jeśli żadne gniazdo nie jest całkowicie niedostępne, każde gniazdo wkrótce powinno otrzymać mapę aktualnego grafu sieci.

Jak pokazałem w rozdziale 7., grafy doskonale nadają się do wyszukiwania tras. Jeśli będziemy znać trasę do miejsca przeznaczenia wiadomości, będziemy wiedzieli, w którym kierunku ją wysłać.

Poniższa funkcja `findRoute`, która bardzo przypomina funkcję `findRoute` z rozdziału 7., szuka drogi do danego węzła w sieci. Nie zwraca jednak całej trasy, a jedynie następny krok. Kolejne gniazdo zdecyduje, gdzie wysłać wiadomość, na podstawie aktualnych informacji o sieci.

```
function findRoute(from, to, connections) {
  let work = [{at: from, via: null}];
  for (let i = 0; i < work.length; i++) {
    let {at, via} = work[i];
    for (let next of connections.get(at) || []) {
      if (next == to) return via;
      if (!work.some(w => w.at == next)) {
        work.push({at: next, via: via || next});
      }
    }
  }
  return null;
}
```

Teraz możemy utworzyć funkcję wysyłającą wiadomości na dużą odległość. Jeśli wiadomość będzie zaadresowana do bezpośredniego sąsiada, zostanie dostarczona jak zwykle. Jeśli nie, zostanie spakowana w obiekt i wysłana do sąsiada znajdującego się bliżej celu za pomocą żądania typu `"route"`, które będzie stanowić sygnał dla sąsiada, aby powtórzyć tę samą operację.

```
function routeRequest(nest, target, type, content) {
  if (nest.neighbors.includes(target)) {
    return request(nest, target, type, content);
  } else {
    let via = findRoute(nest.name, target,
                        nest.state.connections);
    if (!via) throw new Error(`No route to ${target}`);
    return request(nest, via, "route",
                   {target, type, content});
  }
}

requestType("route", (nest, {target, type, content}) => {
  return routeRequest(nest, target, type, content);
});
```

Na prosty system komunikacji nałożyliśmy kilka warstw funkcjonalności, aby ułatwić sobie korzystanie z niego. To całkiem wierny (choć uproszczony) model działania prawdziwych sieci komputerowych.

Jedną z typowych cech sieci komputerowych jest to, że są zawodne — budowane na nich abstrakcje są pomocne, ale nie można wyabstrahować błędu sieciowego. Dlatego programowanie sieciowe w dużym stopniu polega na przewidywaniu i rozwiązywaniu problemów.

Funkcje asynchroniczne

Wrony są znane z tego, że ważne informacje duplikują i przechowują w kilku gniazdach. Dzięki temu dane nie zostają utracone, gdy jastrząb zniszczy jedno z nich.

Jeśli gniazdo potrzebuje informacji, której nie ma we własnej pamięci, może wysyłać zapytania do losowych gniazd w sieci, aż znajdzie takie, które posiada odpowiednie dane.

```
requestType("storage", (nest, name) => storage(nest, name));
function findInStorage(nest, name) {
  return storage(nest, name).then(found => {
    if (found != null) return found;
    else return findInRemoteStorage(nest, name);
  });
}

function network(nest) {
  return Array.from(nest.state.connections.keys());
}

function findInRemoteStorage(nest, name) {
  let sources = network(nest).filter(n => n != nest.name);
  function next() {
    if (sources.length == 0) {
      return Promise.reject(new Error("Nie znaleziono."));
    } else {
      let source = sources[Math.floor(Math.random() *
                                      sources.length)];
      sources = sources.filter(n => n != source);
      return routeRequest(nest, source, "storage", name)
        .then(value => value != null ? value : next(),
              next);
    }
  }
  return next();
}
```

Ponieważ obiekt `connections` jest słownikiem (`Map`), nie działa na niego metoda `Object.keys`. Ma on **metodę** `keys`, ale zwracającą iterator, a nie tablicę. Iterator (lub wartość iterowalną) można przekonwertować na tablicę za pomocą funkcji `Array.from`.

Jednak nawet obietnice nie pozwolą uniknąć niezręczności w tym kodzie. Operacje asynchroniczne łączy się w łańcuchy w niezbyt oczywisty sposób. Znowu potrzebujemy funkcji rekurencyjnej (`next`), która będzie modelowała pętlę przeglądania gniazd.

A działania wykonywane przez ten kod będą całkowicie liniowe — najpierw musi zakończyć się wykonywanie jednej czynności, aby można było rozpocząć następną. W synchronicznym modelu programowania łatwiej byłoby to wyrazić.

Na szczęście w JavaScripcie obliczenia asynchroniczne można wyrażać przy użyciu pseudosynchronicznego kodu. Funkcja `async` niejawnie zwraca obietnicę, która może poczekać na inne obietnice w sposób sprawiający wrażenie synchronicznego wykonywania.

Funkcję `findInStorage` możemy zmodyfikować do następującej postaci:

```
async function findInStorage(nest, name) {
  let local = await storage(nest, name);
  if (local != null) return local;
  let sources = network(nest).filter(n => n != nest.name);
  while (sources.length > 0) {
    let source = sources[Math.floor(Math.random() *
                                    sources.length)];
    sources = sources.filter(n => n != source);
    try {
      let found = await routeRequest(nest, source, "storage",
                                     name);
      if (found != null) return found;
    } catch (_) {}
  }
  throw new Error("Nie znaleziono.");
}
```

Funkcję asynchroniczną oznacza się słowem kluczowym `async` stawianym przed słowem `function`. Metody można także definiować jako asynchroniczne, stawiając słowo kluczowe `async` przed ich nazwą. W chwili wywołania takiej funkcji lub metody zostaje zwrócona obietnica. Kiedy pojawi się wynik, obietnica zostaje rozwiązana. Pojawienie się wyjątku powoduje odrzucenie obietnicy.

Wewnątrz funkcji asynchronicznej można używać słowa `await`. Jeśli zostanie wstawione przed wyrażeniem, stanowi sygnał, aby z dalszym wykonywaniem funkcji poczekać na rozwiązanie obietnicy.

Taka funkcja, w odróżnieniu od zwykłych funkcji języka JavaScript, nie jest normalnie wykonywana od początku do końca za jednym zamachem. Może zostać *zamrożona* w dowolnym punkcie za pomocą słowa kluczowego `await` i wznowiona w późniejszym czasie.

Pisanie bardziej skomplikowanego kodu asynchronicznego przy użyciu tej notacji jest zazwyczaj łatwiejsze niż przy bezpośrednim użyciu obietnic. Nawet gdy programista chce zrobić coś, co nie pasuje do modelu synchronicznego, na przykład wykonać kilka operacji jednocześnie, bez problemu połączy `await` z bezpośrednim użyciem obietnic.

Generatory

Możliwość wstrzymywania i wznawiania jest cechą wyłącznie funkcji asynchronicznych. W języku JavaScript dostępne są także funkcje zwane **generatorami**. Są one podobne do funkcji asynchronicznych, ale nie wykorzystują obietnic.

Generatorem jest funkcja zdefiniowana za pomocą słowa kluczowego `function*` (słowo kluczowe `function` z gwiazdką). Wywołany generator zwraca iterator. O iteratorach była mowa w rozdziale 6.

```
function* powers(n) {
  for (let current = n;; current *= n) {
    yield current;
  }
}
for (let power of powers(3)) {
  if (power > 50) break;
  console.log(power);
}
// → 3
// → 9
// → 27
```

Kiedy wywołamy funkcję `powers`, to początkowo będzie zamrożona. Następnie każde wywołanie `next` na iteratorze będzie powodowało wykonywanie funkcji do wyrażenia `yield`, po czym nastąpi jej wstrzymanie i ustawienie zwróconej wartości jako następnej wartości zwróconej przez iterator. Gdy funkcja zwróci sterowanie (ta w przykładzie nigdy tego nie robi), iterator kończy swoją pracę.

Generatory często bardzo ułatwiają pisanie iteratorów. Iterator klasy `Group` (z ćwiczenia „Iterowalne grupy” w rozdziale 6.) można zdefiniować jako generator w następujący sposób:

```
Group.prototype[Symbol.iterator] = function*() {
  for (let i = 0; i < this.members.length; i++) {
    yield this.members[i];
  }
};
```

Nie trzeba już tworzyć obiektu do przechowywania stanu iteracji — generatory automatycznie zapisują swój stan lokalny po każdym wygenerowaniu wartości. Wyrażenia `yield` mogą występować tylko bezpośrednio w generatorach, tzn. nie mogą znajdować się w zdefiniowanych w nich funkcjach wewnętrznych. Kiedy generator generuje wartość, zapisuje tylko lokalne środowisko i miejsce, w którym utworzył wartość.

Funkcja asynchroniczna to specjalny rodzaj generatora. W chwili wywołania tworzy obietnicę, która zostaje rozwiązana w chwili zakończenia działania i odrzucona w przypadku wystąpienia wyjątku. Kiedy wygeneruje obietnicę, wynik tej obietnicy (wartość lub zgłoszony wyjątek) jest wynikiem wyrażenia `await`.

Pętla zdarzeń

Programy asynchroniczne są wykonywane po kawałku. Każdy fragment może rozpocząć pewne operacje i zaplanować wykonanie kodu, gdy operacje te zostaną ukończone lub się nie powiodą. Między tymi fragmentami program bezczynnie oczekuje na kolejną czynność.

Zatem funkcje zwrotne nie są bezpośrednio wywoływane przez kod, który je zaplanował. Jeśli wywołam `setTimeout` w funkcji, to funkcja ta zakończy działanie, zanim zostanie wywołana funkcja zwrotna. A kiedy funkcja zwrotna zakończy działanie, sterowanie nie wraca do funkcji, która zaplanowała jej wykonanie.

Operacje asynchroniczne są wykonywane na własnym pustym stosie wywołań funkcji. To jedna z przyczyn, dla których zarządzanie wyjątkami w kodzie asynchronicznym bez użycia obietnic jest takie trudne. Jako że na początku każdego wywołania zwrotnego stos jest prawie pusty, procedury `catch` w chwili zgłoszenia wyjątku nie będą obecne na stosie.

```
try {
  setTimeout(() => {
    throw new Error("Siup");
  }, 20);
} catch (_) {
  // To nie zostanie wykonane
  console.log("Złapany!");
}
```

Niezależnie od tego, jak krótko po sobie następują zdarzenia — na przykład przekroczenia limitu czasu albo nadejścia żądań — środowisko JavaScript wykonuje programy pojedynczo. Można sobie to wyobrazić tak, jakby program był wykonywany w jednej dużej pętli zwanej **pętlą zdarzeń** (ang. *event loop*). Kiedy nie ma nic do roboty, pętla zostaje wstrzymana. Kiedy natomiast pojawiają się zdarzenia, zostają dodane do kolejki i ich kod jest wykonywany po kolei. A ponieważ dwie operacje nie mogą być wykonywane jednocześnie, powolny kod może opóźnić obsługę innych zdarzeń.

Poniższy przykład ustawia limit czasu, a następnie ociąga się aż do przekroczenia go, powodując opóźnienie przekroczenia limitu czasu.

```
let start = Date.now();
setTimeout(() => {
  console.log("Upływ limitu czasu po ", Date.now() - start);
}, 20);
while (Date.now() < start + 50) {}
console.log("Marnowanie czasu do ", Date.now() - start);
// → Marnowanie czasu do 50
// → Upływ limitu czasu po 55
```

Rozwiązanie lub odrzucenie obietnicy zawsze jest nowym zdarzeniem. Nawet gdy obietnica jest już rozwiązana, oczekiwanie na nią spowoduje wykonanie funkcji zwrotnej dopiero po zakończeniu bieżącego skryptu, nie od razu.

```
Promise.resolve("Zrobione.").then(console.log);
console.log("Ja najpierw!");
// → Ja najpierw!
// → Zrobione.
```

W dalszych rozdziałach opisuję różne inne typy zdarzeń wykorzystywanych w pętli zdarzeń.

Błędy asynchroniczne

Kiedy program jest wykonywany synchronicznie, w jednym ciągu, jedynymi zmianami stanu są te, których dokonuje sam program. W przypadku programów asynchronicznych jest inaczej — w czasie ich wykonywania występują luki, podczas których wykonywany jest inny kod.

Spójrzmy na przykład. Nasze wrony lubią wiedzieć, ile małych wykluło się w całej miejscowości w ciągu roku. Gniazda przechowują tę informację w swojej pamięci. Poniższy kod próbuje wyliczyć wartości z wszystkich gniazd dla określonego roku:

```
function anyStorage(nest, source, name) {
  if (source == nest.name) return storage(nest, name);
  else return routeRequest(nest, source, "storage", name);
}

async function chicks(nest, year) {
  let list = "";
  await Promise.all(network(nest).map(async name => {
```

```
    list += `${name}: ${
      await anyStorage(nest, name, `Małe w ${year}`)
    }\n`;
  }));
  return list;
}
```

Konstrukcja `async name =>` pokazuje, że funkcje strzałkowe także można definiować jako asynchroniczne za pomocą słowa kluczowego `async`.

Na pierwszy rzut oka ten kod nie budzi podejrzeń… Mapuje asynchroniczną funkcję strzałkową przez zbiór gniazd, tworząc tablicę obietnic, a następnie przy użyciu `Promise.all` czeka na nie wszystkie, aby zwrócić utworzoną przez nie listę.

A jednak ten kod zawiera poważny błąd. Zawsze będzie zwracał tylko jeden wiersz wyniku z gniazda, które najpóźniej odpowiedziało.

Wiesz dlaczego?

Problem dotyczy operatora `+=`, który pobiera *bieżącą* wartość `list`, jaka była w czasie rozpoczęcia wykonywania instrukcji, po czym, po zakończeniu oczekiwania przez `await`, ustawia wiązanie `list` na tę wartość z dodatkiem łańcucha.

Tylko że między rozpoczęciem wykonywania instrukcji i czasem jej zakończenia występuje asynchroniczna luka. Wyrażenie `map` zostaje wykonane, zanim cokolwiek zostaje dodane do listy, w związku z czym każdy operator `+=` zaczyna od pustego łańcucha i kończy, po zakończeniu pobierania danych z pamięci, ustawieniem `list` na jednowierszową listę — wynik dodania swojego wiersza do pustego łańcucha.

Tej sytuacji łatwo można uniknąć przez zwrócenie wierszy ze zmapowanych obietnic i wywołanie `join` na wyniku `Promise.all` zamiast budowania listy poprzez zmienianie wiązania. Jak zwykle obliczanie nowych wartości jest mniej zagrożone błędem niż modyfikowanie istniejących.

```
async function chicks(nest, year) {
  let lines = network(nest).map(async name => {
    return name + ": " +
    await anyStorage(nest, name, `chicks in ${year}`);
  });
  return (await Promise.all(lines)).join("\n");
}
```

Łatwo popełnić taki błąd, zwłaszcza gdy używa się metody `await`, i należy pamiętać, gdzie w kodzie znajdują się luki. Zaletą bezpośredniej asynchroniczności w języku JavaScript (uzyskiwanej przez wykorzystanie funkcji zwrotnych, obietnic lub metody `await`) jest to, że łatwo dostrzec te luki.

Podsumowanie

Programowanie asynchroniczne umożliwia wyrażanie oczekiwania na zakończenie długotrwałych operacji bez zamrażania programu. Środowiska JavaScript zwykle implementują ten styl programowania przy użyciu funkcji zwrotnych, czyli wywoływanych po zakończeniu określonej operacji. Pętla zdarzeń planuje wywołanie takich funkcji zwrotnych w odpowiednim momencie, jednej po drugiej, aby ich wywołania nie nakładały się na siebie.

Programowanie asynchroniczne jest łatwiejsze dzięki obietnicom, czyli obiektom reprezentującym czynności, które mogą zakończyć się w przyszłości, i funkcjom asynchronicznym, które umożliwiają pisanie programów asynchronicznych tak, jakby były synchroniczne.

Ćwiczenia

Śledzenie skalpela

Wrony w miasteczku mają stary skalpel, za pomocą którego od czasu do czasu wykonują misje specjalne, na przykład przecinają siatki przeciw owadom lub opakowania. Aby można było szybko znaleźć aktualne miejsce przechowywania skalpela, po każdym jego przeniesieniu zostaje dodany wpis do pamięci gniazda, z którego został zabrany, oraz gniazda, do którego został przeniesiony. Wpis ten ma nazwę `"scalpel"` i zawiera informację o nowym miejscu przechowywania przyrządu.

To oznacza, że aby znaleźć skalpel, wystarczy podążać śladem tych wpisów jak śladem okruszków chleba. W ten sposób zawsze w końcu uda się dotrzeć do aktualnego miejsca przechowywania przyrządu.

Napisz funkcję asynchroniczną `locateScalpel`, która będzie w taki sposób znajdowała skalpel, zaczynając od gniazda, na którym zostanie wywołana. Możesz wykorzystać zdefiniowaną wcześniej funkcję `anyStorage` w celu uzyskania dostępu do pamięci w dowolnych gniazdach. Skalpel jest już tak długo w obiegu, że można przyjąć, iż każde gniazdo ma już w swojej pamięci wpis `"scalpel"`.

Następnie napisz tę samą funkcję bez użycia `async` i `await`.

Czy nieudane żądania prawidłowo mają postać odrzuceń zwróconej obietnicy w obu wersjach? Dlaczego?

Budowa metody Promise.all

Dla tablicy obietnic metoda `Promise.all` zwraca obietnicę oczekującą na ukończenie wszystkich obietnic w tej tablicy. Następnie generuje tablicę wartości wynikowych. Jeżeli któraś obietnica w tablicy nie zostanie spełniona, to samo dzieje się z obietnicą zwróconą przez `all`, a przyczyną niepowodzenia jest przyczyna tej niespełnionej obietnicy.

Zaimplementuj coś podobnego w postaci zwykłej funkcji o nazwie `Promise_all`.

Pamiętaj, że po tym, jak obietnica zostanie spełniona lub odrzucona, nie może zostać ponownie spełniona ani odrzucona i dalsze wywołania funkcji, które ją rozwiązują, są ignorowane. To może uprościć sposób obsługi odrzucenia obietnicy.


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

JavaScript: oto język Internetu!

JavaScript ma długą i ciekawą historię. Obecnie to jeden z najpopularniejszych języków programowania. Jego zalety to nowoczesność, wszechstronność, elastyczność i wydajność. Jest przy tym idealny dla początkujących użytkowników: można się go łatwo nauczyć i równocześnie od razu przyzwyczaić się do stosowania dobrych praktyk i pisania czystego, ładnego kodu. Mimo to uzyskanie prawdziwej biegłości wymaga pracy i ćwiczeń. Jest to jednak wysiłek, który warto podjąć, gdyż JavaScript jest doskonałym wyborem dla profesjonalnych twórców aplikacji. Co więcej, wszystko wskazuje na to, że jeszcze długo będzie rozwijany i doskonalony przez skupioną wokół niego społeczność entuzjastów.

To trzecie, wzbogacone i uzupełnione wydanie popularnego podręcznika programowania dla początkujących. Znalazło się tu wyczerpujące wyjaśnienie podstawowych zasad programowania oraz struktury języka JavaScript. Omówiono techniki testowania kodu i obsługi błędów, tworzenia kodu modułowego, zaprezentowano również koncepcję programowania asynchronicznego. Teorii towarzyszą przykłady kodu, opisy projektów oraz liczne ćwiczenia do samodzielnego wykonania. Poszczególne koncepcje i techniki są przedstawiane na przykładach konkretnych, działających aplikacji, takich jak gra przeglądarkowa, prosty język programowania i program do rysowania.

W książce między innymi:

- solidne podstawy: składnia, struktury sterujące i praca z danymi
- zasady programowania obiektowego i funkcyjnego
- tworzenie skryptów do wykonywania w przeglądarkach
- podstawy projektowania aplikacji sieciowych
- model DOM i jego zastosowanie
- korzystanie z Node.js

Marijn Haverbeke

jest niemieckim niezależnym programistą i poliglotą. Napisał kilka książek o programowaniu. Uwielbia pisać kod w wielu różnych językach programowania. Obecnie analizuje i testuje systemy bazodanowe i interfejsy API, a także tworzy przeróżne aplikacje open source. Chętnie dzieli się swoją wiedzą.

ISBN 978-83-283-6362-5