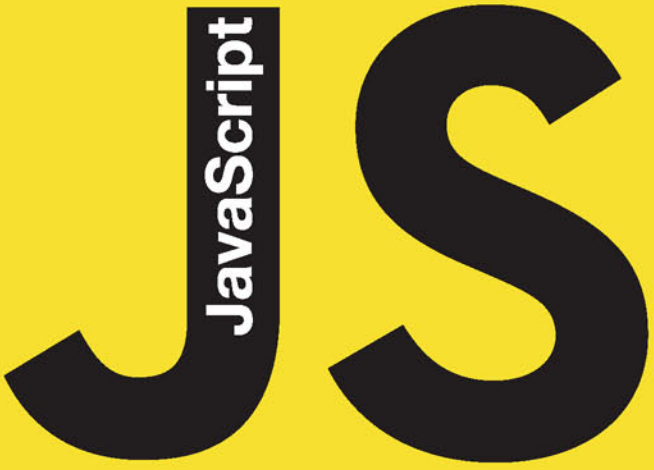
O'REILLY®

Helion

KYLE SIMPSON

ASYNCHRONICZNOŚĆ *i* WYDAJNOŚĆ

TAJNIKI JĘZYKA



Tytuł oryginału: You Don't Know JS: Async & Performance

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-2172-4

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)

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

Printed in Poland.

Spis treści

ROZDZIAŁ 3.
Obietnice

W rozdziale 2. przedstawiłem dwie główne kategorie wad użycia wywołań zwrotnych w celu wyrażenia asynchroniczności w programie oraz zarządzania współbieżnością: brak sekwencyjności i brak zaufania. Skoro dokładnie poznałeś problemy, to możemy przejść do wzorców pozwalających na ich rozwiązanie.

Problemem, którym chcę się zająć na początku, jest odwrócenie kontroli. Zaufanie to dość delikatna materia i bardzo łatwo można je stracić.

Przypomnij sobie, jak kod zapewniający kontynuację działania programu umieściliśmy w funkcji wywołania zwrotnego przekazywanego później do innej części kodu (potencjalnie nawet do zewnętrznego) i trzymaliśmy kciuki, aby ten kod działał prawidłowo po uruchomieniu wywołania zwrotnego.

Zdecydowaliśmy się na takie podejście, ponieważ chcieliśmy wyrazić: „Oto co się wydarzy *później*, po zakończeniu wykonywania bieżących kroków".

Czy można zrobić cokolwiek, aby nie dochodziło do wspomnianego odwrócenia kontroli? Czy nie lepiej zastosować zupełnie inne podejście — zamiast przekazywać na zewnątrz kod kontynuujący działanie programu, oczekiwać poinformowania o zakończeniu zadania, co pozwoli kodowi programu na określenie kolejnych kroków?

Takie podejście jest określane mianem *obietnicy*.

Obietnice zaczęły szturmem zdobywać świat JavaScript, ponieważ programiści i twórcy specyfikacji rozpaczliwie szukali sposobu na wyrwanie się z piekła wywołań zwrotnych. Tak naprawdę większość nowych asynchronicznych API dodawanych na platformie JavaScript i DOM została zbudowana na bazie obietnic. Dlatego też dobrym pomysłem będzie dokładne poznanie obietnic.

Słowo „natychmiast" jest dość często używane w rozdziale i ogólnie rzecz biorąc, odnosi się do pewnej akcji rozwiązania obietnicy. Jednak w praktycznie wszystkich przypadkach, słowa „natychmiast" użyłem w odniesieniu do zachowania kolejki zadań (patrz rozdział 1.), a nie w sensie ściśle synchronicznego *teraz*.

Czym jest obietnica?

Kiedy programista decyduje się na naukę nowej technologii bądź wzorca, to pierwszy jego krok zwykle można opisać jako „pokaż mi kod". Całkiem naturalne jest szybkie rozpoczęcie pracy z nową technologią, a następnie stopniowe jej poznawanie w praktyce.

Jednak okazuje się, że pewne kwestie mogą umknąć, gdy zapoznajesz się tylko z samym API. Obietnica to jedno z tych narzędzi, którego przeznaczenie i sposób użycia mogą być zupełnie oczywiste na podstawie analizy jego stosowania przez innych. Ograniczenie się do jedynie poznawania i używania API okaże się znacznie trudniejszym podejściem.

Dlatego też zanim pokażę Ci przykład kodu opartego na obietnicy, najpierw chciałbym dokładnie wyjaśnić jej koncepcję. Mam nadzieję, że dzięki temu będziesz radził sobie znacznie lepiej podczas integracji obietnic we własnym kodzie asynchronicznym.

Mając na uwadze powyższe kwestie, spójrzmy na dwie różne analogie pokazujące, czym *jest* obietnica.

Przyszła wartość

Wyobraź sobie następującą sytuację: udajesz się do restauracji typu fast food, zamawiasz cheeseburgera i płacisz za tę kanapkę. Przez złożenie i opłacenie zamówienia dokonałeś żądania *wartości* zwrotnej (cheeseburgera). W ten sposób zainicjowałeś transakcję.

Jednak bardzo często się zdarza, że kanapka nie jest dostępna natychmiast. Sprzedawca daje Ci więc coś zamiast cheeseburgera: kartkę z numerem zamówienia. Ten numer zamówienia jest *obietnicą*, która gwarantuje Ci otrzymanie cheeseburgera w przyszłości.

W ręku trzymasz rachunek i numer zamówienia. Ponieważ numer zamówienia przedstawia *cheeseburgera w przyszłości*, nie musisz się więcej martwić tą kanapką, poza tym, że jesteś głodny!

Podczas oczekiwania na przyrządzenie kanapki możesz zajmować się innymi rzeczami, na przykład wysłać do przyjaciela SMS o treści: „Cześć, czy masz ochotę coś przekąsić? Wybrałem się na cheeseburgera".

Pozostajesz spokojny o *cheeseburgera w przyszłości*, choć jeszcze nie trzymasz go w ręku. Twój mózg pozwala na to, ponieważ numer zamówienia traktuje jako miejsce zarezerwowane dla cheeseburgera. Wspomniane miejsce zarezerwowane oznacza wartość *niezależną od czasu*. To jest *przyszła wartość*.

W pewnej chwili słyszysz: „Zamówienie numer 113!". Podchodzisz do lady, trzymając numer zamówienia, który następnie podajesz sprzedawcy i w zamian odbierasz swojego cheeseburgera.

Innymi słowy, Twoja *przyszła wartość* jest już gotowa. Obietnicę wartości wymieniłeś na konkretną wartość.

Poza przedstawioną powyżej sytuacją mogą się zdarzyć jeszcze inne. Na przykład po wywołaniu numeru zamówienia podchodzisz do lady, ale zamiast upragnionej kanapki sprzedawca z przykrością informuje Cię: „Bardzo nam przykro, ale cheeseburgery się skończyły". Pomijając frustrację klienta, w takiej sytuacji poznajesz jeszcze ważną charakterystykę *przyszłej wartości* — może wskazywać sukces lub niepowodzenie.

Za każdym razem, gdy zamawiasz cheeseburgera, wiesz, że otrzymasz wskazaną kanapkę lub smutną wiadomość o jej braku i wówczas będziesz musiał zdecydować się na coś innego do jedzenia.

W kodzie nie wszystko będzie tak proste, ponieważ użyty w powyższej metaforze numer zamówienia może w ogóle nie zostać wywołany. W takim przypadku w nieskończoność pozostajemy w stanie nierozwiązanym. Do tego tematu jeszcze powrócimy w dalszej części rozdziału.

Wartości teraz i później

Przedstawiona powyżej koncepcja może wydawać się zbyt abstrakcyjna, aby zastosować ją w kodzie. Przechodzimy więc do konkretów.

Jednak zanim przejdę do przedstawienia obietnic, jeszcze na chwilę powrócimy do doskonale nam znanego kodu — wywołania zwrotne! — i zobaczymy, jak obsługuje wspomnianą *przyszłą wartość*.

Podczas tworzenia kodu odpowiedzialnego za operacje na wartości, na przykład przeprowadzającego operację matematyczną, niezależnie od tego, czy zdajesz sobie sprawę, czy nie, to jednak przyjmujemy niezwykle ważne założenie dotyczące tej wartości: to jest konkretna wartość *teraz*.

```
var x, y = 2;

console.log( x + y ); // Wynik: NaN <-- ponieważ wartość 'x' nie została jeszcze przypisana.
```

W przypadku operacji `x + y` przyjęte zostaje założenie, że obie wartości (`x` i `y`) zostały zdefiniowane. Pod względami, które wyjaśnię za chwilę, przyjmujemy założenie, że wartości `x` i `y` zostały *rozwiązane*.

Byłoby nonsensem oczekiwać, że operator + sam z siebie jest w stanie wstrzymywać się z przeprowadzeniem operacji aż do chwili, gdy obie wartości (`x` i `y`) zostaną rozwiązane (będą gotowe). Takie podejście doprowadziłoby do powstania chaosu w programie, ponieważ wykonanie niektórych poleceń kończyłoby się *teraz*, natomiast innych *później*.

Jak można traktować relacje między dwoma poleceniami, gdy którekolwiek z nich (lub nawet oba) nie zostały jeszcze zakończone? Jeżeli działanie polecenia nr 2 opiera się na zakończeniu wykonywania polecenia nr 1, to mamy dwie możliwe sytuacje. Pierwsza — polecenie nr 1 będzie zakończone *teraz* i wszystko przebiegnie zgodnie z oczekiwaniami. Druga — wykonywanie polecenia nr 1 jeszcze się nie zakończyło, a więc działanie polecenia nr 2 zakończy się niepowodzeniem.

Jeżeli taka sytuacja brzmi dla Ciebie znajomo (patrz rozdział 1.), to dobrze!

Powracamy do omawianej operacji matematycznej `x + y`. Wyobraź sobie, że chcesz powiedzieć: „Dodaj `x` i `y`, ale jeśli którakolwiek z wymienionych wartości nie jest jeszcze gotowa, to poczekaj i dodaj je natychmiast, gdy tylko stanie się to możliwe”.

Być może w tej chwili pomyślałeś o wywołaniach zwrotnych. Dobrze…

```
function add(getX,getY,cb) {
    var x, y;
    getX( function(xVal){
        x = xVal;
        // Czy obie wartości są gotowe?
        if (y != undefined) {
```

```
            cb( x + y );      // Obliczenie sumy.
        }
    } );
    getY( function(yVal){
        y = yVal;
        // Czy obie wartości są gotowe?
        if (x != undefined) {
            cb( x + y );      // Obliczenie sumy.
        }
    } );
}

// Funkcje fetchX() i fetchY() są
// synchroniczne lub asynchroniczne.
add( fetchX, fetchY, function(sum){
    console.log( sum ); // To było łatwe, prawda?
} );
```

Poświęć chwilę na podziwianie piękna (lub jego brak) powyższego fragmentu kodu.

Wprawdzie brzydota powyższego kodu pozostaje niezaprzeczalna, ale jednocześnie warto zwrócić uwagę na ważny aspekt przedstawionego podejścia asynchronicznego.

W omawianym fragmencie kodu `x` i `y` potraktowaliśmy jako przyszłe wartości, a zdefiniowana operacja `add(..)` nie sprawdza, czy obie wymienione wartości są od razu dostępne. Innymi słowy znormalizowaliśmy *teraz* i *później*, co pozwala opierać się na przewidywalnym wyniku operacji `add(..)`.

Dzięki użyciu funkcji `add(..)` przyjęte podejście pozostaje chwilowo spójne — mamy takie samo zachowanie dla *teraz* i *później* — znacznie łatwiej zrozumieć działanie przedstawionego kodu asynchronicznego.

Ujmując rzecz jeszcze prościej — nieustannie obsługujemy zarówno *teraz*, jak i *później*. Obie operacje stają się typu *później* i wszystkie są asynchroniczne.

Oczywiście tego rodzaju podejście oparte na wywołaniach zwrotnych pozostawia wiele do życzenia. To jest pierwszy mały krok na drodze do poznania korzyści wynikających z przyszłych wartości bez przejmowania się aspektem czasu — kiedy i czy w ogóle wspomniane wartości będą dostępne.

Wartość obietnicy

Więcej informacji szczegółowych o obietnicach na pewno przedstawię w dalszej części rozdziału, więc nie przejmuj się, jeśli coś jest jeszcze dla Ciebie niezrozumiałe. Poniżej pokazałem, jak przykład dodawania liczb `x` i `y` można wyrazić za pomocą funkcji obietnic.

```
function add(xPromise,yPromise) {
    // Wywołanie Promise.all([ .. ]) pobiera tablicę obietnic.
    // Wartością zwrotną jest nowa obietnica oczekująca
    // na zakończenie wszystkich pozostałych.
    return Promise.all( [xPromise, yPromise] )

    // Kiedy obietnica zostanie rozwiązana, pobieramy
    // otrzymane wartości 'X' i 'Y', a następnie je dodajemy.
    .then( function(values){
        // W poniższym poleceniu values to tablica komunikatów
        // otrzymanych z wcześniej rozwiązanych obietnic.
        return values[0] + values[1];
    } );
}
```

```
// Funkcje fetchX() i fetchY() zwracają obietnice dla
// odpowiednich wartości, które mogą być dostępne
// teraz lub później.
add( fetchX(), fetchY() )

// Mamy obietnicę odpowiedzialną za
// obliczenie sumy dwóch liczb.
// Teraz łączymy wywołania then(..) w oczekiwaniu
// na rozwiązanie zwróconej obietnicy.
.then( function(sum){
    console.log( sum ); // To było łatwiejsze!
} );
```

Mamy dwie warstwy obietnic w powyższym fragmencie kodu.

Pierwsza — funkcje `fetchX()` i `fetchY()` są wywoływane bezpośrednio, a zwracane przez nie wartości (obietnice!) są przekazywane funkcji `add(..)`. Wprawdzie wartości wspomnianych obietnic mogą być dostępne *teraz* lub *później*, ale zachowanie każdej obietnicy jest znormalizowane, aby było takie samo niezależnie od okoliczności. Wartości `X` i `Y` używamy w sposób niezależny od czasu. To są przyszłe wartości.

Druga warstwa to obietnica, że funkcja `add(..)` utworzy (za pomocą wywołania `Promise.all([..])`) i zwróci odpowiednie dane, na które oczekujemy za pomocą wywołania `then(..)`. Kiedy zakończy się wykonywanie operacji `add(..)`, przyszła wartość `sum` jest już dostępna i można ją wyświetlić. Wewnątrz funkcji `add(..)` ukrywamy logikę oczekiwania na przyszłe wartości `X` i `Y`.

Wewnątrz funkcji `add(..)` wywołanie `Promise.all([ .. ])` tworzy obietnicę (oczekującą na rozwiązanie `promiseX` i `promiseY`). Wywołanie połączone z `.then(..)` tworzy inną obietnicę, której wiersz `values[0] + values[1]` dostarcza natychmiastowe rozwiązanie (wynik operacji dodawania). Dlatego też wywołanie `then(..)` znajdujące się na końcu wywołania `add(..)` — na końcu fragmentu kodu — jest w rzeczywistości wykonywane względem drugiej zwróconej obietnicy, a nie pierwszej utworzonej przez `Promise.all([ .. ])`. Ponadto do drugiego wywołania `then(..)` nie dołączamy kolejnego, następuje utworzenie kolejnej obietnicy, którą będziemy obserwować i której będziemy używać. Szczegółowe omówienie tego rodzaju łańcucha obietnic przedstawię w dalszej części rozdziału.

Podobnie jak w przypadku zamówienia cheeseburgera, istnieje niebezpieczeństwo, że rozwiązanie obietnicy zakończy się niepowodzeniem (odrzuceniem), a nie sukcesem (spełnieniem). W przeciwieństwie do spełnionej obietnicy, gdzie wartość zawsze jest programowa, wartość w sytuacji niepowodzenia — zwykle określana mianem *powodem odrzucenia* — może być przypisana bezpośrednio przez logikę programu lub też być przypisana pośrednio na skutek zgłoszenia wyjątku w trakcie działania aplikacji.

W obietnicy wywołanie `then(..)` może w rzeczywistości pobrać dwie funkcje, pierwszą, przeznaczoną do obsługi sytuacji spełnienia (jak pokazałem wcześniej), i drugą, do obsługi sytuacji odrzucenia:

```
add( fetchX(), fetchY() )
.then(
    // Procedura obsługi w sytuacji spełnienia obietnicy.
    function(sum) {
        console.log( sum );
    },
```

```
        // Procedura obsługi w sytuacji odrzucenia obietnicy.
        function(err) {
            console.error( err ); // Koszmar!
        }
    );
```

Jeżeli wystąpi niepowodzenie podczas pobierania wartości `X` lub `Y` bądź też niepowodzenie w trakcie operacji dodawania, to obietnica zwracana przez `add(..)` zostanie odrzucona. Wartość z obietnicy będzie przekazana drugiej procedurze obsługi zdefiniowanej w `then(..)`.

Ponieważ obietnica hermetyzuje stan zależny od czasu — oczekiwanie na spełnienie lub odrzucenie — więc z zewnątrz sama w sobie pozostaje niezależna od czasu, a tym samym może być złożona (łączona) w przewidywalny sposób niezależnie od czasu lub otrzymanego wyniku.

Co więcej, po rozwiązaniu obietnicy pozostaje taka na zawsze — staje się *wartością niemodyfikowalną* w danym miejscu — i może być pobierana wielokrotnie, w zależności od potrzeb.

Ponieważ po rozwiązaniu obietnica pozostaje niemodyfikowana na zewnątrz, wartość tę można bezpiecznie przekazywać do na przykład firm trzecich bez obaw o jej przypadkową lub złośliwą modyfikację. W szczególności dotyczy to sytuacji, gdy wiele komponentów jest zainteresowanych rozwiązaniem obietnicy. Dany komponent nie ma żadnego wpływu na możliwość prowadzonej przez inny obserwacji rozwiązania obietnicy. Wprawdzie niemodyfikowalność może wydawać się zagadnieniem akademickim, ale tak naprawdę to jeden z podstawowych i najważniejszych aspektów projektu obietnicy, którego nie można zignorować.

Przedstawiłem jedną z oferujących największe możliwości i jedną z najważniejszych koncepcji dotyczących obietnic. Jeżeli poświęcisz dużo pracy, to ten sam efekt możesz osiągnąć za pomocą zawiłego połączenia wywołań zwrotnych. To jednak naprawdę nie będzie efektywna strategia, zwłaszcza jeśli będziesz zmuszony nieustannie ją stosować.

Obietnica dostarcza łatwego do wielokrotnego użycia mechanizmu hermetyzacji i tworzenia przyszłych wartości.

Zdarzenie ukończenia

Jak mogłeś zobaczyć, pojedyncza obietnica zachowuje się, jak przyszła wartość. Istnieje jeszcze inny sposób traktowania rozwiązania obietnicy: mechanizm kontroli przepływu działania programu („tymczasowe to, a później to”) dla co najmniej dwóch kroków w zadaniu asynchronicznym.

Przyjmujemy założenie wywołania funkcji `foo(..)` w celu wykonania pewnego zadania. Nie znamy lub też nie przejmujemy się wieloma szczegółami dotyczącymi tego zadania. Jego wykonanie może zakończyć się od razu lub zająć pewną ilość czasu.

Musimy jedynie wiedzieć, kiedy funkcja `foo(..)` zakończy działanie, co pozwoli nam na przejście do kolejnego zadania. Innymi słowy, potrzebny jest mechanizm poinformowania o zakończeniu wykonywania funkcji `foo(..)`, który pozwoli na kontynuację działania programu.

W sposób typowy dla JavaScript sprawa przedstawia się następująco — jeżeli zachodzi potrzeba nasłuchiwania powiadomień, to prawdopodobnie przychodzą Ci na myśl zdarzenia. Dlatego naszą potrzebę można wyrazić jako potrzebę nasłuchiwania *zdarzenia ukończenia* (lub inaczej *zdarzenia kontynuacji*) emitowanego przez `foo(..)`.

Nazwa wspomnianego zdarzenia (ukończenia lub kontynuacji) zależy od Twojej perspektywy. Czy koncentrujesz się bardziej na operacjach wykonywanych za pomocą funkcji `foo(..)`, czy jednak na tym, co będzie po zakończeniu działania wymienionej funkcji? Obie perspektywy są poprawne i użyteczne. Zdarzenie powiadomienia informuje nas, że wykonywanie funkcji `foo(..)` zostało zakończone i można przejść do kolejnego kroku. Istotnie, wywołanie zwrotne przekazane do wywołania przez zdarzenia powiadomienia samo w sobie jest tym, co wcześniej określiliśmy mianem kontynuacji. Zdarzenie ukończenia jest nieco bardziej skoncentrowane na funkcji `foo(..)`, która aktualnie nas bardzo interesuje. Dlatego też w pozostałej części rozdziału będę używał nazwy *zdarzenie ukończenia*.

W podejściu opartym na wywołaniach zwrotnych powiadomieniem będzie wywołanie zwrotne uruchomione przez zadanie (funkcja `foo(..)`). Z kolei w przypadku obietnic następuje odwrócenie sytuacji i to funkcja `foo(..)` nasłuchuje zdarzenia, po którego otrzymaniu odpowiednio kontynuuje wykonywanie kodu.

Na początek spójrz na przedstawiony poniżej pseudokod:

```
foo(x) {
    // Rozpoczęcie zadania, którego wykonanie może chwilę zabrać.
}

foo( 42 )

on (foo "completion") {
    // Teraz można przejść do kolejnego kroku!
}

on (foo "error") {
    // Ups! Coś poszło nie tak w funkcji foo(..).
}
```

Wywołujemy funkcję `foo(..)`, a następnie definiujemy dwie procedury nasłuchiwania zdarzeń, po jednej dla zdarzeń `completion` i `error`. Te zdarzenia przedstawiają dwa możliwe sposoby zakończenia działania funkcji `foo(..)`. Ogólnie rzecz ujmując, funkcja `foo(..)` nawet nie „wie", że wywołujący ją kod nasłuchuje wymienionych zdarzeń, co stanowi doskonały przykład *separacji zadań*.

Niestety, przedstawione podejście wymaga zastosowania pewnej magii środowiska JavaScript, która nie istnieje (i prawdopodobnie byłaby nieco niepraktyczna). Poniżej przedstawiłem znacznie bardziej naturalny sposób wyrażenia w kodzie JavaScript podejścia opartego na obietnicach:

```
function foo(x) {
    // Rozpoczęcie zadania, którego wykonanie może chwilę potrwać.

    // Komponent nasłuchujący zdarzeń powinien
    // mieć możliwość odpowiedniego zareagowania.

    return listener;
}
var evt = foo( 42 );

evt.on( "completion", function(){
    // Teraz można przejść do kolejnego kroku!
} );

evt.on( "failure", function(err){
    // Ups! Coś poszło nie tak w funkcji foo(..).
} );
```

Funkcja `foo(..)` zapewnia wyraźną możliwość dostarczania danych z mechanizmu subskrypcji zdarzeń, a kod wywołujący otrzymuje i rejestruje dwie procedury obsługi zdarzeń.

Odwrotne działanie w porównaniu z tradycyjnym kodem bazującym na wywołaniach zwrotnych powinno być oczywiste i jest pożądane. Zamiast przekazywać wywołanie zwrotne funkcji `foo(..)`, mamy obiekt `evt` otrzymujący wywołania zwrotne.

Jak sobie przypominasz z rozdziału 2., wywołania zwrotne same w sobie przedstawiają odwrócenie kontroli. Dlatego też odwrócenie wzorca wywołań zwrotnych można w rzeczywistości uznać za „odwrócenie odwrócenia", czyli inaczej *brak odwrócenia kontroli* — przywrócenie kontroli z powrotem do pierwotnego kodu. Mamy więc rozwiązanie, którego oczekiwaliśmy od samego początku.

Jedną z ważniejszych korzyści jest to, że wiele oddzielnych fragmentów kodu otrzymuje możliwość nasłuchiwania zdarzeń. Wszystkie nasłuchujące komponenty będą niezależnie od siebie poinformowane o zakończeniu działania funkcji `foo(..)`, co pozwoli im na wykonanie kolejnych kroków:

```
var evt = foo( 42 );

// Funkcja bar(..) nasłuchuje zdarzenia rozgłaszanego przez foo(..).
bar( evt );

// Funkcja baz(..) również nasłuchuje zdarzenia rozgłaszanego przez foo(..).
baz( evt );
```

Brak odwrócenia kontroli pozwala na zastosowanie eleganckiego podziału zadań, a funkcje `bar(..)` i `baz(..)` nie muszą być angażowane w sposób wywołania `foo(..)`. Podobnie funkcja `foo(..)` nie musi nic wiedzieć o istnieniu `bar(..)` i `baz(..)` lub o oczekiwaniu przez inny komponent na powiadomienie o zakończeniu działania `foo(..)`.

W zasadzie obiekt `evt` jest neutralnym łącznikiem zewnętrznym między oddzielnymi zadaniami.

„Zdarzenia" obietnicy

Jak pewnie się domyśliłeś, możliwość nasłuchiwania zdarzeń przez obiekt `evt` stanowi analogią dla obietnicy.

W przypadku podejścia opartego na obietnicy przedstawiony wcześniej fragment kodu utworzy `foo(..)` i zwróci egzemplarz obietnicy `Promise`, która następnie zostanie przekazana do `bar(..)` i `baz(..)`.

Nasłuchiwane „zdarzenia" rozwiązania obietnicy tak naprawdę nie są zdarzeniami (choć zachowują się właśnie jak zdarzenia) i zwykle nie są nazywane `completion` lub `error`. Zamiast tego używamy `then(..)` do zarejestrowania zdarzenia `then`. Ujmując rzecz precyzyjniej, `then(..)` rejestruje zdarzenia `fulfillment` i (lub) `rejection`, mimo że wymienione słowa nie są wyraźnie stosowane w kodzie.

Spójrz na przedstawiony poniżej fragment kodu:

```
function foo(x) {
    // Rozpoczęcie zadania, którego wykonanie może chwilę potrwać.

    // Przygotowanie i zwrot obietnicy.
    return new Promise( function(resolve,reject){
```

```
        // Ostatecznie następuje wywołanie resolve(..) lub reject(..),
        // które to funkcje są wywołaniami zwrotnymi
        // rozwiązania obietnicy.
    } );
}

var p = foo( 42 );

bar( p );

baz( p );
```

Powyższy wzorzec wraz z wywołaniem `new Promise( function(..){ .. } )` jest określany mianem *konstruktora z ujawnieniem* (ang. *revealing constructor*). Przekazywana funkcja jest wykonywana natychmiast (nie ma asynchronicznego odroczenia jej uruchomienia, jak ma to miejsce w przypadku wywołań zwrotnych do `then(..)`) i otrzymuje dwa parametry. W omawianym przypadku to `resolve` i `reject`. To są funkcje rozwiązania obietnicy. Funkcja `resolve(..)` wskazuje na spełnienie obietnicy, natomiast `reject(..)` na jej odrzucenie.

Prawdopodobnie domyśliłeś się, jak może przedstawiać się kod wewnątrz funkcji `bar(..)` i `baz(..)`:

```
function bar(fooPromise) {
    // Nasłuchiwanie ukończenia foo(..).
    fooPromise.then(
        function(){
            // Działanie funkcji foo(..) zostało zakończone,
            // więc funkcja bar(..) może wykonać swoje zadanie.
        },
        function(){
            // Ups! Coś poszło nie tak w funkcji foo(..).
        }
    );
}
// To samo dotyczy funkcji baz(..).
```

Rozwiązanie obietnicy niekoniecznie wymaga wysłania komunikatu, jak miało to miejsce podczas analizy obietnic jako przyszłych wartości. Może to być po prostu sygnał kontroli przepływu działania programu; takie właśnie podejście zastosowałem we wcześniejszym fragmencie kodu.

Inne rozwiązanie przedstawia się następująco:

```
function bar() {
    // Działanie funkcji foo(..) zostało zakończone,
    // więc funkcja bar(..) może wykonać swoje zadanie.
}

function oopsBar() {
    // Ups! Ponieważ coś poszło nie tak w funkcji foo(..),
    // więc nie doszło do uruchomienia funkcji bar(..).
}
// To samo dotyczy funkcji baz() i oopsBaz().


var p = foo( 42 );

p.then( bar, oopsBar );

p.then( baz, oopsBaz );
```

Jeżeli już wcześniej spotkałeś się z kodem opartym na obietnicach, to być może sądzisz, że dwa ostatnie wiersze powyższego kodu można zapisać w postaci `p.then( .. ).then( .. )`, czyli z zastosowaniem łączenia zamiast po prostu jako `p.then(..); p.then(..)`. Jednak taka zmiana spowodowałaby zdefiniowanie zupełnie innego zachowania, więc bądź szczególnie ostrożny! Różnica może nie być od razu widoczna, ale tak naprawdę mamy tutaj do czynienia z całkiem innym wzorcem asynchronicznym, z którym się dotąd nie spotkaliśmy: *splitting* (podział) kontra *forking* (rozwidlenie). Nie przejmuj się! Do tego zagadnienia jeszcze powrócimy w rozdziale.

Zamiast przekazać obietnicę `p` do funkcji `bar(..)` i `baz(..)`, wykorzystujemy obietnicę do kontrolowania, kiedy wymienione funkcje mają zostać wykonane, o ile w ogóle. Podstawowa różnica sprowadza się do procedury obsługi błędów.

W podejściu przedstawionym w pierwszym fragmencie kodu funkcja `bar(..)` jest wywoływana niezależnie od wyniku działania funkcji `foo(..)` — sukces bądź niepowodzenie. Tutaj funkcja `bar(..)` ma własną logikę do wykonania, gdy zostanie poinformowana o zakończonym niepowodzeniem uruchomieniu `foo(..)`. Oczywiście to samo dotyczy również funkcji `baz(..)`.

W drugim fragmencie kodu funkcja `bar(..)` jest wywoływana tylko wtedy, gdy działanie `foo(..)` zakończy się powodzeniem. W przeciwnym razie nastąpi wywołanie `oopsBar(..)`. To samo dotyczy także funkcji `baz(..)`.

Tak naprawdę żadnego z wymienionych powyżej podejść nie można uznać za jedyne poprawne. Zdarzają się różne sytuacje, w których preferowane będzie raz jedno, a raz drugie.

Niezależnie od sytuacji obietnica `p` pochodząca z funkcji `foo(..)` zostanie użyta do określenia kolejnych podejmowanych działań.

Co więcej, ponieważ oba fragmenty kodu kończą się dwukrotnym wywołaniem `then(..)` względem tej samej obietnicy `p`, otrzymujemy potwierdzenie dla wcześniejszego stwierdzenia, że obietnica (po rozwiązaniu) na zawsze zachowuje konkretne rozwiązanie (spełnienie lub odrzucenie) i tym samym może być obserwowana dowolną liczbę razy.

Po rozwiązaniu obietnicy `p` kolejny krok zawsze będzie taki sam, zarówno dla *teraz*, jak i *później*.

Określanie typu na podstawie then()

W świecie obietnic mamy jeden niezwykle ważny szczegół. Skąd można wiedzieć, czy pewna wartość jest autentyczną obietnicą, czy nią nie jest? Ujmując rzecz jeszcze bardziej bezpośrednio: czy dana wartość będzie zachowywała się jak obietnica?

Biorąc pod uwagę fakt konstruowania obietnic za pomocą składni `new Promise(..)`, możesz uznać, że polecenie `p instanceof Promise` jest wystarczającym sprawdzeniem. Niestety, istnieje wiele powodów, dla których tak nie jest.

Przede wszystkim wartość obietnicy możesz otrzymać z innego okna przeglądarki internetowej (element `iframe` itd.), które będzie miało własną obietnicę inną niż znajdująca się w bieżącym oknie lub ramce. W takim przypadku identyfikacja egzemplarza obietnicy zakończy się niepowodzeniem.

Co więcej, biblioteka lub framework mogą stosować własne obietnice zamiast oferowanej przez specyfikację ES6 natywnej implementacji (`Promise`). Tak naprawdę w starszych wersjach przeglądarek internetowych pozbawionych implementacji `Promise` użycie bibliotek przeznaczonych do obsługi obietnic może sprawdzać się doskonale.

Kiedy w dalszej części rozdziału będziemy analizować proces rozwiązywania obietnicy, oczywiste stanie się, dlaczego ogromne znaczenie ma możliwość sprawdzania i prawidłowego rozpoznawania wartości niebędących autentycznymi obietnicami. Teraz musisz uwierzyć mi na słowo, że to ma znaczenie krytyczne dla funkcjonowania całości.

Dlatego też sposobem pozwalającym na rozpoznanie obietnicy (lub komponentu o działaniu przypominającym obietnicę) byłoby zdefiniowanie tak zwanego *thenable*, czyli obiektu zawierającego metodę o nazwie `then(..)`. Przyjmuje się założenie, że wszelka tego rodzaju wartość jest zgodna z obietnicą.

Ogólne pojęcie dla *sprawdzania typu* po przyjęciu założenia o wartości *typu* w oparciu o istnienie pewnych właściwości to tak zwane *kacze typowanie* (ang. *duck typing*). „Jeżeli coś wygląda jak kaczka i wydaje dźwięki takie jak kaczka to musi być kaczką" — patrz inna książka z tej serii, zatytułowana *Typy i składnia*. Poniżej przedstawiłem przykład kaczego typowania pozwalającego na sprawdzenie, czy obiekt można uznać za *thenable*, czyli zawierający metodę `then(..)`:

```
if (
    p !== null &&
    (
        typeof p === "object" ||
        typeof p === "function"
    ) &&
    typeof p.then === "function"
) {
    // Przyjmujemy założenie, że obiekt jest thenable!
}
else {
    // Obiekt nie zawiera metody then(..).
}
```

Fuj! Pomijając fakt, że przedstawiona powyżej brzydka logika musi być zaimplementowana w różnych miejscach, zdefiniowany tutaj kod zawiera jeszcze o wiele poważniejsze błędy.

Jeżeli spróbujesz spełnić obietnicę za pomocą dowolnej wartości obiektu (lub funkcji) zawierającego wewnątrz funkcję `then(..)`, ale nie chcesz tego obiektu traktować jako obietnicy lub *thenable*, to powstanie problem. Wspomniany obiekt będzie uznany za *thenable*, a tym samym potraktowany według specjalnych reguł (przedstawię je w dalszej części rozdziału).

Z wymienioną sytuacją mamy do czynienia nawet wtedy, kiedy nie wiemy, że dany element zawiera funkcję `then(..)`. Spójrz na poniższy fragment kodu:

```
var o = { then: function(){} };

// Poniższe polecenie powoduje, że 'v' jest połączone z 'o' w stylu '[[Prototype]]'.
var v = Object.create( o );

v.someStuff = "świetnie";
v.otherStuff = "nie tak świetnie";

v.hasOwnProperty( "then" );      // Fałsz.
```

Element `v` w ogóle nie wygląda jak obietnica. To jest po prostu zwykły obiekt wraz z pewnymi właściwościami. Prawdopodobnie planujesz przekazywać wartość `v` tak jak każdy inny obiekt.

Jednak nie zdajesz sobie sprawy, że obiekt `v` jest również połączony (`[[Prototype]]` — patrz inna książka z tej serii, zatytułowana *Wskaźnik this i prototypy obiektów*) z innym obiektem `o`, który ma zdefiniowaną metodę `then(..)`. Dlatego też w trakcie operacji kaczego typowania następuje przyjęcie założenia, że obiekt `v` jest *thenable*. O nie!

Wspomniane połączenie nawet nie musi być utworzone celowo, jak przedstawiłem poniżej:

```
Object.prototype.then = function(){};
Array.prototype.then = function(){};

var v1 = { hello: "świecie" };
var v2 = [ "Hello", "Świecie" ];
```

W powyższym fragmencie kodu obiekty `v1` i `v2` są *thenable*. Nie możesz kontrolować lub przewidzieć, czy jakikolwiek inny kod przypadkowo bądź złośliwie nie wstawi metody `then(..)` do `Object.prototype`, `Array.prototype` lub innego natywnego prototypu. Ponadto jeśli wskazany element jest funkcją niewywołującą żadnych parametrów jako wywołań zwrotnych, to każda obietnica rozwiązania za pomocą tego rodzaju wartości będzie w sposób niezauważony zawieszona na zawsze! Istne szaleństwo.

To brzmi mało prawdopodobnie lub wręcz nieprawdopodobnie? Być może.

Musisz jednak pamiętać, że jeszcze przed wydaniem ES6 istniało wiele doskonale znanych bibliotek niebazujących na obietnicach, ale zawierających metody o nazwie `then(..)`. W przypadku części ze wspomnianych bibliotek ich twórcy zdecydowali się na zmianę nazw metod, aby tym samym uniknąć kolizji nazw (to jest rozwiązanie do niczego!). Natomiast w przypadku innych autorzy po prostu ograniczyli się do komunikatu informującego o niezgodności biblioteki z kodem opartym na obietnicach, ponieważ nie potrafili sobie poradzić inaczej.

Podjęta przez twórców standardu ES6 decyzja o przyjęciu nazwy wcześniej niezarezerwowanej — i dotyczącej całkowicie ogólnej koncepcji — prawdopodobnie prowadzi do powstawania błędów, których wykrycie jest trudne. Przecież nazwa właściwości `then` oznacza, że żadna wartość (lub jakikolwiek jej delegat) w przeszłości, teraźniejszości i przyszłości nie może mieć funkcji o nazwie `then(..)`, zarówno nazwanej tak przypadkowo, jak i celowo. W przeciwnym razie wspomniana wartość będzie w systemach bazujących na obietnicach uznana za *thenable*.

Nie podoba mi się przyjęte rozwiązanie polegające na użyciu kaczego typowania do rozpoznawania obietnic. Można było zastosować inne rozwiązania, na przykład „branding” lub nawet „antybranding”, które wydają się być kompromisowe. Jednak to nie jest jeszcze powód do smutku i zniechęcenia. Jak się przekonasz w dalszej części tekstu, kacze typowanie może okazać się użyteczne. Nie zapominaj tylko, że jednocześnie będzie niebezpieczne, jeśli element niebędący obietnicą zostanie błędnie za nią uznany.

Kwestie zaufania i obietnice

Wcześniej przedstawiłem dwie solidne analogie wyjaśniające różne aspekty użycia obietnic w kodzie działającym asynchronicznie. Jednak jeśli na tym poprzestaniemy, to pominiemy prawdopodobnie najważniejszą cechę charakterystyczną wzorca bazującego na obietnicach, czyli zaufanie.

Podczas gdy analogie przyszłej wartości i zdarzenia ukończenia odgrywały wyraźną rolę we wzorcach kodu przeanalizowanych w poprzednim rozdziale, to nie całkiem pozostaje jasne, dlaczego lub jak obietnice zostały zaprojektowane do rozwiązania przedstawionych w rozdziale 2. problemów związanych z odwróceniem kontroli. Jeżeli zaczniesz dokładniej analizować koncepcję, to odkryjesz pewne istotne możliwości pozwalające na przywrócenie zaufania w kodzie działającym asynchronicznie.

Zaczynamy od przypomnienia problemów z zaufaniem, które występują w kodzie bazującym jedynie na wywołaniach zwrotnych. Po przekazaniu wywołania zwrotnego do narzędzia `foo(..)` możemy się spodziewać:

- zbyt wczesnego uruchomienia wywołania zwrotnego;
- zbyt późnego (lub nawet braku) uruchomienia wywołania zwrotnego;
- uruchomienia wywołania zwrotnego zbyt małą lub zbyt dużą liczbę razy;
- niepowodzenia podczas przekazywania wszelkich niezbędnych zmiennych środowiskowych lub parametrów;
- ukrycia wszelkich błędów lub wyjątków, które mogą zostać zgłoszone.

Obietnica ma cechy charakterystyczne, które celowo zostały jej nadane, aby zapewnić użyteczne, powtarzalne rozwiązania wszystkich wymienionych powyżej problemów.

Zbyt wczesne wywołanie

W przypadku zbyt wczesnego wywołania w kodzie może powstać efekt Zalgo (patrz rozdział 2.), gdy raz zadanie jest kończone synchronicznie, a innym razem asynchronicznie, co może prowadzić do wystąpienia stanu wyścigu.

Obietnica z definicji nie jest podatna na omawiany problem, ponieważ zdarzenie natychmiast spełnionej obietnicy (na przykład `new Promise(function(resolve){ resolve(42); })`) nie może być synchroniczne.

Dlatego też po wywołaniu metody `then(..)` w obietnicy, nawet jeśli została rozwiązana, to wywołania zwrotne dostarczane do `then(..)` zawsze będą uruchamiane asynchronicznie (więcej informacji na ten temat przedstawiłem w rozdziale 1., w podrozdziale „Zadania”).

Nie trzeba dłużej stosować sztuczek typu wywołania `setTimeout(..,0)`. Obietnica automatycznie uniemożliwia powstanie efektu Zalgo.

Zbyt późne wywołanie

Podobnie jak w poprzednim punkcie, wywołania zwrotne zarejestrowane w metodzie `then(..)` obietnicy są automatycznie uruchamiane po wywołaniu `resolve(..)` lub `reject(..)` przez obietnicę. Wspomniane wywołania zwrotne będą w sposób przewidywalny uruchamiane w kolejnym momencie asynchronicznym (patrz podrozdział „Zadania" w rozdziale 1.).

Skoro nie ma możliwości synchronicznego wykonania wywołania zwrotnego, nie istnieje też niebezpieczeństwo, że synchroniczny łańcuch zadań będzie wykonany w sposób, który w efekcie opóźnia uruchomienie kolejnych wywołań zwrotnych. Oznacza to, że podczas rozwiązywania obietnicy wszystkie zarejestrowane wywołania zwrotne `then(..)` będą kolejno wykonane, natychmiast w trakcie następnej asynchronicznej chwili (ponownie patrz podrozdział „Zadania" w rozdziale 1.). W poszczególnych wywołaniach zwrotnych nic nie opóźni ani nie będzie miało wpływu na uruchomienie kolejnych wywołań zwrotnych.

Spójrz na poniższy fragment kodu:

```
p.then( function(){
    p.then( function(){
        console.log( "C" );
    } );
    console.log( "A" );
} );
p.then( function(){
    console.log( "B" );
} );
// A B C
```

W powyższym kodzie `C` nie może zakłócić lub poprzedzić wykonania `B`, co wynika ze sposobu definiowania i działania obietnic.

Problemy związane z tworzeniem harmonogramu obietnic

Trzeba zwrócić uwagę na istnienie wielu niuansów podczas tworzenia harmonogramu obietnic, gdy względna kolejność między wywołaniami zwrotnymi powoduje, że dwie oddzielne obietnice nie będą niezawodnie przewidywalne.

Jeżeli dwie obietnice `p1` i `p2` zostały rozwiązane, to powinno być oczywiste, że polecenia `p1.then(..);` i `p2.then(..);` spowodują uruchomienie wywołań zwrotnych w następującej kolejności: najpierw dla `p1`, później dla `p2`. Istnieją jednak pewne sytuacje, w których może stać się inaczej, na przykład:

```
var p3 = new Promise( function(resolve,reject){
    resolve( "B" );
} );

var p1 = new Promise( function(resolve,reject){
    resolve( p3 );
} );

p2 = new Promise( function(resolve,reject){
    resolve( "A" );
} );

p1.then( function(v){
    console.log( v );
} );
```

```
p2.then( function(v){
    console.log( v );
} );

// Kolejność: A B  <-- nie B A, jak można było oczekiwać.
```

Do tego zagadnienia jeszcze powrócimy w dalszej części rozdziału. Jak możesz zobaczyć, obietnica `p1` została rozwiązana nie za pomocą natychmiast dostępnej wartości, ale inną obietnicą `p3`, której rozwiązaniem jest wartość `B`. Przedstawione rozwiązanie to rozpakowanie `p3` na `p1`, ale w sposób asynchroniczny. Dlatego też wywołania zwrotne `p1` są uruchamiane *po* wywołaniach zwrotnych `p2` w asynchronicznej kolejce zadań (patrz podrozdział „Zadania” w rozdziale 1.).

Aby uniknąć tego rodzaju koszmarów, nigdy nie powinieneś polegać na niczym, co dotyczy kolejności lub harmonogramu wywołań zwrotnych w różnych obietnicach. Tak naprawdę dobrą praktyką jest uniknięcie tworzenia kodu w sposób, w którym kolejność wielu wywołań zwrotnych w ogóle ma znaczenie. Unikaj takiego podejścia, gdy tylko możesz.

Brak uruchomienia wywołania zwrotnego

Ta obawa pojawia się niezwykle często. Za pomocą obietnic można ją rozwiać na wiele różnych sposobów.

Przede wszystkim nic (nawet błąd JavaScript) nie może uniemożliwić obietnicy wygenerowania powiadomienia o jej rozwiązaniu (o ile oczywiście zostanie rozwiązana). Jeżeli dla obietnicy zarejestrujesz wywołania zwrotne uruchamiane po jej spełnieniu lub odrzuceniu, a obietnica zostanie rozwiązana, to jedno z dwóch wspomnianych wywołań zwrotnych zawsze będzie uruchomione.

Oczywiście gdy sam kod wywołań zwrotnych zawiera błędy JavaScript, możesz otrzymać wyniki inne od oczekiwanych, ale to nie zmienia faktu, że wywołania zwrotne zostaną uruchomione. W dalszej części książki dowiesz się, jak otrzymać powiadomienie o błędzie w wywołaniu zwrotnym, ponieważ tego rodzaju błędy nie są ukrywane.

Co się stanie w sytuacji, gdy obietnica nie zostanie rozwiązana w przedstawiony powyżej sposób? Nawet taka sytuacja została przewidziana przez twórców obietnic, a odpowiedzią jest wyższy poziom abstrakcji o nazwie *wyścig*:

```
// Funkcja pomocnicza przeznaczona do wyczerpania limitu czasu przeznaczonego dla obietnicy.
function timeoutPromise(delay) {
    return new Promise( function(resolve,reject){
        setTimeout( function(){
            reject( "Czas minął!" );
        }, delay );
    } );
}


// Konfiguracja wyczerpania limitu czasu dla funkcji foo().
Promise.race( [
    foo(),                  // Próba wywołania funkcji foo().
    timeoutPromise( 3000 )  // Dostępny czas to 3 sekundy.
] )
.then(
    function(){
        // Funkcja foo(..) została wywołana w podanym czasie!
    },
```

```
        function(err){
            // Funkcja foo() została odrzucona lub po prostu nie była
            // uruchomiona w przeznaczonym na to czasie, więc za pomocą
            // 'err' podajemy, która z wymienionych sytuacji wystąpiła.
        }
    );
```

W powyższej sytuacji mamy znacznie więcej szczegółów dotyczących przekroczenia limitu czasu w obietnicy, ale do tego zagadnienia jeszcze później wrócimy.

Co ważniejsze, możemy być pewni otrzymania sygnału wygenerowanego przez `foo()` w celu uniknięcia zawieszenia programu w nieskończoność.

Uruchomienie wywołania zwrotnego zbyt małą lub zbyt dużą liczbę razy

Z definicji *jeden* to odpowiednia liczba uruchomień wywołania zwrotnego. Dlatego też „zbyt mało" oznacza tutaj zero wywołań zwrotnych, co właściwie odpowiada przeanalizowanej przed chwilą sytuacji „braku" wywołań zwrotnych.

Przypadek „zbyt wiele" jest łatwy do wyjaśnienia. Obietnice są definiowane tak, aby mogły być rozwiązane tylko jeden raz. Jeżeli z jakiegokolwiek powodu obietnica tworzy kod próbujący wielokrotnie wywołać funkcje `resolve(..)` lub `reject(..)` bądź obie, to obietnica zaakceptuje tylko pierwsze rozwiązanie i po prostu zignoruje pozostałe.

Skoro obietnica może być rozwiązana tylko jeden raz, to każde z zarejestrowanych wywołań zwrotnych `then(..)` również zostanie uruchomione tylko jednokrotnie.

Oczywiście, jeżeli to samo wywołanie zwrotne zarejestrujesz więcej niż tylko jeden raz, na przykład `p.then(f); p.then(f);`, to liczba uruchomień będzie odpowiadała liczbie rejestracji danego wywołania zwrotnego. Gwarancja jednokrotnego wywołania funkcji odpowiedzi nie chroni Cię przed sytuacją, w której sam strzelasz sobie w stopę.

Niepowodzenie podczas przekazywania wszelkich niezbędnych zmiennych środowiskowych lub parametrów

Obietnica może mieć co najwyżej jedną wartość rozwiązania (spełnienie lub odrzucenie).

Jeżeli obietnica nie zostanie wyraźnie rozwiązana jedną z podanych wyżej wartości, to wartością będzie `undefinied`, która jest typowa dla kodu utworzonego w JavaScript. Niezależnie od wartości będzie ona zawsze przekazywana do wszystkich zarejestrowanych wywołań zwrotnych (zapewniających obsługę spełnienia i odrzucenia), zarówno teraz, jak i w przyszłości.

Musisz koniecznie pamiętać o jednej kwestii. Jeżeli wywołasz `resolve(..)` lub `reject(..)` z wieloma parametrami, to wszystkie kolejne parametry poza pierwszym zostaną po cichu zignorowane. Wprawdzie może się to wydawać złamaniem wspomnianej wcześniej gwarancji, ale niekoniecznie tak jest, ponieważ większa liczba parametrów oznacza nieprawidłowe użycie mechanizmu obietnicy. Istnieją także mechanizmy ochrony przed innymi nieprawidłowymi sposobami użycia API, na przykład przed wielokrotnym wywołaniem `resolve(..)`. Zachowanie obietnicy można więc uznać za spójne (choć nieco frustrujące).

Jeżeli chcesz przekazać wiele wartości, to musisz je opakować inną pojedynczą wartością, którą następnie przekażesz. Przykładem może być tablica (`array`) lub obiekt (`object`).

W przypadku środowiska funkcje w JavaScript zawsze zawierają się w zakresie, w którym zostały zdefiniowane (patrz inna książka z tej serii, zatytułowana *Zakresy i domknięcia*), a więc nadal będą miały dostęp do dostarczonego przez Ciebie elementu. Oczywiście to samo dotyczy również projektu bazującego tylko na wywołaniach zwrotnych, więc to nie jest korzyść specyficzna jedynie dla obietnic, choć jednocześnie jest to gwarancja, na której można polegać.

Ukrycie wszelkich błędów lub wyjątków, które mogą zostać zgłoszone

W zasadzie jest to powtórzenie przeanalizowanego poprzednio przypadku. Jeżeli obietnica zostanie odrzucona wraz z *powodem* (inaczej z komunikatem o błędzie), to ta właśnie wartość zostanie przekazana do wywołania zwrotnego obsługującego odrzucenie obietnicy.

Jednak tutaj mamy kwestię odgrywającą znacznie większą rolę. Jeżeli w dowolnym punkcie tworzenia obietnicy lub obserwacji jej rozwiązania nastąpi zgłoszenie błędu bądź wyjątku JavaScript, takiego jak `TypeError` lub `ReferenceError`, taki wyjątek zostanie przechwycony, a dana obietnica będzie zmuszona do odrzucenia.

Spójrz na poniższy fragment kodu:

```
var p = new Promise( function(resolve,reject){
    foo.bar();  // Brak definicji 'foo', a więc mamy błąd!
    resolve( 42 );  // Nigdy nie dotrzemy do tego wiersza :-(
} );

p.then(
    function fulfilled(){
        // Nigdy nie dotrzemy do tego wiersza :-(
    },
    function rejected(err){
        // Wartością 'err' będzie obiekt wyjątku TypeError,
        // zgłoszonego w wierszu zawierającym wywołanie foo.bar().
    }
);
```

Wyjątek JavaScript zgłoszony w wierszu zawierającym wywołanie `foo.bar()` spowoduje odrzucenie obietnicy. Ten wyjątek można przechwycić i obsłużyć.

Mamy tutaj do czynienia z niezwykle ważnym szczegółem, ponieważ w efekcie eliminujemy inną potencjalną sytuację, w której może pojawić się Zalgo. Wspomniane błędy mogą spowodować powstanie reakcji synchronicznej, podczas gdy brak błędów będzie asynchroniczny. Obietnice zmieniają zachowanie wyjątków JavaScript na asynchroniczne, a tym samym znacznie zmniejszają niebezpieczeństwo wystąpienia stanu wyścigu.

Co się stanie w sytuacji, gdy obietnica będzie spełniona, ale zgłoszenie wyjątku JavaScript nastąpi na etapie obserwacji, to znaczy w zarejestrowanym wywołaniu zwrotnym `then(..)`? Nawet wówczas błędy nie zostaną utracone, ale sposób ich obsługi może być pewnym zaskoczeniem, dopóki nie zaczniesz analizować dokładniej całej sytuacji:

```
var p = new Promise( function(resolve,reject){
    resolve( 42 );
} );
```

```
p.then(
    function fulfilled(msg){
        foo.bar();
        console.log( msg ); // Nigdy nie dotrzemy do tego wiersza :-(
    },
    function rejected(err){
        // Nigdy nie dotrzemy również do tego wiersza :-(
    }
);
```

Zaczekaj, wydaje się, że w przedstawionym powyżej fragmencie kodu wyjątki zgłaszane przez wywołanie `foo.bar()` pozostaną ukryte. Nie musisz się obawiać, nie będą ukryte. Mamy jednak nieco poważniejszy błąd, jakim jest brak możliwości nasłuchiwania zgłoszenia wspomnianych wyjątków. Wywołanie zwrotne `p.then(..)` zwraca inną obietnicę, *która* z kolei zostanie odrzucona wraz z wyjątkiem `TypeError`.

Dlaczego nie możemy po prostu wywołać zdefiniowanej tutaj procedury obsługi błędów? Wprawdzie wydaje się, że to logiczne rozwiązanie, ale jednocześnie stanowi złamanie podstawowej reguły, jaką jest brak możliwości modyfikacji obietnicy po jej rozwiązaniu. Obietnica `p` została już spełniona i ma wartość `42`. Nie może więc zostać zmieniona na odrzuconą, ponieważ wystąpił błąd podczas obserwacji rozwiązania tej obietnicy.

Poza złamaniem jednej z podstawowych reguł obietnicy wspomniane rozwiązanie mogłoby okazać się dewastujące, gdyby dla obietnicy `p` istniało wiele zarezerwowanych wywołań zwrotnych `then(..)`. W takim przypadku część wywołań byłaby uruchomiona, a inne nie i bardzo trudno byłoby ustalić, dlaczego tak się dzieje.

Obietnice, którym można ufać?

Ostatnim szczegółem pozostałym do przeanalizowania jest kwestia zaufania w trakcie stosowania wzorca obietnicy.

Bez wątpienia zauważyłeś, że stosowanie obietnic nie oznacza całkowitego pozbycia się wywołań zwrotnych. Obietnica po prostu zmienia miejsce, do którego przekazywane jest wywołanie zwrotne. W omawianym przypadku zamiast przekazywać wywołanie zwrotne do `foo(..)`, z wymienionej funkcji otrzymujemy coś (pozornie autentyczną obietnicę) i przekazujemy wywołanie zwrotne w inne miejsce.

Mógłbyś w tym miejscu zapytać, dlaczego takie rozwiązanie ma wzbudzać większe zaufanie niż oparte na jedynie wywołaniach zwrotnych. Skąd mamy pewność, że otrzymujemy godną zaufania obietnicę? Czy to nie jest po prostu domek z kart, a my możemy zaufać tylko dlatego, że nam też się ufa?

Jednym z najważniejszych szczegółów dotyczących obietnic i jednocześnie najczęściej przeoczanym jest fakt, że obietnice zawierają także rozwiązanie kwestii zaufania. Znajduje się ono w natywnej implementacji obietnicy ES6, a dokładnie w wywołaniu `Promise.resolve(..)`.

Jeżeli funkcji `Promise.resolve(..)` przekażesz wartość natychmiastową, inną niż obietnica i niebędącą *thenable*, to dana obietnica będzie spełniona za pomocą przekazanej wartości. W przedstawionym poniżej fragmencie kodu obietnice `p1` i `p2` zachowują się w identyczny sposób:

```
var p1 = new Promise( function(resolve,reject){
    resolve( 42 );
```

```
} );

var p2 = Promise.resolve( 42 );
```

Jeżeli funkcji `Promise.resolve(..)` przekażesz autentyczną obietnicę, to z powrotem otrzymasz tę samą obietnicę:

```
var p1 = Promise.resolve( 42 );

var p2 = Promise.resolve( p1 );

p1 === p2; // Prawda.
```

Co ważniejsze, jeśli funkcji `Promise.resolve(..)` przekażesz wartość *thenable* niebędącą obietnicą, to nastąpi próba rozpakowania tej wartości. Proces rozpakowania będzie kontynuowany aż do wyodrębnienia ostatecznej wartości nieprzypominającej obietnicy.

Czy przypominasz sobie to, co wcześniej powiedzieliśmy o wartościach *thenable*?

Spójrz na poniższy fragment kodu:

```
var p = {
    then: function(cb) {
        cb( 42 );
    }
};

// Przedstawione rozwiązanie działa, jeśli tylko masz dużo szczęścia.
p
.then(
    function fulfilled(val){
        console.log( val ); // 42
    },
    function rejected(err){
        // Nigdy nie dotrzemy do tego wiersza.
    }
);
```

Wprawdzie element `p` jest *thenable*, ale to na pewno nie jest autentyczna obietnica. Na szczęście tutaj to uzasadnione rozwiązanie, ponieważ w większości przypadków `p` będzie obietnicą. Jednak co się stanie, jeśli zastosujemy następujące podejście:

```
var p = {
    then: function(cb,errcb) {
        cb( 42 );
        errcb( "Złowieszczy uśmiech" );
    }
};


p
.then(
    function fulfilled(val){
        console.log( val ); // 42
    },
    function rejected(err){
        // Ups! Ta funkcja nie powinna zostać wywołana.
        console.log( err ); // Złowieszczy uśmiech.
    }
);
```

W powyższym fragmencie kodu element `p` jest *thenable*, choć jednocześnie niezbyt dobrze zachowuje się jako obietnica. Czy to kod o złośliwym działaniu? Czy to jedynie skutek braku wiedzy programisty o sposobie działania obietnic? Szczerze mówiąc, to nie ma żadnego znaczenia. Niezależnie od tego, jakie są powody utworzenia tego kodu, nie można mu ufać, jeśli ma taką postać.

Dowolną z przedstawionych powyżej wersji `p` można przekazać funkcji `Promise.resolve(..)` — wtedy otrzymasz znormalizowany, bezpieczny wynik, którego oczekujesz:

```
Promise.resolve( p )
.then(
    function fulfilled(val){
        console.log( val ); // 42
    },
    function rejected(err){
        // Nigdy nie dotrzemy do tego wiersza.
    }
);
```

Funkcja `Promise.resolve(..)` będzie akceptowała dowolną wartość *thenable*, a następnie rozpakuje ją na jej postać inną niż *thenable*. Jednak wynikiem wywołania `Promise.resolve(..)` jest rzeczywista, autentyczna obietnica, której można ufać. Jeżeli przekazana została autentyczna obietnica, to otrzymasz ją z powrotem, a więc nie ma w ogóle żadnych wad zastosowania filtrowania przez funkcję `Promise.resolve(..)`, aby zyskać zaufanie.

Przyjmujemy założenie o wywołaniu funkcji pomocniczej `foo(..)`. Nie mamy pewności, czy można zaufać wartości zwrotnej, że będzie zachowywała się jak obietnica. Jednak przynajmniej wiemy, że jest *thenable*. Dzięki funkcji `Promise.resolve(..)` otrzymujemy godne zaufania opakowanie obietnicy:

```
// Nie stosuj po prostu poniższego kodu:
foo( 42 )
.then( function(v){
    console.log( v );
} );

// Zamiast powyższego użyj następującego kodu:
Promise.resolve( foo( 42 ) )
.then( function(v){
    console.log( v );
} );
```

Kolejną korzyścią płynącą z opakowania przez `Promise.resolve(..)` wartości zwrotnej (*thenable* lub innej) dowolnej funkcji jest łatwy sposób znormalizowania danego wywołania funkcji na doskonale zachowujące się zadanie asynchroniczne. Jeżeli wywołanie `foo(42)` czasami zwraca natychmiastową wartość, a w innych przypadkach obietnicę, to dzięki wywołaniu `Promise.resolve( foo(42) )` masz pewność, że wynikiem zawsze będzie obietnica. Uniknięcie powstawania efektu Zalgo skutkuje tworzeniem lepszego kodu.

Budowa zaufania

Mam nadzieję, że przedstawione wcześniej informacje w pełni wyjaśniły Ci, dlaczego obietnice są warte zaufania, i co ważniejsze, dlaczego wspomniane zaufanie ma znaczenie krytyczne podczas tworzenia solidnego, niezawodnego i łatwego w konserwacji oprogramowania.

Czy w języku JavaScript możesz utworzyć kod asynchroniczny bez zaufania? Oczywiście, że tak. Programiści JavaScript przez niemal dwie dekady tworzyli kod działający asynchronicznie, mając do dyspozycji jedynie wywołania zwrotne.

Kiedy zaczniesz kwestionować to, na ile można zaufać wykorzystywanym mechanizmom, czy są w rzeczywistości niezawodne i przewidywalne, to zaczniesz zdawać sobie sprawę, jak niewielkim zaufaniem można obdarzyć wywołania zwrotne.

Obietnice to wzór, dzięki któremu wywołania zwrotne otrzymują semantykę godną zaufania. Ich zachowanie staje się bardziej racjonalne i niezawodne. Dzięki uniknięciu odwrócenia kontroli w wywołaniach zwrotnych kontrola pozostaje w systemie godnym zaufania (obietnice), zaprojektowanym specjalnie w celu zapewnienia przejrzystości kodu działającego asynchronicznie.

Łańcuch przepływu kontroli

Wspomniałem już o tym wielokrotnie, ale powtórzę ponownie — obietnica nie stanowi po prostu mechanizmu dla przeprowadzanej w jednym kroku operacji typu „to, a później tamto". Obietnica na pewno jest elementem konstrukcyjnym, ale okazuje się, że mamy możliwość połączenia wielu obietnic i tym samym przedstawienia sekwencji kroków asynchronicznych.

Kluczem, dzięki któremu wspomniane rozwiązanie może działać zgodnie z oczekiwaniami, są dwie cechy nierozłącznie wiążące się z obietnicami:

- W trakcie każdego wywołania `then(..)` w obietnicy następuje utworzenie i zwrot nowej obietnicy, z którą można *połączyć* inną obietnicę.
- Wartość zwrotna wywołania `then(..)` spełniająca wywołanie zwrotne (pierwszy parametr) jest automatycznie ustawiona jako wartość spełniająca połączoną obietnicę (z punktu pierwszego).

Zacznijmy od zilustrowania powyższej koncepcji, a następnie zobaczymy, jak to pomaga w utworzeniu asynchronicznej sekwencji kroków przepływu działania programu. Spójrz na poniższy fragment kodu:

```
var p = Promise.resolve( 21 );

var p2 = p.then( function(v){
    console.log( v );   // 21

    // Spełnienie p2 za pomocą wartości 42.
    return v * 2;
} );


// Połączenie p2.
p2.then( function(v){
    console.log( v );   // 42
} );
```

Przez zwrot obliczonej wartości `v * 2` (na przykład `42`) spełniamy obietnicę `p2`, która została utworzona przez pierwsze wywołanie `then(..)`. Po wywołaniu `then(..)` dla obietnicy `p2` jej spełnieniem będzie wartość wygenerowana przez polecenie `return v * 2`. Oczywiście `p2.then(..)` tworzy jeszcze inną obietnicę, którą moglibyśmy przechowywać w zmiennej `p3`.

Jednak konieczność utworzenia zmiennej przejściowej `p2` (lub `p3` itd.) jest nieco irytująca. Na szczęście bardzo łatwo możemy połączyć obietnice:

```
var p = Promise.resolve( 21 );

p
.then( function(v){
    console.log( v );   // 21

    // Wartość 42 jest spełnieniem połączonej obietnicy.
    return v * 2;
} )
// Tutaj mamy połączoną obietnicę.
.then( function(v){
    console.log( v );   // 42
} );
```

W powyższym fragmencie kodu pierwsze wywołanie `then(..)` to pierwszy kod w sekwencji asynchronicznej, natomiast drugie wywołanie `then(..)` jest drugim krokiem tej sekwencji. Kolejne kroki można dodawać tak długo, dopóki istnieje potrzeba. Połączenie z poprzednim wywołaniem `then(..)` automatycznie powoduje utworzenie obietnicy.

W przedstawionym rozwiązaniu pominęliśmy jeden aspekt. Co się stanie w sytuacji, gdy krok 2. ma zostać wykonany dopiero po zakończeniu asynchronicznego zadania zdefiniowanego w kroku 1.? Używamy polecenia `return` o natychmiastowym działaniu, co powoduje natychmiastowe spełnienie połączonej obietnicy.

Kluczem pozwalającym na zachowanie prawdziwie asynchronicznego działania sekwencji jest sposób funkcjonowania wywołania `Promise.resolve(..)` po przekazaniu mu obietnicy lub wartości *thenable* zamiast wartości końcowej. Wywołanie `Promise.resolve(..)` bezpośrednio zwraca autentyczną obietnicę lub rozpakowuje wartość otrzymaną jako *thenable* — ta operacja jest przeprowadzana rekurencyjnie aż do rozpakowania wszystkich wartości *thenable*.

Tego samego rodzaju rozpakowanie ma miejsce, jeśli wartością zwrotną procedury obsługi obietnicy (spełnienie bądź odrzucenie) jest wartość *thenable* lub obietnica. Spójrz na poniższy fragment kodu:

```
var p = Promise.resolve( 21 );

p.then( function(v){
    console.log( v );   // 21

    // Utworzenie obietnicy i jej zwrot.
    return new Promise( function(resolve,reject){
        // Spełnienie obietnicy za pomocą wartości 42.
        resolve( v * 2 );
    } );
} )
.then( function(v){
    console.log( v );   // 42
} );
```

Wprawdzie wartość `42` opakowaliśmy w zwracaną obietnicę, ale wspomniana wartość nadal będzie rozpakowywana i stanowi rozwiązanie powiązanej obietnicy, a drugie wywołanie `then(..)` otrzymuje wartość `42`. Jeżeli wprowadzimy synchroniczność do tej opakowanej obietnicy, wszystko nadal będzie elegancko działało w taki sam sposób, jak dotąd:

```
var p = Promise.resolve( 21 );

p.then( function(v){
    console.log( v );   // 21

    // Utworzenie obietnicy i jej zwrot.
    return new Promise( function(resolve,reject){
        // Wprowadzenie asynchroniczności!
        setTimeout( function(){
            // Spełnienie obietnicy za pomocą wartości 42.
            resolve( v * 2 );
        }, 100 );
    } );
} )
.then( function(v){
    // Uruchomienie po opóźnieniu 100 ms względem poprzedniego kodu.
    console.log( v );   // 42
} );
```

To jest rozwiązanie o niewiarygodnych możliwościach. Możemy teraz tworzyć sekwencje składające się z dowolnej liczby asynchronicznych kroków, a każdy z nich może być wykonany z opóźnieniem (lub bez!), w zależności od potrzeb.

Oczywiście wartość przekazywana między poszczególnymi krokami w omawianym przykładzie jest opcjonalna. Jeżeli nie zostanie wyraźnie zwrócona pewna wartość, to przyjmuje się, że będzie nią `undefined`, obietnice nadal będą łączone ze sobą w dokładnie ten sam sposób. Rozwiązanie danej obietnicy staje się tym samym sygnałem wskazującym na możliwość przejścia do następnego kroku.

Aby jeszcze dokładniej zilustrować łączenie, zdefiniowanie opóźnienia w wykonaniu kolejnej obietnicy (bez komunikatu o jej rozwiązaniu) przenosimy do funkcji pomocniczej, którą można wielokrotnie wywołać w różnych krokach:

```
function delay(time) {
    return new Promise( function(resolve,reject){
        setTimeout( resolve, time );
    } );
}

delay( 100 ) // Krok 1.
.then( function STEP2(){
    console.log( "Krok 2 (po upływie 100 ms)" );
    return delay( 200 );
} )
.then( function STEP3(){
    console.log( "Krok 3 (po upływie kolejnych 200 ms)" );
} )
.then( function STEP4(){
    console.log( "Krok 4 (następne zadanie)" );
    return delay( 50 );
} )
.then( function STEP5(){
    console.log( "Krok 5 (po upływie kolejnych 50 ms)" );
} )
...
```

Wywołanie `delay(200)` powoduje utworzenie obietnicy, która zostanie spełniona w ciągu 200 ms. Następnie zwracamy ją z pierwszego wywołania `then(..)`, co powoduje, że drugie wywołanie `then(..)` zostaje opóźnione o 200 ms.

Jak wcześniej wspomniałem, pod względem technicznym mamy dwie obietnice — obietnicę z opóźnieniem 200 ms oraz połączoną z nią drugą, która pochodzi z drugiego wywołania `then(..)`. Być może łatwiej zrozumiesz sposób działania rozwiązania, gdy w myślach połączysz obie obietnice, ponieważ mechanizm obietnic automatycznie łączy ich stany. Pod tym względem polecenie `return delay(200)` można potraktować jako utworzenie obietnicy zastępującej zwróconą wcześniej.

Jednak szczerze mówiąc, sekwencja opóźnień bez przekazywania komunikatów to nie jest najlepszy przykład opartej na obietnicach kontroli przepływu działania programu. Przejdziemy teraz do scenariusza, który jest znacznie praktyczniejszy.

Spójrzmy na kod, w którym zamiast liczników czasu wykorzystujemy żądania Ajax:

```
// Przyjmujemy założenie o dostępności funkcji ajax( {url}, {callback} ).

// Żądanie Ajax bazujące na obietnicy.
function request(url) {
    return new Promise( function(resolve,reject){
        // Procedura obsługi w postaci wywołania zwrotnego ajax(..)
        // powinna być funkcją resolve(..) dla tej obietnicy.
        ajax( url, resolve );
    } );
}
```

Zaczynamy od zdefiniowania funkcji `request(..)` tworzącej obietnicę, która będzie przedstawiała kompletne wywołanie `ajax(..)`:

```
request( "http://dowolny.adres.url.1/" )
.then( function(response1){
    return request( "http://dowolny.adres.url.2/?v=" + response1 );
} )
.then( function(response2){
    console.log( response2 );
} );
```

Programiści często spotykają się z sytuacją, w której bazująca na obietnicach asynchroniczna kontrola przepływu ma się odbywać za pomocą narzędzi nieprzystosowanych do użycia obietnic, na przykład takiego jak funkcja `ajax(..)` w powyższym kodzie, oczekująca przekazania jej wywołania zwrotnego. Wprawdzie natywny w specyfikacji ES6 mechanizm `Promise` nie rozwiązuje automatycznie tego problemu za nas, ale praktycznie wszystkie biblioteki dostarczają odpowiednie rozwiązanie. Z reguły proces nosi nazwę *lifting*, *promisifyng* lub podobną. Do wymienionych technik powrócimy w dalszej części książki.

Używając zwracającego obietnicę wywołania `request(..)`, niejawnie tworzymy pierwszy krok naszego łańcucha. Odbywa się to przez wywołanie funkcji wraz z pierwszym adresem URL oraz połączenie zwróconej obietnicy z pierwszym wywołaniem `then(..)`.

Po otrzymaniu odpowiedzi `response1` używamy tej wartości do przygotowania drugiego adresu URL i wykonania drugiego wywołania `request(..)`. Obietnica utworzona przez drugie wywołanie `request(..)` zostaje zwrócona (polecenie `return`), a krok trzeci w asynchronicznej kontroli przepływu działania programu czeka na zakończenie wywołania Ajax. Na koniec wyświetlamy zawartość `response2` po jej otrzymaniu.

Tworzony tutaj łańcuch obietnic to nie tylko wyrażony w wielokrokowej sekwencji asynchronicznej mechanizm kontroli przepływu działania programu, ale działa również jako kanał komunikatów przekazujących odpowiednie komunikaty między poszczególnymi krokami.

Co się stanie w przypadku wystąpienia problemów w jednym z kroków łańcucha obietnic? Wystąpienie błędu lub zgłoszenie wyjątku odbywa się dla poszczególnych obietnic, co oznacza możliwość przechwycenia błędu w dowolnym miejscu łańcucha. Wspomniane przechwycenie działa w rodzaju „wyzerowania" łańcucha i przywrócenia go do normalnego działania w danym miejscu:

```
// Krok 1.
request( "http://dowolny.adres.url.1/" )

// Krok 2.
.then( function(response1){
    foo.bar(); // Wartość undefined, mamy błąd!

    // Nigdy nie dotrzemy do tego wiersza.
    return request( "http://dowolny.adres.url.2/?v=" + response1 );
} )

// Krok 3.
.then(
    function fulfilled(response2){
        // Nigdy nie dotrzemy do tego wiersza.
    },
    // Procedura obsługi błędów w przypadku odrzucenia obietnicy.
    function rejected(err){
        console.log( err );
        // Błąd TypeError zgłoszony przez funkcję foo.bar().
        return 42;
    }
)

// Krok 4.
.then( function(msg){
    console.log( msg );      // 42
} );
```

Kiedy błąd wystąpi w kroku 2., to zostanie wychwycony w kroku 3. przez procedurę obsługi błędów odrzuconej obietnicy. Wartość zwrotna (w omawianym przykładzie to `42`), o ile jakakolwiek zostanie wygenerowana przez wspomnianą procedurę obsługi błędów, spełni obietnicę w kolejnym kroku (4.), a tym samym cały łańcuch znów będzie znajdował się w stanie spełnienia obietnicy.

Jak już wcześniej wspomniałem, podczas zwrotu obietnicy z procedury obsługi dla spełnionej obietnicy zostanie ona rozpakowana, a kolejny krok może być wykonany po upływie pewnej ilości czasu. To samo dotyczy także procedur obsługi dla odrzuconych obietnic. Jeżeli w kroku 3. zamiast obietnicy będzie zwrócona wartość 42, to ta obietnica może być opóźniona w kroku 4. Zgłoszenie wyjątku wewnątrz procedury obsługi spełnionej bądź odrzuconej obietnicy powoduje, że kolejna (połączona) obietnica jest natychmiast odrzucana wraz z danym wyjątkiem.

Jeżeli wywołamy `then(..)` w obietnicy i przekażemy tylko procedurę obsługi odpowiedzialną za spełnioną obietnicę, to przyjmuje się, że procedura obsługi dla odrzuconej obietnicy zostanie zastąpiona domyślną.

```
var p = new Promise( function(resolve,reject){
    reject( "Ups!" );
} );
```

```
var p2 = p.then(
    function fulfilled(){
        // Nigdy nie dotrzemy do tego wiersza.
    }
    // Jeżeli dla odrzuconej obietnicy zostanie pominięta procedura
    // obsługi lub będzie dostarczona wartość inna niż funkcja,
    // to nastąpi użycie domyślnej procedury:
    // function(err) {
    //     throw err;
    // }.
);
```

Jak możesz zobaczyć, domyślna procedura obsługi po prostu ponownie zgłasza błąd, co wymusza na obietnicy `p2` (połączonej) odrzucenie z tego samego powodu. W ten sposób błąd jest propagowany dalej w łańcuchu obietnic aż do chwili napotkania wyraźnie zdefiniowanej procedury obsługi dla odrzuconej obietnicy.

Więcej szczegółów dotyczących obsługi błędów w kodzie bazującym na obietnicach poznasz w dalszej części książki. Zapewniam Cię, że w tym zakresie pozostało jeszcze sporo niuansów do uwzględnienia.

Jeżeli do wywołania `then(..)` nie zostanie przekazana poprawna funkcja służąca w charakterze procedury obsługi dla spełnionej obietnicy, to również będzie użyta domyślna:

```
var p = Promise.resolve( 42 );

p.then(
    // Jeżeli dla spełnionej obietnicy zostanie pominięta procedura
    // obsługi lub będzie dostarczona wartość inna niż funkcja,
    // to nastąpi użycie domyślnej procedury:
    // function(v) {
    //     return v;
    // }.
    null,
    function rejected(err){
        // Nigdy nie dotrzemy do tego wiersza.
    }
);
```

Jak możesz zobaczyć, domyślna procedura obsługi dla spełnionej obietnicy otrzymaną wartość po prostu przekazuje do kolejnego kroku (obietnicy).

Dla wzorca w postaci wywołania `then(null,function(err){ .. })` — czyli obsługa jedynie odrzucenia (o ile wystąpi) i pozwolenie na przekazanie dalej spełnionej obietnicy — istnieje pewien skrót w API: `catch(function(err){ .. })`. Dokładniejsze omówienie funkcji `catch(..)` znajdziesz w następnym podrozdziale.

Pokrótce przejrzymy teraz zachowanie obietnic pozwalające na zastosowanie połączonej kontroli przepływu działania programu:

- Wywołanie `then(..)` wraz z jedną obietnicą automatycznie powoduje wygenerowanie nowej procedury, która stanie się wartością zwrotną danego wywołania.
- Jeżeli wewnątrz procedur obsługi spełnienia lub odrzucenia obietnicy nastąpi zwrot wartości lub zgłoszenie wyjątku, to nowo zwrócona (możliwa do połączenia) obietnica zostanie odpowiednio rozwiązana.

- Jeżeli procedura obsługi spełnienia lub odrzucenia obietnicy zwróci obietnicę, to będzie rozpakowana, a rozwiązanie danej obietnicy stanie się rozwiązaniem połączonej obietnicy zwróconej przez bieżące wywołanie `then(..)`.

Wprawdzie łączenie kontroli przepływu działania programu jest użyteczne, ale prawdopodobnie najlepsze podejście polega na potraktowaniu tej możliwości jako dodatkowej korzyści wynikającej ze sposobu łączenia obietnic ze sobą, a nie jako celowo opracowanej funkcjonalności. Jak to już wielokrotnie dokładnie wspominałem, obietnice normalizują asynchroniczność i hermetyzują zależną od czasu wartość stanu. *Dlatego* też można je ze sobą łączyć w tak użyteczny dla nas sposób.

Nie ulega wątpliwości, że sekwencyjne wyrażenie łańcucha (to, następnie to, później tamto itd.) jest ogromnym usprawnieniem w porównaniu z bałaganem wprowadzanym przez wywołania zwrotne, o czym szczegółowo opowiedziałem w rozdziale 2. W przypadku obietnic nadal mamy pewną ilość szkieletu kodu (`then(..)` i `function(){ .. }`), którego istnienie trzeba uwzględnić. W rozdziale 4. poznasz znacznie lepszy wzorzec przeznaczony do wyrażenia sekwencyjnej kontroli przepływu działania programu — generator.

Terminologia — rozwiązanie, spełnienie i odrzucenie

Może istnieć pewne zamieszanie związane z pojęciami *rozwiązanie*, *spełnienie* i *odrzucenie*. Warto więc je wyjaśnić, zanim przejdziemy do dokładniejszego omawiania obietnic. Zacznijmy od spojrzenia na konstruktora `Promise(..)`:

```
var p = new Promise( function(X,Y){
    // X() dla spełnionej obietnicy.
    // Y() dla odrzuconej obietnicy.
} );
```

Jak możesz zobaczyć, w konstruktorze są zdefiniowane dwa wywołania zwrotne o nazwach `X` i `Y`. Pierwsze jest *zwykle* używane do oznaczenia spełnionej obietnicy, natomiast drugie *zawsze* oznacza obietnicę jako odrzuconą. Mógłbyś w tym miejscu zapytać, co oznacza *zwykle* i czy ma znaczenie, aby tym parametrom nadawać odpowiednie nazwy.

Ostatecznie to przecież jest kod utworzony przez Ciebie i nazwy identyfikatorów nie są interpretowane przez silnik jako mające znaczenie. Dlatego też pod względem technicznym nazwy nie mają znaczenia, zarówno `foo(..)`, jak i `bar(..)` są tak samo funkcjonalne. Jednak używane słowa mogą wpływać nie tylko na sposób myślenia o kodzie, ale także na to, jak inni programiści Twojego zespołu będą traktować ten kod. Niewłaściwe zrozumienie starannie opracowanego kodu asynchronicznego zdecydowanie będzie gorszą sytuacją niż użycie alternatywy w postaci kodu opartego na wywołaniach zwrotnych.

Wygląda więc na to, że stosowane nazewnictwo parametrów konstruktora `Promise` ma jednak pewne znaczenie.

Najłatwiej jest zdefiniować nazwę drugiego parametru. Niemalże w całej literaturze poświęconej obietnicom stosowana jest nazwa `reject(..)`, czyli odrzucenie. Ponieważ ona (jako jedyna!) dokładnie określa przeznaczenie funkcji, to stanowi doskonały wybór dla nazwy. Gorąco zachęcam Cię, aby zawsze używać nazwy `reject(..)` dla procedury obsługi odrzuconej obietnicy.

Nieco więcej niejasności dotyczy pierwszego parametru, który w literaturze poświęconej obietnicom jest często nazywany `resolve(..)`. To słowo jest oczywiście powiązane ze słowem *resolution* (ang. — „rozwiązanie"), które w literaturze (także w tej książce) jest używane w określeniu do przypisania ostatecznej wartości lub stanu obietnicy. W tekście wielokrotnie już użyłem sformułowania „rozwiązać obietnicę" w kontekście oznaczającym spełnienie lub odrzucenie obietnicy.

Skoro pierwszy parametr jest używany do wskazania na spełnioną obietnicę, wobec tego czy zamiast nazwy `resolve(..)` nie powinniśmy stosować `fulfill(..)`? Czy słowo *fulfill* (ang. „spełnienie") nie jest trafniejsze? Aby udzielić odpowiedzi na tak zadane pytanie, spójrz na dwie metody znajdujące się w API obietnic:

```
var fulfilledPr = Promise.resolve( 42 );

var rejectedPr = Promise.reject( "Ups!" );
```

Wywołanie `Promise.resolve(..)` tworzy obietnicę, która będzie rozwiązana za pomocą przypisanej jej wartości. W omawianym przykładzie `42` to normalna, niebędąca obietnicą wartość. Dlatego też następuje utworzenie spełnionej obietnicy `fulfilledPr` o wartości `42`. Z kolei wywołanie `Promise.reject("Ups!")` tworzy obietnicę `rejectedPr` odrzuconą z powodu `"Ups!"`.

Pokażę teraz, dlaczego słowo „rozwiązać" (na przykład użyte w `Promise.resolve(..)`) jasno i zdecydowanie lepiej wyraża przeznaczenie funkcji, o ile oczywiście będzie wyraźnie zastosowane w kontekście, który może skutkować spełnieniem lub odrzuceniem:

```
var rejectedTh = {
    then: function(resolved,rejected) {
        rejected( "Ups!" );
    }
};

var rejectedPr = Promise.resolve( rejectedTh );
```

Jak już wcześniej wspomniałem w rozdziale, wartością zwrotną wywołania `Promise.resolve(..)` jest otrzymana bezpośrednio prawdziwa obietnica bądź też następuje rozpakowanie otrzymanej wartości *thenable*. Jeżeli wspomniana rozpakowana wartość ujawnia stan odrzucenia, to obietnica zwrócona przez `Promise.resolve(..)` jest w rzeczywistości tym samym stanem odrzucenia.

Dlatego też `Promise.resolve(..)` to dobra, odpowiednia nazwa dla metody API, ponieważ w rzeczywistości może skutkować spełnieniem lub odrzuceniem obietnicy.

Pierwszy parametr wywołania zwrotnego konstruktora `Promise(..)` zostanie rozpakowany na postać wartości *thenable* (skutek działania identyczny jak wywołania `Promise.resolve(..)`) lub też na autentyczną obietnicę:

```
var rejectedPr = new Promise( function(resolve,reject){
    // Rozwiązaniem tej obietnicy będzie odrzucenie.
    resolve( Promise.reject( "Ups!" ) );
} );

rejectedPr.then(
    function fulfilled(){
        // Nigdy nie dotrzemy do tego wiersza.
    },
    function rejected(err){
        console.log( err ); // "Ups!"
    }
);
```

Powinno być teraz jasne, że `resolve(..)` to odpowiednia nazwa dla pierwszego parametru (wywołania zwrotnego) konstruktora `Promise(..)`.

Wspomniana wcześniej funkcja `reject(..)` *nie* przeprowadza rozpakowania znanego z funkcji `resolve(..)`. Jeżeli funkcji `reject(..)` przekażesz obietnicę lub wartość *thenable*, to ta niezmodyfikowana wartość będzie wskazana jako powód odrzucenia obietnicy. Kolejne procedury obsługi odrzucenia będą otrzymywały przekazaną do `reject(..)` rzeczywistą obietnicę lub wartość *thenable*, a nie wartość natychmiastową.

Skierujmy teraz naszą uwagę na wywołania zwrotne dostarczane do `then(..)`. Jak powinny być one nazwane (zarówno w literaturze, jak i w kodzie)? Gorąco zachęcam do stosowania nazw `fulfilled(..)` i `rejected(..)`:

```
function fulfilled(msg) {
    console.log( msg );
}

function rejected(err) {
    console.error( err );
}

p.then(
    fulfilled,
    rejected
);
```

W przypadku pierwszego parametru `then(..)` nie ma wątpliwości, że zawsze dotyczy spełnionej obietnicy, a tym samym nie potrzeba stosować podwójnej terminologii „rozwiązania”. Przy okazji warto dodać, że w specyfikacji ES6 używane są `onFulfilled(..)` i `onRejected(..)` jako nazwy dwóch omówionych wywołań zwrotnych. Dlatego też można je uznać za odpowiednie pojęcia.

Obsługa błędów

Jak dotąd, widziałeś wiele przykładów, jak odrzucenie obietnic — zarówno celowo za pomocą wywołania `reject(..)`, jak i przypadkowo, na skutek zgłoszenia wyjątków JavaScript — pozwala na zastosowanie obsługi błędów w programowaniu asynchronicznym. Powróćmy teraz do tego tematu i zajmijmy się pewnymi szczegółami, które wcześniej zostały pominięte.

Dla większości programistów najbardziej naturalną formą obsługi błędów jest synchroniczna konstrukcja `try-catch`. Niestety, jest jedynie synchroniczna, a tym samym nie nadaje się do zastosowania w kodzie działającym asynchronicznie:

```
function foo() {
    setTimeout( function(){
        baz.bar();
    }, 100 );
}


try {
    foo();
    // Później zgłosi błąd globalny w baz.bar().
}
```

```
catch (err) {
    // Nigdy nie dotrzemy do tego wiersza.
}
```

Konstrukcję `try-catch` zdecydowanie warto stosować, choć nie sprawdza się ona w operacjach asynchronicznych, przynajmniej bez pewnego dodatkowego wsparcia ze strony środowiska; powrócimy do tego w rozdziale 4., podczas omawiania generatorów.

W wywołaniach zwrotnych zostały wypracowane pewne standardy w zakresie obsługi błędów, w tym oczywiście wspomniany już wcześniej styl *najpierw błąd*.

```
function foo(cb) {
    setTimeout( function(){
        try {
            var x = baz.bar();
            cb( null, x ); // Sukces!
        }
        catch (err) {
            cb( err );
        }
    }, 100 );
}

foo( function(err,val){
    if (err) {
        console.error( err ); // Porażka :-(
    }
    else {
        console.log( val );
    }
} );
```

Konstrukcja `try-catch` w powyższym fragmencie kodu działa jedynie z następującej perspektywy: wywołanie funkcji `baz.bar()` natychmiast zakończy się sukcesem lub porażką, synchronicznie. Jeżeli funkcja `baz.bar()` sama dla siebie była asynchroniczną funkcją ukończenia zadania, to wszelkie występujące w niej błędy asynchroniczne nie będą możliwe do przechwycenia.

Wywołanie zwrotne przekazywane `foo(..)` oczekuje otrzymania sygnału błędu poprzez użycie zarezerwowanego pierwszego parametru o nazwie `err`. Obecność wymienionego parametru oznacza wystąpienie błędu. Z kolei brak parametru `err` jest traktowany jako oznaka sukcesu.

Tego rodzaju obsługa błędów jest pod względem technicznym możliwa do zastosowania w kodzie asynchronicznym, ale nie komponuje się z nim zbyt dobrze. Wiele poziomów wywołań zwrotnych sprawdzających najpierw, czy wystąpił błąd, wraz z wszechobecnymi poleceniami `if` niewątpliwie doprowadzi Cię do przedstawionego w rozdziale 2. piekła wywołań zwrotnych.

Powracamy więc do obsługi błędów w obietnicach wraz z przekazaną do `then(..)` procedurą obsługi odrzucenia obietnicy. W obietnicach nie stosujemy popularnego podejścia wywołań zwrotnych w stylu „najpierw błąd”, ale zamiast tego wykorzystujemy styl podzielonego wywołania zwrotnego. Mamy więc po jednym wywołaniu zwrotnym przeznaczonym dla spełnienia i odrzucenia:

```
var p = Promise.reject( "Ups!" );

p.then(
    function fulfilled(){
        // Nigdy nie dotrzemy do tego wiersza.
    },
```

```
        function rejected(err){
            console.log( err ); // "Ups!"
        }
    );
```

Wprawdzie przedstawiony powyżej wzorzec obsługi błędów ma sens, ale niuanse dotyczące obsługi błędów w obietnicach są zwykle nieco trudniejsze do zrozumienia w pełni.

Spójrz na poniższy fragment kodu:

```
var p = Promise.resolve( 42 );

p.then(
    function fulfilled(msg){
        // Liczby nie obsługują funkcji dotyczących ciągów tekstowych,
        // a więc następuje zgłoszenie błędu.
        console.log( msg.toLowerCase() );
    },
    function rejected(err){
        // Nigdy nie dotrzemy do tego wiersza.
    }
);
```

Jeżeli funkcja `msg.toLower()` jak najbardziej słusznie zgłosi błąd (a tak się stanie!), to dlaczego nasza procedura błędów nie zostanie o tym fakcie poinformowana? Jak już wcześniej wyjaśniłem, ponieważ *ta konkretna* procedura obsługi błędów jest przeznaczona dla obietnicy `p`, która została spełniona za pomocą wartości `42`. Obietnica `p` jest niemodyfikowalna, a więc jedyną obietnicą, którą można poinformować o błędzie, jest obietnica zwracana przez `p.then(..)`. Jednak w omawianym przykładzie wspomniana obietnica nie przechwytuje błędu.

Teraz powinno być już jasne, dlaczego obsługa błędów w kodzie bazującym na obietnicach jest podatna na błędy (celowa gra słów). Bardzo łatwo można doprowadzić do ukrycia błędów, co niezwykle rzadko jest działaniem zamierzonym.

Jeżeli API obietnic używasz w nieprawidłowy sposób i wystąpi błąd uniemożliwiający prawidłowe utworzenie obietnicy, to wynikiem będzie natychmiastowe zgłoszenie wyjątku, *a nie odrzucenie obietnicy*. Oto wybrane przykłady niepoprawnego użycia kodu prowadzące do niepowodzenia operacji tworzenia obietnicy: `new Promise(null)`, `Promise.all()`, `Promise.race(42)` itd. Obietnica nie może zostać odrzucona, jeśli nie użyto na tyle prawidłowego API, aby najpierw w rzeczywistości utworzyć obietnicę!

Otchłań rozpaczy

Jeff Atwood zauważył kilka lat temu, że języki programowania są bardzo często skonfigurowane w sposób, który domyślnie wpędza programistów w *otchłań rozpaczy* — wypadki są karane i trzeba włożyć więcej wysiłku, aby osiągnąć zamierzony efekt. Jeff błagał o utworzenie „otchłani sukcesu", gdzie domyślnie przeprowadzana jest oczekiwana (zakończona sukcesem) operacja, a tym samym trudniej jest ponieść porażkę[1].

Obsługa błędów w obietnicach to bez wątpienia projekt w stylu „otchłani rozpaczy". Domyślnie przyjęte jest założenie, że wszelkie błędy mają być ukrywane przez stan obietnicy. Dlatego też jeśli

[1] *http://blog.codinghorror.com/falling-into-the-pit-of-success/*

zapomnisz o obserwacji wspomnianego stanu, błąd pozostaje nieujawniony i doprowadza programistę do rozpaczy.

Aby uniknąć utraty informacji o wystąpieniu błędu, niektórzy programiści są przekonani, że najlepszą praktyką w przypadku łańcucha obietnic jest zawsze umieszczenie na końcu ostatecznego polecenia `catch(..)`, jak pokazałem w poniższym fragmencie kodu:

```
var p = Promise.resolve( 42 );

p.then(
    function fulfilled(msg){
        // Liczby nie obsługują funkcji dotyczących ciągów tekstowych,
        // a więc następuje zgłoszenie błędu.
        console.log( msg.toLowerCase() );
    }
)
.catch( handleErrors );
```

Ponieważ procedura obsługi odrzuconej obietnicy nie jest przekazywana do `then(..)`, więc użyta będzie domyślna, której działanie polega po prostu na przekazaniu błędu do kolejnej obietnicy w łańcuchu. Dlatego też błędy zarówno dostarczane do obietnicy `p`, jak i powstałe *po* jej rozwiązaniu (na przykład za pomocą `msg.toLowercase()`) zostaną przekazane do zdefiniowanej na końcu funkcji `handleErrors(..)`.

Problem mamy więc rozwiązany, prawda? Nie tak szybko!

Co się stanie, jeśli błąd wystąpi w samej funkcji `handleErrors(..)`? Który komponent przechwyci taki błąd? W kodzie nadal pozostaje niepilnowana obietnica zwracana przez wywołanie `catch(..)`. Nie przechwytujemy jej i nie rejestrujemy dla niej procedury obsługi uruchamianej po odrzuceniu obietnicy.

Na końcu łańcucha nie można po prostu umieścić kolejnego wywołania `catch(..)`, ponieważ ono również może zakończyć się niepowodzeniem. W przypadku ostatniego kroku w dowolnym łańcuchu obietnic, niezależnie od tego, jaki on będzie, zawsze istnieje ryzyko wystąpienia nieprzechwyconego błędu wewnątrz nieobserwowanej obietnicy.

Wydaje się, że to problem niemożliwy do rozwiązania.

Obsługa nieprzechwyconych błędów

Pełne rozwiązanie tego problemu nie należy do prostych zadań. Istnieją inne (wielu powiedziałoby, że prostsze) podejścia.

W pewnych bibliotekach opartych na obietnicach dodano metody przeznaczone do rejestracji czegoś w stylu procedury obsługi dla „globalnego nieobsłużonego odrzucenia obietnicy”, która to procedura będzie wywoływana zamiast zgłoszenia globalnego błędu. Jednak w tym przypadku sposób identyfikacji błędu jako nieprzechwyconego polega na ustawieniu licznika odliczającego pewną ilość czasu, na przykład trzy sekundy od chwili odrzucenia. Jeżeli obietnica jest odrzucona, ale przed upływem wspomnianego czasu nie została zarejestrowana żadna procedura obsługi błędu, to przyjmowane jest założenie, że procedura obsługi nie będzie już nigdy zarejestrowana i błąd pozostanie nieprzechwycony.

W praktyce wymienione rozwiązanie sprawdzało się doskonale w wielu bibliotekach, ponieważ w większości wzorców użycia zwykle nie występowało zbyt duże opóźnienie między odrzuceniem obietnicy i zauważeniem tego faktu. Jednak podejście samo w sobie jest problematyczne, ponieważ wspomniane wcześniej trzy sekundy to dowolnie ustalona wartość (nawet jeśli na podstawie działań empirycznych). Ponadto na pewno wystąpią sytuacje, gdy odrzucenie obietnicy chcesz wstrzymać przez pewien nieokreślony czas i naprawdę nie chcesz, aby dla tego rodzaju fałszywych alarmów została wywołania procedura obsługi nieprzechwyconych błędów.

Inna często pojawiająca się sugestia to taka, że obietnice powinny mieć dodaną funkcję `done(..)`, której zadaniem jest oznaczenie łańcucha obietnic jako zakończonego. Funkcja `done(..)` nie tworzy i nie zwraca obietnicy, a więc wywołania zwrotne przekazane do `done(..)` oczywiście nie będą obarczone koniecznością zgłaszania problemów do nieistniejącej połączonej obietnicy.

Powstaje więc pytanie, co się dzieje zamiast tego. Mamy do czynienia z sytuacją, której zwykle oczekujemy w przypadku wystąpienia nieprzechwyconego błędu — wyjątek wewnątrz procedury obsługi odrzucenia `done(..)` będzie zgłaszany jako globalny nieprzechwycony błąd (w zasadzie w konsoli programisty):

```
var p = Promise.resolve( 42 );

p.then(
    function fulfilled(msg){
        // Liczby nie obsługują funkcji dotyczących ciągów tekstowych,
        // a więc następuje zgłoszenie błędu.
        console.log( msg.toLowerCase() );
    }
)
.done( null, handleErrors );

// Jeżeli funkcja handleErrors(..) jest odpowiedzialna za dany wyjątek,
// w tym miejscu zostanie on zgłoszony globalnie.
```

Przedstawione podejście może wydawać się znacznie atrakcyjniejsze niż nigdy niekończący się łańcuch lub też dowolnie definiowany maksymalny limit dozwolonego czasu. Jednak największy problem związany z powyższym rozwiązaniem polega na tym, że nie stanowi ono części standardu ES6. Dlatego też niezależnie, jak dobrze się prezentuje, raczej nie można go uznać za niezawodne i powszechnie stosowane rozwiązanie.

Czy naprawdę nie mamy żadnego wyjścia? Niekoniecznie.

Przeglądarki internetowe oferują unikalną możliwość niedostępną w kodzie — mogą monitorować i dokładnie wiedzieć, kiedy dany obiekt zostanie odrzucony i będzie usunięty przez mechanizm usuwania nieużytków. Dlatego też przeglądarka internetowa może monitorować obiekty obietnic i jeśli zauważy jego usunięcie przez mechanizm usuwania nieużytków, a obietnica została odrzucona, to przeglądarka wie o wystąpieniu nieprzechwyconego błędu i może o tym powiadomić programistę za pomocą odpowiedniego komunikatu wyświetlanego w konsoli.

Gdy powstawała ta książka, twórcy przeglądarek Chrome i Firefox poczynili pierwsze próby zaimplementowania opisanej powyżej metody, choć taki sposób obsługi nieprzechwyconych błędów można w najlepszym przypadku uznać za niekompletny.

Jednak jeżeli obiekt obietnicy nie zostanie usunięty za pomocą mechanizmu usuwania nieużytków — o to nietrudno, zwłaszcza podczas stosowania wielu różnych wzorców tworzenia kodu — to wspomniany mechanizm nie okaże się zbyt użyteczny w informowaniu i diagnozowaniu po cichu odrzuconych obietnic.

Czy istnieje jakakolwiek inna alternatywa? Tak.

Otchłań sukcesu

Przedstawione poniżej rozważania są czysto teoretyczne i pokazują, jak pewnego dnia może się zmienić zachowanie obietnic. Jestem przekonany, że wówczas będziemy dysponowali o wiele lepszym rozwiązaniem od obecnego. Jak sądzę, zmiana będzie możliwa do wprowadzenia nawet w wersjach JavaScript nowszych niż ES6, ponieważ raczej w żaden sposób nie będzie niezgodna z obietnicami ES6. Co więcej, powinna być możliwa do zastosowania w skryptach typu polyfill, o ile zachowasz ostrożność. Spójrz na poniższe punkty:

- Obietnica może w trakcie kolejnego zadania lub tyknięcia pętli zdarzeń domyślnie zgłaszać (w konsoli programisty) wszelkie odrzucenia, jeżeli dokładnie w tym momencie dla obietnicy nie była zarejestrowana żadna procedura obsługi błędów.
- Jeśli chcesz, aby odrzucona obietnica wstrzymała stan odrzucenia na nieokreślony czas, zanim wspomniane odrzucenie zostanie zauważone, można wywołać funkcję `defer()`, która zawiesza automatyczne zgłaszanie błędów dla danej obietnicy.

Jeżeli obietnica jest odrzucona, domyślnie ten fakt zostanie wyraźnie zgłoszony w konsoli programisty, a nie będzie pominięty. Możesz zdecydować się na zgłoszenie niejawne (przez rejestrację procedury obsługi błędów przed odrzuceniem obietnicy) lub wyraźnie (za pomocą funkcji `defer()`). W obu wymienionych przypadkach *zachowujesz* kontrolę nad fałszywymi alarmami.

Spójrz na poniższy fragment kodu:

```
var p = Promise.reject( "Ups!" ).defer();

// Funkcja foo(..) zawiera kod bazujący na obietnicy.
foo( 42 )
.then(
    function fulfilled(){
        return p;
    },
    function rejected(err){
        // Obsługa błędów w foo(..).
    }
);
...
```

Kiedy tworzymy obietnicę `p`, to wiemy o konieczności zaczekania, zanim będzie można zaobserwować i użyć jej odrzucenia. Wywołujemy więc funkcję `defer()`, która chroni przed zgłoszeniem błędu globalnego. Funkcja `defer()` po prostu zwraca tę samą obietnicę na potrzeby łączenia.

Obietnica zwrócona przez `foo(..)` od razu otrzymuje procedurę obsługi błędów, co oznacza niejawne wykluczenie globalnego zgłaszania błędów.

Jednak obietnica zwrócona przez wywołanie `then(..)` nie ma dołączonej funkcji `defer()` lub procedury obsługi błędów. Dlatego jeżeli zostanie odrzucona (z poziomu dowolnej procedury obsługi rozwiązania obietnicy), odrzucenie *to* zostanie zgłoszone w konsoli programisty jako nieprzechwycony błąd.

Tak przedstawia się projekt otchłani sukcesu. Domyślnie wszystkie błędy są obsługiwane lub zgłaszane — prawie wszyscy programiści oczekują takiego podejścia w praktycznie każdej sytuacji. Możesz zarejestrować własną procedurę obsługi lub też zdecydować się na odłożenie obsługi błędów na *później* — ponosisz dodatkową odpowiedzialność w tym konkretnym przypadku.

Jedyne prawdziwe niebezpieczeństwo związane z przedstawionym podejściem polega na tym, że po użyciu wywołania `defer()` zapomnisz o rzeczywistej obserwacji i obsłużeniu odrzucenia obietnicy.

Nie zapominaj, że musisz celowo wywołać `defer()`, aby przejść do stylu otchłani rozpaczy — domyślnie mamy otchłań sukcesu — a więc niewiele więcej można zrobić w celu ochrony siebie przed własnymi błędami.

Mimo wszystko uważam, że nadal istnieje nadzieja dla obsługi błędów w mechanizmie obietnic (w nowszych wydaniach JavaScript niż ES6). Wierzę w przemyślenie obecnej sytuacji i rozważenie alternatywy. W międzyczasie przedstawione podejście możesz zaimplementować samodzielnie (to będzie wymagające ćwiczenie dla Ciebie) lub wykorzystać bazującą na obietnicach, sprytnie działającą bibliotekę, która to zrobi za Ciebie!

Dokładnie ten model obsługi i zgłaszania błędów został zaimplementowany w mojej bibliotece abstrakcji obietnic o nazwie *asynquence*, której bardziej szczegółowe omówienie znajdziesz w dodatku A.

Wzorce obietnic

Wcześniej wspomniałem o wzorcu sekwencji łańcucha obietnic (przepływ kontroli w stylu „to, następnie to, a później tamto"). Istnieje znacznie więcej wariantów wzorców asynchronicznych możliwych do wykorzystania w kodzie bazującym na obietnicach. Wzorce te pozwalają na uproszczenie wyrażenia asynchronicznego przepływu kontroli — co pomaga w utworzeniu sensowniejszego i łatwiejszego w konserwacji kodu — nawet w najbardziej skomplikowanych fragmentach programów.

Dwa tego rodzaju wzorce zostały wbudowane bezpośrednio w natywnej implementacji obietnic ES6 (`Promise`). Dlatego też bez konieczności podejmowania jakichkolwiek dodatkowych kroków wspomniane wzorce możesz wykorzystać w charakterze elementów konstrukcyjnych dla innych wzorców.

Promise.all([..])

W sekwencji asynchronicznej (łańcuch obietnic) w danym momencie jest koordynowane tylko jedno zadanie asynchroniczne, czyli krok 2. następuje dokładnie po kroku 1., natomiast krok 3. jest wykonywany dokładnie po kroku 2. Co w sytuacji, gdy chcemy, aby co najmniej dwa kroki były wykonywane jednocześnie (czyli „współbieżnie")?

W terminologii programowania klasycznego brama jest mechanizmem wstrzymującym kontynuację działania programu aż do zakończenia wykonywania dwóch lub więcej jednoczesnych (współbieżnych) zadań. Nie ma znaczenia, w jakiej kolejności wspomniane zadania zostaną ukończone. Po prostu wszystkie muszą być zakończone, aby nastąpiło otworzenie bramy i umożliwienie dalszego działania programu.

W przypadku API obietnic omawiany wzorzec istnieje w postaci wywołania `all([ .. ])`.

Przyjmujemy założenie, że jednocześnie chcesz wykonać dwa żądania Ajax i przed wykonaniem trzeciego zaczekać na zakończenie dwóch pierwszych, niezależnie od kolejności, w jakiej to nastąpi. Spójrz na poniższy fragment kodu:

```
// Wywołanie request(..) to narzędzie Ajax oparte na obietnicach,
// podobne do zdefiniowanego wcześniej w rozdziale.

var p1 = request( "http://dowolny.adres.url.1/" );
var p2 = request( "http://dowolny.adres.url.2/" );

Promise.all( [p1,p2] )
.then( function(msgs){
    // Obie obietnice p1 i p2 są spełnione
    // i tutaj dostarczają swoje komunikaty.
    return request(
        "http://dowolny.adres.url.3/?v=" + msgs.join(",")
    );
} )
.then( function(msg){
    console.log( msg );
} );
```

Wywołanie `Promise.all([ .. ])` oczekuje jednego argumentu w postaci tablicy (`array`) składającej się z egzemplarzy obietnic. Obietnica zwrócona przez wywołanie `Promise.all([ .. ])` otrzymuje komunikat spełnienia (w omawianym przykładzie to `msg`) będący tablicą wszystkich komunikatów spełnienia przekazanych obietnicom w tej samej kolejności, w jakiej zostały podane (niezależnie od kolejności spełnienia obietnic).

Pod względem technicznym tablica wartości przekazanych wywołaniu `Promise.all([ .. ])` może zawierać obietnice, wartości *thenable*, a nawet wartości natychmiastowe. Każda wartość na liście jest przekazywana za pomocą wywołania `Promise.resolve(..)`, aby mieć pewność, że otrzymamy oczekiwaną, autentyczną obietnicę. Wartość natychmiastowa będzie po prostu znormalizowana na postać obietnicy dla tej właśnie wartości. Jeżeli tablica będzie pusta, to główna obietnica zostanie natychmiast spełniona.

Główna obietnica zwrócona przez `Promise.all([ .. ])` będzie spełniona wtedy i tylko wtedy, gdy spełnione są wszystkie tworzące ją obietnice. Jeżeli choć jedna ze wspomnianych obietnic będzie odrzucona, to główna obietnica zwracana przez `Promise.all([ .. ])` zostanie natychmiast odrzucona i pozbędzie się przy tym wszystkich wartości uzyskanych z pozostałych obietnic.

Pamiętaj o dołączeniu do każdej obietnicy procedury obsługi błędów i odrzucenia, nawet i zwłaszcza w przypadku obietnic zwracanych przez wywołanie `Promise.all([ .. ])`.

Promise.race([..])

Wprawdzie wywołanie `Promise.all([ .. ])` koordynuje wiele współbieżnych obietnic i przyjmuje założenie, że wszystkie są niezbędne do spełnienia, ale czasami zachodzi potrzeba udzielenia odpowiedzi na jedynie „pierwszą obietnicę, która przekroczyła linię mety" i pominięcia pozostałych.

Ten wzorzec jest w programowaniu klasycznym określany mianem zatrzasku, natomiast w obietnicach nosi nazwę wyścigu.

Wprawdzie metafora „pierwsza obietnica, która przekroczyła linię mety" jest tutaj odpowiednia, ale niestety słowo „wyścig" to nieco nieodpowiednie pojęcie, ponieważ stan wyścigu zwykle wiąże się z błędami w programie, jak to wyjaśniłem w rozdziale 1. Nie myl więc wywołania `Promise.race([..])` ze stanem wyścigu.

Wywołanie `Promise.race([..])` również oczekuje jednego argumentu w postaci tablicy (`array`) zawierającej jedną lub więcej obietnic, wartości *thenable* lub wartości natychmiastowych. Nie ma żadnego praktycznego sensu wystąpienia wyścigu między wartościami natychmiastowymi, ponieważ pierwsza wymieniona zdecydowanie wygrywa — podobnie jak ma to miejsce w wyścigu biegaczy, w którym wygrywa uczestnik pierwszy przekraczający linię mety.

Podobnie jak w przypadku `Promise.all([..])`, także wywołanie `Promise.race([..])` zakończy się spełnieniem, jeśli rozwiązanie dowolnej obietnicy jest spełnieniem. Natomiast wynikiem wywołania `Promise.race([..])` będzie odrzucenie, gdy rozwiązaniem dowolnej obietnicy jest odrzucenie.

Wyścig wymaga przynajmniej jednego „uczestnika". Dlatego też gdy nastąpi przekazanie pustej tablicy zamiast natychmiastowego rozwiązania, główna obietnica `race([..])` nigdy nie zostanie rozwiązana. To jest strzał we własną stopę! Specyfikacja ES6 powinna wspominać, jaki będzie skutek przekazania pustej tablicy — na przykład spełnienie, odrzucenie lub po prostu zgłoszenie pewnego rodzaju błędu synchronicznego. Niestety, z powodu pierwszeństwa w bibliotekach bazujących na obietnicach wprowadzonych przed pojawieniem się obietnic w ES6 można się spotkać z różnym zachowaniem w przedstawionej kwestii. Dlatego należy zachować ostrożność i nigdy nie przekazywać pustej tablicy.

Powróćmy do wcześniejszego przykładu jednoczesnych żądań Ajax, ale w kontekście wyścigu między obietnicami `p1` i `p2`:

```
// Wywołanie request(..) to oparta na obietnicach funkcja pomocnicza Ajax,
// podobna do zdefiniowanej we wcześniejszej części rozdziału.
var p1 = request( "http://dowolny.adres.url.1/" );
var p2 = request( "http://dowolny.adres.url.2/" );

Promise.race( [p1,p2] )
.then( function(msg){
    // Wyścig zostanie wygrany przez p1 lub p2.
    return request(
        "http://dowolny.adres.url.3/?v=" + msg
    );
} )
.then( function(msg){
    console.log( msg );
} );
```

Ponieważ tylko jedna obietnica wygrywa, wartość spełniająca jest komunikatem, a nie tablicą, jak ma to miejsce w przypadku `Promise.all([ .. ])`.

Limit czasu dla wyścigu

Przedstawiony poniżej przykład widziałeś już wcześniej. Tutaj posłuży nam do zilustrowania, jak funkcję `Promise.race([ .. ])` można wykorzystać do wyrażenia wzorca wyczerpania limitu czasu:

```
// Funkcja foo() została oparta na obietnicach.

// Zakończenie działania zdefiniowanej wcześniej funkcji timeoutPromise(..).

// Obietnica odrzucona po upływie określonego czasu.

// Konfiguracja czasu dostępnego dla funkcji foo().
Promise.race( [
    foo(),                  // Próba wywołania funkcji foo().
    timeoutPromise( 3000 )  // Na wywołanie mamy trzy sekundy.
] )
.then(
    function(){
        // Funkcja foo(..) została spełniona we wskazanym czasie!
    },
    function(err){
        // Funkcja foo() została odrzucona lub po prostu
        // nie zakończyła działania w ustalonym czasie.
        // Sprawdzamy więc 'err', aby poznać przyczynę błędu.
    }
);
```

Przedstawiony powyżej wzorzec wyczerpania przydzielonego limitu czasu sprawdza się doskonale w większości przypadków. Nadal pozostają pewne niuanse do uwzględnienia, choć szczerze mówiąc, dotyczą one po równo wywołań `Promise.race([ .. ])`, jak i `Promise.all([ .. ])`.

Wywołanie finally()

Kluczowe pytanie, które trzeba sobie zadać brzmi: „Co się stanie z odrzuconą lub zignorowaną obietnicą”? Nie zadaję tego pytania z perspektywy wydajności aplikacji — obiekt takiej obietnicy zostanie usunięty przez mechanizm usuwania nieużytków — ale raczej z perspektywy behawioralnej (na przykład efekty uboczne itd.). Obietnicy nie można anulować — i nie powinno być takiej możliwości, ponieważ to oznaczałoby zniszczenie zaufania związanego z brakiem możliwości modyfikacji obietnicy z zewnątrz, o czym wspominałem we wcześniejszej części rozdziału — a więc pozostaje jedynie ciche jej zignorowanie.

Co się stanie, jeśli funkcja `foo()` użyta w poprzednim przykładzie rezerwuje pewne zasoby, ale wcześniej następuje upływ limitu czasu przeznaczonego dla obietnicy, która tym samym zostanie zignorowana? Czy istnieje jakikolwiek mechanizm pozwalający na zwolnienie zasobów po upłynięciu limitu czasu przeznaczonego dla obietnicy? Czy w inny sposób można zniwelować efekty uboczne występujące po upłynięciu wspomnianego limitu czasu przeznaczonego dla obietnicy? A co w sytuacji, gdy jedynie chcieliśmy zarejestrować fakt upłynięcia czasu dla obietnicy?

Część programistów zaproponowała, aby obietnice zawierały wywołanie zwrotne `finally(..)` wywoływane zawsze po rozwiązaniu obietnicy. W wymienionym wywołaniu można by umieścić wszelki niezbędny kod czyszczący. Na chwilę obecną wywołanie zwrotne `finally(..)` nie znajduje się

w specyfikacji, ale może pojawić się w specyfikacji ES7 lub nowszych. Czas pokaże, musimy cierpliwie poczekać.

Poniżej przedstawiłem przykład tego, jak można używać wywołania zwrotnego `finally(..)`:

```
var p = Promise.resolve( 42 );

p.then( something )
.finally( cleanup )
.then( another )
.finally( cleanup );
```

W różnych bibliotekach opartych na obietnicach wywołanie `finally(..)` nadal tworzy i zwraca nową obietnicę (w celu podtrzymania łańcucha). Jeżeli wartością zwrotną funkcji `cleanup(..)` będzie obietnica, zostanie dołączona do łańcucha. Oznacza to niebezpieczeństwo wystąpienia omówionych już wcześniej w rozdziale problemów związanych z brakiem obsłużenia odrzucenia obietnicy.

W oczekiwaniu na (ewentualne) wprowadzenie rozwiązania w specyfikacji ES7 lub nowszej można utworzyć statyczną funkcję pomocniczą, pozwalającą na obserwację (bez jakiegokolwiek zakłócania) rozwiązania obietnicy:

```
// Sprawdzenie bezpieczne dla skryptu typu polyfill.
if (!Promise.observe) {
    Promise.observe = function(pr,cb) {
        // Obserwacja z boku rozwiązania obietnicy pr.
        pr.then(
            function fulfilled (msg){
                // Zdefiniowanie w harmonogramie asynchronicznego wywołania zwrotnego (jako zadania).
                Promise.resolve( msg ).then( cb );
            },
            function rejected(err){
                // Zdefiniowanie w harmonogramie asynchronicznego wywołania zwrotnego (jako zadania).
                Promise.resolve( err ).then( cb );
            }
        );

        // Zwrot początkowej obietnicy.
        return pr;
    };
}
```

Oto przykład użycia powyższego mechanizmu w omawianym wcześniej przykładzie przydzielającym czas dla obietnicy:

```
Promise.race( [
    Promise.observe(
        foo(),                      // Próba wywołania foo().
        function cleanup(msg){
            // Operacje czyszczące po wywołaniu foo(), nawet jeśli
            // nie zostało zakończone przed upływem przyznanego czasu.
        }
    ),
    timeoutPromise( 3000 )  // Udzielenie trzech sekund.
] )
```

Ta funkcja pomocnicza `Promise.observe(..)` to jedynie ilustracja pokazująca, jak można obserwować zakończenie obietnicy bez jakiegokolwiek jej zakłócania. W innych bibliotekach obietnic można znaleźć jeszcze inne rozwiązania. Niezależnie od tego, na co się zdecydujesz, zyskasz miejsce, dzięki któremu będziesz miał pewność, że obietnice nie będą przypadkowo zignorowane bez śladu.

Warianty wywołań all([..]) i race([..])

Wprawdzie w specyfikacji ES6 rodzime obietnice są dostarczane wraz z wbudowanymi wywołaniami `Promise.all([ .. ])` i `Promise.race([ .. ])`, ale istnieje jeszcze kilka często używanych wzorców będących wariantami tych wymienionych.

`none([ .. ])`

Ten wzorzec przypomina wywołanie `all([ .. ])`, ale procedury obsługi spełnienia i odrzucenia są poprzestawiane. Wszystkie obietnice muszą być odrzucone — odrzucenia stają się wartościami spełniającymi i na odwrót.

`any([ .. ])`

Ten wzorzec przypomina wywołanie `all([ .. ])`, ale ignoruje wszelkie odrzucenia. Dlatego tylko jedno jest wymagane do spełnienia zamiast *wszystkich*.

`first([ .. ])`

Ten wzorzec przypomina użycie wywołania `any([ .. ])`, co oznacza zignorowanie wszelkich odrzuceń i spełnienie całości tuż po spełnieniu pierwszej obietnicy.

`last([ .. ])`

Ten wzorzec przypomina wywołanie `first([ .. ])`, ale wygrywa tylko ostatnie spełnienie.

Niektóre biblioteki abstrakcji obietnic zapewniają obsługę wymienionych powyżej wariantów, ale równie dobrze można je zdefiniować samodzielnie za pomocą mechanizmu obietnic — `race([ .. ])` i `all([ .. ])`.

Poniżej przedstawiłem przykład użycia wzorca `first([ .. ])`:

```
// Sprawdzenie bezpieczne dla skryptu typu polyfill.
if (!Promise.first) {
    Promise.first = function(prs) {
        return new Promise( function(resolve,reject){
            // Iteracja przez wszystkie obietnice.
            prs.forEach( function(pr){
                // Normalizacja wartości.
                Promise.resolve( pr )
                // Pierwsze spełnienie wygrywa całość i jednocześnie
                // stanowi rozwiązanie dla głównej obietnicy.
                .then( resolve );
            } );
        } );
    };
}
```

Przedstawiona implementacja `first([ .. ])` nie zostanie odrzucona, jeśli wszystkie jej obietnice będą odrzucone — po prostu ulegnie zawieszeniu, podobnie jak w przypadku wywołania `Promise.race([ .. ])`. Jeżeli zachodzi potrzeba, można zdefiniować logikę dodatkową monitorującą odrzucanie poszczególnych obietnic i po odrzuceniu wszystkich wywołującą `reject()` dla głównej obietnicy. Zastosowanie takiego podejścia pozostawiam Czytelnikowi jako ćwiczenie.

Współbieżne iteracje

Czasami zachodzi potrzeba przeprowadzenia iteracji przez listę obietnic i wykonania w nich pewnych zadań, podobnie jak ma to miejsce w tablicach synchronicznych, na przykład `forEach(..)`, `map(..)`, `some(..)` i `every(..)`. Jeżeli zadanie przeznaczone do wykonania przez wszystkie obietnice jest ściśle synchroniczne, to wszystko przebiegnie sprawnie, podobnie jak w przypadku użycia `forEach(..)` w poprzednim fragmencie kodu.

Jednak gdy wspomniane zadania są ściśle asynchroniczne lub mogą bądź powinny być wykonane współbieżnie, można wykorzystać asynchroniczne wersje tych funkcji pomocniczych dostarczanych przez wiele bibliotek.

Na przykład przeanalizujmy asynchroniczną funkcję pomocniczą `map(..)` pobierającą tablicę wartości (mogą to być obietnice lub może być cokolwiek innego) oraz funkcję (zadanie) wykonywane dla każdego elementu tablicy. Wartością zwrotną funkcji `map(..)` jest obietnica spełniana przez tablicę przechowującą (w tej samej kolejności mapowania) spełniające wartości asynchroniczne pochodzące z poszczególnych zadań:

```
if (!Promise.map) {
    Promise.map = function(vals,cb) {
        // Nowa obietnica oczekująca na wszystkie mapowane obietnice.
        return Promise.all(
            // Uwaga: zwykła funkcja tablicy o nazwie map(..),
            // jej wartością zwrotną jest tablica wartości
            // wstawiana do tablicy obietnic.
            vals.map( function(val){
                // Zastąpienie 'val' nową obietnicą, która będzie
                // rozwiązana po mapowaniu asynchronicznym 'val'.
                return new Promise( function(resolve){
                    cb( val, resolve );
                } );
            } )
        );
    };
}
```

W powyższej implementacji `map(..)` nie można asynchronicznie zasygnalizować odrzucenia, ale w przypadku wystąpienia synchronicznego wyjątku lub błędu wewnątrz mapującego wywołania zwrotnego (`cb(..)`) obietnica zwrócona przez główne wywołanie `Promise.map(..)` zostanie odrzucona.

Poniżej przedstawiłem użycie wywołania `map(..)` wraz z listą obietnic zamiast prostych wartości:

```
var p1 = Promise.resolve( 21 );
var p2 = Promise.resolve( 42 );
var p3 = Promise.reject( "Ups!" );

// Podwajamy wartości listy, nawet jeśli
// znajdują się w obietnicach.
Promise.map( [p1,p2,p3], function(pr,done){
    // Upewniamy się, że element jest obietnicą.
    Promise.resolve( pr )
    .then(
        // Wyodrębnienie wartości jako 'v'.
        function(v){
            // Mapowanie spełnienia 'v' na nową wartość.
            done( v * 2 );
        },
```

```
        // Lub mapowanie na komunikat odrzucenia obietnicy.
        done
    );
} )
.then( function(vals){
    console.log( vals );     // [42,84,"Ups!"]
} );
```

Powtórzenie wiadomości o API obietnic

Powtórzymy wiadomości na temat oferowanego przez specyfikację ES6 API obietnic, które było wzmiankowane w rozdziale.

Przedstawione poniżej API jest natywne jedynie w specyfikacji ES6, ale istnieją zgodne ze specyfikacją skrypty typu polyfill (a nie jedynie rozszerzone biblioteki bazujące na obietnicach), które mogą zdefiniować obiekt `Promise` i całe związane z nim zachowanie. W ten sposób natywne obietnice można przetestować nawet w przeglądarkach internetowych opracowanych przed wydaniem specyfikacji ES6. Jednym ze wspomnianych skryptów typu polyfill jest utworzony przeze mnie *Native Promise Only*[2].

Konstruktor new Promise()

Tak zwany „konstruktor z ujawnieniem” `Promise(..)` musi być używany wraz ze słowem kluczowym `new`, a ponadto konieczne jest podanie funkcji wywołania zwrotnego, która będzie wywołana synchronicznie (natychmiast). Wspomnianej funkcji przekazujemy dwie kolejne funkcje wywołania zwrotnego zapewniające możliwość rozwiązania obietnicy. Tym dwóm funkcjom najczęściej nadajemy nazwy `resolve(..)` i `reject(..)`:

```
var p = new Promise( function(resolve,reject){
  // Funkcja resolve(..) jest wywoływana w celu rozwiązania (spełnienia) obietnicy.
  // Funkcja reject(..) jest wywoływana w celu odrzucenia obietnicy.
} );
```

Funkcja `reject(..)` po prostu odrzuca obietnicę, natomiast `resolve(..)` może spełnić obietnicę lub ją odrzucić, w zależności od przekazanej wartości. Jeżeli funkcji `resolve(..)` zostanie przekazana wartość natychmiastowa, inna niż obietnica i inna niż *thenable*, to obietnica będzie spełniona za pomocą tej wartości.

Natomiast w przypadku przekazania funkcji `resolve(..)` autentycznej obietnicy lub wartości *thenable*, wartość ta zostanie rekurencyjnie rozpakowana, a jej ostateczny stan lub postać będą zaadaptowane przez obietnicę.

Wywołania Promise.resolve(..) i Promise.reject(..)

Skrótem dla utworzenia już odrzuconej obietnicy jest wywołanie `Promise.reject(..)`. Dlatego też obie przedstawione poniżej obietnice są identyczne:

```
var p1 = new Promise( function(resolve,reject){
    reject( "Ups!" );
```

[2] *https://github.com/getify/native-promise-only*

```
} );

var p2 = Promise.reject( "Ups!" );
```

Funkcja `Promise.resolve(..)` jest zwykle używana do utworzenia już spełnionej obietnicy w sposób podobny do wywołania `Promise.reject(..)`. Jednak funkcja `Promise.resolve(..)` dodatkowo rozpakowuje wartości *thenable* (o czym już wielokrotnie wspominałem). W przedstawionym przykładzie zwrócona obietnica adaptuje ostateczne rozwiązanie w postaci przekazanej wartości *thenable*, co może skutkować spełnieniem lub odrzuceniem obietnicy:

```
var fulfilledTh = {
    then: function(cb) { cb( 42 ); }
};
var rejectedTh = {
    then: function(cb,errCb) {
        errCb( "Ups!" );
    }
};

var p1 = Promise.resolve( fulfilledTh );
var p2 = Promise.resolve( rejectedTh );

// Obietnica p1 będzie spełniona.
// Obietnica p2 będzie odrzucona.
```

Pamiętaj, że funkcja `Promise.resolve(..)` nie podejmie żadnych działań, i jeśli przekażesz jej autentyczną obietnicę, po prostu bezpośrednio zwróci wartość. Nie występuje więc żadne obciążenie związane z wywołaniem `Promise.resolve(..)` dla wartości o nieznanej naturze, gdy okaże się, że na przykład dana wartość jest autentyczną obietnicą.

then(..) i catch(..)

Każdy egzemplarz obietnicy (*nie* przestrzeni nazw API `Promise`) ma metody `then(..)` i `catch(..)` pozwalające na zarejestrowanie procedur obsługi odpowiedzialnych za spełnienie i odrzucenie obietnicy. Podczas rozwiązywania obietnicy następuje wywołanie jednej z dwóch wymienionych procedur obsługi (ale nie obu jednocześnie). Te procedury obsługi zawsze będą wywoływane asynchronicznie (patrz podrozdział „Zadania” w rozdziale 1.).

Metoda `then(..)` pobiera jeden parametr lub dwa parametry. Pierwszy jest przeznaczony dla wywołania zwrotnego spełniającego obietnicę, natomiast drugi dla wywołania zwrotnego odrzucającego obietnicę. W przypadku pominięcia któregokolwiek z wymienionych parametrów lub przekazania mu wartości niebędącej funkcją będzie zastosowane domyślne wywołanie zwrotne. W przypadku domyślnego wywołania zwrotnego spełniającego obietnicę następuje po prostu przekazanie komunikatu. Z kolei domyślne wywołanie zwrotne odrzucające obietnicę po prostu ponownie zgłasza (propaguje) przyczynę błędu.

Metoda `catch(..)` pobiera jedynie parametr w postaci wywołania zwrotnego odrzucenia obietnicy i automatycznie stosuje domyślną procedurę obsługi, jak to zostało już omówione. Innymi słowy to jest odpowiednik wywołania `then(null,..)`.

```
p.then( fulfilled );

p.then( fulfilled, rejected );

p.catch( rejected ); // Lub p.then( null, rejected ).
```

Metody `then(..)` i `catch(..)` ponadto tworzą i zwracają nową obietnicę, która może być użyta do wyrażenia łańcucha obietnic kontroli przepływu działania programu. Jeżeli wywołanie zwrotne spełniające lub odrzucające obietnicę zgłosi wyjątek, to obietnica będąca wartością zwrotną wywołania zostanie odrzucona. Jeżeli którekolwiek z wymienionych wywołań zwrotnych zwróci wartość natychmiastową, inną niż obietnica, lub wartość *thenable*, to ta wartość stanie się spełnieniem zwracanej obietnicy. Natomiast jeśli procedura obsługi spełnienia obietnicy zwraca obietnicę lub wartość *thenable*, to ta wartość zostanie rozpakowana i stanie się rozwiązaniem dla zwracanej obietnicy.

Promise.all([..]) i Promise.race([..])

Statyczne metody pomocnicze `Promise.all([..])` i `Promise.race([..])` oferowane przez API obietnic w ES6 tworzą obietnicę, która staje się wartością zwrotną. Rozwiązanie tej obietnicy jest kontrolowane całkowicie przez przekazywaną tablicę obietnic.

W przypadku metody `Promise.all([..])` wszystkie przekazywane obietnice muszą być spełnione, aby została spełniona obietnica będąca wartością zwrotną metody. Jeżeli którakolwiek obietnica będzie odrzucona, to obietnica główna również zostanie natychmiast odrzucona (pozbywając się przy okazji wyników uzyskanych przez inne obietnice). W przypadku spełnienia otrzymujesz tablicę wartości spełniających wszystkie przekazane obietnice. Z kolei w przypadku odrzucenia otrzymujesz po prostu wartość odrzucenia pierwszej obietnicy. Tego rodzaju wzorzec jest zwykle określany mianem bramy — wszystkie dane muszą dotrzeć i dopiero wtedy brama zostanie otworzona.

W przypadku metody `Promise.race([..])` wygrywa tylko pierwsza rozwiązana (spełniona lub odrzucona) obietnica. Rozwiązanie tej pierwszej obietnicy staje się także rozwiązaniem obietnicy będącej wartością zwrotną metody. Tego rodzaju wzorzec jest zwykle określany mianem zatrzasku — pierwszy element otwierający zatrzask może przejść. Spójrz na poniższy fragment kodu:

```
var p1 = Promise.resolve( 42 );
var p2 = Promise.resolve( "Witaj, świecie!" );
var p3 = Promise.reject( "Ups!" );

Promise.race( [p1,p2,p3] )
.then( function(msg){
    console.log( msg );     // 42
} );

Promise.all( [p1,p2,p3] )
.catch( function(err){
    console.error( err );   // "Ups!"
} );

Promise.all( [p1,p2] )
.then( function(msgs){
    console.log( msgs );    // [42,"Witaj, świecie!"]
} );
```

Bądź ostrożny! Jeżeli do wywołania `Promise.all([ .. ])` przekażesz pustą tablicę, to obietnica będzie spełniona natychmiast. Z kolei w przypadku `Promise.race([ .. ])` dojdzie do zawieszenia na zawsze i obietnica nigdy nie zostanie rozwiązana.

API obietnic w specyfikacji ES6 jest całkiem proste. Sprawdza się wystarczająco dobrze w większości prostych przypadków asynchronicznych, a ponadto stanowi dobry punkt wyjścia podczas reorganizowania kodu z piekła wywołań zwrotnych na nieco lepszą postać.

Jednak asynchroniczność jest znacznie bardziej skomplikowana, a same aplikacje mogą mieć pewne wymagania, których spełnienie okazuje się trudne ze względu na ograniczenia obietnic. W kolejnym podrozdziale zajmiemy się wspomnianymi ograniczeniami i potraktujemy je jako motywację do wykorzystania zalet oferowanych przez biblioteki bazujące na obietnicach.

Ograniczenia obietnic

Do wielu szczegółów przedstawionych w tym podrozdziale nawiązałem już we wcześniejszej części rozdziału. Powtarzając pewne informacje, skoncentruję się w szczególności na ograniczeniach obietnic.

Sekwencyjna obsługa błędów

Obsługa błędów w kodzie opartym na obietnicach została szczegółowo omówiona we wcześniejszej części rozdziału. Ograniczenia wynikające z konstrukcji obietnic — w szczególności sposób ich łączenia — mogą bardzo łatwo zastawić pułapki, a błąd w łańcuchu obietnic może przypadkowo zostać bez śladu zignorowany.

Mamy jeszcze jeden aspekt, który należy wziąć pod uwagę w przypadku błędów w kodzie bazującym na obietnicach. Ponieważ łańcuch obietnic to po prostu kolejne połączone ze sobą obietnice, nie mamy więc pewnej jednostki, do której można się odwołać jako jednego *komponentu* łańcucha. Skutkiem jest brak możliwości obserwacji z zewnątrz wszelkich błędów, które mogły wystąpić.

Jeżeli tworzysz łańcuch obietnic niezawierający procedury obsługi błędów, to każdy błąd występujący w łańcuchu będzie propagowany w dół łańcucha aż do jego zaobserwowania (poprzez rejestrację w pewnym miejscu procedury obsługi odrzucenia obietnicy). Dlatego też w tym konkretnych przypadku odwołanie do ostatniej obietnicy w łańcuchu jest wystarczające (`p` w powyższym fragmencie kodu), ponieważ można w niej zarejestrować procedurę obsługi odrzucenia. Następnie wspomniana procedura będzie informowała o wszelkich propagowanych błędach:

```
// foo(..), KROK2(..) i KROK3(..) to funkcje pomocnicze
// zawierające kod bazujący na obietnicach.

var p = foo( 42 )
.then( KROK2 )
.then( KROK3 );
```

Wprawdzie może się to wydawać mylące, ale w powyższym fragmencie kodu `p` nie prowadzi do pierwszej obietnicy w łańcuchu (tej pochodzącej z wywołania `foo(42)`), ale zamiast tego do ostatniej, czyli pochodzącej z wywołania `then( KROK3 )`.

Ponadto żaden krok w łańcuchu obietnic nie przeprowadza samodzielnie obsługi błędów. Oznacza to możliwość zarejestrowania w `p` procedury obsługi odrzucenia obietnicy, która będzie powiadamiana w przypadku wystąpienia jakichkolwiek błędów w łańcuchu:

```
p.catch( handleErrors );
```

Jednak jeśli jakikolwiek krok w łańcuchu będzie miał własną procedurę obsługi błędów (prawdopodobnie ukrytą), to wymieniona procedura `handleErrors` nie zostanie poinformowana o błędzie. Takie rozwiązanie może być oczekiwane przez Ciebie — przecież odrzucenie zostało obsłużone — choć jednocześnie może być *niechciane*. Całkowity brak możliwości poinformowania (o „już obsłużonych" błędach dotyczących odrzucenia) jest w pewnych sytuacjach dość poważnym ograniczeniem.

Praktycznie mamy takie samo ograniczenie jak w przypadku konstrukcji `try-catch`, która może przechwycić wyjątek i po prostu go ukryć. Dlatego też nie mamy do czynienia z ograniczeniem charakterystycznym dla obietnic, ale raczej z *sytuacją*, dla której być może trzeba będzie znaleźć pewne rozwiązanie zastępcze.

Niestety, wielokrotnie nie jest przechowywane odniesienie do kroków pośrednich w łańcuchu obietnic. Bez wspomnianych odniesień nie można przypisać procedur obsługi błędów w sposób pozwalający na ich niezawodną obserwację.

Jedna wartość

Z definicji obietnica ma tylko jedną wartość spełniającą lub jeden powód odrzucenia obietnicy. W prostych przykładach to jest zupełnie wystarczające, natomiast w bardziej skomplikowanych scenariuszach może okazać się czynnikiem ograniczającym.

Tutaj najczęściej udzielaną radą jest przygotowanie opakowania dla wartości (takiego jak obiekt lub tablica) pozwalającego na umieszczenie w nim wielu komunikatów. Wprawdzie tego rodzaju rozwiązanie działa, ale nieustanne pakowanie i rozpakowywanie komunikatów w każdym kroku łańcucha obietnic może być niewygodne i uciążliwe.

Podział wartości

Czasami powyższą sytuację można potraktować jako sygnał, że możesz lub powinieneś podzielić problem na co najmniej dwie obietnice.

Wyobraź sobie istnienie funkcji narzędziowej `foo(..)` asynchronicznie generującej dwie wartości (`x` i `y`):

```
function getY(x) {
    return new Promise( function(resolve,reject){
        setTimeout( function(){
            resolve( (3 * x) - 1 );
        }, 100 );
    } );
}

function foo(bar,baz) {
    var x = bar * baz;

    return getY( x )
    .then( function(y){
        // Opakowanie obu wartości i umieszczenie ich w kontenerze.
        return [x,y];
    } );
}

foo( 10, 20 )
.then( function(msgs){
```

```
    var x = msgs[0];
    var y = msgs[1];

    console.log( x, y );     // 200 599
} );
```

Przede wszystkim zaczynamy od modyfikacji wartości zwrotnej `foo(..)`, aby tym samym uniknąć konieczności opakowania `x` i `y` pojedynczą tablicą przekazywaną później obietnicy. Zamiast tego każdą z wymienionych wartości opakowujemy oddzielną obietnicą:

```
function foo(bar,baz) {
    var x = bar * baz;

    // Zwrot obu obietnic.
    return [
        Promise.resolve( x ),
        getY( x )
    ];
}

Promise.all(
    foo( 10, 20 )
)
.then( function(msgs){
    var x = msgs[0];
    var y = msgs[1];
    console.log( x, y );
} );
```

Czy tablica obietnic to tak naprawdę jest lepsze rozwiązanie niż tablica wartości przekazywanych za pomocą jednej obietnicy? Pod względem syntaktycznym nie będzie to zbyt duże usprawnienie.

Oba przedstawione powyżej podejścia są niezwykle blisko związane z teorią projektu obietnic. W przyszłości będzie można łatwiej przeprowadzić refaktoryzację w celu podziału obliczeń wartości `x` i `y` na oddzielne funkcje. Jest znacznie lepiej, gdy kod wywołujący samodzielnie decyduje o synchronizacji dwóch obietnic — na przykład za pomocą wywołania `Promise.all([ .. ])`, choć na pewno to nie jest jedyne możliwe rozwiązanie — niż gdy tego rodzaje szczegóły są zdefiniowane wewnątrz funkcji `foo(..)`.

Rozpakowanie i rozprowadzenie argumentów

Przypisania `var x = ..` i `var y = ..` nadal można uważać za niewygodne obciążenie. Istnieje możliwość zastosowania pewnej funkcjonalnej sztuczki w funkcji narzędziowej (podziękowania za tę sztuczkę należy złożyć Reginaldowi Braithwaite'owi, którego znajdziesz na Twitterze pod nickiem *@raganwald*):

```
function spread(fn) {
    return Function.apply.bind( fn, null );
}

Promise.all(
    foo( 10, 20 )
)
.then(
    spread( function(x,y){
        console.log( x, y );     // 200 599
    } )
)
```

To już znacznie lepsze rozwiązanie! Oczywiście można też osadzić w funkcji interesujący nas kod i tym samym uniknąć konieczności tworzenia dodatkowej funkcji pomocniczej:

```
Promise.all(
    foo( 10, 20 )
)
.then( Function.apply.bind(
    function(x,y){
        console.log( x, y );     // 200 599
    },
    null
) );
```

Wprawdzie powyższe sztuczki mogą być uznane za użyteczne, ale specyfikacja ES6 oferuje jeszcze lepsze rozwiązanie: destrukturyzację. Polecenie przypisania pozwalające na destrukturyzację tablicy przedstawia się następująco:

```
Promise.all(
    foo( 10, 20 )
)
.then( function(msgs){
    var [x,y] = msgs;

    console.log( x, y );     // 200 599
} );
```

Co najlepsze, specyfikacja ES6 oferuje także możliwość destrukturyzacji za pomocą parametru:

```
Promise.all(
    foo( 10, 20 )
)
.then( function([x,y]){
    console.log( x, y );     // 200 599
} );
```

W ten sposób zastosowaliśmy mantrę „jedna wartość dla jednej obietnicy" i jednocześnie ograniczyliśmy do minimum kod wymagany do obsługi przyjętego rozwiązania.

Więcej informacji na temat form destrukturyzacji w ES6 znajdziesz w innej książce z tej serii, zatytułowanej *ES6 & Beyond*.

Jedno rozwiązanie

Jedną z najbardziej charakterystycznych cech obietnicy jest to, że może być ona rozwiązana tylko jednokrotnie (spełnienie bądź odrzucenie). W wielu sytuacjach asynchronicznych wartość jest pobierana wyłącznie raz, co sprawdza się doskonale.

Jednak istnieje wiele sytuacji asynchronicznych zaliczających się do zupełnie innego modelu — bardziej przypomina zdarzenia i (lub) strumienie danych. Na pierwszy rzut oka nie ma pewności, czy obietnice będą dobrze sprawdzały się w takich przypadkach i czy w ogóle można je tutaj zastosować. Bez przygotowania znacznej abstrakcji na bazie obietnic obietnice zupełnie nie sprawdzają się w sytuacji wymagającej obsługi rozwiązania składającego się z wielu wartości.

Wyobraź sobie następującą sytuację — chcesz wykonać sekwencję kroków asynchronicznych w odpowiedzi na bodziec (na przykład zdarzenie), który występuje wielokrotnie, tak jak w przypadku kliknięcia przycisku.

Przedstawione poniżej rozwiązanie prawdopodobnie nie działa w oczekiwany przez Ciebie sposób:

```
// Funkcja click(..) dołącza zdarzenie "click" do elementu modelu DOM.
// Funkcja request(..) zawiera bazujący na obietnicach kod do wykonywania żądań Ajax.

var p = new Promise( function(resolve,reject){
    click( "#mybtn", resolve );
} );

p.then( function(evt){
    var btnID = evt.currentTarget.id;
    return request( "http://dowolny.adres.url.1/?id=" + btnID );
} )
.then( function(text){
    console.log( text );
} );
```

Powyższe rozwiązanie będzie się sprawdzać tylko wtedy, gdy przycisk zostanie kliknięty wyłącznie jeden raz. W przypadku drugiego kliknięcia przycisku obietnica `p` jest już rozwiązana, a więc drugie wywołanie funkcji `resolve(..)` zostanie zignorowane.

Zamiast stosować powyższe podejście, prawdopodobnie powinieneś odwrócić paradygmat i utworzyć zupełnie nowy łańcuch obietnic dla każdego wywoływanego zdarzenia:

```
click( "#mybtn", function(evt){
    var btnID = evt.currentTarget.id;

    request( "http://dowolny.adres.url.1/?id=" + btnID )
    .then( function(text){
        console.log( text );
    } );
} );
```

Powyższe podejście sprawdza się, ponieważ zupełnie nowa sekwencja obietnic zostanie utworzona dla każdego zdarzenia `click` wywołanego przez kliknięcie przycisku.

Jednak pomijając aspekt estetyczny związany z koniecznością zdefiniowania całego łańcucha obietnic w procedurze obsługi zdarzeń, przedstawione podejście pod pewnymi względami stanowi złamanie idei podziału zadań. Być może bardzo chcesz zdefiniować procedurę obsługi zdarzeń w zupełnie innym miejscu kodu, gdy przygotujesz *odpowiedź* na zdarzenie (łańcuch obietnic). Bez mechanizmów w postaci funkcji pomocniczych takie rozwiązanie będzie trudne do osiągnięcia w przypadku stosowania przedstawionego powyżej podejścia.

Inny sposób obejścia wymienionego wcześniej ograniczenia to przygotowanie pewnego rodzaju komponentu „możliwego do obserwacji”, który następnie można by dołączyć do łańcucha obietnic. Istnieją biblioteki tworzące wspomniane abstrakcje (na przykład *RxJS*), ale te abstrakcje mogą wydawać się na tyle skomplikowane, że całkowicie przesłonią naturę obietnic. Skomplikowana abstrakcja wiąże się z ważnymi kwestiami, które należy rozważyć: czy tego rodzaju mechanizmom można *zaufać* podobnie jak ufamy obietnicom? Do wzorca komponentu „możliwego do obserwacji” powrócimy jeszcze w dodatku B.

Inercja

Jedną z barier uniemożliwiających użycie obietnic we własnym kodzie może być fakt, że obecnie istniejący kod nie został przystosowany do obsługi obietnic. Jeżeli masz ogromną ilość kodu bazującego na wywołaniach zwrotnych, wówczas znacznie łatwiej jest pozostać przy dotychczasowym stylu tworzenia kodu.

„Zmieniająca się baza kodu (opartego na wywołaniach zwrotnych) pozostaje zmienna (z wywołaniami zwrotnymi) aż do chwili, gdy kodem zajmie się sprytny programista, który rozumie sposób działania obietnic".

Obietnice stosują inny paradygmat, a więc podejście do tworzenia kodu może być w pewnych przypadkach jedynie nieco inne, natomiast w innych znacznie odmienne. Musisz być tego świadom, ponieważ możliwość zastosowania obietnic wymaga zmiany doskonale sprawdzającego się dotąd sposobu tworzenia kodu.

Spójrz na poniższy fragment kodu przedstawiający rozwiązanie oparte na wywołaniach zwrotnych:

```
function foo(x,y,cb) {
    ajax(
        "http://dowolny.adres.url.1/?x=" + x + "&y=" + y,
        cb

    );
}

foo( 11, 31, function(err,text) {
    if (err) {
        console.error( err );
    }
    else {
        console.log( text );
    }
} );
```

Czy od razu wiesz, jaki będzie pierwszy krok podczas operacji konwersji powyższego kodu z bazującego na wywołaniach zwrotnych na kod oparty na obietnicach? Wszystko zależy od Twojego doświadczenia. Im większe doświadczenie, tym bardziej komfortowe położenie. Trzeba pamiętać, że obietnice nie zawierają etykiety z dokładnymi instrukcjami, co należy zrobić — nie ma tutaj jednego uniwersalnego rozwiązania — a więc odpowiedzialność za wykonywane kroki spoczywa na Tobie.

Jak już wcześniej wyjaśniłem, zdecydowanie potrzebujemy przeznaczonej do wykonywania żądań Ajax funkcji pomocniczej opartej na obietnicach zamiast na wywołaniach zwrotnych. Wspomnianej funkcji pomocniczej można nadać nazwę `request(..)`. Masz możliwość utworzenia własnej wersji tego rodzaju funkcji pomocniczej, jak to zrobiłem w omawianym przykładzie. Jednak obciążenie związane z koniecznością ręcznego definiowania opartych na obietnicach opakowań dla każdej funkcji bazującej na wywołaniach zwrotnych jeszcze bardziej zmniejsza prawdopodobieństwo, że w ogóle zdecydujesz się przeprowadzić refaktoryzację kodu na postać opartą na obietnicach.

Obietnice nie oferują bezpośredniego rozwiązania dla wymienionego ograniczenia. Jednak większość bibliotek obietnic zawiera pewne funkcje pomocnicze. Jeżeli nie chcesz korzystać z zewnętrznej biblioteki, spójrz na przedstawiony poniżej przykład wspomnianej wcześniej funkcji pomocniczej:

```
// Sprawdzenie bezpieczne dla skryptu typu polyfill.
if (!Promise.wrap) {
    Promise.wrap = function(fn) {
        return function() {
            var args = [].slice.call( arguments );

            return new Promise( function(resolve,reject){
                fn.apply(
                    null,
                    args.concat( function(err,v){
                        if (err) {
                            reject( err );
                        }
                        else {
                            resolve( v );
                        }
                    } )
                );
            } );
        };
    };
}
```

Dobrze, powyższy kod to nieco więcej niż tylko prosta funkcja pomocnicza. Jednak pomimo przerażającego wyglądu wcale nie jest taka skomplikowana, jak można by się spodziewać. Pobiera funkcję oczekującą podania jako jej ostatni parametr wywołania zwrotnego w stylu „najpierw błąd" i zwraca nową funkcję, która automatycznie tworzy obietnicę będącą wartością zwrotną tej funkcji. Zastąpienie wywołania zwrotnego obietnicą i przygotowanie procedur obsługujących spełnienie oraz odrzucenie obietnicy następuje automatycznie.

Zamiast marnować czas na dokładne przedstawienie *sposobu* działania funkcji pomocniczej `Promise.wrap(..)`, lepiej spójrz na przykład jej użycia:

```
var request = Promise.wrap( ajax );

request( "http://dowolny.adres.url.1/" )
.then( .. )
..
```

Uff, to całkiem łatwe!

Wynikiem działania funkcji `Promise.wrap(..)` *nie* jest obietnica, lecz funkcja tworząca obietnicę. W pewnym sensie funkcję tworzącą obietnicę można określić mianem fabryki obietnicy. Dla tego rodzaju komponentu proponuję nazwę „promisory", jako połączenie słów „promise" (ang. — „obietnica") i „factory" (ang. — „fabryka").

Operacja opakowania funkcji oczekującej wywołania zwrotnego w taki sposób, aby stała się funkcją bazującą na obietnicach, jest często określana mianem *liftingu* lub *promisifyingu*. Nie istnieje jednak standardowe pojęcie określające otrzymywaną funkcję (nie licząc określenia „funkcja lifting"), więc uważam, że zaproponowana przeze mnie nazwa „promisory" jest lepsza i lepiej oddaje przeznaczenie funkcji.

„Promisory" to nie jest wymyślone słowo. Takie słowo istnieje i oznacza obietnicę do spełnienia. Dokładnie na tym polega działanie omawianych powyżej funkcji, a więc słowo „promisory" doskonale pasuje do terminologii!

Tak więc funkcja `Promise.wrap(ajax)` powoduje wygenerowanie funkcji typu promisory o nazwie `ajax(..)` wywoływanej przez `request(..)`. Następnie funkcja typu promisory utworzy obietnice dla odpowiedzi Ajax.

Jeżeli wszystkie funkcje byłyby typu promisory, to nie istniałaby konieczność ich samodzielnego tworzenia. Dlatego też potrzebę wykonania tego dodatkowego kroku można uznać za powód do wstydu. Przynajmniej wzorzec opakowania jest (najczęściej) powtarzalny, co pozwala na jego umieszczenie w funkcji pomocniczej `Promise.wrap(..)`, jak pokazałem wcześniej, i tym samym ułatwienie sobie zadania tworzenia kodu.

Powracamy teraz do wcześniejszego przykładu. Funkcji typu promisory potrzebujemy dla wywołania zarówno `ajax(..)`, jak i `foo(..)`:

```
// Utworzenie funkcji typu promisory dla wywołania ajax(..).
var request = Promise.wrap( ajax );

// Refaktoryzacja foo(..). Z zewnątrz nadal powinna być oparta
// na wywołaniach zwrotnych w celu zachowania zgodności z innymi
// komponentami programu. Natomiast wewnątrz używamy wywołania
// request(..) opartego na obietnicach.
function foo(x,y,cb) {
    request(
        "http://dowolny.adres.url.1/?x=" + x + "&y=" + y
    )
    .then(
        function fulfilled(text){
            cb( null, text );
        },
        cb
    );
}

// Teraz na potrzeby omawianego kodu tworzymy
// funkcję typu promisory dla wywołania foo(..).
var betterFoo = Promise.wrap( foo );

// Następnie używamy przygotowanej funkcji typu promisory.
betterFoo( 11, 31 )
.then(
    function fulfilled(text){
        console.log( text );
    },
    function rejected(err){
        console.error( err );
    }
);
```

Oczywiście podczas refaktoryzacji funkcji `foo(..)` w celu użycia nowej funkcji typu promisory o nazwie `request(..)` możemy po prostu funkcję `foo(..)` również zdefiniować jako funkcję typu promisory, zamiast pozostawać przy funkcji opartej na wywołaniach zwrotnych i tym samym konieczności użycia kolejnej funkcji typu promisory o nazwie `betterFoo(..)`. Decyzja zależy po prostu od tego, czy funkcja `foo(..)` musi pozostać oparta na wywołaniach zwrotnych, aby zapewnić zgodność z pozostałymi fragmentami kodu.

Spójrz na poniższy fragment kodu:

```
// Teraz funkcja foo(..) również jest typu promisory, ponieważ
// jest delegatem dla wywołania request(..) typu promisory.
```

```
function foo(x,y) {
    return request(
        "http://dowolny.adres.url.1/?x=" + x + "&y=" + y
    );
}
foo( 11, 31 )
.then( .. )
..
```

Wprawdzie obietnice w specyfikacji ES6 nie są dostarczane wraz z natywnymi funkcjami pomocniczymi, takimi jak funkcje typu promisory i umożliwiające opakowanie, ale dostarcza je większość bibliotek, a ponadto możesz je utworzyć samodzielnie. Ostatecznie to ograniczenie obietnic można obejść bez zbyt dużych problemów (zwłaszcza w porównaniu z problemami związanymi z piekłem wywołań zwrotnych!).

Obietnicy nie można anulować

Gdy obietnica zostanie utworzona i zostaną zarejestrowane procedury obsługi odpowiedzialne za jej spełnienie lub odrzucenie, żaden czynnik zewnętrzny nie może zatrzymać jej działania.

Wiele bibliotek dostarczających obietnice zawiera możliwość anulowania obietnic, choć to jest koszmarny pomysł! Wielu programistów żałuje, że obietnice nie zostały natywnie wyposażone w możliwość ich anulowania. Jednak problem z anulowaniem obietnicy polega na tym, że wspomniana możliwość pozwoliłaby konsumentowi lub obserwatorowi obietnicy mieć wpływ na obserwację tej samej obietnicy przez innego konsumenta. W ten sposób doszłoby do utraty zaufania do przyszłej wartości (dotyczącej braku możliwości jej zmiany z zewnątrz), a ponadto stanowiłoby to ucieleśnienie antywzorca *action at a distance*[3]. Niezależnie od tego, jak bardzo użyteczna wydaje się możliwość anulowania obietnicy, tak naprawdę może doprowadzić do takiego samego koszmaru, z jakim spotykamy się w przypadku wywołań zwrotnych.

Spójrz na poniższy scenariusz wyczerpania limitu czasu dla obietnicy, z którym spotkaliśmy się już wcześniej.

```
var p = foo( 42 );

Promise.race( [
    p,
    timeoutPromise( 3000 )
] )
.then(
    doSomething,
    handleError
);

p.then( function(){
    // Nadal występuje, nawet w przypadku upływu czasu przeznaczonego dla obietnicy :-(
} );
```

Dla obietnicy `p` „wyczerpanie limitu czasu" to czynnik zewnętrzny, a więc działa ona nadal, choć prawdopodobnie oczekujemy zupełnie odmiennego zachowania.

Jednym z rozwiązań jest inwazyjne zdefiniowanie wywołań zwrotnych odpowiedzialnych za rozwiązanie obietnicy:

[3] *http://www.wikiwand.com/en/Action_at_a_distance_(computer_programming)*

```
var OK = true;

var p = foo( 42 );

Promise.race( [
    p,
    timeoutPromise( 3000 )
    .catch( function(err){
        OK = false;
        throw err;
    } )
] )
.then(
    doSomething,
    handleError
);

p.then( function(){
    if (OK) {
        // Tylko wtedy, gdy nie nastąpiło przekroczenie limitu czasu! :-)
    }
} );
```

Powyższy kod nie należy do eleganckich, jednak działa, choć daleko mu do idealnego. Ogólnie rzecz biorąc, powinieneś starać się unikać tego rodzaju scenariuszy.

Jeśli jednak nie można tego uniknąć, wtedy brak elegancji rozwiązania powinien być wskazówką, że *anulowanie* to funkcjonalność na wyższym poziomie abstrakcji zbudowanej na bazie obietnic. Zachęcam Cię do tego, byś zamiast samodzielnie opracowywać sztuczki, zainteresował się bibliotekami obietnic.

Opracowana przeze mnie biblioteka abstrakcji obietnic o nazwie *asynquence* oferuje wspomnianą funkcjonalność, a także funkcję `abort()` pozwalającą na anulowanie sekwencji. Wymienione oraz pozostałe możliwości biblioteki przedstawię dokładniej w dodatku A.

Pojedyncza obietnica naprawdę nie stanowi mechanizmu kontroli przepływu działania programu (przynajmniej nie w sensie mającym znaczenie dla aplikacji), a dokładnie do tego odwołuje się *anulowanie*. Dlatego też anulowanie obietnicy to koszmarny pomysł.

Z drugiej strony łańcuch obietnic jako całość — tutaj lubię używać nazwy sekwencja — *stanowi* wyrażenie kontroli przepływu działania programu, a tym samym jest odpowiedni do anulowania zdefiniowanego na tym poziomie abstrakcji.

Pojedyncza obietnica nie powinna być możliwa do anulowania, choć sensowne jest zachowanie możliwości anulowania *sekwencji*, ponieważ w przeciwieństwie do obietnicy, sekwencji nie przekazujesz jako jednej niemodyfikowalnej wartości.

Wydajność obietnicy

To konkretne ograniczenie jest proste i jednocześnie skomplikowane.

Porównanie liczby elementów używanych podczas wykonywania prostego zadania asynchronicznego za pomocą wywołań zwrotnych i liczby elementów wykorzystywanych podczas użycia obietnic jasno pokazuje, że obietnice wymagają znacznie większej aktywności. To może skłonić

do wniosku, że obietnice charakteryzują się mniejszą wydajnością. Powróć na chwilę do prostej listy gwarancji zaufania oferowanego przez obietnicę i porównaj to z rozwiązaniem ad hoc, które tworzysz na bazie wywołań zwrotnych, aby uzyskać ten sam poziom ochrony.

Więcej pracy do wykonania i więcej komponentów do ochrony oznacza, że obietnice *są* wolniejsze w porównaniu z czystymi i pozbawionymi zaufania wywołaniami zwrotnymi. To jest oczywiste i prawdopodobnie proste do zrozumienia.

Być może zastanawiasz się, o ile wolniejsze będą obietnice. Cóż, to niezwykle trudne pytanie, na które nie ma jednoznacznej odpowiedzi.

To przypomina porównywanie jabłek i pomarańczy, więc prawdopodobnie mamy do czynienia z niewłaściwym pytaniem. Powinieneś raczej sprawdzić, czy system wywołań zwrotnych z ręcznie wprowadzonymi zabezpieczeniami odpowiadającymi stosowanym w obietnicach będzie szybszy niż implementacja obietnic.

Jeżeli obietnice faktycznie mają uzasadnione ograniczenie w kwestii wydajności, wynika ono bardziej z oferowanego poziomu zaufania — nie masz wyboru, które mechanizmy ochrony chcesz zastosować; zawsze otrzymujesz je wszystkie.

Niezależnie od tego, jeśli przyjmiemy ogólne założenie o *nieco mniejszej wydajności* obietnic względem odpowiadających im wywołań zwrotnych pozbawionych zaufania — oczywiście zakładając istnienie miejsc, w których przypadku można pogodzić się z brakiem zaufania — to czy oznacza to, że obietnic należy unikać? Czy w aplikacji naprawdę liczy się jedynie kod wykonywany najszybciej, jak to możliwe, bez względu na wszystko inne?

Pytanie retoryczne: jeżeli wymagasz kodu działającego jak najszybciej, to czy JavaScript jest odpowiednim językiem dla tego rodzaju zadań? Oczywiście JavaScript może być zoptymalizowany do wydajnego działania aplikacji, o czym przekonasz się w rozdziałach 5. i 6. Czy jednak obsesja na punkcie wydajności i niechęć do pogodzenia się z jej minimalnym spadkiem na skutek użycia obietnic pomimo ich niewątpliwych zalet naprawdę są właściwe?

Inna drobna kwestia polega na tym, że obietnice *wszystko* wykonują asynchronicznie. Oznacza to, że pewne natychmiast (synchronicznie) kończone kroki nadal wstrzymują wykonanie następnego kroku do zadania (patrz rozdział 1.). Dlatego też sekwencja zadań obietnic może być zakończona nieco później niż ta sama sekwencja oparta na wywołaniach zwrotnych.

Oczywiście pytanie, które trzeba sobie w tym miejscu zadać, brzmi: czy potencjalnie niewielki spadek wydajności jest warty wszystkich pozostałych korzyści oferowanych przez obietnice i omówionych w rozdziale?

Według mnie w praktycznie wszystkich przypadkach, w których istnieje obawa o wydajność oferowaną przez obietnice, tak naprawdę mamy do czynienia z antywzorcem odrzucania oferowanych przez nie korzyści (wynikających z zaufania do obietnic i możliwości ich łączenia) przez uniknięcie ich stosowania.

Powinieneś domyślnie używać obietnic w kodzie, a następnie przeprowadzać profilowanie i analizę ścieżek działania aplikacji o znaczeniu krytycznym. Czy obietnice naprawdę stanowią wąskie gardło, czy jednak tylko czysto teoretycznie zmniejszają wydajność aplikacji? Tylko wówczas, mając

w ręku prawidłowo wykonane testy wydajności (patrz rozdział 6.), możesz odpowiedzialnie i rozważnie uwzględnić obietnice w zidentyfikowanych obszarach o znaczeniu krytycznym.

Wprawdzie obietnice są nieco wolniejsze w działaniu, ale w zamian oferują dużo większy poziom zaufania, przewidywalność braku wystąpienia efektu Zalgo i możliwość łączenia obietnic ze sobą. Być może prawdziwe ograniczenie nie wiąże się z wydajnością obietnic, ale z brakiem umiejętności wykorzystania oferowanych przez nie korzyści?

Podsumowanie

Obietnice są wspaniałe. Używaj ich. Rozwiązują problem związany z *odwróceniem kontroli* stanowiący prawdziwe utrapienie w przypadku kodu bazującego tylko na wywołaniach zwrotnych.

Obietnice nie oznaczają całkowitego pozbycia się wywołań zwrotnych, a jedynie przekierowują sterowanie nimi do godnego zaufania mechanizmu umieszczonego między aplikacją i innym narzędziem.

Łańcuch obietnic zaczyna stanowić lepszy (choć bez wątpienia niedoskonały) sposób synchronicznego wyrażenia asynchronicznej kontroli przepływu działania programu. Dzięki temu możemy znacznie lepiej zaplanować i obsługiwać asynchronicznie działający kod. Istnieje jeszcze lepsze rozwiązanie *tego* problemu, które poznasz w rozdziale 4.!

Skorowidz

U

W

Z

SPRAWDŹ, JAKIE ZAGADKI KRYJE W SOBIE STARY, DOBRY JS!

Istnieje wiele podręczników do nauki JavaScriptu. Większość z nich nie wyczerpuje trudniejszych i bardziej zaawansowanych zagadnień, których zrozumienie — choć wymaga wysiłku — jest warunkiem osiągnięcia prawdziwej biegłości w tym języku. JavaScript jest jednym z przystępniejszych języków programowania i można go używać, znając jedynie podstawy. Równocześnie jednak ten łatwy i zachęcający język zawiera wiele zaawansowanych, złożonych mechanizmów, których stosowanie w praktyce rozszerzy możliwości programisty w zadziwiający sposób. Szkoda, że tak niewielu programistów stara się dogłębnie poznać JavaScript!

Niniejsza książka jest częścią serii w całości poświęconej temu językowi. Założeniem autora było skupić się właśnie na tych głębszych aspektach języka JavaScript i wnikliwie je przeanalizować, a następnie, bazując na takich solidnych podstawach, pokazać praktyczne zastosowanie opisanych koncepcji. Owszem, JavaScript może być z powodzeniem wykorzystywany bez głębszej znajomości, jednak prawdziwą biegłość i kontrolę nad swoim kodem uzyskasz dopiero po zrozumieniu kilku trudniejszych koncepcji, z których część opisano w tej właśnie książce.

Dzięki tej książce:

- zrozumiesz zaawansowane i złożone koncepcje JavaScriptu
- nabierzesz biegłości w programowaniu asynchronicznym w języku JavaScript
- nauczysz się stosować obietnice JavaScript i wykorzystasz je do pisania asynchronicznych API
- będziesz wykorzystywać generatory do wyrażania asynchroniczności w sposób sekwencyjny i wyglądający na synchroniczny
- dowiesz się, w jaki sposób zoptymalizować wydajność na poziomie programu za pomocą wątków roboczych, SIMD i stylu asm.js
- poznasz nieocenione zasoby i techniki przeznaczone do przeprowadzania testów jednostkowych oraz dostrajania wyrażeń i poleceń

KYLE SIMPSON

— jest Teksańczykiem, propagatorem Open Web i wielkim pasjonatem wszystkiego, co związane z językiem JavaScript. Ma dar przekazywania wiedzy, a przy tym zaraża entuzjazmem. Pisze książki, prowadzi warsztaty, występuje na konferencjach o tematyce technicznej oraz pozostaje aktywnym członkiem społeczności OSS.

Helion

40876 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

ISBN 978-83-283-2172-4

9 788328 321724

cena: 39,90 zł

O'REILLY®