**UWAGA:**

Pierwsze sekcje tego szkolenia dotyczą ogólnie przetwarzania wielowątkowego i multiprocessingu. Informacje dotyczące multiprzetwarzania w JavaScript znajdziesz od sekcji nazwanej **“Wielowątkowość w JavaScript”.**

**Czym jest wielowątkowość i multiprzetwarzanie?**

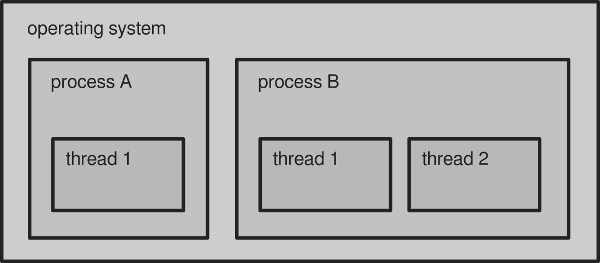
W trakcie pracy na komputerze nie wyobrażamy sobie, abyśmy nie mieli możliwości uruchamiania wielu programów równocześnie. Założę się, że nierzadko robisz kilka czynności równocześnie, np. czytasz to szkolenie, w tle słuchasz muzyki i w kolejnym oknie masz uruchomionego Facebook’a.

Podchodząc do tego bardziej technicznie - każdy program to tak naprawdę **oddzielny proces** uruchamiany na odpowiednich **rdzeniach procesora**.

Każdy z tych procesów składa się z **wątków**.

Wątki realizują podzadania danego programu/procesu (np. abyś mógł słuchać muzyki z Spotify, konieczne jest, aby poprawnie działał wątek odpowiedzialny za transfer danych między serwerem a aplikacją oraz inny wątek służący do obsługiwania głośnika komputera).   
Takie właśnie funkcjonowanie i współpracowanie ze sobą wielu wątków równocześnie w ramach jednego procesu, nazywamy multithreadingiem (wielowątkowością).

Każdy proces **musi posiadać chociaż jeden wątek**, który jest uruchamiany na początku działania programu i nazywamy go **wątkiem głównym**.



**Podział na procesy i wątki**

**Różnica między procesem a wątkiem**

Zasadnicza i główna różnica między tymi dwoma tworami sprowadza się do tego, że każdy proces ma przydzielony **na własność pewien obszar pamięci**. Domyślnie obszar ten (choć można to zmienić dzięki technikom **inter-process communication - IPC**) nie może być współdzielony z innym procesem. I tak na przykład zasoby aplikacji, z której słuchasz muzyki nie mogą być wykorzystywane przez przeglądarkę internetową i vice versa.

Natomiast wątki należące do jednego procesu **współdzielą między sobą zasoby** i mogą wymieniać informacje. Z racji, że wątki wchodzą w skład danego procesu, są one nazywane również **lekkimi procesami** (ang. lightweight processes).

Operacje jakie realizowane są w ramach pojedynczego programu, rozkładane są na poszczególne wątki i dzięki temu możemy otrzymywać **efekt współbieżnego** działania uruchomionych programów.

W przeciwnym razie, gdyby wątki nie istniały, każda pojedyncza instrukcja aplikacji musiałaby być uruchamiana jako odrębny proces, co znacznie utrudniłoby pracę z zasobami.

**Współbieżność vs równoległość**

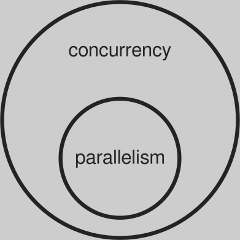
Tak jak już wspomniałem, proces jest podzielony na wątki, aby możliwe było przyspieszenie działania programu. Dzięki temu teoretycznie mamy możliwość rozłożenia funkcjonalności procesu w taki sposób, że operacje programu będą iluzorycznie wykonywały się równolegle (dlaczego iluzorycznie? czytaj dalej!)   
Wyobraź sobie, że program renderujący film może tak podzielić dane kawałki filmu, że jego poszczególne części będą przetwarzane przez różne wątki.

Dzięki temu, w momencie gdy standardowe renderowanie filmu zajęłoby 1 h, to dzieląc program na 2 równoległe wątki, renderowanie **mogłoby** zająć 30 minut. Mówię mogłoby, bo w praktyce nie jest tak kolorowo.

Musisz bowiem wiedzieć, że nigdy nie masz 100% gwarancji, że wątki będą wykonywane równolegle w tym samym czasie.

Procesor z pewnych ograniczeń często **pozoruje zrównoleglenie** operacji.

Wtedy mówimy o tzw. **współbieżności (ang. concurrency)**. Może to wydawać się nieco mylące, ale musisz wiedzieć, że **równoległość to nie współbieżność**! To, czy program faktycznie wykona się w pełni równolegle zależy od niskopoziomowych czynników architektury procesora, nad którymi schylimy się za chwilę.



***Współbieżność vs równoległość***

**Kiedy faktycznie mamy do czynienia z równoległością?**

Aby procesor faktycznie był w stanie wykonywać czynności w pełni równolegle, musi on posiadać więcej niż jeden rdzeń ([**rdzeń fizyczny**](https://pl.wikipedia.org/wiki/Rdze%C5%84_procesora) lub [**wirtualny**](https://forum.pcformat.pl/Rdzenie-Wirtualne-t)).

Jak się okazuje bowiem, rdzeń procesora jest w stanie przetwarzać tylko **jedną operację równocześnie**.

Nie oznacza to jednak, że na jednym rdzeniu CPU może być uruchomiony tylko jeden proces. Na jednym rdzeniu fizycznym możemy odpalać większą ilość programów, ale ich wątki wykonywane będą **współbieżnie**, **nie równolegle**.

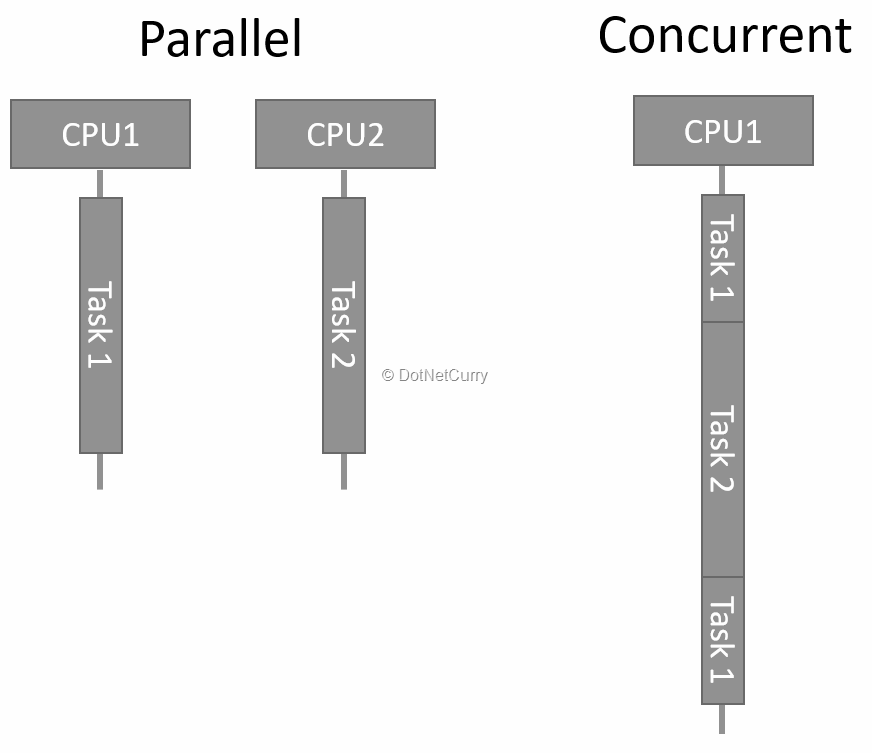
Współbieżność wykonywania polega na uruchamianiu każdego wątku w pętli na określony czas. Jądro systemu kontroluje, ile tzw. czasu procesora ma zostać przydzielone na wykonanie wątku w pojedynczym cyklu. Powiem więcej - mimo że, dane wątki różnych procesów działają na jednym rdzeniu w pętli, to i tak użytkownik ma wrażenie ich równoległego działania.

Wynika to z tego, że przydzielany czas procesora na wykonanie operacji, jest na tyle krótki, że układ nerwowy człowieka nie potrafi dostrzec jakichkolwiek przeskoków między wątkami.

Dla lepszego zobrazowania, mając uruchomioną na tym samym rdzeniu, np. aplikację odtwarzającą muzykę i inną - wyświetlającą strony internetowe, następuje tak szybkie przeskakiwanie między wykonywaniem tych programów, że użytkownik odnosi wrażenie, jakby **wykonywały się one równolegle** (a wykonują się tak naprawdę współbieżnie).

Tak więc może to wyglądać następująco (zakładając, że procesy uruchomione są na jednym rdzeniu):

* w pojedynczym cyklu procesora, 100 ms wykonuje się aplikacja od muzyki, po tym czasie jest ona wstrzymywana i na kolejne 100 ms uruchomiona jest aplikacja wyświetlająca strony internetowe. I tak w kółko. Przez to, że czas nieustannego wykonywania tych pojedynczych programów jest tak krótki, użytkownik ma wrażenie, jakby procesy te wykonywały się faktycznie w pełni równolegle.



**UWAGA**:

CPU1, CPU2 to rdzenie procesora.

**Problemy wielowątkowego przetwarzania**

Jak już wiemy, wątki mogą dzielić ten sam obszar pamięci. Dzięki temu możemy łatwo integrować ze sobą działanie kilku wątków działających w obrębie jednego procesu.

Problem jednak pojawia się w momencie, gdy dwa wątki chcą skorzystać z tego samego zasobu pamięci w tym samym czasie. Przykładem może być sytuacja, gdy jeden wątek **odczytuje dane z pamięci**, a drugi równocześnie **coś do niej zapisuje**.

Wówczas mówimy o problemach:

* **Data race** - występuje, gdy pierwszy wątek modyfikuje pewien obszar pamięci, a drugi chce równocześnie coś z niego odczytać.   
  Jeżeli wątek zapisujący nie zakończy swojej pracy przed rozpoczęciem procesu czytania, to zostaną pobrane naruszone dane.
* **Race condition** występuje, gdy wątki wykonują na danym zasobie pamięci operacje w nieustalonej kolejności. Chodzi o przypadek, gdy, np. pewien wątek **pobierający informacje z bazy danych** będzie wykonywał się przed wątkiem, **przetwarzającym te dane**.   
  Wówczas kolejność operacji będzie nieprawidłowa (jak mają zostać przetworzone dane, gdy nie zostały jeszcze w pełni pobrane z bazy?).

Dlatego, aby zabezpieczyć się przed wyżej wymienionymi zagrożeniami i niepożądanymi działaniami, musimy tworzyć programy, tzw. **thread safe** lub upewnić się, że korzystamy z bibliotek, które właśnie to bezpieczeństwo zapewniają.

**Rozwiązanie powyższych problemów**

Zarówno data-race jak i race condition to rzeczywiste problemy w technologii informatycznej. Niegdyś, gdy efektywność wielowątkowości była dość prymitywna, stanowiła ona poważne zagrożenie dla życia ludzi.   
Nieraz mówi się o Therac-25, czyli maszynie służącej do radioterapii, której to właśnie błędy oprogramowania wynikające z niewłaściwego działania procesów spowodowały śmierć kilku pacjentów.

Obecnie możliwymi rozwiązaniami dla powyższych problemów są:

* **synchronizacja** - synchronizując wykonanie kilku wątków, zapewniamy, że tylko jeden będzie wykonywał się w danym czasie. Dzięki temu zabezpieczamy się przed równoczesnym operowaniem na danym zasobie pamięci.
* **operacje atomowe** operacje będące potencjalnym zagrożeniem dla bezpieczeństwa przetwarzania danych, możemy czasami ustawić jako atomowe, przez co dane będą niepodzielne i niemożliwe będzie ich naruszenie przez 2 wątki równocześnie
* **dane niemutowalne** zagrożenie w przetwarzaniu wielowątkowym jest niwelowane w momencie, gdy wątki mogą tylko czytać dane, bez możliwości ich modyfikowania.   
  Nie wiem, czy pamiętasz, ale zapewnianie niemutowalności danych jest główną filozofią programowania funkcyjnego. Dlatego też często mówimy o tym paradygmacie jako wątkowo-bezpiecznym.

**Wielowątkowość w JavaScript**

Przechodząc już konkretnie do JS-a oraz silników przeglądarek… Powinieneś wiedzieć, że JavaScript jest językiem jednowątkowym (AFAIK). Uzasadnienie jest dość logiczne, ponieważ współbieżne generowanie komponentów na tej samej stronie mogłoby spowodować wiele problemów i nieprzewidzianych zachowań.

To, że kod jest jednowątkowy nie wyklucza możliwości wykonywania kodu w sposób asynchroniczny. Dzięki temu zyskujemy możliwość nieblokowania strony w momencie, gdy chcemy, np. załadować dany element czy skrypt. Aby to realizować wykorzystamy **promises** pojawiające się w JS, ale również **funkcje async**. Operacje asynchroniczne są bowiem realizowane przy użyciu, tzw. **Event Queue**. Kolejka ta wykonuje operacje po tym jak wątek główny zakończy przetwarzanie - dzięki temu asynchroniczne operacje nie opóźnią wykonywania kluczowych dla strony operacji.

Zacznijmy od zrozumienia, czym jest twór **Promise** i jak wykorzystywać go w swoich programach.

**Promise**

Aby móc dobrze i w pełni zrozumieć, jak działa i z czym związany jest omawiany twór, zacznijmy od wyróżnienia następujących tworów w języku:

* **Kod produkujący** - to kod, który wykonuje określone operacje, np. nawiązuje połączenie z określonym API i pobiera odpowiednie informacje. To wszystko dzieje się po to, aby móc ostatecznie przekazać efekty swojej pracy do kodu konsumującego.
* **Kod konsumujący** - to ta część kodu, do której przekazujemy rezultat producera.
* **Promise** - to specjalny obiekt, który łączy kod konsumujący i produkujący. Dzięki promise-owi, producer może przekazać wartości do konsumera, a konsumer jest w stanie je odczytać.

**Przykład z życia wzięty**

Aby lepiej zrozumieć zasadę działania kodu produkujące, konsumującego i całych promise-ów, przeanalizuj poniższy przykład z życia:

Jesteś klientem ekskluzywnej restauracji i ubrany w garnitur i lśniące pantofle, z włosami równo zaczesanymi do tyłu (przesadziłem? :) ), zamawiasz u kelnera swoje danie. Jednak ze względu na dużą unikalność potraw, istnieje pewna niepewność, że kucharz aktualnie nie ma potrzebnych składników do przyrządzenia potrawy. Wówczas po otrzymaniu odpowiedniej informacji, musisz zareagować na taki brak dostępności potrawy i wybrać inną. Istnieje jednak też niemała szansa, że wszystko przebiegło zgodnie z oczekiwaniami, a potrawa zostanie normalnie przyrządzona i kelner z uśmiechem na ustach przyniesie ją wprost do Twojego stolika.

Zapytasz teraz, jak to się ma do programowania i wykorzystania promise-ów w kodzie. Już odpowiadam! Zauważ, że w przytoczonym przykładzie możemy łatwo wyróżnić część **logiki produkującej** (kod produkujący) oraz **logikę konsumującą** (kod konsumujący). Tą pierwszą stanowią czynności polegające na zamówieniu (wywołaniu producera) dania oraz czekaniu aż zostanie zwrócony odpowiedni rezultat (danie jest dostępne/nie jest dostępne). Sam natomiast proces sprawdzenia dostępności i przygotowania dania jest częścią logiki produkującej (która zwraca odpowiedni wynik do klienta - wspomnianego kodu konsumującego).

Zauważ, że konsumer zawsze będzie czekał na efekt producera i dopiero po otrzymaniu odpowiedniego rezultatu podejmował kroki (czy przystąpić do konsumpcji posiłku czy zamówić inne danie)?

Dodatkowo, w momencie, gdy klient będzie czekał na odpowiedź od obsługującego go klienta (producera), może zająć się współbieżnymi czynnościami. Jak, na przykład, rozmowa z żoną o tym, jak spędziła dzień, czy co u ich dzieci.

Właśnie tak działają promise-y, a powyżej widzisz ich barwne przedstawienie w rzeczywistości. Pozwalają one porządkować kod, aby określone czynności były uruchamiane synchroniczne - w określonej kolejności, równocześnie nie blokując stanu całej aplikacji i pozwalają na uruchomienie i wykonanie innych czynności (których promise nie dotyczy).

**Jak tworzyć Promise?**

Aby stworzyć obiekt Promise-a, należy wykorzystać poniższy syntax konstruktora:

| let promise = new Promise(function(resolve, reject) {  // executor (kod produkujący)  }); |
| --- |

Ciało umieszczone w ciele obiektu Promise jest kodem produkującym. Kiedy taki obiekt jest tworzony, następuje automatyczne wywołanie executora. W rezultacie otrzymamy odpowiedni wynik, który będzie mógł zostać przekazany do konsumera (czyli odnosząc się do wcześniej przytoczonej analogii - kelner będzie mógł poinformować klienta o stanie zamówienia).

Parametry resolve i reject są parametrami domyślnymi i automatycznie są dostarczane przez JavaScript.

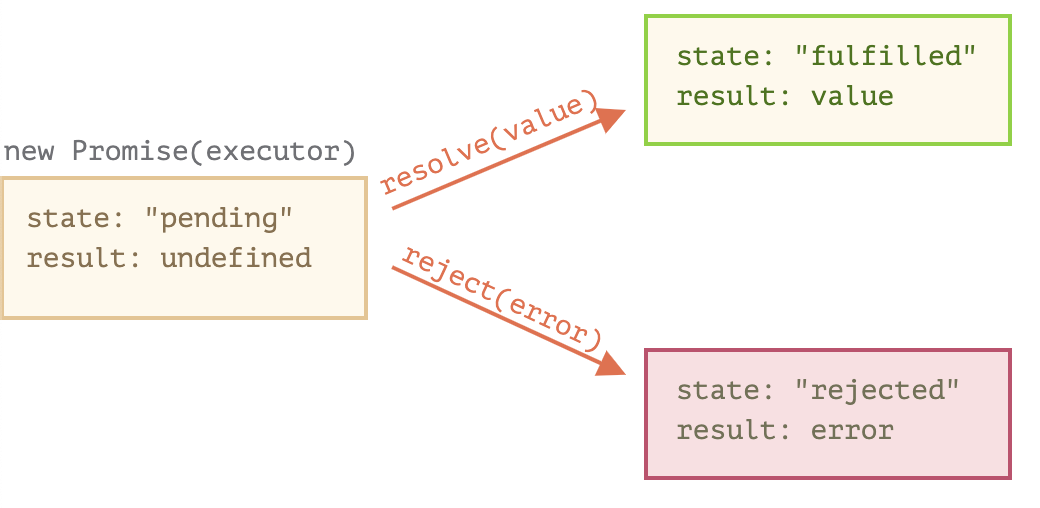
Parametry te są callbackami, które wywoływane są w zależności od sytuacji:

* resolve(data) - jest wywoływany, jeżeli czynności wewnątrz executora zakończą się **sukcesem**
* reject(error) - jest wywoływany, jeżeli czynności wewnątrz executora zakończą się **błędem**.

Podsumowując zatem - po utworzeniu promise-a, następuje automatyczne wywołanie executera. Na końcu kończy on swoją pracę i wywołuje callback resolve, kiedy wszystko zakończyło się sukcesem lub reject, gdy nastąpił błąd.

Oprócz tego, że promise automatycznie wywołuje executor, to tak utworzony obiekt posiada również atrybuty, do których możemy się odwoływać:

* state - początkowo przyjmuje wartość “pending”, następnie jest określany jako “fulfilled”, gdy callback resolve zostanie wywołany lub “rejected” w przypadku wywołania “reject”.
* result — początkowo undefined, następnie jego wartość jest zmieniana na value, gdy wywołujemy resolve(value) lub error, gdy wywołujemy reject(error).



*Źródło medium.com*

**Jak wywoływać Promise i handlować stanami?**

Wiesz już, że z każdym obiektem promise możemy powiązać dwa rezultaty: resolve lub error. Przekazywane są one do funkcjonalności kodu, która ma zostać wywołana po zakończeniu działania kodu produkującego (do konsumera). Spójrz poniżej, jak wygląda przykład obsługiwania takiego konsumera.

| promise.then(  result => ..., // ciało uruchamiane, gdy promise zakończy się sukcesem  error => ... // ciało uruchamiane, gdy promise zakończy się błędem ); |
| --- |

Aczkolwiek to wszystko to wciąż głównie sucha teoria. Dlatego nie mogę się doczekać, aż dojdziemy do sekcji niżej, w której zaprezentuję pierwszy przykład wykorzystania promise.

**Pierwszy przykład wykorzystania Promise**

Aby dobrze zrozumieć, z czym się wiąże wykorzystanie Promise-ów w praktyce, przeanalizujmy pierwszy prosty przykład. Utworzymy bowiem promise, który będzie w kodzie produkującym wywoływał funkcję setTimeout() i po 1 sekundzie wyświetlał napis “Done” (setTimeout to funkcja, która usypia działanie dowolnej funkcji na określony czas).

Przykład z resolve:

| let promise = new Promise(function(resolve, reject) {  setTimeout(() => resolve("Job done!"), 1000); });  promise.then(  result => alert(result), // wyświetli "done" po 1 sekundzie  error => alert(error) // nie uruchomi się ); |
| --- |

Przykład z reject:

| let promise = new Promise(function(resolve, reject) {  setTimeout(() => reject(new Error("Job failed!")), 1000); });  promise.then(  result => alert(result), // nie uruchomi się  error => alert(error) // wyświetli "Job failed!"  ); |
| --- |

Jeżeli natomiast chcesz obsługiwać tylko i wyłącznie “poprawnie zakończone” działanie egzekutora, możesz zapisać:

| let promise = new Promise(function(resolve, reject) {  setTimeout(() => resolve("Job done!"), 1000); });  promise.then(alert); // wyświetli "Job done" |
| --- |

**catch**

Decydując się natomiast na obsługiwanie tylko i wyłącznie błędów, możemy użyć funkcji catch.

| let promise = new Promise((resolve, reject) => {  setTimeout(() => reject(new Error("Job failed!")), 1000); });  promise.catch(alert); // wyświetli "Job failed!" po 1 sekundzie |
| --- |

W takim przypadku, gdybyśmy postanowili wywołać również resolve(), to z racji tego, iż nie obsługujemy takiego rezultatu, program nie ukaże nam żadnego rezultatu.

**finally**

Finally jest ostatnim przykładem funkcji, którą możemy wywołać na rzecz promise. W callbacku przekazanym do tej funkcji, możemy umieścić operacje, które chcemy, aby były wykonywane **zawsze**, niezależnie od wyniku (niezależnie, czy został zwrócony error czy result). W praktyce, w bloku finally umieszcza się, np. operacje czyszczące pamięć, czy zamykające połączenie, które było konieczne do otwarcia podczas realizacji kodu produkcyjnego promise-a.

Poniżej przedstawię Ci już praktyczniejszy przykład wykorzystania promise-a, wykorzystując wszystkie poznane dotąd zagadnienia.

| function checkMail() {  return new Promise((resolve, reject) => {  if (Math.random() > 0.5) {  resolve('Mail wysłany');  } else {  reject(new Error('Nie udało się wysłać maila'));  }  }); }  checkMail()  .then(console.log)  .catch(console.error)  .finally(() => {  console.log('Operacja zakończona');  }); |
| --- |

Poniżej przedstawię Ci jeszcze inny (aczkolwiek dla mnie mniej elegancki) sposób na obsługiwanie rezultatów zwracanych z promise:

| checkMail()  .then(  (mail) => { console.log(mail); },  (err) => {console.error(err); })  .finally(() => {  console.log('Operacja zakończona'); }); |
| --- |

Zauważ, że to my, programiści, musimy ręcznie definiować, czy w danym momencie zostanie wywołana funkcja resolve czy też reject. Tworząc własnego promise-a, to tylko od nas zależy, kiedy zdecydujemy się na zakończenie wykonywania kodu produkcyjnego i zwrócimy result, a kiedy error.

Oczywiście w JS powszechnym zabiegiem jest wykorzystywanie gotowych promise-ów, gdzie to ich autorzy decydują, jak ma się zachowywać egzekutor. Dlatego przeanalizujmy niżej przykład praktycznego wykorzystania Promise.

**Przykład Promise - loadScript**

Wyobraź sobie, że loadScript() to przykład funkcji, która odpowiada za załadowanie zewnętrznych zasobów do naszego skryptu. Będzie to jeden z przykładów funkcji zwracającej promise. To dlatego, że musimy tutaj jasno sprecyzować kolejność wykonywanych działań. Tzn. nie możemy przystąpić do renderowania elementów na stronie, które wykorzystują zewnętrzną biblioteką, dopóki jej nie załadujemy. Natomiast inne kroki, jak np. wyświetlenie podstawowego tekstu, mogą być wykonywane asynchronicznie w stosunku do ładowania biblioteki.

Dlatego też tutaj ładowanie skryptu idealnie sprawdza się jako promise.

| function loadScript(src) {  return new Promise(function(resolve, reject) {  let script = document.createElement('script');  script.src = src;   script.onload = () => resolve(script);  script.onerror = () => reject(new Error(`Script load error for ${src}`));   document.head.append(script);  }); }  let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js");  promise.then(  script => alert(`${script.src} is loaded!`),  error => alert(`Error: ${error.message}`) );  alert("Loading basic code (that does not require ajax library)"); |
| --- |

**Przykład promise - loadImageAsync**

Poniżej zaprezentuję Ci, jak zaimplementować funkcjonalność asynchronicznego ładowania obrazu.

| function loadImageAsync(url) {  return new Promise(function(resolve, reject) {  var image = new Image();   image.onload = function() {  resolve(image);  };   image.onerror = function() {  reject(new Error('Could not load image at ' + url));  };   image.src = url;  }); }  loadImageAsync('one.png')  .then(function(image) {  console.log('Image loaded', image);  }, function(err) {  console.error('Error loading image', err);  }); |
| --- |

Myślę, że powyższy przykład nie wymaga wielu tłumaczeń. Tak stworzony promise może się przydać, gdy próbujemy w sposób uporządkowany ładować obrazek na stronie, np. gdy nie chcemy, aby ładowanie obrazka blokowało wyświetlanie kolejnych elementów na stronie.

**Promises chaining**

Z obiektem Promise związany jest jeszcze jeden dość istotny aspekt. Otóż .then() możemy wołać niezliczoną ilość razy. Każde kolejne wywołanie, dodaje nową “funkcję subskrybowaną” do listy subskrybcji. Dzięki takiemu rozwiązaniu, możemy uzależniać od siebie kolejne wyniki zwracane jako result.

| new Promise(function(resolve, reject) {   setTimeout(() => resolve(1), 1000);   }).then(function(result) {   alert(result);   return result \* 2;  }).then(function(result) {   alert(result);  return result \* 2;  }).then(function(result) {   alert(result);  return result \* 2;  }); |
| --- |

Zwróć uwagę na to, że alerty wyświetlają coraz większe wartości result - kolejno 1, 2, 4.

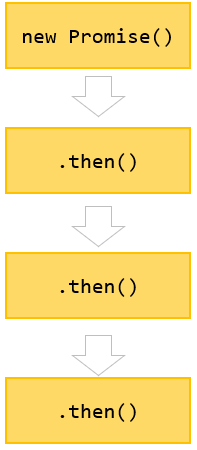
Można więc powiedzieć, że do kolejnych wywołań then(), przekazywane są wartości result zwrócone przez poprzednie wywołania. Teoretycznie rzecz ujmując, then na jednym obiekcie Promise (w sposób łańcuchowy) możemy wywoływać dowolną ilość razy. Mają tutaj na myśli “łancuch”, odnoszę się do przypadku takiego jak wyżej (czyli występuje pojedyncze odwołanie do promise, a później wielokrotnie wywołujemy na nim then).

Miej na uwadze to, iż odwoływanie się do utworzonego promise-a, przez wielokrotne podawanie jego nazwy nie jest już łańcuchowym wywołaniem. Jest to zasadnicza różnica, ponieważ w ten sposób nie otrzymamy efektu takiego jak w przytoczonym wcześniej przykładzie. W takiej sytuacji, result zawsze będzie przyjmował tę samą wartość (równą 1).

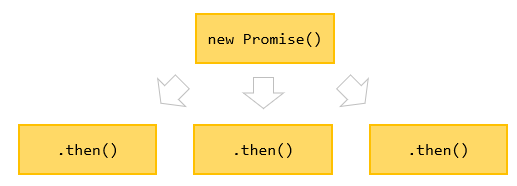
| let promise = new Promise(function(resolve, reject) {  setTimeout(() => resolve(1), 1000); });  promise.then(function(result) {  alert(result); // 1  return result \* 2; });  promise.then(function(result) {  alert(result); // 1  return result \* 2; });  promise.then(function(result) {  alert(result); // 1  return result \* 2; }); |
| --- |

Jest to dość podchwytliwe i wymaga zapamiętania. W pierwszym przykładzie, zwracany wyniki są od siebie uzależnione, w drugim natomiast są niezależne.

Poniżej jeszcze schematy działania łańcuchowego i niezależnego powiązania wywołań then().



Łańcuchowe wywołanie



Niezależne wywołanie

**Przykład praktycznego wykorzystania promise-a i łańcuchowego wywołania**

Ciekawym przykładem, który ukazuje praktyczne wykorzystanie promise-a oraz łańcuchowego wywołania kolejnych rezultatów jest **fetch()**. Funkcja ta służy do asynchronicznego pobierania danych z zewnętrznych zasobów, np. z API.

| fetch('/devs-mentoring-api/courses/students-list.json')  .then(function(response) {  return response.text(); // tutaj otrzymujemy odpowiedź, z której wyciągamy JSON-a ...  })  .then(function(text) { // ... i przekazujemy tutaj  alert(text); // {"name": "Jan", "isStudent": true}  }); |
| --- |

Powyższy przykład możemy zapisać również w nieco krótszej wersji:

| fetch('/devs-mentoring-api/courses/students-list.json')  .then(response => response.text(); // tutaj otrzymujemy odpowiedź, z której wyciągamy JSON-a ...  })  .then(text => alert(text); // {"name": "Jan", "isStudent": true}  }); |
| --- |

**Kolejny Promise w then?**

Łańcuchowe wywołanie nie musi odwoływać się tylko do standardowych operacji. Możemy również z poziomu funkcji wykonywanej w ramach then(), zwracać kolejny obiekt Promise i go obsługiwać.

Załóżmy, że po otrzymaniu JSON-a (tak jak wyżej), będziemy chcieli pobrać UUID użytkownika, którego informacje zostały umieszczone w tym JSONie i przekazać go do kolejnego API - aby otrzymać inne informacje, np. terminarz wszystkich spotkań konkretnego usera.

Aby rozwiązać taki problem, będziemy potrzebowali odwołać się do dwóch niezależnych endpointów API, czyli stworzyć dwa Promise-y. Jeden z nich będzie postaci: **/devs-mentoring-api/courses/student-details/Jan.json**, drugi natomiast **/devs-mentoring-api/calendar/UUID** (gdzie UUID pobierzemy z wcześniej otrzymanego JSON-a).

| fetch('/devs-mentoring-api/courses/student-details/Jan.json')  .then(response => response.json) // wyciągnięcie jsona z response (dane userów)  .then(user => fetch(`/devs-mentoring-api/calendar/${user.uuid}`) // pobranie response zawierającego spotkania  .then(response => response.json) // wyciągnięcie jsona z response (spotkania)  .then(meetings => ...) // tutaj możemy zarządzać spotkaniami |
| --- |

Powyższy przykład pokazuje, że łańcuchowe wywołanie funkcji na Promise może również służyć do tworzenia kolejnego obiektu i niezależne obsługiwanie nowego requestu w sposób asynchroniczny.

**Podsumowanie**

Temat promise-ów (i w zasadzie ogólnie programowania asynchronicznego) nie należy do najłatwiejszych zagadnień. Dlatego postanowiłem, że na zwieńczenie tego szkolenia, dodam krótkie podsumowanie, w którym wylistuję główne zalety płynące z wykorzystania obiektu Promise. Teraz Twoim zadaniem będzie dogłębne zapoznanie się z poruszonymi tutaj zagadnieniami, przećwiczenie tego wszystkiego w praktyce oraz skonfrontowanie zdobytej wiedzy ze swoim mentorem. W ten sposób najlepiej ugruntujesz poznaną wiedzę i doszlifujesz nabyte umiejętności.

| Zalety wykorzystania obiektu Promise:   * Ułatwia nam zarządzanie kolejnością wykonywanych operacji i nieblokowaniem głównego eventu skryptu (Promise użyjemy, gdy przykładowo będziemy chcieli przeprowadzić asynchroniczne ładowanie zasobów) * Możemy obsługiwać dowolną ilość eventów i wywołań kodu produkującego przez dodawanie nowych obiektów do “listy subskrybcji” |
| --- |

**Nakierunkowanie na rekrutację**

W tej sekcji zawrzemy mniej powszechne informacje i sposoby na zarządzanie Promise-ami, o które często nasi kandydaci są pytani na rozmowach kwalifikacyjnych. Przykładami takich zagadnień są:

* Promise.all
* Promise.race
* Promise.any

Zadanie rekrutacyjne:

Często pojawiającym się zagadnieniem na interview w kontekście Promise-ów, jest tzw. “Promise hell” oraz sprawdzenie, jakie sposoby na jego rozwiązanie zna kandydat.

Przeanalizuj poniższy przykład:

| function myPromise(counter){ return new Promise((resolve, reject) => {  setTimeout(() => {  resolve(`MY PROMISE ${counter}`);  }, 2000);  }) }  function callPromise(){  myPromise(1).then(result => console.log(result));   myPromise(2).then(result => console.log(result));  myPromise(3).then(result => console.log(result));  console.log('LAST LOG'); } callPromise(); |
| --- |

Jakiego wyniku się spodziewasz? Pierwsza myśl podpowiada nam, że będzie to:

| MY PROMISE 1  MY PROMISE 2  MY PROMISE 3  LAST LOG |
| --- |

Jednak niekoniecznie! Potencjalnie możliwym do pojawienia się wynikiem może być:

| LAST LOG  MY PROMISE 1  MY PROMISE 3  MY PROMISE 2 |
| --- |

Wynika to z asynchronicznej natury wykonywanych operacji. Wywołując w sposób łańcuchowy .then() na obiekcie Promise, nie mamy pewności, że eventy zawsze wykonają się w tej samej kolejności. Dlatego też określamy ten “problem” jako Promise Hell. Rozwiązanie tego problemu nie jest wcale skomplikowane - wystarczy użyć async/await, które poznasz w następnym szkoleniu.

Więcej o Promise Hell znajdziesz tutaj: <https://medium.com/@ringgagustavino/avoid-promise-hell-6d9a518aea8c>

**Ciekawostka**

Jeżeli zastanawiasz się, jaka jest relacja między ciągłym zwiększaniem procesorów a szybkością wykonywania algorytmu (dzięki zrównoleglenia operacji), to polecamy zapoznać się i poznać więcej informacji dotyczących **Prawa Amdahla** - [Prawo Amdahla - Obliczenia równoległe](http://mateuszmigot.pl/blog/prawo-amdahla/).