**8. Wprowadzenie do JavaScriptu część 1**

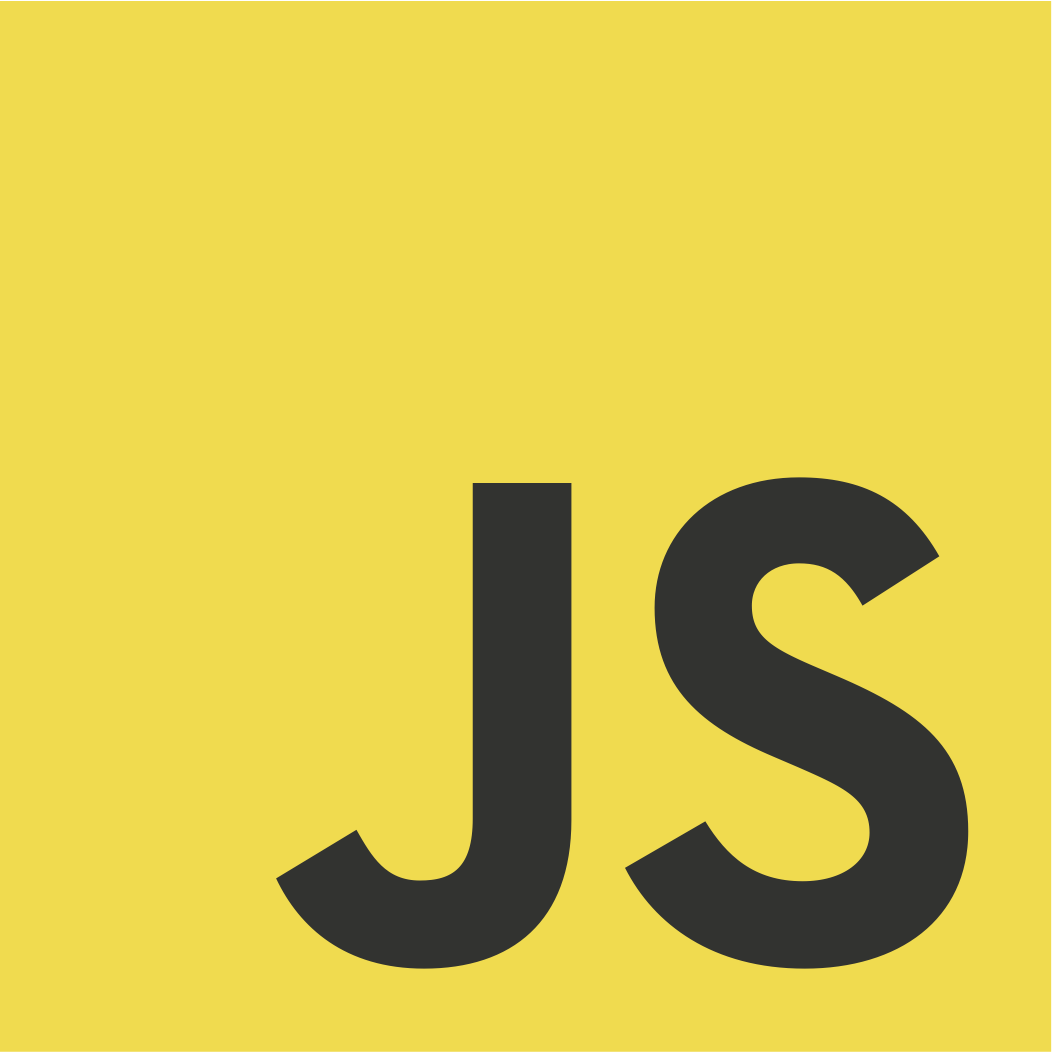
**Wyzwania:**

* Poznasz podstawy programowania w języku JavaScript.
* Nauczysz się debugować kod i rozwiązywać problemy.
* Poznasz narzędzia ułatwiające pracę.

**8.1. Co to w ogóle jest ten JavaScript i po co mi to?**

**Czym jest JavaScript?**

JavaScript to skryptowy język programowania wykorzystywany najczęściej przy budowaniu stron internetowych, a mówiąc dokładniej, odpowiadający za wykonywanie różnego rodzaju operacji po stronie klienta (użytkownika odwiedzającego stronę).

[](https://kodilla.com/static/bootcamp/8-1-js-main.png)

Możesz się również spotkać z nazwami ECMAScript, ES5, Native Javascript, VanillaJS. Wbrew niektórym “specjalistom" jQuery to zupełnie co innego. O samej bibliotece jQuery dowiesz się więcej w kolejnym module.

JavaScript to bardzo przyjemny i elastyczny język. Na jego korzyść przemawia już sam fakt, że aby zacząć w nim pisać, wystarczy Ci sama przeglądarka internetowa.

Ze swojej strony polecam przeglądarkę Chrome, ale może być to równie dobrze Firefox, Safari - celowo pomijam IE :) Warto, aby twoja przeglądarka była zaktualizowana do najnowszej wersji.

Możesz również skorzystać z narzędzi jak np.:

* Twoje Projekty - edytor Kodilli
* <http://codepen.io/>
* <https://jsfiddle.net/>
* <https://plnkr.co/>
* <https://thimble.mozilla.org/>

Wszystkie te narzędzia służą do tego samego - do interpretowania kodu HTML, CSS i JS.

Może się nieco zdziwisz i zapytasz:

"Ale po co mi to?"

No chociażby po to, aby móc pokazać kod swojemu mentorowi, wysyłając mu jedynie link w razie potrzeby konsultacji lub po prostu do przetestowania swoich pomysłów.  
Nieco dalej dowiesz się, jak z niego korzystać oraz jak to robić u siebie na komputerze.

**Do czego służy JavaScript**

Wróćmy na chwilę do tego, co już wiesz i dodajmy do tego JavaScript.

Budowa strony internetowej składa się z trzech głównych etapów:

* stworzenie struktury strony, czyli jej zakodowanie/opis elementów z wykorzystaniem znaczników języka HTML,
* ostylowanie, nadanie unikalnego wyglądu stronie przez opisanie wyglądu jej poszczególnych elementów przy użyciu reguł CSS,
* opracowanie dynamicznych elementów strony, interakcja z użytkownikiem - do tego służą skrypty JavaScript.

Zauważ, że do tej pory tworzyłeś statyczne strony - interakcja była możliwa tylko z użyciem pseudoklas CSS, takich jak :hover czy :focus. Dzięki wykorzystaniu JavaScriptu tych możliwości jest o wiele więcej.

Właśnie tutaj zaczyna się Twoja przygoda z JavaScriptem, jednak jeszcze zanim przejdziesz do pisania kodu, zajrzyj na kilka swoich ulubionych stron i spróbuj zgadnąć, które elementy wykorzystują JS.

Kilka przykładów wykorzystania:

* slider z obrazkami, które przełączają się co jakiś odstęp czasu,
* hamburger menu, wysuwane menu,
* elementy wykorzystujące kliknięcie, niepowodujące przeładowania strony,
* walidacja formularzy,
* dodanie produktów do koszyka,
* polubienie/łapka w górę,
* ukrycie elementu informującego o plikach cookie po kliknięciu przycisku,
* odliczanie czasu, często spotykane w sklepach internetowych,
* modal, czyli wyskakujące na stronie okienka,
* doczytywanie postów (nieskończony scroll),
* gry.

To tylko niepełna lista przykładów zastosowania tego języka.

**Możliwości JS-a. Dlaczego warto się tego nauczyć?**

Opisane do tej pory możliwości dotyczą wyłącznie zastosowania skryptów w przeglądarce. Jednak do tej pory powstało wiele bibliotek i narzędzi opartych o JavaScript, które niekoniecznie wiążą się z wykonywaniem działań tylko na elementach HTML-a.

Oto kilka z nich:

**jQuery**

To bardzo przydatna biblioteka ułatwiająca wykonywanie operacji na elementach strony w przeglądarce klienta. Jest to narzędzie, które w dużym stopniu może zastąpić czysty JavaScript przy wykonywaniu działań na stronie po jej wczytaniu. O tym dowiesz się w kolejnym module.

[](https://kodilla.com/static/bootcamp/8-1-jquery-logo.gif)

**NodeJS**

Jest to framework, który wykonuje kod JavaScript, a wykorzystuje się go po stronie serwera. Mówiąc najprościej, z jego użyciem możemy zastąpić PHP. O ten framework oparte są również różne narzędzia, jednym z nich jest Grunt.

[](https://kodilla.com/static/bootcamp/8-1-node-logo.png)

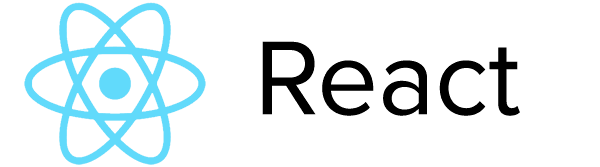
**GruntJS**

Task runner, który pozwala na zautomatyzowanie nużących zadań, takich jak kompilacja Sass, kompresja kodu czy odświeżanie przeglądarki po zmianie plików. Tak jak było przed chwilą wspomniane, swoje działanie zawdzięcza on NodeJS.

[](https://kodilla.com/static/bootcamp/8-1-grunt-logo.png)

**Frameworki SPA**

Dziś ogromną popularnością cieszą się frameworki jak: Angular, Knockout, Ember, ReactJS, Angular2. Służą one do tworzenia internetowych aplikacji, tzw. SPA (Single Page Application). Cechują się one dużą szybkością działania w porównaniu do tradycyjnych stron internetowych. Są napisane w JavaScripcie i jego podstawowa znajomość jest niezbędna, aby rozpocząć z nimi pracę.

[](https://kodilla.com/static/bootcamp/8-1-react-logo.png)

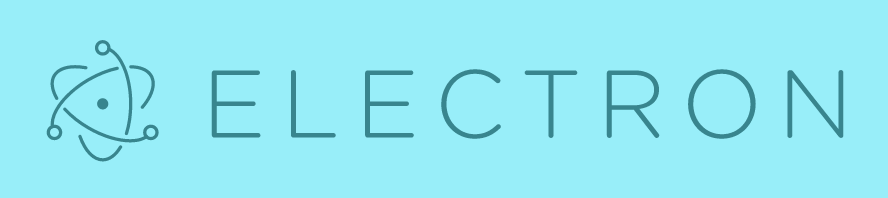
**Aplikacje mobilne**

Dzięki wykorzystaniu takich frameworków jak: Cordova, Phonegap, Ionic, React Native lub Native Script można tworzyć aplikacje mobilne takie, jak strony internetowe. Swego czasu jeden z dużych serwisów społecznościowych, a dokładniej LinkedIn, przez długi czas posiadał swoją aplikację wykonaną właśnie w ten sposób. Plusem tego rozwiązania jest wieloplatformowość. Jedna aplikacja, na wiele systemów.

[](https://kodilla.com/static/bootcamp/8-1-ionic-logo.jpg)

**Aplikacje desktopowe**

Prym wiedzie tutaj narzędzie Electron, a aplikacje dzięki niemu stworzone to choćby komunikator Slack lub edytor kodu Atom. Tutaj również pojawia się wieloplatformowość.

[](https://kodilla.com/static/bootcamp/8-1-electron-logo.png)

Sporo tych możliwości prawda?

**Podsumowanie rozdziału**

Po pierwsze, nie przerażaj się - nie musisz znać wszystkiego! Chcę Ci jedynie pokazać, jak duży potencjał ma w sobie ten język programowania. Dziś na rynku pracy poszukiwani są specjaliści dla każdej z tych dziedzin z osobna.

Wspólnym, kluczowym elementem jest tutaj znajomość JavaScriptu, a więc warto go znać.  
No dobra, dosyć tej teorii.

Jeśli masz wątpliwości do powyższego materiału, to - zanim zatwierdzisz - zapytaj na czacie :)

Zapoznałe(a)m się!

**8.2. Pierwsze kroki, jak zacząć**

Stwórz nowy projekt.

Zacznij od stworzenia struktury projektu. Będą potrzebne podstawowe pliki i odpowiednie foldery.

Po pierwsze, stwórz sobie osobny folder na pliki JavaScript. Na razie będzie w nim tylko jeden plik, ale w przyszłości zapewne pojawi się ich więcej. Osobny folder na style i obrazki pewnie też się przyda, dlatego odtwórz na dysku poniższą strukturę dla swojego nowego projektu.

/workspace

/js

scripts.js

/css

style.css

/images

index.html

W ten sposób utrzymasz pliki w porządku i będziesz mógł je odnaleźć bez wysiłku.

**Plik index.html**

Edytuj swój plik index.html i wklej w nim poniższy kod.

<!-- index.html -->

<html>

<head>

<meta charset="utf-8">

<title>Page title</title>

<link rel="stylesheet" href="css/style.css">

</head>

<body>

<h1>Welcome to my page</h1>

<script type="text/javascript" src="js/scripts.js"></script>

</body>

</html>

Za pomocą tagu <script src="js/scripts.js"></script> ładujemy skrypt z zewnętrznego pliku. Jest to zalecane podejście, chociaż nie jedyna możliwość.

Zwróć uwagę na to, że plik JavaScript został dodany tuż przed tagiem zamykającym </body>. To nie jest przypadek, to dobra praktyka.

**Wyjaśnienie:**

Kiedy nasza strona jest ładowana, jest czytana od góry do dołu, a wszystkie polecenia, takie jak załadowanie pliku CSS czy JS, to konieczność pobrania go i wczytania.

W tym miejscu (tuż przed zamknięciem body) można stwierdzić, że cała struktura strony została już wczytana - bo jest na dole :) ) - a plik JavaScript będzie mógł wykonywać operacje na elementach strony bez wpływania na czas jej załadowania.

Gdybyśmy wczytali plik na samym początku (np. w sekcji head) pliku HTML, mogłoby się okazać, że nasz skrypt został wczytany i nastąpi próba jego wykonania, lecz przeglądarka nie zdążyła wczytać wszystkich elementów strony, a co za tym idzie może brakować np. buttona, który będzie wymagany do działania skryptu. Dodatkowo wczytywanie skryptów zablokuje pojawianie się elementów strony, co sprawi, że w pełnej okazałości pojawi się ona użytkownikowi później, niż gdyby skrypty były na dole.

Taki błąd ciężko dostrzec, bo wszystko dzieje się bardzo szybko.

**Zapamiętaj, bo to ważne.**

Po prostu zapamiętaj - skrypt JS-a jest wykonywany natychmiast po jego wczytaniu - i zawsze postępuj tak, jak radzimy. Wstawiaj style w sekcji <head>, pliki JavaScript tuż przed zamknięciem </body>, a ustrzeżesz się przed wieloma niespodziewanymi sytuacjami.

**Plik scripts.js**

Tutaj zawrzemy nasz kod w języku JavaScript.

W tym miejscu musisz pisać zgodnie z zasadami języka. Nie ma tutaj miejsca na żadne pomyłki.

Jeśli coś Ci nie działa, to znaczy, że gdzieś jest błąd, który należy znaleźć i naprawić.

Ten plik jest wykonywany przez przeglądarkę od razu po jego wczytaniu. Prawda jest taka, że może on zostać wykonany nawet wtedy, gdy strona nie zdąży się w pełni wczytać. Dlatego jest ważne, aby był on dołączany na końcu HTML-a.

Spróbujmy coś zadziałać.

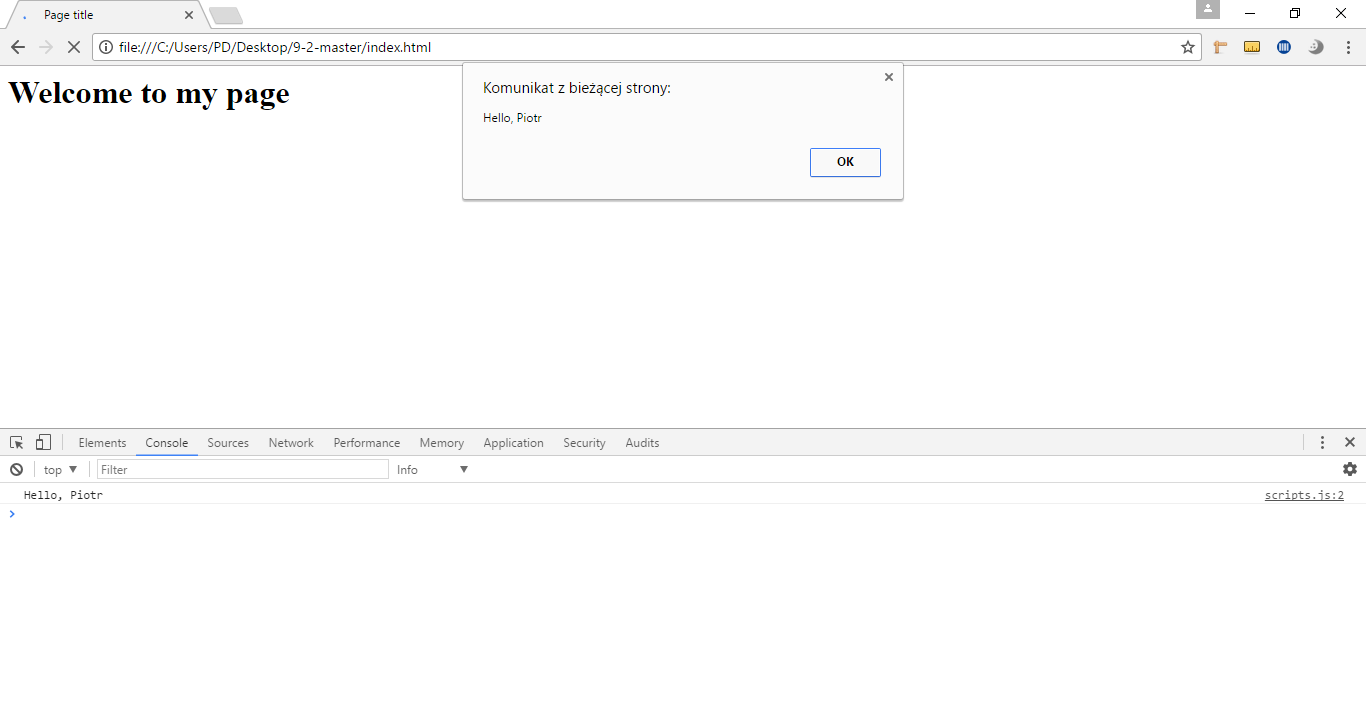
// scripts.js

var name = prompt('Enter your name');

alert('Hello, ' + name);

console.log('Hello, ' + name);

**Spodziewany wynik**

[](https://kodilla.com/static/bootcamp/8-2-result.png)

Teraz odśwież stronę i zobacz, co się stanie.

Spodziewany rezultat to wyskakujące okienko z polem tekstowym. Po wpisaniu imienia powinno się pojawić kolejne okienko z wyświetlonym tekstem Hello [wpisane imię];

**Co się tutaj dzieje?**

W poniższych krokach jest opisane, co się dzieje, kiedy wczytujesz stronę w przeglądarce.

Po kolei:

* Otwierasz plik index.html w przeglądarce.
* Przeglądarka wczytuje ten plik linia po linii.
* Przeglądarka najpierw rozpoznaje typ pliku i czyta jego zawartość.
* Wczytywana jest sekcja head i podpięty w niej plik CSS (wczytywanie dzieje się w tle).
* Następnie sekcja body. Po kolei interpretowane i wyświetlane są tagi języka HTML. Do tego momentu z pewnością wczytały się style, dlatego od razu nadawany jest im wygląd zgodny z zawartymi tam regułami.
* Pod koniec wczytywany jest plik scripts.js. Interpretacja tagów strony się kończy, bo to już końcówka pliku. Zatem strona już jest i można ją przeglądać.
* Kiedy załadowany zostaje plik scripts.js, następuje jego wykonanie.

**Omówienie kodu skryptu**

Teraz przejdziemy do omówienia fragmentów kodu w powyższym skrypcie. Będą tu umieszczone jego fragmenty.

Najpierw skok na głęboką wodę, ale potem jeszcze zajmiemy się tymi poleceniami. Każde z nich będzie jeszcze opisane w dalszej części kursu.

Wracajmy do zadania. Znasz już działanie skryptu jako całości, a teraz dowiesz się jak on działa krok po kroku.

Komentarz

Najpierw jest komentarz.

// scripts.js file

Jest to komentarz liniowy, wszystko, co następuje po tych ukośnikach -'//'- aż do końca linii, nie będzie interpretowane podczas działania kodu. Zostanie po prostu pominięte.

Jest to przydatny sposób nie tylko na wpisanie jakiegoś komentarza dotyczącego np. funkcji, ale również na “wyłączenie" dowolnego fragmentu kodu w czasie sprawdzania poprawności działania. Za pomocą komentarza można też po prostu wyłączyć niedopracowane funkcje zakłócające działanie kodu lub takie, które zostawiamy sobie “na później".

Przykład użycia komentarza w kodzie:

// There is a linear comment here, we can not use enter,

// unless we do the next one again

// alert('test');

// the function above has been commented and will not be executed

Pobranie wartości od użytkownika

Kolejna linia to instrukcja.

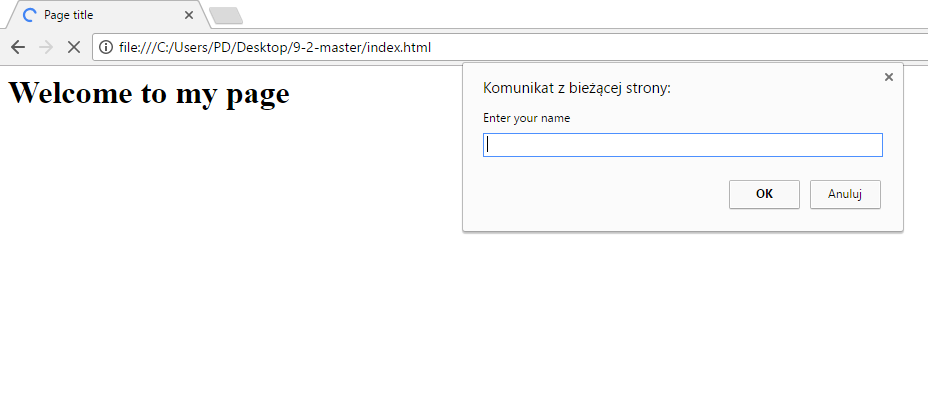
var name = prompt('Enter your name');

Dzieje się tutaj kilka rzeczy.

var name oznacza zadeklarowanie zmiennej o nazwie name.

prompt('Enter your name'); to wywołanie funkcji prompt z przekazaniem parametru. W ten sposób ta funkcja odpowiada za wyświetlenie pierwszego okienka z pytaniem o imię oraz polem do wpisania wartości.

To jest okno z napisem "Komunikat z bieżącej strony":

[](https://kodilla.com/static/bootcamp/8-2-result-name.png)

Funkcje mają to do siebie, że mogą zwracać jakiś wynik. W tym wypadku funkcja promptzwraca wartość wpisaną w okienku.

Ostatnim elementem jest "=", czyli operacja przypisania wartości z prawej strony do zmiennej name. Dzieje się tu tyle, że zmienna otrzymuje wartość podaną w okienku z pytaniem o imię.

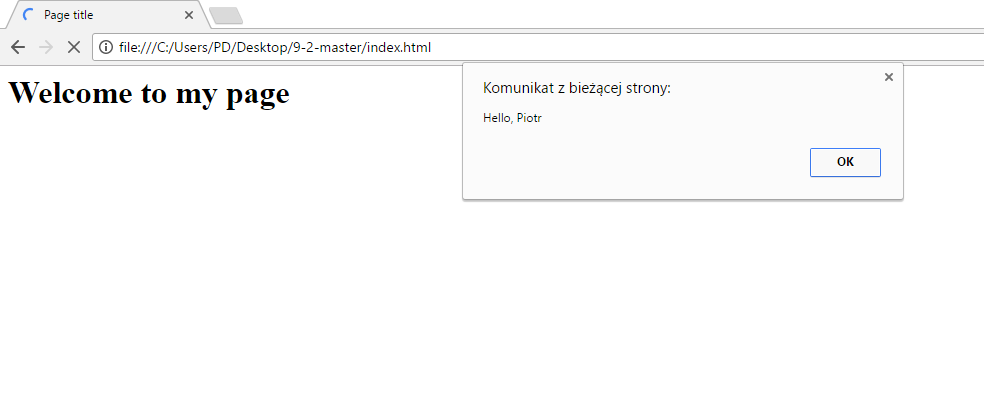
W skrócie, pobraliśmy wartość wpisaną przez użytkownika i zapamiętaliśmy ją w zmiennej.

Wyświetlenie wartości zmiennej

Kolejna linia odpowiada za wyświetlenie wartości w okienku alert.

alert('Hello, ' + name);

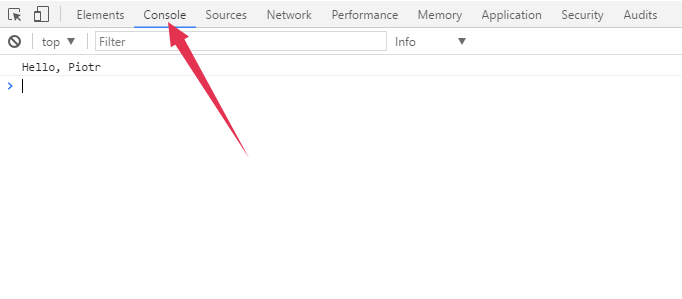
To wywołanie funkcji alert z parametrem. Funkcja alert wyświetla wartość w okienku wyskakującym w przeglądarce. Okienko alerta jest bardzo podobne do prompt, ale z tą różnicą, że nie posiada pola na wpisanie wartości i można je jedynie zatwierdzić przyciskiem OK.

[](https://kodilla.com/static/bootcamp/8-2-final.png)

Kolejny wiersz:

console.log('Hello, ' + name);

Odpowiada dokładnie za to samo, co alert, z tym że wartość zostanie wyświetlona w zakładce Konsola w narzędziach deweloperskich.

[](https://kodilla.com/static/bootcamp/8-2-console-view.png)

**Podsumowanie**

No cóż, pierwsze koty za płoty. Właśnie nauczyliśmy się prawidłowo podpinać skrypt do swojej strony, tworząc przy tym strukturę folderów, z którą się nie zgubisz :)

**Zadanie: Pierwszy skrypt**

Teraz już wiesz, jak zadeklarować zmienną i przechowywać dane, omówiliśmy też jeden ze sposobów na pobranie danych od użytkownika strony, a także na wyświetlenie informacji za pomocą alertu oraz w ulubionym miejscu każdego developera, czyli devTools w zakładce konsola.

Podepnij plik JavaScript o dowolnej nazwie do swojej strony. Wyświetl prompt z zapytaniem o imię, a następnie wyświetl je z użyciem funkcji alert. Możesz skorzystać z kodu dostępnego w rozdziale. Dodatkowo wyświetl ten sam tekst w konsoli.

Stwórz przedstawioną strukturę plików. Przyda Ci się ona w kolejnych zadaniach.

1. W pliku index.html umieść kod zawarty w rozdziale.
2. Podobnie powinieneś postąpić z plikiem scripts.js
3. Uruchom stronę w przeglądarce.
4. Zainicjuj repozytorium gita w głównym folderze projektu.
5. Stwórz nowe repozytorium na GitHubie i wyślij tam swój projekt. Pamiętaj o udostępnieniu go mentorowi na GitHubie. Link do projektu wstaw poniżej.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/76/104869)

Początek formularza

 Wyslij link

Dół formularza

**8.3. Środowisko pracy**

**Edytor kodu**

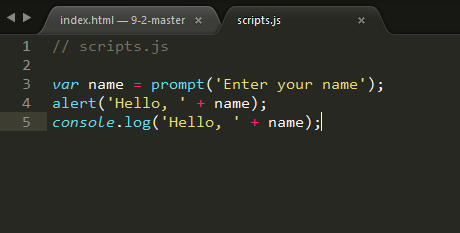
Warto zadbać o to, aby Twój edytor posiadał podświetlanie składni - w ten sposób unikniesz wielu literówek. Język wymaga używania pewnych słów, tzw. 'słów kluczowych', które są stałym elementem kodu.

Zadbaj także o prawidłowe wcięcia (indientacja) w kodzie. Mówiąc prawidłowe, mam na myśli 'zawsze rób tak samo', a unikniesz niedomkniętych bloków kodu (blok to wszystko pomiędzy znakami '{' oraz '}'). Dodatkowo, większość edytorów podświetla otwarcia i domknięcia nawiasów oraz klamer.

Polecenia pisz jedno pod drugim, czasem możesz je grupować oraz oddzielać pustą linią.

Tak jak na obrazku poniżej.

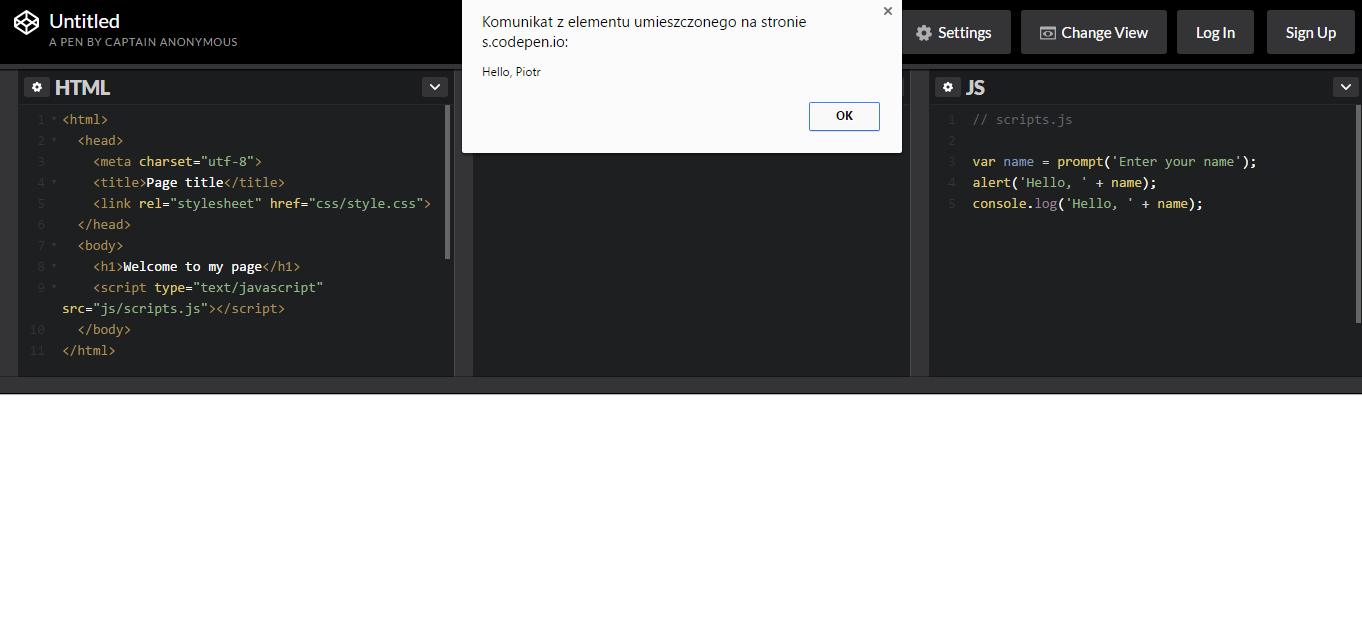
To tyle odnośnie środowiska pracy.

[](https://kodilla.com/static/bootcamp/8-3-editor.png)

Zwracam uwagę na kodowanie znaków oraz wybranie typu pliku, aby prawidłowo podświetlić składnię.

**Edytor online**

Kolejną z możliwości jest edycja kodu online, za pomocą narzędzi takich jak edytor Kodilli lub Codepen. Często udostępniają one dodatkowe możliwości takie jak możliwość szybkiego dołączenia bibliotek, kompilacja Sassa, ale także podpowiadanie składni, czy też wskazywanie błędów.

[](https://kodilla.com/static/bootcamp/8-3-codepen-editor.png)

Tutaj zwykle podświetlanie składni jest standardem. Ogromnym plusem edytorów online jest brak konieczności instalacji dodatkowych programów na komputerze, dostępność online oraz możliwość udostępnienia kodu komukolwiek poprzez skopiowanie linku.

Osobiście często korzystam z tego typu rozwiązań, choćby po to, aby sprawdzić jakiś fragment kodu bez konieczności tworzenia nowego projektu.

**Dodatkowe narzędzia**

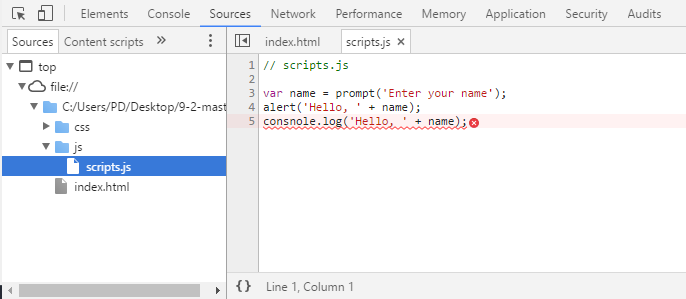
Poza edytorem istnieją dodatkowe narzędzia do sprawdzania poprawności kodu.

Jedne z nich to:

* <http://jshint.com/>
* <http://www.jslint.com/>

Pomogą Ci one wyjść z niejednej opresji.

Żartobliwie mówiąc, zanim zaczniesz pytać wszystkich dookoła 'dlaczego mi nie działa', możesz skorzystać z jednego z nich oraz sprawdzić 'co mówi konsola' ;)

[](https://kodilla.com/static/bootcamp/8-3-console-view.png)

Pamiętaj, że wystąpienie błędu zazwyczaj przerywa wykonywanie skryptu - to znaczy, że kod wykona się albo do końca albo do momentu wykrycia błędu.

Aby przetestować działanie kodu lub jego poprawność, możesz go także po prostu wpisać w konsoli. Na przykład dowiesz się jaki jest wynik dzielenia przez zero według Twojej przeglądarki :)

**Podsumowanie rozdziału**

Dowiedzieliśmy się w czym możesz tworzyć kod, poznaliśmy także podstawowy sposób na uniknięcie błędów przy tworzeniu własnych skryptów.

Omówiliśmy też narzędzia do sprawdzenia poprawności kodu pod względem składni, a także zobaczyliśmy jak wyglądają błędy w konsoli. Wybór należy do Ciebie.

**Zadanie: Instalacja JS-hint**

Stwórz nowe repozytorium na GitHubie, nazwij je np. "zadanie\_8\_3", i sklonuj je do siebie na dysk. Skopiuj do niego pliki *package.json* i *.gitignore* z jednego z wcześniejszych projektów. Następnie zainstaluj w projekcie wszystkie pakiety zdefiniowane w *package.json* za pomocą komendy npm install.

Wzorując się na poprzednim module, dodaj pakiet [JSHint](https://www.npmjs.com/package/jshint) do taska test w skryptach NPMa.

W tym celu:

1. zainstaluj pakiet jshint, pamiętając o fladze --save,
2. sprawdź w jaki sposób wywoływać jshint - użyj do tego flagi --help lub [dokumentacji](http://jshint.com/docs/cli/),
3. w *package.json:*
   1. dodaj task test:js, który będzie uruchamiał jshint dla wszystkich plików js/\*.js,
   2. w tasku test dodaj wywołanie taska test:js.
4. przetestuj swoją konfigurację.

Jeśli wszystko działa poprawnie, a konfiguracja naszego task runnera jest zrobiona wedle poprzeniego modułu, w terminalu wyświeli się wynik sprawdzania plików js/\*.js po wywołaniu któregkolwiek z tasków: test:js, test oraz build.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/76/104870)

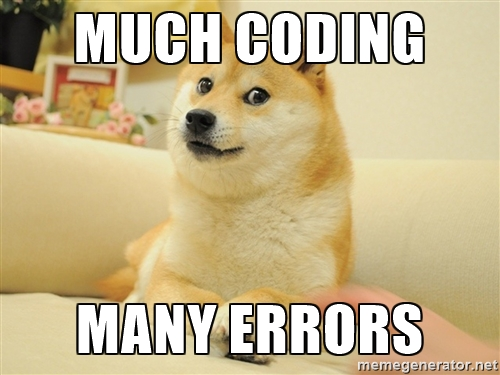
Początek formularza

 Wyslij link

Dół formularza

**8.4. Początki zawsze są trudne**

**Jest tak, jak mówi pieseł.**

[](https://kodilla.com/static/bootcamp/8-4-dog.png)

Każdy z nas popełnia błędy, nieomylnych deweloperów po prostu nie ma. Na szczęście wraz z rosnącym doświadczeniem popełnia się coraz mniej błędów i coraz łatwiej je wyłapać jeszcze przed uruchomieniem skryptu - na to potrzeba jednak czasu i wielu napisanych linii kodu.

**Podłączanie skryptu JavaScript do strony**

Żeby użyć języka JavaScript na naszej stronie, musimy dodać do niej jego skrypty. W pierwszych krokach już to robiliśmy, tutaj poznasz dodatkowe możliwości - a nuż Ci się przyda lub ktoś Cię o to zapyta, kiedy pójdziesz na rozmowę kwalifikacyjną.

Metody są bardzo podobne do tych, które są stosowane jak w przypadku stylów CSS:

* przypisywanie zdarzeń bezpośrednio do elementu w HTML; jednak w dzisiejszych czasach unikamy tej metody. <button onclick="alert('test')">Click me!</button>
* dołączanie skryptu w HTML-u. Wpisując kod pomiędzy tagami <script></script>, np. <script type="text/javascript"> alert('JavaScript is here!'); </script>
* ładowanie skryptu z zewnętrznego pliku (zalecane) <script src="scripts.js"></script>. W ten sposób dołączamy nasz skrypt do strony. Wewnątrz tego tagu nic nie piszemy, ale musimy zadbać o to, aby atrybut src wskazywał właściwy plik. W tym przypadku jest to plik o nazwie scripts.js znajdujący się w tym samym folderze, co plik HTML

Ostatnie dwie metody możemy teoretycznie wykorzystać w dowolnym miejscu na stronie.

Wklej te elementy w swój kod HTML lub w edytorze online, aby sprawdzić jak to działa. Zawartość pliku scripts.js zdążyliśmy już poznać - znajdziesz ją w początkowym rozdziale.

Plik lokalny to nie jedyna możliwość, skrypt możesz doczytać z dowolnego źródła, na przykład z jakiegoś serwera (najczęściej z tzw. CDN-ów).

**Debugowanie**

Jeszcze zanim zacznie się pisać kod, warto nauczyć się sprawdzania, czemu nie działa (jeśli tak się zdarzy, będziesz już wiedział, co robić).

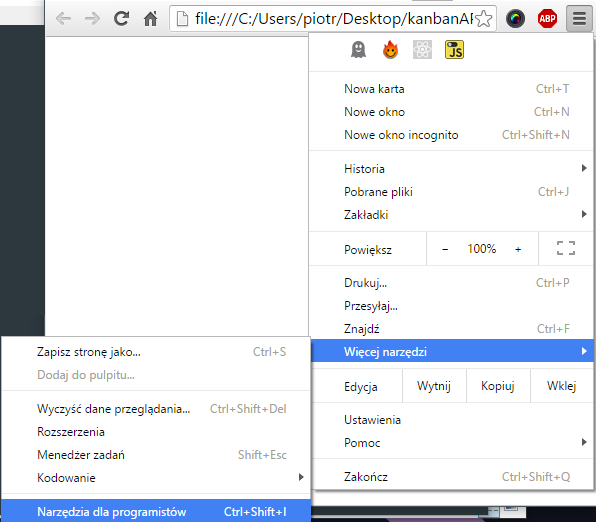
Debugowanie to proces odnajdowania błędów w kodzie i ich naprawiania. Przyda Ci się do tego konsola, narzędzia deweloperskie oraz wspomniane wcześniej JSLint/JSHint.

Poniżej możesz zobaczyć przykładowy błąd, który został wywołany celowo, aby pokazać Ci jak to wygląda.

Najpierw włącz przeglądarkę i wejdź na obojętnie jaką stronę, a następnie włącz narzędzia deweloperskie. Możesz to zrobić z menu lub skrótem klawiszowym.

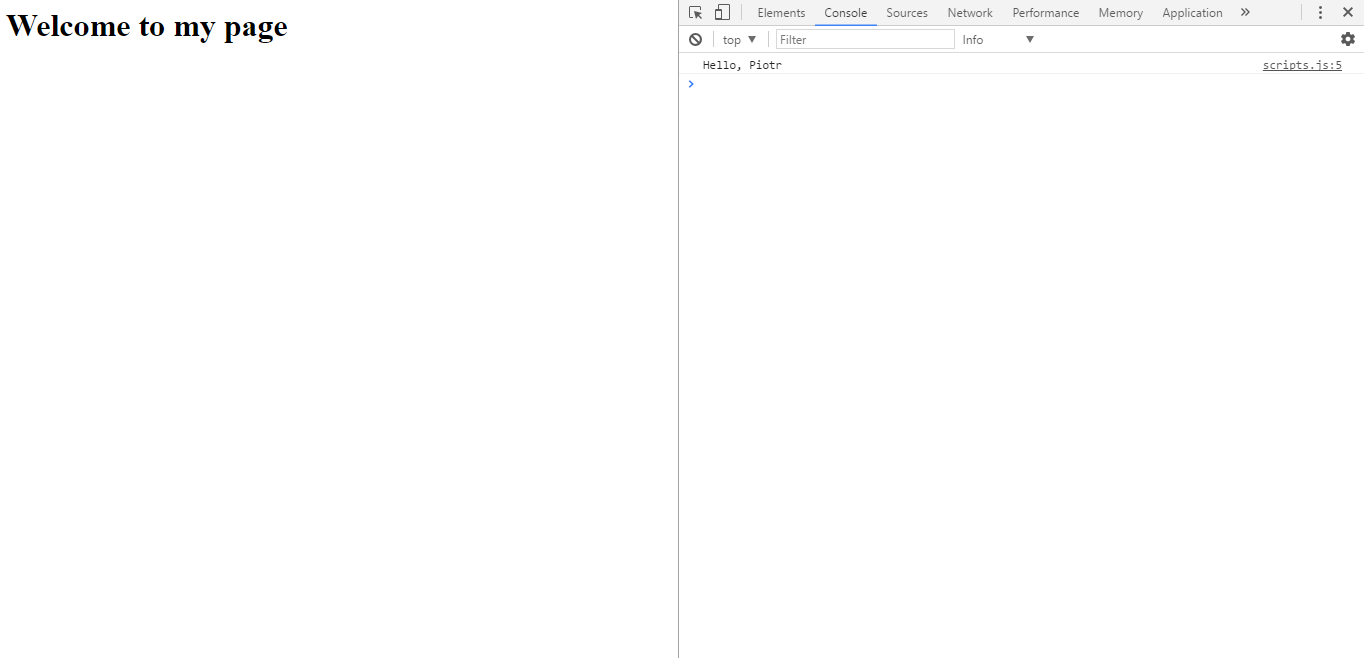
W przeglądarce Chrome znajdź w menu Narzędzia dla programistów lub użyj skrótów:

* Windows: ctrl + shift + i
* Mac: alt + cmd + i
* lub znajdź w menu Narzędzia dla programistów

[](https://kodilla.com/static/bootcamp/8-4-more-tools.png)

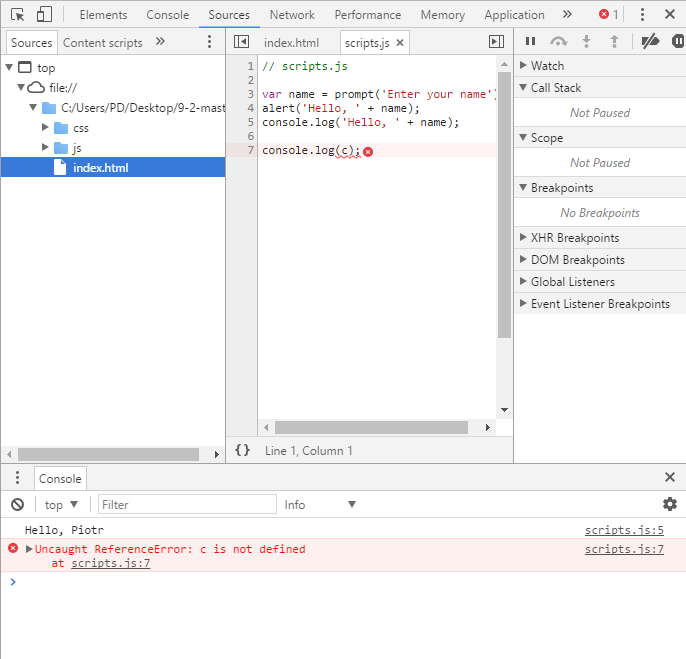
Przejdź w zakładkę konsola.

Powinien Ci się ukazać widok podobny do poniższego:

[](https://kodilla.com/static/bootcamp/8-4-hello.png)

Jeśli przydarzy Ci się jakaś literówka, źle zadeklarowana zmienna lub cokolwiek, co nie pozwoli na prawidłowe działanie skryptu, w konsoli zostanie wyświetlony błąd. Narzędzia deweloperskie to nasz przyjaciel i będziemy z niego bardzo często korzystać.

Zobacz przykładowy błąd na obrazku poniżej:

[](https://kodilla.com/static/bootcamp/8-4-error-console.png)

Błąd składa się z komunikatu, w którym jest on opisany (co właściwie się stało, że nie działa); kliknięcie na nazwę pliku/miejsca, gdzie błąd powstał (nazwa z prawej strony wiersza), umożliwia podejrzenie fragmentu kodu, w którym jest błąd.

Odnośnie przedstawionego błędu to mówi on, że zmienna c wcale nie istnieje i rzeczywiście tak jest, bo nie została przeze mnie zadeklarowana w żadnym miejscu.

**Typowe problemy**

Na początku typowe problemy to przede wszystkim literówki w nazwach zmiennych, funkcji lub słów kluczowych (nazwy wykorzystywane przez JavaScript, tj. function, var, if, else, switch).

Nieprawidłowa ścieżka do pliku skryptu.

Aby sprawdzić, czy plik rzeczywiście zostaje wczytany do strony, można w nim wpisać metodę 'alert('test');' na samym jego początku. Jeśli plik jest dobrze dołączony, z pewnością zobaczysz stosowny komunikat.

Kolejnym najczęstszym problemem jest niedomykanie klamer/bloków.

Blokiem określamy wszystko, co znajduje się wewnątrz nawiasów klamrowych, łącznie z nimi.

{

// anything inside the block

}

Problemów z domykaniem bloków można łatwo uniknąć. Po prostu dbaj o wcięcia (indentacja). Instrukcja i klamra otwierająca oraz zamykająca zawsze powinny posiadać tę samą wielkość wcięcia, a kod wewnątrz bloku powinien być dodatkowo wcięty.

W poniższym linku są opisane tzw. best practices, czyli dobre praktyki w zakresie wyglądu i czytelności kodu.

https://github.com/airbnb/javascript

Kolejnym problemem jest również niespodziewane działanie kodu przez to, że zmienna ma inną wartość niż się spodziewamy.

Aby sprawdzić, jaką wartość ma zmienna w danym momencie, polecam posłużyć się wbudowaną funkcją console.log(variable name);. Wyświetli ona aktualną wartość zmiennej w konsoli przeglądarki.

Przykład

var variable = 'test';

console.log(variable);

**Komentarze**

Możesz stosować dwa rodzaje komentarzy. Przydają się one nie tylko do tego, aby coś w nich napisać, ale także do wyłączania fragmentów kodu.

W końcu czasem nie chcemy czegoś kasować, a jedynie odłożyć na bok. Komentarz sprawia, że kod zostaje pominięty.

**Komentarz liniowy**

Można w nim zawrzeć tylko jedną linię. Jest aktywny od miejsca, w którym wstawi się dwa ukośniki //, aż do końca wiersza.

// linear comment

// Below I have commented on a function call to disable it

// console.log('The commented will not be called');

**Komentarz blokowy**

Można zawrzeć w nim wiele linii. Rozpoczyna się znakami /\*, a kończy \*/. Wszystko, co znajduje się pomiędzy tymi znakami, jest zakomentowane.

/\*

block comment

console.log('The commented will not be called');

\*/

**Podsumowanie rozdziału**

Wiesz już co zrobić, jeśli coś pójdzie nie tak. Na co zwrócić uwagę, kodując oraz jak pozbyć się fragmentów kodu bez konieczności ich usuwania.

Mam także nadzieję, że zapoznałeś się również z dobrymi praktykami pisania kodu i używasz tylko spacji lub tylko tabów do wcinania kodu.

[](https://kodilla.com/static/bootcamp/8-4-comment.png)

Chwila przerwy.

* <http://dogescript.science/>
* <https://chrome.google.com/webstore/detail/doge-my-site/ghefodhmgeofnfpbmjlkglokpgomcodc>

**Zadanie: Dodajemy komentarz do skryptu**

1. W pliku scripts.js zakomentuj linię z poleceniem alert dowolnym typem komentarza.
2. Dodaj jeszcze dwa inne komentarze.
3. Stwórz nowe repozytorium i wyślij tam swój projekt. Pamiętaj o udostępnieniu go mentorowi na Githubie. Link do projektu wstaw poniżej.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/76/104871)

Początek formularza

 Wyslij link

Dół formularza

**8.5. Typy danych, po co Ci zmienne**

**Jak zadeklarować zmienne**

Zmienne definiujemy słowem kluczowym var (ang. *variable*, czyli zmienna). Służą one do przechowywania danych.

Podczas działania skryptu z pewnością wykorzystasz je niejednokrotnie. Do zmiennych możesz przypisać wartości różnego typu, a operacja tworzenia zmiennej nazywa się deklaracją.

**Instrukcja przypisania**

Instrukcja przypisania służy do nadania początkowej wartości zmiennej lub zmiany jej wartości.

Robi się to poprzez wykorzystanie znaku równości (=) pomiędzy nazwą zmiennej a jej wartością.

Po lewej stronie zapisujemy zmienną, a po prawej wartość, jaka ma jej zostać nadana.

**Deklaracja zmiennej, bez przypisania wartości:**

var variable;

Deklaracja zmiennej, z nadaniem jej wartości początkowej:

var variable = "I'm a variable with string value";

Deklaracja wielu zmiennych na raz:

var variable,

variable2 = 19,

variable3;

Poniższy przykład ilustruje przypisanie wartości typu integer, string oraz boolean.

var foo = 12;

// foo is equal to: 12;

var string = 'test';

// string is equal to: 'test';

var isSomething = true;

// isSomething is equal to: true

W dalszej części (Typy danych) poznasz dodatkowe właściwości tej instrukcji.

**Gdzie umieszczać zmienne**

Zmienne staraj się umieszczać na początku skryptu lub funkcji.

**Wartość domyślna**

Jeśli to tylko możliwe, nadawaj zmiennym wartość początkową. Ta wartość może potem posłużyć jako wartość domyślna. Unikniesz w ten sposób potencjalnych błędów z wyświetlaniem wartości undefined.

**Oczywiście rób to z głową - nic na siłę.**

Deklaruj kilka zmiennych naraz - nie ma sensu się rozdrabniać - możemy napisać słowo kluczowe var tylko raz, a następnie umieścić listę zmiennych jedna pod drugą, oddzielonych przecinkami - to znacznie zwiększy czytelność kodu.

**Nazewnictwo zmiennych**

Nazywając zmienne, powinniśmy używać tzw. zapisu camelCase. Pierwsza litera wyrazu pisana jest małą literą, a każdy kolejny wyraz powinien być pisany wielką literą bez odstępu

Przykład:

// scripts.js

var areaOfTriangle = 0,

userName = 'Piotr';

Nazywaj zmienne po imieniu, na przykład zmienną, która będzie przechowywać wartość pola trójkąta, nazwij triangleArea, a tę przechowującą obiekt z danymi użytkownika - userData. W ten sposób łatwo odnajdziesz szukane wartości.

Staraj się nie używać zmiennych o nazwach x, y, z czy też x132j - ich funkcja może być dla Ciebie oczywista w momencie pisania kodu, ale kiedy wrócisz do swojego skryptu po kilku dniach, możesz mieć problem ze zrozumieniem jak on działa - tak samo mogą mieć inni programiści, z którymi będziesz współpracować.

Zmienna może posiadać jeden z typów danych przedstawionych poniżej:

**Typy danych**

W języku JavaScript do zmiennej możemy przypisać kilka rodzajów danych.

**Boolean**

Jest to typ logiczny, który posiada dokładnie dwie wartości - true i false, czyli prawda i fałsz. Pomaga przy określaniu prostych właściwości, np.

var isActive = true;

Warto nazywać zmienne tego typu z przedrostkiem is - wtedy od razu wiadomo, z jakim typem danych mamy do czynienia.

**String (napis)**

String jest ciągiem pojedynczych znaków (liter, cyfr, znaków interpunkcyjnych i specjalnych), np. 'Jan Kowalski'.

Do określania tego typu możemy użyć następującej składni:

var userName = "Jan Kowalski";

lub

var userName = 'Jan Kowalski';

Tutaj nie ma znaczenia, czy używasz zapisu apostrofów ' (singlequote) czy cudzysłowów "(doublequote) - Należy pamiętać, aby konsekwentnie rozpoczynać i kończyć deklaracje napisu tym samym znakiem.

Każdy string rozpoczęty znakiem apostrofu zakończy się znakiem apostrofu. Z cudzysłowem jest tak samo. Wszystko, co znajduje się pomiędzy tymi znakami, będzie traktowane jako napis.

Ale co się stanie, kiedy w naszym tekście zechcemy użyć cudzysłowu lub apostrofu? Przecież te znaki oznaczają początek lub koniec napisu.

Zatem kilka przykładów.

Jeśli chcesz zadeklarować napis, możesz to zrobić za pomocą apostrofu lub cudzysłowu.

var firstSentence = 'I like a song that is called 'Only you.';

var secondSentence = "I love a song that is called "Only you.";

Powyższy przykład zawiera w sobie błędy. JavaSript uzna, że 'Only you' to zmienna, ponieważ tuż przed początkiem zdania natrafi na apostrof.

Ponieważ napis zaczął się apostrofem, kolejny napotkany apostrof będzie uznany za kończący deklarację napisu.

Nie taki mieliśmy zamiar. Na szczęście są na to dwa sprawdzone sposoby.

Pierwszy to 'eskejpowanie' (od ang. 'uciekać, unikać') znaków specjalnych. Służy do tego backslash, czyli po prostu znak '\'. Jeśli chcemy, aby nasz znak specjalny (kończący napis) był mimo wszystko uznany za zwykły znak, musimy go poprzedzić backslashem.

var firstSentence = 'I like a song that is called \'Only you\'.';

var secondSentence = "I love a song that is called \'Only you\'.";

W ten sposób "mówimy" Javascriptowi, aby ten znak po prostu wypisał.

Drugim sposobem jest użycie apostrofu i cudzysłowu naprzemiennie. Jeśli rozpoczniemy napis cudzysłowem, to dopiero kolejny cudzysłów zakończy naszą deklarację napisu, a wewnątrz bez dodatkowych konsekwencji możemy używać apostrofów. Taką samą zależność możemy wykorzystać, kiedy ciąg znaków rozpoczynamy apostrofem.

Ilustruje to poniższy przykład:

var firstSentence = 'I like a song that is called "Only you".';

var secondSentence = "I love a song that is called 'Only you'.";

Zachęcam Cię do sprawdzenia tego w konsoli.

**Liczby**

W języku JavaScript mamy tylko 1 typ liczb, które możemy zapisać jako stałe (int) lub zmiennoprzecinkowe (float).

var age = 23;

var money = 25.90;

**Tablice (ang. *arrays*)**

Są zazwyczaj używane do określania zbiorów elementów.

var hobby = ['HTML', 'CSS', 'JavaScript'];

Poszczególne elementy oddzielane są przecinkiem ( , ). Tablice mogą zawierać w sobie różne typy danych jednocześnie.

Poniżej widzisz tablicę zawierającą dane różnego rodzaju, znajdziesz w niej nawet zagnieżdżoną tablicę.

var test = ['HTML', 5, 'World', 9.99, ['Lorem', 'Ipsum', 2]];

Numeracja elementów w tablicy zaczyna się od 0.

Jeżeli chcemy odczytać tekst z pozycji 3 (z powyższej tablicy), posłużymy się takim kodem:

var foo = test[2]; // World

Wartość zmiennej foo równa będzie World

Liczbę w nawiasie kwadratowym nazywamy indeksem tablicy.

Zapis:

var foo = test[4];

var bar = foo[1];

**Obiekty**

To struktura danych, która może zawierać w sobie pozostałe typy.

Cechuje się tym, że opisuje się ją za pomocą atrybutów oraz wartości tych atrybutów. W przypadku nazw atrybutów obowiązuje takie samo nazewnictwo, jak przy zmiennych.

Obiekt definiujemy za pomocą klamer { oraz }. Jest to tak zwany literał obiektowy będący zalecanym sposobem tworzenia obiektów.

Obiekt może być pusty:

var object = {};.

W momencie deklaracji może też zawierać jakieś właściwości (para atrybut, wartość):

var variable = 'variable value';

var object = {

attribute: 'attribute value',

otherAttribute: 123 + 456,

attributeWithVariableValue: variable

};

Atrybuty wewnątrz obiektu oddzielamy przecinkiem.

Obiekty świetnie służą do tego, aby trzymać w nich struktury, grupować właściwości w jednej zmiennej.

Przykładem może być:

var person = {

name: 'Jon',

surname: 'Snow',

siblings: 5,

knownThings: ['Nothing','Night Watch']

};

Są dwie możliwości odczytania wartości któregoś z atrybutów obiektu: wpisanie jego nazwy po kropce lub podanie jego nazwy w nawiasach kwadratowych.

var userName = person.name;

var userSurname = person['surname'];

Istotne jest, aby przy używaniu nawiasów kwadratowych nazwa atrybutu była wpisana w cudzysłów. Inaczej będzie ona traktowana jak kolejna zmienna.

W nawiasy kwadratowe może zostać również wstawiona zmienna. Jednak zasada działania pozostaje taka sama.

Ciekawą własnością obiektów jest to, że atrybuty można przypisywać do nich dynamicznie, tzn. nadać wartość jakiemuś atrybutowi nawet jeśli do tej pory obiekt go nie zawierał.

var person = {

name: 'Piotr'

};

person.age = 26;

Mimo tego, że na początku obiekt 'person' nie posiada atrybutu 'wiek', to możliwe jest jego dodanie bez konieczności wcześniejszej deklaracji.

Atrybut wiek zostaje po prostu dodany do obiektu 'person'.

**undefined**

W języku JavaScript zmienna, zanim zostanie do niej przypisana wartość, posiada wartość domyślną undefined.

var person;

console.log(person); // undefined

**null**

Null służy do wskazywania, że zmienna nie odnosi się do prawidłowych danych. Null jest specjalną wartością oznaczającą "wartość nieustaloną". Natomiast undefined oznacza, że dany obiekt nie istnieje, nie został nigdzie zadeklarowany lub nie została mu nadana wartość.

**NaN**

Not a Number, czyli 'to nie jest liczba'. Występuje np. przy mnożeniu liczba razy string.

console.log(10 \* 'napis'). // NaN

**Infinity**

Nieskończoność. W obliczeniach z pewnością wystąpił błąd - dzieje się tak na przykład wtedy, gdy spróbujesz dzielić przez zero.

**Operacje na zmiennych**

W języku JavaScript, jak i w każdym innym języku programowania, mamy możliwość wykonywania operacji na danych. Możemy dodawać, odejmować, dzielić i nie tylko.

Poniższa tabela przedstawia dostępne operatory.

**Operatory arytmetyczne**

| **Nazwa** | **Operator** |
| --- | --- |
| Dodawanie | + |
| Odejmowanie | - |
| Mnożenie | \* |
| Dzielenie | / |
| Dzielenie modulo | % |
| Inkrementacja, zwiększenie wartości o 1 | ++ |
| Dekrementacja, zmniejszenie wartości o 1 | -- |

**Dodawanie**

Podstawowa operacja, bez której żaden z komputerów nie mógłby działać. Nie będę wyjaśniał, na czym polega ta operacja z matematycznego punktu widzenia, ponieważ sam dowód zajmuje 162 strony (<https://en.wikipedia.org/wiki/Principia_Mathematica#Quotations>).

Wypiszmy wszystkie dostępne możliwości. W dodawaniu liczb nie ma niespodzianek, ale dodawać do siebie można również ciągi znaków, czyli *stringi*.

var foo = 'lorem' + 'Ipsum';

Wartość foo to teraz loremIpsum.

foo = 'lorem' + 1 + 2;

foo wynosi lorem12 ponieważ typ pierwszej zmiennej to string.

foo = 1+2+'lorem';

foo wynosi 3lorem, ponieważ typ pierwszej zmiennej to number.

Zachęcam Cię do sprawdzenia wyników działań w konsoli. Kilka przykładów znajduje się poniżej:

foo = 'lorem' + ['1', 2, 3.4]; // 'lorem1, 2, 3.4'

foo = ['1', 2, 3.4] + 'lorem'; // '1,2,3.4lorem'

foo = 9 + ['1', 2, 3.4]; // '91,2,3.4'

foo = ['1', 2, 3.4] + 9; // '1,2,3.49'

foo = ['1', 2, 3.4] + [1, '2', 3.4]; // '1,2,3.41,2,3.4'

foo = ['1', 2, 3.4] + {bar: 'lorem'}; // 1, 2, 3.4 [object Object]

foo = {bar: 'lorem'} + [1, '2', 3.4]; // [object Object] 1, 2, 3.4

foo = ['1', 2, 3.4] + undefined; // 1, 2, 3.4 undefined

foo = undefined + [1, '2', 3.4]; // undefined 1, 2, 3.4

Operacje na tablicach i obiektach nie są już takie proste, dlatego do ich wykonywania służą specjalne funkcje.

**Inkrementacja i Dekrementacja**

Każdą zdefiniowaną zmienną typu number możemy zwiększyć lub zmniejszyć o 1 - tego typu operacje są niezwykle przydatne i często wykorzystywane, ponieważ sposób zapisu jest o wiele krótszy.

Inkrementacja oznacza to samo, co dodanie 1 do bieżącej wartości.

var points = 1;

points = points + 1;

lub skrócona wersja

var points = 1;

points += 1;

Możemy to robić w dwojaki sposób.

Wykorzystamy także console.log(). Zachęcam Cię do sprawdzenia poniższych zależności w swoim skrypcie.

var points = 1;

console.log( 'The current value of points is: ', points ); // 1

console.log( 'The current value of points is: ', points++ ); // 1

console.log( 'The current value of points is: ', points ); // 2

Wykonaj powyższy kod i przeanalizuj wartości.

Łatwo się domyślisz, że w przypadku kiedy operator '++' jest z prawej strony, najpierw zostaje wyświetlona (użyta) wartość zmiennej, a dopiero potem jest ona zwiększana o 1.

var points = 1;

console.log( 'The current value of points is: ', points ); // 1

console.log( 'The current value of points is: ', ++points ); // 2

console.log( 'The current value of points is: ', points ); // 2

Kiedy operator '++' znajduje się z lewej strony, kolejność działań jest inna. Najpierw następuje zwiększenie wartości, a dopiero potem jej wyświetlenie (użycie).

Dekrementacja działa analogicznie do inkrementacji z tą różnicą, że wartość 1 jest odejmowana.

points = points - 1;

points -= 1;

**Modulo**

Jest to operacja wyznaczania reszty z dzielenia dwóch liczb.

Używa się jej najczęściej do sprawdzania, czy dany element jest n-tą wartością zbioru lub czy jego wartość jest liczbą parzystą. Operujemy tutaj tylko na wartościach całkowitych (bez części po przecinku).

var x = 15,

foo;

foo = x % 2; // 1 bo 15 / 2 = 7 i 1 reszty

foo = x % 3; // 0 bo 15 / 3 = 5 i 0 reszty

foo = x % 5; // 0 bo 15 / 5 = 3 i 0 reszty

**Podsumowanie rozdziału**

To była solidna dawka wiedzy - teraz już wiesz, czym są zmienne i jak ich używać.

Znasz typy danych i wiesz, jak je możesz wykorzystać.

Potrafisz nie tylko nadawać im wartość, ale także wykonywać na nich działania.

Wszystko czego nauczyliśmy się w tym rozdziale, to Twoje możliwości. Jak z nich skorzystasz, zależy tylko od Ciebie.

**Zadanie: Pole trójkąta**

1. Zadeklaruj zmienne a, h oraz triangleArea. Nadaj im wartość początkową dla a oraz h (jakieś liczby).
2. Przypisz do triangleArea działanie a\*h/2.
3. Na koniec wypisz console.log('Triangle field with base a: ' + a + ' and height h: ' + h + ' is equal to: ' + triangleArea);  
   Do pobrania danych możesz także skorzystać z funkcji prompt.
4. Wyślij zmiany na nowe repozytorium na GitHubie. Pamiętaj o udostępnieniu go mentorowi na GitHubie. Link do projektu wstaw poniżej.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/76/104872)

Początek formularza

 Wyslij link

Dół formularza

**8.6. Instrukcje warunkowe if, switch oraz short if**

Dzięki tym instrukcjom możemy wykonywać kod w zależności od sytuacji. Ich działanie opiera się o sprawdzenie wartości logicznej.

Dlatego najpierw dowiesz się, co to jest ta wartość logiczna oraz jak ją uzyskać.

**Operatory logiczne**

Wynik działań logicznych to true lub false. Używając poniższych operatorów, możemy sprawdzić np. czy wartości są takie same.

Jeżeli warunek operacji logicznej jest spełniony, zwracany jest wynik true, w przeciwnym wypadku operacja zwraca false.

| **Operator** | **Opis** | **Znak** |
| --- | --- | --- |
| Operator równości wartości | Sprawdza, czy wartości po obu stronach są równe lub takie same. | == |
| Operator równości wartości i typu | Sprawdza, czy wartości po obu stronach są równe lub takie same. Oraz czy typ wartości jest ten sam. | === |
| Operator różności | Sprawdza, czy wartości po obu stronach są różne. Odwrotność operatora równości. | != |
| Operator większy niż | Sprawdza, czy wartość po lewej stronie jest większa niż wartość po prawej. | > |
| Operator mniejszy niż | Sprawdza, czy wartość po lewej stronie jest mniejsza niż wartość po prawej. | < |
| Operator większy lub równy | Sprawdza, czy wartość po lewej stronie jest większa od wartości po prawej lub jej równa. | >= |
| Operator mniejszy lub równy | Sprawdza, czy wartość po lewej stronie jest mniejsza od wartości po prawej lub jej równa. | <= |
| Operator negacji. | Odwraca wartość logiczną zmiennej lub wyrażenia. | ! |

Operatory logiczne wykorzystuje się najczęściej w instrukcjach warunkowych, o których więcej napiszemy nieco niżej.

if ( 1 == 1 ) {

// code

}

Opisowo:

Czy wartość 1 jest równa 1? Oczywiście tak, więc wyrażenie zwróci wartość true, a następnie wykona "jakiś kod" :)

if ( !(10 <= 1 )) {

// code

}

Opisowo:

Czy wartość 10 jest mniejsza lub równa równa 1? Oczywiście nie, więc wyrażenie będzie posiadać wartość false. Jednak przed operacją porównania występuje negacja (!). Czyli false zostanie zmieniony na true.

Oznacza to tyle, że całe wyrażenie ma wartość true, czyli prawda.

Z tej okazji zostanie wykonany "jakiś kod" :)

Wyrażenia można także ze sobą łączyć.

**&& - Łącznik "i"**

Pozwala na łączenie dwóch warunków ze sobą.

Aby wynik był prawdą, oba warunki składowe muszą być prawdą.

**|| - Łącznik "lub"**

Pozwala na łączenie dwóch warunków ze sobą.

Aby wynik był prawdą, przynajmniej jeden z warunków składowych musi być prawdą.

**Przykłady**

if ( (1 == 1) && (10 <= 1) ) {

// code

}

Czy wyrażenie pierwsze ma wartość true? Już wiemy, że tak.

Czy drugie wyrażenie ma wartość true? Nie.

Operator && da nam true tylko w przypadku, kiedy obie wartości mają wartość true, więc całe powyższe wyrażenie będzie miało wartość false.

if ( (1 == 1) || (10 <= 1) ) {

// code

}

Tutaj ten sam przykład z taką różnicą, że do łączenia warunków użyliśmy operatora ||.

Operator || da nam true tylko w przypadku, kiedy przynajmniej jedna z wartości będzie miała wartość true. W tym przypadku już pierwsze wrażenie jest prawdziwe, więc mamy pewność, że całe wyrażenie będzie miało wartość true.

Kiedy już pierwsza wartość jest true, drugie wyrażenie nie jest sprawdzane, bo niezależnie od jego wartości całość jest już spełniona.

**Instrukcja warunkowa if**

Instrukcja warunkowa składa się z jednego lub więcej bloków. Każdy z nich (poza else) poprzedzony jest warunkiem. Istotne jest to, że spośród wszystkich bloków zostanie wykonany jeden lub żaden.

Warunek to instrukcja warunkowa, zwraca true lub false.

Blok to operacje zawarte pomiędzy { oraz }.

if (condition1) {

//Code to execute if condition1 is true

} else if (condition2) {

//Code to execute if condition2 is true

} else {

//Code to execute if none of the previous conditions prove to be true

}

Wytłumaczmy sobie po kolei, co dzieje się w powyższym kodzie.

**Sprawdzany jest pierwszy warunek (linia 1).**

Warunek może być prawdziwy lub nie.

Wartość true (jest prawdziwy) - kod zawarty wewnątrz bloku warunku wykona się (linia 2). Dalej będzie wykonywany kod znajdujący się po wszystkich blokach instrukcji if.

Wartość false (nie jest prawdziwy) - przechodzimy do następnego warunku (linia 3).

**Sprawdzany jest drugi warunek (linia 3).**

Sytuacja przypomina tę powyżej - są dwie możliwości. Warunek może być spełniony lub nie.

Wartość true - wykonujemy kod zawierający się w bloku przypisany dla tego warunku (linia 4), a reszta warunków pominięta. Następnie będziemy wykonywać kod znajdujący się po wszystkich blokach instrukcji if.

Wartość false - rozpatrujemy kolejne warunki else if (jeśli istnieją).

Jeśli wszystkie poprzednie warunki nie zostały spełnione, w końcu trafimy na else (linia 5). Jeśli do tego dojdzie, to zostanie wykonany kod znajdujący się w bloku (linia 6), a następnie kod znajdujący się za instrukcją warunkową.

Zanim przejdziemy do zadania, spójrz na poniższy przykład.

Nasza instrukcja warunkowa jest uzależniona od deklaracji w pierwszej linijce kodu, jej wartość może być całkowicie dowolna. Teraz przeanalizujmy nasze warunki:

* Linia 3 - jeśli zmienna userName ma wartość 'Piotr' - wyświetlamy komunikat 'Witaj Piotrze!'. W innym przypadku przechodzimy do następnego warunku.
* Linia 5 - jeśli zmienna userName jest pusta - wyświetlamy komunikat 'Nie wprowadziłeś imienia!'. W innym przypadku przechodzimy do instrukcji else.
* Linia 7 - jeśli powyższe warunki nie są poprawne, to wyświetlamy zawartość else - 'Witaj ', a następnie wartość przypisaną do userName.

Kod wykorzystany w ćwiczeniu powyżej jest interaktywny, możesz dowolnie modyfikować wartość zmiennej userName oraz instrukcje warunkowe.

**Switch**

Jest bardzo podobny w działaniu do instrukcji if. Jednak switch przyjmuje wartość, a następnie porównuje ją z poszczególnymi przypadkami (case). Jeżeli nie uda się dopasować wartości do żadnej opcji, uruchamiane jest domyślne zachowanie default (odpowiednik else).

Załóżmy, że chcemy spytać użytkownika, czy lubi truskawki:

var question = prompt('Do you like strawberries?');

switch (question.toLowerCase()) {

case 'yes':

alert('Great, me too!');

break;

case 'no':

alert('Really?');

break;

default:

alert("You did not give a clear answer, so you're probably still wondering.");

}

Różnica między switch a instrukcją if jest taka, że za każdym razem sprawdzamy wartość przekazaną do instrukcji switch. Tu nie ma możliwości używania operatorów porównania.

switch (value) {

case 'alignedto1':

// code that executes after this case

break;

case 'alignedto2':

// code that executes after this case

break;

default:

// Code that executes after default

}

**Co się tutaj dzieje?**

Przekazujemy zmienną "value" do instrukcji switch.

Sprawdzamy, czy wartość jest taka sama jak w pierwszym case.

Jeśli tak, to wykonujemy kod następujący po tym przypadku, a jeśli nie, to czas na kolejny case.

Jeśli któryś case jest prawdziwy, to zostaje wykonany kod następujący po nim. Instrukcja break; sprawia, że przestajemy sprawdzać kolejne przypadki i opuszczamy cały switch.

Jeśli instrukcja dojdzie do przypadku default, to zostanie on wykonany. Jeśli zapomnimy dać break; po którymś z case'ów, to będą sprawdzane kolejne warunki - jest to i plus i minus takiego rozwiązania.

Użycie switch jest dosyć specyficzne, bo nie ma tutaj możliwości sprawdzenia warunku, np. czy jakaś wartość jest mniejsza od wartości 7, ale za to można wykonać kilka case'ów poprzez pominięcie instrukcji break.

**Przykładowe zadanie**

Powiedzmy, że mamy 3 przyciski na stronie.

Jeśli klikniemy pierwszy, to trzeba zmienić kolor strony, zagrać melodyjkę i wyświetlić wiadomość.

Jeśli drugi - zagrać melodyjkę i wyświetlić wiadomość, a jeśli trzeci - tylko wyświetlić wiadomość.

var buttonNumber = 1;

switch(buttonNumber) {

case 1:

console.log('Change the color of the page.')

case 2:

console.log('Play the tune.')

case 3:

console.log('Display message.')

default:

console.log('This case was made because there is no break')

}

Oczywiście da się to zapisać za pomocą konstrukcji z if i byłoby to równie dobre rozwiązanie, ale nieco dłuższe w zapisie.

**Short if**

Mamy możliwość korzystania ze skróconej wersji instrukcji if/else.

(condition) ? code\_if\_condition\_is\_true : code\_if\_condition\_is\_false

**Jak to działa?**

Po lewej stronie znaku ? zostaje zapisany warunek instrukcji, a po prawej możliwe scenariusze, które są zależne od wartości warunku.

Po lewej stronie znaku : zapisujemy kod, który zostanie wykonany w przypadku spełnienia warunku, a po prawej kod, który będzie wykonany w przeciwnym wypadku. Są tutaj tylko dwie możliwości - true lub false.

Skrócona wersja działa tak samo jak poniższy kod:

if (condition) {

// code\_if\_condition\_is\_true

} else {

// code\_if\_condition\_is\_false

}

**Przykład:**

Aby przypisać wartość zmiennej whatIsBigger, używamy następującej składni:

var whatIsBigger = a > b ? 'Variable a is bigger' : 'Variable b is bigger';

console.log(whatIsBigger);

Zmienna whatIsBigger przyjmie wartość 'Variable a is bigger', jeżeli a > b , a wartość 'Variable b is bigger', kiedy a <= b.

**Podsumowanie rozdziału**

Dowiedzieliśmy się, czym są wartości logiczne oraz operatory logiczne.

Wiesz, jak (wykorzystując instrukcje warunkowe) podzielić kod na fragmenty, aby wykonać tylko jego część.

**Zadanie: Instrukcje warunkowe w praktyce**

1. Stwórz zmienne potrzebne do wykonania działania (a \* a) - (2 \* a \* b) - (b \* b) , a jego wynik przypisz do zmiennej value. Wyświetl wynik w konsoli, a następnie za pomocą instrukcji warunkowej sprawdź, czy wynik jest dodatni czy ujemny, a potem wyświetl informację wynik dodatni lub wynik ujemny w konsoli.
2. Dodaj instrukcję, która sprawdzi, czy wynik jest równy zero i wyświetli tę informację.
3. Stwórz nowe repozytorium i wyślij tam swoje zadanie, a link do niego wklej poniżej. Pamiętaj o udostępnieniu repozytorium swojemu mentorowi.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/76/104873)

Początek formularza

Dół formularza