O'REILLY®

Wydanie II

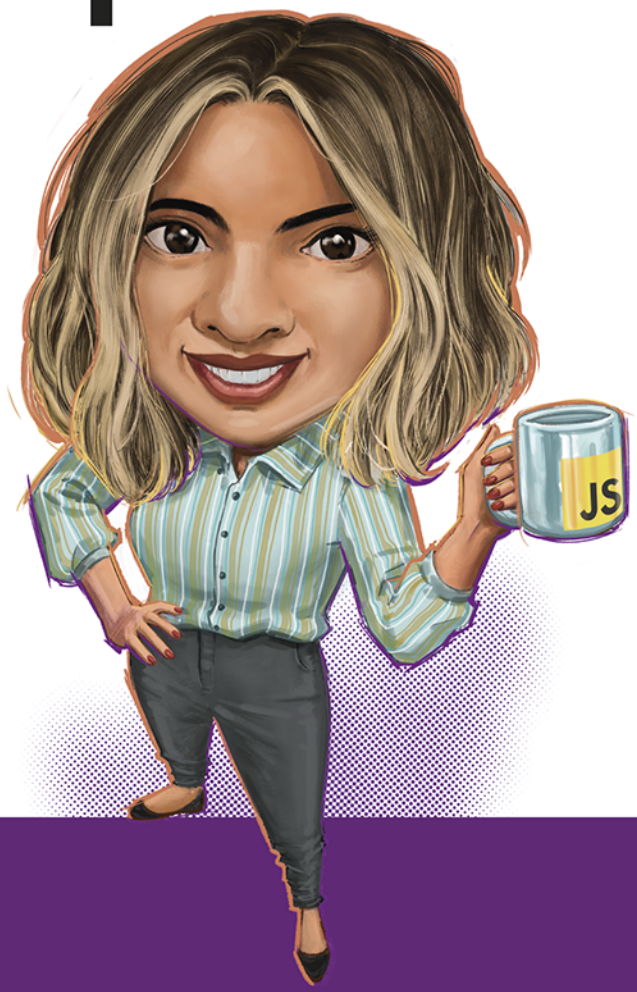
Head First

Programowanie w JavaScript

Rusz głową!

Przewodnik po tworzeniu nowoczesnych aplikacji w JavaScript

Eric Freeman
Elisabeth Robson



Helion

Tytuł oryginału: Head First JavaScript Programming: A Learner's Guide to Modern JavaScript, 2nd Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-289-2265-5

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/prjsr2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści (podsumowanie)

Spis treści (ten właściwy)

Wprowadzenie

Twój mózg myśli o JavaScripcie. Czytając tę książkę, *Ty* starasz się czegoś nauczyć, natomiast Twój *mózg* wyświadcza Ci przysługę, starając się, by cała ta wiedza *nie została zapamiętana*. Twój mózg myśli sobie: „Lepiej zostawić miejsce na istotne rzeczy, takie jak to, których dzikich zwierząt należy unikać oraz czy jeżdżenie na snowboardzie nago jest kiepskim pomysłem, czy nie". Jak więc możesz *przekonać* swój mózg, że Twoje życie zależy od poznania JavaScriptu?

szybki skok na głębokie wody javascriptu

1 Czas się zamoczyć

JavaScript to dane Ci supermoce. To **prawdziwy języka programowania** internetu, który pozwala **dodawać** do stron WWW **zachowania**. Zapomnij o suchych, nudnych i statycznych stronach — korzystając z języka JavaScript, będziesz w stanie porozumiewać się ze swoimi użytkownikami, pobierać z internetu dane do wyświetlania na swoich stronach, rysować grafikę bezpośrednio na nich i robić wiele innych świetnych rzeczy. Co więcej, znając JavaScript, będziesz także mógł zapewniać swoim użytkownikom **całkowicie nowe** możliwości.

Jako programista posługujący się językiem JavaScript znajdziesz się w dobrym towarzystwie — jest on nie tylko jednym z **najbardziej popularnych** języków programowania, lecz także językiem **obsługiwanym** przez wszystkie nowoczesne przeglądarki i stosowanym w wielu środowiskach, które nie są powiązane z przeglądarkami. Więcej o tym napiszemy dalej w tej książce, a na razie bierzmy się do roboty!

pisanie prawdziwego kodu

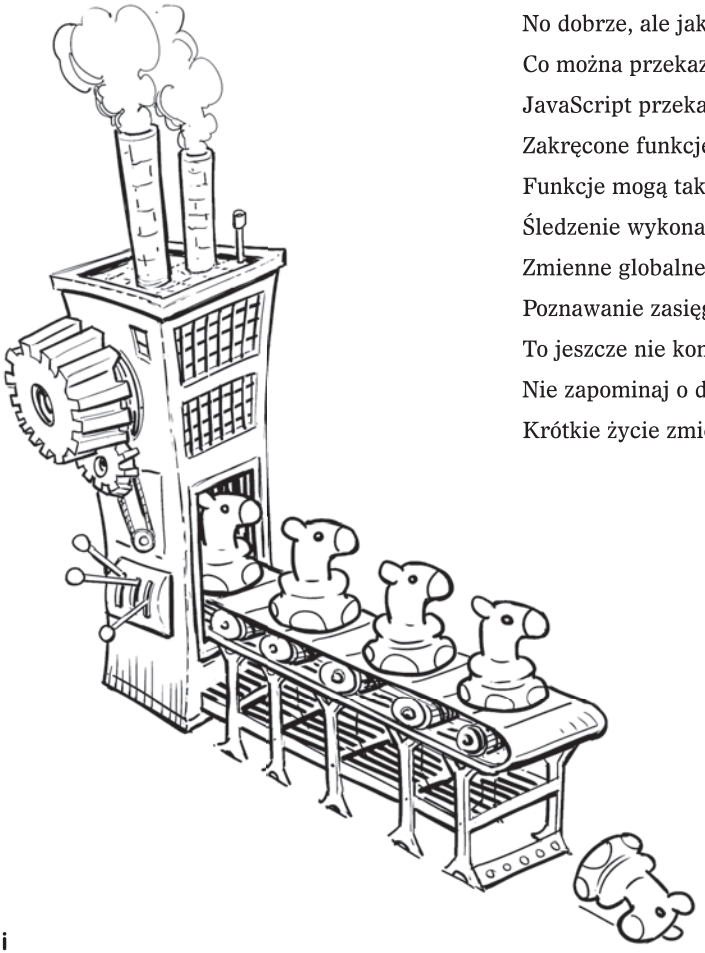
2 Idziemy dalej

Znasz już zmienne, typy, wyrażenia..., możesz więc pójść o krok dalej. Chodzi o to, że już trochę poznałeś język JavaScript. Wiesz na tyle dużo, by móc napisać jakiś **prawdziwy kod**. Kod, który robi coś interesującego, którego ktoś chciałby używać. Wciąż brakuje Ci jednak **praktycznego doświadczenia** w pisaniu kodu. Mamy zamiar temu zaradzić właśnie tu i teraz! A w jaki sposób? Skacząc na główkę na głęboką wodę i pisząc prostą grę — w całości w języku JavaScript. Cel jest bardzo ambitny, jednak będziemy go realizować krok po kroku. Chodź, zaczynamy! Jeśli zechcesz przy okazji uruchomić kolejny start-up, nie będziemy Ci przeszkadzać — kod jest Twój.

przedstawienie funkcji

3 Stawiamy na funkcjonalność

Przygotuj się na użycie pierwszej ze swoich supermocy. Zdobyłeś już nieco umiejętności programistycznych; teraz nadszedł czas, aby rozwinąć je jeszcze bardziej przy użyciu **funkcji**. Funkcje zapewniają możliwość pisania kodu, który można stosować we wszelkich możliwych okolicznościach, kodu **używanego wielokrotnie**, którym można znacznie łatwiej **zarządzać** i, wreszcie, który można **wyodrębnić**, nadać mu łatwą do zapamiętania nazwę, a potem zapomnieć o całej jego złożoności i zająć się innymi ważnymi problemami. Przekonasz się, że funkcje to nie tylko droga, która zmieni Cię z autora skryptów w programistę. Są one kluczowym czynnikiem określającym styl programowania w języku JavaScript. W tym rozdziale zaczniemy od podstaw: poznasz mechanikę funkcji i tajniki ich działania, a dalej w tej książce będziesz stopniowo powiększać swoją wiedzę i umiejętności ich stosowania. Zacznij więc budować solidne podstawy znajomości JavaScriptu i zrób to *już teraz*.



porządkowanie naszych danych

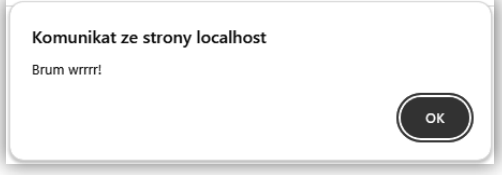
4 Tablice

JavaScript to nie tylko liczby, łańcuchy znaków i wartości logiczne. Dotychczas pisałeś jedynie kod JavaScript, w którym były używane wartości **typów prostych** — proste łańcuchy znaków, liczby i wartości logiczne, takie jak `"Fido"`, `23` oraz `true`. Korzystając z takich wartości, można zrobić naprawdę dużo, jednak w którymś momencie będziesz musiał zacząć posługiwać się znacznie **większą ilością danych**. Przykładowo mogą to być wszystkie produkty umieszczone w koszyku zakupowym albo utwory na liście odtwarzania, albo gwiazdozbiory i współrzędne poszczególnych gwiazd, albo cały katalog produktów. Do tego potrzebujesz jednak czegoś bardziej... *sexy*. W języku JavaScript preferowanym typem danych dla takich uporządkowanych zbiorów informacji jest **tablica**, a w tym rozdziale dokładnie przeanalizujemy, jak umieszczać dane w tablicach, przekazywać tablice oraz jak na nich operować. Dalej w książce omówimy także kilka innych sposobów **strukturyzacji danych**, jednak zaczniemy od tablic.

zrozumieć obiekty

5 Wycieczka do Obiektowa

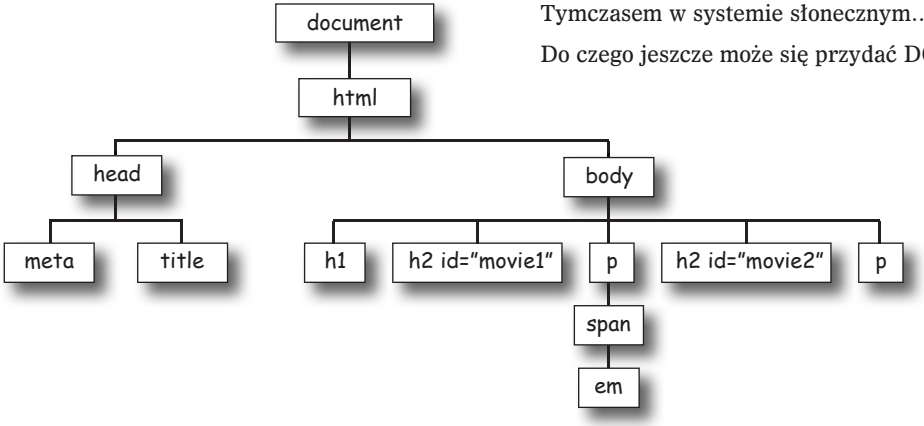
Do tej pory w tworzonym kodzie używałeś jedynie danych typów podstawowych oraz tablic. Dodatkowo podchodziłeś do programowania w **sposób proceduralny** — korzystałeś z prostych instrukcji, instrukcji warunkowych oraz pętli, ewentualnie umieszczałeś je w funkcjach — to właściwie nie jest **programowanie obiektowe**. Prawdę powiedziawszy, to *w ogóle* nie jest programowanie obiektowe! Tu i tam, nawet o tym nie wiedząc, użyłeś kilku obiektów, jednak na razie jeszcze nie napisałeś żadnego własnego obiektu. Najwyższy czas, żeby zostawić to stare i nudne proceduralne miasteczko i zacząć tworzyć własne **obiekty**. W tym rozdziale dowiesz się, dlaczego stosowanie obiektów sprawi, że Twoje życie stanie się znacznie lepsze — przynajmniej pod względem **programistycznym** (niestety, w jednej książce nie możemy poprawić Twojej znajomości mody *i* jednocześnie nauczyć programowania w języku JavaScript). I jeszcze jedno ostrzeżenie: kiedy już poznasz obiekty, nigdy nie będziesz chciał ich porzucić. Wyślij nam pocztówkę, kiedy już dojedziesz do krainy obiektów.



interakcja ze stronami WWW

6 Poznajemy DOM

Przebyłeś już długą drogę, poznając JavaScript. Powoli z żółtodzioba zmieniłeś się w twórcę prostych skryptów, a potem w końcu w **programistę**. Jednak wciąż czegoś Ci brakuje. Abyś mógł naprawdę wykorzystać całą swoją znajomość języka JavaScript, musisz się dowiedzieć, jak prowadzić interakcję ze stronami WWW, w których umieszczasz swoje skrypty. Tylko to pozwoli Ci tworzyć strony, które są **dynamiczne**, które reagują, odpowiadają i aktualizują swoją zawartość już po jej wczytaniu przez przeglądarkę. W jaki sposób można prowadzić interakcję ze stroną WWW? Służy do tego **DOM**, nazywany także **modelem obiektów dokumentu**. W tym rozdziale opiszemy go szczegółowo i wyjaśnimy, jak można z niego korzystać i jak go używać wraz z językiem JavaScript, by nauczyć nasze strony wykonywania wielu nowych sztuczek.



typy, równość, konwersje i cały ten jazz

7 Poważne typy

Nadszedł czas, by poważnie przyjrzeć się typom. Jedną ze wspaniałych cech JavaScriptu jest to, że można w nim zrobić całkiem dużo bez jego szczegółowej znajomości. Aby jednak **perfekcyjnie opanować język**, dostać awans i zacząć robić to, co naprawdę chcesz robić w życiu, musisz się zaprzyjaźnić z **typami**. Czy pamiętasz, co już dawno, w pierwszym rozdziale tej książki, powiedzieliśmy na temat JavaScriptu? Że nie może się poszczycić rozpuszczoną, popularną, akademicką definicją? No cóż... To prawda, jednak brak życia studenckiego nie zatrzymał ani Steve'a Jobsa, ani Billa Gatesa, nie zatrzymał także języka JavaScript. Oznacza to jednak, że JavaScript nie ma... hm... doskonale przemyślanego systemu typów i znajdziemy w nim sporo **dziwactw**. Nie obawiaj się jednak, w tym rozdziale dokładnie wszystko wyjaśnimy, dzięki czemu już niebawem nauczysz się unikać tych zawstydzających problemów z typami.

łączenie wszystkiego w całość

8 Tworzenie aplikacji

Włóż to do swojego przybornika z narzędziami, czyli do skrzynki, w której umieszczasz wszystkie swoje nowe umiejętności programistyczne, wiedzę dotyczącą modelu DOM, a nawet HTML-a i CSS-a. W tym rozdziale wykorzystasz wszystkie te umiejętności, by utworzyć pierwszą prawdziwą **aplikację internetową**. Wystarczy już **prymitywnych namiastek gier**, w których na jednowymiarowej planszy pływa jeden okręt. W tym rozdziale wykonasz **pełne doświadczenie**: dużą, atrakcyjną planszę do gry, wiele okrętów oraz obsługę wprowadzania danych przez użytkownika. Struktura strony, na której będzie prowadzona gra, powstanie w języku HTML, jej wygląd określony zostanie przy użyciu stylów CSS, a zachowanie samej gry będzie napisane w języku JavaScript. Przygotuj się: to będzie jazda na całego, rozdział, w którym pełnym gazem będziesz zmierzać w kierunku pisania naprawdę poważnego kodu.

obsługa zdarzeń

9 Programowanie asynchroniczne

Po przeczytaniu tego rozdziału zdasz sobie sprawę z tego, że to nie są przelewki i nie jesteśmy już w Kansas. Do tej pory pisałeś kod, który zazwyczaj wykonywany był od samego początku do końca — oczywiście Twój kod mógł być nieco bardziej skomplikowany i zawierać kilka funkcji, obiektów i metod, jednak w jakimś momencie ten kod po prostu był wykonany wiersz po wierszu. Cóż, jest nam bardzo przykro, że mówimy Ci to tak późno, jednak **typowy kod JavaScript nie jest pisany w taki sposób**. Kod pisany w tym języku **reaguje na zdarzenia**. Jakiego rodzaju zdarzenia? Może to być kliknięcie strony przez użytkownika, przesłanie danych z serwera, upływ czasu w przeglądarce, jakaś zmiana wprowadzona w modelu DOM oraz wiele innych. W rzeczywistości w niewidoczny dla nas sposób w przeglądarce **cały czas** zachodzą jakieś zdarzenia. W tym rozdziale jeszcze raz przemyślisz swoje podejście do sposobu pisania kodu JavaScript i dowiesz się, dlaczego trzeba pisać kod reagujący na zdarzenia oraz jak należy to robić.

10

Funkcje anonimowe i funkcje wyższego rzędu

Wyzwolone funkcje

Poznaj funkcje, a potem baw się na całego. Każda sztuka, rzemiosło czy też dyscyplina sportowa ma swoją kluczową cechę, która odróżnia graczy przeciętnych od wirtuozów. W przypadku języka JavaScript jest to prawdziwe i dokładne zrozumienie funkcji. W języku JavaScript funkcje mają kluczowe znaczenie, a wiele technik służących do **projektowania i organizacji** kodu bazuje na zaawansowanej znajomości funkcji i umiejętności korzystania z nich. Droga prowadząca do poznania funkcji na takim poziomie jest interesująca i niejednokrotnie wymagająca dla mózgu, zatem dobrze się do niej przygotuj. W tym rozdziale będziesz się czuł jak dziecko oprowadzane przez Willy'ego Wonkę po fabryce czekolady — będziesz w nim kontynuował poznawanie funkcji w języku JavaScript, a przy okazji zobaczysz dziwaczne, wariackie i cudowne rzeczy.

nowoczesna składnia, zasięg leksykalny i domknięcia

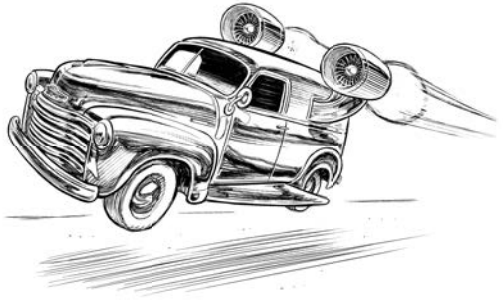
11 Poważne funkcje

W poprzednim rozdziale rozłożyłeś funkcje na czynniki pierwsze, ale wciąż musisz się o nich jeszcze sporo dowiedzieć. W tym rozdziale pójdziemy jeszcze dalej — to będzie prawdziwa jazda na całego. Pokażemy Ci, jak bawić się ze składnią, dzięki zastosowaniu zaawansowanych technik obsługi argumentów, parametrów i przypisań. Następnie przyjrzymy się zasięgom i niektórym z najdrobniejszych szczegółów związanych z zarządzeniem nimi w języku JavaScript. Ta podróż przez niuanse zasięgów doprowadzi nas do serca domknięć (ang. *closures*) — koncepcji często owianej tajemnicą, lecz jednocześnie kluczowej dla opanowania języka JavaScript. Po przeczytaniu tego rozdziału będziesz potrafił pisać bardziej ekspresyjny kod JavaScript, niż myślałeś, że to możliwe.

zaawansowane sposoby konstruowania obiektów

12 Tworzenie obiektów

Dotychczas wszystkie obiekty tworzyłeś własnoręcznie. Opracowując każdy z nich, korzystałeś z **literału obiektowego**, w którym podawałeś wszystkie właściwości i metody. Na niewielką skalę takie rozwiązanie będzie się sprawdzać, jednak podczas tworzenia poważnego kodu będziesz potrzebować czegoś lepszego. Właśnie w tym miejscu do akcji wkraczają **klasy**. Dzięki klasom tworzenie obiektów jest znacznie łatwiejsze, a wszystkie budowane obiekty mogą być zgodne z jednym **wzorcem** — oznacza to, że klasy zapewniają, że wszystkie obiekty będą miały te same właściwości i metody. Kiedy korzystasz z klas, kod obiektów może być znacznie bardziej **zwięzły** i mniej podatny na występowanie błędów, zwłaszcza gdy tworzysz bardzo dużo obiektów. Bierzmy się zatem do pracy...



pozostałości

A Dziesięć najważniejszych zagadnień (których nie opisaliśmy)

Opisaliśmy naprawdę sporo zagadnień i już niemal udało Ci się skończyć tę książkę. Będziemy za Tobą tęsknili, jednak zanim pozwolimy Ci odejść, musimy jeszcze coś powiedzieć, bo nie czulibyśmy się w porządku, wypuszczając Cię w świat bez tych informacji. Nie ma możliwości, byśmy w tym stosunkowo niewielkim rozdziale zdołali zmieścić wszystko, co ewentualnie mogłoby się przydać. Prawdę mówiąc, wcześniej *zamieściliśmy* w tym rozdziale wszystko to, co przydałoby się, żebyś wiedział o programowaniu w języku JavaScript, lecz musieliśmy zmniejszyć rozmiar czcionki do 0,00004 punktu. Wszystko się zmieściło, ale nikt nie był w stanie tego przeczytać. Dlatego większość tego tekstu odrzuciliśmy, pozostawiając tu jedynie dziesięć najważniejszych zagadnień.

Idziemy dalej

Znasz już zmienne, typy, wyrażenia…, możesz więc pójść o krok dalej. Chodzi o to, że już trochę poznałeś język JavaScript. Wiesz na tyle dużo, by móc napisać jakiś **prawdziwy kod**. Kod, który robi coś interesującego, którego ktoś chciałby używać. Wciąż brakuje Ci jednak **praktycznego doświadczenia** w pisaniu kodu. Mamy zamiar temu zaradzić właśnie tu i teraz! A w jaki sposób? Skacząc na główkę na głęboką wodę i pisząc prostą grę — w całości w języku JavaScript. Cel jest bardzo ambitny, jednak będziemy go realizować krok po kroku. Chodź, zaczynamy! Jeśli zechcesz przy okazji uruchomić kolejny start-up, nie będziemy Ci przeszkadzać — kod jest Twój.

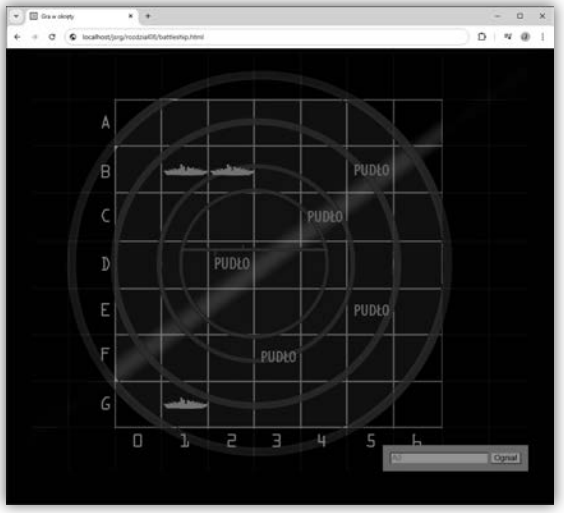
Napiszmy grę w okręty

To będzie pojedynek między Tobą a przeglądarką: przeglądarka ukrywa okręty, a Twoim zadaniem jest je znaleźć i zniszczyć. Oczywiście, w odróżnieniu od tradycyjnej wersji gry Ty nie będziesz miał żadnych własnych okrętów. Twoim zadaniem jest odnalezienie i zatopienie okrętów przeglądarki w jak najmniejszej liczbie prób.

Cel: zatopić statki przeglądarki w jak najmniejszej liczbie prób. Po zakończeniu rozgrywki gra wyświetli wynik, zależny od tego, jak dobrze sobie poradziłeś.

Przygotowania: kiedy gra zostanie wczytana, komputer rozmieszcza swoje statki na wirtualnej planszy w kształcie siatki. Po ich rozmieszczeniu gracz jest proszony o pierwszy ruch.

Przebieg gry: przeglądarka wyświetla okno z prośbą o podanie sprawdzanego pola na planszy. W odpowiedzi na ruch gracza wyświetlane są komunikaty: „Trafienie”, „Pudło” lub „Zatopiłeś mój okręt!”.



To jest to, do czego dążymy: ładna siatka 7×7 z trzema okrętami do zatopienia. Teraz zaczniemy od czegoś prostszego, jednak kiedy poznasz JavaScript nieco lepiej, dokończymy grę, by wyglądała jak ta powyżej — z atrakcyjną grafiką i wszystkimi innymi elementami… Obsługę dźwięku zostawimy Ci jako ćwiczenie do wykonania we własnym zakresie.

Pierwsza próba…

…uproszczony okręt

W pierwszym podejściu spróbujemy napisać coś prostszego od pełnej wersji gry z graficzną planszą 7×7 i trzema statkami. Zaczniemy skromnie od jednowymiarowej siatki składającej się z siedmiu pól i jednego statku. Będzie to prosta wersja, jednak celem w tym momencie jest zaprojektowanie podstawowego kodu gry, a nie jej finalnego wyglądu i sposobu działania.

Nie przejmuj się; zaczynając od takiej uproszczonej wersji gry, zapewniasz sobie świetną pozycję wyjściową do późniejszych prac nad pełną wersją. Dodatkowo będzie to doskonały krok do napisania pierwszego prawdziwego programu w języku JavaScript (oczywiście, nie licząc „poważnej aplikacji biznesowej” z rozdziału 1., odliczającej piwa korzenne). Dlatego też w tym rozdziale napiszemy prostszą wersję gry, a pełną wersją zajmiemy się dalej w tej książce, kiedy już poznasz JavaScript nieco lepiej.

Zamiast siatki 7×7, jak pokazana powyżej, zaczniemy od siatki 1×7. I będziemy poszukiwać tylko jednego okrętu.

Zauważ, że każdy okręt zajmuje dokładnie trzy komórki siatki (podobnie jak w prawdziwej grze).

Zacznijmy od projektu wysokiego poziomu

Doskonale wiemy, że będziemy potrzebować zmiennych, a do tego liczb, łańcuchów znaków, instrukcji `if`, wyrażeń warunkowych i pętli… Jednak gdzie i ile ich będzie? I jak te wszystkie elementy połączymy w jeden program? Aby odpowiedzieć na te pytania, potrzebujemy więcej informacji na temat tego, co gra powinna robić.

Najpierw powinniśmy określić ogólny przepływ sterowania w grze. Podstawowy pomysł jej działania wygląda następująco:

1 Użytkownik uruchamia grę.

A Gra umieszcza okręt w losowym miejscu siatki.

2 Zaczyna się rozgrywka.

Powtarzamy poniższe czynności aż do zatopienia okrętu.

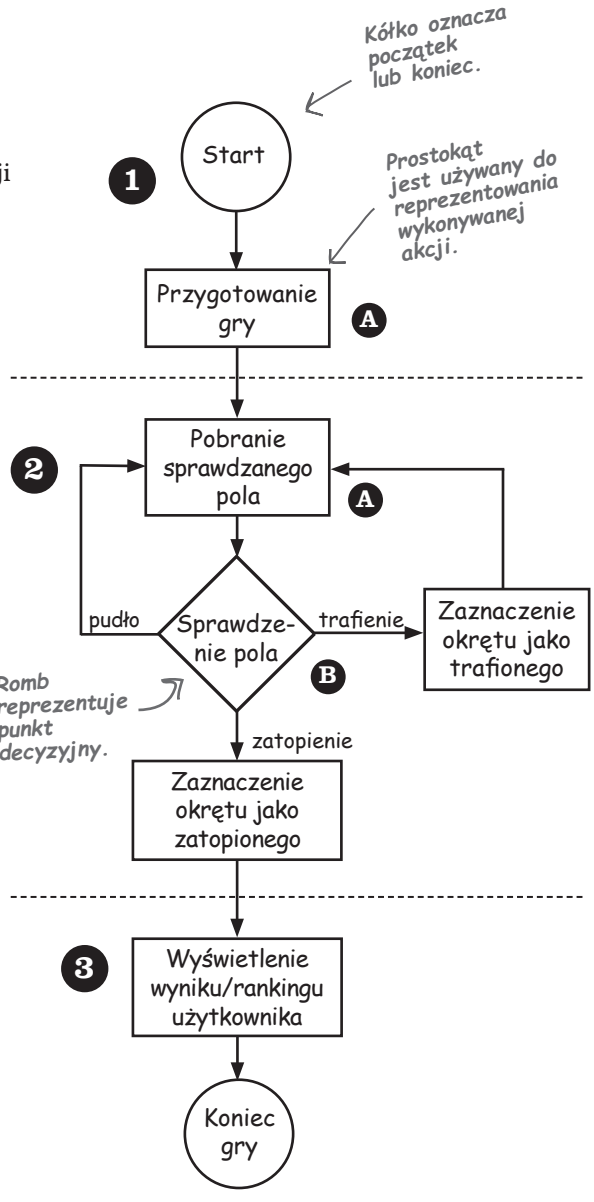
A Pytamy użytkownika o sprawdzane pole siatki („2", „0" itd.).

B Porównujemy podane pole z położeniem okrętu, określając przy tym, czy użytkownik trafił, spudłował, czy też zatopił okręt.

3 Koniec gry.

Wyświetlamy klasyfikację użytkownika określoną na podstawie liczby prób.

Teraz mamy już ogólne pojęcie o tym, co program musi robić. Później określimy kilka dodatkowych szczegółów dotyczących czynności wykonywanych w poszczególnych krokach.



No i proszę. Prawdziwy schemat blokowy

Kilka dodatkowych szczegółów

Po przeanalizowaniu ogólnego projektu już całkiem dobrze wiemy, jak gra ma działać, i dysponujemy profesjonalnym schematem blokowym, jednak zanim zajmiemy się pisaniem kodu, warto ustalić jeszcze kilka szczegółów.

Reprezentowanie okrętu

Zacznijmy od określenia sposobu reprezentowania statku na siatce. Musisz pamiętać, że siatka ta jest, powiedzmy to wyraźnie, *wirtualna*. Innymi słowy, nie istnieje nigdzie w programie. Dopóki zarówno gra, jak i użytkownik wiedzą, że okręt jest ukryty w trzech sąsiadujących komórkach siatki (przy czym komórek jest siedem, a pierwsza z nich ma numer 0), dopóty sama siatka nie musi być reprezentowana w kodzie. Możesz mieć ochotę, by utworzyć coś, co pozwoli zapisać te siedem komórek, a następnie oznaczyć trzy z nich jako zajęte przez okręt, jednak nie jest to konieczne. Musimy jedynie wiedzieć, jakie komórki siatki zajmuje okręt, np. 1., 2. i 3.

Pobieranie danych od użytkownika

A co z pobieraniem danych od użytkownika? Można w tym celu użyć funkcji `prompt`. Zawsze wtedy, gdy będziemy musieli pobrać nową komórkę do sprawdzenia, użyjemy tej funkcji, by wyświetlić komunikat i pobrać liczbę z zakresu od 0 do 6.

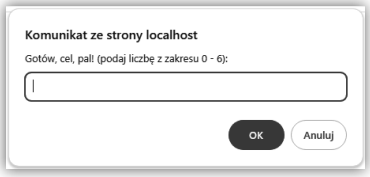
Wyświetlanie wyników

A wyniki? Na razie do prezentacji wyników gry będziemy korzystać z funkcji `alert`. Jest to nieco toporne rozwiązanie, ale działa. (W pełnej wersji gry zamieszczonej dalej w tej książce będziemy modyfikować samą stronę WWW, jednak zanim do tego dojdziemy, czeka nas jeszcze długa droga).

1 **Gra po uruchomieniu** tworzy okręt i umieszcza go w trzech sąsiadujących ze sobą komórkach wiersza składającego się w sumie z siedmiu komórek.
Komórki te są jedynie liczbami całkowitymi; np. okręt pokazany na poniższym rysunku zajmuje komórki 1., 2. i 3.

2 **Zaczyna się rozgrywka.** Prosimy użytkownika o wytypowanie komórki.

A



B Sprawdzamy, czy użytkownikowi udało się trafić w którąkolwiek z komórek zajmowanych przez okręt. Liczbę trafień należy zapamiętywać w jakiejś zmiennej.

3 **Gra się kończy,** kiedy wszystkie trzy komórki zajmowane przez okręt zostaną trafione i wartość zmiennej przechowującej liczbę trafień osiągnie 3. W tym momencie wyświetlamy użytkownikowi informację, ilu prób potrzebował, by zatopić okręt.

Przykładowa interakcja z grą

Analiza pseudokodu

Musimy mieć jakiś sposób na planowanie i zapisywanie naszego kodu. Zaczniemy od napisania *pseudokodu*. Pseudokod to coś pomiędzy normalnym kodem w języku JavaScript i zwyczajnymi zdaniami opisującymi działanie programu. Jak się zaraz przekonasz, pomoże Ci to przemyśleć i zaplanować poszczególne czynności, które program ma wykonywać, bez konieczności pisania *faktycznego kodu*.

Poniższy pseudokod reprezentujący prostą wersję gry w okręty składa się z sekcji opisującej potrzebne zmienne oraz sekcji opisującej logikę programu. Zmienne informują o tym, jakie informacje musimy śledzić i przechowywać w kodzie; natomiast logika opisuje, jaki kod musimy wiernie zaimplementować, by powstała działająca gra.

ZADEKLARUJ trzy *zmienne* do przechowywania położenia każdej z trzech komórek okrętu. Nazwij je `location1`, `location2` oraz `location3`.

ZADEKLARUJ *zmienną* przechowującą komórkę wskazaną przez użytkownika do sprawdzenia. Nazwij ją `guess`.

ZADEKLARUJ *zmienną* przechowującą liczbę trafień. Nazwij ją `hits` i początkowo *przypisz* jej wartość 0.

Zmienne, których będziemy potrzebować.

ZADEKLARUJ *zmienną* przechowującą liczbę prób. Nazwij ją `guesses` i początkowo *przypisz* jej wartość 0.

ZADEKLARUJ *zmienną* przechowującą informację, czy okręt został zatopiony, czy nie. Nazwij ją `isSunk` i *przypisz* jej wartość początkową `false`.

PĘTLA WHILE: *dopóki okręt nie jest zatopiony*

  POBIERZ komórkę do sprawdzenia

  A to jest logika gry.

  PORÓWNAJ podaną wartość z dopuszczalnymi, prawidłowymi wartościami

  JEŚLI wartość podana przez użytkownika nie jest prawidłową liczbą:

    POPROŚ użytkownika o podanie prawidłowej liczby

  W PRZECIWNYM RAZIE:

    DODAJ jeden do wartości zmiennej `guesses`

    JEŚLI położenie podane przez użytkownika odpowiada jednej z komórek okrętu:

      DODAJ jeden do liczby trafień

      JEŚLI liczba trafień wynosi 3:

        USTAW wartość zmiennej `isSunk` na `true`

        WYŚWIETL komunikat „Zatopiłeś mój okręt!”

      KONIEC JEŚLI

    KONIEC JEŚLI

  KONIEC W PRZECIWNYM RAZIE

KONIEC PĘTLI WHILE

WYŚWIETL statystyki użytkownika

To nie jest JavaScript, ale już zapewne widzisz, w jaki sposób zacząć implementować tę logikę w kodzie.

Zauważ, że używamy odpowiednich wcięć, żeby ułatwić Ci analizę pseudokodu. Dokładnie w taki sam sposób będziemy formatować faktyczny kod gry.

Zaostrz ołówek

Załóż, że nasza wirtualna siatka wygląda w następujący sposób:

A komórki zajmowane przez okręt są zapisane w trzech zmiennych lokalnych.

```
location1 = 3;
location2 = 4;
location3 = 5;
```

Przyjmij, że użytkownik wpisał następującą sekwencję sprawdzanych komórek:

```
1, 4, 2, 3, 5
```

A teraz, opierając się na pseudokodzie przedstawionym na poprzedniej stronie, przejdź każdy krok kodu i zobacz, jak będzie wykonywany w przypadku podania powyższej sekwencji danych wejściowych. Swoje uwagi zapisz niżej. Kilka początkowych kroków zapisaliśmy za Ciebie. Jeśli pierwszy raz analizujesz i próbujesz prześledzić działanie pseudokodu, nie spiesz się i spróbuj go dobrze zrozumieć.

Jeśli potrzebujesz wskazówki, zajrzyj do naszej odpowiedzi podanej pod koniec tego rozdziału.

location1	location2	location3	guess	guesses	hits	isSunk
3	4	5	—	0	0	false
3	4	5	1	1	0	false

Pierwszy wiersz pokazuje początkowe wartości zmiennych, zanim użytkownik podał pierwszą sprawdzaną komórkę siatki.

→ Rozwiązanie na stronie 74

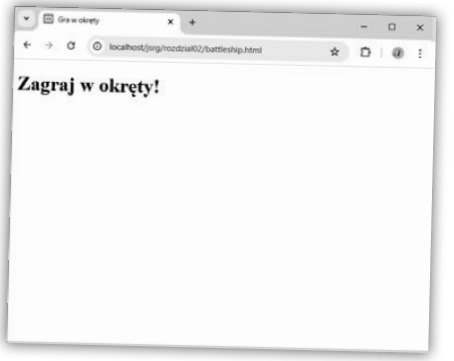
Zanim przejdziemy dalej, nie zapomnij o kodzie HTML!

Nie uda Ci się dojść daleko, jeśli zapomnisz o kodzie HTML, do którego będziesz mógł dołączyć kod JavaScript. Wpisz zatem poniższy kod w edytorze i zapisz go w nowym pliku o nazwie „battleship.html". Kiedy to zrobisz, wrócimy do pisania kodu aplikacji.

Kod HTML strony gry w okręty jest bardzo prosty; strona ta jest potrzebna wyłącznie do dołączenia pliku JavaScript, bo to właśnie w nim wszystko będzie się działo.

```
<!doctype html>
<html lang="pl">
  <head>
    <title>Gra w okręty</title>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>Zagraj w okręty!</h1>
    <div id="results"></div>
    <script src="battleship.js"></script>
  </body>
</html>
```

Odwołanie do pliku JavaScript zostało umieszczone u dołu elementu <body>; kiedy zatem przeglądarka zacznie wykonywać kod JavaScript, strona już będzie wyświetlona.



A to zobaczysz po wyświetleniu strony w przeglądarce. Musimy napisać jakiś kod, który rozpocznie i przeprowadzi rozgrywkę!

MOC UMYSŁU

Rozruszaj swoje dendryty.

Być może wyprzedzamy nieco fakty, ale spróbuj się zastanowić, jakiego rodzaju kodu można by użyć w celu wygenerowania losowego położenia okrętu, tak by po każdym wczytaniu strony w przeglądarce było ono inne. Jakie czynniki trzeba wziąć pod uwagę, by prawidłowo umieścić okręt? Zachęcamy, żebyś swoje pomysły zapisał poniżej.

Pisanie kodu prostej wersji gry w okręty

Przedstawiony wcześniej pseudokod wykorzystamy jako wzorzec do napisania prawdziwego kodu JavaScript. Najpierw zajmiemy się wszystkimi niezbędnymi zmiennymi. Spójrz jeszcze raz na pseudokod:

Potrzebujemy tych trzech zmiennych, aby zapisać położenie okrętu.

ZADEKLARUJ trzy *zmienne* do przechowywania położenia każdej z trzech komórek okrętu. Nazwij je `location1`, `location2` oraz `location3`.

ZADEKLARUJ *zmienną* przechowującą komórkę wskazaną przez użytkownika do sprawdzenia. Nazwij ją `guess`.

ZADEKLARUJ *zmienną* przechowującą liczbę trafień. Nazwij ją `hits` i początkowo *przypisz* jej wartość 0.

ZADEKLARUJ *zmienną* przechowującą liczbę prób. Nazwij ją `guesses` i początkowo *przypisz* jej wartość 0.

Trzy kolejne zmienne (guess, hits oraz guesses) są związane z próbami zatopienia okrętu.

ZADEKLARUJ *zmienną* przechowującą informację, czy okręt został zatopiony, czy nie. Nazwij ją `isSunk` i *przypisz* jej wartość początkową `false`.

I jeszcze jedna, która przechowuje informację, czy okręt został zatopiony, czy nie.

A teraz zapiszmy te zmienne w pliku JavaScript. W tym samym katalogu, w którym wcześniej utworzyłeś plik „battleship.html", utwórz nowy plik o nazwie „battleship.js" i wpisz w nim poniższe deklaracje zmiennych:

Oto nasze trzy zmienne określające położenie okrętu. Nie będziemy się nad tym rozwodzić i po prostu przypiszemy im wartości 3, 4 i 5.

Dalej w tym rozdziale wrócimy do tych zmiennych i napiszemy kod, który generuje losowe położenie okrętu, by nieco utrudnić użytkownikowi zadanie.

Nie zdefiniujemy ich jako stałych, gdyż ich wartości będą później modyfikowane.

```
let location1 = 3;
let location2 = 4;
let location3 = 5;


let guess;
let hits = 0;
let guesses = 0;


let isSunk = false;
```

Zmienna guess nie będzie miała wartości aż do momentu, gdy użytkownik określi komórkę, którą chce sprawdzić.

Tym dwóm zmiennym przypiszemy początkowe wartości 0.

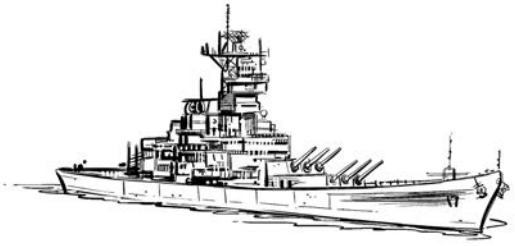
I w końcu zmiennej isSunk przypisujemy wartość false. Zmienimy ją na true, kiedy okręt zostanie zatopiony.

Kodowanie na poważnie

Jeśli nie przypiszemy zmiennej wartości początkowej, JavaScript przypisze jej wartość domyślną, którą jest `undefined`. Wartość tę można sobie wyobrazić jako stwierdzenie: „tej zmiennej nie została jeszcze przypisana żadna wartość". O `undefined` oraz kilku innych dziwnych wartościach porozmawiamy dalej w tej książce.

A teraz zajmijmy się logiką gry

Poradziliśmy sobie ze zmiennymi, możemy się zatem zająć implementacją pseudokodu opisującego logikę gry. Podzielimy go na kilka fragmentów. Pierwszą rzeczą, którą trzeba zrobić, jest zaimplementowanie pętli: pętla musi działać aż do zatopienia okrętu. Następnie zajmiemy się pobraniem od użytkownika numeru komórki, którą chce sprawdzić, i weryfikacją jego poprawności — no wiesz, musimy się upewnić, że jest to liczba z zakresu od 0 do 6. Później przejdziemy do napisania logiki, która sprawdzi trafienie okrętu i ustali, czy został zatopiony, czy nie. Na samym końcu zadbamy o wyświetlenie użytkownikowi niewielkiego raportu z informacją, ilu prób wymagało zatopienie okrętu.



Krok 1. Przygotować pętlę, pobrać dane od użytkownika i sprawdzić ich poprawność.

PĘTLA WHILE: *dopóki okręt nie jest zatopiony*
    POBIERZ komórkę do sprawdzenia
    PORÓWNAJ podaną wartość z dopuszczalnymi, prawidłowymi wartościami
    JEŚLI wartość podana przez użytkownika nie jest prawidłową liczbą
        POPROŚ użytkownika o podanie prawidłowej liczby
    W PRZECIWNYM RAZIE

Krok 2. Sprawdzić wskazaną komórkę. Czy mamy trafienie, czy pudło?

        DODAJ jeden do wartości zmiennej `guesses`
        JEŚLI położenie podane przez użytkownika odpowiada jednej z komórek okrętu
            DODAJ jeden do liczby trafień
            JEŚLI liczba trafień wynosi 3

Krok 3. Sprawdzić, czy okręt został zatopiony.

                USTAW wartość zmiennej `isSunk` na `true`
                WYŚWIETL komunikat „Zatopiłeś mój okręt!”
            KONIEC JEŚLI
        KONIEC JEŚLI
    KONIEC W PRZECIWNYM RAZIE
KONIEC PĘTLI WHILE

Krok 4. Wyświetlić użytkownikowi komunikat z wynikami.

WYŚWIETL statystyki użytkownika

Do zrobienia:

- [ ] Napisać pętlę i pobrać komórkę do sprawdzenia.
- [ ] Sprawdzić trafienie.
- [ ] Sprawdzić, czy okręt został zatopiony.
- [ ] Wyświetlić użytkownikowi statystyki gry.

Krok 1. Przygotowanie pętli i pobranie danych

- [ ] Napisać pętlę i pobrać komórkę do sprawdzenia.
- [ ] Sprawdzić trafienie.
- [ ] Sprawdzić, czy okręt został zatopiony.
- [ ] Wyświetlić użytkownikowi statystyki gry.

Teraz zaczniemy przekształcać logikę gry opisaną w pseudokodzie na prawdziwy kod JavaScript. Odwzorowanie pseudokodu na kod JavaScript nie jest bezpośrednie, zatem tu i tam pojawią się drobne różnice. Pseudokod ma przedstawiać ideę tego, *co* kod powinien robić, a teraz musimy napisać kod JavaScript, który będzie wiedział, *jak* to wykonać.

Zacznijmy od kodu, jakim już dysponujemy, a później zajmiemy się tylko tymi fragmentami, które musimy dodać (aby zaoszczędzić kilka drzew tu i tam lub kilka elektronów, jeśli czytasz tę książkę w wersji elektronicznej).

Zmiennymi już się zajmowaliśmy, jednak dołączyliśmy je tutaj, by niczego nie pominąć.

ZADEKLARUJ zmienne

```
let location1 = 3;
let location2 = 4;
let location3 = 5;
let guess;
let hits = 0;
let guesses = 0;
let isSunk = false;
```

To jest początek pętli. Dopóki okręt nie jest zatopiony, kontynuujemy grę; a zatem wykonujemy pętlę.

Pamiętaj, że instrukcja while używa wyrażenia warunkowego, by określić, czy pętla ma zostać wykonana. W tym przypadku sprawdzamy, czy zmienna isSunk ma wartość false. Gdy okręt zostanie zatopiony, przypiszemy jej wartość true.

PĘTLA: *dopóki okręt nie jest zatopiony*

```
while (isSunk == false) {
```

POBIERZ komórkę do sprawdzenia

```
   guess = prompt("Gotów, cel, pal! (podaj liczbę z zakresu 0 - 6):");

}
```

Za każdym razem, gdy zaczniemy wykonywać pętlę while, poprosimy użytkownika o podanie komórki, którą chce sprawdzić. Do tego celu użyjemy wbudowanej funkcji prompt. Więcej informacji na jej temat znajdziesz na następnej stronie…

MOC UMYSŁU

Czy jeśli uruchomisz ten kod, gra kiedykolwiek się zakończy?

Jak działa funkcja prompt?

Przeglądarka udostępnia wbudowaną funkcję pozwalającą na pobieranie danych od użytkowników; nosi ona nazwę `prompt`. W dużym stopniu przypomina funkcję `alert` — podobnie jak ona, powoduje wyświetlenie okna dialogowego z podanym komunikatem. Jednak oprócz tego w oknie znajduje się pole tekstowe, w którym użytkownik może coś wpisać. Ta odpowiedź użytkownika, w formie łańcucha znaków, jest przekazywana jako wynik wywołania funkcji `prompt`. Jeśli jednak użytkownik zamknie okno dialogowe, zwrócona zostanie wartość `null`.

W tym miejscu przypisujemy wynik wywołania funkcji prompt do zmiennej guess.

```
guess = prompt("Gotów, cel, pal! (podaj liczbę z zakresu 0 - 6):");
```

Do funkcji prompt przekazywany jest łańcuch znaków, który następnie jest wyświetlany w oknie dialogowym jako instrukcje dla użytkownika.

Głównym zadaniem funkcji prompt jest pobranie danych od użytkownika. W zależności od urządzenia oznacza to zazwyczaj wyświetlenie okna dialogowego.

"5"

Kiedy funkcja pobierze już dane wejściowe od użytkownika, zwraca je do kodu, który ją wywołał. W naszym przypadku dane wejściowe, w formie łańcucha znaków, są zapisywane w zmiennej guess.

Może Cię kusić, by wypróbować ten nowy kod...

...jednak nie rób tego. Jeśli to zrobisz, przeglądarka zacznie wykonywać pętlę *nieskończoną*; będzie bezustannie prosić o podanie numeru komórki do sprawdzenia, jednak nie dysponuje jeszcze możliwością zakończenia realizacji pętli (może z wyjątkiem skorzystania z możliwości usunięcia procesu przeglądarki z wykorzystaniem narzędzi systemowych).

Krok 2. Sprawdzanie komórki wskazanej przez użytkownika

- [x] Napisać pętlę i pobrać komórkę do sprawdzenia.
- [ ] Sprawdzić trafienie.
- [ ] Sprawdzić, czy okręt został zatopiony.
- [ ] Wyświetlić użytkownikowi statystyki gry.

Jeśli zajrzysz do pseudokodu, zauważysz, że pierwszą rzeczą, jaką należy zrobić w celu sprawdzenia komórki podanej przez użytkownika, jest upewnienie się, iż informacje podane przez niego są prawidłowe. Jeśli takie będą, trzeba sprawdzić, czy udało się trafić okręt, czy też próba użytkownika zakończyła się pudłem. Oprócz tego trzeba odpowiednio zaktualizować wartości zmiennych `guesses` i `hits`. Zacznijmy zatem od sprawdzenia poprawności danych wejściowych i jeśli okażą się prawidłowe, powiększymy o `1` wartość zmiennej `guesses`. Kiedy to zrobimy, sprawdzimy, czy próba użytkownika zakończyła się trafieniem, czy pudłem.

```
// Tu są umieszczone deklaracje zmiennych

while (isSunk == false) {
    guess = prompt("Gotów, cel, pal! (podaj liczbę z zakresu 0 - 6):");
    if (guess < 0 || guess > 6) {
        alert("Proszę podać prawidłowy numer komórki!");
    } else {
        guesses = guesses + 1;
    }
}
```

Poprawność sprawdzamy, upewniając się, że podana liczba jest z zakresu od 0 do 6.

Jeśli podana liczba nie jest prawidłowa, poinformujemy o tym użytkownika, wyświetlając stosowny komunikat.

A jeśli liczba jest poprawna, przechodzimy dalej i dodajemy jeden do zmiennej guesses, by dysponować prawidłową liczbą prób wykonanych przez użytkownika.

Przyjrzyjmy się nieco dokładniej warunkom sprawdzającym poprawność podanej liczby. Już wiesz, że mamy sprawdzić, czy należy ona do przedziału od 0 do 6; ale jak dokładnie działa to wyrażenie warunkowe? Przeanalizujemy je fragment po fragmencie.

Spróbujmy to przeczytać w sposób zrozumiały dla człowieka: to wyrażenie będzie prawdziwe, jeśli wartość zmiennej guess będzie mniejsza od zera LUB jeśli wartość guess będzie większa od sześciu. Jeśli którykolwiek z tych warunków zostanie spełniony, będzie to oznaczało, że dane wejściowe wpisane przez użytkownika nie są prawidłowe.

```
if (guess < 0 || guess > 6) {
```

Naprawdę są to jedynie dwa, połączone ze sobą, proste warunki. Pierwszy z nich sprawdza, czy wartość zmiennej guess jest mniejsza od zera.

A ten warunek sprawdza, czy wartość guess jest większa od 6.

A to jest operator alternatywy logicznej — OR — łączący ze sobą dwa warunki w taki sposób, że całe wyrażenie przyjmie wartość true, jeśli którykolwiek z warunków będzie mieć wartość true. Jeśli jednak oba warunki będą mieć wartość false, to także całe wyrażenie przyjmie wartość false — a to będzie oznaczać, że podana liczba jest z zakresu od 0 do 6, czyli jest prawidłowa.

Nie ma głupich pytań

P: Zauważyłem, że w oknie wyświetlanym przy użyciu funkcji `prompt` jest przycisk *Anuluj*. Jaką wartość zwraca funkcja, jeśli klikniemy ten przycisk?

O: Jeśli klikniemy przycisk anulujący okno dialogowe, funkcja `prompt` nie zwróci łańcucha znaków, lecz wartość `null`. Już wiesz, że wartość ta oznacza „brak wartości", co jest odpowiednie w tej sytuacji, gdyż użytkownik zamknął okno bez podawania wartości. Moglibyśmy skorzystać z faktu, że wartość zwrócona przez funkcję `prompt` wynosi `null`, by sprawdzać, czy użytkownik nie kliknął przycisku *Anuluj*, a jeśli tak się stało, to moglibyśmy np. zakończyć grę. W naszym kodzie tego nie robimy, jednak warto zapamiętać to rozwiązanie, gdyż możemy z niego skorzystać dalej w tej książce.

P: Powiedzieliście, że funkcja `prompt` zawsze zwraca łańcuch znaków. W jaki zatem sposób możemy porównywać wartość łańcuchową, taką jak "0" lub "6", z liczbami, takimi jak 0 i 6?

O: Aby w takim przypadku wykonać porównania `guess < 6` oraz `guess > 0`, JavaScript próbuje przekonwertować łańcuch znaków zapisany w zmiennej `guess` na liczbę. O ile tylko wpiszemy samą liczbę, np. 4, JavaScript będzie wiedział, jak przekonwertować łańcuch `"4"` na liczbę 4. Zagadnieniami konwersji danych zajmiemy się bardziej szczegółowo dalej w tej książce.

P: A co się stanie, gdy użytkownik wpisze w oknie coś innego niż liczbę; np. `"jeden"` lub `"koniec"`?

O: W takim przypadku JavaScript nie będzie w stanie przekonwertować łańcucha na liczbę i wykonać porównania. Będziemy więc porównywać `"jeden"` z 6 oraz `"koniec"` z 6; oba porównania zwrócą wartość `false`, co będzie oznaczało PUDŁO. W lepszej wersji gry dane wprowadzane przez użytkownika będziemy sprawdzać bardziej uważnie, by upewnić się, że użytkownik najpierw wpisał liczbę.

P: Czy operator OR zwraca `true`, jeśli tylko jeden z warunków przyjmie wartość `true`, czy także wtedy, jeśli oba będą spełnione?

O: Także wtedy, jeśli oba warunki przyjmą wartość `true`. Wynikiem operatora OR (`||`) jest `true`, jeśli którykolwiek z warunków ma wartość `true` bądź jeśli oba warunki mają wartość `true`. Jeśli oba warunki mają wartość `false`, to także operator LUB zwróci wartość `false`.

P: A jakie jest działanie operatora koniunkcji logicznej?

O: Operator AND (`&&`) działa podobnie do operatora OR, ale przyjmuje wartość `true`, jeśli oba łączone warunki mają wartość `true`.

P: Czym jest pętla nieskończona?

O: Doskonałe pytanie. Pętla nieskończona to jeden z wielu problemów prześladujących programistów. Pamiętasz zapewne, że pętla wymaga warunku i działa tak długo, jak długo warunek ten będzie spełniony. Jeśli nasz kod nie robi niczego, by zmienić wartość warunku pętli, tak by warunek pętli kiedyś przyjął wartość `false`, pętla będzie działać w nieskończoność. Na wieki. Dopóki nie zamkniesz przeglądarki lub ponownie nie uruchomisz komputera.

Dwuminutowy przewodnik po operatorach boolowskich

Operatory boolowskie są używane w wyrażeniach logicznych, które mogą zwracać wartości `true` bądź `false`. Istnieją dwa rodzaje takich operatorów: operatory porównania oraz operatory logiczne.

Operatory porównania

Operatory porównania porównują dwie wartości. Poniżej przedstawiamy najczęściej używane operatory porównania:

- `<` oznacza „mniejszy",
- `>` oznacza „większy",
- `==` oznacza „równy",
- `===` oznacza „dokładnie równy",
- `<=` oznacza „mniejszy lub równy",
- `>=` oznacza „większy lub równy",
- `!=` oznacza „różny".

Operatory logiczne

Operatory logiczne łączą dwa wyrażenia logiczne i zwracają jedną wartość logiczną (`true` lub `false`). Istnieją dwa dwuargumentowe operatory logiczne.

- `||` oznacza alternatywę logiczną (ang. *OR*); zwraca wartość `true`, jeśli *którekolwiek* z łączonych wyrażeń ma wartość `true`.
- `&&` to koniunkcja logiczna (ang. *AND*); zwraca `true` wyłącznie wtedy, kiedy *oba* wyrażenia mają wartość `true`.

Istnieje jeszcze jeden operator logiczny — NOT — operator negacji logicznej; przy czym operuje on nie na dwóch, a na jednym wyrażeniu.

- `!` oznacza logiczną negację (ang. *NOT*); zwraca `true`, jeśli wartością wyrażenia jest `false`.

Czy użytkownikowi udało się trafić?

- [x] Napisać pętlę i pobrać komórkę do sprawdzenia.
- [ ] Sprawdzić trafienie.
- [ ] Sprawdzić, czy okręt został zatopiony.
- [ ] Wyświetlić użytkownikowi statystyki gry.

Właśnie tutaj zaczyna się robić ciekawie — użytkownik podał numer komórki, którą chce sprawdzić, a my musimy napisać kod, który określi, czy udało mu się trafić okręt. Konkretnie rzecz biorąc, musimy napisać kod, który sprawdzi, czy podana liczba odpowiada jednej z trzech komórek zajmowanych przez okręt. Jeśli tak będzie, powiększymy o jeden wartość zmiennej `hits`.

Poniżej przedstawiliśmy pierwsze podejście do napisania kodu wykrywającego trafienia; teraz przeanalizujemy go dokładnie.

```
if (guess == location1) {
    hits = hits + 1;
} else if (guess == location2) {
    hits = hits + 1;
} else if (guess == location3) {
    hits = hits + 1;
}
```

Jeśli podany numer komórki odpowiada położeniu zapisanemu w zmiennej location1, okręt został trafiony, a zatem inkrementujemy zmienną hits.

W przeciwnym razie, jeśli podany numer komórki odpowiada położeniu zapisanemu w zmiennej location2, postępujemy tak samo.

I w końcu, jeśli podany numer komórki odpowiada położeniu zapisanemu w zmiennej location3, inkrementujemy wartość zmiennej hits.

A jeśli żaden z tych warunków nie zostanie spełniony, wartość zmiennej hits się nie zmieni.

Zwróć uwagę na zastosowanie wcięć we wszystkich blokach if-else. Dzięki nim kod jest łatwiejszy do analizy, zwłaszcza kiedy występuje w nim wiele zagnieżdżonych bloków.

Zaostrz ołówek

Co myślisz na temat naszego pierwszego podejścia do napisania kodu wykrywającego trafienie okrętu? Czy kod nie wygląda, jakby był bardziej skomplikowany, niż jest to konieczne? Czy nie powtarzamy kodu w sposób wyglądający na, powiedzmy, nadmiarowy? Czy nie można go jakoś uprościć? Czy umiałbyś uprościć ten kod, bazując na tym, co już wiesz o operatorze || (czyli operatorze alternatywy logicznej)? Zanim przejdziesz dalej, koniecznie sprawdź naszą odpowiedź podaną na końcu tego rozdziału.

→ Rozwiązanie na stronie 78

Dodanie kodu wykrywającego trafienia

- [x] Napisać pętlę i pobrać komórkę do sprawdzenia.
- [x] Sprawdzić trafienie.
- [ ] Sprawdzić, czy okręt został zatopiony.
- [ ] Wyświetlić użytkownikowi statystyki gry.

Poskładajmy w całość wszystkie elementy przedstawione na kilku poprzednich stronach.

```
// Tu są umieszczone deklaracje zmiennych

while (isSunk == false) {
    guess = prompt("Gotów, cel, pal! (podaj liczbę z zakresu 0 - 6):");
    if (guess < 0 || guess > 6) {
        alert("Proszę podać prawidłowy numer komórki!");
    } else {
        guesses = guesses + 1;

        if (guess == location1 || guess == location2 || guess == location3) {
            hits = hits + 1;
        }
    }
}
```

PĘTLA: *dopóki okręt nie jest zatopiony*

POBIERZ komórkę do sprawdzenia

Sprawdzamy komórkę podaną przez użytkownika.

DODAJ jeden do wartości zmiennej guesses

Numer wydaje się prawidłowy, zatem powiększamy liczbę prób o jeden.

JEŚLI położenie podane przez użytkownika odpowiada jednej z komórek okrętu

DODAJ jeden do liczby trafień

Jeśli wartość zmiennej guess odpowiada jednej z trzech komórek zajmowanych przez okręt, inkrementujemy licznik trafień.

Wszystkie trzy warunki połączyliśmy w jedno wyrażenie, używając operatorów || (alternatywy logicznej). Należy więc czytać je w następujący sposób: jeśli guess jest równa location1 LUB guess jest równa location2, LUB guess jest równa location3, inkrementujemy wartość zmiennej hits.

Krok 3. Hej, zatopiłeś mój okręt!

- [x] Napisać pętlę i pobrać komórkę do sprawdzenia.
- [x] Sprawdzić trafienie.
- [x] Sprawdzić, czy okręt został zatopiony.
- [ ] Wyświetlić użytkownikowi statystyki gry.

To już prawie koniec — niemal w całości zaimplementowaliśmy logikę gry. Kiedy spojrzymy na pseudokod, zobaczymy, że pozostało jeszcze sprawdzenie, czy gracz nie uzyskał trzech trafień. Jeśli mu się to udało, zatopił nasz okręt. A jeśli okręt został zatopiony, musimy przypisać zmiennej `isSunk` wartość `true` i poinformować użytkownika o zniszczeniu okrętu. Napiszmy ten fragment kodu osobno, zanim dodamy go do naszego programu.

Najpierw sprawdzamy, czy są trzy trafienia.

Przyjrzyj się jeszcze raz pętli while. Co się stanie, kiedy zmienna isSunk przyjmie wartość true?

```
if (hits == 3) {
    isSunk = true;
    alert("Zatopiłeś mój okręt!");
}
```

Jeśli tak, przypisujemy zmiennej isSunk wartość true.

Dodatkowo informujemy użytkownika o zatopieniu okrętu!

Krok 4. Prezentacja informacji o zakończonej grze

- [x] Napisać pętlę i pobrać komórkę do sprawdzenia.
- [x] Sprawdzić trafienie.
- [x] Sprawdzić, czy okręt został zatopiony.
- [x] Wyświetlić użytkownikowi statystyki gry.

Po przypisaniu zmiennej `isSunk` wartości `true` pętla `while` zostanie zakończona. Program, który tak dobrze poznaliśmy, zakończy wykonywanie ciała pętli `while` i zanim się zorientujemy, nasza gra zacznie zmierzać do zakończenia. Jednak wciąż powinniśmy wyświetlić użytkownikowi jakieś informacje, aby wiedział, jak mu poszło. Poniżej przedstawiliśmy kod, który będzie to robić:

```
let stats = "Potrzebowałeś " + guesses + " prób, by zatopić okręt, " +
            "czyli Twoja efektywność wynosi: " + (3/guesses) + ".";
alert(stats);
```

W tej instrukcji tworzymy łańcuch znaków, który zawiera komunikat informujący użytkownika, ilu prób potrzebował do zatopienia okrętu oraz jaką osiągnął efektywność. Zwróć uwagę, że łańcuch jest podzielony na fragmenty (zarówno po to, by umieścić w nim wartość zmiennej guesses, jak i po to, by łatwiej go było zapisać w wielu wierszach), połączone operatorem +. Na razie po prostu wpisz kod w takiej postaci, w jakiej go przedstawiliśmy. Wyjaśnienia nieco później.

Dodajmy zatem do naszego programu komunikat oraz kod do wykrywania zatopienia.

```
// Tu są umieszczone deklaracje zmiennych

while (isSunk == false) {
    guess = prompt("Gotów, cel, pal! (podaj liczbę z zakresu 0 - 6):");
    if (guess < 0 || guess > 6) {
        alert("Proszę podać prawidłowy numer komórki!");
    } else {
        guesses = guesses + 1;

        if (guess == location1 || guess == location2 || guess == location3) {
            hits = hits + 1;

            if (hits == 3) {

                isSunk = true;

                alert("Zatopiłeś mój okręt!");
            }
        }
    }
}
let stats = "Potrzebowałeś " + guesses + " prób, by zatopić okręt, " +
            "czyli Twoja efektywność wynosi: " + (3/guesses) + ".";
alert(stats);
```

PĘTLA WHILE: *dopóki okręt nie jest zatopiony*

POBIERZ komórkę do sprawdzenia

DODAJ jeden do wartości zmiennej `guesses`

JEŚLI położenie podane przez użytkownika odpowiada jednej z komórek okrętu

DODAJ jeden do liczby trafień

JEŚLI liczba trafień wynosi 3

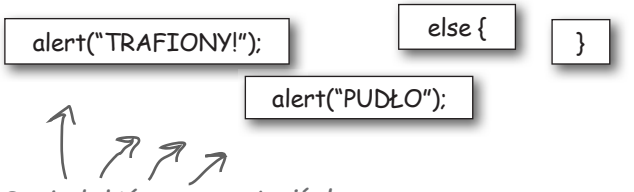
USTAW wartość zmiennej `isSunk` na `true`

WYŚWIETL komunikat „Zatopiłeś mój okręt!”

WYŚWIETL statystyki użytkownika

Ćwiczenie

Czy pamiętasz, jak stwierdziliśmy, że pseudokod nie jest idealny? Cóż, pominęliśmy pewne fragmenty naszego początkowego pseudokodu: nie informujemy użytkownika o tym, czy jego próba zakończyła się TRAFIENIEM, czy PUDŁEM. Czy potrafisz wstawić te fragmenty kodu w odpowiednich miejscach naszego programu?



Oto kod, który masz wstawić do programu.

```
// Tu są umieszczone deklaracje zmiennych

while (isSunk == false) {
    guess = prompt("Gotów, cel, pal! (podaj liczbę z zakresu 0 - 6):");
    if (guess < 0 || guess > 6) {
        alert("Proszę podać prawidłowy numer komórki!");
    } else {
        guesses = guesses + 1;
        if (guess == location1 || guess == location2 || guess == location3) {
            hits = hits + 1;
            if (hits == 3) {
                isSunk = true;
                alert("Zatopiłeś mój okręt!");
            }
        }
    }
}
let stats = "Potrzebowałeś " + guesses + " prób, by zatopić okręt, " +
            "czyli Twoja efektywność wynosi: " + (3/guesses) + ".";
alert(stats);
```

Tu jest naprawdę sporo nawiasów klamrowych, które musisz dopasować. Jeśli masz z tym jakiś problem, narysuj pionowe linie bezpośrednio tu — w książce.

Rozwiązanie na stronie 77

To koniec implementowania logiki!

- [x] Napisać pętlę i pobrać komórkę do sprawdzenia.
- [x] Sprawdzić trafienie.
- [x] Sprawdzić, czy okręt został zatopiony.
- [x] Wyświetlić użytkownikowi statystyki gry.

Świetnie! Udało się przekształcić cały pseudokod na prawdziwy kod w języku JavaScript. Udało się nawet znaleźć niezaimplementowane fragmenty pseudokodu i uzupełnić kod programu. Poniżej zamieściliśmy kompletny kod programu do gry w okręty. Upewnij się, że wpisałeś go w całości i zapisałeś w pliku o nazwie „battleship.js”:

```
let location1 = 3;
let location2 = 4;
let location3 = 5;
let guess;
let hits = 0;
let guesses = 0;
let isSunk = false;

while (isSunk == false) {
    guess = prompt("Gotów, cel, pal! (podaj liczbę z zakresu 0 - 6):");
    if (guess < 0 || guess > 6) {
        alert("Proszę podać prawidłowy numer komórki!");
    } else {
        guesses = guesses + 1;

        if (guess == location1 || guess == location2 || guess == location3) {
            alert("TRAFIONY!");
            hits = hits + 1;
            if (hits == 3) {
                isSunk = true;
                alert("Zatopiłeś mój okręt!");
            }
        } else {
            alert("PUDŁO");
        }
    }
}
let stats = "Potrzebowałeś " + guesses + " prób, by zatopić okręt, " +
            "czyli Twoja efektywność wynosi: " + (3/guesses) + ".";
alert(stats);
```

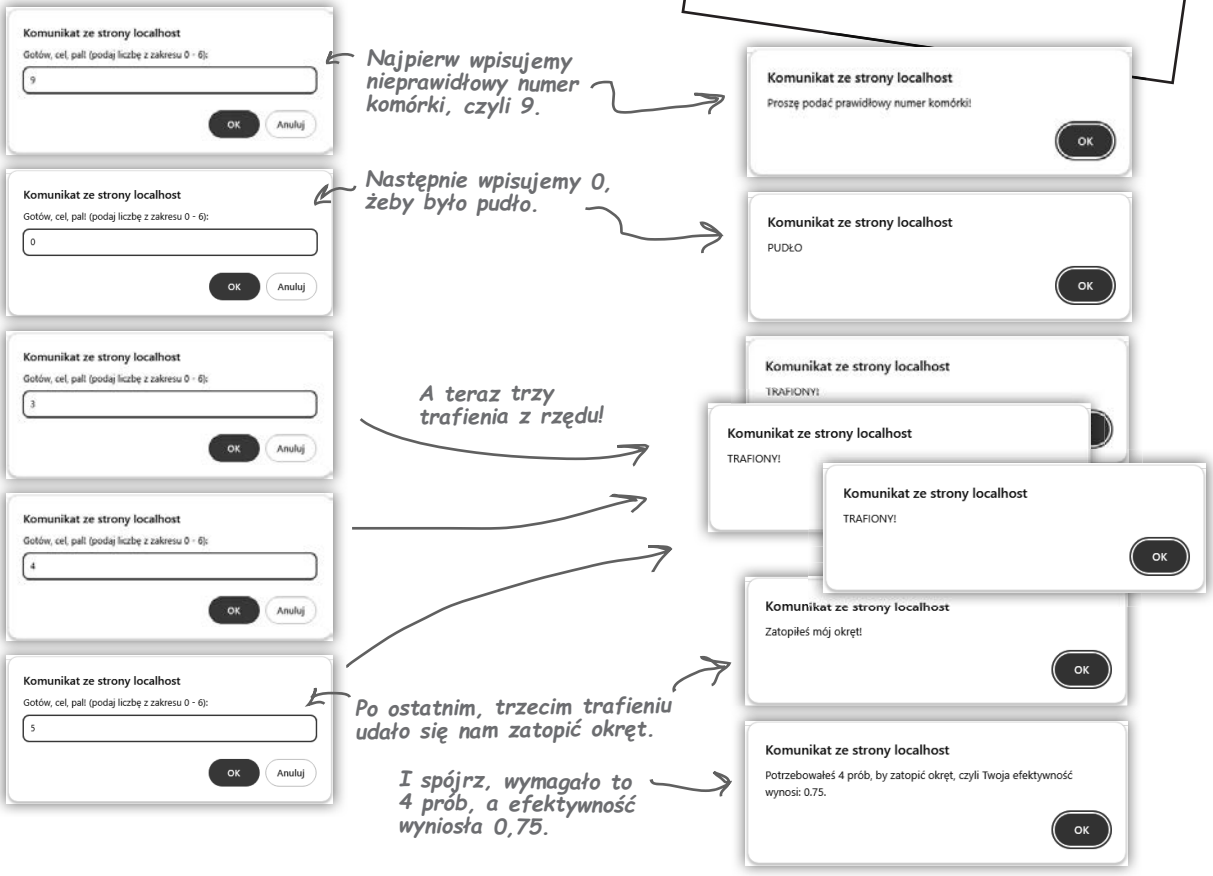
Chwilka na zapewnienie jakości

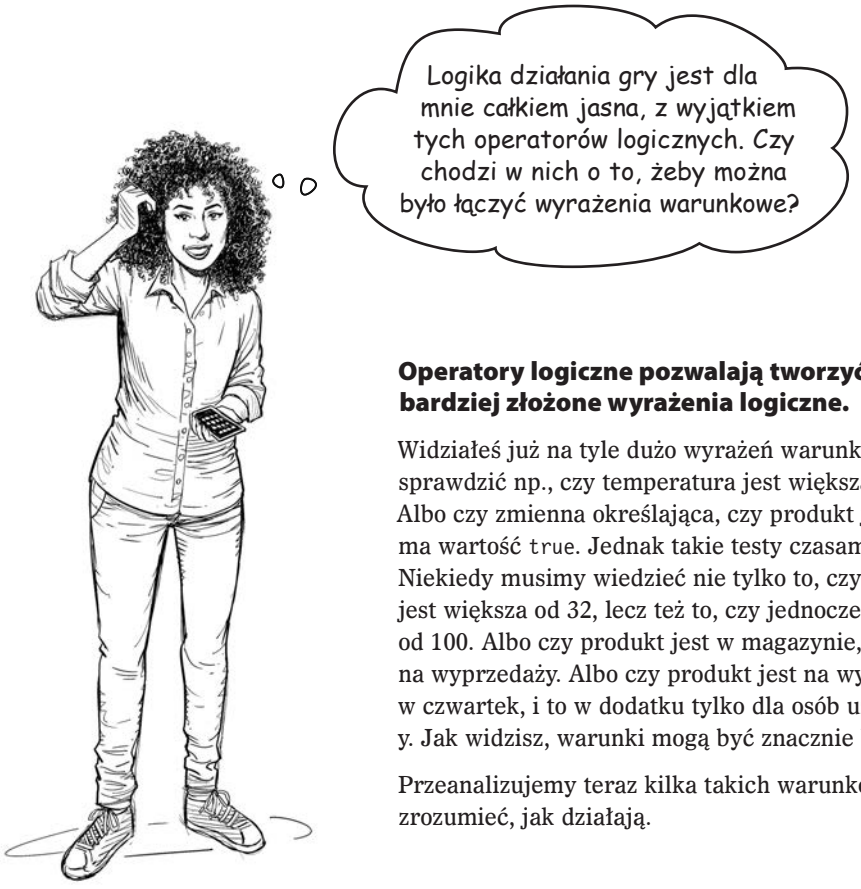
Sprawdzanie jakości (określane często jako „QA” od angielskich słów *Quality Assurance*) jest procesem testowania jakości oprogramowania w celu wykrycia defektów. Spróbujemy zatem przetestować trochę nasz kod. Kiedy będziesz gotowy, wyświetl w przeglądarce stronę „battleship.html” i zacznij grać. Próbuj robić różne rzeczy. Czy program działa bez zastrzeżeń? A może zauważyłeś jakieś problemy? Jeśli tak, zapisz je tutaj. Wyniki naszych testów możesz znaleźć poniżej.

Uwagi z testów

Zapisz wszystko, co nie działa tak, jak powinno, albo to, co można by poprawić.

Tak wygląda nasza interakcja z grą.





Operatory logiczne pozwalają tworzyć bardziej złożone wyrażenia logiczne.

Widziałeś już na tyle dużo wyrażeń warunkowych, by sprawdzić np., czy temperatura jest większa od 32 stopni. Albo czy zmienna określająca, czy produkt jest w magazynie, ma wartość `true`. Jednak takie testy czasami nie wystarczają. Niekiedy musimy wiedzieć nie tylko to, czy wartość zmiennej jest większa od 32, lecz też to, czy jednocześnie jest mniejsza od 100. Albo czy produkt jest w magazynie, a także czy jest na wyprzedaży. Albo czy produkt jest na wyprzedaży tylko w czwartek, i to w dodatku tylko dla osób uznawanych za VIP-y. Jak widzisz, warunki mogą być znacznie bardziej złożone.

Przeanalizujemy teraz kilka takich warunków, by lepiej zrozumieć, jak działają.

Załóżmy, że musimy sprawdzić, czy produkt jest dostępny w magazynie (`inStock`) I jest oferowany na wyprzedaży (`onSale`). Moglibyśmy to zrobić w następujący sposób:

```
if (inStock == true) {
    if (onSale == true) {
        // to wygląda na świetny interes!
        alert("kupuj, kupuj, kupuj!");
    }
}
```

Najpierw sprawdzamy, czy produkt jest dostępny w magazynie.

A jeśli jest, to czy jest oferowany na wyprzedaży.

A jeśli jest, czy można wykonać jakąś akcję, np. go kupić!

Zwróć uwagę, że ten kod zostanie wykonany wyłącznie w przypadku, gdy oba warunki zostaną spełnione!

Kod ten możemy uprościć, łącząc oba warunki w jeden. W odróżnieniu od prostej wersji gry w okręty, w której sprawdzaliśmy, czy numer komórki jest mniejszy od zera LUB większy od sześciu, teraz chcemy wiedzieć, czy zmienna `inStock` ma wartość `true` I czy zmienna `onSale` ma wartość `true`. Spójrz, jak można to zrobić…

Oto nasz operator I. Za jego pomocą to połączone wyrażenie przyjmie wartość true wyłącznie wtedy, kiedy zarówno pierwszy, jak I drugi warunek przyjmą wartość true.

```
if (inStock == true && onSale == true) {
    // to wygląda na świetny interes!
    alert("kupuj, kupuj, kupuj!");
}
```

Ten kod nie tylko jest bardziej zwięzły, lecz także bardziej czytelny. Porównaj go z kodem przedstawionym na poprzedniej stronie.

Jednak wcale nie musimy na tym poprzestawać — możemy łączyć wiele operatów logicznych, by na wiele sposobów tworzyć złożone wyrażenia warunkowe:

Teraz w jednym wyrażeniu warunkowym użyliśmy zarówno operatora I, jak i LUB. W tym przypadku ma ono następujące znaczenie: jeśli produkt jest w magazynie I jest na wyprzedaży LUB jego cena jest mniejsza od 60, możemy go kupić.

```
if (inStock == true && (onSale == true || price < 60)) {
    // to wygląda na świetny interes!
    alert("kupuj, kupuj, kupuj!");
}
```

Zwróć uwagę na zastosowanie nawiasów w celu połączenia warunków, dzięki czemu najpierw zostanie określony wynik operatora LUB, a dopiero potem będzie on użyty do określenia wyniku wyrażenia z operatorem I.

Zaostrz ołówek

Mamy tutaj całą grupę wyrażeń logicznych, które czekają na określenie wartości. Wypełnij puste miejsca, a następnie, nim przejdziesz do dalszej lektury, sprawdź wyniki na końcu tego rozdziału.

```
let temp = 81;
let willRain = true;
let humid = (temp > 80 && willRain == true);
```

Jaka jest wartość zmiennej **humid**? _____

```
let guess = 6;
let isValid = (guess >= 0 && guess <= 6);
```

Jaka jest wartość zmiennej **isValid**? _____

```
let kB = 1287;
let tooBig = (kB > 1000);
let urgent = true;
let sendFile =
    (urgent == true || tooBig == false);
```

Jaka jest wartość zmiennej **sendFile**? _____

```
let keyPressed = "N";
let points = 142;
let level;
if (keyPressed == "Y" ||
      (points > 100 && points < 200)) {
    level = 2;
} else {
    level = 1;
}
```

Jaka jest wartość zmiennej **level**? _____

→ Rozwiązanie na stronie 75

Ćwiczenie

Jakub i Wilhelm, obaj z księgowości, pracują nad nową aplikacją do sprawdzania cen w firmowej witrynie. Obaj napisali instrukcje warunkowe, korzystając w nich z wyrażeń warunkowych. Obaj są pewni, że ich kod jest prawidłowy. Który z księgowych ma rację? Czy księgowi w ogóle powinni pisać kod? Zanim przejdziesz dalej, porównaj swoją odpowiedź z prawidłową, którą zamieściliśmy pod koniec rozdziału.

Rozwiązanie Jakuba

```
if (price < 200 || price > 600) {
    alert("Cena jest zbyt niska lub zbyt wysoka! Nie kupować tego gadżetu.");
} else {
    alert("Cena jest w porządku! Można kupować.");
}
```

Kto ma rację: Jakub czy Wilhelm?

Rozwiązanie Wilhelma

```
if (price >= 200 || price <= 600) {
    alert("Cena jest w porządku! Można kupować.");
} else {
    alert("Cena jest zbyt niska lub zbyt wysoka! Nie kupować tego gadżetu.");
}
```

Rozwiązanie na stronie 76

Czy możemy pogadać o rozwlekłości Twojego kodu?

Nie wiemy, jak zwrócić Ci uwagę, jednak Twój sposób zapisu wyrażeń warunkowych jest trochę rozwlekły. O co nam chodzi? Pokażemy to na poniższym przykładzie.

W wyrażeniach warunkowych często porównujemy zmienne logiczne z wartościami true bądź false.

```
if (inStock == true) {
    ...
}
```

A inStock jest zmienną logiczną przechowującą wartość true lub false.

Jak się okazuje, taki zapis jest pewną przesadą. Cały sens wyrażenia warunkowego polega na tym, że ma zwrócić wartość `true` lub `false`, ale nasza zmienna `inStock` już *zawiera* jedną z tych wartości. Nie musimy więc już z niczym jej porównywać — wystarczy zapisać samą zmienną. Oznacza to, że powyższą instrukcję można zapisać w następujący sposób:

```
if (inStock) {
    ...
}
```

Kiedy użyjemy samej zmiennej logicznej i przyjmie ona wartość true, całe wyrażenie warunkowe także przyjmie wartość true, więc blok instrukcji if zostanie wykonany.

A jeśli zmienna inStock przyjmie wartość false, wyrażenie warunkowe także przyjmie tę wartość i blok instrukcji if zostanie pominięty.

Choć niektórzy mogliby twierdzić, że nasza początkowa, dłuższa wersja kodu jest jednocześnie bardziej zrozumiała, jednak w praktyce częściej można się spotkać z tą bardziej zwięzłą formą zapisu. Sam się przekonasz, że ten krótszy zapis jest łatwiejszy do odczytu.

Ćwiczenie

Poniżej zamieściliśmy dwie instrukcje z wyrażeniami korzystającymi ze zmiennych logicznych `onSale` oraz `inStock`. Obie te instrukcje określają wartość zmiennej `buyIt`. Dla obu instrukcji rozpatrz wszystkie możliwe kombinacje wartości zmiennych `inStock` oraz `onSale`. Która z wersji instrukcji narazi Cię na większe wydatki?

```
let buyIt = (inStock || onSale);
```

onSale	inStock	buyIt	buyIt
true	*true*		
true	*false*		
false	*true*		
false	*false*		

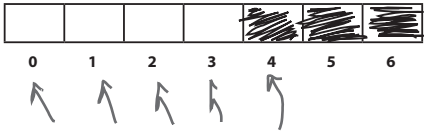
```
let buyIt = (inStock && onSale);
```

→ Rozwiązanie na stronie 76

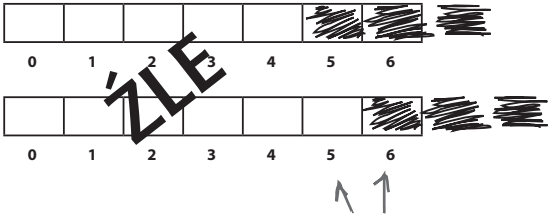
Kończymy prostą wersję gry w okręty

Owszem, wciąż mamy do załatwienia jeszcze jedną sprawę: aktualnie położenie okrętu jest podane na stałe — niezależnie od tego, jak wiele razy zagramy w naszą grę, okręt zawsze będzie zajmował komórki numer 3, 4 i 5. Takie rozwiązanie zdaje egzamin w czasie testowania gry, żeby jednak gra była nieco bardziej interesująca dla użytkownika, okręt musi być rozmieszczany losowo.

Cofnijmy się zatem nieco i zastanówmy, jaki jest prawidłowy sposób rozmieszczania okrętu w jednowymiarowej siatce składającej się z siedmiu komórek. Potrzebujemy położenia startowego, które umożliwi umieszczenie okrętu w trzech kolejnych komórkach siatki. A to oznacza, że nasze początkowe położenie musi mieć wartości z zakresu od 0 do 4.



Możemy zacząć rozmieszczać okręt w komórkach o numerach: 0, 1, 2, 3 oraz 4 i wciąż w siatce pozostanie na tyle dużo miejsca, by umieścić okręt w trzech kolejnych komórkach.



Natomiast rozpoczęcie rozmieszczania od komórki 5. lub 6. nie pozwoli umieścić w siatce całego okrętu.

Jak przypisywać wartości losowe?

Teraz, kiedy już wiemy, gdzie zacząć rozmieszczanie okrętu (w komórce o numerze od zera do cztery), wystarczy użyć tej wartości do określenia dwóch kolejnych zajmowanych przez okręt komórek siatki.

```
let location1 = randomLoc;
let location2 = location1 + 1;
let location3 = location2 + 1;
```

Używamy losowo wybranego położenia oraz dwóch kolejnych komórek siatki.

No dobrze, ale w jaki sposób możemy wygenerować liczbę losową? Z tym zadaniem zwrócimy się do języka JavaScript oraz jego wbudowanych funkcji matematycznych, a zwłaszcza dwóch, które pozwalają na generowanie liczb losowych. Przyjrzyjmy się zatem nieco dokładniej wbudowanym funkcjom języka, natomiast bardziej ogólne informacje o funkcjach podamy dalej w tej książce. Na razie chodzi tylko o skorzystanie z funkcji, by zrobić to, co musimy.

Przepis na generowanie liczb losowych

Zaczniemy od funkcji `Math.random`. Wywołanie tej funkcji zwraca w wyniku losową liczbę dziesiętną:

To jest nasza zmienna — randomLoc. Chcemy zapisać w niej liczbę losową z zakresu od 0 do 4.

Math.random jest standardowym elementem języka JavaScript — funkcją zwracającą liczbę losową.

```
let randomLoc = Math.random();
```

Jedyny problem polega na tym, że zwraca ona liczby takie jak 0,128 lub 0,830, lub 0,9, lub 0,42, czyli liczby z zakresu od 0 do 1 (przy czym liczba 1 nigdy nie jest zwracana). Musimy zatem wymyślić jakiś sposób, by uzyskać liczbę z zakresu od 0 do 4.

Potrzebujemy liczby losowej z zakresu od 0 do 4 — czyli 0, 1, 2, 3 i 4 — a nie liczby dziesiętnej, takiej jak 0,34. Jak możemy to zrobić? Możemy zacząć od pomnożenia przez 5 liczby zwróconej przez element `Math.random`, aby uzyskać wynik nieco bardziej zbliżony do tego, czego potrzebujemy. Chodzi o coś takiego:

Jeśli pomnożymy liczbę losową przez 5, uzyskamy liczbę losową z zakresu od 0 do 5, lecz mniejszą niż 5, czyli uzyskamy takie liczby jak 0,13983, 4,231, 2,3451 lub 4,999.

```
let randomLoc = Math.random() * 5;
```

*Pamiętaj, że * oznacza mnożenie.*

Jesteśmy już znacznie bliżej! Teraz pozostaje jedynie pozbyć się części ułamkowej, by uzyskać liczbę całkowitą. Możemy skorzystać z kolejnej wbudowanej funkcji JavaScriptu, a konkretnie z funkcji `Math.floor`.

Możemy skorzystać z wbudowanej funkcji Math.floor, by przekształcić liczbę na najbliższą liczbę całkowitą.

```
let randomLoc = Math.floor(Math.random() * 5);
```

A zatem liczba taka jak 0,13983 zostaje przekształcona na 0, liczba 2,3451 na 2, a 4,999 na 4.

Nie ma głupich pytań

P: Dlaczego wtedy, gdy zależy nam na liczbie z zakresu od 0 do 4, pomnożyliśmy wartość losową przez 5? Czemu użyliśmy wyrażenia `Math.floor(Math.random() * 5)`?

O: Dobre pytanie. Otóż funkcja `Math.random` generuje liczbę z zakresu od 0 do 1, lecz mniejszą od 1. Maksymalną wartością, jaką może zwrócić `Math.random`, jest 0,999. Jeśli pomnożymy ją przez 5, możemy uzyskać co najwyżej wartość 4,999... Z kolei funkcja `Math.floor` zaokrągla liczbę w dół, a zatem 1,2 zostanie zaokrąglona do 1, podobnie jak 1,9999. Jeśli wygenerujemy liczbę od 0 do 4,999, to uzyskamy w efekcie liczbę z zakresu od 0 do 4. Nie jest to jedyny sposób, w jaki można to zrobić, a w innych językach programowania często wykonuje się to inaczej; jednak takie rozwiązanie jest najczęściej spotykane w kodzie JavaScript.

P: Gdybym zatem chciał uzyskać liczbę losową z zakresu od 0 do 100 (włącznie), musiałbym użyć wyrażenia `Math.floor(Math.random() * 101)`?

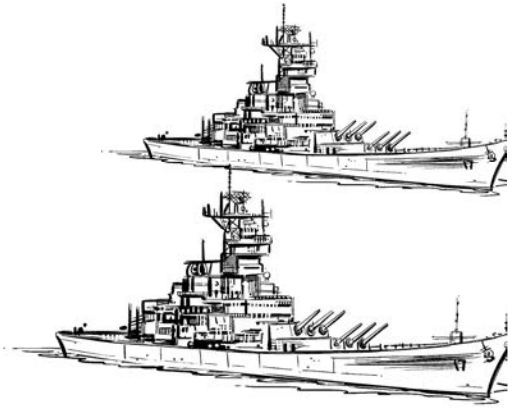
O: Właśnie tak! Mnożąc wynik przez 101 i zaokrąglając go w dół, masz gwarancję, że uzyskana liczba losowa będzie mniejsza lub równa 100.

P: Po co te nawiasy za `Math.random()`?

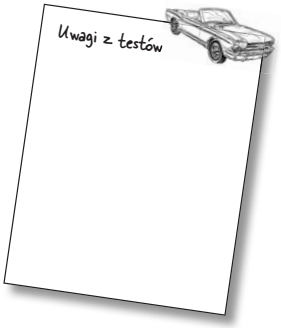
O: Nawiasów używamy zawsze wtedy, gdy „wywołujemy" funkcję. Czasami do funkcji trzeba przekazać wartość, tak jak np. podczas wyświetlania komunikatu, a czasami przekazywanie wartości nie jest konieczne. Jednak zawsze wtedy, gdy wywołujemy funkcję (niezależnie od tego, czy jest to funkcja wbudowana, czy nie), konieczne jest użycie pary nawiasów. Na razie się tym nie przejmuj; wszystkie szczegóły związane z funkcjami wyjaśnimy w następnym rozdziale.

P: Nie jestem w stanie uruchomić swojej wersji gry w okręty. Na stronie nie widzę nic z wyjątkiem nagłówka „Zagraj w okręty!". Jak mogę sprawdzić, co robię źle?

O: Właśnie w takich sytuacjach może się przydać konsola JavaScript. Jeśli popełnisz jakiś błąd, taki jak pominięcie cudzysłowu w łańcuchu znaków, JavaScript zazwyczaj poskarży się na błąd składniowy w kodzie, a może nawet wyświetli numer wiersza, w którym błąd wystąpił. Jednak niektóre błędy są znacznie bardziej subtelne. Jeśli np. przez pomyłkę napiszesz `isSunk = false` zamiast `isSunk == false`, to żaden błąd JavaScriptu się nie pojawi, a mimo to program nie będzie działał zgodnie z oczekiwaniami. W przypadku błędów tego typu można spróbować rejestrowania wartości zmiennych w różnych miejscach kodu przy użyciu funkcji `console.log`, by w ten sposób zlokalizować problem.



Zadbajmy o jakość



Mamy już wszystko, czego potrzebujemy. Poskładajmy zatem wszystkie fragmenty kodu (zrobiliśmy to już poniżej) i zastąpmy wcześniejszy kod określający położenie okrętu nową wersją. Kiedy Ty też to zrobisz, wypróbuj grę kilka razy, by przekonać się, jak szybko jesteś w stanie zatopić okręt wroga.

```
let randomLoc = Math.floor(Math.random() * 5);
let location1 = randomLoc;
let location2 = location1 + 1;
let location3 = location2 + 1;
let guess;
let hits = 0;
let guesses = 0;
let isSunk = false;

while (isSunk == false) {
    guess = prompt("Gotów, cel, pal! (podaj liczbę z zakresu 0 - 6):");
    if (guess < 0 || guess > 6) {
        // tu należy umieścić resztę kodu...
```

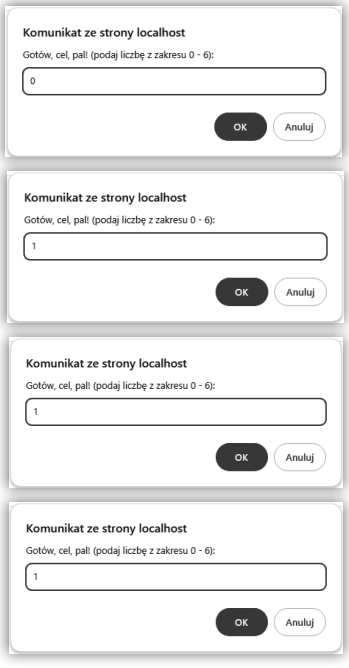
Dalej, zastąp stare deklaracje zmiennych location tymi nowymi instrukcjami.

Poniżej przedstawiliśmy przebieg jednej z naszych sesji testowych. Teraz, kiedy okręt jest umieszczany w losowym miejscu, gra stała się nieco bardziej interesująca. Uważamy, że udało się nam uzyskać całkiem dobry wynik...

Ćwiczenie

Chwileczkę, zauważyliśmy coś, co wygląda na błąd. Wskazówka: kiedy wpiszemy 0, 1, 1, 1, wszystko się psuje! Czy potrafisz powiedzieć, co się dzieje?

A oto nasze próby...

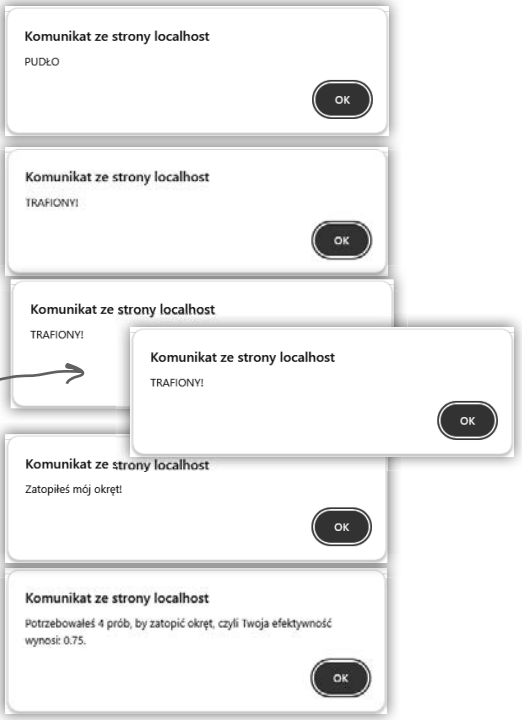


Za pierwszym razem spudłowaliśmy.

Za drugim razem zaliczyliśmy trafienie.

A później zaczęliśmy wpisywać tę samą cyfrę i za każdym razem uzyskiwaliśmy trafienie!

Po trzecim trafieniu udało się nam zatopić okręt! **Ale tu coś nie gra!** *Nie powinniśmy zatopić okrętu, trafiając trzy razy w to samo miejsce.*



Wpisaliśmy 0, 1, 1, 1, a okręt znajduje się w komórkach 1., 2. i 3.

Cóż za nerwy!
Wszystko wisi na włosku!

Czy uda się nam ***znaleźć*** błąd?

Czy będziemy umieli go ***poprawić***?

Oczekuj w napięciu na znacznie lepszą, poprawioną wersję gry w okręty zamieszczoną dalej w tej książce.

W międzyczasie zastanów się, czy przychodzi Ci do głowy jakiś pomysł na to, jak poprawić ten błąd.

Uwagi z testów

Znaleźliśmy błąd! Wpisywanie tej samej cyfry będącej trafieniem okrętu pozwala go zatopić, choć nie powinno.

Gratulujemy pierwszego prawdziwego programu w języku JavaScript... i kilka słów o wielokrotnym używaniu kodu

Prawdopodobnie zwróciłeś uwagę, że skorzystaliśmy już z kilku *wbudowanych funkcji* języka JavaScript, takich jak `alert`, `prompt`, `console.log` oraz `Math.random`. Funkcje te przy bardzo małym nakładzie pracy pozwalają na wyświetlanie okien dialogowych, zapisywanie wyników w oknie konsoli i generowanie liczb losowych, a wszystko w niemal magiczny sposób. Jednak te wbudowane funkcje są jedynie przygotowanym kodem, który ktoś dla nas napisał, a ich siła polega na tym, że można ich wielokrotnie używać, wykonując odpowiednie wywołanie tam, gdzie chcemy i kiedy jest to potrzebne.

Możemy podać wiele informacji na temat wbudowanych funkcji JavaScriptu — jak należy z nich korzystać, jakie wartości należy przekazywać w ich wywołaniach itd. — zaczniesz je powoli poznawać w następnym rozdziale, poświęconym tworzeniu własnych funkcji.

Zanim jednak tam dotrzesz, masz jeszcze do przejrzenia „Celne spostrzeżenia” i rozwiązania wszystkich ćwiczeń. A tak... i spokojny sen, żeby wszystkie te informacje ułożyły się w Twojej głowie.

CELNE SPOSTRZEŻENIA

- Logikę działania programu pisanego w języku JavaScript można przedstawić w formie schematu blokowego, pokazującego punkty decyzyjne oraz akcje.
- Zanim zaczniesz pisać program, warto naszkicować sposób jego działania, zapisując go w formie pseudokodu.
- **Pseudokod** to przybliżenie wszystkich czynności, które musi realizować prawdziwy kod.
- Istnieją dwa rodzaje operatorów boolowskich: operatory porównania oraz operatory logiczne. Operatory te zastosowane w wyrażeniach zawsze zwracają wartości logiczne, czyli `true` lub `false`.
- Operatory **porównania** porównują dwie wartości i zwracają wartości logiczne `true` lub `false`. Przykładowo operatora porównania < („mniejszy”) możemy używać w następujący sposób: `3 < 6`. To wyrażenie zwróci wartość `true`.
- Operatory **logiczne** łączą dwie wartości logiczne. Przykładowo `true || false` zwraca `true`, a `true && false` zwraca `false`.
- Liczbę losową z zakresu od 0 do 1 (włącznie z 0, lecz z wyłączeniem 1) można wygenerować przy użyciu funkcji **`Math.random`**.
- Funkcja **`Math.floor`** zaokrągla przekazaną liczbę dziesiętną w dół do najbliższej liczby całkowitej.
- Pamiętaj, by używając funkcji **`Math.floor`** oraz **`Math.random`**, zawsze zapisywać **`Math`** wielką literą, a nie małą.
- Funkcja **`prompt`** wyświetla okno dialogowe z komunikatem oraz miejscem, w którym użytkownik może wpisać jakąś wartość.
- W tym rozdziale używaliśmy funkcji **`prompt`**, by pobierać dane od użytkownika, oraz funkcji `alert`, by wyświetlać w przeglądarce wyniki gry w okręty.

Zaostrz ołówek

Rozwiązanie ze strony 50

Załóż, że nasza wirtualna siatka wygląda w następujący sposób:

A komórki zajmowane przez okręt są zapisane w trzech zmiennych lokalnych.

```
location1 = 3;
location2 = 4;
location3 = 5;
```

Użyj następującej sekwencji liczb jako lokalizacji sprawdzanych komórek:

```
1, 4, 2, 3, 5
```

A teraz, opierając się na pseudokodzie przedstawionym na stronie 51, przejdź każdy krok kodu i zobacz, jak będzie wykonywany w przypadku podania powyższej sekwencji danych wejściowych. Swoje uwagi zapisz niżej. Oto nasze rozwiązanie:

location1	location2	location 3	guess	guesses	hits	isSunk
3	4	5	—	0	0	false
3	4	5	1	1	0	false
3	4	5	4	2	1	false
3	4	5	2	3	1	false
3	4	5	3	4	2	false
3	4	5	5	5	3	true

Ćwiczenie

Rozwiązanie ze strony 67

Poniżej zamieściliśmy dwie instrukcje z wyrażeniami korzystającymi ze zmiennych logicznych `onSale` oraz `inStock`. Obie te instrukcje określają wartość zmiennej `buyIt`. Dla obu instrukcji rozpatrz wszystkie możliwe kombinacje wartości zmiennych `inStock` oraz `onSale`. Która z wersji instrukcji narazi Cię na większe wydatki? Ta z operatorem LUB (`||`)!

```
let buyIt = (inStock || onSale);
```

onSale	inStock	buyIt	buyIt
true	*true*	*true*	*true*
true	*false*	*true*	*false*
false	*true*	*true*	*false*
false	*false*	*false*	*false*

```
let buyIt = (inStock && onSale);
```

Zaostrz ołówek

Rozwiązanie ze strony 65

Mamy tutaj całą grupę wyrażeń logicznych, które czekają na określenie wartości. Wypełnij puste miejsca. Oto nasze rozwiązanie:

```
let temp = 81;
let willRain = true;
let humid = (temp > 80 && willRain == true);
```

Jaka jest wartość zmiennej **humid**? *true*

```
let guess = 6;
let isValid = (guess >= 0 && guess <= 6);
```

Jaka jest wartość zmiennej **isValid**? *true*

```
let kB = 1287;
let tooBig = (kB > 1000);
let urgent = true;
let sendFile =
    (urgent == true || tooBig == false);
```

Jaka jest wartość zmiennej **sendFile**? *true*

```
let keyPressed = "N";
let points = 142;
let level;
if (keyPressed == "Y" ||
      (points > 100 && points < 200)) {
    level = 2;
} else {
    level = 1;
}
```

Jaka jest wartość zmiennej **level**? *2*

Ćwiczenie

Rozwiązanie ze strony 66

Jakub i Wilhelm, obaj z księgowości, pracują nad nową aplikacją do sprawdzania cen w firmowej witrynie. Obaj napisali instrukcje warunkowe, korzystając w nich z wyrażeń warunkowych. Obaj są pewni, że ich kod jest prawidłowy. Który z księgowych ma rację? Czy księgowi w ogóle powinni pisać kod? Oto nasze rozwiązanie:

Rozwiązanie Jakuba

```
if (price < 200 || price > 600) {
    alert("Cena jest zbyt niska lub zbyt wysoka! Nie kupować tego gadżetu.");
} else {
    alert("Cena jest w porządku! Można kupować.");
}
```

Rozwiązanie Wilhelma

```
if (price >= 200 || price <= 600) {
    alert("Cena jest w porządku! Można kupować.");
} else {
    alert("Cena jest zbyt niska lub zbyt wysoka! Nie kupować tego gadżetu.");
}
```

Jakub jest lepszym programistą (i zapewne także lepszym księgowym). Rozwiązanie Jakuba działa, a Wilhelma nie. Aby przekonać się, dlaczego tak jest, wypróbujmy trzy różne wartości zmiennej price (zbyt dużą, zbyt małą i odpowiednią) i dla każdej z nich przekonajmy się, jakie wartości przyjmują wyrażenia zastosowane przez obu księgowych.

price	Jakub	Wilhelm
100	true komunikat: Nie kupuj!	true komunikat: Kupuj!
700	true komunikat: Nie kupuj!	true komunikat: Kupuj!
400	false komunikat: Kupuj!	true komunikat: Kupuj!

Jeśli wartość zmiennej price wyniesie 100, to 100 jest mniejsze od 200, zatem wyrażenie Jakuba jest prawdziwe (pamiętajmy, że w razie użycia operatora LUB wystarczy, by jedno z łączonych wyrażeń było prawdziwe, żeby wartość całego wyrażenia także była prawdą), a zatem wyświetlamy komunikat, by NIE kupować.

Jednak wyrażenie Wilhelma także jest prawdziwe, a to ze względu na użycie warunku price <= 600. A zatem wartością całego wyrażenia Wilhelma także jest true, co spowoduje wyświetlenie użytkownikowi komunikatu zachęcającego do kupna, choć cena jest zbyt niska.

Okazuje się, że wyrażenie Wilhelma zawsze przyjmuje wartość true, zupełnie niezależnie od wartości zmiennej price, a zatem jego kod zawsze będzie zachęcał do kupowania. Wilhelm powinien ograniczyć się do przeglądania ksiąg rachunkowych.

Ćwiczenie

Rozwiązanie ze strony 61

Czy pamiętasz, jak stwierdziliśmy, że pseudokod nie jest idealny? Cóż, pominęliśmy pewne fragmenty naszego początkowego pseudokodu: nie informujemy użytkownika o tym, czy jego próba zakończyła się TRAFIENIEM, czy PUDŁEM. Czy potrafisz wstawić te fragmenty kodu w odpowiednich miejscach naszego programu? Oto nasze rozwiązanie:

```
// Tu są umieszczone deklaracje zmiennych

while (isSunk == false) {
    guess = prompt("Gotów, cel, pal! (podaj liczbę z zakresu 0 - 6):");
    if (guess < 0 || guess > 6) {
        alert("Proszę podać prawidłowy numer komórki!");
    } else {
        guesses = guesses + 1;
        if (guess == location1 || guess == location2 || guess == location3) {

            alert("TRAFIONY!");
            if (hits == 3) {
                isSunk = true;
                alert("Zatopiłeś mój okręt!");
            }
        }
            else {
                alert("PUDŁO");
            }
        }
    }
let stats = "Potrzebowałeś " + guesses + " prób, by zatopić okręt, " +
            "czyli Twoja efektywność wynosi: " + (3/guesses) + ".";
alert(stats);
```

Zaostrz ołówek

Rozwiązanie ze strony 58

Co myślisz na temat naszego pierwszego podejścia do napisania kodu wykrywającego trafienie okrętu? Czy kod nie wygląda, jakby był bardziej skomplikowany, niż jest to konieczne? Czy nie powtarzamy kodu w sposób wyglądający na, powiedzmy, nadmiarowy? Czy nie można go jakoś uprościć? Czy umiałbyś uprościć ten kod, bazując na tym, co już wiesz o operatorze || (czyli operatorze alternatywy logicznej)? A oto rozwiązanie:

Wielokrotnie używamy tego samego kodu.

```
if (guess == location1) {
    hits = hits + 1;
} else if (guess == location2) {
    hits = hits + 1;
} else if (guess == location3) {
    hits = hits + 1;
}
```

Gdybyśmy kiedyś musieli zmienić sposób aktualizowania trafień, musielibyśmy wprowadzać modyfikacje w trzech miejscach. Zmiany tego typu często są przyczyną błędów i problemów w kodzie.

Zresztą nie tylko o to chodzi. Ten kod jest po prostu znacznie bardziej złożony, niż jest to konieczne. Dodatkowo jest także trudniejszy do analizy, dłuższy i wymaga więcej wpisywania.

Jednak dzięki zastosowaniu logicznego operatora LUB możemy połączyć wszystkie testy w taki sposób, że jeśli wskazana przez użytkownika komórka odpowiada wartości którejkolwiek ze zmiennych, location1, location2 lub location3, to całe wyrażenie przyjmie wartość true i liczba trafień zostanie zaktualizowana.

```
if (guess == location1 || guess == location2 || guess == location3) {
    hits = hits + 1;
}
```

Czy to wyrażenie nie wygląda znacznie prościej? Nie wspominamy w ogóle o tym, że jest łatwiejsze do zrozumienia.

A jeśli kiedykolwiek będziemy musieli zmienić sposób aktualizowania liczby trafień, wystarczy to zrobić w jednym miejscu, co w znacznie mniejszym stopniu naraża nas na błędy.

Skorowidz

G

Ł

M

N

O

P

R

S

Ś

T

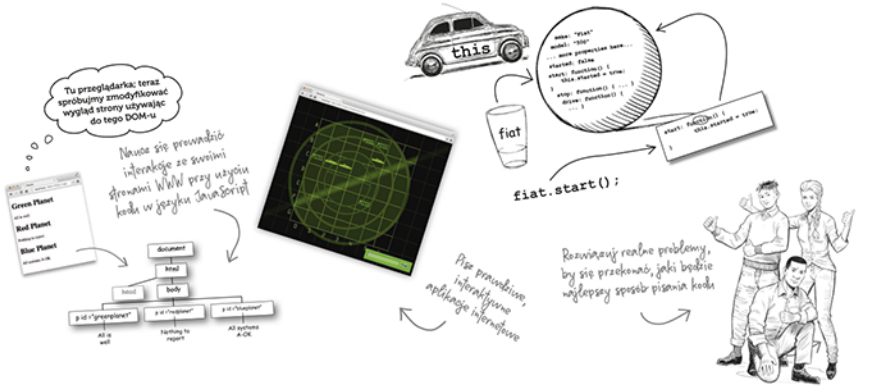
U

W

Z

UWAGA! Podręcznik przyjazny dla mózgu!

JavaScript to supermoc programowania internetu! Zapomnij o suchych, nudnych i statycznych stronach — JavaScript umożliwia interakcję z użytkownikami, pobieranie danych z sieci, rysowanie grafiki i robienie wielu innych świetnych i funkcjonalnych rzeczy! A to dopiero początek: JavaScript jest jednym z najpopularniejszych języków programowania, a jego zastosowanie wykracza daleko poza internet!



To drugie wydanie świetnego podręcznika opracowanego zgodnie z najnowszymi odkryciami nauk poznawczych, teorii uczenia się i neurofizjologii. Dzięki temu zaangażujesz swój mózg, użyjesz wielu zmysłów i niepostrzeżenie poznasz nowoczesny język JavaScript — począwszy od jego podstaw, a skończywszy na najnowszych możliwościach. Poznasz niuanse typów stosowanych w JavaScripcie i w końcu zrozumiesz domknięcia. Będziesz grać w gry, rozwiązywać zagadki, odkrywać tajemnice, a przede wszystkim pisać prawdziwy kod, by w końcu zacząć tworzyć własne aplikacje!

Jest to jedna z tych rzadkich książek, które mogę polecić bez żadnych zastrzeżeń!

— prof. David Gelernter
Uniwersytet Yale

Pożegnaj się z nudnymi samouczkami. Powitaj rewolucyjny sposób nauki JavaScriptu!

— Doreen Lorenzo
Uniwersytet Teksański w Austin

Oto niezastąpione narzędzie dla zmotywowanych, niezależnych uczniów!

— Josh Sharfman
Shalhevet High School

Dr Eric Freeman
jest informatykiem, badaczem, autorem książek technicznych. Wcześniej pełnił funkcję dyrektora technicznego w Walt Disney Company, dziś jest wykładowcą Uniwersytetu Teksańskiego.

Elisabeth Robson
jest inżynierem programowania, instruktorką i autorką książek. Jest współzałożycielką firmy WickedlySmart. Zajmuje się tworzeniem produktów edukacyjnych dla profesjonalnych programistów.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-2265-5
9 788328 922655

Cena: 139,00 zł