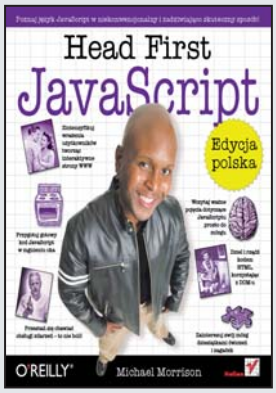
Head First JavaScript. Edycja polska



Autor: Michael Morrison
Tłumaczenie: Piotr Rajca
ISBN: 978-83-246-1548-3
Tytuł oryginału: Head First JavaScript (Head First)
Format: 200x230, stron: 624

Poznaj język JavaScript w niekonwencjonalny i zadziwiająco skuteczny sposób!

Dziś statyczne witryny WWW giną w ogromnej masie podobnych sobie stron, przy braku zainteresowania współczesnych użytkowników sieci. Aby się wyróżnić, trzeba zaproponować oglądającym coś innego niż tylko ładnie sformatowany tekst i schludną grafikę. Jednym z pomysłów na zwiększenie atrakcyjności witryny WWW jest wprowadzenie na nią elementów interaktywnych. Istnieje wiele rozwiązań służących do tworzenia takich elementów. Jednym z najczęściej wykorzystywanych jest JavaScript. Ten interpretowany po stronie przeglądarki język pozwala między innymi na kontrolowanie niemal wszystkich elementów HTML w oparciu o obiektowy model dokumentu (DOM), obsługę zdarzeń generowanych przez użytkownika i weryfikację poprawności danych wprowadzanych do formularza.

Dzięki książce „Head First JavaScript. Edycja polska" poznasz JavaScript w nietypowy, a przy tym bardzo skuteczny sposób. Ponieważ została ona napisana w oparciu o najnowsze teorie uczenia się, błyskawicznie przyswoisz sobie wiedzę o tym języku. Nauczysz się osadzać kod JavaScript w dokumentach HTML, przetwarzać dane i sterować wykonywaniem skryptu za pomocą konstrukcji warunkowych. Dowiesz się, jak korzystać z obiektowego modelu dokumentu, tworzyć i obsługiwać pliki cookie oraz procedury obsługi zdarzeń. Poznasz także techniki programowania obiektowego i sposoby wykrywania czy usuwania błędów. Przeczytasz również o technologii AJAX, opierającej się na języku JavaScript.

- Podstawowe elementy JavaScript
- Praca ze zmiennymi
- Interakcja z przeglądarką
- Wyrażenia warunkowe i pętle
- Organizacja kodu i korzystanie z funkcji
- Obiektowy Model Dokumentu
- Obiekty w JavaScript
- Testowanie skryptów
- Wykorzystanie JavaScript w technologii AJAX

Twój czas jest cenny – wykorzystaj go na poznanie JavaScript z pomocą nowoczesnych metod nauki!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści (podsumowanie)

Spis treści (teraz na poważnie)

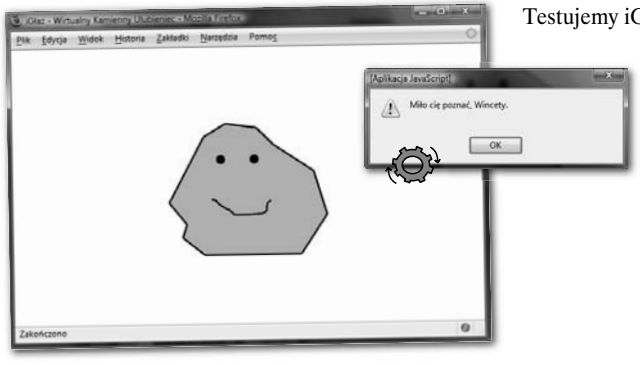
Wprowadzenie

Twój mózg koncentruje się na JavaScripcie. Siedzisz, próbując się czegoś *nauczyć*, ale Twój *mózg* twierdzi, że cała ta nauka *nie jest ważna*. Twój mózg twierdzi: „Lepiej zostawić miejsce na jakieś ważne rzeczy, takie jak to, których dzikich zwierząt należy unikać albo czy jeżdżenie nago na snowboardzie jest dobrym pomysłem, czy nie". W jaki zatem sposób *możesz* przekonać swój mózg, by uznał, że poznanie JavaScriptu to dla Ciebie kwestia życia lub śmierci?

Interaktywna WWW

1 W odpowiedzi na wirtualny świat

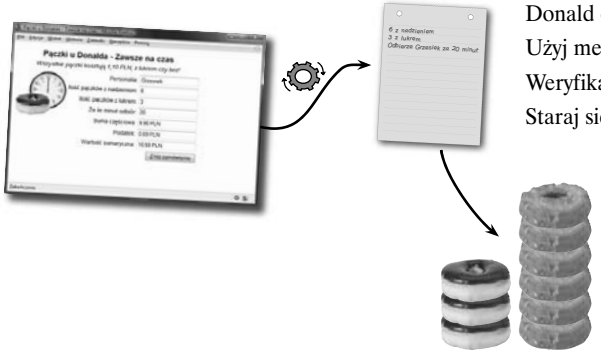
Czy męczy Cię już myślenie o WWW w kategoriach statycznych stron? To już widziałem i przerobiłem. Takie rzeczy zazwyczaj nazywają książkami. Trzeba przyznać, że doskonale się one nadają do czytania i nauki, i w ogóle są super. Ale... nie są **interaktywne**. To samo dotyczy także stron WWW, jeśli nie uzyskają one nieznacznej pomocy ze strony języka JavaScript. Pewnie, że można przesłać formularz i może nawet zastosować tu i tam kilka trików, używając w tym celu umiejętnie napisanego kodu HTML i CSS, jednak takie rozwiązania nie są w stanie „ożywić" martwych — statycznych stron WWW. Prawdziwa **interaktywność** wymaga znacznie większego **wkładu intelektualnego** i **nakładu pracy**, jednak zapewnia efekty, które *zwrócą się z nawiązką*.



Przechowywanie danych

2 Wszystko ma swoje miejsce

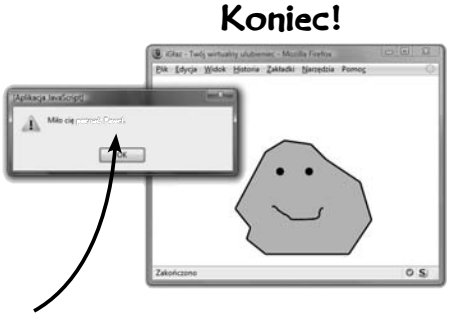
W praktyce często nie dostrzegamy znaczenia, jakie ma posiadanie miejsca do przechowywania swoich rzeczy. Korzystając z języka JavaScript, na pewno zwrócimy na to uwagę. W tym przypadku nie mamy bowiem luksusu posiadania osobnej garderoby lub garażu na trzy samochody. W JavaScripcie **wszystko musi mieć swoje miejsce**, a Twoim zadaniem jest zadbanie, by faktycznie tak się stało. W tym rozdziale będziemy się zajmować **danymi** oraz zagadnieniami, które ich dotyczą — jak *je reprezentować*, *przechowywać* oraz jak *je odnaleźć*, kiedy zostaną już gdzieś zapisane. Jako specjalista do spraw przechowywania danych w JavaScripcie będziesz w stanie zająć się dowolnym kodem, zapanować nad chaosem obecnym wśród używanych w nim danych, dzięki czemu będziesz mógł uporządkować go wedle swojej woli, korzystając przy tym z zalet wirtualnych etykiet i pojemników.



Poznajemy klienta

3 W głąb przeglądarki

Czasami kod JavaScript musi „wiedzieć", co się dzieje w świecie naokoło niego. Twoje skrypty mogą początkowo być tworzone jako kod umieszczany bezpośrednio w stronach WWW, jednak w ostateczności i tak będą istnieć i działać w świecie kreowanym przez przeglądarkę WWW, nazywaną także klientem. **„Sprytne" skrypty** często będą potrzebowały pewnych informacji o świecie, w którym „żyją"; w takich sytuacjach mogą zdobyć te dane, **komunikując się z przeglądarką**. Niezależnie od tego, czy jest to określenie wymiarów ekranu, czy też dostęp do jakiegoś przycisku przeglądarki, to utrzymując dobre stosunki z przeglądarką, skrypty mogą bardzo wiele zyskać.

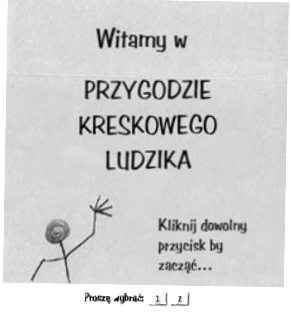


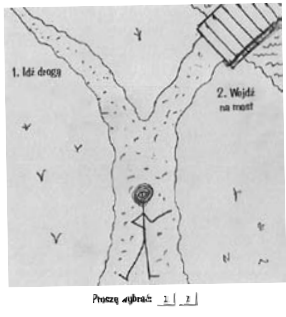
Podejmowanie decyzji

4

Jeśli droga się rozwidla, nie wahaj się skręcić

Życie polega na podejmowaniu decyzji. Stanąć czy jechać, mieszać czy wstrząsać, iść na ugodę czy do sądu... Tak naprawdę bez możliwości podejmowania decyzji nic nigdy nie udałoby się zrobić. To samo dotyczy JavaScriptu — **decyzje pozwalają skryptom dokonywać wyboru pomiędzy różnymi możliwymi wynikami**. *To właśnie podejmowanie decyzji tworzy „historię" naszego skryptu*, a swoją historię mają wszystkie skrypty, nawet te najbardziej prozaiczne.
Czy wierzę w to, co wpisał użytkownik, i bez wahania zarezerwuję mu uczestnictwo w ekspedycji badawczej Białowieży? A może bardzo dokładnie sprawdzę wpisane informacje, by nie okazało się, że tak naprawdę użytkownik chciałby jedynie dojechać autobusem w okolice Białowieży? Sam musisz podjąć decyzję i dokonać wyboru!





Pętle

5 Ryzykując powtórzeniem

Niektórzy mówią, że powtórzenia to podstawa sukcesu. Oczywiście, że robienie wciąż nowych i interesujących rzeczy jest ekscytujące, lecz jednak codzienne życie składa się z czynności, które wielokrotnie powtarzamy. Maniakalne czyszczenie rąk, nerwowe tiki bądź klikanie przycisku *Odpowiedz wszystkim* po odebraniu każdego dziwacznego lub śmiesznego e-maila! No dobrze, może w rzeczywistości powtórzenia nie zawsze są takie znowu wspaniałe. Niemniej w świecie JavaScriptu powtórzenia mogą być niesłychanie przydatne i użyteczne. Sam nie wiesz, jak często pojawia się potrzeba kilkakrotnego wykonania pewnego fragmentu kodu... I właśnie w takich sytuacjach w pełni można docenić zalety pętli. Gdyby nie one, musiałbyś spędzać bardzo dużo czasu, wielokrotnie kopiując i wklejając ten sam fragment kodu.

Funkcje

6 Redukuj i używaj wielokrotnie

Gdyby w świecie JavaScriptu zaistniał jakiś ruch środowiskowy, to zapewne na jego czele stanęłyby funkcje. Funkcje pozwalają na pisanie bardziej efektywnego kodu i sprawiają, że łatwiej można go wielokrotnie używać. Funkcje są zorientowane na wykonywanie konkretnych zadań i znacząco ułatwiają organizację kodu. Wygląda to jak całkiem dobry życiorys! W praktyce niemal wszystkie skrypty, może za wyjątkiem tych najprostszych, mogą skorzystać na reorganizacji kodu i zastosowaniu funkcji. Choć trudno jest ocenić znaczenie i wpływ przeciętnej funkcji, to jednak należy zaznaczyć, że odgrywają one znaczącą rolę w staraniach, by skrypty były możliwie jak najbardziej przyjazne dla środowiska.

Formularze i sprawdzanie poprawności danych

7

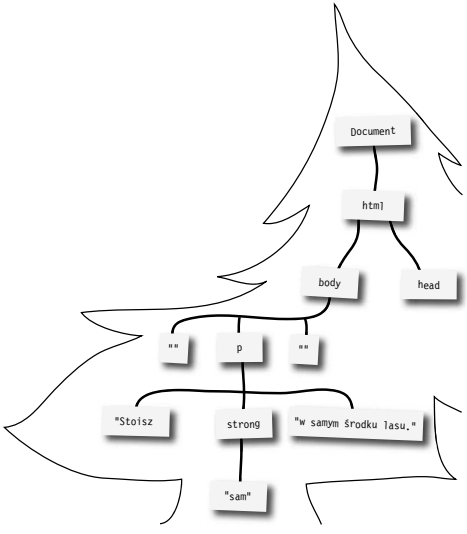
Aby użytkownik powiedział nam wszystko

Nie musisz być ani szczególnie grzeczny, ani chytry, by pobierać od użytkowników dane, korzystając z możliwości, jakie zapewnia JavaScript. Nie ma natomiast żadnych wątpliwości co do tego, że musisz zachować przy tym dużą ostrożność i uwagę. Ludzie wykazują dziwną tendencję do popełniania błędów, co oznacza, że *nie możesz* z góry zakładać, iż informacje podawane przez użytkowników będą *precyzyjne* bądź prawidłowe. I tu do akcji wkracza JavaScript. Przekazując **informacje wpisane w polach formularza do odpowiedniego kodu** napisanego w tym języku, i to bezpośrednio po ich podaniu, możesz nie tylko **poprawić niezawodność aplikacji**, lecz także nieco **odciążyć serwer**. Musimy *oszczędzać cenną przepustowość łączy na ważniejsze rzeczy*, takie jak wideo z zapierającymi dech w piersiach wyczynami kaskaderskimi lub urocze zdjęcia naszego ulubionego zwierzaka.

Modyfikacje stron WWW

8 Krojenie i przyprawianie HTML-a przy użyciu DOM

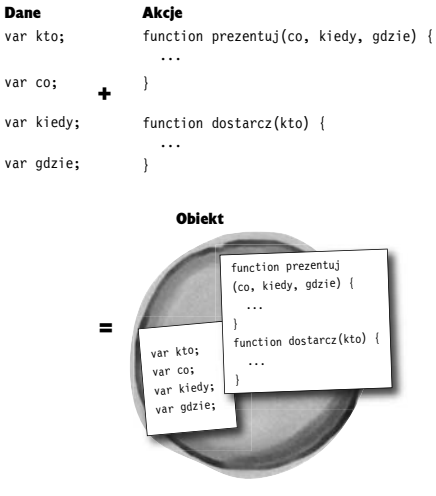
Uzyskanie kontroli nad zawartością stron WWW dzięki wykorzystaniu JavaScriptu w dużym stopniu przypomina pieczenie. Nie wiąże się co prawda z takim samym bałaganem... jednak z drugiej strony nie daje w efekcie tak smakowitych rezultatów. Niemniej masz *pełny dostęp do składników HTML-a* tworzących stronę WWW, a co więcej — masz możliwość *modyfikowania* przepisu tej strony. A zatem **JavaScript pozwala na modyfikowanie kodu HTML strony** i dostosowywanie go do własnych potrzeb, co daje bardzo ciekawe możliwości. A wszystko to dzięki *kolekcji standardowych obiektów*, określanej mianem **obiektowego modelu dokumentu** (w skrócie **DOM**, od angielskich słów *Document Object Model*).



9 Ożywianie danych

Obiekty jako Frankendane

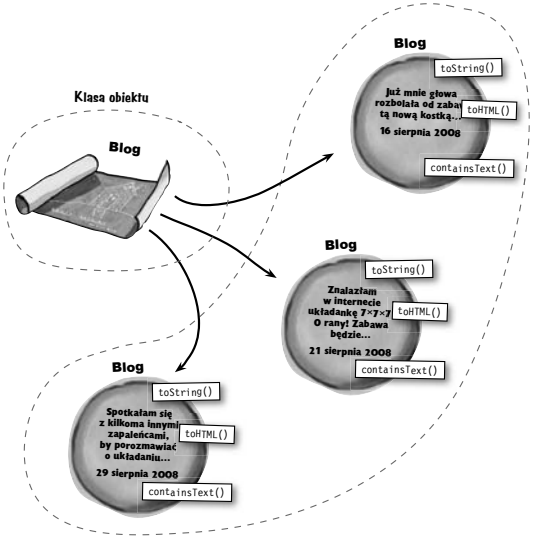
Obiekty JavaScriptu nie są tak makabryczne, jak dobry lekarz mógłby przypuszczać. Niemniej są ciekawe, gdyż łączą różne elementy języka w jedną całość, która ma większe możliwości niż suma jej składowych. **Obiekty łączą dane z akcjami**, tworząc w ten sposób nowy typ danych, który w znacznie większej mierze niż opisane wcześniej typy przypomina coś „żywego". W efekcie możemy stworzyć *tablice, które same będą potrafiły posortować swoją zawartość, łańcuchy znaków dysponujące możliwością samodzielnego przeszukiwania swojej treści*, a skryptom może wyrosnąć sierść i mogą zacząć wyć do księżyca! No dobra, te dwie ostatnie rzeczy nie są możliwe, ale rozumiesz już, o co chodzi.



10 Tworzenie własnych obiektów

Zrób to po swojemu, używając własnych obiektów

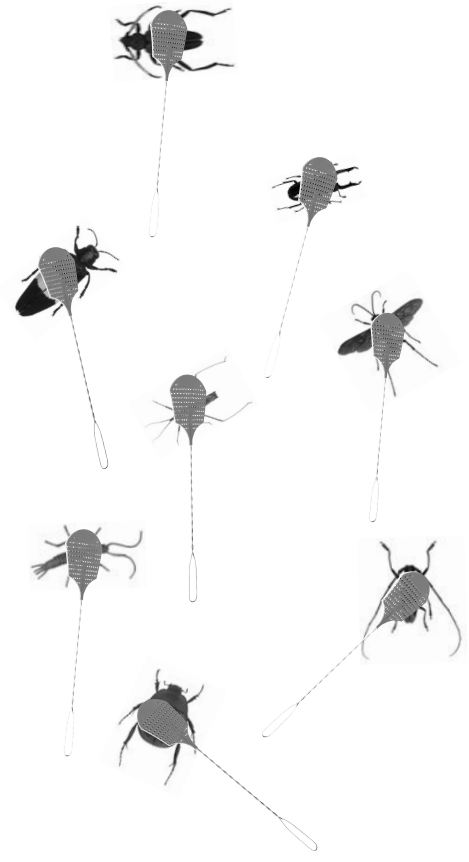
Gdyby to było takie łatwe, na pewno byś sobie z tym sam poradził. W świecie JavaScriptu nie ma gwarancji zwrotu poniesionych kosztów, jednak nikt nigdy nie będzie Ci bronił robić wszystkiego po swojemu. Niestandardowe obiekty w JavaScripcie są odpowiednikiem świetnej, wielkiej, gorącej, mistrzowskiej, okazyjnej promocji. To jeden niestandardowy kubek kawy! A dzięki własnym obiektom JavaScriptu możesz zaparzyć sobie kawę, która zrobi dokładnie to, co będziesz chciał — a wszystko to dzięki korzyściom, jakie zapewniają właściwości i metody. W efekcie powstaje obiektowy, nadający się do wielokrotnego stosowania kod, który rozszerza możliwości języka JavaScript... tylko i wyłącznie dla Ciebie!



Zabijaj pluskwy — na śmierć!

11 Dobre skrypty na złej drodze

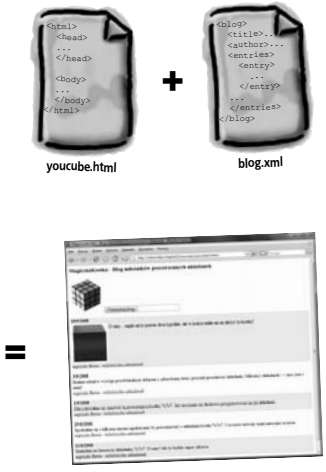
Nawet najlepiej zaplanowany kod JavaScript może czasami zawieść. Kiedy tak się stanie, a kiedyś na pewno to nastąpi, to Twoim zadaniem będzie opanować nerwy i nie wpadać w panikę. Najlepsi programiści to nie tacy, którzy nigdy nie popełniają błędów — tacy są raczej kłamcami. Najlepsi programiści to tacy, którzy **potrafią wytropić pluskwy i pozbyć się błędów**, które sami popełnili. Co ważniejsze, najlepsi pogromcy pluskiew **rozwijają dobre praktyki kodowania**, które minimalizują niebezpieczeństwo popełniania najbardziej złośliwych i trudnych do wytropienia błędów. Z drugiej strony *trochę prewencji nie zaszkodzi*. Pamiętaj, że pluskwy pojawiają się zawsze, będziesz zatem potrzebował arsenału, by z nimi walczyć...



Dynamiczne dane

12 Szybkie i wrażliwe aplikacje internetowe

W nowoczesnym internecie bardzo duże znaczenie ma szybkość i trafność reakcji — powszechnie oczekuje się, że strony będą błyskawicznie reagować na każdą zachciankę użytkownika. Albo przynajmniej marzą o tym niektórzy użytkownicy i twórcy aplikacji internetowych. Ważną rolę w realizacji tego marzenia odgrywa język JavaScript, stanowiący kluczowy element technologii o nazwie **Ajax**, zapewniającej możliwość dynamicznego modyfikowania zawartości i wyglądu stron WWW. Dzięki Ajaksowi strony WWW w znacznie większym stopniu przypominają standardowe aplikacje komputerowe, gdyż mogą **szybko i dynamicznie pobierać oraz zapisywać dane, błyskawicznie odpowiadając na poczynania użytkownika** i to bez przeładowywania stron.



6. Funkcje

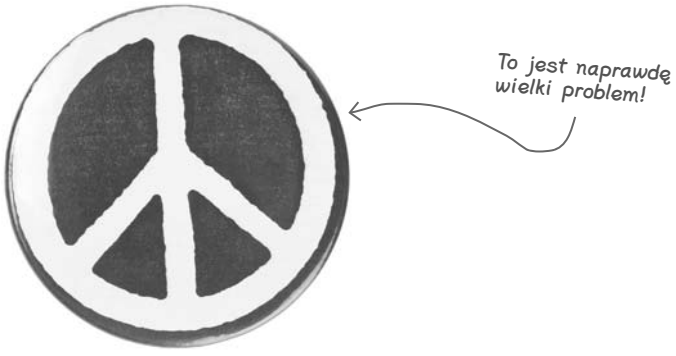
Redukuj i używaj wielokrotnie

Gdyby w świecie JavaScriptu zaistniał jakiś ruch środowiskowy, to zapewne na jego czele stanęłyby funkcje. Funkcje pozwalają na pisanie bardziej efektywnego kodu i sprawiają, że łatwiej można go wielokrotnie używać. Funkcje są zorientowane na wykonywanie konkretnych zadań i znacząco ułatwiają organizację kodu. Wygląda to jak całkiem dobry życiorys! W praktyce niemal wszystkie skrypty, może za wyjątkiem tych najprostszych, mogą skorzystać na reorganizacji kodu i zastosowaniu funkcji. Choć trudno jest ocenić znaczenie i wpływ przeciętnej funkcji, to jednak należy zaznaczyć, że odgrywają one znaczącą rolę w staraniach, by skrypty były możliwie jak najbardziej przyjazne dla środowiska.

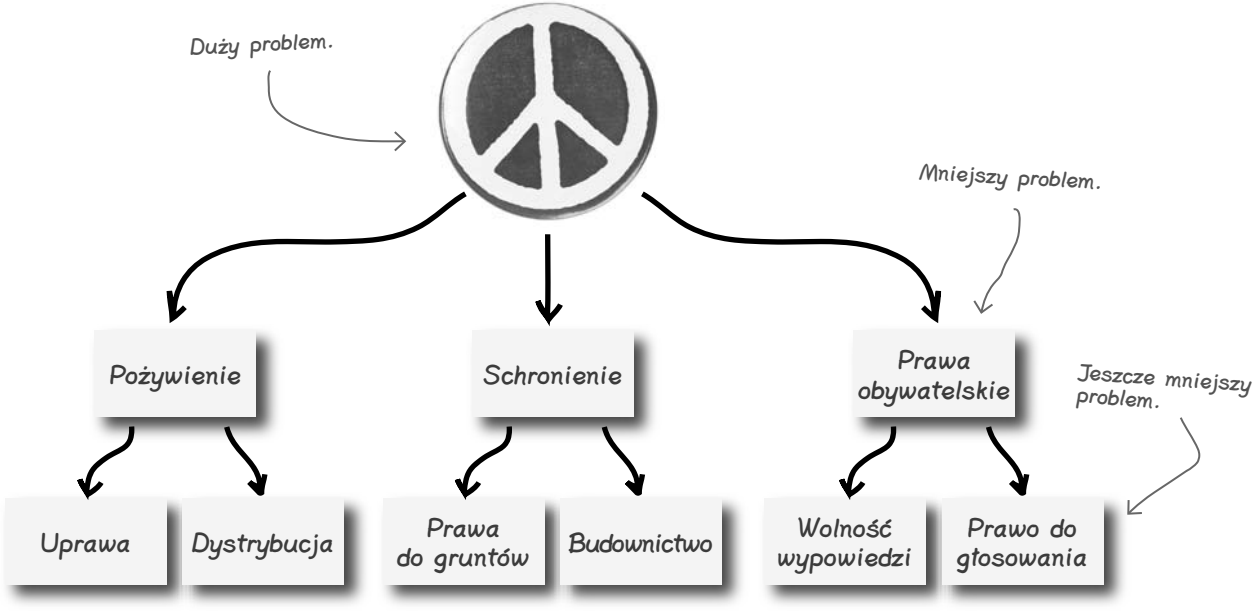
Matka wszystkich problemów

Skoro już o tym wspomnieliśmy, to warto zauważyć, że tworzenie skryptów na stronach WWW ma na celu rozwiązywanie problemów. Niezależnie od tego, jak duży i skomplikowany jest problem, jeśli zostanie on odpowiednio przemyślany, to zawsze znajdzie się rozwiązanie. No ale co z **naprawdę wielkimi problemami**?

Pokój na świecie



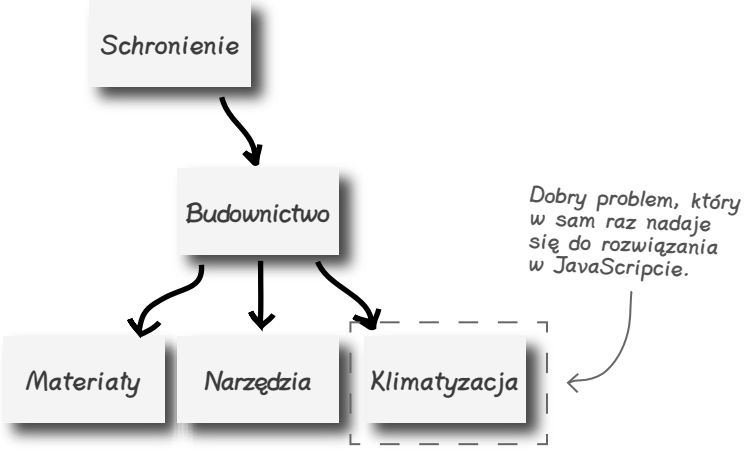
Cała sztuczka z rozwiązywaniem dużych problemów polega na podzieleniu ich na mniejsze, które będzie można łatwiej rozwiązać. A jeśli i te problemy są zbyt duże, to i je należy podzielić.



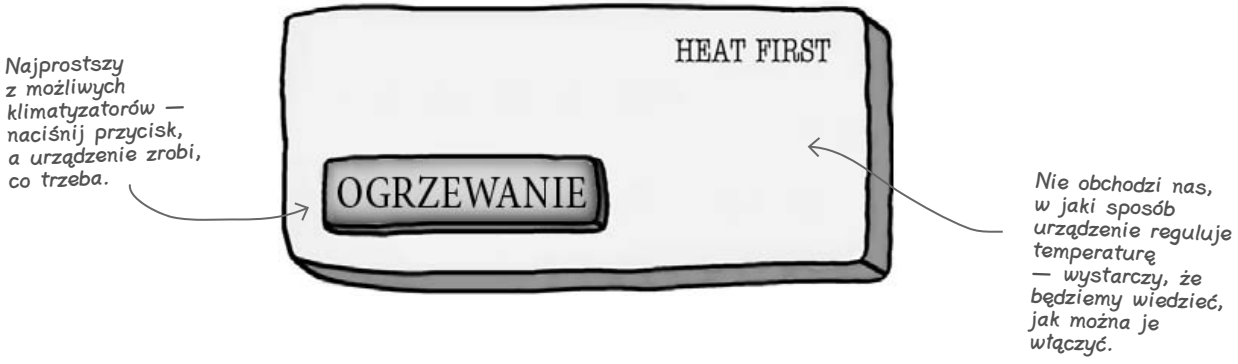
Kontynuuj ten proces dalej i dalej, i dalej…

Rozwiązuj duże problemy poprzez rozwiązywanie problemów mniejszych

Kontynuując dzielenie problemu pokoju na świecie na coraz to mniejsze problemy, dojdziesz w końcu do problemu na tyle małego, że można go rozwiązać, wykorzystując JavaScript.



Spróbuj wymyślić JavaScriptowy odpowiednik problemu sterowania klimatyzacją, który wywoływałby skryptowy odpowiednik klimatyzatora, aby ten regulował temperaturę otoczenia. Najprostsze urządzenie do sterowania klimatyzacją, jakie można sobie wyobrazić, składałoby się wyłącznie z przycisku *Ogrzewanie*.



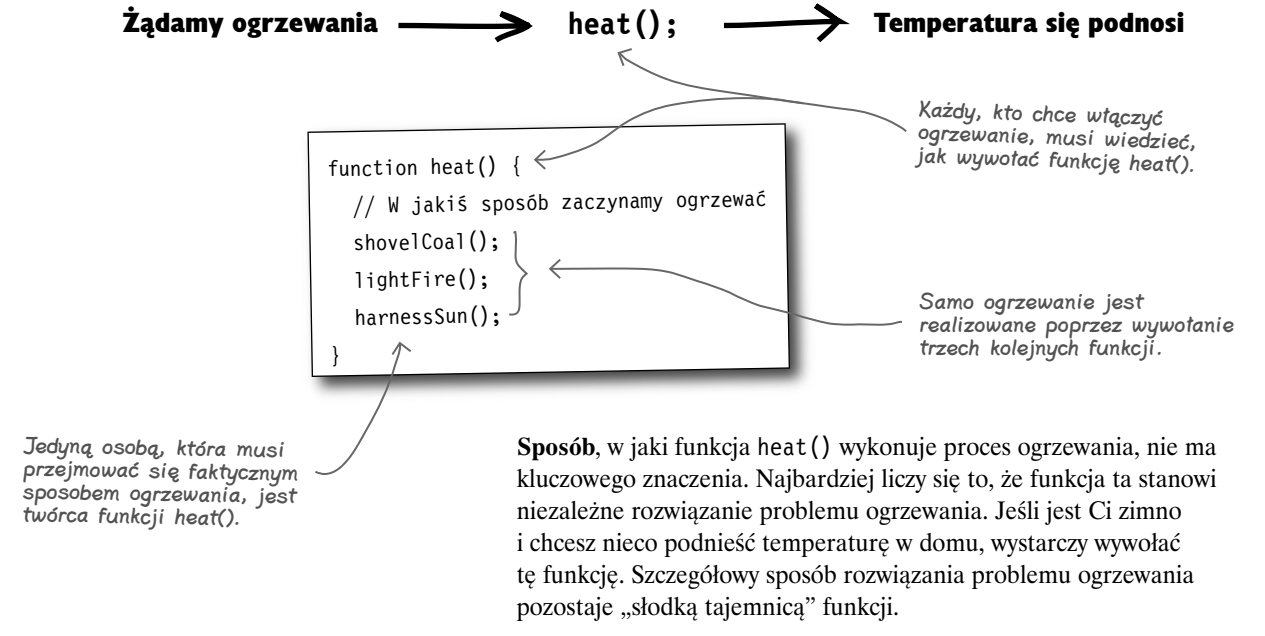
Zauważ, że urządzenie nie zdradza żadnej informacji, która mogłaby nam powiedzieć, w jaki sposób reguluje się temperaturę otoczenia. Wystarczy, że naciśniemy przycisk, a wszystko zostanie zrobione za nas. Ot — i tak oto problem klimatyzacji został rozwiązany!

Funkcje jako narzędzia do rozwiązywania problemów

Przycisk *Ogrzewanie* służący do włączania klimatyzacji jest odpowiednikiem **funkcji** w języku JavaScript. Cały pomysł jest bardzo podobny do działania rzeczywistej klimatyzacji — ktoś żąda włączenia ogrzewania w domu, a funkcja mu to zapewnia. Szczegóły procesu ogrzewania są ukryte wewnątrz funkcji i nie mają najmniejszego znaczenia dla kodu, który ją wywołuje. Pod tym względem można wyobrazić sobie funkcję jako pewnego rodzaju „czarną skrzynkę” — informacje mogą do niej wpływać i wypływać, jednak to, co się dzieje wewnątrz, zależy wyłącznie od skrzynki i dlatego nie ma znaczenia dla kodu umieszczonego poza nią.

Funkcje przekształcają duże problemy w mniejsze.

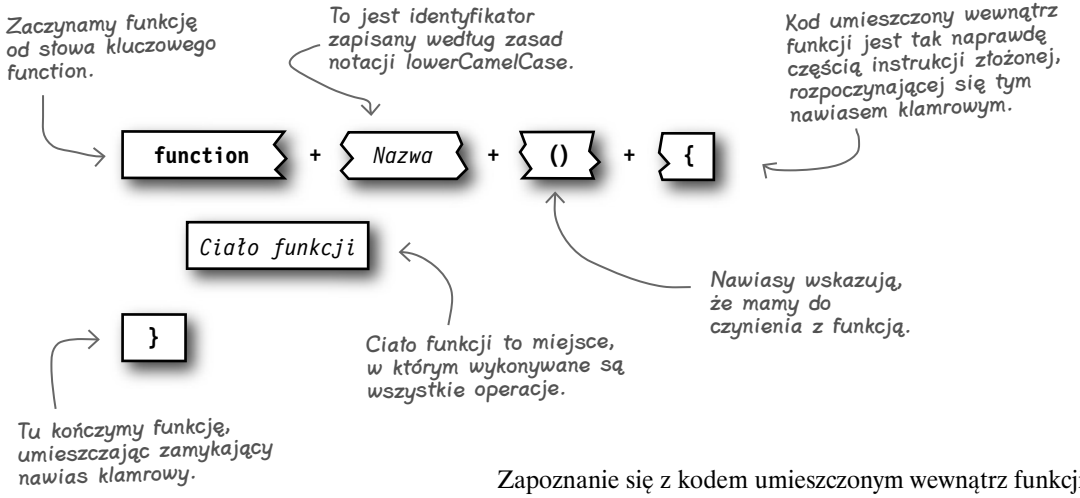
Stworzenie odpowiednika przycisku *Ogrzewanie* w kodzie JavaScript wymagałoby wywołania funkcji o nazwie `heat()`...



Sposób, w jaki funkcja `heat()` wykonuje proces ogrzewania, nie ma kluczowego znaczenia. Najbardziej liczy się to, że funkcja ta stanowi niezależne rozwiązanie problemu ogrzewania. Jeśli jest Ci zimno i chcesz nieco podnieść temperaturę w domu, wystarczy wywołać tę funkcję. Szczegółowy sposób rozwiązania problemu ogrzewania pozostaje „słodką tajemnicą” funkcji.

Tworzenie funkcji w praktyce

Kiedy podejmiesz decyzję napisania funkcji, automatycznie staniesz się autorem rozwiązania pewnego problemu. Stworzenie funkcji wymaga zastosowania spójnej składni, wiążącej jej nazwę z kodem, który dana funkcja ma wykonywać. W najprostszej wersji składnia funkcji ma następującą postać:



Zapoznanie się z kodem umieszczonym wewnątrz funkcji `heat()` pozwala spojrzeć na składnię funkcji z nieco innej perspektywy:

```
function heat() {
  // W jakiś sposób zaczynamy ogrzewać
  shovelCoal();
  lightFire();
  harnessSun();
}
```

Tak naprawdę ogrzewanie jest realizowane wewnątrz funkcji.

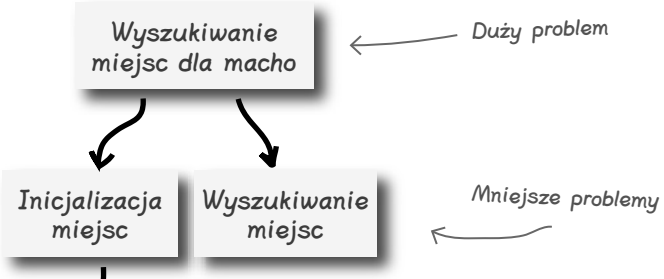
Ciało funkcji jest zapisane wewnątrz nawiasów klamrowych — czyli w rzeczywistości jest to doskonale nam już znana instrukcja złożona.

WYTĘŻ UMYSŁ

Gdzie we wcześniejszej części książki do rozwiązywania problemów wykorzystaliśmy funkcje?

Funkcje, które już poznałeś

Jeśli chcesz znaleźć doskonałe przykłady funkcji służących do rozwiązywania konkretnych problemów, wystarczy, że zajrzysz do aplikacji Mandango — wyszukiwarki miejsc dla prawdziwych macho (znajdziesz je na serwerze FTP wydawnictwa Helion, *ftp://ftp.helion.pl/przyklady/hfjsc.zip*). Przykładem tym jest funkcja służąca do inicjalizacji danych dotyczących dostępności miejsc.



Podproblem polegający na inicjalizacji miejsc był na tyle mały, że udało się go nam rozwiązać i zaimplementować w postaci funkcji, a konkretnie rzecz biorąc: funkcji `initSeats()`:

```
function initSeats() {
  // Inicjalizacja wyglądu wszystkich miejsc
  for (var i = 0; i < seats.length; i++) {
    for (var j = 0; j < seats[i].length; j++) {
      if (seats[i][j]) {
        // Określamy dostępne miejsce
        document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_avail.png";
        document.getElementById("seat" + (i * seats[i].length + j)).alt = "Miejsce dostępne";
      }
      else {
        // Określamy zajęte (niedostępne) miejsce
        document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_unavail.png";
        document.getElementById("seat" + (i * seats[i].length + j)).alt = "Miejsce zajęte";
      }
    }
  }
}
```

Funkcja `initSeats()` jest jednym z elementów aplikacji Mandango. Funkcja ta jest wywoływana dopiero w procedurze obsługi zdarzeń `onload`; a zatem zostanie ona wykonana dopiero po wczytaniu całej strony.

Funkcja initSeats() nie jest jedynym narzędziem do rozwiązywania problemów wykorzystywanym w aplikacji Mandango.

```
<body onload="initSeats();">
    <div style="margin-top:25px; text-align:center">
      <img id="seat0" src="" alt="" />
      ...
      <img id="seat1" src="" alt="" />
      <img id="seat35" src="" alt="" /><br />
      <input type="button" id="findseats" value="Szukaj miejsc" onclick="findSeats();" />
    </div>
  </body>
</html>
```

Nie ma niemądrych pytań

P: Możesz jeszcze raz przypomnieć konwencje używane podczas określana nazw funkcji?

O: Podczas określania nazw identyfikatorów w JavaScripcie używana jest konwencja lowerCamelCase, w myśl której pierwsze słowo identyfikatora jest zapisywane małą literą, a wszystkie pozostałe — wielką. A zatem funkcja służąca do oceniania filmów nosiłaby nazwę `ocenFilm()`, natomiast funkcja usuwająca z sali kinowej widza, który zapomniał wyłączyć telefonu komórkowego, mogłaby się nazywać: `usunHalasliwegoWidza()`.

P: Czy funkcje zawsze służą do dzielenia dużych problemów na mniejsze?

O: Niekoniecznie. Czasami przydatne może być także zastosowanie funkcji do dzielenia pracy nad kodem. Innymi słowy, może się zdarzyć, że jeden problem będzie rozwiązywany przez kilka współpracujących ze sobą funkcji.

W takim przypadku podział kodu na funkcje ma nam umożliwić organizację pracy i ułatwić tworzenie funkcji, z których każda będzie mieć jedno precyzyjnie określone przeznaczenie. Na podobnej zasadzie pracownicy są zatrudniani na stanowiskach o różnych nazwach, by każdy z nich mógł się skupić na rozwiązywaniu unikalnych i konkretnych problemów. Takie funkcje mogą, choć nie muszą, rozwiązywać konkretne problemy; niemniej na pewno poprawiają strukturę skryptów, gdyż pozwalają dzielić ich kod na części.

P: A skąd wiadomo, kiedy i jaki fragment kodu należy zaimplementować jako funkcję?

O: Niestety, nie ma żadnego magicznego sposobu, który pozwalałby określić, kiedy warto napisać fragment kodu jako funkcję. Można jednak wskazać pewne czynniki, które można potraktować jako sygnały i podpowiedzi dotyczące możliwości tworzenia funkcji. Jednym z nich jest sytuacja, w której pewien fragment kodu będzie się powtarzać. Powielanie kodu niemal nigdy nie jest ani dobrym, ani zalecanym rozwiązaniem, gdyż zmusza nas do pielęgnacji i ewentualnej modyfikacji kodu w kilku miejscach skryptu.

A zatem taki powtarzający się kod doskonale nadaje się do zaimplementowania w formie funkcji. Innym sygnałem, na jaki należy zwracać uwagę, są długie, trudne do zarządzania bloki kodu, które w prosty sposób można podzielić na kilka logicznych części. Taki kod to doskonała okazja do zastosowania „podziału pracy" — czyli podzielenia tego kodu na części i umieszczenia ich w odrębnych funkcjach.

P: Czy nie wspominałeś o tym, że funkcje mogą akceptować argumenty i zwracać wartości? Czy o czymś zapomniałem?

O: Nie. Bez wątpienia są funkcje, które wymagają przekazania danych i jednocześnie zwracają jakieś wyniki. Jak się okaże, właśnie taka będzie funkcja `heat()`, którą poznasz w dalszej części rozdziału.

Ćwiczenie

Nazwa funkcji ma bardzo duże znaczenie, gdyż może przekazywać informacje o jej przeznaczeniu. Spróbuj wymyślić nazwy dla opisanych poniżej funkcji, pamiętając przy tym, by w odpowiedni sposób je zapisać.

Poproś o fotel przy przejściu →  → **Odbierz bilet.**

.....

Poproś o zwrot kosztów →  → **Odbierz przyznane pieniądze.**

.....

Wyrzuć popcorn →  → **Popcorn został rozsypany na innych widzów.**

.....

Ćwiczenie Rozwiązanie

Nazwa funkcji ma bardzo duże znaczenie, gdyż może przekazywać informacje o jej przeznaczeniu. Spróbuj wymyślić nazwy dla opisanych poniżej funkcji, pamiętając przy tym, by w odpowiedni sposób je zapisać.

Poproś o fotel przy przejściu →  → **Odbierz bilet.**

poprosOMiejscePrzyPrzejsciu()

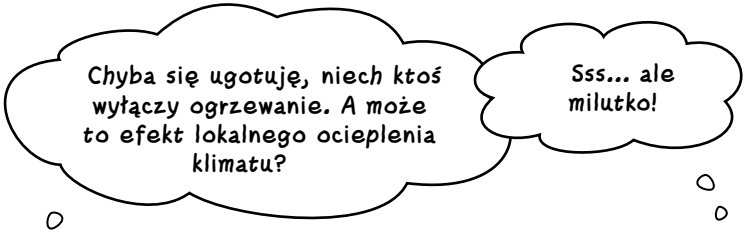
Poproś o zwrot kosztów →  → **Odbierz przyznane pieniądze.**

poprosOZwrot()

Wyrzuć popcorn →  → **Popcorn został rozsypany na innych widzów.**

wyrzucPopcorn()

Z powodzeniem można by wymyślić inne nazwy dla tych funkcji. W każdym razie są to przykłady sensownych nazw zapisanych zgodnie z założeniami notacji lowerCamelCase.

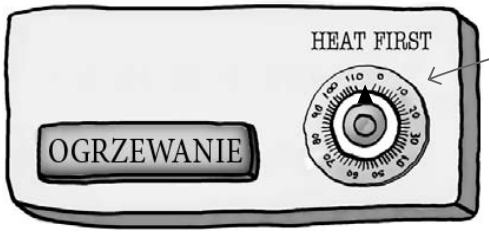


Zbyt gorąco by cokolwiek zrobić

W międzyczasie okazało się, że próby zapewnienia pokoju na świecie poprzez kontrolę temperatury otoczenia zaczynają przynosić więcej szkód niż pożytku. Wygląda na to, że przycisk *Ogrzewanie* działa aż nazbyt dobrze… choć może to jest problem z funkcją `heat()`, do której powinniśmy przekazywać więcej informacji? Niezależnie od przyczyny, coś z tym trzeba zrobić.

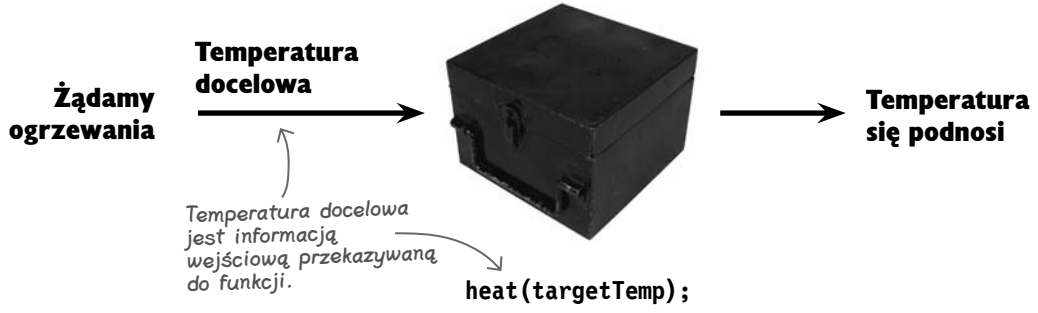
Lepsza klimatyzacja dzięki większej ilości danych

Wróćmy zatem do naszej klimatyzacji… Urządzenie to nie wie, kiedy należy przerwać ogrzewanie, gdyż nie określiliśmy żadnej temperatury docelowej. A zatem nasza klimatyzacja nie ma wystarczającej ilości informacji, by efektywnie rozwiązać problem. Oznacza to, że kiedy ktoś naciśnie przycisk *Ogrzewanie*, klimatyzacja zacznie ogrzewać… i nigdy już nie przestanie!



Pokrętło regulacji temperatury pozwala użytkownikom określić temperaturę docelową.

Poprawiony klimatyzator pozwala teraz na określenie temperatury docelowej, stanowiącej informację wejściową, dzięki której można usprawnić proces ogrzewania.



Ćwiczenie

Napisz kod poprawionej funkcji `heat()`, która pobiera temperaturę docelową i działa (czyli generuje ciepło) wyłącznie wtedy, gdy temperatura zmierzona w danej chwili jest mniejsza od docelowej. Podpowiedź: skorzystaj z hipotetycznej funkcji `getTemp()`, by określić, jaka jest temperatura w danej chwili.

Napisaliśmy pierwszy wiersz kodu, by ułatwić Ci rozwiązanie ćwiczenia.

```
function heat(targetTemp)
```

Ćwiczenie Rozwiązanie

Napisz kod poprawionej funkcji `heat()`, która pobiera temperaturę docelową i działa (czyli generuje ciepło) wyłącznie wtedy, gdy temperatura zmierzona w danej chwili jest mniejsza od docelowej. Podpowiedź: skorzystaj z hipotetycznej funkcji `getTemp()`, by określić, jaka jest temperatura w danej chwili.

```
function heat(targetTemp)
  while (getTemp() < targetTemp) {
    // W jakiś sposób zaczynamy ogrzewać
    shovelCoal();
    lightFire();
    harnessSun();
  }
}
```

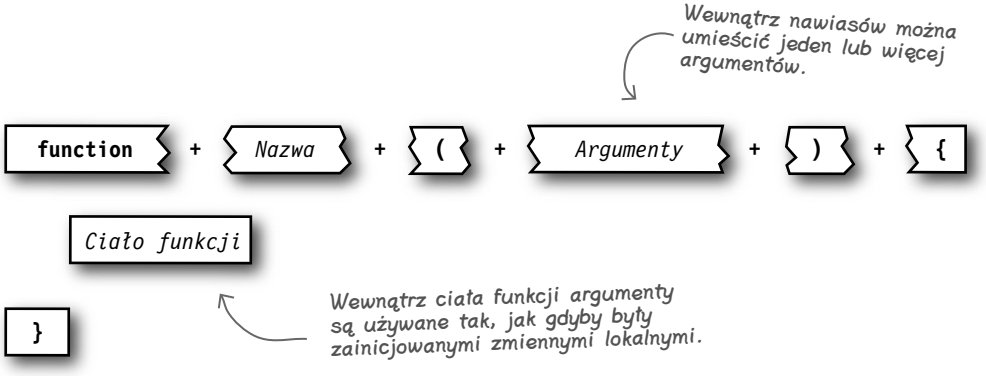
Temperatura docelowa jest przekazywana do funkcji jako argument jej wywołania.

Obecnie funkcja generuje ciepło wyłącznie wtedy, gdy aktualna temperatura jest mniejsza od temperatury docelowej.

Temperatura docelowa jest używana jako element warunku sprawdzanego w pętli while.

Przekazywanie informacji do funkcji

Dane są przekazywane do funkcji przy użyciu tak zwanych **argumentów**.
Przyjrzyj się jeszcze raz składni funkcji i zwróć szczególną uwagę na argumenty umieszczane pomiędzy nawiasami, za nazwą tworzonej funkcji:



Nie ma żadnego limitu określającego maksymalną liczbę argumentów, jakie można przekazać do funkcji, jednak ze względów praktycznych warto dołożyć starań, by nie było ich więcej niż dwa lub trzy. Argumentem przekazywanym do funkcji może być niemal każda informacja: stała (`Math.PI`), zmienna (`temp`), jak również literał (`23`).

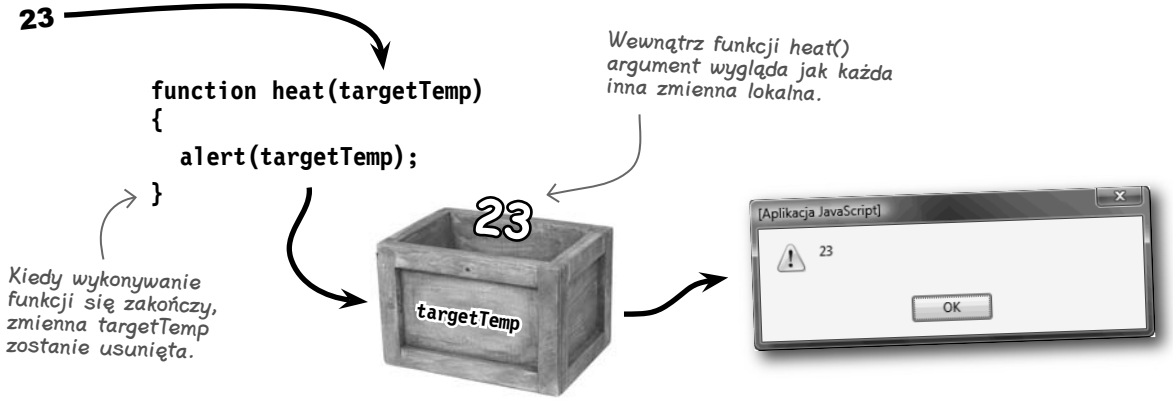
Argumenty funkcji jako dane

Kiedy korzystając z mechanizmu argumentów, przekazujemy do funkcji pewne dane, to wewnątrz tej funkcji stają się one zainicjowanymi zmiennymi lokalnymi. Przykładem może być funkcja `heat()`, do której przekazujemy argument określający temperaturę docelową:

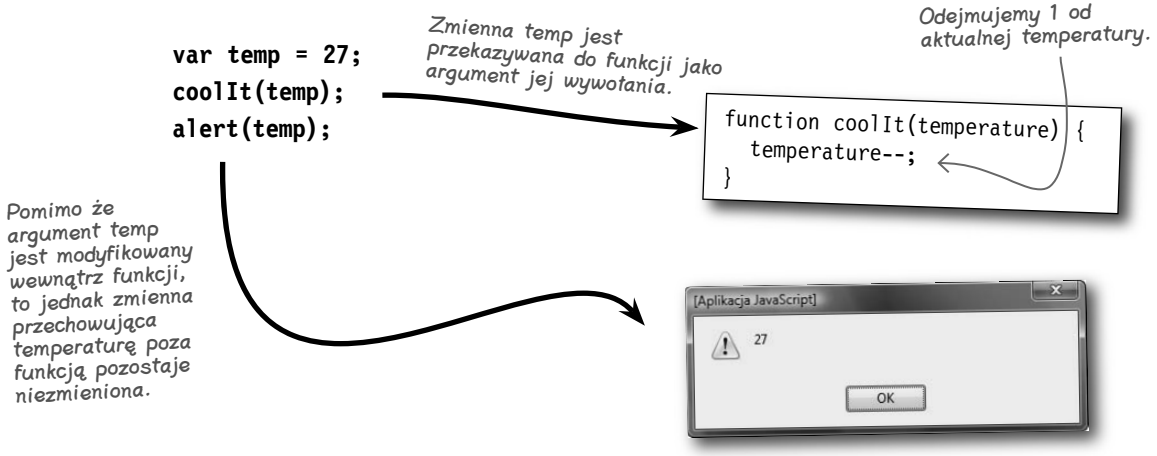
```
heat(23);
```

Zadana temperatura docelowa jest przekazywana do funkcji jako literał liczbowy.

Wewnątrz funkcji `heat()` do informacji o temperaturze docelowej odwołujemy się jak do zwyczajnej zainicjowanej zmiennej lokalnej o wartości 23. Wartość tę można wyświetlić, wywołując wewnątrz funkcji `heat()` informacyjne okienko dialogowe:



Choć argumenty funkcji są bardzo podobne do zmiennych lokalnych, to jednak wszelkie modyfikacje argumentów wprowadzane **wewnątrz** funkcji **nie** mają żadnego wpływu na jakiekolwiek zmienne **poza** funkcją. Warto zapamiętać, że reguła ta nie odnosi się do obiektów przekazywanych do funkcji, jednak tymi zagadnieniami zajmiemy się bardziej szczegółowo w rozdziałach 9. i 10.



Funkcje eliminują powtarzający się kod

Funkcji można z powodzeniem używać nie tylko do dzielenia dużych problemów na mniejsze, które będzie można łatwiej rozwiązać. Oprócz tego — dzięki możliwości **uogólniania zadań** — funkcje doskonale nadają się do eliminowania wielokrotnie powtarzającego się kodu. Choć niejednokrotnie powtarzający się kod nie jest dokładnie identyczny we wszystkich miejscach, to jednak często można go uogólnić i stworzyć jedną wersję, która następnie zostanie umieszczona w funkcji. Następnie można korzystać z tej funkcji, zamiast powielać fragment kodu.

Poniżej przedstawiliśmy trzy różne fragmenty kodu, które wykonują podobne zadania. Fragmenty te z powodzeniem można uogólnić i zaimplementować w formie jednej ogólnej funkcji nadającej się do **wielokrotnego zastosowania**:

Wyrażenie obliczające cenę biletu ze zniżką jest niepotrzebnie powtarzane.

```
// Bilet na seans popołudniowy jest tańszy o 10%
biletPopoludniowy = biletNormalny * (1 – 0.10);
```

```
// Bilet dla seniorów jest tańszy o 15%
biletSeniorski = biletNormalny * (1 – 0.15 );
```

```
// Bilet dla dzieci jest tańszy o 20%
biletDzieciecy = biletNormalny * (1 – 0.20);
```

Powyższe zadania polegają na obliczeniu ceny trzech różnych zniżkowych biletów do kina. Można je jednak uogólnić do jednego zadania, które będzie obliczać cenę biletu na podstawie dowolnej zniżki, określonej jako wartość procentowa.

Funkcje potrafią także zwracać dane.

```
function cenaZeZnizka(cena, znizkaProcentowa) {
  return (cena * (1 – (znizkaProcentowa / 100)));
}
```

Dysponując funkcją obliczającą ceny biletów ze zniżką, możemy zmodyfikować trzy przedstawione wcześniej zadania, poprawiając przy tym ich efektywność:

```
// Bilet na seans popołudniowy jest tańszy o 10%
biletPopoludniowy = cenaZeZnizka(biletNormalny, 10);
```

```
// Bilet dla seniorów jest tańszy o 15%
biletSeniorski = cenaZeZnizka(biletNormalny, 15);
```

```
// Bilet dla dzieci jest tańszy o 20%
biletDzieciecy = cenaZeZnizka(biletNormalny, 20);
```

BĄDŹ ekspertem do spraw wydajności

Poniżej przedstawiliśmy kod funkcji findSeats() z aplikacji Mandango w całej jego okazałości. Korzystając ze swojej nowej wiedzy dotyczącej efektywności, zakreśl podobne fragmenty kodu, które można by w uogólnionej postaci zaimplementować jako funkcję.

```
function findSeats() {
  // Jeśli jakieś miejsce już zostało wybrane, to ponownie inicjujemy
  // wszystkie miejsca, by anulować wcześniejszy wybór.
  if (selSeat >= 0) {
    selSeat = -1;
    initSeats();
  }

  // Przeglądamy wszystkie miejsca, poszukując dostępnych
  var i = 0, finished = false;
  while (i < seats.length && !finished) {
    for (var j = 0; j < seats[i].length; j++) {
      // Sprawdzamy, czy aktualnie analizowane miejsce i dwa następne są dostępne
      if (seats[i][j] && seats[i][j + 1] && seats[i][j + 2]) {
        // Ustawiamy wybór i aktualizujemy wygląd miejsca
        selSeat = i * seats[i].length + j;
        document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_select.png";
        document.getElementById("seat" + (i * seats[i].length + j)).alt = "Twoje miejsce";
        document.getElementById("seat" + (i * seats[i].length + j + 1)).src = "seat_select.png";
        document.getElementById("seat" + (i * seats[i].length + j + 1)).alt = "Twoje miejsce";
        document.getElementById("seat" + (i * seats[i].length + j + 2)).src = "seat_select.png";
        document.getElementById("seat" + (i * seats[i].length + j + 2)).alt = "Twoje miejsce";

        // Prosimy użytkownika o zaakceptowanie wyboru miejsca
        var accept = confirm("Miejsca od " + (j + 1) + " do " + (j + 3) +
          " w rzędzie " + (i + 1) + " są wolne. Wybrać je?");
        if (accept) {
          // Użytkownik zaakceptował miejsce — kończymy (wychodzimy z wewnętrznej pętli)
          finished = true;
          break;
        }
        else {
          // Użytkownik odrzucił wybór, zatem anulujemy wybór i szukamy dalej
          selSeat = -1;
          document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_avail.png";
          document.getElementById("seat" + (i * seats[i].length + j)).alt = "Miejsce dostępne";
          document.getElementById("seat" + (i * seats[i].length + j + 1)).src = "seat_avail.png";
          document.getElementById("seat" + (i * seats[i].length + j + 1)).alt = "Miejsce dostępne";
          document.getElementById("seat" + (i * seats[i].length + j + 2)).src = "seat_avail.png";
          document.getElementById("seat" + (i * seats[i].length + j + 2)).alt = "Miejsce dostępne";
        }
      }
    }

    // Inkrementujemy licznik zewnętrznej pętli
    i++;
  }
}
```

BĄDŹ ekspertem do spraw wydajności

Poniżej przedstawiliśmy kod funkcji findSeats() z aplikacji Mandango w całej jego okazałości. Korzystając ze swojej nowej wiedzy dotyczącej efektywności, zakreśl podobne fragmenty kodu, które można by w uogólnionej postaci zaimplementować jako funkcję.

```
function findSeats() {
  // Jeśli jakieś miejsce już zostało wybrane, to ponownie inicjujemy
  // wszystkie miejsca, by anulować wcześniejszy wybór.
  if (selSeat >= 0) {
    selSeat = -1;
    initSeats();
  }

  // Przeglądamy wszystkie miejsca, poszukując dostępnych
  var i = 0, finished = false;
  while (i < seats.length && !finished) {
    for (var j = 0; j < seats[i].length; j++) {
      // Sprawdzamy, czy aktualnie analizowane miejsce i dwa następne są dostępne
      if (seats[i][j] && seats[i][j + 1] && seats[i][j + 2]) {
        // Ustawiamy wybór i aktualizujemy wygląd miejsca
        selSeat = i * seats[i].length + j;
        document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_select.png";
        document.getElementById("seat" + (i * seats[i].length + j)).alt = "Twoje miejsce";
        document.getElementById("seat" + (i * seats[i].length + j + 1)).src = "seat_select.png";
        document.getElementById("seat" + (i * seats[i].length + j + 1)).alt = "Twoje miejsce";
        document.getElementById("seat" + (i * seats[i].length + j + 2)).src = "seat_select.png";
        document.getElementById("seat" + (i * seats[i].length + j + 2)).alt = "Twoje miejsce";

        // Prosimy użytkownika o zaakceptowanie wyboru miejsca
        var accept = confirm("Miejsca od " + (j + 1) + " do " + (j + 3) +
          " w rzędzie " + (i + 1) + " są wolne. Wybrać je?");
        if (accept) {
          // Użytkownik zaakceptował miejsce — kończymy (wychodzimy z wewnętrznej pętli)
          finished = true;
          break;
        }
        else {
          // Użytkownik odrzucił wybór, zatem anulujemy wybór i szukamy dalej
          selSeat = -1;
          document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_avail.png";
          document.getElementById("seat" + (i * seats[i].length + j)).alt = "Miejsce dostępne";
          document.getElementById("seat" + (i * seats[i].length + j + 1)).src = "seat_avail.png";
          document.getElementById("seat" + (i * seats[i].length + j + 1)).alt = "Miejsce dostępne";
          document.getElementById("seat" + (i * seats[i].length + j + 2)).src = "seat_avail.png";
          document.getElementById("seat" + (i * seats[i].length + j + 2)).alt = "Miejsce dostępne";
        }
      }
    }

    // Inkrementujemy licznik zewnętrznej pętli
    i++;
  }
}
```

Ponieważ każdy z tych sześciu fragmentów kodu wykonuje dokładnie to samo zadanie, zatem można je zamienić w jedną funkcję.

Te fragmenty kodu powtarzają się. Możemy w nich wyróżnić pewne atrybuty...

Właściwość length wciąż jest używana do określania liczby elementów w tablicy wewnętrznej.

Tworzenie funkcji określającej status miejsc

Teraz, gdy autorzy aplikacji Mandango zobaczyli, co oznacza poprawianie efektywności, aż palą się do dodawania nowych funkcji, dzięki którym ich kod mógłby być jeszcze bardziej wydajny (kody wszystkich przykładów prezentowanych w książce znajdziesz na serwerze FTP wydawnictwa Helion, *ftp://ftp.helion.pl/przyklady/hfjsc.zip*). Jednak, by móc napisać funkcję `setSeat()`, Szymek i Jasiek muszą najpierw określić, jakie argumenty będą konieczne do jej działania. Argumenty te można podać, analizując powtarzające się fragmenty kodu i wskazując dane, które w każdym z nich mają inne wartości. Analiza powtarzających się fragmentów funkcji `findSeats()`, pozwala wyróżnić następujące argumenty:

Numer miejsca

Numer miejsca, którego status będzie ustawiany. Nie jest to jednak indeks tabeli, lecz numer liczony od lewej do prawej i z góry w dół, zaczynając od 0.

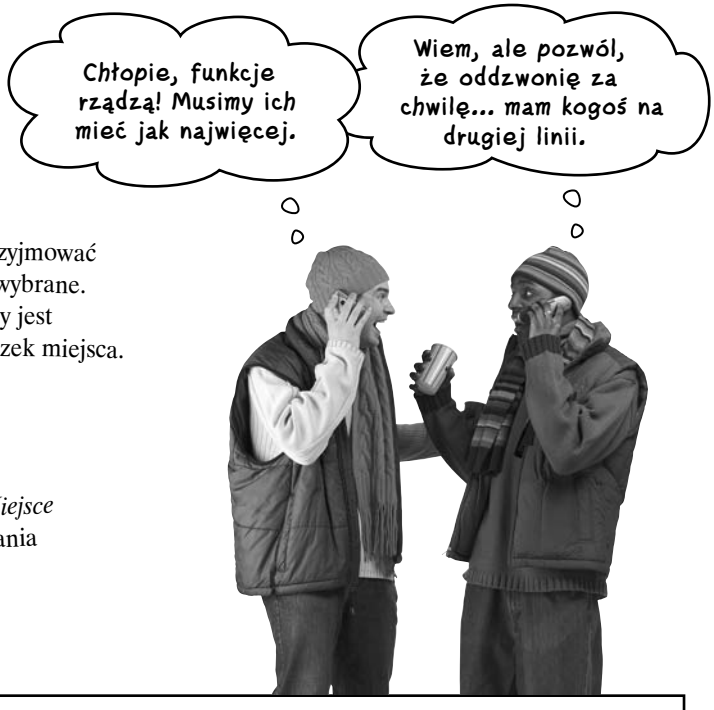
Atrybuty funkcji findSeats() zostały wskazane poprzez analizę powtarzającego się kodu, który niebawem umieścimy w nowej funkcji.

Status

Status miejsca. Może on przyjmować wartości: dostępne, zajęte, wybrane. Na jego podstawie określany jest wyświetlany na stronie obrazek miejsca.

Opis

Opis aktualnego statusu miejsca. Może przyjmować wartości: *Miejsce dostępne*, *Miejsce zajęte* oraz *Twoje miejsce*. Służy do określania wartości atrybutu `alt` elementu obrazka reprezentującego dane miejsce.



Zaostrz ołówek

Napisz kod funkcji `setSeat()` dla aplikacji Mandango.

.....

.....

.....

.....

Zaostrz ołówek

Napisz kod funkcji `setSeat()` dla aplikacji Mandango.

Konkretne dane zastosowane w oryginalnym kodzie zostały teraz zastąpione ogólnymi argumentami.

Poszczególne argumenty zostały od siebie oddzielone przecinkami.

```
function setSeat(seatNum, status, description) {
  document.getElementById("seat" + seatNum).src = "seat_" + status  + ".png";
  document.getElementById("seat" + seatNum).alt = description ;
}
```

Bardziej elegancki i przejrzysty kod dzięki zastosowaniu funkcji

Umieszczenie powtarzającego się kodu w funkcji `setSeat()` w znaczący sposób uprościło kod funkcji `findSeats()`. Obecnie w kodzie funkcji `findSeats()` zostało umieszczonych aż sześć wywołań funkcji `setSeat()`, co stanowi znaczącą poprawę, jeśli chodzi o powtarzanie i wielokrotne stosowanie kodu.

Nasza nowa funkcja setSeat() jest wywoływana w sześciu miejscach.

Numer miejsca, status oraz opis są przekazywane w każdym wywołaniu funkcji setSeat().

```
function findSeats() {
  ...
  // Przeglądamy wszystkie miejsca, poszukując dostępnych
  var i = 0, finished = false;
  while (i < seats.length && !finished) {
    for (var j = 0; j < seats[i].length; j++) {
      // Sprawdzamy, czy aktualnie analizowane miejsce i dwa następne są dostępne
      if (seats[i][j] && seats[i][j + 1] && seats[i][j + 2]) {
        // Ustawiamy wybór i aktualizujemy wygląd miejsca
        selSeat = i * seats[i].length + j;
        setSeat(i * seats[i].length + j, "select", "Twoje miejsce");
        setSeat(i * seats[i].length + j + 1, "select", "Twoje miejsce");
        setSeat(i * seats[i].length + j + 2, "select", "Twoje miejsce");

        // Prosimy użytkownika o zaakceptowanie wyboru miejsca
        var accept = confirm("Miejsca od " + (j + 1) + " do " + (j + 3) +
          " w rzędzie " + (i + 1) + " są wolne. Wybrać je?");
        if (accept) {
          // Użytkownik zaakceptował miejsce – kończymy (wychodzimy z wewnętrznej pętli)
          finished = true;
          break;
        }
        else {
          // Użytkownik odrzucił wybór, zatem anulujemy wybór i szukamy dalej
          selSeat = -1;
          setSeat(i * seats[i].length + j, "avail", "Miejsce dostępne");
          setSeat(i * seats[i].length + j + 1, "avail", "Miejsce dostępne");
          setSeat(i * seats[i].length + j + 2, "avail", "Miejsce dostępne");
        }
      }
    }

    // Inkrementujemy licznik zewnętrznej pętli
    i++;
  }
}
```

Funkcja setSeat() jeszcze bardziej poprawia kod aplikacji Mandango

Jednak nie tylko funkcja `findSeats()` skorzystała na zaimplementowaniu naszej nowej funkcji `setSeat()`. Poprawiona została także efektywność działania funkcji `initSeats()`, gdyż także w niej znajdował się kod określający status miejsc.

```
function initSeats() {
  // Inicjalizacja wyglądu wszystkich miejsc
  for (var i = 0; i < seats.length; i++) {
    for (var j = 0; j < seats[i].length; j++) {
      if (seats[i][j]) {
        // Określamy dostępne miejsce
        document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_avail.png";
        document.getElementById("seat" + (i * seats[i].length + j)).alt = "Miejsce dostępne";
      }
      else {
        // Określamy zajęte (niedostępne) miejsce
        document.getElementById("seat" + (i * seats[i].length + j)).src = "seat_unavail.png";
        document.getElementById("seat" + (i * seats[i].length + j)).alt = "Miejsce zajęte";
      }
    }
  }
}
```

Dwa całkiem złożone wiersze kodu zostały zastąpione jednym stosunkowo prostym wywołaniem funkcji.

```
function initSeats() {
  // Inicjalizacja wyglądu wszystkich miejsc
  for (var i = 0; i < seats.length; i++) {
    for (var j = 0; j < seats[i].length; j++) {
      if (seats[i][j]) {
        // Określamy dostępne miejsce
        setSeat(i * seats[i].length + j, "avail", "Miejsce dostępne");
      }
      else {
        // Określamy zajęte (niedostępne) miejsce
        setSeat(i * seats[i].length + j, "unavail", "Miejsce zajęte");
      }
    }
  }
}
```

Poprzez uogólnienie operacji określenia statusu miejsca funkcja setSeat() doskonale się spisuje w bardzo różnych sytuacjach

Jak zatem widać, stosunkowo prosta funkcja składająca się z dwóch wierszy kodu jest obecnie wywoływana aż w ośmiu miejscach kodu aplikacji Mandango. Nie tylko **upraszcza** to kod skryptu, lecz także znacznie ułatwia jego utrzymanie, gdyż gdybyś w przyszłości musiał zmodyfikować sposób zmiany statusu miejsc, wystarczy zmienić go raz — w kodzie funkcji `setSeat()` — a nie w ośmiu różnych miejscach całego skryptu. Żaden programista o zdrowych zmysłach nie chciałby zmieniać kodu w ośmiu miejscach, jeśli mógłby tego uniknąć. Łatwość utrzymania… fajna rzecz.

KLUCZOWE ZAGADNIENIA

- Funkcje pozwalają nam dzielić **duże problemy na małe**, dzięki czemu można je łatwiej rozwiązać.
- Funkcje zapewniają mechanizmy pozwalające wydzielać zadania i implementować je w postaci niezależnych fragmentów kodu **nadających się do wielokrotnego stosowania.**
- Funkcje są doskonałym sposobem **eliminacji powtarzającego się kodu**, gdyż kod umieszczony wewnątrz nich może być używany dowolnie wiele razy.
- **Argumenty** pozwalają przekazywać do funkcji informacje stanowiące **dane wejściowe** dla realizowanego przez nią zadania.

Nie ma niemądrych pytań

P: Czy istnieje jakieś ograniczenie liczby argumentów przekazywanych do funkcji?

O: I tak, i nie. Nie — za wyjątkiem wielkości pamięci operacyjnej komputera nie istnieje żadne rzeczywiste ograniczenie liczby argumentów, jakie można przekazywać do funkcji. Jeśli jednak funkcja wymaga tylu argumentów, że wielkość pamięci komputera zaczyna być problemem, to powinieneś chyba trochę odpocząć i przemyśleć dokładnie to, co robisz, gdyż liczba argumentów musi być ogromna. W praktyce ograniczenia liczby argumentów funkcji są raczej związane z projektowaniem kodu — chodzi o to, by liczba argumentów była sensowna i aby stosowanie funkcji nie było koszmarnie złożone. Ogólnie rzecz biorąc, zaleca się, by liczba argumentów wywołania nie przekraczała kilku.

P: Dowiedziałem się, że funkcje zamieniają duże problemy na małe, pozwalają dzielić pracę nad skryptem i eliminować powtarzający się kod. A co jest najważniejsze?

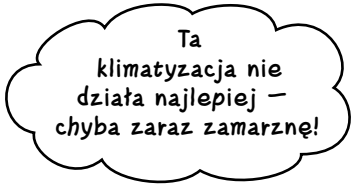
O: Wszystko jest równie ważne. Funkcje mogą być pożyteczne i przydatne pod wieloma względami, a w wielu przypadkach dobre funkcje pozwalają osiągnąć wiele celów. Nic zatem nie stoi na przeszkodzie, by napisać funkcję, która jednocześnie rozwiązuje podproblem, zapewnia podział zadania na części i eliminuje powielanie kodu. W praktyce są to trzy podstawowe cele, dla których tworzymy funkcje. Jeśli jednak musisz się skoncentrować na jednym aspekcie funkcji, to zapewne będzie to podział pracy, który w praktyce oznacza, że każda funkcja ma jedno ściśle określone przeznaczenie. Jeśli każda funkcja będzie się koncentrować na rozwiązaniu jednego zadania, to Twój skrypt bardzo na tym skorzysta.

P: Jeszcze raz spytam: czy funkcje są umieszczane w sekcji nagłówka, czy w treści stron WWW?

O: Funkcje należy umieszczać wewnątrz znaczników `<script>`, w nagłówku strony, bądź też w niezależnych plikach JavaScript, które następnie zostaną zaimportowane w nagłówku strony.

P: Gdybym naprawdę chciał, żeby funkcja zmieniała wartość argumentu, to w jaki sposób mogę to zrobić?

O: Nie ma bezpośredniej możliwości modyfikowania wartości argumentów funkcji, tak by modyfikacje te zostały zauważone poza funkcją. Jeśli zatem chcesz zmienić pewną informację przekazaną do funkcji jako argument jej wywołania, to musisz zwrócić tę informację jako wynik działania funkcji. Czytaj dalej, a dowiesz się, jak to zrobić!

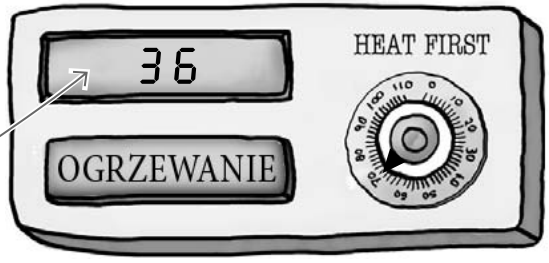


Zima w lipcu — informacje zwrotne z funkcji

Choć aplikacja Mandango wiele zyskała dzięki zastosowaniu funkcji, to jednak na froncie kontroli temperatury otoczenia funkcje nie radzą sobie już aż tak dobrze. Wygląda na to, że klimatyzator, którego kod został napisany w JavaScripcie, wciąż nie działa prawidłowo, a niektórzy użytkownicy powoli zamarzają, marząc o tym, by naciśnięcie przycisku *Ogrzewanie* powodowało ciągłe i nieprzerwane grzanie.

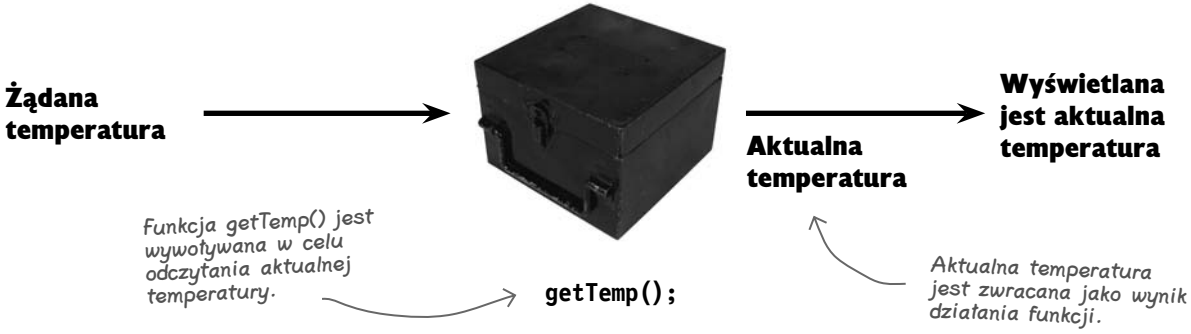
Znaczenie informacji zwrotnych

Dzięki zastosowaniu argumentów funkcji nasz klimatyzator w swojej aktualnej postaci pozwala na ustawienie temperatury, jednak nie podaje aktualnej, choć jest ona dla nas ważna, gdyż stanowi punkt odniesienia, względem którego nastawiamy temperaturę docelową. Poza tym zdarza się, że różne klimatyzatory podają różne temperatury i to nawet jeśli znajdą się w tym samym miejscu. Wszystko to prowadzi nas do jednego wniosku — konieczne są jakieś informacje zwrotne… musimy znać aktualną temperaturę, abyśmy mogli ustawić sensowną temperaturę docelową.



Wyświetlacz „Aktualna temperatura" pozwala użytkownikom zorientować się, jaka jest aktualna temperatura otoczenia, dzięki czemu są oni w stanie dokładniej określić temperaturę docelową.

Teraz nasz klimatyzator co pewien czas wyświetla aktualną temperaturę, pomagając tym samym użytkownikom nastawić optymalną temperaturę, jaką chcieliby uzyskać.



A zatem naprawdę będziemy potrzebowali jakiegoś sposobu, który zapewni funkcji możliwość zwracania informacji do kodu, który tę funkcję wywołał.

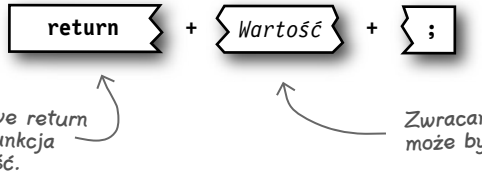
WYTĘŻ UMYSŁ

Jak sądzisz, w jaki sposób można nakłonić funkcję do zwrócenia jakichś danych?

Zwracanie danych z funkcji

Zwracanie danych z funkcji wymaga zastosowania słowa kluczowego `return` wraz z umieszczonymi za nim informacjami, które chcemy zwrócić. Informacje te zostaną przekazane do kodu, który wywołał funkcję.

Wartość wynikowa pozwala zwracać dane z funkcji do kodu, który ją wywołał.



Instrukcję `return` można umieścić w dowolnym miejscu funkcji; musisz jednak pamiętać, że jej wykonanie spowoduje natychmiastowe zakończenie działania funkcji. A zatem instrukcja ta powoduje nie tylko zwrócenie wartości wynikowej, lecz także zakończenie wykonywania funkcji. Na przykład funkcja `getTemp()` kończy działanie, zwracając wartość aktualnej temperatury otoczenia:

```
function getTemp() {
  // Odczyt i konwersja aktualnej temperatury
  var rawTemp = readSensor();
  var actualTemp = convertTemp(rawTemp);
  return actualTemp;
}
```

Dane odczytywane z czujnika temperatury są podawane w dziwacznym formacie, który należy przekształcić na wartość wyrażoną w stopniach.

Aktualna temperatura jest zwracana przy użyciu instrukcji return jako wynik wywołania funkcji.

Jeśli jesteś uważny, to zapewne pamiętasz, że funkcja `getTemp()` została już wcześniej zastosowana w kodzie obsługującym działanie klimatyzatora:

```
function heat(targetTemp) {
  while (getTemp() < targetTemp) {
    // W jakiś sposób zaczynamy ogrzewać
    ...
  }
}
```

Funkcja getTemp() zwraca wartość, która jest używana w warunku sterującym działaniem pętli w funkcji heat().

Wartość wynikowa funkcji zastępuje jej wywołanie.

Wartość wynikowa funkcji `getTemp()` **zastępuje** wywołanie tej funkcji i staje się elementem warunku logicznego, sprawdzanego w pętli `while`.

Wiele szczęśliwych wartości wynikowych

Ponieważ instrukcja `return` powoduje natychmiastowe przerwanie wykonywania funkcji, zatem można jej używać nie tylko do zwracania wartości, lecz także do sterowania działaniem funkcji. Nie jest to jedyne zastosowanie wartości wynikowych, jednak funkcje bardzo często używają ich, by informować kod wywołujący o pomyślnym zakończeniu działania bądź o zaistniałych problemach. Nasza przykładowa funkcja `heat()` stwarza okazję do przedstawienia tego sposobu wykorzystania instrukcji `return`.

```
function heat(targetTemp) {
  if (getTemp() >= targetTemp)
    return false;
  while (getTemp() < targetTemp) {
    // W jakiś sposób zaczynamy ogrzewać
    ...
  }
  return true;
}
```

Czy pamiętasz zmienną actualTemp? To właśnie jej wartość jest zwracana przez funkcję getTemp().

Nie musimy włączać ogrzewania; dlatego też zwracamy wartość false i kończymy działanie funkcji.

To właśnie kod umieszczony w tym miejscu odpowiada za ogrzewanie, co z kolei ma wpływ na temperaturę otoczenia, a co za tym idzie — także na wartość wynikową, zwracaną przez funkcję getTemp().

Ogrzewanie zakończone, tym razem zwracamy wartość true, by poinformować o pomyślnym wykonaniu zadania.

Funkcja `heat()` demonstruje, w jaki sposób logiczna wartość wynikowa może kontrolować przebieg działania funkcji, a przy okazji informować o jej pomyślnym bądź nieudanym wykonaniu. Wyłącznie do celów sterowania działaniem funkcji wystarczy użyć samej instrukcji `return`, bez żadnej wartości wynikowej — w takim przypadku instrukcja ta stanowiłaby sposób zakończenia wywołania funkcji. Oto kolejna wersja funkcji `heat()`, która nie używa wartości wynikowych do przekazania informacji o pomyślnym wykonaniu.

```
function heat(targetTemp) {
  if (getTemp() >= targetTemp)
    return;
  while (getTemp() < targetTemp) {
    // W jakiś sposób zaczynamy ogrzewać
    ...
  }
}
```

Ta instrukcja return przerywa działanie funkcji, gdyż ogrzewanie nie jest konieczne.

Funkcja nawet bez pomocy instrukcji return kończy się tam, gdzie powinna.

Instrukcji return można używać do przerywania działania funkcji nawet bez podawania wartości wynikowej.

Instrukcja return bez tajemnic

Temat tego wywiadu:
Sekrety specjalisty od kończenia funkcji

Head First: Witaj. Słyszałem, że jesteś doprawdy nieobliczalna — zdolna do przerwania każdej funkcji i wyjścia z niej.

Return: To prawda. Umieść mnie w dowolnej funkcji, a ja błyskawicznie zakończę jej działanie. Co więcej, mogę nawet zabrać ze sobą jakieś dane.

Head First: A dokąd się udajesz, kończąc funkcję?

Return: No cóż… Nie zapominaj, że każda funkcja jest wywoływana z jakiegoś innego miejsca kodu; a zatem zakończenie wykonywania funkcji oznacza powrót do kodu, który ją wywołał. Podobnie rzecz się ma w przypadku zwracania wartości wynikowej — jest ona przekazywana do kodu, który wywołał funkcję.

Head First: Ale jak to działa?

Return: Najlepiej będzie, jeśli wyobrazisz sobie funkcję jako wyrażenie, które zwraca wartość. Jeśli funkcja nie zwraca wartości, to wyrażenie nie ma żadnego wyniku. Jeśli jednak funkcja zwróci wartość, a naprawdę wiele z nich to robi, to wartość ta staje się wynikiem wyrażenia.

Head First: A zatem, skoro funkcja jest wyrażeniem, to czy można jej wartość wynikową zapisać w jakiejś zmiennej?

Return: I tak, i nie. Widzisz, sama funkcja nie jest wyrażeniem — jest nim wywołanie funkcji, co nie zmienia faktu, że możesz, a czasami nawet powinieneś umieścić wywołanie funkcji w takim miejscu, by zwrócona przez nie wartość wynikowa została zapisana w zmiennej. To właśnie w tym miejscu na scenę wkraczają wyrażenia — w momencie przetwarzania wywołania funkcji jest ono traktowane jako wyrażenie, którego wynikiem jest wartość zwrócona przez funkcję.

Head First: Rozumiem. Ale co się dzieje z tym wyrażeniem, kiedy funkcja niczego nie zwraca?

Return: Jeśli użyjesz mnie bez podawania żadnej wartości, to funkcja niczego nie zwróci, a wyrażenie będzie puste.

Head First: A czy to nie będzie stanowiło problemu?

Return: Nie, nie przypuszczam. Powinieneś pamiętać, że programiści zaprzątają sobie głowę wykorzystaniem wartości wynikowej wyłącznie wtedy, gdy wiedzą, że dana funkcja może coś zwrócić. Jeśli funkcja z założenia nie ma zwracać żadnych informacji wynikowych, to nie powinieneś przejmować się jej wartością wynikową ani starać się jej używać.

Head First: No jasne! Wróćmy zatem do twoich umiejętności przerywania funkcji. Czy nie sądzisz, że takie gwałtowne przerywanie funkcji, zanim sama dotrze do swego naturalnego końca, jest złe i niewłaściwe?

Return: Otóż nie, i zaraz ci wyjaśnię dlaczego. To, że funkcja ma swój pierwszy i ostatni wiersz, wcale nie oznacza, że została zaprojektowana w taki sposób, by każdy wiersz jej kodu był zawsze wykonywany. W rzeczywistości myśl, że funkcja ma swój początek i koniec, może być nawet niebezpieczna. „Naturalny" koniec funkcji może wypadać w środku jej kodu, jeśli tylko jakiś zdolny programista mnie tam umieści.

Head First: Nie rozumiem. Czy chcesz mi powiedzieć, że w niektórych funkcjach naturalne jest to, że pewne fragmenty kodu nigdy nie zostaną wykonane?

Return: Nigdy nie mówię nigdy, mogę cię natomiast zapewnić, że w wielu przypadkach działanie funkcji może przebiegać kilkoma różnymi ścieżkami… a ja często pomagam w ich tworzeniu. Oczywiście, może się także zdarzyć, że funkcja będzie wykonywana „od deski do deski", czyli od pierwszego do ostatniego wiersza kodu, i nigdy o mnie nawet nie usłyszy albo użyje mnie na końcu do zwrócenia wartości wynikowej.

Head First: Hm… rozumiem. A zatem dajesz funkcjom możliwości i to zarówno jeśli chodzi o zwracanie wyników, jak i kontrolę działania.

Return: Brawo, w końcu coś ci zaczęło świtać.

Head First: No tak… lepiej późno niż wcale. Dzięki za rozmowę i poświęcony czas.

Return: Nie ma sprawy. Najwyższy czas, by to skończyć!

Ćwiczenie

Wygląda na to, że JavaScript został przyłapany w samym centrum skandalu klimatyzacyjnego. Osoby zrzeszone w ugrupowaniu Umiarkowani Krytycy Radykalnego Ocieplenia Powietrza, w skrócie: UKROP, napisały skrypt głoszący ich ostrzeżenie dotyczące lokalnego ocieplenia. Jednak członkom ZAMIEC-i, czyli Zrzeszenia Anonimowych Miłośników Inteligentnej Emisji Ciepła — zagorzałym przeciwnikom idei URKOP-u — udało się dokonać sabotażu kodu tego skryptu. Twoim zadaniem jest wskazanie prawidłowych i sabotowanych fragmentów kodu oraz ujawnienie prawdziwych postulatów UKROP-u.

```
function showClimatMsg() {
  return;
  alert(constructMessage());
}

function constructMessage() {
  var msg = "";

  msg += "Globalne "; // "Lokalne "

  if (getTemp() > 27)
    msg += "ocieplenie ";
  else
    msg += "ochłodzenie ";

  if (true)
    msg += "nie jest ";
  else
    msg = "jest ";

  if (getTemp() <= 21)
    return msg + "bujdą!";
  else
    return msg + "faktem!";

  return "Ja w to nie wierzę.";
}

function getTemp() {
  // Odczyt i konwersja aktualnej temperatury
  var rawTemp = readSensor();
  var actualTemp = convertTemp(rawTemp);
  return 18;
}
```

Mitośniczka UKROP-u.

Aktywista ZAMIEC-i.

Ćwiczenie Rozwiązanie

Wygląda na to, że JavaScript został przyłapany w samym centrum skandalu klimatyzacyjnego. Osoby zrzeszone w ugrupowaniu Umiarkowani Krytycy Radykalnego Ocieplenia Powietrza, w skrócie: UKROP, napisały skrypt głoszący ich ostrzeżenie dotyczące lokalnego ocieplenia. Jednak członkom ZAMIEC-i, czyli Zrzeszenia Anonimowych Miłośników Inteligentnej Emisji Ciepła — zagorzałym przeciwnikom idei URKOP-u — udało się dokonać sabotażu kodu tego skryptu. Twoim zadaniem jest wskazanie prawidłowych i sabotowanych fragmentów kodu oraz ujawnienie prawdziwych postulatów UKROP-u.

Ta instrukcja skutecznie uniemożliwia wyświetlenie jakiegokolwiek komunikatu.

Tekst, który miał być wyświetlany, został umieszczony w komentarzu, by nie pojawiał się w ostrzeżeniu.

Ten kod jest w porządku, gdyż komunikat zależy od aktualnej temperatury.

Instrukcja if, której warunek jest zawsze spełniony, nie ma większego sensu.

Instrukcja if/else skutecznie uniemożliwi dotarcie do tego miejsca kodu, a zatem ta instrukcja return jest bezcelowa.

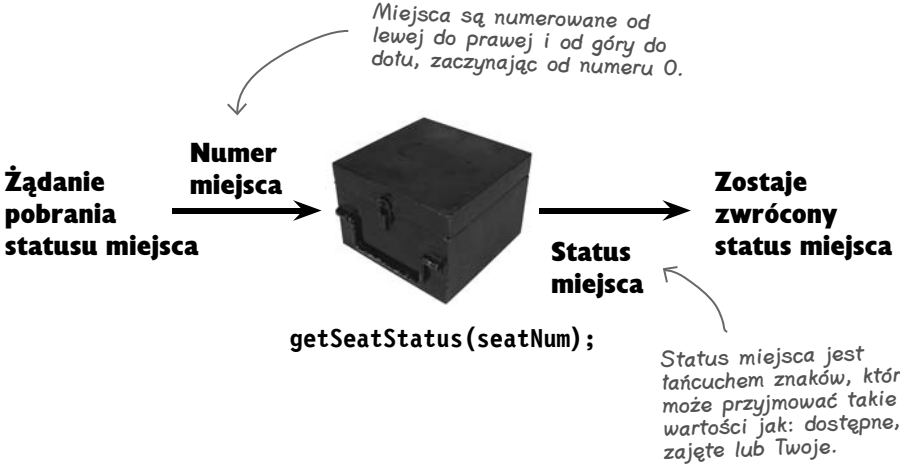
Musimy zwrócić faktyczną temperaturę zmierzoną przez czujnik.

```
function showClimatMsg() {
  return;
  alert(constructMessage());
}

function constructMessage() {
  var msg = "";

  msg += "Globalne "; // "Lokalne "

  if (getTemp() > 27)
    msg += "ocieplenie ";
  else
    msg += "ochłodzenie ";

  if (true)
    msg += "nie jest ";
  else
    msg = "jest ";

  if (getTemp() <= 21)
    return msg + "bujdą!";
  else
    return msg + "faktem!";

  return "Ja w to nie wierzę.";
}

function getTemp() {
  // Odczyt i konwersja aktualnej temperatury
  var rawTemp = readSensor();
  var actualTemp = convertTemp(rawTemp);
  return 18; actualTemp
}
```

Lokalne ocieplenie jest faktem!

Odczyt statusu miejsca

W siedzibie Mandango Szymek i Jasiek mają już powyżej uszu słuchania o zmianach klimatycznych i są gotowi do wprowadzenia pewnych poprawek do swojego skryptu. Niektórzy użytkownicy zgłaszali problemy z rozróżnianiem kolorów siedzeń i chcieliby mieć możliwość kliknięcia dowolnego miejsca w celu sprawdzenia jego dostępności. Wygląda na to, że w kodzie aplikacji Mandango powinna pojawić się nowa funkcja.



Magnesiki z JavaScriptem

W kodzie funkcji `getSeatStatus()` brakuje kilku ważnych elementów, które są jej niezbędne do określenia statusu konkretnego miejsca. W pierwszej kolejności, funkcja sprawdza, czy dane miejsce należy do sekwencji trzech, aktualnie wybranych miejsc. Jeśli nie należy, to funkcja sprawdza w tablicy miejsc czy dane miejsce jest dostępne czy nie. Uzupełnij brakujące miejsca kodu magnesikami widocznymi u dołu strony.

```
function getSeatStatus(seatNum) {
  if (..... != -1 && (..... == ..... || ..... == (..... + 1) || ..... == (..... + 2)))
    return "Twoje";
  else if (..... [Math.floor(..... / ..... [0].length)][ ..... % ..... [0].length])
    return "dostępne";
  else
    return "zajęte";
}
```

seats

selSeat

seatNum

Magnesiki z JavaScriptem

W kodzie funkcji `getSeatStatus()` brakuje kilku ważnych elementów, które są jej niezbędne do określenia statusu konkretnego miejsca. W pierwszej kolejności, funkcja sprawdza, czy dane miejsce należy do sekwencji trzech, aktualnie wybranych miejsc. Jeśli nie należy, to funkcja sprawdza w tablicy miejsc czy dane miejsce jest dostępne czy nie. Uzupełnij brakujące miejsca kodu magnesikami widocznymi u dołu strony.

Kiedy żadne miejsce nie jest wybrane, zmienna globalna selSeat ma wartość –1; ten warunek sprawdzamy w pierwszej kolejności.

Mamy do czynienia z sekwencją trzech miejsc położonych obok siebie, zatem musimy sprawdzić to miejsce i dwa następne.

```
function getSeatStatus(seatNum) {
  if (selSeat != -1 &&
    (seatNum == selSeat || seatNum == (selSeat + 1) || seatNum == (selSeat + 2)))
    return "Twoje";
  else if (seats[Math.floor(seatNum / seats[0].length)][seatNum % seats[0].length])
    return "dostępne";
  else
    return "zajęte";
}
```

Określamy wiersz w tablicy, w jakim należy szukać informacji o dostępności miejsca, dzieląc jego numer przez liczbę miejsc w rzędzie i zaokrąglając uzyskaną wartość do liczby całkowitej.

Określamy kolumnę miejsca poprzez obliczenie reszty z dzielenia numeru miejsca przez liczbę miejsc w rzędzie.

W tym miejscu można by podać na stałe wartość 9, jednak w takim przypadku, gdybyśmy zmienili liczbę miejsc w rzędzie, skrypt przestałby działać prawidłowo.

Prezentacja statusu miejsca

Pobieranie statusu miejsca jest bardzo wygodne, jednak udostępnienie użytkownikom możliwości sprawdzania statusu dowolnie wybranego miejsca wymaga narzędzia, które pozwoli nam wyświetlić status kliknięcia miejsca. Kolejna funkcja, `showSeatStatus()`, zapewnia proste rozwiązanie tego problemu, sprytnie przekazując wykonanie najtrudniejszego zadania napisanej wcześniej funkcji `getSeatStatus()`.

Aby pobrać status miejsca, do funkcji getSeatStatus() musimy przekazać numer miejsca, które nas interesuje.

```
function showSeatStatus(seatNum) {
  alert("To miejsce jest " + getSeatStatus(seatNum) + ".");
}
```

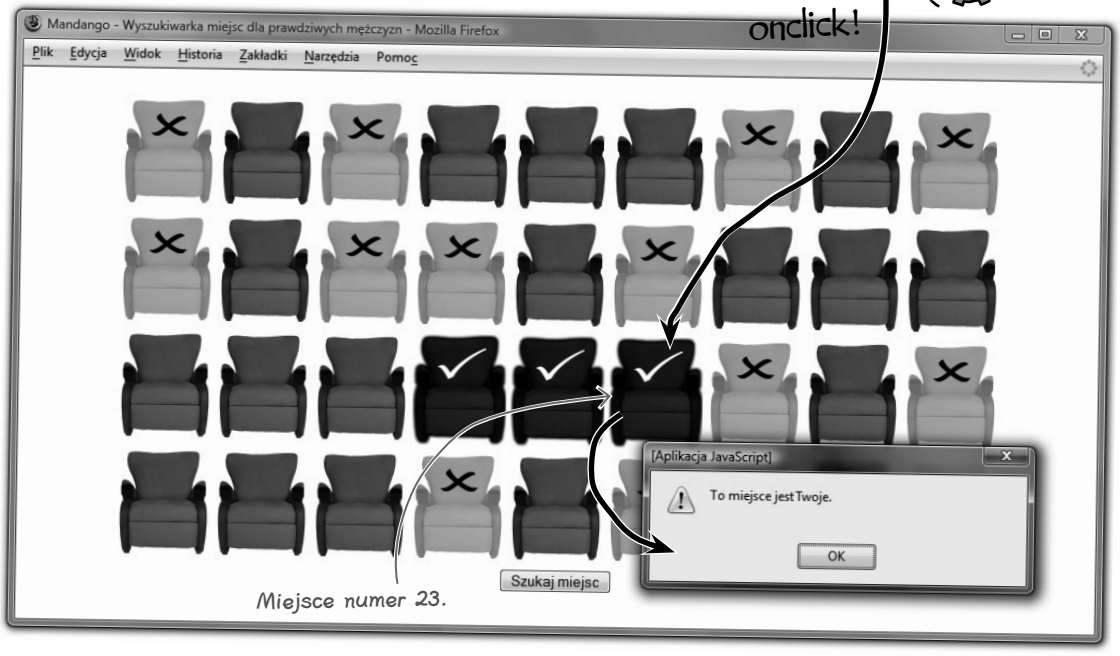
Łączymy łańcuchy znaków, by utworzyć komunikat informujący o statusie miejsca.

Możemy połączyć funkcję z obrazkiem

Skojarzenie tej funkcji z obrazkiem wyświetlanym na stronie Mandango pozwala użytkownikowi kliknąć dowolne miejsce, by sprawdzić jego status. W tym celu każdy obrazek miejsca wyświetlany na stronie musi mieć zdefiniowaną procedurę obsługi zdarzeń `onclick`, w której będzie wywoływana funkcja `showSeatStatus()`. Oto przykład takiej procedury obsługi zdarzeń:

Funkcja showSeatStatus() zostanie wywołana, gdy użytkownik kliknie obrazek "seat23".

```
<im id="seat23" src="" alt="" onclick="showSeatStatus(23);" />
```



Jedno kliknięcie — tyle wystarczy, by wyświetlić status dowolnego miejsca w informacyjnym okienku dialogowym. Na możliwości tej skorzystają wszyscy, którzy mają trudności z rozróżnieniem kolorów miejsc prezentowanych na stronie albo lubią klikać i w ten sposób sprawdzać statusy miejsc.

KLUCZOWE ZAGADNIENIA

- Polecenie `return` pozwala zwracać dane z funkcji do kodu, który ją wywołał.
- Kiedy pewna informacja jest zwracana jako wynik wykonania funkcji, to zastępuje ona jej wywołanie.
- Funkcja może zwrócić tylko jedną informację.
- Instrukcja `return` może być stosowana bez podawania żadnej danej wynikowej, w takim przypadku po prostu przerywa jej działanie.

Powielanie kodu nigdy nie jest dobre

Skrypt aplikacji Mandango działa całkiem dobrze, lecz jego autorzy zaczynają się powoli martwić o utrzymanie go w dłuższej perspektywie. Mówiąc konkretnie, Jasiek przeprowadził pewne badania i dowiedział się, że nowoczesne aplikacje internetowe często mogą wiele zyskać na separacji kodu HTML, JavaScript i CSS.

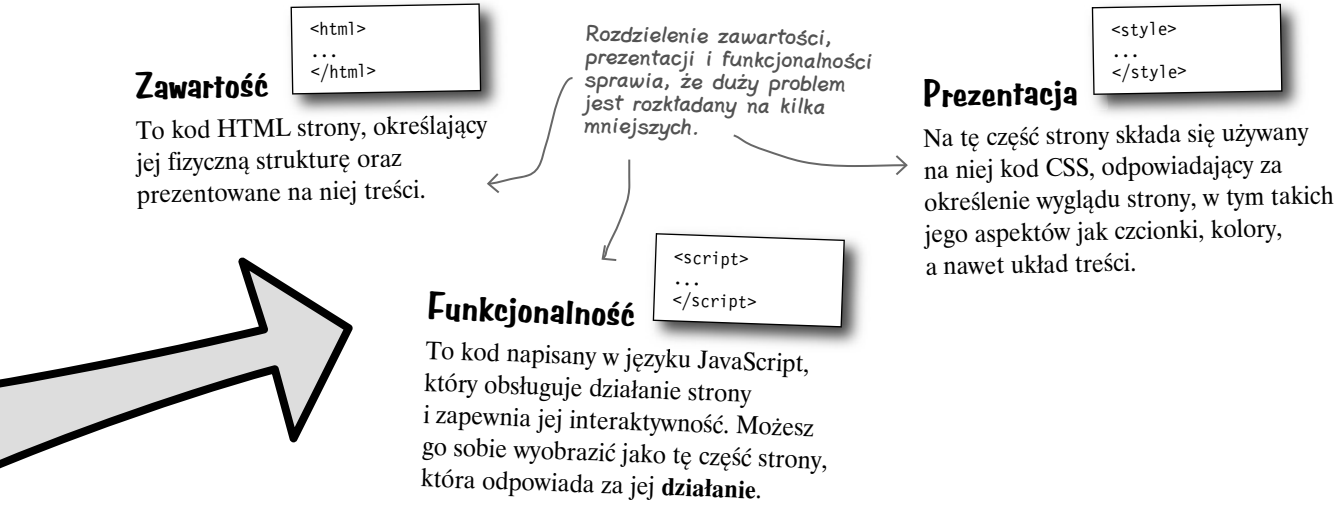

```
<html>
  <head>
    <title>Mandango - Wyszukiwarka miejsc dla prawdziwych mężczyzn</title>

    <script type="text/javascript">
      ...
      function initSeats() {
        ...
      }

      function getSeatStatus(seatNum) {
         ...
      }

      function showSeatStatus(seatNum) {
        alert("To miejsce jest " + getSeatStatus(seatNum) + ".");
      }

      function setSeat(seatNum, status, description) {
        document.getElementById("seat" + seatNum).src = "seat_" + status + ".png";
        document.getElementById("seat" + seatNum).alt = description;
      }

      function findSeats() {
        ...
      }
    </script>
  </head>

  <body onload="initSeats();">
    <div style="margin-top:25px; text-align:center">
      <img id="seat0" src="" alt="" onclick="showSeatStatus(0);" />
      <img id="seat1" src="" alt="" onclick="showSeatStatus(1);" />
      <img id="seat2" src="" alt="" onclick="showSeatStatus(2);" />
      <img id="seat3" src="" alt="" onclick="showSeatStatus(3);" />
      <img id="seat4" src="" alt="" onclick="showSeatStatus(4);" />
      <img id="seat5" src="" alt="" onclick="showSeatStatus(5);" />
      <img id="seat6" src="" alt="" onclick="showSeatStatus(6);" />
      <img id="seat7" src="" alt="" onclick="showSeatStatus(7);" />
      <img id="seat8" src="" alt="" onclick="showSeatStatus(8);" /><br />
      <img id="seat9" src="" alt="" onclick="showSeatStatus(9);" />
      <img id="seat10" src="" alt="" onclick="showSeatStatus(10);" />
      <img id="seat11" src="" alt="" onclick="showSeatStatus(11);" />
      <img id="seat12" src="" alt="" onclick="showSeatStatus(12);" />
      <img id="seat13" src="" alt="" onclick="showSeatStatus(13);" />
      <img id="seat14" src="" alt="" onclick="showSeatStatus(14);" />
      <img id="seat15" src="" alt="" onclick="showSeatStatus(15);" />
      <img id="seat16" src="" alt="" onclick="showSeatStatus(16);" />
      <img id="seat17" src="" alt="" onclick="showSeatStatus(17);" /><br />
      <img id="seat18" src="" alt="" onclick="showSeatStatus(18);" />
      <img id="seat19" src="" alt="" onclick="showSeatStatus(19);" />
      <img id="seat20" src="" alt="" onclick="showSeatStatus(20);" />
      <img id="seat21" src="" alt="" onclick="showSeatStatus(21);" />
      <img id="seat22" src="" alt="" onclick="showSeatStatus(22);" />
      <img id="seat23" src="" alt="" onclick="showSeatStatus(23);" />
      <img id="seat24" src="" alt="" onclick="showSeatStatus(24);" />
      <img id="seat25" src="" alt="" onclick="showSeatStatus(25);" />
      <img id="seat26" src="" alt="" onclick="showSeatStatus(26);" /><br />
      <img id="seat27" src="" alt="" onclick="showSeatStatus(27);" />
      <img id="seat28" src="" alt="" onclick="showSeatStatus(28);" />
      <img id="seat29" src="" alt="" onclick="showSeatStatus(29);" />
      <img id="seat30" src="" alt="" onclick="showSeatStatus(30);" />
      <img id="seat31" src="" alt="" onclick="showSeatStatus(31);" />
      <img id="seat32" src="" alt="" onclick="showSeatStatus(32);" />
      <img id="seat33" src="" alt="" onclick="showSeatStatus(33);" />
      <img id="seat34" src="" alt="" onclick="showSeatStatus(34);" />
      <img id="seat35" src="" alt="" onclick="showSeatStatus(35);" /><br />
      <input type="button" id="findseats" value="Szukaj miejsc" onclick="findSeats();" />
    </div>
  </body>
</html>
```

Kod JavaScript i HTML występujący wspólnie można izolować w atrybutach HTML obsługujących zdarzenia.

Na stronie aplikacji Mandango kod JavaScript przeplata się z kodem HTML.

Separacja funkcjonalności od zawartości

Ale o co cała ta afera z mieszaniem kodu? Przecież wszystko działa bez najmniejszych problemów, prawda? Problem ten wygląda jednak zupełnie inaczej, kiedy spojrzy się na dokumenty HTML wykorzystujące skrypty JavaScript nie jako zwyczajne strony WWW, lecz jako **aplikacje**. A podobnie jak wszystkie aplikacje, także te pisane w języku JavaScript wymagają starannego zaplanowania i zaprojektowania, zwłaszcza jeśli zależy nam na tym, by odniosły sukces w dłuższej perspektywie. Należy zauważyć, że dobre aplikacje zawierają mniej błędów i łatwiej je pielęgnować, kiedy ich zawartość, prezentacja i funkcjonalność są od siebie wyraźnie oddzielone. A jak widać, w aktualnej wersji aplikacji Mandango o jakiejkolwiek separacji nie może być mowy.



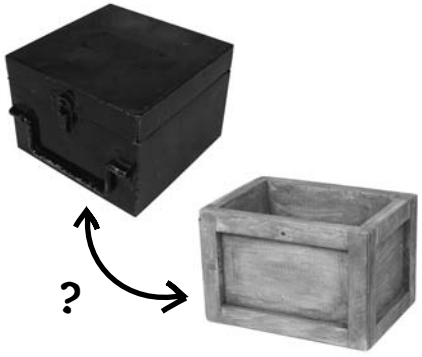
Zagadnienia związane z separacją kodu można wyobrazić sobie w następujący sposób. Załóżmy, że Jasiek i Szymek znaleźli naprawdę świetny skrypt do zarządzania miejscami w sali kinowej i chcieliby go zastosować zamiast własnego. W takim przypadku musieliby przebudować aplikację w taki sposób, by korzystała z kodu nowego skryptu, jednak tym samym naraziliby się na ryzyko uszkodzenia struktury strony, gdyż ich kod JavaScript jest ściśle zintegrowany z kodem HTML. Byłoby zatem znacznie lepiej, gdyby kod HTML został odseparowany, a powiązanie kodu JavaScript i HTML występowało wyłącznie w kodzie JavaScript.

Separacja funkcjonalności od zawartości ułatwia tworzenie aplikacji internetowych oraz ich późniejsze utrzymanie.

WYTĘŻ UMYSŁ

Czy masz jakiś pomysł na zastosowanie funkcji w celu odseparowania funkcjonalności od zawartości w aplikacji Mandango?

Funkcje są zwykłymi danymi



Abyś mógł efektywnie rozdzielić kod, musisz najpierw zrozumieć, w jaki sposób funkcje są powiązane ze zdarzeniami. Do tej pory kojarzenie to było realizowane przy użyciu atrybutów HTML. Istnieje jednak inny sposób, który przez wiele osób jest uważany za znacznie lepszy od mieszania kodu HTML i JavaScript. Jednak zastosowanie go wymaga zupełnie innego spojrzenia na funkcje.

Choć to zaskakujące, jednak w rzeczywistości funkcje są zwyczajnymi danymi. Tak, to prawda. Cała tajemnica polega na tym, że ciało funkcji jest wartością, a jej nazwa — zmienną. Poniżej przedstawiliśmy sposób prezentacji funkcji, który znasz i do którego jesteś przyzwyczajony:

Ta funkcja jest tworzona w doskonale znany, standardowy sposób.

```
function showSeatStatus(seatNum) {
  alert("To miejsce jest " + getSeatStatus(seatNum) + ".");
}
```

Ten kod działa bardzo dobrze, jednak tę samą funkcję można utworzyć także w inny sposób:

Nazwą funkcji jest w tym przypadku nazwa zmiennej.

```
var showSeatStatus  = function(seatNum) {
  alert("To miejsce jest " + getSeatStatus(seatNum) + ".");
} ;
```

Wartością zmiennej jest ciało funkcji, które jeśli zostanie wyrażone właśnie w taki sposób, jest także określane jako literał funkcyjny.

Powyższy przykład pokazuje, w jaki sposób można utworzyć funkcję, wykorzystując przy tym składnię charakterystyczną dla określania wartości zmiennych. Jak widać, używane są przy tym nawet te same elementy kodu: unikalny identyfikator (nazwa funkcji) oraz wartość (ciało funkcji). Kiedy ciało funkcji występuje samo, bez nazwy funkcji, nazywane jest **literałem funkcyjnym**.

Ta niesamowita informacja dotycząca funkcji jest tak interesująca, gdyż pokazuje, że funkcjami można posługiwać się podobnie jak zmiennymi. Jak sądzisz, co robi przedstawiony poniżej fragment kodu?

```
var myShowSeatStatus = showSeatStatus;
```

Zapisuje funkcję showSeatStatus() w zmiennej myShowSeatStatus.

Wywołania i odwołania do funkcji

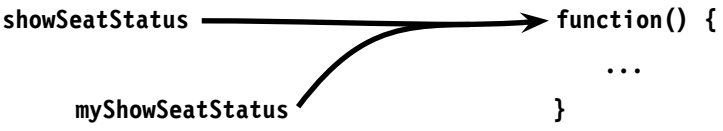
Kiedy przypiszesz nazwę funkcji do innej zmiennej, udzielisz jej dostępu do ciała tej funkcji. Innymi słowy, będziesz mógł wywołać tę funkcję, tak jak to pokazaliśmy na poniższym przykładzie:

```
alert(myShowSeatStatus(23));
```

Wywołanie tej samej funkcji showSeatStatus() przy użyciu zmiennej myShowSeatStatus.

Końcowy efekty wywołania `myShowSeatStatus()` jest identyczny jak wywołania `showSeatStatus()`, gdyż w rzeczywistości obie funkcje **odwołują** się do tego samego kodu. Właśnie z tego powodu nazwa funkcji jest także nazywana **odwołaniem** lub **referencją do funkcji**.

Tak naprawdę funkcja jest jedynie zmienną, której wartością jest odwołanie do ciała funkcji.



Rozróżnienie pomiędzy **odwoływaniem** się do funkcji a jej **wywoływaniem** odbywa się na podstawie pary nawiasów (`()`) umieszczanych za jej nazwą. W przypadku odwoływania się do funkcji nawiasy te nie występują, natomiast w przypadku wywoływania — są obowiązkowe. Co więcej, w przypadku wywoływania funkcji w nawiasach często pojawiają się dodatkowe argumenty.

Wywołanie funkcji myShowSeatStatus(), które w efekcie jest dokładnie tym samym co wywołanie funkcji showSeatStatus().

```
var myShowSeatStatus = showSeatStatus;
myShowSeatStatus(23);
```

Zapisanie w zmiennej myShowSeatStatus odwołania do funkcji.

Ćwiczenie

Przeanalizuj przedstawiony poniżej fragment kodu i zapisz liczbę, która zostanie wyświetlona w okienku dialogowym.

```
function doThis(num) {
  return num++;
}

function doThat(num) {
  return num--;
}
var x = doThis(11);
var y = doThat;
var z = doThat(x);
x =  y(z);
y = x;
alert(doThat(z – y));
```

Przeanalizuj przedstawiony poniżej fragment kodu i zapisz liczbę, która zostanie wyświetlona w okienku dialogowym.

Ćwiczenie Rozwiązanie

```
function doThis(num) {
  return num++;
}

function doThat(num) {
  return num--;
}
var x = doThis(11);
var y = doThat;
var z = doThat(x);
x =  y(z);
y = x;
alert(doThat(z – y));
```

```
x = 12
y = doThat
z = doThat(12) = 11
x = doThat(11) = 10
y = 10
alert(doThat(11 – 10))
```

Nie ma niemądrych pytań

P: Czy separacja zawartości jest naprawdę tak ważna, by trzeba było zaprzątać sobie nią głowę?

O: I tak, i nie. W przypadku niewielkich aplikacji umieszczanie w jednym pliku kodu HTML, JavaScript i CSS nie jest niczym niewłaściwym. Znaczenie i korzyści, jakie zapewnia separacja kodu, można znacznie łatwiej zauważyć dopiero w większych aplikacjach, w których tego kodu jest bardzo dużo. W takich dużych aplikacjach znacznie trudniej jest wypracować sobie bardziej ogólne pojęcie o działaniu kodu, zwłaszcza gdy różne rodzaje kodu są ze sobą wymieszane, a co za tym idzie — znacznie trudniej będzie modyfikować i pielęgnować takie aplikacje. Dzięki separacji kodu można czuć się bezpieczniej i mieć nadzieję, że zmiany funkcjonalne nie spowodują żadnych problemów w strukturze lub wyglądzie strony. Co więcej, dzięki separacji kodu nad tym samym projektem mogą pracować osoby o różnych specjalizacjach i doświadczeniach.

Na przykład projektant może pracować nad strukturą i prezentacją strony bez obaw, że jego zmiany spowodują pojawienie się błędów w funkcjonalnym kodzie JavaScript, którego projektant może nie rozumieć.

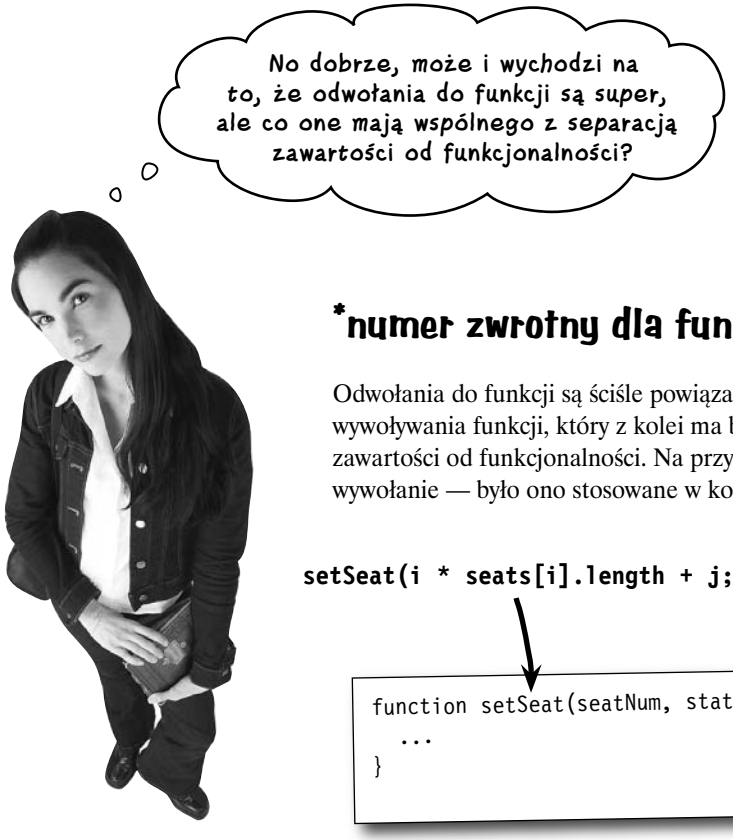
P: Skoro funkcje to tylko dane, to w jaki sposób mogę odróżnić funkcję od zwyczajnej zmiennej?

O: Różnica pomiędzy funkcją a „zwyczajną” zmienną sprowadza się do tego, w jakim celu i jak jej używasz. Dane powiązane z funkcją (czyli jej kod) można wykonywać. Jeśli masz zamiar wywołać funkcję, to informujesz o tym interpreter JavaScriptu, umieszczając za jej nazwą nawiasy oraz w razie potrzeby listę argumentów.

P: Jaki jest cel tworzenia odwołań do funkcji?

O: W odróżnieniu od normalnych zmiennych, które przechowują powiązane z nimi dane jako wartości w pamięci, funkcje przechowują odwołania do swego kodu. A zatem wartością zmiennej funkcyjnej nie jest sam kod funkcji, lecz odwołanie do miejsca w pamięci, w którym ten kod się znajduje. Zatem w przypadku funkcji zmienna przypomina nieco adres pocztowy, który wskazuje, o jaki dom chodzi, lecz nie jest tym domem.

Funkcje wykorzystują odwołania, a nie faktyczne wartości, gdyż rozwiązanie to jest bardziej efektywne niż tworzenie i przechowywanie wielu kopii kodu. A zatem przypisując funkcję procedurze obsługi zdarzeń (co niebawem) zrobisz, w rzeczywistości przypisujesz jedynie odwołanie do jej kodu, a nie sam kod.



*numer zwrotny dla funkcji

Odwołania do funkcji są ściśle powiązane ze specjalnym sposobem wywoływania funkcji, który z kolei ma bardzo dużo wspólnego z separacją zawartości od funkcjonalności. Na przykład spotkałeś już poniższe wywołanie — było ono stosowane w kodzie aplikacji Mandango:

```
setSeat(i * seats[i].length + j; "select", "Twoje miejsce");
```

```
function setSeat(seatNum, status, description) {
  ...
}
```

Jednak to nie jest jedyny możliwy sposób wywoływania funkcji. Inne funkcje, nazywane **funkcjami zwrotnymi**, mogą być wywoływane bez jakiegokolwiek jawnego działania ze strony programisty.

WYTĘŻ UMYSŁ

Jak sądzisz, czy takie funkcje zwrotne mogłyby się przydać w aplikacji Mandango? Jeśli tak, to w jaki sposób można by je wykorzystać?

Pogawędki przy kominku

Temat dzisiejszej pogawędki:
Konfrontacja Funkcji normalnej z Funkcją zwrotną

Funkcja normalna:

A zatem to ty jesteś tą „panienką z okienka", która nie przyjmuje wywołań lokalnych? Skąd to stanowisko?

Słucham… jak przeglądarka? To doprawdy egzotyczne podejście. Sądzę, że po prostu nie potrafisz się dogadać z tymi spośród nas, które w normalny sposób komunikują się z kodem skryptu.

Też coś… a czy kogoś to w ogóle obchodzi? Wcale! Kto się przejmuje tym, co dzieje się poza skryptem?

Wiesz… w tym przypadku może i masz trochę racji. Lubię wiedzieć, kiedy zostanie zakończone wczytywanie strony albo kiedy użytkownik coś kliknie czy wpisze. I twierdzisz, że gdyby nie ty, w ogóle bym nie wiedziała o takich zdarzeniach?

Cóż, miło usłyszeć, że w rzeczywistości wcale się tak bardzo nie różnimy.

Nie dzwoń do mnie, sama się odezwę.

Funkcja zwrotna:

Żadne tam stanowisko. Po prostu służę do innych celów. Podobnie jak przeglądarka WWW wolę, gdy wywołania przychodzą do mnie z odległych, egzotycznych miejsc.

Słuchaj… tu nie chodzi o to, która z nas jest lepsza lub gorsza. Wszystkie jesteśmy częściami kodu, ja po prostu zapewniam możliwość uzyskania dostępu do kodu przeróżnym outsiderom. Gdyby nie ja, nie miałabyś pojęcia o jakichkolwiek zdarzeniach zachodzących poza skryptem.

Prawdę mówiąc — wszyscy. Pamiętaj, że celem tworzenia skryptów jest poprawienie funkcjonalności i atrakcyjności stron. Gdyby skrypt nie posiadał żadnych sposobów wykrywania zachodzących poza nim zdarzeń i reagowania na nie, to trudno by było mówić o poprawianiu czegokolwiek.

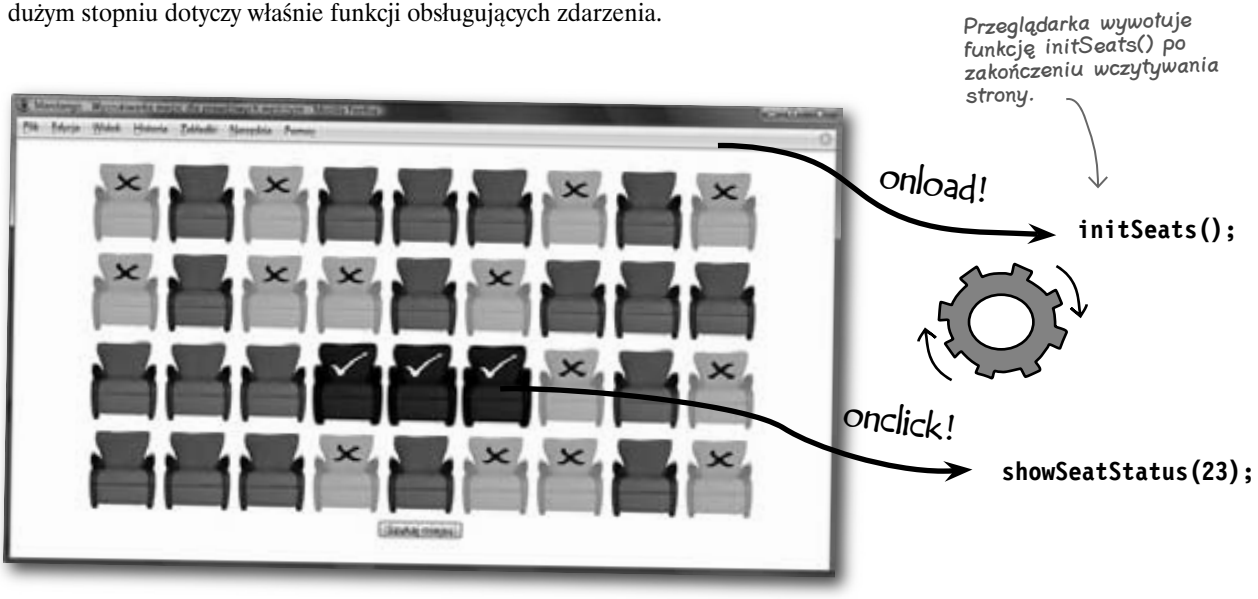
Właśnie. Przeglądarka wywołuje mnie, a ja w wielu przypadkach wywołuję ciebie, ponieważ reagowanie na zdarzenia zewnętrzne często wymaga wykonania kilku funkcji.

No właśnie. A zatem, jak sądzę, kiedyś się jeszcze spotkamy.

Ciekawe… Życzę powodzenia.

Zdarzenia, funkcje zwrotne i atrybuty HTML

Funkcje zwrotne, wywoływane przez przeglądarkę, a nie przez nasz kod, stosujemy już od dłuższego czasu. Najczęściej znajdują one zastosowanie podczas obsługi zdarzeń. Na przykład trudno byłoby wyobrazić sobie działanie aplikacji Mandango bez takich funkcji zwrotnych. W praktyce problem mieszania kodu HTML i JavaScript w bardzo dużym stopniu dotyczy właśnie funkcji obsługujących zdarzenia.



Funkcje zwrotne są kojarzone ze zdarzeniami w kodzie HTML aplikacji Mandango:

Atrybut onload języka HTML umożliwia skojarzenie funkcji initSeats() ze zdarzeniem onload.

```
<body onload="initSeats();">
<img id="seat26" src="" alt="" onclick="showSeatStatus(23);" />
```

Atrybut onclick języka HTML umożliwia skojarzenie funkcji showSeatStatus() ze zdarzeniem onclick.

Ta technika kojarzenia funkcji obsługujących zdarzenia ze zdarzeniami przy wykorzystaniu atrybutów HTML działa doskonale, jednak ma pewną wadę — wymaga mieszania kodu HTML i JavaScript. Na szczęście odwołania do funkcji pozwalają uniknąć tego niepotrzebnego bałaganu i odseparować oba rodzaje kodu.

Określanie procedur obsługi zdarzeń przy użyciu odwołań do funkcji

Zamiast kojarzyć funkcje zwrotne z procedurami obsługi zdarzeń, używając w tym celu atrybutów HTML, można to zrobić bezpośrednio w kodzie HTML. Innymi słowy, w ogóle nie będziesz musiał ingerować w kod HTML — wystarczy określić funkcję zwrotną, używając odwołania do funkcji, a wszystko wyłącznie z poziomu kodu JavaScript.

```
window.onload = initSeats;
```

Za nazwą funkcji nie ma nawiasów, gdyż nie chcemy wywoływać tej funkcji — interesuje nas odwołanie do niej.

Zdarzenie onload jest właściwością obiektu window.

Odwołanie do funkcji initSeats() zostaje zapisane we właściwości onload.

Jak zatem widać, określenie procedury obsługi zdarzeń z poziomu kodu JavaScript polega na zapisaniu odwołania do funkcji w odpowiedniej właściwości odpowiedniego obiektu. W powyższym przykładzie przypisanie odwołania do funkcji we właściwości `onload` sprawi, że każde zgłoszenie zdarzenia `onload` spowoduje wywołanie funkcji `initSeats()`. Co więcej, wywołanie funkcji następuje automatycznie w odpowiedzi na zgłoszenie zdarzenia. Oto, jak wygląda ten proces:

onload! → `window.onload();` → `initSeats();`

Zdarzenie onload wywołuje procedurę obsługi zdarzeń za pośrednictwem właściwości window.onload...

...a ponieważ we właściwości tej zostało zapisane odwołanie do funkcji, zatem w celu obsługi zdarzenia onload zostanie wywołana funkcja initSeats().

Ogromną zaletą wykorzystania odwołań do funkcji i właściwości obiektów w celu określenia funkcji obsługujących zdarzenia jest możliwość całkowitego odseparowania kodu JavaScript od kodu HTML — nie ma już potrzeby umieszczania jakiegokolwiek kodu JavaScript w atrybutach znaczników HTML.

```
<body onload="initSeats();">   →   <body>
```

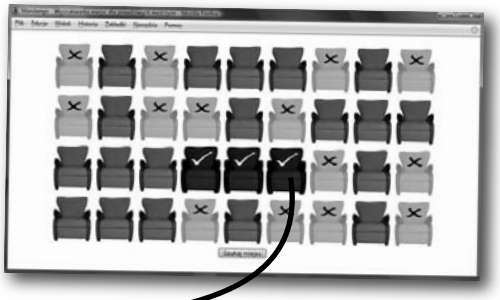
W końcu znacznik `<body>` może pozostać znacznikiem `<body>`, gdyż funkcja obsługująca zdarzenia `onload` została określona z poziomu kodu JavaScript. Musimy tylko zadbać o to, by kod określający funkcję obsługującą zdarzenia został wykonany możliwie jak najwcześniej; dlatego też zazwyczaj jest on umieszczany w sekcji nagłówka strony.

Jest jednak pewien problem. Co zrobić, gdy do funkcji obsługującej zdarzenia musimy przekazać jakiś argument, konieczny do jej prawidłowego wykonania? Problem ten nie występuje w przypadku zdarzeń `onload` używanych w aplikacji Mandango, jednak prawidłowe obsłużenie zdarzenia `onclick` wymaga już przekazania argumentu — numeru kliknięte miejsca. Odwołania do funkcji nie umożliwiają przekazywania argumentów… a zatem musimy przyjrzeć się temu zagadnieniu dokładniej.

Odwołania zapewniają wygodny sposób powiązania funkcji ze zdarzeniami, które te funkcje mają obsługiwać.

Literały funkcyjne spieszą z odsieczą

W wyniku kliknięcia obrazka fotela w aplikacji Mandango ma zostać wywołana funkcja `showSeatStatus()`, a do niej przekazany argument — liczba określająca numer klikniętego miejsca. W tym przypadku zwyczajne przypisanie odwołania do funkcji nie zda egzaminu, gdyż nie pozwoli nam przekazać argumentu. Mamy zatem problem. Na szczęście istnieje pewne rozwiązanie. Polega ono na użyciu literału funkcyjnego do utworzenia odwołania do funkcji i wywołaniu funkcji `showSeatStatus()` z określonym argumentem wewnątrz tego literału.



Pobieramy obiekt obrazka fotela, aby uzyskać dostęp do jego właściwości onclick.

```
document.getElementById("seat23").onclick = function(evt) {
  showSeatStatus(23);
};
```

Literał funkcyjny zawiera wywołanie metody showSeatStatus(), dzięki czemu możemy przekazać do niej argument.

Literał funkcyjny jest zapisywany we właściwości onclick jako odwołanie do funkcji.

Obiekt zdarzenia jest automatycznie przekazywany do procedury obsługi zdarzeń jako pierwszy argument jej wywołania.

Literał funkcyjny jest używany wyłącznie jako „pojemnik", w którym zostanie umieszczone wywołanie funkcji `showSeatStatus()`; niemniej pełni on kluczową rolę, gdyż zapewnia możliwość przekazania do niej argumentu określającego numer klikniętego miejsca. Możesz wyobrazić sobie ten literał funkcyjny jako bezimienną funkcję obsługującą zdarzenia. Właśnie z tego powodu literały funkcyjne są także nazywane **funkcjami anonimowymi**.

Powyższy kod pokazuje, że JavaScript udostępnia obiekt zdarzenia, który jest przekazywany do procedury obsługi zdarzeń, w tym przypadku pod postacią argumentu `evt`. Obiekt ten zawiera informacje charakterystyczne dla konkretnego typu zdarzenia. W tym przypadku nie interesują nas żadne dodatkowe informacje o zdarzeniu, zatem możemy zignorować argument `evt` i w ogóle go nie używać.

Literały funkcyjne pozwalają nam tworzyć anonimowe funkcje obsługujące zdarzenia.

Ćwiczenie

Skojarz funkcję `initSeats()` z procedurą obsługi zdarzeń onload, używając do tego celu literału funkcyjnego, a nie odwołania do funkcji.

.....

.....

.....

Ćwiczenie Rozwiązanie

Skojarz funkcję `initSeats()` z procedurą obsługi zdarzeń onload, używając do tego celu literału funkcyjnego, a nie odwołania do funkcji.

```
window.onload = function(evt) {
  initSeats();
};
```

Funkcja initSeats() jest wywoływana wewnątrz literału funkcyjnego obsługującego zdarzenie onload.

Argument evt jest ignorowany, gdyż procedura obsługi zdarzeń onload nie potrzebuje żadnych informacji o zdarzeniu.

Czym jest kojarzenie?

Wciąż pozostaje nam do rozwiązania jeden problem związany z kojarzeniem procedur obsługi zdarzeń przy wykorzystaniu literałów funkcyjnych. Wiemy już, że procedurę obsługi zdarzeń `onload` możemy określić w sekcji nagłówka strony WWW, w znaczniku `<script>`, używając w tym celu zwyczajnego kodu JavaScript. Rozwiązanie to będzie działać dobrze, gdyż kod takiej procedury zostanie wykonany dopiero po zakończeniu wczytywania strony (bo właśnie wtedy generowane jest zdarzenie `onload`), czyli w tym samym momencie, w którym zostałaby wywołana funkcja `initSeats()` umieszczona w kodzie HTML w atrybucie `onload` znacznika `<body>`. Rodzi się jednak pytanie: „Kiedy następuje skojarzenie innych literałów funkcyjnych z procedurami obsługi zdarzeń?”.

Kod obsługujący zdarzenie onload jest doskonałym miejscem do określenia procedur obsługi wszelkich innych zdarzeń.

Aby odpowiedzieć na to pytanie, powinniśmy wrócić do funkcji zwrotnej obsługującej zdarzenia `onload`, która jest idealnym miejscem na określenie wszystkich pozostałych procedur obsługi zdarzeń wykorzystywanych na stronie.

```
window.onload = function() {
  // Skojarzenie innych procedur obsługi zdarzeń
  ...
  // Inicjalizacja prezentacji
  initSeats();
}
```

W ramach obsługi zdarzenia onload można określić procedury obsługi wszystkich innych zdarzeń używanych na stronie.

Kod, który jest ściśle związany z obsługą zdarzenia onload, także zostanie wykonany w tej procedurze obsługi zdarzenia.

Powyższy przykład wyraźnie pokazuje, że procedura obsługi zdarzenia `onload` staje się centralnym miejscem, w którym określane są wszystkie inne procedury obsługi zdarzeń wykorzystywane na stronie. A zatem procedura ta nie tylko wykonuje standardowe czynności inicjalizacyjne, takie jak określenie wyglądu poszczególnych miejsc, lecz także określa pozostałe procedury obsługi zdarzeń.

Nie ma niemądrych pytań

P: Dlaczego funkcje zwrotne mają takie znaczenie?

O: Funkcje zwrotne są bardzo ważne, gdyż pozwalają reagować na zdarzenia zachodzące poza Twoim kodem. Zamiast wywoływać funkcję ze swojego kodu, tworzysz funkcję, która oczekuje na pewne zdarzenia i jest gotowa do ich obsługi. Kiedy takie zdarzenie nastąpi, przeglądarka ma obowiązek poinformować o tym fakcie odpowiednią funkcję zwrotną, czyli w rezultacie wywołać ją. Całe Twoje zadanie polega na przygotowaniu teatru przyszłych zdarzeń, czyli skojarzeniu funkcji zwrotnej z „czynnikiem", który ma powodować jej wykonanie, takim jak zdarzenie.

P: Czy funkcje zwrotne to coś innego niż procedury obsługi zdarzeń?

O: Tak. Inny często spotykany sposób wykorzystania funkcji zwrotnych przedstawimy w rozdziale 12. Wykorzystamy je tam do obsługi danych przesyłanych z serwera do przeglądarki przy wykorzystaniu techniki Ajax.

P: Wciąż nie do końca „czuję" literały funkcyjne. Czym one są i niby dlaczego są takie ważne?

O: Literał funkcyjny to sama treść funkcji, bez żadnej nazwy, czyli coś podobnego do każdej innej informacji, takiej jak łańcuch znaków lub liczba. Są one ważne, gdyż doskonale nadają się do użycia w sytuacjach, gdy potrzeba nam szybkiej, jednorazowej funkcji zwrotnej, czyli wtedy, gdy potrzeba nam funkcji, która zostanie wywołana tylko raz i to w dodatku nie przez nasz kod. A zatem z powodzeniem można stworzyć literał funkcyjny i przypisać go bezpośrednio do właściwości obsługi zdarzenia. Rozwiązanie to da identyczne efekty co stworzenie normalnej, nazwanej funkcji i przypisanie odwołania do niej. W praktyce stosowanie literałów funkcyjnych sprowadza się zatem do efektywności pisania kodu, gdyż jest wygodne w sytuacjach, gdy tworzenie zwyczajnych, nazwanych funkcji nie jest konieczne. Nie należy także zapominać, że w rzeczywistości literały funkcyjne są konieczne tylko w bardziej złożonych sytuacjach, kiedy zwyczajne odwołania do funkcji okażą się niewystarczające; na przykład gdy konieczne będzie przekazanie argumentów do funkcji zwrotnej.

Zaostrz ołówek

Uzupełnij brakujący kod nowej wersji procedury obsługi zdarzenia onload aplikacji Mandango.

```
window.onload = function() {
  // Skojarzenie procedury obsługi kliknięcia przycisku Szukaj miejsc
  document.getElementById("findseats").............  = ..............;

  // Skojarzenie procedury obsługi kliknięcia obrazka miejsca
  document.getElementById("seat0").........  = function(evt){.............};
  document.getElementById("seat1").........  = function(evt){.............};
  document.getElementById("seat2").........  = function(evt){.............};
  ...

  // Inicjalizacja prezentacji
  .........
}
```

Zaostrz ołówek
Rozwiązanie

Uzupełnij brakujący kod nowej wersji procedury obsługi zdarzenia onload aplikacji Mandango.

Cała procedura obsługi zdarzenia onload jest jednym dużym literałem funkcyjnym.

Funkcja findSeats() jest kojarzona ze zdarzeniem onclick przy użyciu odwołania do funkcji.

```
window.onload = function() {

  // Skojarzenie procedury obsługi kliknięcia przycisku Szukaj miejsc

  document.getElementById("findseats").onclick = findSeats;


  // Skojarzenie procedury obsługi kliknięcia obrazka miejsca

  document.getElementById("seat0").onclick = function(evt){showSeatStatus(0);};

  document.getElementById("seat1").onclick = function(evt){showSeatStatus(1);};

  document.getElementById("seat2").onclick = function(evt){showSeatStatus(2);};

  ...


  // Inicjalizacja prezentacji

  initSeats();

}
```

W końcu wywoływana jest funkcja initSeats(), która kończy obsługę zdarzenia onload.

Podczas określania procedur obsługi zdarzeń obrazków poszczególnych miejsc korzystamy z właściwości onclick odpowiednich obiektów strony.

Funkcja showSeatStatus() jest wywoływana wewnątrz literału funkcyjnego, dzięki czemu można do niej przekazać argument.

KLUCZOWE ZAGADNIENIA

- Funkcje zwrotne są wywoływane przez przeglądarkę w odpowiedzi na zdarzenia zachodzące poza **skryptem**.
- Odwołań do funkcji można używać do **przypisywania funkcji** w taki sposób, jak gdyby były **zwyczajnymi zmiennymi**.
- Odwołania do funkcji pozwalają kojarzyć funkcje ze zdarzeniami, które mają być obsługiwane, z poziomu kodu JavaScript, **bez konieczności** wprowadzania jakichkolwiek **zmian w kodzie HTML**.
- Literały funkcyjne to **bezimienne funkcje**, które z powodzeniem można wykorzystywać wtedy, gdy stosowanie zwyczajnych, nazwanych funkcji nie jest konieczne.

Nie ma niemądrych pytań

P: Dlaczego procedura obsługi zdarzenia `onload` w aplikacji Mandango została stworzona jako literał funkcyjny?

O: Ponieważ nie ma powodu tworzyć w tym celu zwyczajnej, nazwanej funkcji. Funkcja ta jest wywoływana tylko raz w odpowiedzi na zdarzenie `onload`. Oczywiście nic nie stoi na przeszkodzie, byśmy utworzyli zwyczajną funkcję i przypisali odwołanie do niej we właściwości `window.onload`; jednak powiązanie pomiędzy zdarzeniem i funkcją zwrotną jest bardziej czytelne i zrozumiałe w przypadku użycia literału funkcyjnego niż odwołania do funkcji.

P: Czy inne procedury obsługi zdarzeń muszą być określane wewnątrz procedury obsługi zdarzeń `onload`?

O: Tak. Możesz sądzić, że nic nie stoi na przeszkodzie, by określić je wewnątrz znacznika `<script>` umieszczonego w sekcji nagłówka strony; pamiętaj jednak, że w tym momencie zawartość strony nie jest jeszcze w całości wczytana do przeglądarki, co oznacza, że wywołania metody `getElementById()` mogłyby nie znaleźć żądanych elementów, a procedury obsługi zdarzeń nie zostałyby określone. Zgłoszenie zdarzenia `onload` gwarantuje, że cała zawartość strony będzie już wczytana.

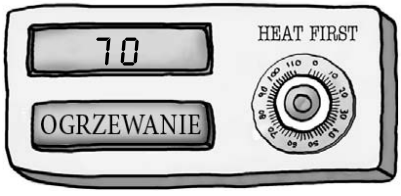
Struktura strony HTML

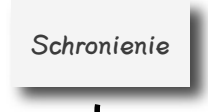
Oddzielenie kodu JavaScript od kodu HTML pokazuje, jak niewielka i prosta jest w rzeczywistości strukturalna część strony aplikacji Mandango. Dzięki usunięciu podatnego na błędy kodu JavaScript kod HTML strony stał się znacznie łatwiejszy do utrzymania.


```
<body>
    <div style="margin-top:25px; text-align:center">
      <img id="seat0" src="" alt="" />
      <img id="seat1" src="" alt="" />
      <img id="seat2" src="" alt="" />
      <img id="seat3" src="" alt="" />
      <img id="seat4" src="" alt="" />
      <img id="seat5" src="" alt="" />
      <img id="seat6" src="" alt="" />
      <img id="seat7" src="" alt="" />
      <img id="seat8" src="" alt="" /><br />
      <img id="seat9" src="" alt="" />
      <img id="seat10" src="" alt="" />
      <img id="seat11" src="" alt="" />
      <img id="seat12" src="" alt="" />
      <img id="seat13" src="" alt="" />
      <img id="seat14" src="" alt="" />
      <img id="seat15" src="" alt="" />
      <img id="seat16" src="" alt="" />
      <img id="seat17" src="" alt="" /><br />
      <img id="seat18" src="" alt="" />
      <img id="seat19" src="" alt="" />
      <img id="seat20" src="" alt="" />
      <img id="seat21" src="" alt="" />
      <img id="seat22" src="" alt="" />
      <img id="seat23" src="" alt="" />
      <img id="seat24" src="" alt="" />
      <img id="seat25" src="" alt="" />
      <img id="seat26" src="" alt="" /><br />
      <img id="seat27" src="" alt="" />
      <img id="seat28" src="" alt="" />
      <img id="seat29" src="" alt="" />
      <img id="seat30" src="" alt="" />
      <img id="seat31" src="" alt="" />
      <img id="seat32" src="" alt="" />
      <img id="seat33" src="" alt="" />
      <img id="seat34" src="" alt="" />
      <img id="seat35" src="" alt="" /><br />
      <input type="button" id="findseats" value="Szukaj miejsc" />
    </div>
</body>
```

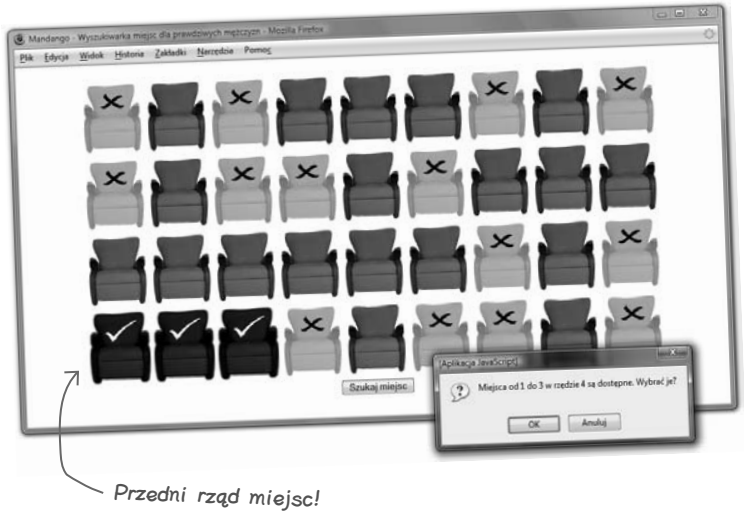
Mały krok dla JavaScriptu...

Choć nie udało się nam rozwiązać problemu pokoju na świecie, to jednak zrobiliśmy krok w dobrym kierunku, wykorzystując JavaScript do sterowania temperaturą w naszym domu. Podział dużych problemów na małe, skoncentrowanie uwagi na pojedynczych, konkretnych zadaniach i możliwość wielokrotnego stosowania kodu — oto, w jaki sposób funkcje mogą poprawić pisany przez nas kod JavaScript.





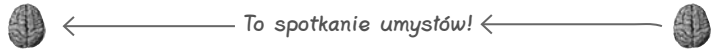
Szymek i Jasiek zastosowali dokładnie te same techniki rozwiązywania problemów, dzięki którym udało się im stworzyć lepiej zorganizowaną i łatwiejszą do utrzymania wersję aplikacji Mandango. Może to nie jest zbyt wiele, ale przynajmniej w świecie prawdziwych macho chodzących z kumplem do kina zapanował spokój…

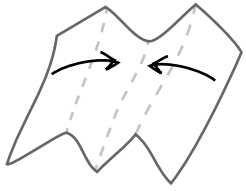


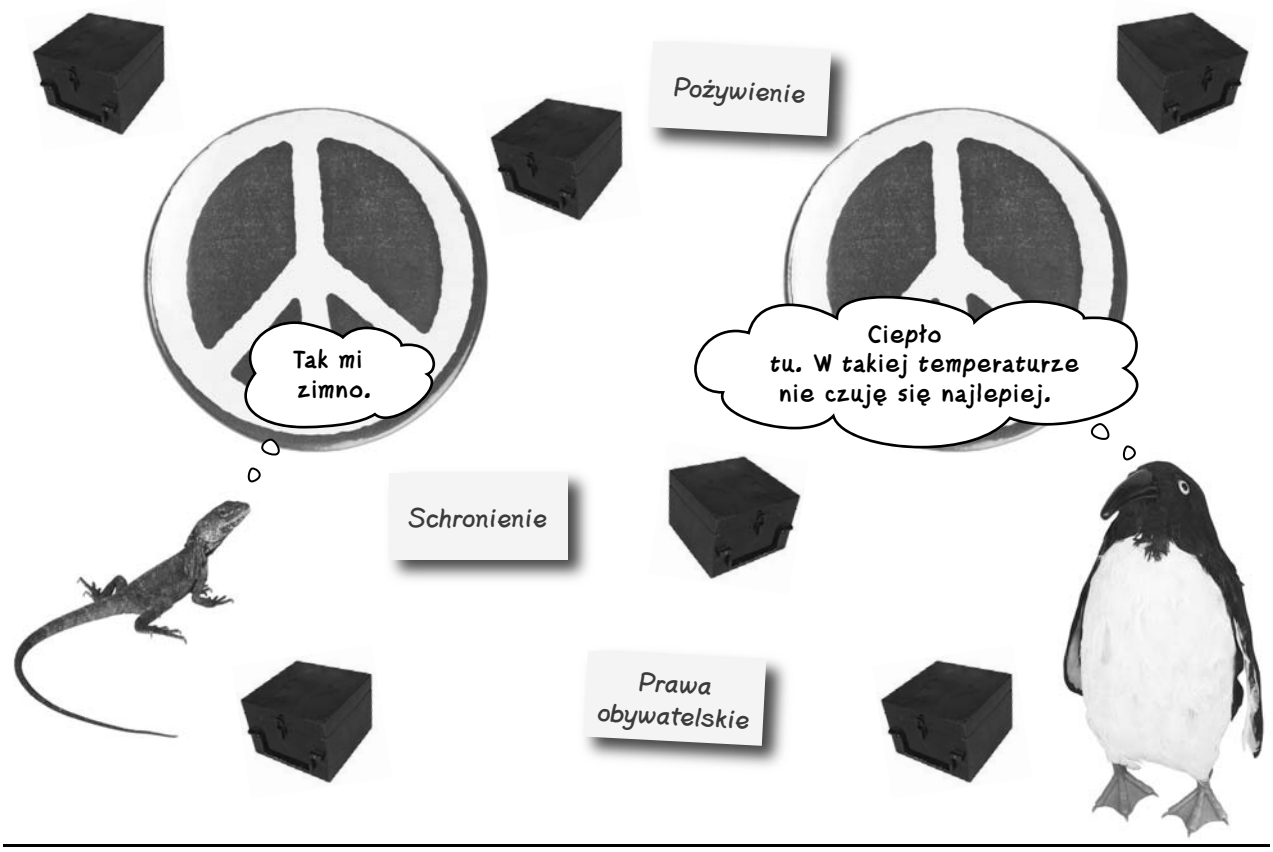
Zaginarka stron

Zegnij stronę wzdłuż pionowych linii, tak by oba mózgi się połączyły, a następnie rozwiąż zagadkę.

Co nowego dodają funkcje do „życia" Twojego kodu JavaScript?







Pokój zawsze jest trudnym zagadnieniem.
Nawet w świecie JavaScriptu jedynie kod o najlepszej organizacji może zapewnić spokój.
Nie jest zatem łatwo zapewnić sobie spokojny i komfortowy byt... przynajmniej w świecie JavaScriptu.