2

Fikcja interaktywna

Czytelnikom, którzy wychowali się na nowoczesnych grach typu MMORPG, realistycznych strzelankach FPS czy symulatorach sportowych wywołujących wrażenie, jakby naprawdę brało się udział w grze, stare gry przystosowane do działania na słabym, przestarzałym sprzęcie i utworzone przy użyciu niewyszukanego oprogramowania mogą wydawać się mało atrakcyjne. W grach z gatunków **fikcji interaktywnej**, jak np. seria *Choose Your Own Adventure*, oraz „wskaż i kliknij" trzeba było stosować całkiem inne chwyty, aby zatrzymać gracza na dłużej. W przypadku niektórych gier, takich jak np. *Zork*, przedstawienie rozbudowanego świata z szerokim wyborem czasowników do interakcji poprzez wiersz poleceń oraz oddanie do dyspozycji gracza dużej liczby obiektów, które można podnieść, zjeść, zbadać itd., było absolutną nowością.

W przypadku gier przygodowych typu „wskaż i kliknij" na NES, takich jak np. *Shadowgate* czy *Maniac Mansion*, daje się odczuć wszechogarniającą atmosferę interaktywnego świata. Należy również podkreślić, że twórcy tych gier zdołali przemycić wiele elementów horroru i komedii, jeśli weźmie się pod uwagę możliwości technologiczne tamtych czasów. Przy okazji jeśli jesteś fanem gry *Maniac Mansion*, JavaScriptu albo ogólnie gier bądź tego gatunku gier, przeczytaj artykuł Douglasa Crockforda na temat jego prac nad wersją tej gry na konsolę NES (*http://www.crockford.com/wrrrld/maniac.html*). Jeśli nie wiesz, kim jest Crockford, podpowiem, że jest to autor książki *JavaScript: The Good Parts, JSLint* oraz ogólnie jest znanym w całej społeczności guru od JavaScriptu.

Możesz pomyśleć, że to przestarzały gatunek gier, ale niektórzy potrafią na nim zarobić. Przykładowo gra *Double Fine Adventure*, najnowszy produkt twórców gier *Monkey Island* i *Maniac Mansion*, przyniosła w 2012 r. prawie 3,5 miliona dolarów zysku. Japońskie symulatory randek też aktualnie cieszą się wielką popularnością i nawet wywierają wpływ na inne gatunki, takie jak początkowo typowy RPG *Harvest Moon* na konsolę NES, w którym gracz jednocześnie stara się o rękę panny i buduje farmę. Przy okazji drobne ostrzeżenie: takie tematy gier jak zdobywanie czy ratowanie dziewczyny wiele osób uważa za sztampowe i niestosowne. Zawsze staraj się dostosować do gustu innych osób, a jeśli tworzysz grę dla innych, uważaj, aby przez nieuwagę nie wyprodukować czegoś, co będzie ich *obrażać*. Obecnie możliwości zarówno kulturowe, jak i techniczne są znacznie szersze niż kiedyś i jeśli nie zbadasz wszystkich dostępnych opcji, zmarnujesz dużą część potencjału. W gatunku opisywanym w tym rozdziale drzemią duże możliwości, jeśli chodzi o interakcję, która może odbywać się w wolnej lub ograniczonej formie, oraz tworzenie nastroju, który może być mroczny, figlarny albo coś pomiędzy tymi dwoma.

Do budowy gry w tym rozdziale użyjemy silnika *impress.js*, który jest zazwyczaj wykorzystywany do tworzenia prezentacji w postaci slajdów. Tworzy on interfejs podobny do książki, którą można przeglądać strona po stronie. Ale można też zaimplementować funkcję pozwalającą przejść do strony o dowolnym numerze. Ponadto możemy korzystać z niektórych technik CSS3, takich jak przejścia, skalowanie czy rotacja, bez potrzeby implementowania ich własnoręcznie. Do stylu gry *Maniac Mansion* zbliżymy się poprzez utworzenie zestawu przedmiotów, które będzie można zbierać do schowka i używać ich, gdy będą potrzebne. Funkcję tę zaimplementujemy za pomocą standardowej techniki przeciągania i upuszczania elementów HTML5. Rozbudowany system rozpoznający dowolny czasownik połączony z jednym lub dwoma dowolnymi rzeczownikami jest bardzo trudno zbudować. Nawet skrypt dla tego typu silnika byłby trudny do napisania. Dlatego ułatwimy sobie pracę, wiążąc wybrane obiekty ze sobą, tak że ich interakcja będzie miała tylko jeden efekt. To spowoduje, że sposób gry będzie trochę przypominał *Minecrafta*, ale w formie kadrów rysunkowych.

Receptura. Stylizowane strony

Jeśli otworzysz stronę *start/index.html* w folderze *fikcja_interaktywna*, to odkryjesz, że zawiera ona niewiele treści. Ładowana jest na nią biblioteka *impress.js*, ale nie ma jeszcze elementów `div` tworzących strony (które będę też czasami nazywał **slajdami**) opowieści. Dodamy je teraz (pogrubiony kod na listingu 2.1).

Listing 2.1. Dodanie stron historii do pliku index.html

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8" />
```

```
    <title>Fikcja interaktywna</title>
    <link href="main.css" rel="stylesheet" />
</head>

<body>
<div id="impress">
    <div id="1" class="step slide">
        <q>Znajdujesz się na pierwszym slajdzie (stronie). Naciśnięcie klawisza
        ↪strzałki w dół lub w prawo spowoduje otwarcie następnej strony. Strzałki
        ↪w górę i w lewo pozwalają przejść wstecz. </q>
    </div>
    <div id="2" class="step slide">
        <q>Do kolejnej strony można też przejść, naciskając spację.</q>
    </div>
    <div id="3" class="step slide">
        <q>Znajdujesz się na ostatniej stronie. Naciśnięcie klawisza strzałki w prawo
        ↪spowoduje przejście na początek.</q>
    </div>
</div>

<script src="impress.js"></script>
<script>impress().init();</script>

</body>
</html>
```

Najpierw przyjrzymy się niepogrubionej szablonowej części kodu. Wiersz z deklaracją kodowania `charset="utf-8"` można by było usunąć, ale dzięki niemu będą wyświetlały się błędy w konsoli nawet podczas używania tylko podstawowych liter z angielskiego alfabetu. Deklaracji tej nie ma w wielu innych przykładach w tej książce, więc jeśli pojawiają się irytujące błędy, po prostu dodaj ją do stron. Ponadto atrybut ten korzystnie wpływa na dostępność dokumentu dla czytników ekranu (dzięki niemu mogą poprawnie wymawiać słowa) oraz umożliwia przeglądarkom tłumaczenie strony na wybrany przez użytkownika język. Kolejną ważną rzeczą, na którą należy zwrócić uwagę, jest to, że pliki JavaScript są wczytywane na samym końcu strony, przed zamykającym znacznikiem `</body>`, a nie w nagłówku. Jest to korzystne, ponieważ dzięki temu przeglądarka najpierw wczyta zawartość strony, a dopiero potem wykona czasochłonne skrypty JavaScript, które mogłyby na chwilę zablokować renderowanie treści. Ponadto zwróć uwagę na brak elementu `canvas`. Nie będzie on też ładowany przez bibliotekę *impress.js*. Do utworzenia tej gry zostaną wykorzystane JavaScript, CSS3 oraz części HTML5 niezwiązane z elementem `canvas`.

Teraz przestudiujemy kod stron. Nie jest skomplikowany. Wszystkie strony znajdują się w elemencie `div` z identyfikatorem `impress`, a poszczególne strony są umieszczone w osobnych elementach `div` z klasami `step` i `slide`. Na razie w treści stron znajdują się instrukcje dotyczące sposobu nawigacji, ale bez dodania odpowiedniego kodu CSS przechodzenie między slajdami nie będzie możliwe. Plik *main.css*, który ładujemy do pliku *index.html*, jeszcze nie istnieje. Czas go utworzyć i dodać do niego treść widoczną na listingu 2.2.

Listing 2.2. Dodanie kodu CSS definiującego formatowanie stron i umożliwiającego nawigację między nimi

```
html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
```

```
b, u, i, center,
dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th, td,
article, aside, canvas, details, embed,
figure, figcaption, footer, header, hgroup,
menu, nav, output, ruby, section, summary,
time, mark, audio, video {
    margin: 0;
    padding: 0;
    border: 0;
    font-size: 100%;
    font: inherit;
    vertical-align: baseline;
}

blockquote, q {
    quotes: none;
}

article, aside, details, figcaption, figure,
footer, header, hgroup, menu, nav, section {
    display: block;
}
ol, ul {
    list-style: none;
}
table {
    border-collapse: collapse;
    border-spacing: 0;
}
b, strong { font-weight: bold }
i, em { font-style: italic }
/* Koniec resetu */
body {
    background: -webkit-gradient(radial, 50% 50%, 0, 50% 50%, 500, from(rgb(0, 40,
    ↪200)), to(rgb(10, 10, 0)));
    background:    -webkit-radial-gradient(rgb(0, 40, 200), rgb(10, 10, 0));
    background:    -moz-radial-gradient(rgb(0, 40, 200), rgb(10, 10, 0));
    background:    -ms-radial-gradient(rgb(0, 40, 200), rgb(10, 10, 0));
    background:    -o-radial-gradient(rgb(0, 40, 200), rgb(10, 10, 0));
    background:    radial-gradient(rgb(0, 40, 200), rgb(10, 10, 0));
}
.impress-enabled .step {
    margin: 0;
    opacity: 0.0;
    -webkit-transition: opacity 1s;
    -moz-transition:    opacity 1s;
    -ms-transition:     opacity 1s;
    -o-transition:      opacity 1s;
    transition:         opacity 1s;
}
.impress-enabled .step.active { opacity: 1 }
.slide {
    display: block;
    width: 700px;
    height: 600px;
    padding: 40px 60px;
    background-color: white;
```

```
    border: 1px solid rgba(0, 0, 0, .3);
    color: rgb(52, 52, 52);
    text-shadow: 0 2px 2px rgba(0, 0, 0, .1);
    border-radius: 5px;
    font-family: 'Open Sans', Arial, sans-serif;
    font-size: 30px;
}
```

W tym arkuszu należy zwrócić uwagę na dwie kwestie. Po pierwsze, znajduje się w nim tzw. reset CSS. W internecie można znaleźć wiele tego rodzaju arkuszy, które mogą mieć nawet kilkaset wierszy kodu. Stosuje się je, ponieważ przeglądarki dla niektórych elementów mają bardzo różne domyślne ustawienia. W resecie CSS definiuje się jednolite ustawienia dla tych wszystkich elementów, dzięki czemu ma się pewność, że będą one we wszystkich przeglądarkach wyglądać tak samo albo bardzo podobnie.

Po drugie, od wiersza zawierającego selektor `body` rozpoczyna się część, w której znajdują się zwykłe reguły formatujące. W tle elementu `body` został ustawiony gradient promienisty, niebieski w środku i przechodzący w ciemnoszary na zewnątrz. Definicja tego gradientu jest zwielokrotniona, ponieważ nie ma jeszcze ustalonego jednolitego standardu implementacji tej części specyfikacji CSS. Podobnie wygląda sytuacja ze zdefiniowaną dalej własnością `opacity`. Przy jej użyciu sprawiamy, że aktywna strona jest widoczna, nieaktywne strony są niewidoczne, a zmiana strony trwa sekundę. Klasa `slide` ma zdefiniowane standardowe ustawienia, aby strony były dobrze widoczne na czarnym i niebieskim tle.

UWAGA

Jeśli chcesz lepiej poznać język CSS3, to warto zajrzeć na stronę *http://css3please.com*.

Jeśli wszystko poszło zgodnie z planem, to strona *index.html* powinna w przeglądarce wyglądać tak jak na rysunku 2.1.

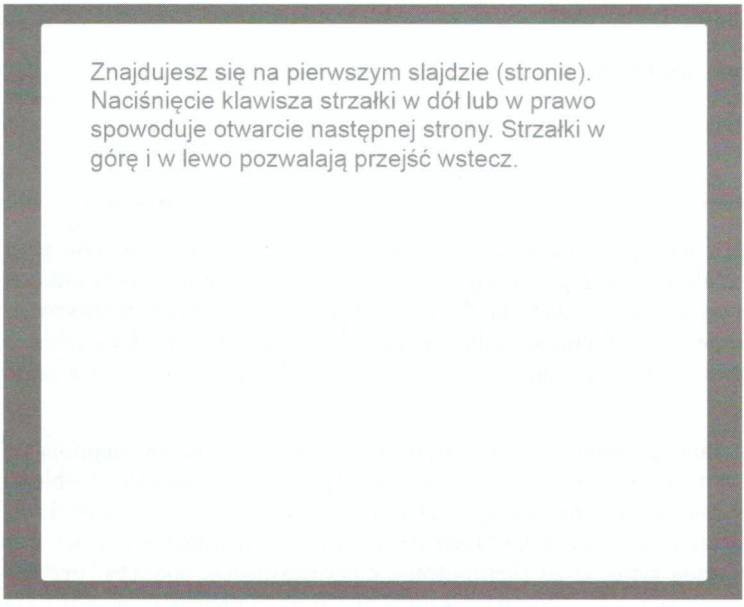
Teraz napiszemy krótką opowieść i zaimplementujemy funkcję zmiany stron książki.

Receptura. Zmienianie stron

W naszej grze na niektórych stronach trzeba będzie podejmować pewne decyzje, a na innych będzie następował koniec gry, co może być zarówno pozytywnym, jak i negatywnym zdarzeniem. Najpierw zajmiemy się stroną decyzyjną. W elemencie `div` o identyfikatorze `1` element `p` zamień na kod oznaczony pogrubieniem na listingu 2.3.

Listing 2.3. Strona decyzyjna

```
<div id="1" class="step slide">
  <div class="slide-text">
    <p>Tam jest maszyna czasu. Przestawić przełącznik?</p>
    <p><a href="#2">PRZEŁĄCZ</a></p>
    <p><a href="#3">POCZEKAJ,</a> aż tata, który jest naukowcem, wróci
    ↪do domu</p>
  </div>
</div>
```



Rysunek 2.1. Gotowy sformatowany slajd

Dodaliśmy łącza do innych stron oraz umieściliśmy elementy p w nowym elemencie div z klasą slide-text. Wkrótce zajmiemy się ich formatowaniem za pomocą CSS. Ale najpierw zobaczymy, dokąd prowadzą te łącza, zastępując elementy div z identyfikatorami 2 i 3 elementami pokazanymi na listingu 2.4.

Listing 2.4. Strony zakończenia gry

```
<div id="2" class="step slide" data-x="1500">
  <div class="slide-text">
    <p>Cofnąłeś się w czasie. To było fajne.</p>
  </div>
  <div class="menu"><a href="#1">ZACZNIJ OD NOWA</a></div>
</div>
<div id="3" class="step slide" data-x="-1500">
  <div class="slide-text">
    <p>Poobijałeś się trochę, aż w końcu Twój tata wrócił do domu.</p>
    <p>Pomógł Ci odrobić pracę domową.</p>
    <p>Było całkiem fajnie, ale nie aż tak.</p>
  </div>
  <div class="menu"><a href="#1">ZACZNIJ OD NOWA</a></div>
</div>
```

Dokonaliśmy kilku zmian w drugiej i trzeciej stronie. Ze względu na sposób działania biblioteki *impress.js* dodaliśmy atrybut data-x. Bez niego wszystkie slajdy byłyby ułożone w stos i kliknięcie powodowałoby, że wszystkie byłyby priorytetowe w nawigacji. Wtedy każde kliknięcie przenosiłoby gracza na ostatnią stronę. Aby zaobserwować ten efekt, wyłącz przezroczystość w regule .impress-enabled .step i kliknij w dowolnym miejscu. Zostaniesz przeniesiony na ostatnią stronę. Definiując atrybut data-x, powodujemy ustawienie drugiej i trzeciej strony na lewo i prawo od pierwszej strony.

Podobnie jak w przypadku pierwszej strony, główny tekst umieszczamy w elemencie `div` z klasą `slide-text`. Ponadto pod grą został umieszczony element `div` z klasą `menu` zawierający łącze do pierwszej strony gry.

Zakończenia tej gry nie są dramatyczne. Jeśli chcesz, możesz je zmienić na bardziej niezwykłe, ale w tym gatunku gry niesamowite jest właśnie to, jak łatwo można utworzyć nowe zakończenie. Wystarczy tylko dodać nowy slajd z informacją, co się wydarzyło. Nie trzeba dla każdego zakończenia tworzyć całej nowej sceny z muzyką i animacjami. Gdy gra RPG na konsolę SNES taka jak *Chrono Trigger* ma kilka zakończeń (i opcję NewGame+), twórcom należy się szacunek nie tylko dlatego, że gra ta jest elegancko wykonana, ale również dlatego, że napisanie nowych tekstów, animacji i dodanie dźwięków jest dużym wysiłkiem, dzięki któremu zagorzali fani produkcji mają dodatkową rozrywkę.

Jako że zakończenia w tym gatunku tworzy się z taką łatwością, można utworzyć wiele mniej i bardziej fascynujących. Wkrótce skorzystamy z tych możliwości, ale na razie napiszemy arkusze stylów dla nowych elementów. Kod widoczny na listingu 2.5 należy dopisać na końcu pliku *main.css*.

Listing 2.5. Formatowanie wnętrza slajdów

```
.slide p{
  margin-top:15px;
  margin-bottom:30px;
  line-height:45px;
}
.slide a, .slide a:visited {
  color: #071549;
  text-decoration:none;
  background-color: #abc;
  padding:5px;
  border-radius: 5px;
  border:1px solid #cde;
}
.slide a:hover, .slide a:active{
  background-color: #cde;
}
.slide .slide-text{
  height: 450px;
}
div.menu{
  border-top:2px solid black;
  padding-top:25px;
  text-align: center;
}
```

Jest to w większości zwykły kod CSS. Jeśli nie wiesz, do czego służy jakiś fragment, to go usuń i sprawdź, co się stanie. Dobrym pomysłem jest skorzystanie w tym celu z dodatku Firebug w Firefoksie albo narzędzi dla webmasterów w Chrome. Arkusz ten zawiera kilka poprawek wizualnych oraz odsuwa menu od części, w której jest opowiadanie. Na razie w tym menu znajduje się tylko przycisk rozpoczęcia gry od nowa wyświetlany na stronach zakończeniowych.

Receptura. Dodanie schowka z obsługą funkcji przeciągania i upuszczania

Aktualnie można przechodzić między stronami i przeżywać własną przygodę. Dobrze by było dodać kilka obrazków i schowek, aby dzięki niektórym funkcjom gra tego rodzaju bardziej przypominała produkcje typu *Maniac Mansion*.

W tej recepturze dodamy trzy kolejne pliki: *bat.png*, *game.js* oraz *dragDrop.js*. Będzie trzeba też trochę pozmieniać w plikach *index.html* i *main.css*. Zaczniemy od dodania do pliku *index.html* wierszy kodu pogrubionych na listingu 2.6.

Listing 2.6. Dodanie kontenerów przedmiotów do pliku index.html

```
<body>
<div id="player_inventory" class="itemable">
  <span class="item-container"><h3>Schowek:</h3>
    <div class="inventory-box empty"></div>
    <div class="inventory-box empty"></div>
    <div class="inventory-box empty"></div>
  </span>
</div>
<div id="impress">
    <div id="1" class="step slide itemable">
      <div class="slide-text">
        <p>Tam jest maszyna czasu. Przestawić przełącznik?</p>
        <p><a href="#2">PRZEŁĄCZ</a></p>
        <p><a href="#3">POCZEKAJ,</a> aż tata, który jest naukowcem, wróci
        ↪do domu</p>
      </div>
      <div class="menu item-container">
        <div class="inventory-box">
          <img src="bat.png" alt="bat" class="item" id="bat">
        </div>
      </div>
    </div>
    <div id="2" class="step slide itemable" data-x="1500">
      <div class="slide-text">
        <p>Cofnąłeś się w czasie. To było fajne.</p>
      </div>
      <div class="menu"><a href="#1">ZACZNIJ OD NOWA</a></div>
    </div>
    <div id="3" class="step slide itemable" data-x="-1500">
      <div class="slide-text">
        <p>Poobijałeś się trochę, aż w końcu Twój tata wrócił do domu.</p>
        <p>Pomógł Ci odrobić pracę domową.</p>
        <p>Było całkiem fajnie, ale nie aż tak.</p>
      </div>
      <div class="menu"><a href="#1">ZACZNIJ OD NOWA</a></div>
      </div>
</div>

<script src="impress.js"></script>
<script>impress().init();</script>

<script src="game.js"></script>
```

```
<script src="dragDrop.js"></script>

</body>
</html>
```

W tym pliku zostały wprowadzone cztery poważne zmiany. Pierwsza to dodanie elementu `div` z identyfikatorem `player_inventory` i kilkoma zagnieżdżonymi elementami. Style dotyczące tego elementu i jego potomków znajdują się na listingu 2.7. Na razie traktuj go jako zawsze widoczny schowek na przedmioty, a dodawaniem i usuwaniem tych przedmiotów zajmiemy się później. Druga zmiana to dodanie klasy `itemable` do elementów slajdów. Zostanie ona wykorzystana później w CSS i JavaScripcie. Trzecia zmiana to dodanie do pierwszego slajdu elementu z klasą `item-container`, zawierającego dodany w tej recepturze obraz kija. Ostatnia zmiana dotyczy ładowania nowych skryptów (*game.js* i *dragDrop.js*) przed zamknięciem elementu `body`.

Teraz spójrzmy na nowe style widoczne na listingu 2.7. Można je dodać na końcu pliku *main.css*.

Listing 2.7. Kod CSS dotyczący schowka i funkcji przeciągania przedmiotów

```
img.item{
  width:100%;
  height:100%;
  cursor: move;
  padding:0px;
}
#player_inventory{
  position: fixed;
  text-align: center;
  width:180px;
  padding:15px;
  border-radius: 5px;
  top: 75px;
  left: 25px;
  background-color: white;
}
.menu .inventory-box{
  margin-right:20px;
}
.inventory-box{
  position: static;
  display: inline-block;
  height:130px;
  width:130px;
  border-radius: 5px;
  text-align:center;
}
.inventory-box.empty{
  border: 2px dashed #000;
}
h3{
  font-weight:bold;
  font-size:30px;
  margin-bottom:15px;
}
```

Zdefiniowaliśmy style półek schowka, elementów kontenerów i przycisków menu znajdujących się na dole każdego slajdu. W klasie `empty` zdefiniowaliśmy obramowanie na wypadek sytuacji, gdy elementy w schowku nie będą miały graficznej reprezentacji. Zastosowany wcześniej reset wyzerował ustawienia elementu `h3` i dlatego zdefiniowaliśmy je na nowo w tej części arkusza.

Teraz czas na utworzenie pliku *dragDrop.js* do obsługi interakcji. Ponieważ jest to dość skomplikowane, kod ten omówię w częściach. Najpierw utwórz nowy plik o nazwie *dragDrop.js* i skopiuj do niego kod widoczny na listingu 2.8.

Listing 2.8. Dodanie procedur obsługi półek schowka

```
var itemBoxes = document.querySelectorAll('.inventory-box');
[].forEach.call(itemBoxes, function(itemBox) {
  itemBox.addEventListener('dragstart', handleDragStart);
  itemBox.addEventListener('dragover', handleDragOver);
  itemBox.addEventListener('drop', handleDrop);
});
```

Najpierw pobieramy wszystkie elementy mające klasę `inventory-box` do zmiennej o nazwie `itemBoxes`. Następnie za pomocą pętli `forEach` do każdego z tych elementów dodajemy procedurę obsługi operacji przeciągania. Metoda `addEventListener` służy do wiązania poszczególnych procedur obsługi zdarzeń. Używając jej, należy pamiętać, że starsze wersje Internet Explorera jej nie obsługują i w zamian trzeba korzystać z metody `attachEvent`. W przypadku pierwszej metody `addEventListener` pierwszy argument określa zdarzenie początku przeciągania. Istnieją też inne procedury obsługi zdarzeń, np. `click`, które również można związać z elementem. Drugi argument to nazwa funkcji wiązanej z elementem i wywoływanej w reakcji na określone zdarzenie.

Może dziwi Cię nietypowa pętla użyta w drugim wierszu powyższego kodu. Na listingu 2.9 jest przedstawiona pętla o takim samym działaniu, ale mająca bardziej typową składnię. Kod ten nie należy do żadnego pliku. Służy jedynie do porównania funkcyjnej pętli `forEach` ze zwykłą pętlą `for`.

Listing 2.9. Bardziej typowa pętla — w stylu proceduralnym, a nie funkcyjnym

```
for (var i=0; i < itemBoxes.length; i++){
  itemBoxes[i].addEventListener('dragstart', handleDragStart);
  ...
}
```

Pętla przedstawiona na listingu 2.9 jest napisana w stylu proceduralnym, a pętla na listingu 2.8 — w stylu funkcyjnym. Może się wydawać, że to kosmetyczna różnica, ale konsekwencje wyboru jednej z metod, zwłaszcza w przypadku dużych i skomplikowanych programów, są bardzo znaczące na korzyść pętli funkcyjnej. Najkrócej mówiąc, w tym stylu tworzy się mniej zmiennych i w razie potrzeby jest tworzony nowy zbiór danych, a nie modyfikowany istniejący. Dzięki temu system jest prostszy, ponieważ mamy większą pewność, że wartości zmiennych i funkcje będą spójne w całym programie.

Sposób działania drugiego wiersza kodu na listingu 2.8 jest dość skomplikowany. Najpierw funkcja `forEach` jest stosowana na rzecz literału tablicowego, `[]`, niezawierającego żadnych elementów. Robimy to, ponieważ mimo że tablice mają funkcję `forEach`, metoda `querySelectorAll` zwraca obiekt `NodeList`, który tej metody nie ma. Gdy używa się funkcji `call`, można udawać, że ją ma. Pierwszy parametr funkcji `call` jest wartością `this` funkcji wykonywanej przed `call`, a więc

UWAGA

W tej książce nie trzymam się ściśle zasad programowania funkcyjnego, ale zrozumienie korzyści, jakie daje ten styl programowania, jest przy używaniu tego języka bardzo ważne. W specyfikacji języka JavaScript ECMAScript 5 zachęca się do przyjęcia tego stylu programowania dla nowych przeglądarek. Natomiast stare przeglądarki można przystosować przy użyciu zewnętrznych bibliotek, takich jak np. *underscore.js*.

w tym przypadku `forEach`. Będzie to `itemBoxes`, a nie `[]`. Pozostałe parametry odnoszą się do parametrów przekazywanych funkcji `forEach`. Jako że pierwszy parametr funkcji `forEach` jest funkcją, która ma zostać wywołana, drugi parametr funkcji `call` (wszystko od słowa kluczowego `function` do klamry `}`) służy jako funkcja do zastosowania na każdym elemencie (`itemBox`) zmiennej `itemBoxes`. Funkcja `forEach` pozwala także na związanie etykiety `itemBox` z formalnym parametrem, którego użyjemy później. Możemy to zrobić, ponieważ mimo że obiekt `NodeList` nie ma funkcji `forEach`, ma metodę `item`, której funkcja `forEach` używa do ustawiania tego formalnego parametru.

To jest chyba najbardziej skomplikowany fragment kodu w całej książce (a przynajmniej do rozdziału 8.). Jeśli zrozumiałeś go za pierwszym razem, to jesteś niesamowity. Jeśli nie, to nie ma powodu do zmartwień. W JavaScripcie jest wiele do nauczenia, a poza tym w większości przypadków użycie pętli `for` zamiast `forEach` nie jest żadnym problemem. W niektórych implementacjach JavaScriptu, które w każdej przeglądarce mogą być inne, pętla `for` bywa nawet szybsza. Pamiętaj też, że nawet najbardziej utalentowani programiści od czegoś zaczynali i żaden cywilizowany człowiek nie wyleje na Ciebie kubła pomyj, gdy będziesz się jeszcze uczyć. Warto też o tym pamiętać, przeglądając kod innych osób i szukając inspiracji do podnoszenia swoich programistycznych umiejętności.

Możemy wrócić do gry. Teraz musimy zdefiniować funkcje, które związaliśmy z półkami na przedmioty. Przejdź na początek pliku *dragDrop.js* i zdefiniuj je wszystkie po kolei. Zaczniemy od funkcji `handleDragStart`, której kod znajduje się na listingu 2.10.

Listing 2.10. Funkcja handleDragStart

```
var draggingObject;
function handleDragStart(e) {
  draggingObject = this;
  e.dataTransfer.setData('text/html', this.innerHTML);
  var dragIcon = document.createElement('img');
  var imageName = this.firstChild.id;
  dragIcon.src = imageName + '.png';
  e.dataTransfer.setDragImage(dragIcon, -10, 10);
}
```

Najpierw zdefiniowaliśmy zmienną o nazwie `draggingObject`. Jest ona globalna, ponieważ będziemy jej potrzebować w różnych funkcjach. Na razie nie przejmuj się zakresem globalnym — później wyjaśnię dokładnie, o co z tym chodzi. Zmienną tę przypisujemy do elementu `inventory-box`, w którym rozpoczyna się przeciąganie. Następnie informujemy, że przekazywane dane są w formacie HTML i znajdują się w elemencie `inventory-box`. Cztery ostatnie wiersze kodu tworzą małą kopię przeciąganego obrazu, która jest wyświetlana pod kursorem. Gdyby usunąć tę część skryptu, funkcja przeciągania i tak by działała, ale sposób wyświetlania obrazu pod kursorem byłby inny w każdej przeglądarce.

Na listingu 2.11 znajduje się kolejna funkcja, `handleDragOver`, którą należy dodać do pliku *dragDrop.js*.

Listing 2.11. Funkcja handleDragOver

```
function handleDragOver(e) {
  e.preventDefault();
}
```

Ta funkcja jest z kolei bardzo prosta. Jej jedynym zadaniem jest zablokowanie domyślnej funkcji.

Na listingu 2.12 znajduje się kod źródłowy funkcji `handleDrop`, która robi znacznie więcej.

Listing 2.12. Funkcja handleDrop

```
function handleDrop(e) {
  e.preventDefault();
  if (draggingObject != this) {
    var draggingGrandpa = draggingObject.parentElement.parentElement;
    var draggedToGrandpa = this.parentElement.parentElement;
    var draggingObjectId = draggingObject.firstChild.id;
    inventoryObject.add(draggedToGrandpa.id, draggingObjectId);
    inventoryObject.remove(draggingGrandpa.id, draggingObjectId);
    draggingObject.innerHTML = this.innerHTML;
    this.innerHTML = e.dataTransfer.getData('text/html');
    this.classList.remove('empty');
    draggingObject.classList.add('empty');
  }
}
```

Aby ta procedura poprawnie działała, potrzebne jest w niej wywołanie `e.preventDefault()`. Większość kodu znajduje się w instrukcji warunkowej, która pozwala na jego wykonanie tylko wtedy, gdy pochodzenie obiektu, `draggingObject`, jest inne niż cel upuszczenia danych — `this`. Trzy kolejne wiersze kodu przypisują zmienne do obiektów kontenerów (z końcówką nazwy `grandpa`) i identyfikatorów tych elementów. Następne dwa wiersze przenoszą obiekt z oryginalnego obiektu `inventoryObject` do docelowego. Wkrótce dowiesz się, czym jest `inventoryObject`, ale najpierw skończymy z tą funkcją. W następnych dwóch wierszach zamieniamy własności `innerHTML` źródła i celu. Natomiast dwa ostatnie wiersze przedstawiają klasę `empty`.

Można też dodać procedury obsługi zdarzeń `dragenter` i `dragleave`, ale w tej recepturze nie jest nam to potrzebne.

Na listingu 2.13 znajduje się wspominany wcześniej obiekt `inventoryObject`. Dodaj ten kod do pliku *game.js*.

Listing 2.13. Obiekt inventoryObject do przechowywania i pobierania elementów

```
var inventoryObject = (function(){
  var inventory = {};
  var itemables = document.getElementsByClassName("itemable");
  [].forEach.call(itemables, function(itemable) {
    inventory[itemable.id] = [];
  });
  var items = document.getElementsByClassName("item");
  [].forEach.call(items, function(item) {
```

```
    var greatGrandpa = item.parentElement.parentElement.parentElement;
    inventory[greatGrandpa.id].push(item.id);
  });
  var add = function(inventorySection, newItem){
    inventory[inventorySection].push(newItem);
    return inventory;
  }
  var remove = function(inventorySection, itemToDelete){
    for (var i = 0; i < inventory[inventorySection].length; i++){
      if (inventory[inventorySection][i] == itemToDelete){
        inventory[inventorySection].splice(i, 1);
      }
    }
    return inventory;
  }
  return {
    get : function(){
            return inventory;
          },
    add : add,
    remove : remove
  }
})();
```

W pierwszym wierszu tego kodu znajduje się deklaracja zmiennej inwentarzowej, której wartość została ustawiona na wynik działania funkcji. Wiadomo, że jest to funkcja, ponieważ na końcu znajduje się nawias. Ze względu na sposób przetwarzania kodu przez parser JavaScript zamknięcie nawiasu znajdujące się w ostatnim wierszu powoduje, że konieczne jest ujęcie funkcji w nawias, tzn. `(function(){…})`. Bez tego dodatku powstałby błąd składniowy.

Następnie inicjujemy obiekt `inventory` pustym literałem `{}`. Później stosujemy opisaną wcześniej składnię funkcyjną w celu zainicjowania obiektu inwentarza pustymi tablicami pod każdym indeksem, używając identyfikatora slajdu lub `player_inventory` jako klucza. W drugim przypadku notacji funkcyjnej zapełniamy schowek obiektami utworzonymi w zależności od tego, co zostało znalezione w pliku HTML.

Dalej deklarujemy dwie nowe funkcje: `add` i `remove`. Nie wymagają specjalnych objaśnień, ale trzeba zwrócić uwagę na metody `push` i `splice`. Są to jedne z najczęściej używanych metod API JavaScriptu. Warto również zauważyć wartości zwrotne tej funkcji. W języku JavaScript jeśli nie zadeklaruje się konkretnej wartości zwrotnej, zwracana jest wartość `undefined`. W różnych sytuacjach zwraca się różne wartości, ale w przypadku manipulacji obiektem, jak w tym kodzie, najlepiej jest zwrócić obiekt, na którym się pracowało (tu: `inventoryObject`). Pomijając inne zalety tego podejścia, dzięki temu można tworzyć łańcuchy wywołań typu `inventoryObject.remove(…).add(…)`.

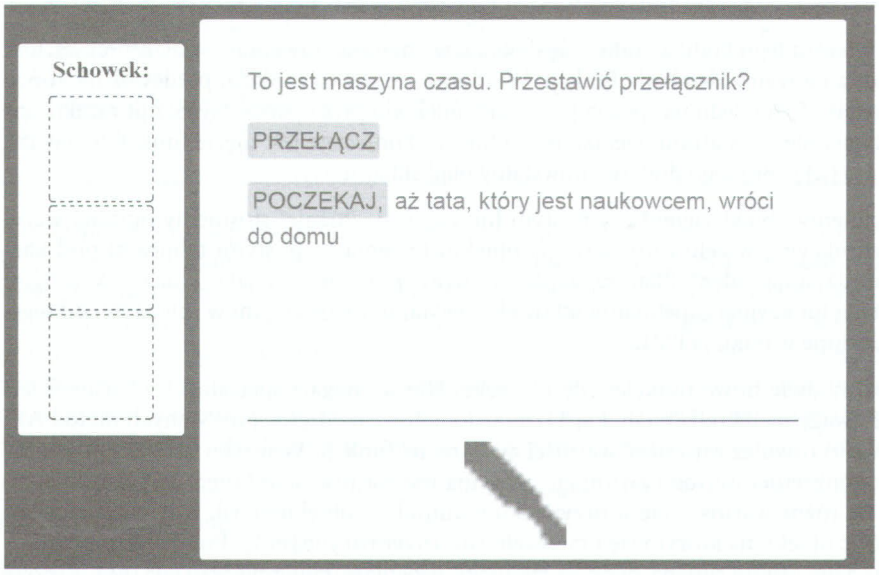
Jakie są implikacje zwracania jako wyniku obiektu magazynu? W istocie wykonaliśmy kod konfiguracyjny, ale mamy też funkcje `add` i `remove`, które nie będą dostępne poza kontekstem zawierającej je funkcji. Co można zrobić w tej sytuacji? Najlepiej sprawić, aby zewnętrzna funkcja zwracała obiekt, który został zadeklarowany w kilku ostatnich wierszach kodu. Obiekt ten zawiera trzy metody: `get`, `add` oraz `remove`. Jeśli chodzi o metody `get` i `remove`, to nie ma to nic wspólnego z implementacją. Wszystko działa tak samo, tylko metody te są teraz publiczne, tzn. dostępne poza funkcją, w której zostały zadeklarowane, poprzez zapis `inventoryObject.add()` i `inventoryObject.remove()`. Najlepsze jest, że te dostępne publicznie metody mają także dostęp do prywatnego obiektu `inventory`, który jest zapełniany we wcześniejszej części funkcji. Opisana

tu koncepcja nazywa się **zamknięciem** (ang. *closure*). Jest to bardzo ważne pojęcie z zakresu określania dostępności informacji w programach.

Jeśli chodzi o funkcję `get`, to zamiast mapować na prywatną implementację poza blokiem `return`, wpisaliśmy kod źródłowy bezpośrednio w linii (zwraca on po prostu wartość prywatnej zmiennej `inventory`).

Przedstawiona technika programowania polegająca na używaniu publicznych i prywatnych metod nazywa się **modularyzacją** (ang. *module pattern*). Jest to jeden z najpowszechniej stosowanych wzorców projektowych i z pewnością wiele razy znajdziesz jego zastosowanie w kodzie napisanym przez innych programistów. Jeżeli interesują Cię metody organizacji i wielokrotnego wykorzystania kodu JavaScript, to dobrze trafiłeś, bo w tej dziedzinie sporo się dzieje. Wśród takich tematów jak szkielety MVC, np. Backbone czy AMD/*common.js* do ładowania rozszerzeń czy klasyczne wzorce **bandy czworga**, na pewno znajdziesz coś ciekawego, aby pogłębić swoją wiedzę.

Ale ta książka jest o tworzeniu gier, nie o wzorcach projektowych. Przedstawiona receptura jest wyjątkowo trudna do zrozumienia, ponieważ zawiera dużo informacji o skomplikowanych zagadnieniach, których znajomość przyda Ci się jednak także w dalszych rozdziałach. Aby się upewnić, że Twój wysiłek nie poszedł na marne, otwórz plik *index.html* w przeglądarce internetowej. Powinieneś zobaczyć stronę widoczną na rysunku 2.2. Możesz przeciągnąć kij ze schowka i umieścić go tam z powrotem.



Rysunek 2.2. Pierwszy slajd gry po dodaniu schowka

Receptura. Dodawanie złożonych interakcji

W tej recepturze utworzymy bardziej złożone interakcje z naszymi obiektami. Nie będzie tu niczego trudniejszego do zrozumienia niż w poprzedniej recepturze, ale kod będzie trzeba znacznie zmodyfikować. Pogrubione wiersze kodu na listingu 2.14 wskazują, jakie zmiany należy wprowadzić w pliku *index.html*.

Listing 2.14. Modyfikacja pliku index.html

```
...
<div id="player_inventory">
  <span class="item-container"><h3>Schowek:</h3>
    <div class="inventory-box empty"></div>
    <div class="inventory-box empty"></div>
    <div class="inventory-box empty"></div>
  </span>
</div>
<div id="impress">
    <div id="slide1" class="step slide">
      <div class="slide-text">
        <p>To jest maszyna czasu. Przestawić przełącznik?</p>
        <p><a href='#slide2'>PRZEŁĄCZ</a></p>
        <p><a href='#slide3'>POCZEKAJ,</a> aż tata, który jest naukowcem, wróci
        ↪do domu</p>
        <div class="event-text"></div>
      </div>
      <div class="menu item-container"><div class="inventory-box"><img src="bat.png"
      ↪alt="bat" class="item" id="bat"></div></div>
    </div>
    <div id="slide2" class="step slide" data-x="1500">
      <div class="slide-text">
        <p>A niech to. Dinozaur. Ciekawe, czy można go czymś walnąć...</p>
        <div class="event-text"></div>
      </div>
      <div class="menu"><div class="inventory-box"><img src="dino.png" alt="dino"
      ↪class="item" id="dino"></div></div>
    </div>
    <div id="slide3" class="step slide" data-x="-1500">
      <div class="slide-text">
        <p>Poobijałeś się trochę, aż w końcu Twój tata wrócił do domu.</p>
        <p>Pomógł Ci odrobić pracę domową.</p>
        <p>Było całkiem fajnie, ale nie aż tak.</p>
        <div class="event-text"></div>
      </div>
      <div class="menu"><a href="#slide1">ZACZNIJ OD NOWA</a></div>
      </div>
</div>

<script src="impress.js"></script>
<script>impress().init();</script>
```

W tej recepturze zmieniliśmy sposób odnoszenia się do elementów. Nie używamy już klasy `itemable` do oznaczania slajdów ani schowka. Ponadto trudno jest posługiwać się elementami mającymi identyfikatory liczbowe i dlatego dodaliśmy przedrostek `slide`. W związku z tym musieliśmy też zmienić łącza do slajdów. Ponadto do każdego slajdu dodaliśmy element `div` z klasą `event-text`. W drugim slajdzie został dodany nowy obiekt zawierający tekst o dinozaurze.

Biblioteka *impress.js* jest ładowana tak samo jak poprzednio, ale trzeba ją trochę dostosować do naszych potrzeb. Zajmujemy się tym na listingu 2.15.

Listing 2.15. Zmiany w bibliotece impress.js

```
var game = {
  stepsTaken: [],
  updateAfterStep: function(stepId){
    this.stepsTaken.push(stepId);
  }
};
...
var getStep = function ( step ) {
  if (typeof step === "number") {
    step = step < 0 ? steps[ steps.length + step] : steps[ step ];
  } else if (typeof step === "string") {
    step = byId(step);
  }
  if (!!step.id === true){
    game.updateAfterStep(step.id);
  };
  return (step && step.id && stepsData["impress-" + step.id]) ? step : null;
};
...
```

W tym kodzie znalazły się dwie ważne zmiany. Pierwsza polega na utworzeniu obiektu `game`. Będzie to jedyny obiekt globalny, jakiego będziemy używać w tym rozdziale. Obiekt ten ma dwa atrybuty: tablicę do przechowywania obejrzanych slajdów oraz funkcję dodającą slajdy do tej tablicy. Kod ten może znajdować się na początku pliku. Druga zmiana to wywołanie w skrypcie *impress.js* funkcji za każdym razem, gdy zostanie otwarty slajd (drugi pogrubiony fragment na listingu 2.15 — w pobliżu wiersza 421. w pliku).

Teraz na listingu 2.16 wracamy do pliku *dragDrop.js*.

Listing 2.16. Zmiany w pliku dragDrop.js

```
(function(){
...
  function handleDrop(e) {
    var dragAppliedTo = this;
    game.things.dropItemInto(draggingObject,
    ↪dragAppliedTo.parentElement.parentElement.id);
    e.preventDefault();
  }
...
})();
```

W pliku zostały wprowadzone dwie zmiany. Po pierwsze, cały kod został umieszczony w samowykonującej się funkcji. Gdybyśmy chcieli, aby ta funkcja została wykonana więcej niż raz (by np. ponownie powiązać procedury nasłuchowe zdarzeń), funkcję tę moglibyśmy uczynić atrybutem globalnego obiektu `game`, zmieniając zawartość pierwszego wiersza na `game.dragdrop = (func` `↪tion)(){`. Ale w tym przypadku wystarczy jednorazowe wywołanie. Druga zmiana dotyczy uproszczenia funkcji `handleDrop`. Teraz przeciągany obiekt i identyfikator kontenera, do którego ten obiekt jest przenoszony, przekazujemy do funkcji `dropItemInto`, czyli metody zdefiniowanej we własności `things` globalnego obiektu `game`.

Zmiany w pliku *game.js* są rozległe i dlatego najlepiej zacząć jego budowę od nowa. Zastąp całą zawartość pliku *game.js* kodem pokazanym na listingu 2.17.

Listing 2.17. Tworzenie własności game.things

```
game.things = (function(){
  var items = {
    bat: {
      name: 'bat',
      effects: {
        'player_inventory': { message: "<p>Wziąłeś kij!</p>",
                              object: "addItem",
                              subject: "deleteItem"
        },
        'dino': { message: "<p>Uderzyłeś dinozaura kijem.</p><p>Rozzłościł się.</p>",
                  subject: 'deleteItem'
        },
        'empty': {
              message: "<p>Odłożyłeś kij.</p>",
              object: "addItem",
              subject: "deleteItem"
        }
      }
    },
    dino: {
      name: 'dino',
      effects: {
        'player_inventory': { message: "<p>Nie możesz przesunąć dinozaura...</p>" }
      }
    }
  };

  var get = function(name){
    return this.items[name];
  };
  var dropItemInto = function(itemNode, target){
    var sourceContext = itemNode.parentElement.parentElement.id;
    if(sourceContext !== target){
      var item = itemNode.firstChild.id;
      var itemObject = this.get(item);

      if (target === 'player_inventory'){
        var effects = itemObject.effects[target];
      }else if(game.slide.getInventory(target)){
        var effects = itemObject.effects[game.slide.getInventory(target)];
      }else{
        var effects = itemObject.effects['empty'];
      };

      var targetObject;
      if (!!effects.object === true){
        if(target==="player_inventory"){
          targetObject = game.playerInventory;
        }else{
          targetObject = game.slide;
        };
```

```
        targetObject[effects.object](itemObject);
      };
      if (!!effects.subject === true){
        if(sourceContext === "player_inventory"){
          var sourceObject = game.playerInventory;
        }else{
          var sourceObject = game.slide;
        };
        sourceObject[effects.subject](itemObject);
      };
      if (!!effects.message === true){
        game.slide.setText(effects.message);
      };
      game.screen.draw();
    };
  };

  return{
    items: items,
    get: get,
    dropItemInto: dropItemInto
  }
})();
```

To całkiem sporo kodu. Ogólnie mówiąc, jest to obiekt zapakowany w samowykonującą się funkcję i przypisany jako własność `things` do obiektu `game`. Jeśli spojrzysz na wartość zwrotną tej funkcji, to zauważysz coś znajomego. Zwraca ona obiekt mapujący publiczne metody na prywatne, co powoduje, że są dostępne np. poprzez notację `game.things.items`. Publiczny interfejs obiektu `game.things` zawiera teraz metodę `items` zwracającą obiekt `items`, metodę `get` zwracającą konkretny element oraz metodę `dropItemInto` wywoływaną w skrypcie *dragDrop.js*.

Wracając na początek funkcji, obiekty przedmiotów są zdefiniowane jako dane zawierające metody wykonywane w reakcji na określone zdarzenia. Jeśli chodzi o strukturę tych danych, obiekt `bat` ma własność `name` o wartości `bat`. Własność `effects` tego obiektu zawiera dane w formacie JSON określające, co się dzieje, gdy obiekt zostanie przeciągnięty na inne obiekty i miejsca w specjalnym pojemniku `player_inventory`. Element `dino` ma mniejsze mapowanie, ponieważ może być przeciągany tylko do schowka gracza. We wszystkich przypadkach są trzy potencjalne efekty: `subject` wskazuje, jaka funkcja zostanie wykonana na przeciąganym obiekcie w zależności od miejsca, z którego obiekt ten jest przeciągany. `object` określa funkcję wywoływaną na obiekcie w zależności od miejsca, w które obiekt ten jest przeciągany. Natomiast `message` zawiera tekst, który zostanie dodany do slajdu w wyniku interakcji.

Dalej znajduje się metoda `get` zwracająca element o nazwie określonej przez przekazany w wywołaniu parametr `name`.

Ostatnia funkcja to `dropItemInto`. Jest ona dość skomplikowana. Przyjmuje dwa parametry: `itemNode` i `target`. Najpierw źródło (`sourceContext`) jest porównywane z celem (`target`). Jeśli są identyczne, to znaczy, że miejsce przeciągnięcia obiektu jest takie samo jak jego źródło, i funkcja nie wykonuje żadnych dalszych działań. W konstrukcji `if else`, `if else` jest sprawdzane, które efekty danego elementu mają zostać wykonane. Następna konstrukcja `if` jest wykonywana, jeśli obiekt `effects` ma zdefiniowaną własność `object`, określa obiekt docelowy (`targetObject`), dla którego należy wykonać efekt, oraz wykonuje ten efekt. Blok rozpoczynający się od `if (!!effects.subject === true){` działa podobnie, tylko dotyczy `subject`, a nie `object`. Ostatnia konstrukcja `if` na tym poziomie wywołuje metodę `game.slide.SetText` ustawiającą

event-text bieżącego slajdu na zawartość własności message obiektu effects. Ostatni pełny wiersz funkcji dropItemInto wywołuje metodę game.screen.draw aktualizującą ekran po zmianie zawartości schowków.

Wiedząc, jak działa funkcja dropItemInto, która wykonuje większość zadań, możemy bliżej przyjrzeć się wykorzystywanym przez nią obiektom. Na listingu 2.18 jest pokazany obiekt game.slide z pliku *game.js*. Kod ten można wkleić bezpośrednio pod zawartością kodu z listingu 2.17.

Listing 2.18. Funkcja game.slide w pliku game.js

```
game.slide = (function(){
  var inventory = {
    slide1: 'bat',
    slide2: 'dino',
    slide3: null
  };
  var addItem = function(item){
    inventory[game.slide.currentSlide()] = item.name;
  };
  var deleteItem = function(item){
    inventory[game.slide.currentSlide()] = null;
  };
  var findTextNode = function(slideId){
    return document.querySelector("#" + slideId + " .slide-text .event-text");
  };
  var getInventory = function(slideId){
    return inventory[slideId];
  };
  var setText = function(message, slideId){
    if (!!slideId === false){
      slideId = currentSlide();
    }
    return findTextNode(slideId).innerHTML = message;
  };
  var currentSlide = function(){
    return game.stepsTaken[game.stepsTaken.length - 1];
  };
  var draw = function(slideId){
    if(!slideId === true){
      slideId = this.currentSlide();
    };
    var item = inventory[slideId];
    var inventoryBox = document.querySelector ('#'+slideId+' .inventory-box');
    if (item === null){
      inventoryBox.innerHTML = "";
      inventoryBox.classList.add("empty");
    }
    else{
      inventoryBox.innerHTML = "<img src='"+item+".png' alt='"+item+"' class='item'
      ↪id='"+item+"'>";
      inventoryBox.classList.remove("empty");
    }
  };

  return {
    addItem: addItem,
    deleteItem: deleteItem,
```

```
    setText: setText,
    getInventory: getInventory,
    draw: draw,
    currentSlide: currentSlide
  };
})();
```

Sposób zdefiniowania i wykonania tej własności powinien już wyglądać znajomo. Podobnie jak wcześniej, sposób działania tego obiektu można rozszyfrować na podstawie metod publicznie udostępnionych w bloku `return`. Na samym początku widać, że obiekt `inventory` zawiera nasze slajdy, których wartości są ustawione na nazwy znajdujących się wewnątrz przedmiotów lub `null`. Dalej są funkcje `addItem` i `deleteItem`, służące do zarządzania schowkiem poprzez dodawanie do slajdów nazw elementów lub ustawianie ich na `null`.

Następna w kolejności jest funkcja `findTextNode` znajdująca element `div` z klasą `event-text` określonego slajdu. Metoda ta nie jest mapowana na atrybut w bloku `return`, a więc jest prywatna, tzn. używana jedynie wewnątrz obiektu `game.slide`. Wywołuje ją tylko jedna funkcja: `setText`.

Funkcja `getInventory` zwraca element znajdujący się w określonym slajdzie na podstawie obiektu `inventory`.

Funkcja `setText` ustawia tekst określonego slajdu przy użyciu przekazanych jej parametrów `message` i `slideId`. Jeśli parametr `slideId` nie jest zdefiniowany, to domyślnie stosowany jest `currentSlide`. Osoby, które wcześniej programowały w jakimś języku z domyślnymi parametrami, mogą myśleć, że następujący kod powinien działać nawet mimo braku parametru `slideId` w wywołaniu funkcji: `function(message, slideId=currentSlide())`. Jednak ten kod nie zadziała. Jeśli potrzebne są domyślne parametry, należy przekazać obiekt i go obsłużyć. W przeciwnym razie wystarczy test na wartość `null`.

Następna jest metoda `currentSlide`, o której już wiele razy wspominałem. To właśnie z jej powodu wcześniej zmodyfikowaliśmy bibliotekę *impress.js*. Funkcja ta pobiera ostatni element z tablicy `stepsTaken`, do której nowa wartość jest zapisywana za każdym razem, gdy otwieramy jakiś slajd.

Ostatnia metoda obiektu `slide` to `draw`. Zaczyna się od znanej nam już sztuczki związanej z domyślnym parametrem. Metoda ta ustawia `slideId` na `currentSlide`. Znajduje element z klasą `inventory-box` slajdu i ustawia jego zawartość na to, co jest opisane w obiekcie `inventory`, tzn. `null` albo konkretny przedmiot. Ponadto dodaje lub usuwa klasę `empty`, zależnie od tego, czy zawiera obraz.

Pozostał nam jeszcze jeden duży obiekt w tej recepturze — `playerInventory` z pliku *game.js*. Jego kod znajduje się na listingu 2.19.

Listing 2.19. Tworzenie obiektu playerInventory

```
game.playerInventory = (function(){
  var items = {
    bat: false
  };
  var clearInventory = function(){
    playerInventoryBoxes = document.querySelectorAll('#player_inventory
    ↪.inventory-box');
    [].forEach.call(playerInventoryBoxes, function(inventoryBox) {
      inventoryBox.classList.add("empty");
      inventoryBox.innerHTML = "";
```

```
  });
};
var addItem = function(item){
  if (this.items[item.name] === false){
    this.items[item.name] = true;
  };
  return this.items;
};
var deleteItem = function(item){
  if (this.items[item.name] === true){
    this.items[item.name] = false;
  };
  return this.items;
};
var draw = function(){
  clearInventory();
  var counter = 0;
  var inventoryBoxes = document.querySelectorAll('#player_inventory
  ↪.inventory-box');
  for(var item in this.items){
    if(this.items[item] === true){
      inventoryBoxes[counter].classList.remove("empty");
      inventoryBoxes[counter].innerHTML = "<img src='"+item+".png' alt='"+item+"'
      ↪class='item' id='"+item+"'>";
    }
    counter = counter + 1;
  };
};
return {
  items: items,
  addItem: addItem,
  deleteItem: deleteItem,
  draw: draw
};
})();
```

Ponownie zastosowaliśmy ten sam wzorzec co wcześniej w tym pliku, tzn. ustawiliśmy obiekt jako atrybut i natychmiast wykonaliśmy funkcję. Nie ma tu nic zaskakującego. Również podobnie jak wcześniej, blok `return` zawiera publiczny interfejs. Przyjrzymy się temu obiektowi od początku, aby dowiedzieć się, co dokładnie robi.

Zmienna `items` służy do przechowywania przedmiotów w obiekcie. Można argumentować, że zamiast zwykłego obiektu lepiej byłoby użyć tablicy. Ale gdyby tablica była bardzo duża, przeszukiwanie jej w celu znalezienia wybranego obiektu byłoby czasochłonne. Lepiej jest znajdować indeks i sprawdzać, czy ma wartość `true`, czy `false`. W związku z tym każdy przedmiot, jaki może pojawić się w schowku gracza, powinien zostać uwzględniony w tym obiekcie i mieć wartość `false`.

Następnie prywatna funkcja `clearInventory` pobiera wszystkie obrazy z elementów `div` pól schowka gracza. Jest wywoływana przed metodą `draw`, aby wszystko wyczyścić, tzn. np. każdemu elementowi `inventoryBox` nadać klasę `empty`.

Metody `addItem` i `deleteItem` są nieciekawe. Ustawiają wartość elementu na `true` i `false` i zwracają `this`.

Metoda `draw` rozpoczyna działanie od skasowania zawartości schowka. Zdefiniowaliśmy zmienną o nazwie `counter` do użycia w pętli `for`, której składni jeszcze w tym rozdziale nie opisywałem. Do pracy z obiektami (w odróżnieniu od tablic) lepiej jest używać pętli `for...in`. Zmienna `counter` jest potrzebna do uzyskania dostępu do elementów HTML — zarówno w celu ich dodawania, jak i ustawiania klasy `empty`.

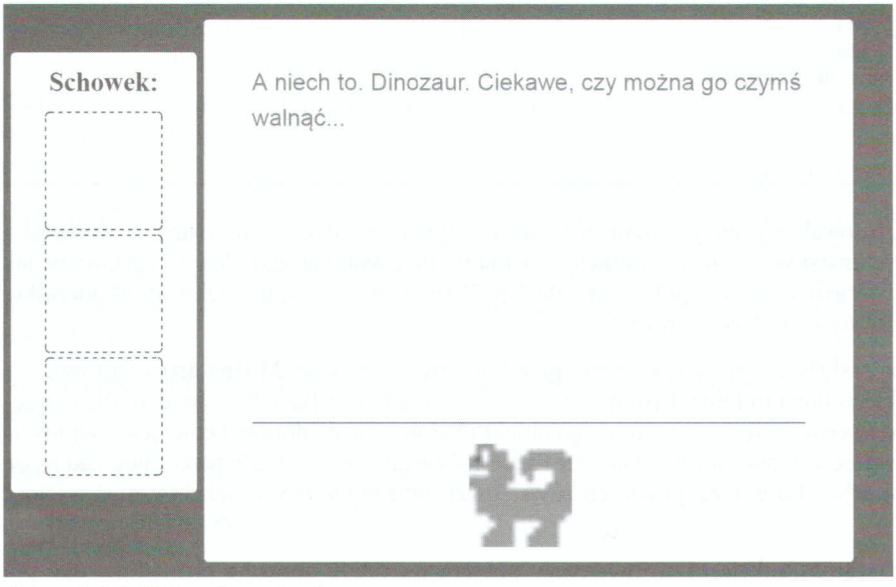
Zgodnie z tym, co napisałem wcześniej, duże obiekty są już omówione i pozostał jeszcze tylko jeden mały. Przedstawioną na listingu 2.20 własność `screen` również należy dodać do pliku *game.js*.

Listing 2.20. Dodanie własności screen do gry

```
game.screen = (function(){
  var draw = function(){
    game.playerInventory.draw();
    game.slide.draw(game.slide.currentSlide());
  };
  return {
    draw: draw
  }
})();
```

`screen` jest obiektem opakowującym dla funkcji `draw` znajdujących się w obiektach `playerInventory` i `slide`.

To wszystko, jeśli chodzi o tę recepturę. Na rysunku 2.3 widać, co się dzieje, gdy uderzymy dinozaura kijem (plik *index.html*).



Rysunek 2.3. Efekt uderzenia dinozaura

Receptura. Okruszki

Gra zaczyna nabierać rumieńców, ale nawigacja na razie zawiera tylko to, co wpiszemy bezpośrednio na slajdach. Przydałaby się nawigacja okruszkowa, aby gracz w każdej chwili mógł przejść do poprzedniego slajdu. Żeby dodać tę nawigację, musimy po raz kolejny zmodyfikować zawartość pliku *impress.js* oraz dodać trochę kodu HTML i CSS. W porównaniu z dwiema poprzednimi ta receptura jest bardzo prosta.

Zaczniemy od modyfikacji pliku *impress.js*. Nie będzie się on znacząco różnił od poprzedniej wersji. Zmiana polega na rozbudowaniu nieco metody `updateAfterSetup`. Wiersze kodu, które należy dodać, są pogrubione na listingu 2.21.

Listing 2.21. Modyfikacja pliku impress.js

```
var game = {
  stepsTaken: [],
  updateAfterStep: function(stepId){
    if (this.stepsTaken.length < 1 || stepId !==
    ↪this.stepsTaken[this.stepsTaken.length-1]){
      this.stepsTaken.push(stepId);
      var numberOfSteps = this.stepsTaken.length;
      var stepsElement = document.getElementById("steps");
      var newStep = document.createElement("li");
      newStep.innerHTML = "" + numberOfSteps +": <a href=#"+stepId+">"+stepId+"</a>";
      var mostRecentStep = stepsElement.firstChild;
      stepsElement.insertBefore(newStep, mostRecentStep);
    };
  }
}
```

Teraz zamiast dodawać krok do tablicy, metoda ta tworzy nowy element `li` z łączem do ostatnio oglądanego slajdu i dodaje go na początku listy.

W pliku *index.html* są potrzebne dwie drobne zmiany. Pierwsza to dodanie listy z identyfikatorem `stepsTaken`, a druga dotyczy zmiany odnośnika w taki sposób, aby powodował odświeżenie strony. Jako że pracujemy z przedmiotami i schowkami, rozwiązanie to jest znacznie prostsze niż kasowanie wszystkiego w interfejsie użytkownika i obiektach zaplecza. Potrzebne zmiany zostały wyróżnione pogrubieniem na listingu 2.22.

Listing 2.22. Implementacja okruszków w pliku indeks.html

```
<div id="player_inventory">
  <span class="item-container"><h3>Schowek:</h3>
    <div class="inventory-box empty"></div>
    <div class="inventory-box empty"></div>
    <div class="inventory-box empty"></div>
  </span>
</div>
<div id="steps-taken">
  <h3>Wykonane kroki:</h3>
  <ul id='steps'>
  </ul>
</div>

<div id="slide3" class="step slide" data-x="-1500">
```

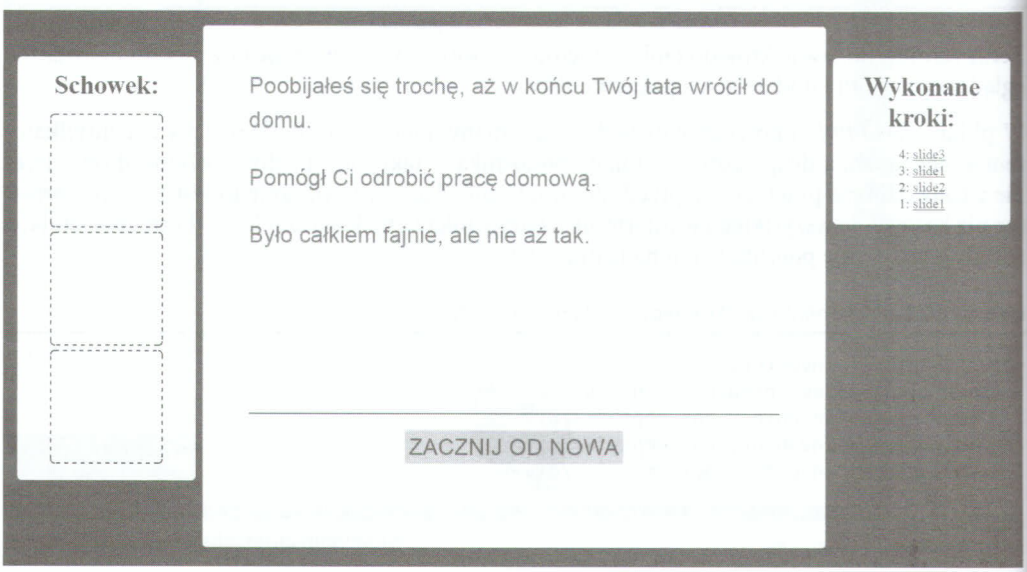
```
  <div class="slide-text">
    <p>Poobijałeś się trochę, aż w końcu Twój tata wrócił do domu.</p>
    <p>Pomógł Ci odrobić pracę domową.</p>
    <p>Było całkiem fajnie, ale nie aż tak.</p>
    <div class="event-text"></div>
  </div>
  <div class="menu"><a href="#" onclick='location = window.location.pathname; return
  ↪false;'>ZACZNIJ OD NOWA</a></div>
...
```

Ostatnia zmiana w tej recepturze dotyczy dodania kodu CSS widocznego na listingu 2.23 do pliku *main.css*.

Listing 2.23. Style okruszków

```
#steps-taken{
  position: fixed;
  text-align: center;
  width:180px;
  padding:15px;
  border-radius: 5px;
  top: 75px;
  right: 25px;
  background-color: white;
}
```

Teraz gdy otworzysz plik *index.html* i obejrzysz kilka stron, na ekranie powinieneś mieć widok podobny do pokazanego na rysunku 2.4.



Rysunek 2.4. Działanie nawigacji okruszkowej

Receptura. Dramatyczne zakończenie

Na razie w grze są dwie możliwości. Można się pobawić albo zdenerwować dinozaura. A gdyby tak dinozaur mógł się tak rozzłościć, że by nas zaatakował? Taki efekt można utworzyć przy użyciu jednej z najdziwniejszych wtyczek do jQuery o nazwie Raptorize (*http://zurb.com/playground/jquery-raptorize*).

Umieść pliki *raptor-sound.mp3* i *raptor-sound.ogg* w tym samym folderze, w którym znajduje się plik *index.html* (potrzebne są dwa pliki dźwiękowe, ponieważ przeglądarki obsługują różne kodeki audio). Ponadto w tym samym folderze umieść pliki jQuery, *jquery.raptorize* oraz *raptor.png*.

Teraz w pliku *index.html* trzeba załadować skrypty, jak pokazano na listingu 2.24.

Listing 2.24. Plik index.html z groźnym dinozaurem

```
<script src="impress.js"></script>
<script>impress().init();</script>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/
➥jquery.min.js"></script>
<script>!window.jQuery && document.write('<script src="jquery-1.4.1.min.js"><\
➥/script>')</script>
<script src="jquery.raptorize.1.0.js"></script>
<script src="game.js"></script>
<script src="dragDrop.js"></script>
```

W kodzie znalazły się trzy nowe elementy `script`. Pierwszy ładuje bibliotekę jQuery z sieci CDN Google. Takie ładowanie skryptów w aplikacjach ma pewne zalety. Drugi element `script` ładuje bibliotekę jQuery z lokalnego źródła, jeśli ładowanie z CDN się nie powiedzie (to najczęściej oznacza, że mamy problemy z połączeniem internetowym, ponieważ sieć CDN Google jest niezawodna). Trzeci element `script` ładuje skrypt *raptorize.js*.

Na listingu 2.25 pokazano, jakie zmiany należy wprowadzić w pliku *game.js*.

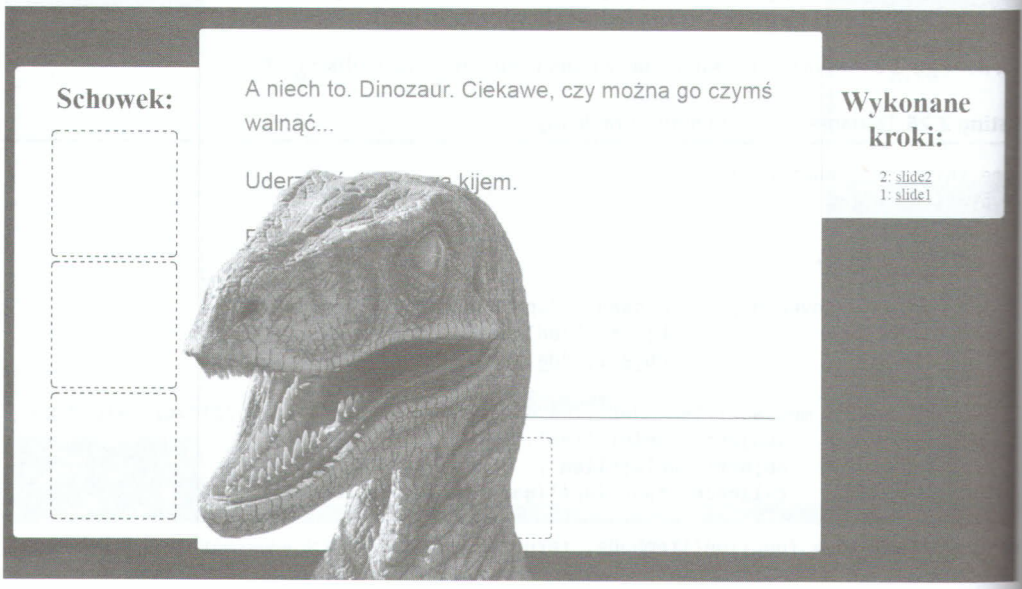
Listing 2.25. Dodanie groźnego dinozaura do gry

```
game.things = (function(){
  var items = {
    bat: {
      name: 'bat',
      effects: {
        'player_inventory': { message: "<p>Podniosłeś kij!</p>",
                              object: "addItem",
                              subject: "deleteItem"
        },
        'dino': { message: "<p>Uderzyłeś dinozaura kijem.</p><p>Rozzłościł się.</p>",
                  subject: 'deleteItem',
                  object: 'deleteItem',
                  callback: function(){game.screen.callDino()}

var dropItemInto = function(itemNode, target){

if (!!effects.message === true){
  game.slide.setText(effects.message);
};
```

```
if(!!effects.callback === true){
  effects.callback();
};
game.screen.draw();

...
game.screen = (function(){
  var draw = function(){
    game.playerInventory.draw();
    game.slide.draw(game.slide.currentSlide());
  };
  var callDino = function(){
    $('body').raptorize({ 'enterOn' : 'timer', 'delayTime' : 2000 });
  };
  return {
    callDino: callDino,
    draw: draw
  }
})();
...
```

Najpierw w obiekcie `things` zmieniliśmy efekt zastosowania kija na dinozaurze. Później sprawiliśmy, że dinozaur i kij znikają, oraz wywołaliśmy funkcję zwrotną powodującą przywołanie dinozaura. Następnie w funkcji `dropItemInto` dodaliśmy test pozwalający sprawdzić, czy istnieje funkcja zwrotna, i wykonujący ją, jeśli jest dostępna. Na końcu dodajemy funkcję `callDino` do obiektu `screen` oraz umieszczamy referencję w bloku `return`, aby była dostępna publicznie.

Jeśli poprawnie wykonałeś wszystkie opisane czynności, to teraz uderzenie dinozaura kijem bardzo rozzłości zwierza i spowoduje pojawienie się wielkiego ryczącego tyranozaura na ekranie, jak widać na rysunku 2.5.



Rysunek 2.5. Dramatyczne zakończenie gry

Podsumowanie

W ten sposób kończy się nasza wycieczka do świata interaktywnej fikcji, podczas której za przewodnika służyła nam biblioteka *impress.js*. W tym rozdziale zostało wprowadzone sporo materiału, od wzorców JavaScript, po programowanie funkcyjne i tyranozaury.

Jeśli odniosłeś wrażenie, że rozdział ten był trudny, to zapewne dlatego, że istotnie taki jest. Myślę, że jest to najtrudniejszy rozdział w całej książce. Język JavaScript czasami jest trochę skomplikowany, co wyjaśnia popularność wielu jego bibliotek takich jak jQuery. Rozdziały, których treść jest oparta na silnikach gier, będą łatwiejsze do zrozumienia. Biblioteki często mają bardziej spójne i zwarte dokumentacje, dzięki czemu łatwiej jest się nimi posługiwać. Natomiast korzystając z czystego JavaScriptu, bardzo łatwo się zgubić w gąszczu niuansów różnych specyfikacji, implementacji i rozmaitych usterek (opisanych przez Douglasa Crockforda).

Czym zająć się teraz? Jeśli chodzi o JavaScript, to powinieneś nauczyć się stosowania wzorców projektowych i programowania funkcyjnego oraz poznać odpowiednie biblioteki. Pracy wystarczy na kilka miesięcy. Od strony interfejsu użytkownika zajęcie na długi czas zapewni dokładne zgłębienie języków HTML5 i CSS3.

A co z grą? Może nie lubisz dinozaurów, może wolisz coś bardziej realnego albo wyższego i z gałęziami? Może przydałby się tekstowy interfejs albo lepsza byłaby nawigacja w stylu północ – południe, wschód – zachód, aby gracz miał wrażenie, że uczestniczy w prawdziwej przygodzie? Możesz też wykorzystać tę grę do utworzenia internetowych kartek z życzeniami, dodać więcej przedmiotów czy zakończeń albo uczynić grę bardziej straszną, śmieszniejszą lub pouczającą. Masz gotowy szablon, który możesz rozbudować w sposób, w jaki jeszcze niedawno pewnie byś nie potrafił.