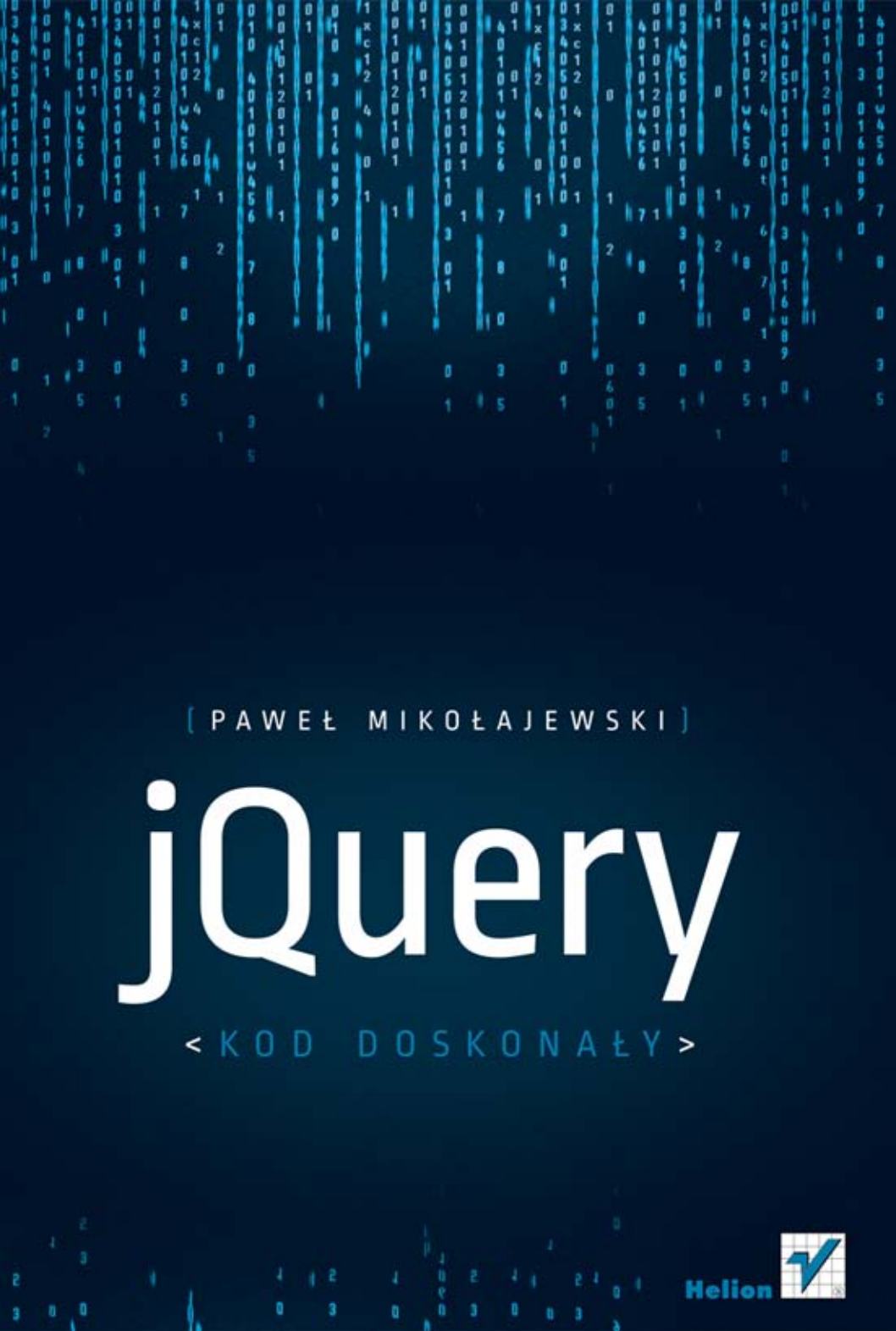


[PAWEŁ MIKOŁAJEWSKI]

jQuery

< KOD DOSKONAŁY >

Helion

Redaktor prowadzący: Ewelina Burska
Projekt okładki: Jan Paluch
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

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

ISBN: 978-83-246-5099-6

Printed in Poland.

Spis treści

Rozdział 5.
Przetestowany kod

Jeśli do tej pory nie miałeś w zwyczaju sprawdzania poprawności swojego kodu JavaScript testami, prawdopodobnie testujesz działanie strony bezpośrednio w przeglądarce, klikając elementy i oczekując na pojawienie się błędu. Kiedy napotkasz błąd, możesz go poprawić i dalej testować działanie skryptów. Jest to bardzo popularne podejście do testowania JavaScript. Przede wszystkim jest jednak błędne.

Manualne testowanie zaimplementowanych funkcjonalności to czasochłonny proces. Wymaganie, aby programista za każdym razem weryfikował działanie skryptów, opóźnia proces tworzenia aplikacji. Oczywiście ten obowiązek może przejąć tester aplikacji, wolelibyśmy jednak zredukować tego rodzaju koszty w procesie rozwoju aplikacji. Ponadto testująca osoba może przeoczyć niektóre niedociągnięcia. Nawet po poprawieniu błędów wkrótce mogą się one ujawnić ponownie, dlatego nawet na późniejszych etapach rozwoju aplikacji musisz nadal sprawdzać funkcjonalność, która została już przetestowana. Jeżeli zdarzy Ci się coś przeoczyć — wówczas błędny kod może znaleźć się w środowisku produkcyjnym aplikacji, a co za tym idzie — klienci odwiedzający stronę dostrzegą błędy. Testowanie manualne prawie zawsze prowadzi do regresji i przypadkowo ujawniających się błędów na stronie.

Rozwiązaniem tych problemów jest wprowadzenie automatycznych testów pokrywających zaimplementowany przez Ciebie kod JavaScript.

Zanim rozpoczniemy pracę z testami, musimy ustalić, w jakim języku będziemy opisywać pożądane przez nas zachowania. Przyjęło się, że powinien być to język angielski (podobnie jak w przypadku nazewnictwa zmiennych i funkcji). Jedynie treści widziane przez użytkowników aplikacji powinny ukazywać się w przeznaczonym dla nich języku. Ponadto powinieneś założyć, że kod Twojej aplikacji może być w przyszłości rozwijany przez programistę, który nie zna Twojego ojczystego języka. Zaleca się prezentowanie wyników testów w języku angielskim.

QUnit — testy jednostkowe

Biblioteka QUnit zaimplementowana przez twórców jQuery umożliwi Ci tworzenie testów jednostkowych pokrywających Twój kod JavaScript. Testy jednostkowe pozwolą Ci na testowanie metod, obiektów i zachowań aplikacji pozostających w izolacji. Dzięki temu będziesz mógł łatwo znaleźć miejsce, w którym wystąpił błąd. Kod źródłowy narzędzia dostępny jest pod adresem *https://github.com/jquery/qunit*.

Aby rozpocząć pracę z QUnit, posłużymy się szablonem HTML z listingu 5.1.

Listing 5.1. *Plik szablonu zestawu testów jednostkowych QUnit*

```
<!DOCTYPE html>
<html>
<head>
  <title>My tests</title>
  <link rel="stylesheet"
href="http://github.com/jquery/qunit/raw/master/qunit/qunit.css
" type="text/css" media="screen">
  <link rel="stylesheet"
href="http://code.jquery.com/qunit/qunit-git.css" />
  <script type="text/javascript"
src="http://github.com/jquery/qunit/raw/master/qunit/qunit.js">
</script>
  <!-- Tutaj umieść skrypty, które będziesz testować -->
  <script type="text/javascript" src="myapp.js"></script>
  <!-- Miejsce na pliki z testami -->
  <script type="text/javascript" src="tests.js"></script>
```

```
</head>
<body>
  <h1 id="qunit-header">Example QUnit Test/h1>
  <h2 id="qunit-banner"></h2>
  <div id="qunit-testrunner-toolbar"></div>
  <h2 id="qunit-userAgent"></h2>
  <ol id="qunit-tests"></ol>
</body>
</html>
```

Plik *myapp.js* zawiera kod, który chcesz przetestować. Natomiast w pliku *tests.js* zawarty jest zestaw testów jednostkowych QUnit. Pamiętaj, aby odnośniki do plików z testami zostały umieszczone dopiero po kodzie, który masz zamiar testować. Wewnątrz znacznika `body` trzymane są elementy, w których będą reprezentowane wyniki testów.

Asercje

Podstawowym narzędziem, jakim posługujemy się, pisząc testy jednostkowe, są asercje. Być może zetknąłeś się już z tym pojęciem podczas pracy z innymi językami programowania. Asercje służą do sprawdzania, czy wartość zwrócona przez kod programu jest zgodna z oczekiwaniami. Pozwala to na automatyczne wykrywanie błędów i szybkie znalezienie błędnie działającego kodu.

W bibliotece QUnit najprostsza z asercji jest realizowana przez funkcję `ok()`. Wykonuje ona sprawdzenie, czy podany argument jest prawdziwy. Przyjrzyjmy się przykładowemu testowi:

```
test( "Simple example", function() {
  var value = 10 > 5;
  ok( value, "We expect 10 to be greater than 5" );
});
```

Najpierw wywołujemy metodę `test()`, która konstruuje test jednostkowy. Pierwszym jej parametrem jest nazwa sprawdzanej funkcjonalności, zobaczysz ją potem w wynikach zestawu testów. Jako drugi parametr przyjmowana jest funkcja, w której będziesz implementować testy. W tym prostym przypadku przypisujemy do zmiennej `value` wartość wyrażenia `10 > 5` (czyli `true`). Następnie wykonywana

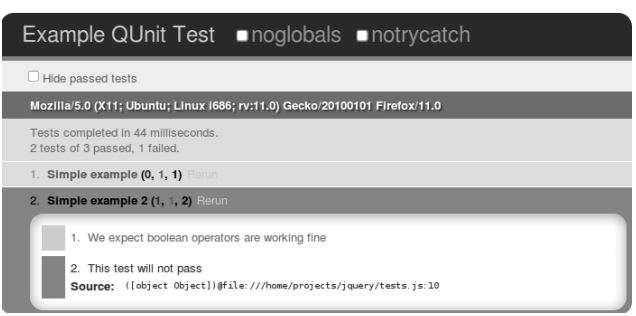
jest asercja `ok(value)`. Jeśli wartość `value` jest prawdziwa, wówczas test zakończony zostaje sukcesem. Po umieszczeniu powyższego kodu w pliku *tests.js* i uruchomieniu w przeglądarce strony z listingu 5.1 zobaczymy wynik uruchomienia testów pokazany na rysunku 5.1.

Rysunek 5.1. *Rezultat wykonania prostego testu w QUnit*

W rezultacie na stronie widzimy między innymi nazwę naszego zestawu testów, wersję przeglądarki, w której zostały one uruchomione, oraz wyniki poszczególnych testów jednostkowych. Dodajmy do naszego zestawu jeszcze jedną asercję, która tym razem nie zakończy się sukcesem:

```
test( "Simple example 2", function () {
  var value1 = true || false,
      value2 = false;
  ok( value1, "We expect boolean operators are working fine");
  ok( value2, "This test will not pass");
});
```

Rezultaty testów można zobaczyć na rysunku 5.2.



Rysunek 5.2. *Testy, które zakończyły się niepowodzeniem, są odpowiednio oznaczone*

Testy, które nie zakończyły się sukcesem, oznaczone są czerwonym kolorem. Ponadto wskazana jest konkretna asercja, która zwróciła błąd wraz z odpowiadającym jej numerem linii kodu. W takim wypadku otrzymujesz natychmiast informację zwrotną, która pozwoli Ci na poprawienie działania aplikacji (lub poprawienie testów).

Porównania

Funkcja `ok()` daje jedynie możliwość sprawdzenia, czy określona wartość jest prawdziwa. Jeżeli za jej pomocą zechcesz porównywać wartości, należałoby przekazać do niej rezultat porównania jako pierwszy argument funkcji:

```
test( "Equality test", function () {
  ok( 5 * 5 == 25, "We expect multiplication works fine");
});
```

Funkcja `ok()` jest tylko jedną z dostępnych asercji. Jeżeli interesuje Cię wynik porównania dwóch wartości, posłuż się funkcją `equal()`. Przyjmuje ona trzy argumenty. Argument pierwszy porównywany jest z drugim i jeśli nie zachodzi między nimi równość, wówczas zwracany jest błąd. Trzecim argumentem jest słowny opis asercji. Test z użyciem `equal()` może przybrać następującą postać:

```
test( "Equality test", function () {
  equal( 5 * 5, 25, "We expect multiplication works fine");
});
```

Ten test zakończy się sukcesem. Możesz również zechcieć upewnić się, że dane obiekty nie są sobie równe. W takim wypadku posłuż się asercją `notEqual()`:

```
test( "Equality test", function() {
  notEqual( 1 , 2, "We expect 1 does not equal 2");
});
```

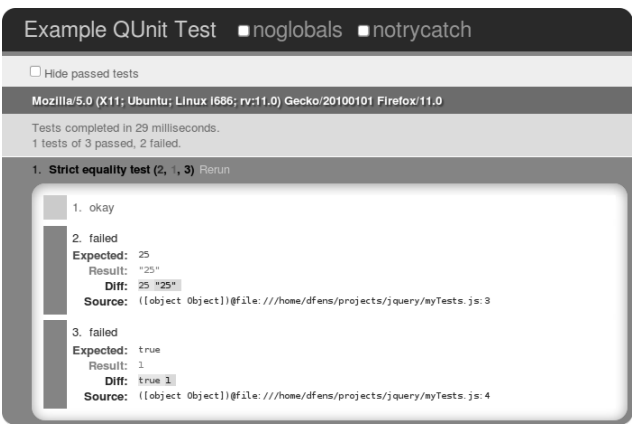
Ten test również zakończy się sukcesem, ponieważ `1 != 2`. Zwróć uwagę, że cały czas mówimy o sprawdzaniu relacji **równości**, a nie **identyczności**. Na przykład następujące asercje:

```
equal( "25", 25);
equal( 1, true);
```

będą prawdziwe i nie zwrócą błędu. Odpowiada to użyciu operatora == zamiast ===. Jeżeli porównując obiekty, chcesz uniknąć niejednoznaczności i sprawdzać przy tym ich typy, posłuż się asercją `strictEqual()`. Używana jest ona dokładnie w ten sam sposób co `equal()`:

```
test( "Strict equality test", function () {
  strictEqual( 1, 1);
  strictEqual( "25", 25);
  strictEqual( 1, true);
});
```

Wyniki powyższego testu pokazane są na rysunku 5.3.



Rysunek 5.3. *Użycie asercji strictEqual() odpowiada posłużeniu się operatorem ===*

Dostępna jest również asercja `notStrictEqual()`, która działa analogicznie jak `notEqual()`. Na przykład następujące asercje zakończą się sukcesem:

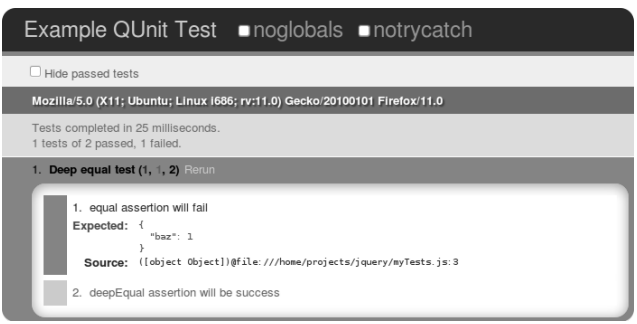
```
notStrictEqual( "25", 25);
notStrictEqual( 1, true);
```

Do Ciebie należy decyzja, której z asercji powinieneś użyć. Możesz również po prostu używać funkcji `ok()`, która za argument będzie przyjmować wynik porównania obiektów.

Jak dotąd dokonywaliśmy asercji jedynie z użyciem prostych obiektów. Aby móc porównywać złożone obiekty, posłuż się metodą `deepEqual()` (lub odpowiednio metodą `notDeepEqual()`):

```
test( "Deep equal test", function () {
  var foo = { baz: 1 }
  equal( foo, { baz: 1}, "equal assertion will fail")
  deepEqual( foo, { baz: 1}, "deepEqual assertion will be
success")
});
```

Wyniki testu pokazane są na rysunku 5.4.



Rysunek 5.4. *W przypadku porównywania złożonych obiektów posłuż się asercją deepEqual()*

Jak widać, użycie asercji `equal()` zakończyło się niepowodzeniem. Metody `equal()` oraz `strictEqual()` nie nadają się do porównywania złożonych obiektów. W ich przypadku asercja zwróci błąd.

Struktura testów

Pisanie testów jeden po drugim w sposób liniowy może utrudnić orientację w kodzie. Podobnie jeśli zawrzesz wszystkie asercje w jednym teście — wtedy testy staną się nieczytelne i trudne w utrzymaniu. Dobrze by było, abyś podzielił pliki z testami na takie, które odnoszą się do osobnych funkcjonalności kodu JavaScript. Ponadto każdy z testów powinien dotyczyć oddzielnej składowej testowanej funkcjonalności.

Aby wyniki testów były bardziej czytelne i utrzymane w pewnym porządku, QUnit udostępnia funkcję `module()`. Pozwala ona na grupowanie testów ze względu na funkcjonalność, jaką obejmują.

Jako argument przyjmuje ona nazwę aktualnie testowanego modułu. Spójrzmy na poniższy przykład:

```
module("Module A test");

test( "Basic test", function () {
  ok(1);
});

test( "Basic test 2", function () {
  ok(true);
});

module("Module B test");

test( "Another test", function () {
  equal( 5, "5");
});
```

W takim wypadku testy są podzielone na dwie grupy. Dwa pierwsze testy należą do pierwszego modułu, a trzeci test do drugiego. Pozwoli Ci to na łatwiejsze utrzymywanie porządku w kodzie. Ponadto przy każdym z wyników testu będzie dodatkowo napisane, jakiego modułu on dotyczy.

Testy asynchroniczne

Dotychczas opisywane testy wykonywane były w sposób synchroniczny. Oznacza to, że każdy kolejny test uruchamiany jest dopiero wtedy, kiedy ostatni test zostanie zakończony. W takim wypadku testowanie funkcji asynchronicznych takich jak `$.ajax()` lub `setTimeout()` zakończy się niepowodzeniem. Spójrzmy na poniższy przykład:

```
test("Asynchronous test", function () {
  setTimeout( function () {
    ok(true);
  }, 1000 );
});
```

Po uruchomieniu takiego testu otrzymujemy w rezultacie następującą informację:

```
0 tests of 1 passed, 1 failed.
...
Expected at least one assertion, but none were run - call
expect(0) to accept zero assertions.
```

Oznacza to, że asercja `ok(true)` nie została w ogóle uruchomiona, ponieważ test został ukończony, zanim doszło do jej wykonania. Aby wykonać testy w sposób asynchroniczny, musisz posłużyć się funkcjami `stop()` i `start()`. Odpowiadają one za wstrzymywanie i wznawianie procesu wykonywania testów. Omawiany test powinien przybrać następującą postać:

```
test("Asynchronous test", function () {
  setTimeout( function () {
    ok(true);
    start();
  }, 1000 );
  stop();
});
```

Kiedy test dobiega końca, uruchomiona zostaje funkcja `stop()` i testy przechodzą w etap wstrzymania. Kolejne testy nie zostaną uruchomione, dopóki nie zostanie wywołana funkcja `start()`. Umożliwi to wykonanie asercji `ok()`, która uruchomiona zostanie dopiero po upływie sekundy. Po wykonaniu asercji następuje wykonanie funkcji `start()`, co przywraca normalny bieg testów. Przypomina to nieco rozwiązanie problemu synchronizacji z użyciem semaforów.

Dostępny jest również test `asyncTest()`, który domyślnie wywołuje `stop()` na końcu testu. Dzięki temu możemy stosować nieco prostszy zapis:

```
asyncTest("Asynchronous test", function () {
  setTimeout( function () {
    ok(true);
    start();
  }, 1000 );
});
```

Podobną postać przybierze wysłanie żądania AJAX do serwera:

```
asyncTest("Asynchronous test", function () {
  $.ajax({
    url: "http://localhost/",
    success: function (data) {
      ok(true);
      start();
    }
  });
});
```

Możesz również zrezygnować z rozwiązywania tego problemu, tworząc synchroniczne żądania AJAX. W tym celu ustaw wartość pola `async` na `false`.

```
test("Asynchronous test", function () {
  $.ajax({
    url: "http://localhost/",
    async: false,
    success: function (data) {
      ok(true);
    }
  });
});
```

Możesz również przed uruchomieniem zestawu testów ustawić synchroniczne żądania AJAX jako domyślne:

```
jQuery.ajaxSetup({async:false});
```

W takim wypadku możesz zaniedbać problem synchronizacji.

Skorowidz

F

I

J

K

L

M

Z

Ż

UŻYWASZ JQUERY, ALE CHCESZ ROBIĆ TO LEPIEJ? PRZEKONAJ SIĘ, JAK MOŻESZ TO OSIĄGNĄĆ!

jQuery to biblioteka programistyczna ułatwiająca korzystanie z języka JavaScript, a jednocześnie niezwykle popularne narzędzie programistyczne, dzięki któremu można znacznie skrócić czas tworzenia skryptów i w dużej mierze uprościć ich kod. jQuery pozwala manipulować strukturą DOM, dynamicznie zmieniać zawartość stron, tworzyć animacje, obsługiwać zdarzenia, wykonywać zapytania AJAX, a także przeprowadzać wiele innych przydatnych operacji.

Duża popularność jQuery nie idzie niestety w parze z należytą znajomością tej biblioteki wśród programistów. Często postrzegają ją oni jako cudowną czarną skrzynkę, która spełnia ich oczekiwania, nie troszczą się jednak zbytnio o sposób jej działania. Ten stan rzeczy ma zmienić książka *jQuery. Kod doskonały*. Osobom mającym pewne doświadczenie w posługiwaniu się JavaScriptem i jQuery przedstawi ona mocne strony biblioteki oraz najlepsze techniki jej wykorzystania. Pomoże też poprawić wydajność pracy i efektywność kodu.

- Zasady tworzenia kodu łatwego w utrzymaniu
- Sposoby poprawiające wydajność działania skryptów
- Sztuczki umożliwiające zwiększenie elastyczności kodu
- Reguły przygotowywania i przeprowadzania testów
- Praktyczne rozwiązania ułatwiające pracę programisty

CHCESZ DOBRZE ZARZĄDZAĆ SWOIM KODEM JAVASCRIPT I PODNIEŚĆ JEGO WYDAJNOŚĆ? TA KSIĄŻKA JEST WŁAŚNIE DLA CIEBIE!

helion.pl
księgarnia internetowa

Nr katalogowy: 8695

Księgarnia internetowa
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

Cena 29,90 zł

ISBN 978-83-246-5099-6

9 788324 650996