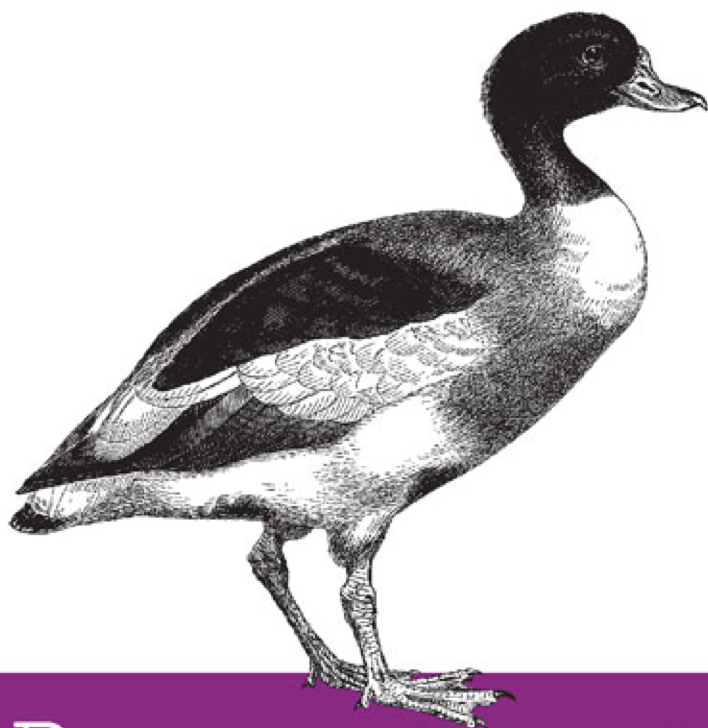
O'REILLY®



Programowanie funkcyjne Krok po kroku

ZMIEŃ SWOJE PODEJŚCIE DO PROGRAMOWANIA!

Joshua Backfield

Tytuł oryginału: Becoming Functional

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-0243-3

© 2015 Helion S.A.

Authorized Polish translation of the English edition of Becoming Functional, ISBN 9781449368173.
© 2014 Joshua Backfield.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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/pfukpk
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/pfukpk.zip

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Przedmowa 7

1. Wprowadzenie 15

Przegląd koncepcji programowania funkcyjnego 15
Typy funkcyjne 16
Funkcje czyste 16
Rekurencja 16
Zmienne niemutowalne 16
Ewaluacja nierygorystyczna 16
Instrukcje 17
Dopasowywanie do wzorca 17
Programowanie funkcyjne i współbieżność 17
Podsumowanie 18

2. Typy funkcyjne 19

Wprowadzenie do firmy XXY 19
Funkcje jako obiekty 22
Refaktoryzacja przy użyciu struktur if-else 22
Refaktoryzacja przy użyciu obiektów funkcji do wyodrębniania pól 24
Funkcje anonimowe 30
Funkcje lambda 30
Domknięcia 33
Funkcje wyższego rzędu 35
Refaktoryzacja funkcji get za pomocą języka Groovy 37
Podsumowanie 38

Kup ksi k   Pole ksi k

3. Funkcje czyste 41
Dane wyjściowe zależą od danych wejściowych 41
Oczyszczanie funkcji 45
Skutki uboczne 50
Podsumowanie 53
Przestawianie się na język Groovy 54

4. Zmienne niemutowalne 59
Mutowalność 59
Niemutowalność 65
Podsumowanie 71

5. Rekurencja 73
Wprowadzenie do rekurencji 74
Rekurencja 77
Rekurencja ogonowa 80
Refaktoryzacja funkcji countEnabledCustomersWithNoEnabledContacts 81
Podsumowanie 83
Wprowadzenie do języka Scala 84

6. Ewaluacje rygorystyczne i nierygorystyczne 87
Ewaluacja rygorystyczna 88
Ewaluacja nierygorystyczna (leniwa) 89
Leniwość może stwarzać problemy 93
Podsumowanie 96

7. Instrukcje 99
Skok na głęboką wodę 100
Proste instrukcje 100
Instrukcje blokowe 102
Wszystko jest instrukcją 104
Podsumowanie 112

Kup ksi k Pole ksi k

8. Dopasowywanie do wzorca ... 113
Proste dopasowania 113
Proste wzorce 115
Wyodrębnianie listy 118
Wyodrębnianie obiektów 120
Konwersja na dopasowywanie do wzorca 122
Podsumowanie 124

9. Funkcyjne programowanie obiektowe ... 125
Hermetyzacja statyczna 125
Obiekty jako kontenery 127
Kod jako dane 129
Podsumowanie 132

10. Podsumowanie ... 134
Od imperatywności do funkcyjności 134
Wprowadzenie funkcji wyższego rzędu 135
Konwersja istniejących metod na funkcje czyste 135
Konwersja pętli na metody rekurencyjne lub ogonoworekurencyjne 136
Konwersja zmiennych mutowalnych na niemutowalne 136
Co dalej? 136
Nowe wzorce projektowe 137
Przekazywanie komunikatów dla osiągnięcia współbieżności 137
Wzorzec Opcja (rozszerzenie wzorca Pusty Obiekt) 137
Czystość metody singletona z zachowaniem obiektowości 138
Wszystko razem 139
Podsumowanie 147

Skorowidz ... 149

Kup ksi k Pole ksi k

Kup ksi k Pole ksi k

ROZDZIAŁ 7.

Instrukcje

Kiedy myślimy o **instrukcji**, mamy na myśli coś takiego jak `Integer x = 1` lub `val x = 1`, gdzie ustawiana jest zmienna. Technicznie rzecz biorąc, ewaluacja tego wiersza nie daje żadnej wartości. Co jednak, jeśli mielibyśmy już zdefiniowaną zmienną i ustawialibyśmy ją później, na przykład za pomocą instrukcji `x = 1`? Niektórzy już wiedzą, że w językach C i Java ta instrukcja rzeczywiście zwraca wartość `1`, tak jak zostało to przedstawione w listingu 7.1.

Listing 7.1. Prosta instrukcja przypisania

```
public class Test {

  public static void main(String[] args) {
    Integer x = 0;
    System.out.println("X wynosi " + (x = 1).toString());
  }
}
```

Instrukcje w programowaniu funkcyjnym wprowadzają koncepcję polegającą na tym, że każdy wiersz kodu powinien mieć wartość zwracaną. Języki imperatywne takie jak Java zawierają koncepcję **operatora trójargumentowego** (ang. ***ternary operator***). Daje to strukturę `if-else`, która przeprowadza ewaluację do pewnej wartości. W listingu 7.2 zostało przedstawione proste użycie operatora trójargumentowego.

Listing 7.2. Prosta instrukcja trójargumentowa

```
public class Test {

  public static void main(String[] args) {
    Integer x = 1;
    System.out.println("X wynosi: " + ((x > 0) ? "dodatnie" : "ujemne"));
  }
}
```


Kup ksi k   Pole ksi k

Gdybyśmy mogli zrobić większy użytek z instrukcji, moglibyśmy zmniejszyć liczbę posiadanych zmiennych. Jeśli ograniczymy liczbę zmiennych, to zredukujemy możliwości ich mutowania, przez co zwiększymy możliwość wykonywania procesów współbieżnych *oraz* osiągnięcia większej funkcyjności!

Skok na głęboką wodę

Twój szef jest bardzo zadowolony z Twoich dokonań w XXY. Jest naprawdę pod wrażeniem programowania funkcyjnego i chce, abyś dokonał konwersji z języka częściowo funkcyjnego na język w pełni funkcyjny. Nie powinno to być trudne, ponieważ przez kilka ostatnich rozdziałów osiągnęliśmy już dość duży stopień funkcyjności.

Wybierzemy język, który działa na **maszynie wirtualnej Javy** (ang. ***Java Virtual Machine*** — JVM), aby nie wprowadzać nowych technologii, takich jak środowisko uruchomieniowe LISP lub Erlang. Moglibyśmy również wybrać języki takie jak Clojure lub Erjang, ale dla celów tej książki użyjemy języka Scala, który ma składnię podobną jak Java i nie wymaga długiej nauki.

Proste instrukcje

Przepiszemy każdą z naszych klas, zacznijmy więc od najprostszego pliku, czyli klasy `Contact`. Przypomnijmy istniejący plik w listingu 7.3.

Listing 7.3. Plik Contact.groovy

```
public class Contact {

        public final Integer contact_id = 0;
        public final String firstName = "";
        public final String lastName = "";
        public final String email = "";
        public final Boolean enabled = true;

        public Contact(Integer contact_id,
                       String firstName,
                       String lastName,
                       String email,
                       Boolean enabled) {
                this.contact_id = contact_id;
                this.firstName = firstName;
                this.lastName = lastName;
                this.email = email;
                this.enabled = enabled;
        }
```


Kup ksi k Pole ksi k

```
    public static List<Customer> setNameAndEmailForContactAndCustomer(
                    Integer customer_id,
                    Integer contact_id,
                    String name,
                    String email) {
            Customer.updateContactForCustomerContact(
                    customer_id,
                    contact_id,
                    { contact ->
                            new Contact(
                                    contact.contact_id,
                                    contact.firstName,
                                    name,
                                    email,
                                    contact.enabled
                            )
                    }
            )
    }
    public void sendEmail() {
            println("Wysyłanie wiadomości e-mail")
    }
}
```

Zrefaktoryzujemy ten kod na odpowiednik w języku Scala, tak jak zostało to przedstawione w listingu 7.4. Zwróć uwagę, że w kodzie w języku Scala definiujemy zmienne instancji w zestawie nawiasów obok nazwy klasy. Mamy również **obiekt** i **klasę**. **Statyczne** metody i składowe znajdują się wewnątrz definicji *obiektu*, a nie *klasy*. Typy definiowane są także raczej *po* niej, a nie *przed* nią.

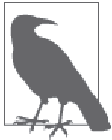
Listing 7.4. Plik Contact.scala

```
object Contact {

    def setNameAndEmailForContactAndCustomer(
      customer_id : Integer,
            contact_id : Integer,
            name : String,
            email : String) : List[Customer] = {
    Customer.updateContactForCustomerContact(
        customer_id,
        contact_id,
        { contact =>
                new Contact(
          contact.contact_id,
          contact.firstName,
          name,
          email,
          contact.enabled
        )
      }
```


Kup książkę Poleć książkę

```
    )
  }
}
class Contact(val contact_id : Integer,
              val firstName : String,
              val lastName : String,
              val email : String,
              val enabled : Boolean) {
  def sendEmail() = {
        println("Wysyłanie wiadomości e-mail")
  }
}
```



Chociaż dla czytelności w tej książce dodawanych jest wiele wierszy, w tym wierszy pustych i definicji metod podzielonych na kilka wierszy, liczba linii kodu spada z 19 do 9. Wynika to ze sposobu, w jaki w języku Java definiujemy składowe i ustawiamy je za pomocą konstruktora.

Instrukcje blokowe

Kolejną klasą, z którą się zmierzymy, jest `Contract`. Jest to nieco trudniejsze, ponieważ używaliśmy obiektu Javy `Calendar`, który nie jest konstruktem zbyt funkcyjnym. Rzućmy okiem na oryginalny plik w listingu 7.5.

Listing 7.5. Plik Contract.groovy

```
import java.util.List;
import java.util.Calendar;

public class Contract {

        public final Calendar begin_date;
        public final Calendar end_date;
        public final Boolean enabled = true;

        public Contract(Calendar begin_date, Calendar end_date, Boolean enabled) {
                this.begin_date = begin_date;
                this.end_date = end_date;
                this.enabled = enabled;
        }

        public Contract(Calendar begin_date, Boolean enabled) {
                this.begin_date = begin_date;
                this.end_date = this.begin_date.getInstance();
                this.end_date.setTimeInMillis(this.begin_date.getTimeInMillis());
                this.end_date.add(Calendar.YEAR, 2);
                this.enabled = enabled;
        }
```


Kup ksi k Pole ksi k

```
        public static List<Customer> setContractForCustomerList(
                List<Integer> ids,
                Boolean status) {
            Customer.updateContractForCustomerList(ids) { contract ->
                new Contract(contract.begin_date, contract.end_date, status)
            }
        }
}
```

Przejdźmy dalej i przekonwertujmy tę klasę, tak jak zostało to przedstawione w listingu 7.6. Spójrzmy najpierw na fragment `List[Integer]`, który przedstawia sposób oznaczania typizowania uogólnionego w Scali. Widzimy również bardzo interesującą składnię `def this(begin_date : Calendar, enabled : ↪Boolean)`, za pomocą której definiujemy konstruktor alternacyjny. Istnieje także wiersz, który zawiera tylko wartość `c`. To poprawne, gdyż wiersz ten traktowany jest jako *instrukcja*, czyli uznawany jest następnie za wartość zwracaną tego bloku kodu.

Listing 7.6. Plik Contract.scala

```
import java.util.Calendar

object Contract {

  def setContractForCustomerList(ids : List[Integer],
                                 status : Boolean) : List[Customer] = {
      Customer.updateContractForCustomerList(ids, { contract =>
    new Contract(contract.begin_date, contract.end_date, status)
   })
  }
}
class Contract(val begin_date : Calendar,
               val end_date : Calendar,
               val enabled : Boolean) {
  def this(begin_date : Calendar, enabled : Boolean) = this(begin_date, {
    val c = Calendar.getInstance()
    c.setTimeInMillis(begin_date.getTimeInMillis)
    c.add(Calendar.YEAR, 2)
    c
  }, enabled)
}
```

Najbardziej interesujące w tej składni jest wywołanie słowa kluczowego `this`, w którym przekazujemy to, co zdaje się być funkcją, tam, gdzie przekazywana powinna być zmienna `end_date`. Dlaczego kompilator nie narzeka, że oczekiwana jest instancja `Calendar`, a nie metoda, która zwraca instancję `Calendar`?

Kompilator inferuje, że nie przekazujesz metody, ale zamiast tego chcesz *przeprowadzić ewaluację* nawiasów `{...}`. Dlatego gdy wywołany jest konstruktor

Kup ksi k Pole ksi k

alternacyjny, wywołujemy rzeczywisty konstruktor, a ewaluacja nawiasów `{...}` daje nam `end_date` typu `Calendar`. Konstruktory alternacyjne działają w podobny sposób, w jaki Java pozwala przeciążać konstruktory, aby przyjmowały różne argumenty.

Blok kodu przedstawiony w listingu 7.7 jest bardzo prosty. Tworzy obiekt `Calendar`, ustawiając czas w milisekundach na podstawie obiektu `begin_date` (przypomina to domknięcie). Następnie do daty dodawane są dwa lata, aby utworzyć datę dwa lata późniejszą wobec momentu zawarcia kontraktu. Na koniec zwracany jest nowo utworzony obiekt `c`, zawierający datę dwa lata późniejszą od daty początkowej `begin_date`.

Listing 7.7. Blok kodu określający wartość dla end_date

```
{
  val c = Calendar.getInstance()
  c.setTimeInMillis(begin_date.getTimeInMillis)
  c.add(Calendar.YEAR, 2)
  c
}
```

Ta instrukcja pozwala nam wyjść poza standardowy paradygmat funkcyjny, w którym każda linia kodu powinna być instrukcją możliwą do bezpośredniego przekazania do innej funkcji lub użycia. Można traktować to jako instrukcję złożoną: mamy kilka instrukcji, które muszą być poddane ewaluacji, aby uzyskać faktycznie wykorzystywaną instrukcję ogólną.

Ten blok kodu jest interesujący, ponieważ pokazuje, że całkiem dosłownie wszystko jest instrukcją. Ostatni wiersz (`c`) jest instrukcją, gdyż zwraca zmienną `c`. Także cały blok kodu jest sam w sobie instrukcją: po poddaniu ewaluacji wykonuje linie kodu w sekwencji i zwraca nową wartość `c`, którą zdefiniowaliśmy.

Wszystko jest instrukcją

W końcu zamierzamy przekonwertować klasę `Customer`, co nie powinno być zbyt trudne. Spójrzmy na oryginalny plik Groovy przedstawiony w listingu 7.8.

Listing 7.8. Plik Customer.groovy

```
import java.util.ArrayList;
import java.util.List;
import java.util.Calendar;
```


Kup ksi k Pole ksi k

```
public class Customer {

    static public List<Customer> allCustomers = new ArrayList<Customer>();
    public final Integer id = 0;
    public final String name = "";
    public final String state = "";
    public final String domain = "";
    public final Boolean enabled = true;
    public final Contract contract = null;
    public final List<Contact> contacts = new ArrayList<Contact>();
    @Lazy public List<Contact> enabledContacts = contacts.findAll { contact ->
        contact.enabled
    }

    public Customer(Integer id,
                    String name,
                    String state,
                    String domain,
                    Boolean enabled,
                    Contract contract,
                    List<Contact> contacts) {
        this.id = id;
        this.name = name;
        this.state = state;
        this.domain = domain;
        this.enabled = enabled;
        this.contract = contract;
        this.contacts = contacts;
    }

    static def EnabledCustomer = { customer -> customer.enabled == true }
    static def DisabledCustomer = { customer -> customer.enabled == false }

    public static List<String> getDisabledCustomerNames() {
        Customer.allCustomers.findAll(DisabledCustomer).collect({customer ->
            customer.name
        })
    }

    public static List<String> getEnabledCustomerStates() {
        Customer.allCustomers.findAll(EnabledCustomer).collect({customer ->
            customer.state
        })
    }

    public static List<String> getEnabledCustomerDomains() {
        Customer.allCustomers.findAll(EnabledCustomer).collect({customer ->
            customer.domain
        })
    }

    public static List<String> getEnabledCustomerSomeoneEmail(String someone) {
        Customer.allCustomers.findAll(EnabledCustomer).collect({customer ->
```


Kup ksi k Pole ksi k

```
            someone + "@" + customer.domain
        })
    }

    public static ArrayList<Customer> getCustomerById(
            ArrayList<Customer> inList,
            final Integer id) {
        inList.findAll({customer -> customer.id == id })
    }

    public static void eachEnabledContact(Closure cls) {
        Customer.allCustomers.findAll { customer ->
            customer.enabled && customer.contract.enabled
        }.each { customer ->
            customer.contacts.each(cls)
        }
    }

    public static List<Customer> updateCustomerByIdList(
            List<Customer> initialIds,
            List<Integer> ids,
            Closure cls) {
        if(ids.size() <= 0) {
            initialIds
        } else if(initialIds.size() <= 0) {
            []
        } else {
            def idx = ids.indexOf(initialIds[0].id)
            def cust = idx >= 0 ? cls(initialIds[0]) : initialIds[0]
            [cust] + updateCustomerByIdList(
                initialIds.drop(1),
                idx >= 0 ? ids.minus(initialIds[0].id) : ids,
                cls
            )
        }
    }

    public static List<Customer> updateContactForCustomerContact(
            Integer id,
            Integer contact_id,
            Closure cls) {
        updateCustomerByIdList(Customer.allCustomers, [id], { customer ->
            new Customer(
                customer.id,
                customer.name,
                customer.state,
                customer.domain,
                customer.enabled,
                customer.contract,
                customer.contacts.collect { contact ->
                    if(contact.contact_id == contact_id) {
                        cls(contact)
                    } else {
```


Kup ksi k Pole ksi k

```
                    contact
                }
            }
        )
    })
}

public static List<Customer> updateContractForCustomerList(
        List<Integer> ids,
        Closure cls) {
    updateCustomerByIdList(Customer.allCustomers, ids, { customer ->
        new Customer(
            customer.id,
            customer.name,
            customer.state,
            customer.domain,
            customer.enabled,
            cls(customer.contract),
            customer.contacts
        )
    })
}

public static def countEnabledCustomersWithNoEnabledContacts = {
                List<Customer> customers, Integer sum ->
    if(customers.isEmpty()) {
        return sum
    } else {
        int addition = (customers.head().enabled &&
            (customers.head().contacts.find({ contact ->
                contact.enabled
            }) == null)) ? 1 : 0
        return countEnabledCustomersWithNoEnabledContacts.trampoline(
            customers.tail(),
            addition + sum
        )
    }
}.trampoline()
}
```

Kiedy konwertujemy tę klasę i obiekt na język Scala (patrz: listing 7.9), jedna rzecz nie działa: nie ma operatora trójargumentowego! Przypomnij sobie konstrukcję `(warunek) ? true : false ?`. Jak widać w pliku Scali, zastąpiliśmy ją prawdziwą instrukcją `if`.

Listing 7.9. Plik Customer.scala

```
object Customer {

  val allCustomers = List[Customer]()

  def EnabledCustomer(customer : Customer) : Boolean = customer.enabled == true
```


Kup ksi k   Pole ksi k

```
def DisabledCustomer(customer : Customer) : Boolean = customer.enabled ==
↪false

def getDisabledCustomerNames() : List[String] = {
  Customer.allCustomers.filter(DisabledCustomer).map({ customer =>
    customer.name
  })
}

def getEnabledCustomerStates() : List[String] = {
  Customer.allCustomers.filter(EnabledCustomer).map({ customer =>
    customer.state
  })
}

def getEnabledCustomerDomains() : List[String] = {
  Customer.allCustomers.filter(EnabledCustomer).map({ customer =>
    customer.domain
  })
}

def getEnabledCustomerSomeoneEmail(someone : String) : List[String] = {
  Customer.allCustomers.filter(EnabledCustomer).map({ customer =>
    someone + "@" + customer.domain
  })
}

def getCustomerById(inList : List[Customer],
  customer_id : Integer) : List[Customer] = {
    inList.filter(customer => customer.customer_id == customer_id)
}

def eachEnabledContact(cls : Contact => Unit) {
  Customer.allCustomers.filter({ customer =>
    customer.enabled && customer.contract.enabled
  }).foreach({ customer =>
    customer.contacts.foreach(cls)
  })
}

def updateCustomerByIdList(initialIds : List[Customer],
                          ids : List[Integer],
                          cls : Customer => Customer) : List[Customer] = {
  if(ids.size <= 0) {
     initialIds
  } else if(initialIds.size <= 0) {
    List()
  } else {
    val precust = initialIds.find(cust => cust.customer_id == ids(0))
    val cust = if(precust.isEmpty) { List() } else { List(cls(precust.get)) }
    cust ::: updateCustomerByIdList(
      initialIds.filter(cust => cust.customer_id == ids(0)),
      ids.drop(1),
```


Kup ksi k Pole ksi k

```
      cls
    )
  }
}

def updateContactForCustomerContact(customer_id : Integer,
                                    contact_id : Integer,
                                    cls : Contact => Contact) :
                                    ↪List[Customer] = {
updateCustomerByIdList(Customer.allCustomers, List(customer_id), { customer =>
  new Customer(
    customer.customer_id,
    customer.name,
    customer.state,
    customer.domain,
    customer.enabled,
    customer.contract,
    customer.contacts.map { contact =>
      if(contact.contact_id == contact_id) {
        cls(contact)
      } else {
        contact
      }
    }
  )
})
}

def updateContractForCustomerList(ids : List[Integer],
                                  cls : Contract => Contract) :
                                  ↪List[Customer] = {
  updateCustomerByIdList(Customer.allCustomers, ids, { customer =>
    new Customer(
      customer.customer_id,
      customer.name,
      customer.state,
      customer.domain,
      customer.enabled,
      cls(customer.contract),
      customer.contacts
    )
  })
}

def countEnabledCustomersWithNoEnabledContacts(customers : List[Customer],
                                               sum : Int) : Integer = {
  if(customers.isEmpty) {
    sum
  } else {
    val addition = if(customers.head.enabled &&
                      customers.head.contacts.exists({ contact =>
                                 contact.enabled
                              })) {
```


Kup ksi k   Pole ksi k

```
          1
        } else {
          0
        }
      countEnabledCustomersWithNoEnabledContacts(customers.tail, addition + sum)
    }
  }
}
class Customer(val customer_id : Integer,
               val name : String,
               val state : String,
               val domain : String,
               val enabled : Boolean,
               val contract : Contract,
               val contacts : List[Contact]) {
}
```

Scala nie zawiera koncepcji trójargumentowych, ponieważ wszystko *jest* już instrukcją. Oznacza to, że ewaluacja instrukcji `if` da jakąś wartość. Możemy napisać `if(warunek) { true } else { false }`, a ewaluacja instrukcji `if` da nam wartość `true` lub `false`.

Spójrzmy teraz na kod w listingu 7.10, który przedstawia sposób, w jaki możemy ustawić zmienną na podstawie instrukcji `if`.

Listing 7.10. Zwrócony rezultat instrukcji if

```
val addition = if(customers.head.enabled &&
   customers.head.contacts.exists({ contact => contact.enabled })) {
     1
   } else {
     0
   }
```

Jak widać, zmienna `addition` otrzyma wartość `1` lub `0` w zależności od ewaluacji instrukcji `if`. Dlaczego jest to o wiele bardziej interesujące niż operator trójargumentowy? Dlatego, że w tym przypadku `if` działa jak normalna instrukcja `if`, co oznacza, iż można dodać dowolną ilość kodu wewnątrz sekcji `true` lub `false` instrukcji `if`. Operator trójargumentowy tak naprawdę dopuszcza stosowanie tylko bardzo prostych wyrażeń, takich jak wartość lub podstawowe wywołanie metody.

Co jednak tak naprawdę znaczy stwierdzenie „wszystko jest instrukcją”? Oznacza to, że wszystko powinno ewaluować do jakiejś wartości. Ale co *to* dokładnie znaczy? Wielu z nas zna standardową metodologię **ziarna** (ang. ***bean***) w języku Java, która polega na posiadaniu zmiennej składowej z metodami zwracającymi i ustawiającymi. Oczywiście metoda zwracająca zwraca jakąś wartość, ale co z metodą ustawiającą? Rzućmy okiem na listing 7.11.

Kup ksi k Pole ksi k

Listing 7.11. Metoda ustawiająca dla pola Foo w klasie Bar, która zwraca sam obiekt

```
public class Bar {
  public Bar setFoo(Foo foo) { this.foo = foo; return this; }
  public Foo getFoo() { return this.foo; }
}
```

Umożliwia to łańcuchowanie wywołań funkcji i ustawianie kilku składowych w jednym wierszu, tak jak zostało to przedstawione w listingu 7.12. Ale dlaczego chcemy to zrobić? Po prostu w ten sposób możemy przedefiniować metody ustawiające i utworzyć zmienne niemutowalne. Dlaczego? Ponieważ wewnątrz metod ustawiających możemy utworzyć nową instancję `Bar` z nową wartością i zwrócić ją! Oznacza to, że implementacja zmiennych niemutowalnych staje się prostsza.

Listing 7.12. Metoda łańcuchowania w obiekcie Bar

```
return bar.setFoo(newFoo).setBaz(newBaz).setQux(newQux);
```

A co z elementami takimi jak pętle `for` — czy to też są instrukcje? Właściwie tak, ale nie w taki sposób jak można sobie wyobrażać. Pętle `for` przyjmują na ogół dwie postacie: normalnej pętli i **wyrażenia** (ang. ***comprehension***). Pierwszy typ pętli został przedstawiony w listingu 7.13.

Listing 7.13. Przykład podstawowej pętli for w języku Scala

```
val x = for(i <- 0 until 10) {
  println(i)
}
```

Uruchomienie tego kodu powoduje wyświetlenie na ekranie liczb od `0` do `9`. Co ważniejsze, dla zmiennej `x` ustawiana jest jakaś wartość — w tym przypadku jest to wartość `Unit`.

Może się to wydawać dziwne, ale w języku Scala `Unit` jest właściwie typem `void` (czyli nie ma faktycznego typu). Oznacza to, że ewaluacja naszej pętli `for` w rzeczywistości nie zwróciła żadnej wartości. Czym więc są wyrażenia? Przyjrzyjmy się wyrażeniu `for` w listingu 7.14.

Listing 7.14. Podstawowe wyrażenie for w języku Scala

```
val x = for(i <- 0 until 10) yield {
  i*2
}
```


Kup ksi k Pole ksi k

Mamy zmienną `x`, która jest listą parzystych liczb z zakresu od `0` do `18`. Wyrażenie pozwala nam wygenerować nową listę jakichś elementów lub czasem iterować przez inną listę. Spójrzmy na listing 7.15, w którym faktycznie przeprowadzamy iterację przez inną listę.

Listing 7.15. Wyrażenie for dla innej listy w języku Scala

```
val x = for(i <- List(1,2,3,4)) yield {
  i*2
}
```

Jaka jest więc różnica między tym a wykorzystaniem dla listy funkcji `map`? Przyjrzyjmy się listingowi 7.16. Ta funkcjonalność jest taka sama jak wyrażenie `for` przedstawione w listingu 7.15.

Listing 7.16. Wywołanie map dla listy w języku Scala

```
val x = List(1,2,3,4).map({ i => i*2 })
```

W takim razie kiedy należy użyć funkcji `map`, a kiedy wyrażenia? Zasadniczo funkcja `map` jest dobra, jeśli masz już listę i musisz przeprowadzić na niej operację. Wyrażenia `for` sprawdzają się, jeśli budujemy listę lub chcemy przeprowadzić określoną operację *n* razy.

Podsumowanie

Poświęciliśmy nieco czasu na przeprowadzenie migracji z języka Java do języka Scala, podkreślając nasze przejście na język funkcyjny, z którego będziemy mogli korzystać w kolejnych rozdziałach. Instrukcje pozwalają zredukować niektóre podstawowe fragmenty kodu, a czasem są konieczne, aby nadal korzystać z określonych paradygmatów *ziarna* Javy. Na przykładach takich jak obiekt `Calendar` zobaczyliśmy, że gdy musimy użyć metod ustawiających, możemy utworzyć instrukcje bloku, aby skonfigurować obiekt `Calendar`.

Instrukcje pokazują nam również, że każda metoda (nawet metody ustawiające) powinna mieć jakąś formę wartości zwracanej. Jeśli mamy metody ustawiające, które są instrukcjami, możemy łatwiej implementować zmienne niemutowalne. Dzięki instrukcjom nasz kod jest też bardziej zwięzły, ponieważ zmuszają nas one do zastanowienia się, dlaczego piszemy konkretny wiersz kodu i co powinien on reprezentować po ewaluacji. W ten sposób możemy lepiej zrozumieć, dlaczego wiersz kodu działa tak, a nie inaczej.

Kup ksi k Pole ksi k

Skorowidz

A

adnotacja @Lazy, 90, 92, 94

B

baza danych, 65, 139
bean, *Patrz:* ziarno
bezpieczeństwo wątków, 92

C

closure, *Patrz:* domknięcie

D

domknięcie, 30, 32, 33, 35, 42, 62
Don't Repeat Yourself, *Patrz:* zasada DRY

E

efekt uboczny, 41
ekspresyjność, 135
ekstraktor, 118
Erjang, 100
ewaluacja
- leniwa, *Patrz:* ewaluacja nierygorystyczna
- nierygorystyczna, 15, 16, 87, 88, 89
- rygorystyczna, 87, 88
- statyczna, 89

F

first-class function, *Patrz:* typ funkcyjny
funkcja, 8
- anonimowa, 30
- czysta, 15, 16, 41, 45, 135
- ekspresyjna, 135
- filter, 44
- findAll, 48
- getCustomerById, 45
- hermetyzacja, 24, 27
- jako obiekt, 21, 22
- lambda, 30
- lista parametrów, 22, 30
- łańcuchowanie wywołań, 111, 131
- nazwa, 22, 30
- nienazwana, 30
- println, 16
- przekazywanie do funkcji, 25, 27
- rekurencyjna, *Patrz:* rekurencja
- wartość zwracana, 22, 30
- wyższego rzędu, 135

G

generic typing, *Patrz:* typizowanie uogólnione
Groovy, 19, 37, 48, 74, 80, 90, 92, 135
- składnia, 38

guard, *Patrz:* strażnik

H

hermetyzacja, 17
- statyczna, 125

Hibernate, 97

Kup ksi k   Pole ksi k

I

immutable variable, *Patrz:* zmienna niemutowalna
instrukcja, 16, 17, 99, 104, 110
  blokowa, 102
  ewaluacja, 8
  if, 8, 114
    konwersja na dopasowywanie do wzorca, 116, 122
  match, 114
interfejs Runnable, 24

J

Java ziarno, *Patrz:* ziarno
język
  Clojure, *Patrz:* Clojure
  Erjang, *Patrz:* Erjang
  Groovy, *Patrz:* Groovy
  Scala, *Patrz:* Scala
JVM, 100

K

komunikat, 17, 137
konstruktor, 104
krotka, 115, 117

L

LISP, 100
lista
  głowa, 75, 118
  mapowanie, 67
  niemutowalna, 65
  ogon, 75, 118
  pusta, 48
  rozłożona, 118

M

makro, 22
mapowanie obiektowo--relacyjne, *Patrz:* ORM
maszyna wirtualna Javy, *Patrz:* JVM
metoda
  singletona, 137, 138
  statyczna, 101, 139
  ustawiająca, 66

N

niemutowalność, 65, 76, 88
nieważność, 43
nonstrict evaluation, *Patrz:* ewaluacja nierygorystyczna
notacja tablicowa, 8
Null Object, *Patrz:* wzorzec projektowy Pusty Obiekt
nullity, *Patrz:* nieważność

O

obiekt, 125
  jako kontener, 127
object-oriented programming, *Patrz:* OOP
OOP, 125, 138
operator
  ::, 118
  sigma, 9
  trójargumentowy, 79, 85, 99, 107, 110
ORM, 97

P

pattern matching, *Patrz:* wzorzec dopasowywanie
programowanie
  funkcyjne, 10, 15, 100, 104, 134
  imperatywne, 9
  obiektowe, 10, *Patrz:* OOP
przetwarzanie równoległe, 17
przypadek końcowy, 73, 75, 82
pure function, *Patrz:* funkcja czysta

R

rachunek lambda, 25, 30
recursion, *Patrz:* rekurencja
refaktoryzacja
  Groovy, 37
  if-else, 22
  obiekt funkcji do wyodrębniania pól, 24
rekurencja, 15, 16, 73, 74, 77, 78, 81, 137
  ogonowa, 80, 136
  Scala, 84

Kup ksi k Pole ksi k

S

Scala, 19, 84, 100, 135, 139
  składnia, 85
setter, *Patrz:* metoda ustawiająca
side effects, *Patrz:* skutki uboczne
skutki uboczne, 16, 50, 53
  implementacja, 50
słowo kluczowe
  case, 114
  match, 114
  this, 103
  volatile, 92
statement, *Patrz:* instrukcja
static evaluation, *Patrz:* ewaluacja statyczna
stos, 74, 79
strażnik, 124
sumowanie, *Patrz:* operator sigma
symbol zastępczy, 65

T

tail recursion, *Patrz:* rekurencja ogonowa
ternary operator, *Patrz:* operator trójargumentowy
trampolina, 80
transakcja bazy danych, 65
tuple, *Patrz:* krotka
typ
  bezpieczeństwo, 24
  funkcyjny, 15, 16, 19, 22
  zwracany, 27
typizowanie
  uogólnione, 26, 27

W

wartość null, 48, 77, 137
wątek, 137
  bezpieczeństwo, 92
  pula, 137
wiersz poleceń, 129
współbieżność, 17, 100, 137
wyjątek, 24
wyrażenie regularne, 113
wzorzec, 114, 121
  dopasowywanie, 16, 17, 113, 118, 119, 120, 128
    warunek, 124
  oparty na obiektach, 118
  projektowy, 137
    Opcja, 137, 138
    Pusty Obiekt, 138
  projektowy
    Strategia, 130
  prosty, 115

Z

zasada DRY, 21, 35, 36
ziarno, 110
zmienna
  domknięta, 34
  globalna, 16
  instancji, 101
  leniwa, 87, 89, 90, 93
  mutowalna, 60, 87, 136
  niemutowalna, 15, 16, 59, 65, 66, 88, 125, 136
znak
  ::, 118
  "", 114
  _, 114
  =>, 124
  łańcuch, 114

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA

Helion SA

O'REILLY®

Programowanie funkcyjne
Krok po kroku

Języki funkcyjne zdobywają wśród programistów coraz większą popularność. Jak bezboleśnie zmienić sposób myślenia na funkcyjny? Ułatwi Ci to niniejsza książka, w całości poświęcona temu podejściu do programowania.

Poznaj teoretyczne podstawy programowania funkcyjnego, a następnie zacznij zgłębiać tajniki typów funkcyjnych, rekurencji oraz zmiennych niepodlegających modyfikacji. Z kolejnych rozdziałów dowiedz się, czym jest ewaluacja rygorystyczna i nierygorystyczna. Zobacz też, jak wykonać dopasowanie do wzorca. Co jeszcze znajdziesz w tej książce? Wprowadzenie do języka Scala, przedstawienie języka Groovy oraz opis technik funkcyjnego programowania obiektowego to tylko niektóre z poruszanych w niej tematów. Jeżeli chcesz zmienić sposób programowania na funkcyjny, to doskonała pozycja dla Ciebie!

Dzięki tej książce:

- poznasz teoretyczne podstawy programowania funkcyjnego
- zaznajomisz się z typami funkcyjnymi
- wykorzystasz funkcje anonimowe
- poznasz nowe wzorce projektowe
- zmienisz swoje podejście do programowania

Przekonaj się, jak podejście funkcyjne może ułatwić Ci życie!

Helion

28689 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

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

ISBN 978-83-283-0243-3

Informatyka w najlepszym wydaniu

cena 34,90 zł