Cay S. Horstmann

JAVA 8

Przewodnik doświadczonego programisty

Tytuł oryginału: Core Java® for the Impatient

Tłumaczenie: Andrzej Stefański
Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-283-1333-0

Authorized translation from the English language edition, entitled: CORE JAVA FOR THE IMPATIENT; ISBN 0321996321; by Cay S. Horstmann; published by Pearson Education, Inc, publishing as Addison Wesley.

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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/jav8pd.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/jav8pd
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

3

Interfejsy i wyrażenia lambda

W tym rozdziale

Java została zaprojektowana jako obiektowy język programowania w latach 90. ubiegłego wieku, w czasie gdy programowanie obiektowe było najważniejszym paradygmatem w tworzeniu oprogramowania. Interfejsy są kluczową funkcjonalnością w programowaniu obiektowym. Pozwalają na określenie, co ma zostać wykonane bez konieczności tworzenia implementacji.

Długo przed pojawieniem się programowania obiektowego istniały funkcjonalne języki programowania (takie jak Lisp), w których to funkcje, a nie obiekty, były najważniejszym mechanizmem tworzącym strukturę programu. Ostatnio programowanie funkcjonalne jest coraz ważniejsze, ponieważ dobrze sprawdza się w przypadku programowania równoległego i zdarzeniowego („reaktywnego"). Java wspiera wyrażenia funkcyjne, które stanowią wygodne połączenie pomiędzy programowaniem obiektowym a funkcjonalnym. W tym rozdziale opowiemy o interfejsach i wyrażeniach lambda.

Najważniejsze punkty tego rozdziału:

- Interfejs określa zestaw metod, które klasa implementująca musi dostarczyć.
- Interfejs stanowi typ nadrzędny (ang. *supertype*) dla każdej klasy, która go implementuje. Dlatego można przypisać instancje klasy do zmiennych, których typ jest określony interfejsem.
- Interfejs może zawierać metody statyczne. Wszystkie zmienne interfejsu są automatycznie uznawane za statyczne i ostateczne (ang. *final*).
- Interfejs może zawierać domyślne metody, które implementująca klasa może odziedziczyć lub przesłonić.
- Interfejsy `Comparable` i `Comparator` są używane do porównywania obiektów.
- Wyrażenie lambda opisuje blok kodu, który może być wykonany później.
- Wyrażenia lambda są konwertowane na interfejsy funkcjonalne.
- Referencje metod i konstruktorów odwołują się do metod lub konstruktorów bez ich wykonywania.
- Wyrażenia lambda i lokalne klasy wewnętrzne mogą uzyskiwać dostęp do zmiennych typu final znajdujących się w zasięgu klasy zewnętrznej.

3.1. Interfejsy

Interfejs to mechanizm pozwalający na zapisanie kontraktu pomiędzy dwoma stronami: dostawcą usług i klasami, które chcą, by ich obiekty mogły być wykorzystywane z usługą. W kolejnych podrozdziałach zobaczysz, jak definiować i wykorzystywać interfejsy w języku Java.

3.1.1. Deklarowanie interfejsu

Popatrzmy na usługę, która operuje na ciągu liczb całkowitych, dając informację o średniej z pierwszych `n` wartości:

```
public static double average(IntSequence seq, int n)
```

Takie sekwencje mogą przyjmować wiele form. Oto przykłady:

- ciąg liczb całkowitych wpisany przez użytkownika,
- ciąg wylosowanych liczb całkowitych,
- ciąg liczb pierwszych,
- ciąg elementów w tablicy zmiennych typu całkowitego,
- ciąg kodów znaków w postaci ciągu znaków (ang. *string*),
- ciąg cyfr w liczbie.

Chcemy zaimplementować *jeden mechanizm* obsługujący wszystkie powyższe rodzaje danych.

Najpierw zobaczmy, jakie wspólne cechy mają ciągi liczb całkowitych. Aby móc obsłużyć taki ciąg, potrzebne są co najmniej dwie metody:

- sprawdzająca, czy istnieje kolejny element,
- pobierająca kolejny element.

Aby zadeklarować interfejs, dostarczasz nagłówki metod w taki sposób:

```
public interface IntSequence {
    boolean hasNext();
    int next();
}
```

Nie musisz implementować tych metod, ale jeśli chcesz, możesz dopisać domyślną implementację — patrz podrozdział 3.2.2, „Metody domyślne”. Jeśli nie ma domyślnej implementacji, mówimy, że metoda jest abstrakcyjna.

Wszystkie metody interfejsu automatycznie stają się publiczne. Dzięki temu nie trzeba dopisywać przy metodach `hasNext` i `next` modyfikatora `public`. Niektórzy programiści dopisują to dla zwiększenia przejrzystości kodu.

Metody w interfejsie wystarczą do zaimplementowania metody wyliczającej średnią `average`:

```
public static double average(IntSequence seq, int n) {
    int count = 0;
    double sum = 0;
    while (seq.hasNext() && count < n) {
        count++;
        sum += seq.next();
    }
    return count == 0 ? 0 : sum / count;
}
```

3.1.2. Implementowanie interfejsu

Popatrzmy teraz na drugą stronę medalu: klasy, które mają być wykorzystywane przez metodę `average`. Muszą one *implementować* interfejs `IntSequence`. Oto przykład takiej klasy:

```
public class SquareSequence implements IntSequence {
    private int i;

    public boolean hasNext() {
        return true;
    }

    public int next() {
        i++;
        return i * i;
    }
}
```

Istnieje nieskończenie wiele liczb, które są kwadratem innej liczby całkowitej, a obiekt tej klasy może zwracać kolejne takie liczby.

Słowo kluczowe `implements` mówi o tym, że klasa `SquareSequence` będzie obsługiwała interfejs `IntSequence`.

Klasa implementująca musi deklarować metody interfejsu jako publiczne. W przeciwnym wypadku będą one miały zasięg pakietu. Ponieważ interfejs wymaga, by metoda była publiczna, kompilator zgłosi błąd.

Poniższy kod oblicza średnią ze 100 pierwszych kwadratów:

```
SquareSequence squares = new SquareSequence();
double avg = average(squares, 100);
```

Wiele klas implementuje interfejs `IntSequence`. Na przykład poniższa klasa zwraca skończony ciąg, a dokładniej: cyfry dodatniej liczby całkowitej, począwszy od najmniej znaczącej:

```
public class DigitSequence implements IntSequence {
    private int number;

    public DigitSequence(int n) {
        number = n;
    }

    public boolean hasNext() {
        return number != 0;
    }

    public int next() {
        int result = number % 10;
        number /= 10;
        return result;
    }

    public int rest() {
        return number;
    }
}
```

Obiekt `new DigitSequence(1729)` zwraca cyfry: 9, 2, 7, 1, zanim funkcja `hasNext` zwróci `false`.

Klasy `SquareSequence` i `DigitSequence` implementują wszystkie metody interfejsu `IntSequence`. Jeśli klasa implementuje tylko niektóre z metod, musi być zadeklarowana z modyfikatorem `abstract`. W rozdziale 4. znajdziesz więcej informacji na temat klas abstrakcyjnych.

3.1.3. Konwersja do typu interfejsu

Poniższy fragment kodu oblicza średnią z wartości ciągu cyfr:

```
IntSequence digits = new DigitSequence(1729);
double avg = average(digits, 100);
    // Przejrzy tylko cztery pierwsze wartości z ciągu
```

Popatrz na zmienną `digits`. Jej typ to `IntSequence`, nie `DigitSequence`. Zmienna typu `Int`
`↪Sequence` odwołuje się do obiektu dowolnej klasy implementującej interfejs `IntSequence`. Możesz zawsze przypisać do zmiennej obiekt, którego typ jest określony implementowanym interfejsem, lub przekazać go do metody oczekującej zmiennej z takim interfejsem.

A oto odrobina przydatnej terminologii. Typ `S` to typ nadrzędny typu `T` (podtypu), jeśli dowolna wartość podtypu może być przypisana do zmiennej typu nadrzędnego bez konwersji. Na przykład interfejs `IntSequence` jest typem nadrzędnym klasy `DigitSequence`.

Choć można deklarować zmienne, używając interfejsu jako ich typu, nie jest możliwe utworzenie instancji obiektu, którego typem będzie interfejs. Wszystkie obiekty muszą być instancjami klas.

3.1.4. Rzutowanie i operator instanceof

Czasem będziesz potrzebował odwrotnej konwersji — z typu nadrzędnego do podtypu. Wtedy zastosuj **rzutowanie**. Na przykład jeśli zdarzy się, że obiekt wskazywany przez zmienną typu `IntSequence` jest w rzeczywistości typu `DigitSequence`, możesz wykonać konwersję typu w taki sposób:

```
IntSequence sequence = ...;
DigitSequence digits = (DigitSequence) sequence;
System.out.println(digits.rest());
```

W tej sytuacji rzutowanie było potrzebne, ponieważ `rest` to metoda klasy `DigitSequence`, ale nie ma jej w `IntSequence`.

W ćwiczeniu 2. znajduje się lepszy przykład.

Możesz wykonać rzutowanie obiektu jedynie do typu jego rzeczywistej klasy lub jednego z jego typów nadrzędnych. Jeśli wykonasz nieprawidłowe rzutowanie, zostanie zgłoszony błąd kompilacji lub wyjątek rzutowania klasy:

```
String digitString = (String) sequence;
    // Nie może zadziałać — IntSequence nie jest typem nadrzędnym dla String
RandomSequence randoms = (RandomSequence) sequence;
    // Może zadziałać, zwróci wyjątek class cast exception, jeśli się nie powiedzie
```

Aby uniknąć zgłoszenia wyjątku, możesz przed wykonaniem rzutowania sprawdzić, czy jest to możliwe za pomocą operatora `instanceof`. Wyrażenie

```
obiekt instanceof Typ
```

zwraca `true`, jeśli obiekt jest instancją klasy, dla której *Typ* jest typem nadrzędnym. Warto to sprawdzać przed wykonaniem rzutowania.

```
if (sequence instanceof DigitSequence) {
    DigitSequence digits = (DigitSequence) sequence;
    ...
}
```

3.1.5. Rozszerzanie interfejsów

Interfejs może **rozszerzać** inny interfejs, dokładając dodatkowe metody do oryginalnych. Na przykład `Closeable` to interfejs z jedną metodą:

```
public interface Closeable {
    void close();
}
```

Jak zobaczysz w rozdziale 5., jest to ważny interfejs wykorzystywany do zwalniania zasobów w sytuacji, gdy wystąpi wyjątek.

Interfejs `Channel` rozszerza ten interfejs:

```
public interface Channel extends Closeable {
    boolean isOpen();
}
```

Klasa, która implementuje interfejs `Channel`, musi obsługiwać obie metody, a jej obiekty mogą być konwertowane do obu typów interfejsów.

3.1.6. Implementacja wielu interfejsów

Klasa może implementować dowolną liczbę interfejsów. Na przykład klasa `FileSequence`, która wczytuje liczby całkowite z pliku, może implementować interfejs `Closeable` i `IntSequence`:

```
public class FileSequence implements IntSequence, Closeable {
    ...
}
```

W takiej sytuacji klasa `FileSequence` ma dwa typy nadrzędne: `IntSequence` i `Closeable`.

3.1.7. Stałe

Każda zmienna zdefiniowana w interfejsie automatycznie otrzymuje atrybuty `public static final`.

Na przykład interfejs `SwingConstants` definiuje stałe opisujące kierunki na kompasie:

```
public interface SwingConstants {
    int NORTH = 1;
    int NORTH_EAST = 2;
    int EAST = 3;
    ...
}
```

Możesz odwoływać się do nich za pomocą pełnej nazwy `SwingConstants.NORTH`. Jeśli Twoja klasa zechce implementować interfejs `SwingConstants`, możesz opuścić przedrostek `Swing` ↪`Constants` i napisać jedynie `NORTH`. Nie jest to jednak często wykorzystywane. Dużo lepiej w przypadku zestawu stałych wykorzystać typ wyliczeniowy — patrz rozdział 4.

Nie możesz umieścić w interfejsie zmiennych instancji. Interfejs określa zachowanie, a nie stan obiektu.

3.2. Metody statyczne i domyślne

W starszych wersjach języka Java wszystkie metody interfejsu musiały być abstrakcyjne — to znaczy bez implementacji. Obecnie możesz dodać metody z implementacją na dwa sposoby: jako metody statyczne i metody domyślne. Poniższe podrozdziały opisują tego typu metody.

3.2.1. Metody statyczne

Nigdy nie było technicznych przeszkód, aby interfejs mógł posiadać metody statyczne, ale nie pasowały one do roli interfejsów jako abstrakcyjnej specyfikacji. To podejście się zmieniło. Szczególnie metody wytwórcze pasują do interfejsów. Na przykład interfejs `IntSequence` może mieć statyczną metodę `digitsOf` generującą ciąg cyfr z przekazanej liczby całkowitej:

```
IntSequence digits = IntSequence.digitsOf(1729);
```

Metoda zwraca instancję klasy implementującej interfejs `IntSequence`, ale przy wywoływaniu nie ma znaczenia, która to będzie klasa.

```
public interface IntSequence {
    ...
    public static IntSequence digitsOf(int n) {
        return new DigitSequence(n);
    }
}
```

W przeszłości często umieszczano metody statyczne w dodatkowej klasie. W bibliotece standardowej można znaleźć pary zawierające interfejs i dodatkową klasę, takie jak `Collection`/`Collections` lub `Path`/`Paths`. Taki podział nie jest już konieczny.

3.2.2. Metody domyślne

Możesz dostarczyć **domyślną** implementację dowolnej metody interfejsu. Musisz oznaczyć taką metodę modyfikatorem `default`.

```
public interface IntSequence {
    default boolean hasNext() { return true; }
```

```
        // Domyślnie sekwencje są nieskończone
    int next();
}
```

Klasa implementująca ten interfejs może przesłonić metodę `hasNext` lub odziedziczyć domyślną implementację.

Wprowadzenie możliwości definiowania metod domyślnych czyni przestarzałym klasyczny wzorzec polegający na tworzeniu interfejsu i klasy, która implementuje większość lub wszystkie jego metody zastosowany w przypadku `Collection/AbstractCol` ↪`lection` czy `WindowListener/WindowAdapter` w Java API. Obecnie należy po prostu implementować metody w interfejsie.

Ważnym zastosowaniem domyślnych metod jest umożliwienie **modyfikowania interfejsów**. Popatrzmy przykładowo na interfejs `Collection`, który jest częścią języka Java od wielu lat. Załóżmy, że jakiś czas temu utworzyłeś klasę

```
public class Bag implements Collection
```

Później, w Java 8, do interfejsu dodano metodę `stream`.

Załóżmy, że metoda `stream` nie ma domyślnej implementacji. W takiej sytuacji klasa `Bag` nie skompiluje się, ponieważ nie implementuje nowej metody. Dodanie metody bez domyślnej implementacji do interfejsu powoduje, że nie zostanie zachowana **kompatybilność źródeł** (ang. *source-compatible*).

Załóżmy jednak, że nie rekompilujesz klasy i po prostu korzystasz ze starego pliku JAR zawierającego skompilowaną klasę. Klasa nadal będzie się ładować, nawet bez brakującej metody. Programy wciążl mogą tworzyć instancje klasy `Bag` i nic nie będzie się działo. (Dodanie metody do interfejsu zachowuje kompatybilność binariów (ang. *binary-compatible*)). Jeśli jednak program wywoła metodę `stream` na instancji klasy `Bag`, wystąpi błąd `Abstract` ↪`MethodError`.

Uczynienie metody domyślną rozwiązuje oba problemy. Klasa `Bag` znowu będzie się kompilowała. A jeśli klasa będzie załadowana bez ponownej kompilacji i zostanie wywołana metoda `stream` na instancji klasy `Bag`, wykonany zostanie kod metody `Collection.stream`.

3.2.3. Rozstrzyganie konfliktów metod domyślnych

Jeśli klasa implementuje dwa interfejsy, z których jeden ma domyślną metodę, a drugi metodę (domyślną lub nie) z taką samą nazwą i typami parametrów, musisz rozstrzygnąć konflikt. Nie zdarza się to zbyt często i zazwyczaj dość łatwe jest rozwiązanie tej sytuacji.

Popatrzmy na przykład. Załóżmy, że mamy interfejs `Person` z metodą `getId`:

```
public interface Person {
    String getName();
    default int getId() { return 0; }
}
```

Załóżmy też, że mamy interfejs `Identified`, również obejmujący taką metodę:

```
public interface Identified {
    default int getId() { return Math.abs(hashCode()); }
}
```

Działanie metody `hashCode` zobaczysz w rozdziale 4. Na razie ważne jest tylko to, że zwraca ona liczby całkowite pobrane z obiektu.

Co się dzieje, gdy tworzysz klasę implementującą oba te interfejsy?

```
public class Employee implements Person, Identified {
    ...
}
```

Klasa dziedziczy dwie metody `getId` dostarczone przez interfejsy `Person` i `Identified`. Nie ma sposobu, by kompilator mógł wybrać, która z nich jest lepsza. Kompilator zgłasza błąd i pozostawia Tobie rozwiązanie tego problemu. Utwórz metodę `getId` w klasie `Employee` i zaimplementuj własny mechanizm nadawania identyfikatorów lub przekaż to do jednej z wywołujących konflikt metod w taki sposób:

```
public class Employee implements Person, Identified {
    public int getId() { return Identified.super.getId(); }
    ...
}
```

Słowo kluczowe `super` pozwala na wywołanie metody typu nadrzędnego. W takim przypadku musimy określić, który typ nadrzędny chcemy wykorzystać. Składnia może nie wyglądać najlepiej, ale jest spójna ze składnią wywoływania metod klasy nadrzędnej, którą zobaczysz w rozdziale 4.

Załóżmy, że interfejs `Identified` nie zawiera domyślnej implementacji `getId`:

```
interface Identified {
    int getId();
}
```

Czy klasa `Employee` może odziedziczyć metodę domyślną z interfejsu `Person`? Na pierwszy rzut oka może to się wydać rozsądne. Ale skąd kompilator ma wiedzieć, czy metoda `Person.getId` robi dokładnie to, czego oczekuje się od metody `Identifier.getId`? Może ona przecież zwracać na przykład informację o tym, z czym dana osoba się identyfikuje, a nie jej numer identyfikacyjny.

Projektanci języka Java postawili na bezpieczeństwo i spójność. Nie ma znaczenia, jaki konflikt występuje między dwoma interfejsami; jeśli przynajmniej jeden z interfejsów zawiera implementację, kompilator zgłasza błąd i pozostawia programiście rozwiązanie problemu.

Jeśli żaden z interfejsów nie zawiera domyślnej lub współdzielonej metody, konflikt się nie pojawi. Implementując klasę, można zaimplementować metodę lub pozostawić ją bez implementacji i zadeklarować klasę jako abstrakcyjną.

Jeśli klasa rozszerza klasę nadrzędną (patrz rozdział 4.) i implementuje interfejs, dziedzicząc taką samą metodę z obu, reguły są prostsze. W takim przypadku liczą się tylko metody klasy nadrzędnej, a metody domyślne z interfejsu są po prostu ignorowane. Jest to w rzeczywistości częstszy przypadek niż konflikty między interfejsami. Szczegóły możesz zobaczyć w rozdziale 4.

3.3. Przykłady interfejsów

Na pierwszy rzut oka interfejsy nie robią zbyt wiele. Interfejs to po prostu zestaw metod, które klasa musi zaimplementować. Dla podkreślenia znaczenia interfejsów w kolejnych podrozdziałach przedstawione zostaną cztery przykłady często wykorzystywanych interfejsów z biblioteki standardowej języka Java.

3.3.1. Interfejs Comparable

Załóżmy, że chcesz posortować tablicę obiektów. Algorytm sortujący porównuje kolejno elementy i zmienia ich kolejność, jeśli jest niewłaściwa. Oczywiście reguły porównywania są inne dla każdej klasy, a algorytm sortujący powinien po prostu wywołać metodę dostarczoną przez klasę. Dopóki wszystkie klasy będą zgodne co do tego, którą metodę należy wywołać, algorytm sortujący może działać. I tutaj właśnie wkraczają na scenę interfejsy.

Jeśli klasa chce umożliwić sortowanie swoich obiektów, powinna implementować interfejs `Comparable`. W tym interfejsie wykorzystana jest dodatkowa sztuczka techniczna. Chcemy porównywać ciągi znaków z ciągami znaków, pracowników z pracownikami itd. Dlatego interfejs `Comparable` ma parametryzowany typ.

```
public interface Comparable<T> {
    int compareTo(T other);
}
```

Na przykład klasa `String` implementuje `Comparable<String>`, więc metoda `compareTo` ma nagłówek

```
int compareTo(String other)
```

Typ danych zawierający pola o parametryzowanych typach, taki jak `Comparable` czy `ArrayList`, jest typem uogólnionym. Więcej na temat typów **uogólnionych** (ang. *generic type*) dowiesz się z rozdziału 6.

Przy wywołaniu `x.compareTo(y)` metoda `compareTo` zwraca wartość całkowitą, pozwalającą określić, czy pierwszeństwo ma `x`, czy `y`. Wartość dodatnia (nie musi być to `1`) oznacza, że element `x` powinien znaleźć się za elementem `y`. Liczba ujemna (nie musi być to `-1`) jest zwracana, gdy element `x` powinien znaleźć się przed elementem `y`. Jeśli `x` i `y` zostaną uznane za równe, zwrócona zostanie wartość `0`.

Zauważ, że wartością zwracaną może być dowolna liczba całkowita. Taka elastyczność się przydaje, ponieważ pozwala zwrócić różnicę nieujemnych liczb całkowitych.

```
public class Employee implements Comparable<Employee> {
    ...
    public int compareTo(Employee other) {
        return getId() - other.getId(); // Poprawne dla identyfikatorów większych lub
równych 0
    }
}
```

Zwracanie różnicy liczb całkowitych może spowodować wystąpienie błędu, gdy liczby mogą przyjmować wartości ujemne. W takiej sytuacji można przekroczyć zakres przy dużych operandach różnych znaków. W takim wypadku należy skorzystać z metody `Integer.compare`, która działa poprawnie dla wszystkich liczb całkowitych.

Gdy porównujesz liczby zmiennoprzecinkowe, nie możesz po prostu zwrócić różnicy. Zamiast tego należy skorzystać ze statycznej metody `Double.compare`. Działa ona poprawnie nawet dla wartości nieskończonych i `NaN`.

Poniżej znajduje się klasa `Employee` z implementacją interfejsu `Comparable` porządkująca pracowników według wynagrodzenia:

```
public class Employee implements Comparable<Employee> {
    ...
    public int compareTo(Employee other) {
        return Double.compare(salary, other.salary);
    }
}
```

Metoda `compare` ma prawo odwoływać się do zmiennej `other.salary`. W języku Java metoda może uzyskać dostęp do prywatnych właściwości dowolnego obiektu swojej klasy.

Klasa `String`, tak jak ponad 100 innych klas w bibliotece języka Java, implementuje interfejs `Comparable`. Możesz wykorzystać metodę `Arrays.sort`, by posortować tablicę obiektów typu `Comparable`:

```
String[] friends = { "Piotrek", "Paweł", "Maria" };
Arrays.sort(friends);  // Teraz zmienna friends ma wartość ["Maria", "Paweł", "Piotr"]
```

Niestety, metoda `Arrays.sort` nie sprawdza podczas kompilacji, czy przekazany do niej parametr jest tablicą obiektów typu `Comparable`. Zgłasza wyjątek, jeśli trafi na element klasy, która nie implementuje interfejsu `Comparable`.

3.3.2. Interfejs Comparator

Teraz załóżmy, że chcemy uszeregować ciągi znaków nie w kolejności alfabetycznej, a według długości (rosnąco). Nie możemy zaimplementować w klasie `String` metody `compareTo` na dwa sposoby ani, w żadnym wypadku, nie możemy jej zmodyfikować, ponieważ nie jest to nasza klasa.

Aby sobie z tym poradzić, skorzystamy z innej wersji metody `Arrays.sort`, której parametry to tablica i **komparator** — instancja klasy, która implementuje interfejs `Comparator`.

```
public interface Comparator<T> {
    int compare(T first, T second);
}
```

Aby porównać długości ciągów znaków, należy zdefiniować klasę implementującą `Comparator<String>`:

```
class LengthComparator implements Comparator<String> {
    public int compare(String first, String second) {
        return first.length() - second.length();
    }
}
```

Aby wykonać porównanie, konieczne jest utworzenie instancji:

```
Comparator<String> comp = new LengthComparator();
if (comp.compare(words[i], words[j]) > 0) ...
```

Porównaj to wywołanie z `words[i].compareTo(words[j])`. Metoda `compare` jest wywoływana na obiekcie komparatora, a nie na samym ciągu znaków.

Choć obiekt `LengthComparator` nie ma stanu, musisz utworzyć instancję, by wywołać metodę `compare` — nie jest to metoda statyczna.

Aby posortować tablicę, przekaż obiekt `LengthComparator` do metody `Arrays.sort`:

```
String[] friends = { "Piotrek", "Paweł", "Maria" };
Arrays.sort(friends, new LengthComparator());
```

Teraz tablica ma postać `["Paweł", "Maria", "Piotrek"]` lub `["Maria", "Paweł", "Piotrek"]`.

W podrozdziale 3.4.2, „Interfejsy funkcjonalne”, zobaczysz, jak można dużo prościej korzystać z klasy `Comparator` dzięki użyciu wyrażenia lambda.

3.3.3. Interfejs Runnable

W czasach, gdy prawie każdy procesor ma wiele rdzeni, warto je wszystkie wykorzystać. Możesz zechcieć uruchomić poszczególne zadania w oddzielnych wątkach lub przekazać je do wykonania w puli wątków. Aby zdefiniować zadanie, należy zaimplementować interfejs `Runnable`. Ten interfejs ma tylko jedną metodę.

```
class HelloTask implements Runnable {
    public void run() {
        for (int i = 0; i < 1000; i++) {
            System.out.println("Hello, World!");
        }
    }
}
```

Jeśli zechcesz wykonać takie zadanie w nowym wątku, utwórz wątek z obiektu typu `Runnable` i uruchom go.

```
Runnable task = new HelloTask();
Thread thread = new Thread(task);
thread.start();
```

W takiej sytuacji metoda `run` jest wykonywana w odrębnym wątku, a bieżący wątek może kontynuować wykonywanie innych zadań.

W rozdziale 10. zobaczysz inne sposoby wykonywania obiektów typu `Runnable`.

Istnieje też interfejs `Callable<T>` dla zadań, który zwraca wynik typu `T`.

3.3.4. Wywołania zwrotne interfejsu użytkownika

W graficznym interfejsie użytkownika musisz określić akcje, jakie mają być wykonane, gdy użytkownik kliknie przycisk, wybierze opcję z menu, przeciągnie suwak itp. O wywołaniach takich funkcji mówi się, że są to **wywołania zwrotne** (ang. *callback*), ponieważ fragment kodu jest wywoływany w odpowiedzi na działanie użytkownika.

W bibliotekach GUI języka Java w wywoływanych funkcjach wykorzystywane są interfejsy. Na przykład w JavaFX do zgłaszania zdarzeń używany jest poniższy interfejs:

```
public interface EventHandler<T> {
    void handle(T event);
}
```

Jest to również uogólniony interfejs, w którym `T` oznacza typ zgłaszanego zdarzenia, takiego jak `ActionEvent` przy kliknięciu przycisku.

Aby określić działanie, zaimplementuj interfejs:

```
class CancelAction implements EventHandler<ActionEvent> {
    public void handle(ActionEvent event) {
        System.out.println("Och, nie!");
    }
}
```

Następnie utwórz obiekt tej klasy i dodaj go do przycisku:

```
Button cancelButton = new Button("Anuluj");
cancelButton.setOnAction(new CancelAction());
```

Ponieważ Oracle promuje JavaFX na następcę biblioteki Swing, w swoich przykładach wykorzystuję JavaFX. (Nie martw się — nie musisz wiedzieć na temat JavaFX więcej, niż zobaczyłeś w powyższych instrukcjach). Nieważne są szczegóły; w każdej bibliotece służącej do tworzenia interfejsu, czy to Swing, JavaFX, czy Android, musisz określić dla utworzonego przycisku kod, który będzie wykonany po jego kliknięciu.

Oczywiście ten sposób definiowania akcji przycisku jest dość kłopotliwy. W innych językach po prostu określasz funkcję do wykonania po kliknięciu przycisku bez zamieszania z definiowaniem i tworzeniem klasy. W kolejnym podrozdziale zobaczysz, jak zrobić to samo w języku Java.

3.4. Wyrażenia lambda

„Wyrażenie lambda" jest blokiem kodu, który możesz przekazać do późniejszego wykonania, raz lub kilka razy. We wcześniejszych podrozdziałach widziałeś wiele sytuacji, gdzie taki blok kodu by się przydał:

- do przekazania metody porównującej do `Arrays.sort`,
- do uruchomienia zadania w oddzielnym wątku,
- do określenia akcji, jaka powinna zostać wykonana po kliknięciu przycisku.

Java jest jednak językiem obiektowym, w którym (praktycznie) wszystko jest obiektem. W Javie nie ma typów funkcyjnych. Zamiast tego funkcje mają postać obiektów, instancji klas implementujących określony interfejs. Wyrażenia lambda udostępniają wygodną składnię do tworzenia takich instancji.

3.4.1. Składnia wyrażeń lambda

Ponownie zajmijmy się przykładem sortowania z podrozdziału 3.3.2, „Interfejs Comparator". Przekazujemy kod sprawdzający, czy jeden ciąg znaków jest krótszy od innego. Obliczamy:

```
pierwszy.length() - drugi.length()
```

Czym są `pierwszy` i `drugi`? Są to ciągi znaków. Java jest językiem o silnym typowaniu i musimy określić również to:

```
(String pierwszy, String drugi) -> pierwszy.length() - drugi.length()
```

Właśnie zobaczyłeś swoje pierwsze **wyrażenie lambda**. Takie wyrażenie jest po prostu blokiem kodu z opisem zmiennych, które muszą zostać do niego przekazane.

Skąd taka nazwa? Wiele lat temu, zanim pojawiły się komputery, zajmujący się logiką Alonzo Church chciał formalnie opisać, co oznacza w przypadku funkcji matematycznej to, że jest ona obliczalna. (Co ciekawe, istnieją takie funkcje, w których przypadku nikt nie wie, jak obliczyć ich wartość). Church użył greckiej litery lambda (λ) do oznaczenia parametrów w taki sposób:

```
λpierwszy. λdrugi[ASt6]. pierwszy.length() - drugi.length()
```

Dlaczego λ? Czy skończyły mu się litery alfabetu? W rzeczywistości w czcigodnej księdze *Principia mathematica* (*http://pl.wikipedia.org/wiki/Principia_mathematica*) do oznaczenia parametrów funkcji wykorzystano znak ^, co było inspiracją do tego, by użyć wielkiej litery lambda (Λ). W końcu jednak wykorzystał on małą literę. Od tego czasu wyrażenia z parametryzowanymi zmiennymi są nazywane *wyrażeniami lambda*.

Jeśli obliczeń wyrażenia lambda nie da się zapisać w jednym wyrażeniu, należy zapisać je w taki sam sposób jak przy tworzeniu metody — wewnątrz nawiasów {} z jawnie zapisaną instrukcją `return`. Na przykład:

```
(String pierwszy, String drugi) -> {
    int różnica = pierwszy.length() < drugi.length();
    if (różnica < 0) return -1;
    else if (różnica > 0) return 1;
    else return 0;
}
```

Jeśli wyrażenie lambda nie ma parametrów, należy umieścić puste nawiasy, jak w przypadku metody bez parametrów:

```
Runnable task = () -> { for (int i = 0; i < 1000; i++) doWork(); }
```

Jeśli typy parametrów wyrażenia lambda mogą być jednoznacznie ustalone, można je pominąć. Na przykład:

```
Comparator<String> comp
    = (pierwszy, drugi) -> pierwszy.length() - drugi.length();
   // Zadziała jak (String pierwszy, String drugi)
```

W tym przypadku kompilator może określić, że zmienne `pierwszy` i `drugi` muszą być typu `String`, ponieważ wyrażenie lambda jest przypisane do komparatora dla tego typu. (Przyjrzymy się temu dokładniej w kolejnym podrozdziale).

Jeśli metoda ma jeden parametr domyślnego typu, możesz nawet pominąć nawiasy:

```
EventHandler<ActionEvent> listener = event ->
    System.out.println("Oh, nie!");
        // Zamiast (event) -> lub (ActionEvent event) ->
```

Nigdy nie określa się typu wartości zwracanej przez wyrażenie lambda. Kompilator jednak ustala go na podstawie treści kodu i sprawdza, czy zgadza się on z oczekiwanym typem. Na przykład wyrażenie

```
(String pierwszy, String drugi) -> pierwszy.length() - drugi.length()
```

może być użyte w kontekście, w którym oczekiwany jest wynik typu `int` (lub typu kompatybilnego, jak `Integer`, `long` czy `double`).

3.4.2. Interfejsy funkcyjne

Jak już widziałeś, w języku Java istnieje wiele interfejsów określających działania, takich jak `Runnable` czy `Comparator`. Wyrażenia lambda są kompatybilne z tymi interfejsami.

Możesz umieścić wyrażenie lambda wszędzie tam, gdzie oczekiwany jest obiekt implementujący *jedną metodę abstrakcyjną*. Taki interfejs nazywany jest **interfejsem funkcjonalnym**.

Aby zademonstrować konwersję do interfejsu funkcjonalnego, przyjrzyjmy się metodzie `Arrays.sort`. Jej drugi parametr wymaga instancji interfejsu `Comparator` zawierającego jedną metodę. Wstawmy tam po prostu wyrażenie lambda:

```
Arrays.sort(słowa,
    (pierwszy, drugi) -> pierwszy.length() - drugi.length());
```

W tle drugi parametr metody `Arrays.sort` zamieniany jest na obiekt pewnej klasy implementującej `Comparator<String>`. Wywołanie metody `Compare` na tym obiekcie powoduje wykonanie treści wyrażenia lambda. Zarządzanie takimi obiektami i klasami jest w pełni zależne od implementacji i dobrze zoptymalizowane.

W większości języków programowania obsługujących literały funkcyjne możesz deklarować typy funkcyjne takie jak `(String, String) -> int`, co powoduje umieszczenie w tak zadeklarowanej zmiennej funkcji i jej wykonanie. W języku Java wyrażenie lambda można wykorzystać *tylko w jeden sposób*: umieścić je w zmiennej, której typem jest interfejs funkcjonalny, tak by została zamieniona w instancję tego interfejsu.

Nie możesz przypisać wyrażenia lambda do zmiennej typu `Object`, wspólnego typu nadrzędnego dla wszystkich klas języka Java (patrz rozdział 4.). `Object` to klasa, a nie interfejs funkcjonalny.

Biblioteka standardowa udostępnia dużą liczbę interfejsów funkcjonalnych (patrz podrozdział 3.6.2, „Wybieranie interfejsu funkcjonalnego”). Jednym z nich jest

```
public interface Predicate<T> {
    boolean test(T t);
    // Dodatkowe domyślne i statyczne metody
}
```

Klasa `ArrayList` ma metodę `removeIf`, której parametrem jest `Predicate`. Jest to zaprojektowane z myślą o zastosowaniu wyrażenia lambda. Na przykład poniższe wyrażenie usuwa wszystkie wartości `null` z tablicy `ArrayList`.

```
list.removeIf(e -> e == null);
```

3.5. Referencje do metod i konstruktora

Zdarza się, że jest już metoda wykonująca działanie, które chciałbyś wykorzystać w innym miejscu kodu. Istnieje specjalna składnia dla *referencji do metod*, która jest nawet krótsza niż wyrażenie lambda wywołujące metodę. Podobny skrót stosuje się w przypadku konstruktorów. Oba rozwiązania zobaczysz w kolejnych podrozdziałach.

3.5.1. Referencje do metod

Załóżmy, że chcesz sortować ciągi znaków bez zwracania uwagi na wielkość liter. Powinieneś wywołać

```
Arrays.sort(strings, (x, y) -> x.compareToIgnoreCase(y));
```

Zamiast tego możesz też przekazać takie wyrażenie:

```
Arrays.sort(strings, String::compareToIgnoreCase);
```

Wyrażenie `String::compareToIgnoreCase` jest **referencją metody**, która stanowi odpowiednik wyrażenia lambda `(x, y) -> x.compareToIgnoreCase(y)`.

Oto inny przykład. Klasa `Objects` definiuje metodę `isNull`. Wywołanie `Objects.isNull(x)` po prostu zwraca wartość `x == null`. Nie widać, aby warto było w tym przypadku tworzyć metodę, ale zostało to zaprojektowane w taki sposób, by przekazywać tu metodę. Wywołanie

```
list.removeIf(Objects::isNull);
```

usuwa wszystkie wartości `null` z listy.

Jako inny przykład załóżmy, że chcesz wyświetlić wszystkie elementy listy. Klasa `ArrayList` ma metodę `forEach` wykonującą funkcję na każdym jej elemencie. Mógłbyś wywołać

```
list.forEach(x -> System.out.println(x));
```

Przyjemniej jednak byłoby, gdybyś mógł przekazać po prostu metodę `println` do metody `forEach`. Możesz to zrobić tak:

```
list.forEach(System.out::println);
```

Jak możesz zobaczyć w tych przykładach, operator `::` oddziela nazwę metody od nazwy klasy czy obiektu. Istnieją trzy rodzaje:

1. *Klasa*`::`*metodaInstancji*
2. *Klasa*`::`*metodaStatyczna*
3. *obiekt*`::`*metodaInstancji*

W pierwszym przypadku pierwszy parametr staje się odbiorcą metody i wszystkie inne parametry są przekazywane do metody. Na przykład `String::compareToIgnoreCase` oznacza to samo co `(x, y) -> x.compareToIgnoreCase(y)`.

W drugim przypadku wszystkie parametry są przekazywane do metody statycznej. Wyrażenie metody `Objects::isNull` jest równoważne `x -> Objects.isNull(x)`.

W trzecim przypadku metoda jest wywoływana na danym obiekcie i parametry są przekazywane do metody instancji. W tej sytuacji `System.out::println` jest równoważne z `x -> System.out.println(x)`.

Gdy istnieje wiele przeładowanych metod z tą samą nazwą, kompilator na podstawie kontekstu będzie próbował ustalić, którą z nich chcesz wywołać. Na przykład istnieje wiele wersji metody `println`. Po przekazaniu do metody `forEach` zmiennej typu `ArrayList<String>` zostanie wybrana metoda `println(String)`.

W referencji do metody możesz wykorzystać parametr `this`. Na przykład `this::equals` jest równoważne z `x -> this.equals(x)`.

W klasie wewnętrznej możesz wykorzystać referencję `this` do klasy zewnętrznej poprzez *KlasaZewnętrzna*`.this::`*metoda*. Możesz też wykorzystać `super` — patrz rozdział 4.

3.5.2. Referencje konstruktora

Referencje konstruktora są odpowiednikami referencji metod, tyle że w ich przypadku nazwą metody jest `new`. Na przykład `Employee::new` jest referencją do konstruktora klasy `Employee`. Jeśli klasa ma więcej niż jeden konstruktor, od kontekstu zależy, który z nich zostanie wywołany.

Poniżej znajduje się przykład wykorzystania takiej referencji konstruktora. Załóżmy, że masz listę ciągów znaków:

```
List<String> names = ...;
```

Potrzebujesz listy pracowników, jednej dla każdej nazwy. Jak zobaczysz w rozdziale 8., możesz wykorzystać strumienie, by wykonać to samo bez pętli: zamienić listę na strumień i wywołać metodę `map`. Powoduje to wykonanie funkcji i zbiera wszystkie rezultaty.

```
Stream<Employee> stream = names.stream().map(Employee::new);
```

Ponieważ `names.stream()` zawiera obiekty `String`, kompilator wie, że `Employee::new` odwołuje się do konstruktora `Employee(String)`.

Możesz tworzyć referencje do konstruktora z typami tablicowymi. Na przykład `int[]::new` jest referencją do konstruktora z jednym parametrem: długością tablicy. Jest to odpowiednik wyrażenia lambda `n -> new int[n]`.

Referencje konstruktora z typami tablicowymi pozwalają na ominięcie ograniczenia języka Java polegającego na tym, że nie jest możliwe skonstruowanie tablicy zmiennych typu prostego. (Szczegóły znajdziesz w rozdziale 6.). Z tego powodu metody takie jak `Stream.toArray` zwracają tablicę zmiennych typu `Object`, a nie tablicę zmiennych takiego samego typu jak zmienne znajdujące się w strumieniu:

```
Object[] employees = stream.toArray();
```

Nie jest to jednak satysfakcjonujące. Użytkownik potrzebuje tablicy pracowników, nie obiektów. Aby rozwiązać ten problem, inna wersja `toArray` akceptuje referencje do konstruktora:

```
Employee[] buttons = stream.toArray(Employee[]::new);
```

Metoda `toArray` wywołuje ten konstruktor, by uzyskać tablicę odpowiedniego typu. Następnie wypełnia ją i zwraca dane w tablicy.

3.6. Przetwarzanie wyrażeń lambda

Wiesz już, jak tworzyć wyrażenia lambda i przekazać je do metody, która oczekuje interfejsu funkcjonalnego. W kolejnych podrozdziałach zobaczysz, jak napisać metody, które mogą korzystać z wyrażeń lambda.

3.6.1. Implementacja odroczonego wykonania

Celem korzystania z wyrażeń lambda jest **odroczone wykonanie**. W końcu jeśli chciałbyś wykonać jakieś polecenia w danym miejscu kodu bezzwłocznie, zrobiłbyś to bez opakowywania go w wyrażenie lambda. Istnieje wiele powodów opóźnienia wykonania kodu — są to:

- wykonanie kodu w oddzielnym wątku,
- wielokrotne wykonanie kodu,
- wykonanie kodu we właściwym miejscu algorytmu (na przykład operacja porównania przy sortowaniu),
- wykonanie kodu w reakcji na zdarzenie (kliknięcie przycisku, odebranie danych itd.),
- wykonanie kodu tylko w razie potrzeby.

Popatrzmy na prosty przykład. Załóżmy, że chcesz powtórzyć działanie *n* razy. Działanie i licznik są przekazywane do metody `repeat`:

```
repeat(10, () -> System.out.println("Witaj, świecie!"));
```

Aby wykorzystać wyrażenie lambda, musimy wybrać (lub w rzadkich przypadkach utworzyć) interfejs funkcjonalny. W takim przypadku możemy użyć na przykład `Runnable`:

```
public static void repeat(int n, Runnable action) {
    for (int i = 0; i < n; i++) action.run();
}
```

Zauważ, że kod z wyrażenia lambda wykonywany jest po wywołaniu `action.run()`.

Skomplikujmy teraz ten przykład. Chcemy do działania przekazać informację o numerze iteracji, w której jest wywoływane. W takim przypadku musimy wybrać interfejs funkcjonalny, który ma metodę z parametrem `int` i zwraca `void`. Zamiast tworzenia własnego mocno polecam wykorzystanie jednego ze standardowych interfejsów opisanych w kolejnym podrozdziale. Standardowym interfejsem do przetwarzania wartości typu `int` jest

```
public interface IntConsumer {
    void accept(int value);
}
```

Oto ulepszona wersja metody `repeat`:

```
public static void repeat(int n, IntConsumer action) {
    for (int i = 0; i < n; i++) action.accept(i);
}
```

A wywołuje się ją w taki sposób:

```
repeat(10, i -> System.out.println("Odliczanie: " + (9 - i)));
```

3.6.2. Wybór interfejsu funkcjonalnego

W większości funkcyjnych języków programowania typy funkcyjne są **strukturalne**. Aby określić funkcję mapującą dwa ciągi znaków na liczby całkowite, korzystasz z typu wyglądającego tak: `Funkcja2<String, String, Integer>` lub tak: `(String, String) -> int`. W języku Java zamiast tego deklarujesz intencję funkcji za pomocą interfejsu funkcjonalnego takiego jak `Comparator<String>`. W teorii języków programowania nazywane jest to **typowaniem nominalnym** (ang. *nominal typing*).

Oczywiście będzie wiele sytuacji, w których zechcesz przyjąć „dowolną funkcję" bez specjalnej semantyki. Istnieje szereg prostych typów funkcyjnych, które można do tego wykorzystać (patrz tabela 3.1), i bardzo dobrym pomysłem jest korzystanie z nich, gdy tylko jest to możliwe.

Tabela 3.1. *Popularne interfejsy funkcjonalne*

Interfejs funkcjonalny	Typy parametrów	Typ zwracany	Nazwa metody abstrakcyjnej	Opis	Inne metody
`Runnable`	brak	`void`	`Run`	Uruchamia działanie bez parametrów i wartości zwracanej	
`Supplier<T>`	brak	`T`	`Get`	Dostarcza wartość typu `T`	
`Consumer<T>`	`T`	`void`	`Accept`	Pobiera wartość typu `T`	`andThen`
`BiConsume<T, U>`	`T, U`	`void`	`Accept`	Pobiera wartości typu `T` i `U`	`andThen`
`Function<T, R>`	`T`	`R`	`Apply`	Funkcja z parametrem typu `T`	`compose`, `andThen`, `identity`
`BiFunction<T, U, R>`	`T, U`	`R`	`Apply`	Funkcja z parametrami typu `T` i `U`	`andThen`
`UnaryOperator<T>`	`T`	`T`	`Apply`	Operator jednoargumentowy dla typu `T`	`compose`, `andThen`, `identity`
`BinaryOperator<T>`	`T, T`	`T`	`Apply`	Operator dwuargumentowy dla typu `T`	`andThen`, `maxBy`, `minBy`
`Predicate<t>`	`T`	`boolean`	`Test`	Funkcja zwracająca wartość logiczną	`and`, `or`, `negate`, `isEqual`
`BiPredicate<T, U>`	`T, U`	`boolean`	`Test`	Dwuargumentowa funkcja zwracająca wartość logiczną	`and`, `or`, `negate`

Dla przykładu załóżmy, że piszesz metodę przetwarzającą pliki spełniające zadane kryteria. Czy powinieneś użyć klasy `java.ioFileFilter` czy `Predicate<File>`? Bardzo polecam korzystanie ze standardowego `Predicate<File>`. Jedynym powodem, by tego nie robić, może być sytuacja, gdy masz już wiele użytecznych metod tworzących instancje `FileFilter`.

Większość standardowych interfejsów funkcjonalnych ma metody nieabstrakcyjne do tworzenia lub łączenia funkcji. Na przykład `Predicate.isEqual(a)` jest odpowiednikiem `a::equals`, ale działa również w sytuacji, gdy `a` ma wartość `null`. Istnieją domyślne metody: `and`, `or`, `negate` do łączenia predykatów. Na przykład `Predicate.isEqual(a).or` ↪`(Predicate.isEqual(b))` jest równoważne z `x -> a.equals(x) || b.equals(x)`.

Tabela 3.2 pokazuje 34 dostępne specjalizacje dla typów prostych: `int`, `long` i `double`. Warto korzystać z tych specjalizacji, by unikać automatycznych przekształceń. Dlatego właśnie wykorzystałem `IntConsumer` zamiast `Consumer<Integer>` w poprzednim podrozdziale.

Tabela 3.2. *Interfejsy funkcjonalne dla typów prostych*
`p, q` *to* *`int, long, double;`* `P, Q` *to* *`Int, Long, Double`*

Interfejs funkcjonalny	Typy parametrów	Typ zwracany	Nazwa metody abstrakcyjnej
`BooleanSupplier`	brak	`Boolean`	`getAsBoolean`
`P``Supplier`	brak	*`P`*	`getAs`*`P`*
`P``Consumer`	*`p`*	`Void`	`accept`
`Obj`*`P`*`Consumer<T>`	`T`, *`p`*	`Void`	`accept`
`P``Function<t>`	*`p`*	`T`	`apply`
`P``To`*`Q`*`Function`	*`p`*	*`Q`*	`applyAs`*`Q`*
`To`*`P`*`Function<T>`	`T`	*`P`*	`applyAs`*`P`*
`To`*`P`*`BiFunction<T, U>`	`T, U`	*`P`*	`applyAs`*`P`*
`P``UnaryOperator`	*`p`*	*`P`*	`applyAs`*`P`*
`P``BinaryOperator`	*`p, p`*	*`P`*	`applyAs`*`P`*
`P``Predicate`	*`p`*	`Boolean`	`test`

3.6.3. Implementowanie własnych interfejsów funkcjonalnych

Niezbyt często znajdziesz się w sytuacji, gdy żaden ze standardowych interfejsów funkcjonalnych nie będzie odpowiedni. Wtedy będzie trzeba stworzyć własny.

Załóżmy, że chcesz wypełnić obraz kolorowymi wzorami, dla których użytkownik określa funkcję zwracającą kolor każdego piksela. Nie ma standardowego typu mapującego `(int, int) -> Color`. Mógłbyś użyć `BiFunction<Integer, Integer, Color>`, ale to grozi automatycznym opakowywaniem (ang. *autoboxing*).

W takim przypadku warto zdefiniować nowy interfejs

```
@FunctionalInterface
public interface PixelFunction {
    Color apply(int x, int y);
}
```

Powinieneś oznaczać interfejsy funkcjonalne adnotacją `@FunctionalInterface`. Ma to dwie zalety. Po pierwsze, kompilator sprawdza, czy oznaczony w ten sposób kod jest interfejsem z jedną metodą abstrakcyjną. Po drugie, dokumentacja wygenerowana przez `javadoc` zawiera informację o tym, że jest to interfejs funkcjonalny.

Możesz już teraz zaimplementować metodę:

```
BufferedImage utwórzObraz (int width, int height, PixelFunction f) {
    BufferedImage image = new BufferedImage(width, height,
        BufferedImage.TYPE_INT_RGB);

    for (int x = 0; x < width; x++)
        for (int y = 0; y < height; y++) {
            Color color = f.apply(x, y);
            image.setRGB(x, y, color.getRGB());
        }
    return image;
}
```

Aby ją wywołać, utwórz wyrażenie lambda zwracające wartość koloru dla dwóch liczb całkowitych:

```
BufferedImage francuskaFlaga = createImage(150, 100,
    (x, y) -> x < 50 ? Color.BLUE : x < 100 ? Color.WHITE : Color.RED);
```

3.7. Wyrażenia lambda i zasięg zmiennych

Z kolejnych podrozdziałów dowiesz się, w jaki sposób zmienne zachowują się wewnątrz wyrażeń lambda. Są to szczegóły techniczne, ale istotne przy pracy z wyrażeniami lambda.

3.7.1. Zasięg zmiennej lambda

Treść wyrażenia lambda ma *taki sam zasięg jak zagnieżdżony blok kodu*. Takie same reguły stosuje się przy konflikcie nazw i przesłanianiu. Nie można deklarować parametru lub zmiennej lokalnej w wyrażeniu lambda o takiej samej nazwie jak zmienna lokalna.

```
int first = 0;
Comparator<String> comp = (first, second) -> first.length() - second.length();
    // Błąd: jest już zmienna o takiej nazwie
```

Wewnątrz metody nie możesz mieć dwóch zmiennych lokalnych o tej samej nazwie, dlatego nie możesz też wprowadzać takich zmiennych w wyrażeniu lambda.

Inną konsekwencją zasady „tego samego zasięgu" jest to, że słowo `this` w wyrażeniu lambda oznacza parametr `this` metody, która tworzy wyrażenie lambda. Dla przykładu rozważmy

```
public class Application() {
    public void doWork() {
        Runnable runner = () -> { ...; System.out.println(this.toString()); ... };
        ...
    }
}
```

Wyrażenie `this.toString()` wywołuje metodę `toString` obiektu `Application`, a nie instancji `Runnable`. Nie ma nic nietypowego w działaniu `this` w wyrażeniu lambda. Zasięg wyrażenia lambda jest zagnieżdżony wewnątrz metody `doWork`, a `this` ma takie samo znaczenie w każdym miejscu tej metody.

3.7.2. Dostęp do zmiennych zewnętrznych

Często w wyrażeniu lambda poztrabny jest dostęp do zmiennych z metody lub klasy wywołującej. Rozważmy taki przykład:

```
public static void powtórzKomunikat(String tekst, int liczba) {
    Runnable r = () -> {
        for (int i = 0; i < liczba; i++) {
            System.out.println(tekst);
        }
    };
    new Thread(r).start();
}
```

Zauważ, że wyrażenie lambda uzyskuje dostęp do zmiennych przekazanych jako parametr w szerszym kontekście, a nie w samym wyrażeniu lambda.

Rozważ wywołanie:

```
powtórzKomunikat("Witaj", 1000); // Wyświetla Witaj 100 razy w oddzielnym wątku
```

Popatrzmy teraz na zmienne `liczba` i `tekst` w wyrażeniu lambda. Widać, że tutaj dzieje się coś nieoczywistego. Kod tego wyrażenia lambda może być wykonywany jeszcze długo po tym, jak wywołanie metody `powtórzKomunikat` się zakończy i zmienne parametrów przestaną być dostępne. W jaki sposób zmienne `tekst` i `liczba` pozostają dostępne przy wykonywaniu wyrażenia lambda?

Aby zrozumieć, co się tutaj dzieje, musimy zmienić swoje podejście do wyrażeń lambda. Wyrażenie lambda ma trzy składniki:

1. blok kodu,
2. parametry,
3. wartości **wolnych** zmiennych — czyli takich zmiennych, które nie są parametrami i nie są zdefiniowane w kodzie.

W naszym przykładzie wyrażenie lambda ma dwie wolne zmienne, `tekst` i `liczba`. Struktura danych reprezentująca wyrażenie lambda musi przechowywać wartości tych zmiennych — w naszym przypadku `"Witaj"` i `1000`. Mówimy, że te wartości zostały przejęte przez wyrażenie lambda. (Sposób, w jaki zostanie to wykonane, zależy od implementacji. Można na przykład zamienić wyrażenie lambda na obiekt z jedną metodą i wartości wolnych zmiennych skopiować do zmiennych instancji tego obiektu).

Techniczny termin określający blok kodu z wartościami wolnych zmiennych to **domknięcie** (ang. *closure*). W języku Java wyrażenia lambda są domknięciami.

Jak widziałeś, wyrażenie lambda ma dostęp do zmiennych zdefiniowanych w wywołującym je kodzie. Dla upewnienia się, że taka wartość jest poprawnie zdefiniowana, istnieje ważne ograniczenie. W wyrażeniu lambda masz dostęp tylko do zmiennych zewnętrznych, których wartość się nie zmienia. Czasem tłumaczy się to, mówiąc, że wyrażenie lambda przechwytuje wartości, a nie zmienne. Na przykład poniższy kod wygeneruje błąd przy kompilacji:

```
for (int i = 0; i < n; i++) {
    new Thread(() -> System.out.println(i)).start();
        // Błąd — nie można pobrać i
}
```

Wyrażenie lambda próbuje wykorzystać zmienną `i`, ale nie jest to możliwe, ponieważ ona się zmienia. Nie ma określonej wartości do pobrania. Zasadą jest, że wyrażenie lambda może uzyskać dostęp jedynie do zmiennych lokalnych z otaczającego kodu, które **efektywnie są stałymi** (ang. *effective final*). Taka zmienna nie jest modyfikowana — jest lub mogłaby być zdefiniowana z modyfikatorem `final`.

To samo dotyczy zmiennych używanych w lokalnej klasie wewnętrznej (patrz podrozdział 3.9, „Lokalne klasy wewnętrzne"). Dawniej ograniczenie było silniejsze — pobrane zmienne musiały być zadeklarowane z modyfikatorem `final`. Zostało to zmienione.

Zmienna rozszerzonej pętli `for` jest faktycznie stałą, ponieważ ma zasięg pojedynczej iteracji. Poniższy kod jest w pełni poprawny:

```
for (String arg : args) {
    new Thread(() -> System.out.println(arg)).start();
        // Można pobrać arg
}
```

Nowa zmienna `arg` jest tworzona przy każdej iteracji i ma przypisywaną kolejną wartość z tablicy `args`. W odróżnieniu od tego zasięgiem zmiennej `i` w poprzednim przykładzie była cała pętla.

Konsekwencją reguły nakazującej stosowanie „efektywnie stałych" zmiennych jest to, że wyrażenie lambda nie może zmodyfikować żadnej z wykorzystywanych wartości. Na przykład:

```
public static void powtórzKomunikat(String tekst, int liczba, int wątki) {
    Runnable r = () -> {
        while (liczba > 0) {
            liczba--; // Błąd: nie można modyfikować przechwyconej wartości
            System.out.println(tekst);
        }
```

```
        };
        for (int i = 0; i < wątki; i++) new Thread(r).start();
    }
```

Jest to w rzeczywistości pozytywna cecha. Jak zobaczysz w rozdziale 10., w sytuacji gdy dwa wątki aktualizują jednocześnie wartość zmiennej `liczba`, jej wartość pozostaje nieokreślona.

Nie licz, że kompilator wychwyci wszystkie błędy dostępu związane z równoczesnym dostępem. Zakaz modyfikowania dotyczy tylko zmiennych lokalnych. Jeśli zmienna `liczba` będzie zmienną instancji lub zmienną statyczną zewnętrznej klasy, błąd nie zostanie zgłoszony, nawet jeśli wynik działania będzie nieokreślony.

Można obejść ograniczenie uniemożliwiające wykonywanie operacji modyfikujących wartość, korzystając z tablicy o długości `1`:

```
int[] licznik = new int[1];
button.setOnAction(event -> licznik[0]++);
```

Dzięki takiej konstrukcji zmienna `licznik` ma w tym kodzie stałą wartość — nigdy się nie zmienia, ponieważ cały czas wskazuje na tę samą tablicę i możesz uzyskać do niej dostęp w wyrażeniu lambda.

Oczywiście taki kod nie będzie bezpieczny przy operacjach wielowątkowych. Poza ewentualnym zastosowaniem w jednowątkowym interfejsie użytkownika jest to bardzo złe rozwiązanie. Implementację bezpiecznego licznika w zastosowaniach wielowątkowych pokażemy w rozdziale 10.

3.8. Funkcje wyższych rzędów

W funkcyjnych językach programowania funkcje są bardzo ważne. Tak samo jak przekazujesz liczby do metod i tworzysz metody generujące liczby, możesz mieć też parametry i zwracane wartości, które są funkcjami. Funkcje, które przetwarzają lub zwracają funkcje, są nazywane **funkcjami wyższych rzędów**. Brzmi to abstrakcyjnie, ale jest bardzo użyteczne w praktyce. Java nie jest w pełni językiem funkcyjnym, ponieważ wykorzystuje interfejsy funkcjonalne, ale zasada jest taka sama. W kolejnych podrozdziałach popatrzymy na przykłady i przeanalizujemy funkcje wyższego rzędu w interfejsie `Comparator`.

3.8.1. Metody zwracające funkcje

Załóżmy, że czasem chcemy posortować tablicę ciągów znaków rosnąco, a czasem malejąco. Możemy stworzyć metodę, która utworzy właściwy komparator:

```
public static Comparator<String> compareInDirecton(int direction) {
    return (x, y) -> direction * x.compareTo(y);
}
```

Wywołanie `compareInDirection(1)` zwraca komparator do sortowania rosnąco, a wywołanie `compareInDirection(-1)` zwraca komparator do sortowania malejąco.

Wynik może być przekazany do innej metody (takiej jak `Arrays.sort`), która pobiera taki interfejs:

```
Arrays.sort(friends, compareInDirection(-1));
```

Generalnie można śmiało pisać metody tworzące funkcje (lub, dokładniej, instancje klas implementujących interfejs funkcjonalny). Przydaje się to do generowania funkcji, które są przekazywane do metod z interfejsami funkcjonalnymi.

3.8.2. Metody modyfikujące funkcje

W poprzednim podrozdziale widziałeś metody, które zwracają komparator zmiennych typu `String` do sortowania rosnąco lub malejąco. Możemy to uogólnić, odwracając każdy komparator:

```
public static Comparator<String> reverse(Comparator<String> comp) {
    return (x, y) -> comp.compare(x, y);
}
```

Ta metoda działa na funkcjach. Pobiera funkcję i zwraca ją zmodyfikowaną. Aby uzyskać niezależny od wielkości znaków porządek malejący, użyj:

```
reverse(String::compareToIgnoreCase)
```

Interfejs `Comparator` ma domyślną metodę `reversed`, która właśnie w ten sposób tworzy odwrotny komparator.

3.8.3. Metody interfejsu Comparator

Interfejs `Comparator` ma wiele użytecznych metod statycznych, które są funkcjami wyższego rzędu generującymi komparatory.

Metoda `comparing` pobiera funkcję tworzącą klucz (ang. *key extractor*), mapującą zmienną typu `T` na zmienną typu, który da się porównać (takiego jak `String`). Funkcja jest wywoływana dla obiektów, które mają być porównane, i porównywane są zwrócone klucze. Na przykład załóżmy, że masz tablicę obiektów klasy `Person`. Możesz je posortować według nazwy w taki sposób:

```
Arrays.sort(people, Comparator.comparing(Person::getName));
```

Komparatory można łączyć w łańcuch za pomocą metody `thenComparing`, która jest wywoływana, gdy pierwsze porównanie nie pozwala określić kolejności. Na przykład:

```
Arrays.sort(people, Comparator
    .comparing(Person::getLastName)
    .thenComparing(Person::getFirstName));
```

Jeśli dwóch ludzi ma takie samo nazwisko, wykorzystywany jest drugi komparator.

Istnieje kilka odmian tych metod. Możesz określić komparator, który ma być wykorzystywany dla kluczy tworzonych przez metody `comparing` i `thenComparing`. Na przykład możemy posortować ludzi według długości ich nazwisk:

```
Arrays.sort(people, Comparator.comparing(Person::getName,
    (s, t) -> s.length() - t.length()));
```

Co więcej, zarówno metoda `comparing`, jak i `thenComparing` istnieją w wersji umożliwiającej uniknięcie opakowywania wartości: `int`, `long` i `double`. Prostszym sposobem sortowania po długości nazwy będzie użycie

```
Arrays.sort(people, Comparator.comparingInt(p -> p.getName().length()));
```

Jeśli utworzona przez Ciebie funkcja klucza może zwrócić `null`, polubisz adaptery `nullsFirst` i `nullsLast`. Te statyczne metody biorą istniejący komparator i modyfikują go w taki sposób, że nie wywołuje on wyjątku, gdy pojawi się wartość `null`, ale uznaje ją za mniejszą lub większą niż normalne wartości. Na przykład załóżmy, że funkcja `getMiddleName` zwraca `null`, jeśli osoba nie ma drugiego imienia. W takiej sytuacji możesz wykorzystać `Comparator.comparing(Person::getMiddleName(), Comparator.nullsFirst(...))`.

Metoda `nullsFirst` potrzebuje komparatora — w tym przypadku takiego, który porównuje dwa ciągi znaków. Metoda `naturalOrder` tworzy komparator dla dowolnej klasy implementującej interfejs `Comparable`. Poniżej znajduje się pełne polecenie sortujące z wykorzystaniem drugiego imienia, które potencjalnie może mieć wartość `null`. Dla zachowania czytelności korzystam ze statycznie importowanego `java.util.Comparator.*`. Zauważ, że typ zmiennej `naturalOrder` jest ustalany z kontekstu.

```
Arrays.sort(people, comparing(Person::getMiddleName,
    nullsFirst(naturalOrder())));
```

Statyczna metoda `reverseOrder` odwraca kolejność.

3.9. Lokalne klasy wewnętrzne

Długo przed pojawieniem się wyrażeń lambda język Java miał mechanizmy do łatwego definiowania klas implementujących interfejs (lub interfejs funkcjonalny). W przypadku interfejsów funkcjonalnych powinieneś koniecznie używać wyrażeń lambda, ale czasem możesz potrzebować zgrabnej składni dla interfejsu, który nie jest interfejsem funkcjonalnym. Klasyczne rozwiązania możesz też napotkać, przeglądając istniejący kod.

3.9.1. Klasy lokalne

Możesz definiować klasę wewnątrz metody. Taka klasa jest nazywana **klasą lokalną**. Powinieneś korzystać z tego tylko w przypadku klas, które są jedynie konstrukcją. Jest tak często, gdy klasa implementuje interfejs, a przy wywołaniu metody ważny jest tylko implementowany interfejs, nie zaś sama klasa.

Na przykład rozważmy metodę

```
public static IntSequence randomInts(int low, int high)
```

która generuje nieskończony ciąg losowych liczb całkowitych w zadanym zakresie.

Ponieważ `IntSequence` jest interfejsem, metoda musi zwrócić obiekt pewnej klasy implementującej ten interfejs. Kod wywołujący metodę nie zwraca uwagi na samą klasę, więc może ona być zadeklarowana wewnątrz samej metody:

```
private static Random generator = new Random();

public static IntSequence randomInts(int low, int high) {
    class RandomSequence implements IntSequence {
        public int next() { return low + generator.nextInt(high - low + 1); }
        public boolean hasNext() { return true; }
    }

    return new RandomSequence();
}
```

Klasa lokalna nie jest deklarowana jako publiczna lub prywatna, ponieważ nie jest dostępna spoza metody.

Tworzenie klasy lokalnej ma dwie zalety. Po pierwsze, jej nazwa jest ukryta wewnątrz metody. Po drugie, metody klasy mogą uzyskać dostęp do zmiennych zewnętrznych, tak jak w przypadku wyrażeń lambda.

W naszym przykładzie metoda `next` korzysta z trzech zmiennych: `low`, `high` i `generator`. Jeśli zmieniasz `RandomInt` w klasę zagnieżdżoną, będziesz musiał utworzyć jawny konstruktor pobierający te wartości i zapisujący je w zmiennych instancji (patrz ćwiczenie 15.).

3.9.2. Klasy anonimowe

W przykładzie z poprzedniego podrozdziału nazwa `RandomSequence` była wykorzystana dokładnie raz: do utworzenia zwracanej wartości. W tym przypadku możesz utworzyć klasę anonimową:

```
public static IntSequence randomInts(int low, int high) {
    return new IntSequence() {
        public int next() { return low + generator.nextInt(high - low + 1); }
        public boolean hasNext() { return true; }
    }
}
```

Wyrażenie

```
new Interfejs() { metody }
```

oznacza: zdefiniuj klasę implementującą interfejs, który ma dane metody, i skonstruuj jeden obiekt tej klasy.

Jak zawsze nawiasy `()` w wyrażeniu `new` oznaczają parametry konstruktora. Wywoływany jest domyślny konstruktor klasy anonimowej.

Zanim w języku Java pojawiły się wyrażenia lambda anonimowe, klasy wewnętrzne były najbardziej zgrabną składnią umożliwiającą tworzenie obiektów funkcjonalnych z interfejsami `Runnable` czy komparatorów. Przeglądając stary kod, będziesz je często spotykać.

Obecnie są one potrzebne jedynie w sytuacji, gdy musisz utworzyć dwie lub więcej metod, jak w powyższym przykładzie. Jeśli interfejs `IntSequence` ma domyślną metodę `hasNext`, jak w ćwiczeniu 15., możesz po prostu wykorzystać wyrażenie lambda:

```
public static IntSequence randomInts(int low, int high) {
    return () -> low + generator.nextInt(high - low + 1);
}
```

Ćwiczenia

1. Utwórz interfejs `Measurable` z metodą `double getMeasure()`, która dostarcza jakąś metrykę obiektu. Zaimplementuj interfejs `Measurable` w klasie `Employee`. Utwórz metodę `double average(Measurable[] objects)`, która oblicza średnią metryk. Wykorzystaj ją do obliczenia średniego wynagrodzenia pracowników, których dane są zapisane w tablicy.
2. Kontynuując poprzednie ćwiczenie, utwórz metodę `Measurable largest(Measurable[] objects)`. Wykorzystaj ją do ustalenia nazwiska pracownika z najwyższym wynagrodzeniem. Do czego użyjesz rzutowania?
3. Jaki jest typ nadrzędny dla typu `String`? Dla typu `Scanner`? Typu `ImageOutputStream`? Zauważ, że każdy typ ma typ nadrzędny. Klasa lub interfejs bez zadeklarowanego typu nadrzędnego otrzymuje jako typ nadrzędny `Object`.
4. Zaimplementuj statyczną metodę `of` w klasie `IntSequence`, która zwraca ciąg parametrów. Na przykład `IntSequence.of(3, 1, 4, 1, 5, 9)` zwraca ciąg sześciu wartości. Dodatkowe punkty możesz dostać za zwrócenie instancji wewnętrznej klasy anonimowej.
5. Zaimplementuj metodę statyczną `constant` w klasie `IntSequence`, która zwraca nieskończony ciąg stałych. Na przykład `IntSequence.constant(1)` zwraca wartości `1 1 1 ...`, w nieskończoność. Dodatkowe punkty za wykonanie tego za pomocą wyrażenia lambda.
6. W tym ćwiczeniu masz za zadanie sprawdzić, co się stanie po dodaniu metody do interfejsu. W Java 7 zaimplementuj klasę `DigitSequence`, która implementuje `Iterator<Integer>`, nie `IntSequence`. Utwórz metody `hasNext`, `next` i nierobiącą niczego metodę `remove`. Napisz program, który wyświetla elementy instancji. W Java 8 klasa `Iterator` ma inną metodę, `forEachRemaining`. Czy Twój kod nadal się kompiluje po przejściu na Java 8? Jeśli umieścisz klasę z Java 7 w pliku JAR i nie będziesz jej kompilował ponownie, czy zadziała w Java 8? A co, jeśli wywołasz metodę

`forEachRemaining`? Poza tym metoda `remove` stała się domyślną metodą w Java 8 wywołującą wyjątek `UnsupportedOperationException`. Co się stanie, jeśli metoda `remove` zostanie wywołana na instancji Twojej klasy?

7. Zaimplementuj metodę `void luckySort(ArrayList<String> strings, Comparator<String> comp)`, która wywołuje `Collections.shuffle` na tablicy typu `ArrayList` do chwili, gdy elementy będą uporządkowane rosnąco w sposób określony przez komparator.

8. Zaimplementuj klasę `Greeter`, która implementuje interfejs `Runnable` i w której metoda `run` wyświetla `n` kopii tekstu `"Witaj, " + target`, gdzie `n` i `target` są ustawiane w konstruktorze. Stwórz dwie instancje z różnymi komunikatami i wykonaj je równolegle w dwóch wątkach.

9. Zaimplementuj metody:

```
public static void runTogether(Runnable... tasks)
public static void runInOrder(Runnable... tasks)
```

   Pierwsza metoda powinna uruchomić każde zadanie w oddzielnym wątku i zakończyć działanie. Druga metoda powinna uruchomić wszystkie zadania w bieżącym wątku i zakończyć działanie po zakończeniu ostatniego z nich.

10. Korzystając z metod `listFiles(FileFilter)` i `isDirectory` z klasy `java.io.File`, napisz metodę zwracającą wszystkie podkatalogi wskazanego katalogu. Wykorzystaj wyrażenie lambda zamiast obiektu `FileFilter`. Wykonaj to samo za pomocą wyrażenia z metodą i anonimowej klasy wewnętrznej.

11. Korzystając z metody `list(FilenameFilter)` klasy `java.io.File`, napisz metodę zwracającą wszystkie pliki ze wskazanego katalogu ze wskazanym rozszerzeniem. Użyj wyrażenia lambda, a nie `FilenameFilter`. Jakie zmienne zewnętrzne wykorzystasz?

12. Mając tablicę obiektów `File`, posortuj je w taki sposób, by katalogi znalazły się przed plikami, a w każdej grupie elementy zostały posortowane według nazwy. Użyj wyrażenia lambda przy implementowaniu interfejsu `Comparator`.

13. Napisz metodę, która pobiera tablicę instancji klas implementujących interfejs `Runnable` i zwraca instancję `Runnable`, której metoda `run` wykonuje kolejno kod instancji obiektów zapisanych w tablicy. Zwróć wyrażenie lambda.

14. Napisz wywołanie `Arrays.sort`, które sortuje pracowników według wynagrodzenia, a w przypadku takich samych wynagrodzeń według nazwiska. Użyj `Comparator.thenComparing`. Następnie wykonaj to samo w odwrotnej kolejności.

15. Zaimplementuj `RandomSequence` z podrozdziału 3.9.1, „Klasy lokalne”, jako klasę zagnieżdżoną poza metodą `randomInts`.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

JAVA 8

Przewodnik doświadczonego programisty

Java to język, który zrewolucjonizował świat programistów.

Jej możliwości zostały docenione przez największe firmy. Dziś Java jest wykorzystywana w najbardziej zaawansowanych projektach. Wszędzie tam, gdzie są wymagane najwyższa wydajność i bezpieczeństwo, nie ma sobie równych! Aktualna wersja tego języka wprowadza wiele nowych konstrukcji i usprawnień — jeżeli chcesz błyskawicznie je poznać, zacznij od tej książki.

Została ona napisana specjalnie z myślą o osobach chcących rozpocząć kodowanie z wykorzystaniem nowości z Javy 8. Jest przeznaczona dla doświadczonych programistów, lecz zawiera też podstawowe informacje na temat technik programowania obiektowego, wyjątków, typów i składni. Dzięki temu możesz błyskawicznie wdrożyć się w nowy język programowania! Natomiast jeśli znasz już Javę, z pewnością zainteresują Cię informacje o wyrażeniach lambda, projekcie Nashorn oraz nowym API do operacji na datach i czasie (*JSR 310*). Książka ta jest najlepszą lekturą dla wszystkich programistów zainteresowanych językiem Java oraz nowościami w Javie 8!

Dzięki tej książce:

- poznasz techniki programowania obiektowego
- zaznajomisz się z podstawowymi konstrukcjami języka
- zrozumiesz działanie wyrażeń lambda
- wykorzystasz nowe możliwości API dla daty i czasu
- przekonasz się, jak szybki jest silnik Nashorn
- zastosujesz najnowsze wydanie języka Java w swoim projekcie

Nowości w Javie 8!

Poznaj możliwości nowej Javy!

Cay S. Horstmann — profesor informatyki oraz wykładowca na Uniwersytecie Stanowym San Jose, Java Champion. Autor książek dla programistów oraz studentów, w tym bestsellerów *Java. Podstawy* oraz *Java. Techniki zaawansowane*.

Helion

37560 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 978-83-283-1333-0

9 788328 313330

Informatyka w najlepszym wydaniu

cena: 69,00 zł