10.1. Wprowadzenie

Być może najważniejszą cechą programowania funkcyjnego jest możliwość pisanie programów w kategoriach „co ma być osiągnięte", a nie – jak w programowaniu imperatywnym – przez specyfikowanie kolejnych kroków algorytmu do wykonania. Takie podejście jest możliwe, gdy w języku programowania możemy traktować fragmenty kodu (funkcje) jako pełnoprawne obiekty, które mogą być przekazywane innym funkcjom i zwracane z innych funkcji. W czystych językach funkcyjnych tak właśnie się dzieje, co więcej, funkcje nie zmieniają żadnych danych (stanów), ważne jest jedynie wyliczanie ich wyników na podstawie podanych argumentów (co ma duże znaczenie np. przy przetwarzaniu równoległym).

W językach bez wsparcia dla programowania funkcyjnego w pewnym stopniu także możemy pisać programy w sposób funkcyjny.

Rozważmy przykład. Mamy listę liczb `src`. Na jej podstawie chcemy utworzyć listę `target` zawierającą kwadraty liczb z listy `src`, które są mniejsze od 10. Rozwiązanie wydaje się proste:

```
List<Integer> src = Arrays.asList(5, 72, 10, 11, 9);
List<Integer> target = new ArrayList<>();
for (Integer n : src) {
  if (n < 10) {
    target.add(n * n);
  }
}
```

Ale kod nie jest zbyt ogólny, uniwersalny. Jest imperatywny, bo musieliśmy dokładnie zapisać każdy krok algorytmu. Co się stanie, jeśli zmienimy warunek selekcji liczb (np. biorąc pod uwagę tylko liczby parzyste) albo operację, która ma być wykonana na wybranych liczbach (np. zamiast podniesienia do kwadratu dodanie jakiejś liczby)? Musielibyśmy za każdym razem pisać podobne pętle jak wyżej. A gdyby trzeba było z listy napisów wybrać te o określonej długości i przekształcić je np. do wielkich liter? Znowu trzeba by było pisać podobne pętle tym razem operujące na listach napisów. A przecież zawsze chodzi o wybór elementów z listy `src` do przetwarzania, wykonanie na nich określonej operacji i dodanie wyników do listy `target`. Przydałaby się pewnie metoda `create`, której moglibyśmy używać np. tak:

```
List<jakiś_typ> target = create( src,
                                 tu podać warunek wyboru elementów z listy src,
                                 tu podać operację na wybranych elementach);
```

To już przypomina styl funkcyjny. Mówimy tylko, co chcemy osiągnąć i najczęściej nie musimy pisać imperatywnego kodu. Ale jak podać warunek wyboru i operację? Jasne jest, że to muszą być jakieś fragmenty kodu, które wykonują odpowiednie obliczenia i zwracają wyniki. Czyli – ogólnie mówiąc – funkcje, które przekazujemy metodzie `create`, a ona będzie je wywoływać.

Czy w Javie przed wersją 8 moglibyśmy coś takiego zrobić? Ależ tak, widzieliśmy to już niejednokrotnie, gdy np. przekazywaliśmy kod do wykonania w wątku jako obiekt

implementujący interfejs `Runnable` czy `Callable` albo gdy podawaliśmy komparatory (obiekty implementujące interfejs `Comparator`) przy sortowaniu list. Potrzebne są więc odpowiednie interfejsy.

Zdefiniujmy interfejs `Filter`, którego metoda `test()` będzie odpowiedzialna za wybór elementów listy (zwraca `true`, jeśli przekazana jej wartość spełnia zapisane w kodzie tej metody warunki, `false` – w przeciwnym razie):

```
public interface Filter<V> {
  boolean test(V v);
}
```

oraz interfejs `Transformer`, którego metoda `transform()` ma wykonać operację na przekazanej wartości i zwrócić jej wynik:

```
public interface Transformer<T,S> {
  T transform(S v);
}
```

Napisanie metody `create()` w kategoriach tych interfejsów jest całkiem łatwe (kod 10.1).

```
static <T, S> List<T> create(List<S> src, Filter<S> f, Transformer<T, S> t) {
  List<T> target = new ArrayList<>();
  for (S e : src)
    if (f.test(e)) target.add(t.transform(e));
  return target;
}
```

Kod 10.1. Metoda z parametrami oznaczającymi kody do wykonania

W programach pisanych w Javie przed wersją 8 takiej metodzie `create(...)` trzeba było przekazywać obiekty implementujące podane interfejsy, np:

```
List<Integer> src = Arrays.asList(5, 72, 10, 11, 9);
List<Integer> target = create(src,
    new Filter<Integer>() {
      public boolean test(Integer n) {
        return n < 10;
      }
    },
    new Transformer<Integer, Integer>() {
      public Integer transform(Integer n) {
        return n*n;
      }
    });
```

To już jest trochę w stylu funkcyjnym, ale mało czytelne, obarczone liniami kodu, które nic nie wnoszą (tworzenie obiektów anonimowych klas wewnętrznych tylko po to, by przekazać kod do wykonania). Tak naprawdę ważne są tu przecież tylko dwie linijki:

```
return n < 10;
```

i

```
return n*n;
```

Między innymi dlatego w Javie w wersji 8 wprowadzono, co prawda w ograniczonym zakresie, wsparcie dla programowania funkcyjnego. Obejmuje ono przede wszystkim ***lambda--wyrażenia*** oraz ***przetwarzanie strumieniowe***.

Zanim poznamy w następnym punkcie lambda-wyrażenia w szczegółach, możemy o nich myśleć jako o bezpośrednio podawanych fragmentach kodu, swoistych funkcjach bez nazwy, które są traktowane „jak prawdziwe obiekty", czyli np. mogą być przekazywane metodom. Dzięki zastosowaniu lambda-wyrażeń poprzedni przykład wywołania metody `create()` upraszczamy praktycznie do jednej linijki:

```
List<Integer>
 target = create(src, n -> n < 10, n -> n*n );
```

Zapis

```
n -> n < 10
```

jest lambda-wyrażeniem, które możemy traktować jako funkcję anonimową, której parametrem jest `n` i która zwraca wynik wyrażenia `n<10`. Nie wchodząc na razie w szczegóły, można powiedzieć, że kompilator dopasowuje to lambda-wyrażenie do drugiego parametru (`Filter<S>`) metody `create(...)`, w wyniku czego użyta tam metoda `test(...)` interfejsu `Filter` uzyskuje odpowiednią implementację (`boolean test(Integer n) { return n < 10; }`). Podobnie dzieje z lambda-wyrażeniem

```
n -> n*n
```

pasującym do wywołania metody `transform(...)` interfejsu `Transformer`.

Przykłady wykorzystania lambda-wyrażeń w różnych kontekstach przedstawia kod 10.2.

```
List<Integer> num = Arrays.asList( 1, 3, 5, 10, 9, 12, 7);
List<String> txt = Arrays.asList("ala", "ma", "kota",
                   "aleksandra", "psa", "azora" );
List<Employee> emp = Arrays.asList(
        new Employee("Kowal", "Jan", 34, 3400.0),
        new Employee("As", "Ala", 27, 4100.0),
        new Employee("Kot", "Zofia", 33, 3700.0),
        new Employee("Puchacz", "Jan", 41 , 3600.0)
        );

System.out.println(
  create(num, n-> n%2!=0, n->n*100)
);

System.out.println(
  create(txt,
         s -> s.startsWith("a"),
         s -> s.toUpperCase() + " " + s.length())
);

List<Employee> doPodwyzki =
    create(emp,
```

```
        e -> e.getAge() > 30 && e.getSalary() < 4000,
        e -> e
      );
System.out.println("Podwyzki powinni uzyskac:");
System.out.println(doPodwyzki);
```

Kod 10.2. Przykład zastosowania lambda-wyrażeń

Uwaga. W kodzie 10.2 wykorzystano prostą klasę `Employee` z polami `lname` (nazwisko), `fname` (imię), `age` (wiek) i `salary` (pensja) oraz metodą `toString()` zwracającą nazwisko i imię. Kod 10.2 wyprowadzi na konsolę następujące wyniki:

```
[100, 300, 500, 900, 700]
[ALA 3, ALEKSANDRA 10, AZORA 5]
Podwyzki powinni uzyskac:
[Kowal Jan, Kot Zofia, Puchacz Jan]
```

Jak widać, stworzona wcześniej metoda `create()` jest wykorzystywana dosyć elastycznie (dla różnych typów danych, różnych warunków, różnych operacji), a dzięki lambda-wyrażeniom sformułowanie tego, co chcemy osiągnąć, jest bardzo proste i czytelne. Jednak metoda `create(...)` nie do końca jest uniwersalna. Być może chcielibyśmy najpierw przetworzyć dane, a filtrować wyniki tego przetwarzania. A może filtrować, przetwarzać i znowu filtrować? Może tylko filtrować, np. zastosowanie `create` we fragmencie kodu 10.2 dla listy pracowników jest nieco sztuczne. Praktycznie nie ma tam transformacji, operacja przetwarzania – metoda `transform()`, będąca pod spodem ostatniego lambda-wyrażenia – zwraca tę samą wartość, którą uzyskała jako argument. I czy zawsze potrzebujemy utworzenia nowej listy (`target`), może wystarczy filtrowanie i przetwarzanie elementów istniejącej (`src`)?

Te wszystkie przypadki możemy łatwo obsłużyć za pomocą ***przetwarzania strumieniowego***. Zanim zapoznamy się z nim dokładniej, tu – na zachętę – pokażemy szczególne przypadki jego użycia, dobrze zastępujące i znacznie poszerzające funkcjonalność stworzonej wcześniej metody `create(...)`, i to bez konieczności definiowania własnych interfejsów.

W kontekście wcześniejszych przykładowych kodów sekwencja działań jest następująca. Od listy metodą `stream()` uzyskujemy strumień (zwany sekwencją), a na danych strumienia możemy wykonywać m.in. operacje przetwarzania (metoda `map`), filtrowania (metoda `filter`) oraz uzyskiwać nowe kolekcje wyników tych operacji (metoda `collect`).

Metodzie `map` podajemy lambda-wyrażenie, którego wynikiem jest przetworzenie jego parametru (działa to tak jak `transform()` z interfejsu `Transformer`). Metoda zwraca strumień, na którym można wykonywać dalsze operacje. Metodzie `filter` podajemy lambda-wyrażenie przekształcające parametr w wartość boolowską (tak jak `filter()`). Filtrowanie zwraca strumień, który pozwala wykonywać dalsze operacje tylko na tych danych, dla których wynik lambdy jest `true`. Możemy zatem wywoływać sekwencje `map-filter`, realizujac kolejne etapy przetwarzania strumienia danych. Metoda `collect` natomiast kończy przetwarzanie strumienia i ma za parametr obiekt klasy `Collector`, który tworzy nową kolekcję i dodaje do niej dane ze strumienia, na rzecz którego została wywołana. Taki kolektor, budujący listę, można uzyskać za pomocą statycznej metody `Collectors.toList()`.

Na tej podstawie możemy tak zapisać poprzedni przykład uzyskania listy kwadratów tych elementów listy `src`, które są mniejsze od 10:

```
import static java.util.stream.Collectors.toList;
// ...
List<Integer> src = Arrays.asList(5, 72, 10, 11, 9);
List<Integer> target = src.stream()
                          .filter(n -> n < 10)
                          .map(n -> n * n)
                          .collect(toList());
```

Może trochę więcej tu kodu niż poprzednio, ale za to nie musimy definiować własnych interfejsów (metody strumieniowe korzystają z interfejsów z pakietu `java.util.function`, które pokrywają częste przypadki użycia lambda-wyrażeń; rozdz. 10.3). Ale co najważniejsze, mamy dużo większą swobodę działania. Możemy np. filtrować kwadraty liczb, czyli najpierw wykonać `map()`, a później `filter()`:

```
List<Integer> num = Arrays.asList(1, 3, 5, 10, 9, 12, 7);
List<Integer>
          out = num.stream()
                   .map(n -> n*n)
                   .filter(n -> n > 80)
                   .collect(toList());
```

co da w wyniku listę liczb:

```
[100, 81, 144]
```

Albo z listy napisów wybrać trzycyfrowe reprezentacje liczb całkowitych, przekształcić je w liczby, wybrać parzyste i utworzyć nową listę ich napisowych reprezentacji:

```
List<String> snum = Arrays.asList("7", "20", "160", "777", "822");
snum = snum.stream()
         .filter(s -> s.length() == 3)
         .map (s-> Integer.parseInt(s))
         .filter(n-> n%2 == 0)
         .map(n-> String.valueOf(n))
         .collect(toList());
```

Powyższy kod da w wyniku listę, złożoną z dwóch napisów `"160"` i `"822"`.

Możemy też ulepszyć poprzedni przykład (kod 10.2), który selekcjonował pracowników do podwyżki. Teraz nie tylko zostanie wybrana odpowiednia grupa pracowników, ale też podwyżki zostaną zrealizowane:

```
List<Employee> emp = ...
emp.stream()
   .filter(e -> e.getAge() > 30 && e.getSalary() < 4000)
   .forEach(e -> e.setSalary(e.getSalary()*1.1));

emp.forEach(
   e -> System.out.printf("%s %.0f\n", e, e.getSalary())
);
```

Ten fragment wyprowadzi na konsolę:

```
Kowal Jan 3740
As Ala 4100
Kot Zofia 4070
Puchacz Jan 3960
```

Warto zwrócić uwagę, że zastosowano tu dwa razy metodę `forEach`. Pierwsze jej użycie dotyczyło strumienia zwracanego przez `filter(...)` i polegało na wykonaniu podanej operacji (ustalenia nowej pensji) na wszystkich elementach zwróconej sekwencji. Metoda `forEach`, tak jak `collect`, jest jedną z metod, która kończy operacje strumieniowe. Drugie `forEach` zastosowano wobec kolekcji (listy pracowników) do wyprowadzenia informacji. Jest to jeden z przykładów nowych (domyślnych) metod w interfejsach kolekcyjnych, którym jako argumenty można podawać lambda-wyrażenia odpowiednich rodzajów. O tych metodach będzie mowa w rozdz. 10.4.

Strumienie w Javie 8 są ważną, całkiem odmienną od kolekcji, koncepcją. Umożliwiają m.in. „leniwe wyliczanie" (czyli obliczanie tylko tego, co jest w danym kontekście potrzebne) oraz łatwe zrównoleglanie obliczeń. Ponadto strumienie można „nakładać" nie tylko na kolekcje, ale też na wiele innych rodzajów obiektów. O tym wszystkim będzie mowa w rozdz. 10.6.

10.2. Interfejsy funkcyjne i lambda-wyrażenia

Jeżeli zestaw metod, deklarowanych w interfejsie, zawiera tylko jedną metodę abstrakcyjną (wyłączając dodane deklaracje abstrakcyjnych publicznych niefinalnych metod z klasy Object), to taki interfejs nazywa się ***funkcyjnym*** *(SAM – Single Abstract Method).*

Przykłady interfejsów funkcyjnych:

```
interface SAMA {
  void doWork(int n);
}

interface SAMB {
  void doWork(String s);

  @Override
  boolean equals(Object);
}

interface SAMC {
  void doWork(Collection c);
  default public void done() { System.out.println("Done"); }
}
```

Są to wszystko interfejsy funkcyjne, bo

- `SAMA` zawiera jedną metodę abstrakcyjną,
- `SAMB` zawiera w deklaracji dwie metody abstrakcyjne, ale metoda `equals` przedefiniowuje metodę `equals` z klasy `Object`,
- `SAMC` zawiera dwie metody, ale tylko jedna z nich jest abstrakcyjna (`doWork`), druga (`done`) jest domyślna i ma implementację.

Interfejsy funkcyjne zostały wybrane jako środek realizacji lambda-wyrażeń w Javie 8. Każde lambda-wyrażenie musi odpowiadać jakiemuś interfejsowi funkcyjnemu i jest typu wyznaczanego przez tego rodzaju interfejsy. W doborze odpowiedniego interfejsu funkcyjnego do lambda-wyrażenia kompilator stosuje ***deskryptor funkcji*** (*function descriptor*) interfejsu funkcyjnego, który opisuje parametry typu (gdy interfejs jest sparametryzowany) oraz typy parametrów, typ wyniku i ew. typ zgłaszanego wyjątku metody interfejsu.

Definiując interfejsy funkcyjne, warto stosować adnotację `@FunctionalInterface`, dzięki której

- kompilator może sprawdzić, czy rzeczywiście dany interfejs jest funkcyjny i poinformować nas o ew. błędach;
- w dokumentacji i kodzie widać od razu, że dany interfejs może być stosowany z lambda-wyrażeniami.

Składnia lambda-wyrażenia jest następująca:

```
(lista_parametrów) -> ciało_lambda
```

gdzie

- `lista_parametrów` oznacza rozdzielone przecinkami deklaracje argumentów przekazywanych lambda-wyrażeniu (jak w nagłówku metody);
- `ciało_lambda` to jedno wyrażenie lub blok instrukcji, ujęty w nawiasy {}.

Uwaga. Jeśli typy parametrów mogą być określone przez kompilator w wyniku konkludowania typów (typy `inferring`), to w deklaracji parametrów można pominąć typy, a wtedy, jeśli parametr jest jeden, można pominąć nawiasy okalające listę parametrów.

Rozważmy prosty przykład lambda-wyrażenia, które otrzymuje jako argument liczbę całkowitą i zwraca jej wartość powiększoną o 1:

```
(int n) -> { return n+1; }
```

Aby w ogóle coś takiego zapisać w programie, musimy mieć odpowiedni interfejs funkcyjny. Załóżmy, że mamy następujący interfejs (widoczny w kontekście kompilacji):

```
interface IntProcessor {
  int process(int n);
}
```

Wtedy tego lambda-wyrażenia możemy użyć wszędzie tam, gdzie może wystąpić typ `IntProcessor`, czyli m.in. w deklaracji zmiennej, przy przypisaniach czy przy przekazywaniu argumentów i zwrocie wyników (kod 10.3).

```
class LamSample {
// ...
  static void arrOp(int[] arr, IntProcessor p) {
    // ...
  }

  static IntProcessor getIntProcessor() {
    return (int n) -> { return n + 1; };
  }

  public static void main(String[] args) {
    IntProcessor p = (int n) -> { return n + 1; }; // 1
    int[] arr = {1, 2, 3 };                        // 2
    arrOp(arr, (int n) -> { return n + 1; } );     // 3
    IntProcessor p1 = getIntProcessor(); // 3
    // ...
  }
}
```

Kod 10.3. Użycie lambda-wyrażeń w przypisaniach, przy przekazywaniu argumentów i zwrocie wyników z metod

Zwróćmy przede wszystkim uwagę na to, że w programie 10.3 podanie lambda-wyrażeń w wierszach z komentarzami //1, //2, //3 metody `main` jest swoistą deklaracją i nie powoduje jeszcze wykonania instrukcji ciała lambdy. Obliczenie lambda-wyrażenia nastąpi dopiero podczas wywołania metody interfejsu `IntProcessor`. Ale skąd ma się wziąć jej implementacja? Otóż, przy przetwarzaniu kodu źródłowego, kompilator natykając się na lambda-wyrażenie, określa jego typ (tzw. *target type*) na podstawie kontekstu użycia. W przykładzie jest to proste:

- w wierszu //1 typem lambda-wyrażenia jest `IntProcessor`, bo taka jest deklaracja zmiennej;
- w wierszu //2 typem jest `IntProcessor`, bo tak jest określony drugi argument metody `arrOp`.

Jednocześnie kompilator dodaje do klasy kod, który w fazie wykonania programu wesprze utworzenie odpowiedniej implementacji metody interfejsu funkcyjnego (w naszym przypadku `int process(int a) {return a +1;}`). Wykonanie (obliczenie) lambda-wyrażenia odbywa się przez wywołanie tej metody, co pokazuje kod 10.4.

```
static void arrOp(int[] arr, IntProcessor p) {
  for (int i = 0; i < arr.length; i++) {
    arr[i] = p.process(arr[i]); // wykonanie lambda
  }
}
```

```
public static void main(String[] args) {
  IntProcessor p = (int n) -> { return n + 1; };
  int x = p.process(1); // wykonanie lambda
  System.out.println(x);
  int[] arr = {1, 2, 3 };
  arrOp(arr, (int n) -> { return n + 1; } );
  System.out.println(Arrays.toString(arr));
}
```

Kod 10.4. Wykonanie lambda-wyrażeń

W wyniku otrzymamy:

```
2
[2, 3, 4]
```

Mamy tu do czynienia z lambda-wyrażeniem, które zwraca wynik. Mogą być też takie, które wyników nie zwracają, wtedy typem ich wyniku jest `void` (lub typ kompatybilny z `void`).

Nasze przykładowe lambda-wyrażenie możemy znacznie uprościć.

- Na podstawie deskryptora funkcji metody interfejsu funkcyjnego `IntProcessor` kompilator może konkludować o typie parametru, więc w lambdzie możemy pominąć typ parametru.
- Ponieważ jest tylko jeden parametr, więc możemy pominąć nawiasy okrągłe.
- Ciało lambda-wyrażenia możemy zapisać jako jedno wyrażenie, wtedy niepotrzebne są nawiasy klamrowe, a wynikiem lambdy jest wynik tego wyrażenia.

W ten sposób uzyskujemy dość zwięzłe i czytelne kody źródłowe, np:

```
arrOp(arr, n -> n+1);
arrOp(arr, n -> n*10);
```

Wyobraźmy sobie teraz, że oprócz interfejsu `IntProcessor` w kontekście kompilacji jest też widoczny interfejs funkcyjny `StringProcessor`. Dodajmy metodę, która wykorzystuje go do przetwarzania tablicy napisów (kod 10.5).

```
interface IntProcessor {
  int process(int n);
}

interface StringProcessor {
 String process(String n);
}

public class Sample {

  static void arrOp(int[] arr, IntProcessor p) {
    for (int i = 0; i < arr.length; i++) {
      arr[i] = p.process(arr[i]);
    }
  }
```

```
  static void sarrOp(String[] arr, StringProcessor p) {
    for (int i = 0; i < arr.length; i++) {
      arr[i] = p.process(arr[i]);
    }
  }

  public static void main(String[] args) {
    // ...
  }
}
```

Kod 10.5. Typy lambda-wyrażeń

Jakiego typu jest lambda-wyrażenie: `n -> n + 1`? Zależy to od kontekstu użycia. Gdy w metodzie `main` napiszemy np.

```
int[] arr = {1, 2, 3 };
arrOp(arr, n -> n+1);
```

to na podstawie konkludowania typów lambda-wyrażenie będzie typu `IntProcessor`, a gdy napiszemy

```
String[] sarr = {"Ala", "kot", "pies" };
sarrOp(sarr, n -> n+1);
```

to, w wyniku konkluzji, typem tego lambda stanie się `StringProcessor`.

Mamy tu do czynienia z ***wyrażeniami wioelotypowymi*** (*polytype expressions*), czyli takimi, które mogą mieć różne typy w zależności od kontekstu użycia. Lambda-wyrażenia stanowią jeden z przypadków wyrażeń wielotypowych w Javie. Stanie się to jeszcze bardziej widoczne (i bardziej funkcjonalne), gdy nieco uogólnimy kody, stosując metody sparametryzowane (kod 10.6).

```
interface Processor<T> {
  T process(T val);
}

public class ArrProc {

  public static <T> void processArr(T[] arr, Processor<T> p ) {
    for (int i = 0; i < arr.length; i++) {
      arr[i] = p.process(arr[i]);
    }
  }

  public static void main(String[] args) {
    String[] sa = { "a", "b", "c" };
    Integer[] ia = {1, 2, 3};
    processArr(sa, v -> v+1); // 1
    processArr(ia, v -> v+1); // 2
    processArr(sa, v-> v.toUpperCase()); // 3
```

```
    processArr(ia, v -> v*v);
    System.out.println(Arrays.toString(sa));
    System.out.println(Arrays.toString(ia));
  }

}
```

Kod 10.6. Lambda-wyrażenia – konkludowanie typów

W pewnym uproszczeniu możemy wyjaśnić ten kod następująco (szczegóły są bardziej skomplikowane – zob. *Java Language Specification*). W wierszach oznaczonych //1, //2, //3 kompilator stwierdza, że jest wywoływana metoda sparametryzowana, wobec tego poszukuje odpowiedniego sparametryzowanego interfejsu funkcyjnego. Konkludowane typy parametrów lambda-wyrażenia oraz jego wyniku wyznaczają aktualny argument typu, a także uczestniczą w doborze „target type" lambda-wyrażenia (dopasowanie do deskryptora funkcji interfejsu funkcyjnego). Do ciała klasy zostanie przez kompilator dodany kod, który w fazie wykonania umożliwi stworzenie odpowiednich implementacji metody `process` z interfejsu `Processor` i to one będą wywoływane w pętli, zawartej w metodzie `processArr`. W rezultacie kod 10.6 wyprowadzi

```
[A1, B1, C1]
[4, 9, 16]
```

Jak już widzieliśmy w poprzednim punkcie, lambda-wyrażenia można stosować jako uproszczenie w użyciu anonimowych klas wewnętrznych, implementujących interfejsy funkcyjne. Podobnie jak w lokalnych klasach wewnętrznych lambda-wyrażenie ma dostęp

- do wszystkich składowych klasy otaczającej, przy czym pola tej klasy mogą być przez lambda-wyrażenie modyfikowane;
- do zmiennych lokalnych, ale muszą one albo być deklarowane ze specyfikatorem **final** albo być efektywnie finalne (*effective final*, por. rozdz. 8.7).

Inaczej niż w anonimowych klasach wewnętrznych lambda-wyrażenie nie wprowadza nowego zasięgu widoczności i dostępności, co oznacza m.in., że

- zmienna `this` jest referencją do obiektu klasy, na rzecz którego jest wywoływana metoda (lub konstruktor) z lambda-wyrażeniem (w klasie anonimowej `this` oznacza obiekt tej klasy);
- słowo kluczowe **super** oznacza nadklasę obiektu (klasy), na rzecz którego jest wywoływana metoda zawierająca lambda-wyrażenie;
- jeśli lambda-wyrażenie występuje w bloku, niedopuszczalne jest przesłanianie zmiennych lokalnych z bloku otaczającego (w klasie lokalnej anonimowej mamy nowy zasięg widoczności, więc możemy używać tych samych identyfikatorów zmiennych, co w bloku otaczającym).

Ilustruje to kod 10.7.

```
interface Worker {
  void work();
}

public class ScopeLambda {

  void doWork(Worker w) {
    w.work();
  }

  void defineAndDoWork(String[] s) {
    String pref = "( ";
    String suff = " )";
    // 1 int i = 7; spowoduje błąd w kompilacji, bo w lambda przesłonięte
    // 2 suff = "*"; spowoduje błąd w kompilacji, bo nieefektywnie finalne
    doWork ( () -> { // lambda bez parametrów
      System.out.println(this.getClass());
      for (int i = 0; i < s.length; i++) {
        s[i] = pref + s[i] + suff;
      }
    });
  }

  public static void main(String[] args) {
    String[] s = { "A", "B", "C" };
    new ScopeLambda().defineAndDoWork(s);
    System.out.println(Arrays.toString(s));
    int i = 7, k = 10;
    // k = 11; spowoduje błąd w kompilacji, bo nieefektywnie finalne
    Worker w = new Worker() {
      int i = 8; // nie ma błędów w kompilacji
      public void work() {
        System.out.println(k);
        int i = 9; // nie ma błędów w kompilacji
        System.out.println(this.getClass());
      }
    };
    w.work();
  }

}
```

Kod 10.7. Podobieństwa i różnice między lambda-wyrażeniem a iplementacją interfejsu funcyjnego w klasie wewnętrznej

Ponieważ wiersze powodujące błędy w kompilacji zawierają komentarze, kod 10.7 wyprowadzi:

```
class ScopeLambda
[( A ), ( B ), ( C )]
10
class ScopeLambda$1
```

Warto zauważyć, że realizacja lambda-wyrażeń w Javie ma istotną zaletę. Jak widać z wyników działania programu 10.7, przy implementacji interfejsu funkcyjnego w anonimowej klasie wewnętrznej po kompilacji powstała dodatkowa klasa o nazwie `Scope Lambda$1` i odpowiadający jej plik klasowy na dysku. W przeciwieństwie do tego i inaczej niż w niektórych innych językach JVM w Javie przy opracowaniu lambda-wyrażeń przez kompilator nie są tworzone i zapisywane w plikach klasowych żadne dodatkowe klasy.

Warto wreszcie podsumować krótko najważniejsze informacje, dotyczące cech oraz składni lambda-wyrażeń w Javie.

- Lambda-wyrażenie może być użyte tylko w następujących kontekstach:
  - w inicjatorze deklaracji zmiennej, również przy inicjacji tablic,
  - w przypisaniu,
  - jako argument wywołania metody lub konstruktora,
  - jako wynik zwracany z metody,
  - w wyrażeniu warunkowym ?:,
  - w wyrażeniu rzutowania.
- Za lambda-wyrażeniem zawsze stoi jakiś interfejs funkcyjny, lambda-wyrażenie jest typu wyznaczanego przez ten interfejs.
- Doboru interfejsu funkcyjnego dokonuje kompilator na podstawie konkludowania typów i porównań z deskryptorem funkcji interfejsu funkcyjnego.
- Odpowiednia implementacja interfejsu funkcyjnego dla lambda-wyrażenia tworzona jest dynamicznie w fazie wykonania programu i nie prowadzi do powstania nowego pliku klasowego.
- Lambda-wyrażenie nie wprowadza nowego zasięgu widoczności i dostępności zmiennych, jest ulokowane zawsze w kontekście klasy i metody, w której jest definiowane.
- Jeżeli kompilator może konkludować (`infer`) typy parametrów lambda-wyrażenia, to na liście parametrów typy możemy pominąć.
- Na liście parametrów nie wolno mieszać deklaracji parametrów bez typu (konkludowanych) i deklaracji parametrów z podanym typem.
- Jeśli mamy jeden parametr bez typu, to nawiasy okalające listę parametrów lambda-wyrażenia można pominąć.
- Jeśli lambda-wyrażenie nie ma parametrów, to jako listę parametrów podajemy puste nawiasy okrągłe.
- Jeśli ciało lambda-wyrażenia składa się z jednego wyrażenia, to nie musimy stosować nawiasów klamrowych, okalających je.
- Jeśli ciało lambda-wyrażenia jest instrukcją lub zestawem instrukcji, to okalamy je nawiasami klamrowymi.
- Lambda wyrażenie może zwracać wynik lub nie; jeśli nie zwraca wyniku, jego typem jest `void` lub typ kompatybilny z `void`).

10.3. Referencje do metod i konstruktorów

Często definicja lambda-wyrażenia sprowadza się do wywołania metody istniejącej klasy lub utworzenia jej obiektu. Wtedy zamiast wywołania tej metody w lambda-wyrażeniu możemy zastosować referencję do metody lub konstruktora. Są cztery rodzaje takich referencji (symbolicznie przez `x` oznaczamy jeden parametr lub listę parametrów w nawiasach okrągłych):

- NazwaKlasy::nazwa_metody_statycznej
  - Klasa::metStat – odpowiada lambda-wyrażeniu x -> Klasa.metStat(x)
- NazwaKlasy::nazwa_metody_niestatycznej
  - Klasa::met – odpowiada lambda-wyrażeniu x -> x.met(x)
- zmienna_niestatyczna::nazwa_metody_niestatycznej
  - Klasa v;
  - v::met – odpowiada lambda-wyrażeniu x -> v.met(x)
- NazwaKlasy::new
  - Klasa::new – odpowiada lambda-wyrażeniu x -> new Klasa(x)

Naturalnie, liczba i typy parametrów oraz typ wyniku metody (konstruktora), do których używamy referencji, muszą odpowiadać deskryptorowi funkcji jakiegoś interfejsu funkcyjnego (który zostanie dobrany do realizacji lambda-wyrażenia). Najlepiej zobaczyć to na przykładzie. Załóżmy, że w kontekście kompilacji widocze są interfejsy funkcyjne:

```
interface Transformer<T,S> {
  T transform(S v);
}

interface Operator<R,T,S> {
  R oper(T v1, S v2);
}
```

oraz następująca klasa `Animal`:

```
class Animal {
  String type;

  public Animal(String type) {
    this.type = type;
  }

  public String toString() {
    return "Animal{" + "type=" + type + '}';
  }
}
```

Zastosowanie referencji do metod i konstruktorów ilustruje kod 10.8.

```
public class MetRef {

  static <T, S> List<T> create(List<S> src, Transformer<T,S> t) {
    List<T> target = new ArrayList<>();
```

```
    for (S s : src) target.add(t.transform(s));
    return target;
  }

  static <R,T,S> List<R> create(List<T> src1, List<S> src2, Operator<R,T,S> o) {
    List<R> res = new ArrayList<>();
    for (int i = 0; i < src1.size(); i++) {
      res.add(o.oper(src1.get(i), src2.get(i)));
    }
    return res;
  }

  public static void main(String[] args) {

    List<String> s = Arrays.asList( "pies", "kot", "ryba" );
    List<String> sn = Arrays.asList( "111", "222" );

    // zamiast
    List out = create(s, e -> e.toUpperCase());
    System.out.println(out);
    // możemy napisać
    out = create(s, String::toUpperCase);
    System.out.println(out);

    String text = "pies i kot są w domu, a ryba pływa";
    // zamiast
    out = create(s, e -> text.indexOf(e));
    System.out.println(out);
    // możemy napisać
    out = create(s, text::indexOf );
    System.out.println(out);

    // zamiast
    out = create(sn, e -> Integer.parseInt(e) );
    System.out.println(out);
    // możemy napisać
    out = create(sn, Integer::parseInt);
    System.out.println(out);

    // zamiast
    out = create(s, e -> new Animal(e));
    System.out.println(out);
    // możemy napisać
    out = create(s, Animal::new);
    System.out.println(out);

    List<String> word = Arrays.asList("alabama", "kociokwik");
    List<Integer> pos = Arrays.asList(1, 5);
    // zamiast
    out = create(word, pos, (e1, e2) -> e1.substring(e2));
    System.out.println(out);
    // możemy napisać
    out = create(word, pos, String::substring);
    System.out.println(out);
  }

}
```

Kod 10.8. Zastosowanie referencji do metod i konstruktorów w lambda-wyrażeniach

Ostatni przykład w programie 10.8 pokazuje jak łatwo, przy dostępności odpowiedniego interfejsu, można zastąpić lambda-wyrażenie z wieloma parametrami prostą referencją do metody. Oto wyniki działania programu 10.8:

```
[PIES, KOT, RYBA]
[PIES, KOT, RYBA]
[0, 7, 24]
[0, 7, 24]
[111, 222, 3333]
[111, 222, 3333]
[Animal{type=pies}, Animal{type=kot}, Animal{type=ryba}]
[Animal{type=pies}, Animal{type=kot}, Animal{type=ryba}]
[labama, kwik]
[labama, kwik]
```

Zauważmy jeszcze, że referencja do metody jest lambda-wyrażeniem typu odpowiedniego interfejsu funkcyjnego, dlatego są możliwe następujące przypisania:

```
Transformer<String, String> ts = String::toUpperCase;
Operator<String, String, Integer> op = String::substring;
```

10.4. Rodzaje i użycie gotowych interfejsów funkcyjnych

Jak już wspominaliśmy, najczęściej nie musimy tworzyć własnych interfejsów funkcyjnych. Po pierwsze, w Javie już od lat jest wiele interfejsów typu SAM (np. znany nam `File Filter` czy `Runnable`). Po drugie, w wersji 8 pojawił się pakiet `java.util.function`, w którym zdefiniowano wiele interfejsów funkcyjnych dla częstych przypadków użycia lambda-wyrażeń.

Zacznijmy od interfejsów, z których już korzystaliśmy w przykładowych programach.

A. Interfejs `ActionListener`. Stosowaliśmy go w programie symulującym zużycie paliwa w samochodzie (rozdz. 8.7). Ma on jedną metodę `void actionPerformed (ActionEvent)`, którą musieliśmy implementować w anonimowej klasie wewnętrznej, podając jej obiekt przy tworzeniu swingowego timera. Z użyciem lambda-wyrażenia możemy napisać to prościej (kod 10.9).

```
import javax.swing.Timer;

class Car extends Vehicle {
  // ...
  private volatile int fuel;
  // ...
  private Timer fuelTimer = new Timer(1000, e -> {
      fuel -= 1;
      if (fuel == 0) stop();
  });
```

```
  @Override
  public Car start() {
    // ...
    fuelTimer.start(); // start Timera
    // ...
    return this;
  }

  @Override
  public Car stop() {
    fuelTimer.stop();
    // ...
    return this;
  }
  // ...
}
```

Kod 10.9. Lambda i `ActionListener`

B. Interfejs `FileFilter`. Ma on jedną metodę boolean `accept(File f)`, wykorzystywaną do selekcji obiektów plikowych np. w trakcie listowania zawartości katalogu metodą `listFiles` (tu jako argument podajemy właśnie implementację metody `accept`). Zatem dla uzyskania tablicy plików z danego katalogu `dir`, mających podane rozszerzenie `ext` i czas modyfikacji późniejszy od podanego `time` możemy napisać (kod 10.10):

```
File[] files = new File(dir).listFiles(
       f -> f.isFile() && f.getName().endsWith(ext)
                       && f.lastModified() > time );
```

Kod 10.10. Lambda-wyrażenie i `FileFilter`

C. Interfejs `Comparator`. Jak pamiętamy, ma jedną metodę `int compare (T obiekt1, T obiekt 2)`. Przy porównywaniu i sortowaniu możemy więc używać wygodnych lambda-wyrażeń, np. do uporządkowania listy napisów wg ich długości (kod 10.11):

```
List<String> lst = Arrays.asList("ala", "ma", "kota", "i", "pieska");
Collections.sort(lst, (s1, s2) -> s2.length() - s1.length());
System.out.println(lst);
```

Kod 10.11. Lambda-wyrażenie i `Comparator`

Powyższy fragment wyprowadzi:

```
[pieska, kota, ala, ma, i]
```

D. Interfejs `Runnable`. Znany nam z wprowadzenia do programowania współbieżnego kod zliczający czas (np. w trakcie jakiejś interakcji użytkownika z programem) za pomocą lambda-wyrażenia, pasującego do deskryptora funkcji (`public void run()`) moglibyśmy napisać tak, jak pokazuje kod 10.12.

```
//...
// współbieżne uruchomienie zadania zliczania czasu
Future<?> ftask = Executors.newSingleThreadExecutor().submit(
    () -> {
      int sec = 0;
      while(true) {
        sec++;
        try { Thread.sleep(1000); } catch(InterruptedException exc) { return; }
        System.out.println(sec/60 + ":" + sec%60);
     }
    }
);
// ...
JOptionPane.showMessageDialog(null, „Close dialog to stop”);
ftask.cancel(true);
```

Kod 10.12. Lambda-wyrażenie i `Runnable`

E. Interfejs `Callable`. Implementację jego jedynej metody `V call() throws Exception` można podać jako lambda-wyrażenie. Kod 10.13 w odrębnym wątku odwraca napis, co sekundę wypisując kolejną wynikową literę, a główny wątek czeka na cały odwrócony napis i wyprowadza go na konsolę.

```
ExecutorService ex = Executors.newCachedThreadPool();
Future<String> task = ex.submit ( () -> {
    char[] src = s.toCharArray();
    char[] trg = new char[src.length];
    for(int i = src.length-1, j=0; i >= 0; i--, j++) {
      trg[j] = src[i];
      System.out.println( trg[j]);
      Thread.sleep(1000);
    }
    return new String(trg);
  }
);
String res = task.get();
System.out.println(res);
```

Kod 10.13. Lambda-wyrażenie i `Callable`

Uwaga. Ogólnie odwracanie napisów powinniśmy realizować za pomocą metody `reverse` klasy `StringBuffer` lub `StringBuilder`.

Pakiet `java.util.function` udostępnia wiele gotowych interfejsów funkcyjnych dla częstych przypadków użycia lambda-wyrażeń. Najważniejsze z nich przedstawia tab. 10.1.

Tablica 10.1. Najważniejsze ogólne interfejsy funkcyjne pakietu `java.util.function`

Interfejs	Metoda abstrakcyjna	Zastosowanie	Odpowiada lambdzie
`Predicate<T>`	`boolean test (T v)`	testowanie warunków	z argumentem typu `T` i wynikiem `boolean`, np. `s -> s.length()>10`
`Function<S, T>`	`T apply(S v)`	transformowanie wartości typu `S` w wartości typu `T` (funkcja z jednym argumentem)	z argumentem typu `S` i wynikiem typu `T`, np. `n -> n.toString()`
`UnaryOperator<T>`	`T apply(T v)`	operowanie na `v` ze zwrotem wyniku tego samego typu (funkcja zwracająca wynik tego samego typu co jej argument, czyli operator)	argument i wynik tego samego typu, np. `st -> st + 10`
`Supplier<T>`	`T get()`	dostarczanie obiektów typu `T`	bez argumentu i z wynikiem typu `T`, np.: `() -> LocalDate.now()` lub prościej: `LocalDate::now`
`Consumer<T>`	`void accept (T v)`	wykonywanie operacji na przekazanym obiekcie bez zwracania wyniku	z argumentem typu `T`, bez wyniku, np. `item -> item.setPrice(100)` `e -> System.out.println(e)` lub prościej: `System.out::println`
`BiPredicate<U,V>`	`boolean test (U u, V v)`	testowanie warunków	z dwoma argumentami i wynikiem boolean, np: `(str, n) -> str.length() > n`
`BiFunction<U,V,R>`	`R apply (U u, V v)`	funkcja z dwoma argumentami, ew. różnych typów	argumenty typów `U`, `V`, wynik typu `R`, np. `(xs,i) -> xs + i*2`
`BinaryOperator<T>`	`T apply (T v1, T v2)`	operator z dwoma argumentami i wynikiem tego samego typu	dwa argumenty i wynik tego samego typu, np. `(s1, s2) -> s1 + s2`
`BiConsumer<U,V>`	`void accept (U u, V v)`	konsumer z dwoma argumentami	argumenty typów `U`, `V`; bez wyniku, np. `(item, n) -> item.setPrice(n)` lub prościej: `Item::setPrice`

Poza tym – ze względów efektywnościowych – w pakiecie `java.util.function` są dostępne interfejsy funkcyjne, w których typy argumentów i/lub wyników są typami prostymi. Program 10.14 ilustruje proste przykładowe użycie interfejsów funkcyjnych pakietu.

```
import java.util.function.*;
import java.time.*;

public class FuncSamples {

  public static void show(Object o) {
    System.out.println(o);
  }

  public static void main(String[] args) {

    Predicate<String> p = s -> s.length() > 10;
    show ( p.test( "aaaa") );

    Function<Integer, String> f = n -> n.toString();
    show ( f.apply(101001). length());

    UnaryOperator<String> uo = st -> st + 10;
    show( uo.apply("Pies") );

    Supplier<LocalDate> su = LocalDate::now;
    show ( su.get() );

    // Klasa Item przedstawia pozycje zakupów
    // każde pozycja ma id, nazwę i cenę
    // konstruktor ustala nazwę i cenę
    // metoda void setPrice(int) - zmienia cenę
    // toString() zwraca opis pozycji zakupów

    Consumer<Item> c = item -> item.setPrice(3500);
    Consumer<Item> c1 = System.out::println;

    Item it = new Item("komputer", 1500);
    c1.accept(it);
    c.accept(it);
    c1.accept(it);

    BiPredicate<String, Integer> bp = (str, n) -> str.length() > n;
    show (bp.test("ala", 1));

    BiFunction<String, Integer, String> bi = (xs,i) -> xs + i*2;
    show ( bi.apply("ala", 101) );

    BinaryOperator<String> bo = (s1, s2) -> s1 + s2;
    show ( bo.apply("bear", "cat") );

    BiConsumer<Item, Integer> bic = (item, n) -> item.setPrice(n);
    bic.accept(it, 2000);
    show (it);
    BiConsumer<Item, Integer> bic2 = Item::setPrice;
    bic2.accept(it, 3000);
    show (it);
  }
}
```

Kod 10.14. Proste przykłady użycia interfejsów funkcyjnych pakietu `java.util.function`

Program 10.14 wypisze:

```
false
6
Pies10
2013-09-21
id=1, name=komputer, price=1500
id=1, name=komputer, price=3500
true
ala202
bearcat
id=1, name=komputer, price=2000
id=1, name=komputer, price=3000
```

Przedstawione proste przykłady nie są zbyt interesujące (ale dobrze pokazują, jak używać metod interfejsów). Całą moc gotowych interfejsów funkcyjnych oraz lambda-wyrażeń możemy docenić przy tworzeniu uniwersalnych metod przetwarzania, które jako argumenty będą m.in. otrzymywać lambda-wyrażenia, odpowiadające określonym interfejsom. We wprowadzeniu do tego rozdziału, a także w poprzednim punkcie, takie metody były już pokazane, wtedy jednak dostarczaliśmy definicji własnych interfejsów. Teraz wykorzystamy gotowe interfejsy funkcyjne.

Stworzymy następujące uniwersalne metody:

- `List<T> findAll(Collection<T> src, Predicate<T> p)` wyszukuje w kolekcji `src` wszystkie elementy spełniające warunek określony przez predykat `p` i zwraca listę tych elementów;
- `List<T> transform(Collection<S> src, Function<S, T> f)` przekształca elementy kolekcji `src` za pomocą funkcji `f` i dodaje do wynikowej listy;
- `List<T> generate(int n, Supplier<T> s)` zwraca listę `n`-elementów, które tworzone są za pomocą podanego jako drugi argument „generatora”;
- `void process(Collection<T> src, Consumer<T> c)` używa konsumenta `c` do zmiany wartości elementów kolekcji `src`.

Ich implementacje przedstawia kod 10.15.

```
import java.util.*;
import java.util.function.*;
// ...
  static <T> List<T> findAll(Collection<T> src, Predicate<T> p) {
    List<T> trg = new ArrayList<>();
    for (T e : src) if (p.test(e)) trg.add(e);
    return trg;
  }

  static <S, T> List<T> transform(Collection<S> src, Function<S, T> f) {
    List<T> trg = new ArrayList<>();
    for (S e : src) trg.add(f.apply(e));
    return trg;
  }
```

```
static <T> List<T> generate(int n, Supplier<T> s) {
  List<T> trg = new ArrayList<>();
  for (int i = 0; i < n; i++) {
    trg.add(s.get());
  }
  return trg;
}

static <T> void process(Collection<T> src, Consumer<T> c) {
  for (T e : src) c.accept(e);
}
```

Kod 10.15. Przykładowe uniwersalne metody, korzystające z gotowych interfejsów funkcyjnych

Działanie tych metod zilustrujemy na przykładzie klasy `Item`, reprezentującej pozycje zakupów. Każdy zakup charakteryzuje się identyfikatorem (liczba całkowita, której wartości są automatycznie zwiększane w miarę tworzenia kolejnych obiektów klasy), nazwą towaru oraz ceną. Konstruktor klasy inicjuje jej obiekty na podstawie pomocniczej klasy `ItemGenerator`, która może generować losowe pozycje zakupów (z nazwą towaru losowaną z przekazanej tablicy oraz ceną z podanego zakresu), albo też wykorzystywać inne metody dostarczania danych (np. z pliku). Konstruktor klasy `Item` korzysta ze statycznej instancji generatora i używa go do ustalenia wartości pól klasy.

```
public Item() {
  ItemGenerator itg = ItemGenerator.INSTANCE; // ale to nie singleton!
  // pobranie danych od generatora
  // i ustalenie nazwy towaru i jego ceny
}
```

Teraz metody `generate`, `findAll`, `transform` i `process` (kod 10.15) zastosujemy do przetwarzania losowo wygenerowanych pozycji zakupów, co ilustruje kod 10.16, a jeden z wyników – listing 10.2.

```
ItemGenerator itg =
        new ItemGenerator(new String[] {"chleb", "masło", "mleko"}, // nazwy
        new int[] {1, 10}); // zakres cen

List<Item> t = generate(10, Item::new);
for (Item e : t) { System.out.println(e); }
System.out.println("---------------------------");
List<Item> out = findAll(t, e -> e.getName().equals("chleb"));
for (Item e : out) { System.out.println(e); }
System.out.println("---------------------------");
process(out, e -> e.setPrice(e.getPrice()+1));
for (Item e : out) { System.out.println(e); }
System.out.println("---------------------------");
List<Integer> list = transform(t, Item::getPrice);
System.out.println(list);
```

Kod 10.16. Użycie uniwersalnych metod z lambda-wyrażeniami na gotowych interfejsach funkcyjnych

```
id=1, name=mleko, price=1
id=2, name=chleb, price=4
id=3, name=mleko, price=6
id=4, name=mleko, price=2
id=5, name=mleko, price=1
id=6, name=chleb, price=3
id=7, name=mleko, price=5
id=8, name=masło, price=1
id=9, name=masło, price=2
id=10, name=mleko, price=2
--------------------------
id=2, name=chleb, price=4
id=6, name=chleb, price=3
--------------------------
id=2, name=chleb, price=5
id=6, name=chleb, price=4
--------------------------
[1, 5, 6, 2, 1, 4, 5, 1, 2, 2]
```

Listing 10.2. Wynik działania programu 10.16

Interfejsy funkcyjne z pakietu `java.util.function` mają też wiele domyślnych metod, które pozwalają na łączenie i przekształcanie predykatów i funkcji. Interfejsy `Predicate` i `BiPredicate` dostarczają m.in. implementacji metod:

- **`and`**`(predykat2)` – zwraca predykat, który logicznie oznacza koniunkcję tego (`this`) predykatu i predykatu podanego jako argument (`predykat2`);
- ***`or`*** `(predykat2)` – zwraca predykat, który logicznie oznacza alternatywę tego (`this`) predykatu i predykatu podanego jako argument (`predykat2`);
- **`negate`**`()` – zwraca predykat, który oznacza logiczne zaprzeczenie tego (`this`) predykatu.

Dzięki temu możliwe staje się ponowne wykorzystanie (łączenie i przekształcanie) zdefiniowanych wcześniej predykatów (kod 10.17).

```
Predicate<Item> chleb = e -> e.getName().equals("chleb");
Predicate<Item> mleko = e -> e.getName().equals("mleko");
Predicate<Item> drogi = e -> e.getPrice() > 3;

// list jest wygenerowaną listą pozycji zakupów (tym razem z pliku)

List<Item> out = findAll(list, chleb.or(mleko) );
System.out.println("Chleb albo mleko");
for (Item e : out) { System.out.println(e); }
out = findAll(list, chleb.negate());
System.out.println("Nie chleb");
for (Item e : out) { System.out.println(e); }
out = findAll(list, chleb.and(drogi));
System.out.println("Drogi chleb");
for (Item e : out) { System.out.println(e); }
```

Kod 10.17. Łączenie i przekształcanie predykatów

Ten fragment może wyprowadzić na konsolę następującą informację:

```
Chleb albo mleko
id=1, name=chleb, price=2
id=2, name=mleko, price=2
id=4, name=chleb, price=5
id=6, name=chleb, price=7
id=7, name=mleko, price=3
Nie chleb
id=2, name=mleko, price=2
id=3, name=masło, price=3
id=5, name=masło, price=2
id=7, name=mleko, price=3
Drogi chleb
id=4, name=chleb, price=5
id=6, name=chleb, price=7
```

Interfejsy `Function` i `BiFunction` (a także pochodne od nich interfejsy definiujące operatory) dostarczają m.in metod, pozwalających na tworzenie funkcji złożonych (kompozycji funkcji).

Funkcja złożona dwóch funkcji `f: X->Y i g: Y->Z`, to funkcja `h: X->Z` taka, że `h(x) = g(f(x))`, co oznacza, że najpierw zostanie zastosowana funkcja `f`, a następnie funkcja `g` z argumentem, będącym wynikiem funkcji `f`.

Interfejs `Function` definiuje m.in. dwie metody domyślne, zwracające funkcje złożone:

- **`compose`**`(funk)` – zwraca funkcję, która przy wywołaniu najpierw zastosuje funkcję `funk`, a następnie tę funkcję, na rzecz której `compose` jest wołane (`this`), z argumentem będącym wynikiem wywołania `funk`;
- **`andThen`**`(funk)` – składa funkcję w „odwrotnym kierunku": najpierw `this`, później `funk`.

Dzięki temu możemy niezależnie od zastosowania zdefiniować pewne operacje i używać później dowolnych ich kombinacji. Ilustruje to przykładowy kod 10.18.

```
UnaryOperator<String> italic = e -> "<italic>"+ e + "</italic>";
UnaryOperator<String> bold = e -> "<bold>"+ e + "</bold>";
Function<String, String> kompoz1 = italic.compose(bold);
Function<String, String> kompoz2 = italic.andThen(bold);

String s = "ala am kota";
System.out.println(bold.apply(s));
System.out.println(italic.apply(s));
System.out.println(kompoz1.apply(s));
System.out.println(kompoz2.apply(s));
```

Kod 10.18. Składanie funkcji

Kod 10.18 wypisze na konsoli:

```
<bold>ala am kota</bold>
<italic>ala am kota</italic>
<italic><bold>ala am kota</bold></italic>
<bold><italic>ala am kota</italic></bold>
```

Również interfejsy `Consumer` i `BiConsumer` umożliwiają łańcuchowanie operacji za pomocą metody `andThen(Consumer after)`, która zwraca konsumenta, stosującego (***na tym samym argumencie***!) najpierw tę (`this`) operację, a następnie operację *`after`*.

Dla przykładu możemy zdefiniować dwie operacja przetwarzania zakupu: dodanie do ceny VAT-u oraz zmniejszenie ceny w wyniku upustu. Załóżmy, że do cen towarów może być dodawana złotówka VAT-u i może też być zastosowany upust o 2 zł. Wygodnie będzie obie operacje zdefiniować oddzielnie, bo mogą być stosowane w różny sposób dla różnych towarów. Możemy w następujący sposób użyć zdefiniowanej wcześniej metody `process`, przetwarzającej kolekcję zakupów za pomocą podanego konsumenta (kod 10.19).

```
Predicate<Item> chleb = e -> e.getName().equals("chleb");
Predicate<Item> drogi = e -> e.getPrice() > 3;

Consumer<Item> vat = e -> e.setPrice(e.getPrice()+1 );
Consumer<Item> upust = e -> e.setPrice(e.getPrice()-2 );

List<Item> out = findAll(list, chleb.and(drogi));
for (Item e : out) { System.out.println(e); }

process(out, vat.andThen(upust) );

for (Item e : out) { System.out.println(e); }
```

Kod 10.19. Łańcuchowanie operacji `get(...)` interfejsu `Consumer`

Fragment ten wypisze na konsoli:

```
id=4, name=chleb, price=5
id=6, name=chleb, price=7
id=4, name=chleb, price=4
id=6, name=chleb, price=6
```

10.5. Lambda-wyrażenia w gotowych metodach standardowych interfejsów i klas Javy

W Javie 8 pojawiły się nowe metody w standardowych klasach i interfejsach (metody statyczne i/lub domyślne), których argumentami mogą być lambda-wyrażenia. Jest ich całkiem sporo. Omówimy krótko wybrane z nich.

W interfejsie `Iterable` dodano domyślną metodę `forEach(Consumer op)`, dzięki której na dowolnych elementach zestawu danych, określanego przez implementację `Iterable` (m.in. na dowolnej kolekcji), możemy wykonywać podaną operację `op`. We wprowadzeniu do tego rozdziału użyliśmy jej do przetwarzania listy pracowników.

Aby przyjrzeć się bliżej działaniu `forEach`, zdefiniujmy metodę zwiększającą wartości elementów przekazanej listy liczb całkowitych. Argumentem lambda-wyrażenia w `forEach` będzie kolejny element listy. Nie mamy dostępu do indeksu tego elementu, zatem nie mogąc przeprowadzić modyfikacji za pomocą metody `set(i, val)` interfejsu `List`, musimy stworzyć i zwrócić nową listę, która będzie zawierać zmienione wartości (kod 10.20).