Zaawansowane programowanie w Javie

Studia zaoczne – Wykład 3

dr hab. Andrzej Zbrzezny, profesor UJD

Katedra Matematyki i Informatyki
Uniwersytet Jana Długosza w Częstochowie

6 kwietnia 2024

Zalecana Literatura

Literatura podstawowa

- Cay Horstmann.
  Java. Przewodnik doświadczonego programisty. Wydanie III.
  Wydawnictwo Helion. Gliwice, październik 2023.
- Joshua Bloch.
  Java. Efektywne programowanie. Wydanie III.
  Wydawnictwo Helion. Gliwice, sierpień 2018.
- Cay Horstmann.
  Java. Podstawy. Wydanie XII.
  Wydawnictwo Helion. Gliwice, grudzień 2022.
- https://dev.java/

Przetwarzanie wyrażeń lambda

Wybór interfejsu funkcyjnego

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

Przetwarzanie wyrażeń lambda

Wybór interfejsu funkcyjnego

- Oczywiście będzie wiele sytuacji, w których zechcemy przyjąć „dowolną funkcję” bez specjalnej semantyki.
- Istnieje szereg prostych typów funkcyjnych, które można do tego wykorzystać, i bardzo dobrym pomysłem jest korzystanie z nich, gdy tylko jest to możliwe.
- Tabela na kolejnych dwóch slajdach pokazuje wybrane interfejsy funkcyjne.

Popularne interfejsy funkcyjne

Functional Interface	Parameter types	Return type	Abstract method name	Description	Other methods
`Runnable`	none	`void`	`run`	Runs an action without arguments or return value	
`Supplier<T>`	none	`T`	`get`	Supplies a value of type `T`	
`Consumer<T>`	`T`	`void`	`accept`	Consumes a value of type `T`	`andThen`
`BiConsumer<T, U>`	`T, U`	`void`	`accept`	Consumes values of types `T` and `U`	`andThen`
`Function<T, R>`	`T`	`R`	`apply`	A function with argument of type `T`	`compose`, `andThen`, `identity`
`BiFunction<T, U, R>`	`T, U`	`R`	`apply`	A function with arguments of types `T` and `U`	`andThen`

Popularne interfejsy funkcyjne

`UnaryOperator<T>`	`T`	`T`	`apply`	A unary operator on the type `T`	`compose`, `andThen`, `identity`
`BinaryOperator<T>`	`T, T`	`T`	`apply`	A binary operator on the type `T`	`andThen`, `maxBy`, `minBy`
`Predicate<T>`	`T`	`boolean`	`test`	A boolean-valued function	`and`, `or`, `negate`, `isEqual`
`BiPredicate<T, U>`	`T, U`	`boolean`	`test`	A boolean-valued function with two arguments	`and`, `or`, `negate`

Przetwarzanie wyrażeń lambda

Wybór interfejsu funkcyjnego

- Załóżmy, że chcemy napisać metodę przetwarzającą pliki spełniające zadane kryteria.
- Powstaje pytanie czy powinniśmy użyć klasy implementującej interfejs `java.io.FileFilter` czy interfejsu `Predicate<File>`?
- Polecane jest korzystanie ze standardowego interfejsu `Predicate<File>`.
- Jedynym powodem, by tego nie robić, może być sytuacja, gdy mamy już wiele użytecznych metod tworzących instancje `FileFilter`.
- Większość standardowych interfejsów funkcyjnych ma metody nieabstrakcyjne do tworzenia lub łączenia funkcji.

Przetwarzanie wyrażeń lambda

Wybór interfejsu funkcyjnego

- Przykładowo, `Predicate.isEqual(a)` jest odpowiednikiem `a::equals`, ale działa również w sytuacji, gdy `a` ma wartość **`null`**.
- Istnieją domyślne metody: `and`, `or` oraz `negate` do łączenia predykatów. Przykładowo, `Predicate.isEqual(a).or(Predicate.isEqual(b))` jest równoważne z `x -> a.equals(x) || b.equals(x)`.
- Tabela na następnym slajdzie pokazuje 34 dostępne specjalizacje dla typów prostych: **`int`**, **`long`** oraz **`double`**.
- Warto korzystać z tych specjalizacji, aby unikać automatycznych przekształceń.

Interfejsy funkcyjne dla typów prostych

W poniższej tabeli `p`, `q` to **`int`**, **`long double`**, natomiast `P`, `Q` to `Int`, `Long`, `Double`.

Interfejs funkcjonalny	**Typy argumentów**	**Typ zwracany**	**Nazwa metody abstrakcyjnej**
BooleanSupplier	brak	**boolean**	getAsBoolean
PSupplier	brak	p	getAsP
PConsumer	p	**void**	accept
ObjPConsumer<T>	T, p	**void**	accept
PFunction<T>	p	T	apply
PToQFunction	p	q	applyAsQ
ToPFunction<T>	T	p	applyAsP
ToPBiFunction<T, U>	T, U	p	applyAsP
PUnaryOperator	p	p	applyAsP
PBinaryOperator	p, p	p	applyAsP
PPredicate	p	**boolean**	test

Przetwarzanie wyrażeń lambda

Implementowanie własnych interfejsów funkcyjnych

- Niezbyt często znajdziemy się w sytuacji, gdy żaden ze standardowych interfejsów funkcjonalnych nie będzie odpowiedni. Wtedy będzie trzeba stworzyć własny.
- Załóżmy, że chcemy wypełnić obraz kolorowymi wzorami, dla których użytkownik określa funkcję zwracającą kolor każdego piksela.
- Nie ma standardowego typu mapującego

```
(int, int) -> Color
```

- Moglibyśmy użyć

```
BiFunction<Integer, Integer, Color>
```

ale to grozi automatycznym opakowywaniem (ang. autoboxing).

Przetwarzanie wyrażeń lambda

Implementowanie własnych interfejsów funkcyjnych

- W takim przypadku warto zdefiniować nowy interfejs

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

- Należy oznaczać interfejsy funkcyjne 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 funkcyjny.

Przetwarzanie wyrażeń lambda

Implementowanie własnych interfejsów funkcyjnych

- Można już teraz zaimplementować metodę `createImage`:

```
BufferedImage createImage(
    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;
}
```

Przetwarzanie wyrażeń lambda

Implementowanie własnych interfejsów funkcyjnych

- Aby ją wywołać, należy utworzyć wyrażenie lambda zwracające wartość koloru dla dwóch liczb całkowitych:

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

Przykład

- Program `r03/r03_06/ImageDemo.java`

Wyrażenia lambda i zasięg zmiennych

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 argumentu lub zmiennej lokalnej w wyrażeniu lambda o takiej samej nazwie jak zmienna lokalna.

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

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

Wyrażenia lambda i zasięg zmiennych

Zasięg zmiennej lambda

- Inną konsekwencją zasady „tego samego zasięgu” jest to, że słowo **this** w wyrażeniu lambda oznacza argument **this** metody, która tworzy wyrażenie lambda. Dla przykładu rozważmy kod:

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

Wyrażenia lambda i zasięg zmiennych

Zasięg zmiennej lambda

- 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.

Wyrażenia lambda i zasięg zmiennych

Dostęp do zmiennych zewnętrznych

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

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

Wyrażenia lambda i zasięg zmiennych

Dostęp do zmiennych zewnętrznych

- Zauważmy, że wyrażenie lambda uzyskuje dostęp do zmiennych przekazanych jako argumenty w szerszym kontekście, a nie w samym wyrażeniu lambda. Rozważmy wywołanie:

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

- Popatrzmy teraz na zmienne `liczba` oraz `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 `repeatMessage` się zakończy i zmienne argumentów przestaną być dostępne.
- Powstaje pytanie w jaki sposób zmienne `tekst` oraz `liczba` pozostają dostępne przy wykonywaniu wyrażenia lambda.

Wyrażenia lambda i zasięg zmiennych

Dostęp do zmiennych zewnętrznych

- Aby zrozumieć, co się tutaj dzieje, musimy zmienić swoje podejście do wyrażeń lambda.
- Wyrażenie lambda ma trzy składniki:
  1. blok kodu,
  2. argumenty,
  3. wartości wolnych zmiennych – czyli takich zmiennych, które nie są argumentami i nie są zdefiniowane w kodzie.
- W naszym przykładzie wyrażenie lambda ma dwie wolne zmienne, `tekst` oraz `liczba`.
- Struktura danych reprezentująca wyrażenie lambda musi przechowywać wartości tych zmiennych – w naszym przypadku „Witaj” oraz 1000.
- Mówimy, że te wartości zostały przejęte przez wyrażenie lambda.

Wyrażenia lambda i zasięg zmiennych

Dostęp do zmiennych zewnętrznych

- 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.

Wyrażenia lambda i zasięg zmiennych

Dostęp do zmiennych zewnętrznych

- Jak widzieliśmy, 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 mamy dostęp tylko do zmiennych zewnętrznych, których wartość się nie zmienia.
- Czasem tłumaczy się to, mówiąc, że wyrażenie lambda przechwytuje wartości, a nie zmienne.
- Przykładowo, poniższy kod wygeneruje błąd przy kompilacji:

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

Wyrażenia lambda i zasięg zmiennych

Dostęp do zmiennych zewnętrznych

- Wyrażenie lambda próbuje wykorzystać zmienną `j`, 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.
- Dawniej ograniczenie było silniejsze – pobrane zmienne musiały być zadeklarowane z modyfikatorem `final`. Zostało to zmienione.

Wyrażenia lambda i zasięg zmiennych

Dostęp do zmiennych zewnętrznych

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

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

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

Wyrażenia lambda i zasięg zmiennych

Dostęp do zmiennych zewnętrznych

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

```
public static void repeatMessage(
    String tekst, int liczba, int wątki)
{
    Runnable r = () -> {
        while (liczba > 0) {
            // Błąd: nie można modyfikować
            // przechwyconej wartości
            liczba--; System.out.println(tekst);
        }
    };
    for (int j = 0; j < wątki; ++j)
        new Thread(r).start();
}
```

Wyrażenia lambda i zasięg zmiennych

Dostęp do zmiennych zewnętrznych

- Jest to w rzeczywistości pozytywna cecha. W sytuacji gdy dwa wątki aktualizują jednocześnie wartość zmiennej `liczba`, jej wartość pozostaje nieokreślona.
- Nie należy liczyć na to, 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.

Przykład

- Program `r03/r03_r07/ScopeDemo.java`.

Wyrażenia lambda i zasięg zmiennych

Dostęp do zmiennych zewnętrznych

- Można obejść ograniczenie uniemożliwiające wykonywanie operacji modyfikujących wartość, używając 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 można 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.

Funkcje wyższych rzędów

Wprowadzenie

- W funkcyjnych językach programowania funkcje są bardzo ważne.
- Tak samo jak przekazuje się liczby do metod i tworzy się metody generujące liczby, można mieć też argumenty 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 funkcyjne, ale zasada jest taka sama.
- W kolejnych punktach popatrzymy na przykłady i przeanalizujemy funkcje wyższego rzędu w interfejsie `Comparator`.

Funkcje wyższych rzędów

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.

Funkcje wyższych rzędów

Metody zwracające funkcje

- 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 funkcyjny).
- Przydaje się to do generowania funkcji, które są przekazywane do metod z interfejsami funkcyjnymi.

Funkcje wyższych rzędów

Metody modyfikujące funkcje

- Na poprzednich slajdach widzieliśmy 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(y, x);
}
```

- Ta metoda działa na funkcjach. Pobiera funkcję i zwraca ją zmodyfikowaną.

Funkcje wyższych rzędów

Metody modyfikujące funkcje

Aby uzyskać niezależny od wielkości znaków porządek malejący, zastosujemy wywołanie:

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

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

Przykład

- Program `r03/r03_r08/HigherOrderDemo.java`.

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 ta jest wywoływana dla obiektów, które mają być porównane, i porównywane są zwrócone klucze.
- Przykładowo załóżmy, że mamy tablicę obiektów klasy `Person`. Możemy je posortować według nazwy w następujący sposób:

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

Funkcje wyższych rzędów

Metody interfejsu `Comparator`

- 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));
```

- Istnieje kilka odmian tych metod. Można określić komparator, który ma być wykorzystywany dla kluczy tworzonych przez metody `comparing` oraz `thenComparing`.
- Przykładowo, można posortować ludzi według długości ich nazwisk:

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

Funkcje wyższych rzędów

Metody interfejsu `Comparator`

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

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

- Jeśli utworzona przez nas funkcja klucza może zwrócić **`null`**, przydatne będą adaptery `nullsFirst` oraz `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.

Funkcje wyższych rzędów

Metody interfejsu `Comparator`

- Na przykład załóżmy, że funkcja `getMiddleName` zwraca **`null`**, jeśli osoba nie ma drugiego imienia.
- W takiej sytuacji można 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`.

Funkcje wyższych rzędów

Metody interfejsu `Comparator`

- Poniżej znajduje się pełne polecenie sortujące z wykorzystaniem drugiego imienia, które potencjalnie może mieć wartość **`null`**.
- Dla zachowania czytelności korzystamy ze statycznie importowanego `java.util.Comparator.*`.
- Zauważmy, że typ zmiennej `naturalOrder` jest ustalany z kontekstu.

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

- Statyczna metoda `reverseOrder` odwraca kolejność.

Przykład

- Program `r03/r03_r08/ComparatorDemo.java`.

Lokalne klasy wewnętrzne

Wprowadzenie

- Na długo przed pojawieniem się wyrażeń lambda język Java miał mechanizmy do łatwego definiowania klas implementujących interfejs (lub interfejs funkcyjny).
- W przypadku interfejsów funkcyjnych powinniśmy koniecznie używać wyrażeń lambda, ale czasem możemy potrzebować zgrabnej składni dla interfejsu, który nie jest interfejsem funkcyjnym.
- Klasyczne rozwiązania można też napotkać, przeglądając istniejący kod.

Lokalne klasy wewnętrzne

Klasy lokalne

- Można definiować klasę wewnątrz metody. Taka klasa jest nazywana klasą lokalną.
- Powinniśmy 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.
- Przykładowo, rozważmy metodę:

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

która generuje nieskończony ciąg losowych liczb całkowitych w zadanym zakresie.

Lokalne klasy wewnętrzne

Klasy lokalne

- 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();
}
```

Lokalne klasy wewnętrzne

Klasy lokalne

- 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 zmienimy `RandomInt` w klasę zagnieżdżoną, będziemy musieli utworzyć jawny konstruktor pobierający te wartości i zapisujący je w zmiennych instancji

Przykład

- Program `r03/r03_r09/LocalClassDemo.java`.

Lokalne klasy wewnętrzne

Klasy anonimowe

- W przykładzie z poprzedniego podrozdziału nazwa `RandomSequence` była wykorzystana dokładnie raz: do utworzenia zwracanej wartości.
- W takim przypadku można 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; }
    }
}
```

Lokalne klasy wewnętrzne

Klasy anonimowe

- 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 argumenty 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 funkcyjnych z interfejsami `Runnable` czy komparatorów.
- Przeglądając stary kod, będziemy je często spotykać.

Przykład

- Program `r03/r03_r09/AnonymousClassDemo.java`.

Lokalne klasy wewnętrzne

Klasy anonimowe

- Obecnie klasy wewnętrzne są potrzebne jedynie w sytuacji, gdy musimy utworzyć dwie lub więcej metod, jak w powyższym przykładzie.
- Jeżeli interfejs `IntSequence` ma domyślną metodę `hasNext`, to można po prostu wykorzystać wyrażenie lambda:

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