WYŻSZA SZKOŁA BANKOWA
Warszawa

Programowanie aplikacji w Java

Maciej Gowin

Zjazd 3 - dzień 2

Linki

Opis

https://maciejgowin.github.io/wsb-java/

Kod źródłowy przykładów oraz zadań

https://github.com/MaciejGowin/wsb-programowanie-aplikacji-java

Język Java: adnotacje

Adnotacje są metadanymi opisującymi kod źródłowy. Mogą one by używane w połączeniu z klasami, metodami, zmiennymi, parametrami, a nawet pakietami.

Adnotacje mogą być usunięte lub zachowane podczas kompilacji. Daje to możliwość ich późniejszego odczytania podczas działania programu.

Java definiuje wbudowane adnotacje. Do najpopularniejszych należą:

`@Override` - sprawdza, czy metoda jest przeciążona.
`@Deprecated` - oznacza metodę jako przestarzałą, zgłaszając ostrzeżenie podczas kompilacji.
`@SuppressWarnings` - informuje kompilator, aby pominął ostrzeżenia danego typu.

Istnieje możliwość tworzenia własnych adnotacji.

Język Java: adnotacje

Deklaracja adnotacji jest podobna do deklaracji interfejsu. Wyróżnia ją jednak znak `@` poprzedzający słowo kluczowe `interface`.

```
public @interface Author {
    String name();
    String company() default "WSB";
}
```

Dodatkowo możemy zdefiniować metody, które opisują elementy adnotacji. Metody te:

- Nie mogą przyjmować parametrów ani zgłaszać wyjątków.
- Wartości zwracane ograniczają się do typów prymitywnych, ciągów znaków `String`, klasy `Class`, typów wyliczeniowych oraz tablic wcześniej wymienionych typów.
- Mogą mieć wartości domyślne.

Język Java: adnotacje

Definicję adnotacji możemy wzbogacić o adnotacje, które definiują kiedy, i dla jakich elementów mogą one zostać użyte.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE})
public @interface Author {
    String name();
    String company() default "WSB";
}
```

`@Retention` - wskazuje, do jakiego momentu dana adnotacja powinna być zachowana.
`@Target` - wskazuje, dla jakiego elementu może zostać użyta.

Adnotacje same w sobie nie wykonują żadnych zadań. Są one przetwarzane przez zewnętrzne narzędzia lub kod programu i na ich podstawie tworzone jest dodatkowe zachowanie.

Język Java: adnotacje

Możliwe wartości dla `RetentionPolicy`:

- `SOURCE` - adnotacja powinna zostać usunięta przez kompilator.
- `CLASS` - adnotacja powinna zostać zapisana w pliku wynikowym class, ale nie powinna być przetworzona podczas uruchomienia przez JVM (domyślne zachowanie).
- `RUNTIME` - adnotacja powinna być dostępna i zachowana przez kompilator oraz JVM.

> Adnotacje typu RUNTIME są używane przez mechanizm refleksji. Korzysta z nich wiele bibliotek do automatycznego dodawania określonych zachowań do klas zdefiniowanych przez użytkownika.

Język Java: adnotacje

Wybrane wartości dla `Target`:

- `TYPE` - adnotacja może zostać użyta na poziomie: klasy, interfejsu, adnotacji lub typu wyliczeniowego
- `FIELD` - adnotacja może zostać użyta na własnościach klasy
- `METHOD` - adnotacja może zostać użyta na poziomie metody
- `PARAMETER` - adnotacja może zostać użyta na poziomie parametru.
- `CONSTRUCTOR` - adnotacja może zostać użyta na poziomie konstruktora.
- `LOCAL_VARIABLE` - adnotacja może zostać użyta na zmiennej lokalnej.
- `ANNOTATION_TYPE` - adnotacja może zostać użyta przy deklaracji adnotacji
- `PACKAGE` - adnotacja może zostać użyta przy deklaracji pakietu

Język Java: refleksja

Refleksja to mechanizm pozwalający na przeglądanie oraz modyfikację klas, interfejsów, metod, konstruktorów i pól klas w czasie działania programu. Wszystkie te operacje mogą zostać wykonane z poziomu kodu źródłowego.

Głównym elementem całego mechanizmu refleksji w Java jest klasa `Class`. Przechowuje ona informacje o klasie i jej składowych. Każda z instancji klasy `Class` opisuje jedną klasę i to dzięki nim mamy dostęp do szczegółowych informacji oraz mechanizmu refleksji.

Adnotacje w połączeniu z mechanizmem refleksji pozwalają na sterowanie jej zachowaniem. To nic innego jak mechanizm tworzenia kodu na podstawie kodu już istniejącego. Adnotacje możemy rozumieć jako dodatkowe wskazówki dla mechanizmu refleksji precyzujące jej działanie.

Język Java: klasa Class

Do stworzenia obiektu typu `Class` możemy użyć jednego z trzech sposobów.

Poprzez metodę Class.forName()

```
class Foo {
}

Class<?> fooClazz = Class.forName("Foo");
```

Język Java: klasa Class

Poprzez metodę Object.getClass()

```
class Foo {
}

Foo foo = new Foo();
Class<?> fooClazz = foo.getClass();
```

Poprzez rozszerzenie .class

```
class Foo {
}

Class<Foo> fooClazz = Foo.class;
```

Programowanie: przykład 35

Przykład użycia refleksji do odczytania informacji o klasie, stworzenia obiektów, ustawienia wartości oraz wywołania metod.

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

class SampleClass {

    private String privateField;
    public String publicField;

    public SampleClass() {
        System.out.println("Invoked SampleClass()");
    }

    private void privateMethod() {
        System.out.println("Invoked privateMethod()");
    }

    public void publicMethod() {
        System.out.println("Invoked publicMethod()");
    }

    @Override
    public String toString() {
        return String.format("{privateField: %s, privateField: %s}", privateField, publicField);
    }
}

public class Main {

    public static void main(String[] args) throws Exception {
        reflectionCreation();
        describeClass(SampleClass.class);
    }

    public static void reflectionCreation() throws Exception {
        // Create instance via reflection
        Constructor<SampleClass> constructor = SampleClass.class.getDeclaredConstructor();
        SampleClass sampleClass = constructor.newInstance();

        // Set fields via reflection
        System.out.printf("sampleClass: %s%n", sampleClass);
        Field publicField = SampleClass.class.getDeclaredField("publicField");
        publicField.set(sampleClass, "value-publicField");

        Field privateField = SampleClass.class.getDeclaredField("privateField");
        privateField.setAccessible(true);
        privateField.set(sampleClass, "value-privateField");

        System.out.printf("sampleClass: %s%n", sampleClass);

        // Invoke methods via reflection
        Method publicMethod = SampleClass.class.getDeclaredMethod("publicMethod");
        publicMethod.invoke(sampleClass);

        Method privateMethod = SampleClass.class.getDeclaredMethod("privateMethod");
        privateMethod.setAccessible(true);
        privateMethod.invoke(sampleClass);
    }

    public static void describeClass(Class<?> clazz) {
        // Get all fields
        System.out.println("Fields:");
        for (Field field: clazz.getDeclaredFields()) {
            System.out.printf(" - %s: %s%n", field.getName(), Modifier.toString(field.getModifiers()));
        }

        // Get all methods
        System.out.println("Methods:");
        for (Method method: clazz.getDeclaredMethods()) {
            System.out.printf(" - %s: %s%n", method.getName(), Modifier.toString(method.getModifiers()));
        }

        // Get all constructors
        System.out.println("Constructors:");
        for (Constructor<?> method: clazz.getDeclaredConstructors()) {
            System.out.printf(" - %s: %s%n", method.getName(), Modifier.toString(method.getModifiers()));
        }
    }
}
```

Programowanie: przykład 36

Połączenie mechanizmu adnotacji z refleksją na przykładzie walidacji obiektów.

```
import validator.NotEmpty;
import validator.Validator;

class Sample {
    @NotEmpty
    private String valueNotEmpty;
    private String valueAny;

    public Sample(String valueNotEmpty, String valueAny) {
        this.valueNotEmpty = valueNotEmpty;
        this.valueAny = valueAny;
    }

    @Override
    public String toString() {
        return String.format("{valueNotEmpty: %s, valueAny: %s}", valueNotEmpty, valueAny);
    }
}

public class Main {

    public static void main(String[] args) throws Exception {
        Sample[] samples = {
                new Sample("test", "test"),
                new Sample("test", null),
                new Sample(null, "test"),
                new Sample(null, null)
        };

        for (Sample sample: samples) {
            System.out.printf("%s: is valid: %s%n", sample.toString(), Validator.validate(sample));
        }
    }
}
```

Język Java: UNICODE

ASCII (ang. American Standard Code for Information Interchange) to 7-bitowy (2^7 = 128 możliwości) system kodowania znaków. Przyporządkowuje on liczbom z zakresu 0–127 litery alfabetu łacińskiego języka angielskiego, cyfry, znaki przestankowe i inne symbole. Jest on stosunkowo ograniczony ze względu na przestrzeń bitów używanych do jego reprezentacji.

Unicode jest standardem kodowania, którego założeniem była możliwość zapisania każdego znanego języka na świecie. Jest on kompatybilny z ASCII, gdyż reprezentacja elementów 0-127 jest zgodna.

Język Java: UTF-8/UTF-16

Implementacja standardu Unicode są systemy kodowania znaków UTF (ang. Unicode Transformation Formats):

- `UTF-8` wykorzystujący 8-bitowe słowa oraz od 1 do 4 bajtów do zakodowania pojedynczego znaku.
- `UTF-16` wykorzystujący 16-bitowe słowa oraz od 1 do 4 bajtów do zakodowania pojedynczego znaku.

Język Java: String

Operacje na ciągach znaków są jednymi z najczęściej używanych. Klasa `String` wprowadza szereg wbudowanych metod:

`length()` - długość ciągu znaków
`isEmpty()` - sprawdzenie, czy ciąg znaków jest pusty
`equals(), equalsIgnoreCase(), compareTo(), compareToIgnoreCase()` - porównanie dwóch ciągów znaków
`startsWith(), endsWith()` - sprawdzenie, czy ciąg znaków rozpoczyna się/kończy się zadanym ciągiem znaków
`contains()` - sprawdzenie ciąg znaków zawiera zadany ciąg znaków
`toLowerCase(), toUpperCase` - zamiana ciągu znaków na małe/wielkie litery

Język Java: String

`replace(), replaceAll(), replaceFirst()` - podmiana zadanego ciągu znaków innym ciągiem znaków.
`trim()` - usunięcie białych znaków z początku i końca ciągu znaków
`substring()` - wyłuskanie ciągu znaków dla zadanych indeksów
`split()` - podział ciągu znaków na tablicę ciągu znaków na podstawie zadanego ciągu znaków
`join()` - połączenie ciągów znaków zadanym ciągiem
`format()` - formatowanie ciągu znaków

> Do konkatenacji ciągu znaków `String` w pętli powinniśmy używać klasy `StringBuilder`. Wynika to z faktu, że podczas konkatenacji zawsze powstaje nowy obiekt będący kopią.

Język Java: wyrażenia regularne

Wyrażenie regularne (ang. regular expression, w skrócie regex lub regexp) – wzorzec opisujący łańcuch symboli.
Wyrażenia regularne mogą określać zbiór pasujących łańcuchów, jak również wyszczególniać istotne części łańcucha.

Przykłady:

`filename-[0-9]+.txt`

`images.+\.jpg`

`^M[a]+ci.*j$`

Programowanie: zadanie 22

Zdefiniuj metodę `String normalizeTitle(String value)` normalizującą tytuł:

- usuwającą spacje z początku i końca,
- ustawiającą pomiędzy każdym słowem jednynie jedną spację,
- zamieniającą każde słowo na format z pierwszą wielką literą.

Dla przykładu:

```
" W pustyni i w PUSZCZy" -> "W Pustyni I W Puszczy"
" o dwóch TAKICH co ukradli      Księżyc" -> "O Dwóch Takich Co Ukradli Księżyc
"pan   tadeusz" -> "Pan Tadeusz"
```

Język Java: interfejs funkcyjny

Istnieją sytuacje, w których tworzymy interfejs realizujący jedno konkretne zadanie. Załóżmy, że biblioteka dostarcza interfejs walidatora:

```
interface Validator {
    boolean isValid(String value);
}
```

Możemy dostarczać różne jego implementacje. Przyjmijmy, że na potrzeby weryfikacji zadanego ciągu znaków chcielibyśmy zweryfikować czy dany ciąg znaków nie jest pusty. Zadanie to moglibyśmy zrealizować przy pomocy klasy anonimowej.

Język Java: interfejs funkcyjny

```
interface Validator {
    boolean isValid(String value);
}

public class Main {

    public static void main(String[] args) {
        Validator notEmptyValidator = new Validator() {

            @Override
            public boolean isValid(String value) {
                return value != null && value.length() > 0;
            }
        };

        System.out.println(notEmptyValidator.isValid(null));
        System.out.println(notEmptyValidator.isValid(""));
        System.out.println(notEmptyValidator.isValid("test"));
    }
}
```

Język Java: interfejs funkcyjny

Interfejsy definiujące jedną metodę abstrakcyjną określane są mianem interfejsów SAM (ang. Single Abstract Method)

Wraz z Java 8 wprowadzona została adnotacja opisująca takie interfejsy: `@FunctionalInterface`.

Każdy interfejs oznaczony tą adnotacją zostanie zweryfikowany podczas kompilacji. Kompilator upewni się, czy interfejs oznaczony taką adnotacją ma tylko jedną metodę abstrakcyjną.

Adnotacja ta nie jest wymagana. Z założenia każdy interfejs o dokładnie jednej metodzie abstrakcyjnej jest interfejsem funkcjonalnym.

Język Java: wyrażenie lambda

W Javie 1.8 wprowadzony został uproszczony zapis pozwalający na definiowanie implementacji interfejsów funkcjonalnych.

Używamy do tego wyrażeń lambda.

```
Validator notEmptyValidator = value -> value != null && value.length() > 0;
```

Interfejs funkcyjny posiada dokładnie jedną metodę abstrakcyjną. Dzięki temu nie musimy explicite definiować metody, którą implementujemy.

> Wyrażania `lambda` umożliwiają definiowanie metod anonimowych.

Język Java: wyrażenie lambda

Format wyrażeń lambda możemy opisać za pomocą:

```
(parameters list) -> body
```

Gdzie:

- `parameters list` : może nie posiadać parametrów lub posiadać ich wiele.
- `body` : może być opisane jednym wyrażeniem lub ich grupą.

> Wyrażanie lambda nie jest wykonywane podczas definicji, ale podczas wywołania.

Język Java: typy wyrażeń lambda

Wyrażenie lambda bez parametrów

```
() -> System.out.println("Invokeed");
```

Wyrażenie lambda z parametrami

```
(a, b) -> System.out.printf("Invoked: %s: %s%n", a, b);
```

Wyrażenie lambda z wieloma wyrażeniami

```
(a, b) -> {
    int sum = a + b;
    System.out.printf("Sum: %s%n", sum);
}
```

Język Java: typy wyrażeń lambda

Wyrażenie lambda z wieloma wyrażeniami i wartością zwracaną

```
(a, b) -> {
    int sum = a + b;
    return sum;
}
```

Wyrażenie lambda z jednym wyrażeniami i wartością zwracaną

```
(a, b) -> a + b;
```

Język Java: wbudowane interfejsy funkcyjne

Java wprowadza szereg wbudowanych interfejsów funkcyjnych. Dostępne są one w pakiecie `java.util.function`.

Function<T, R>

```
public interface Function<T, R> {
    R apply(T t);
}
```

```
Function<String, String> quoted = value -> "\"" + value + "\"";
```

Język Java: wbudowane interfejsy funkcyjne

BiFunction<T, U, R>

```
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

```
BiFunction<Double, Double, Double> average = (a, b) -> (a + b)/2;
```

Supplier<T>

```
public interface Supplier<T> {
    T get();
}
```

```
Supplier<String> sample = () -> "sample";
```

Język Java: wbudowane interfejsy funkcyjne

Consumer<T>

```
public interface Consumer<T> {
    void accept(T t);
```

```
Consumer<String> print = value -> System.out.printf("Print: %s", value);
```

Predicate<T>

```
public interface Predicate<T> {
    boolean test(T t);
}
```

```
Predicate<String> empty = value -> value == null || value.length() == 0;
```

Język Java: wykorzystanie wyrażeń lambda

Wiele wbudowanych klas zawiera metody oraz funkcjonalności operujące na wyrażeniach lambda. Dla przykładu kolekcje Java używają ich do manipulacji na elementach.

```
List<String> users = new ArrayList<>();
users.add("USER1");
users.add("UseR2");
users.add("user3");
users.replaceAll(user -> user != null ? user.toLowerCase() : null);
```

Programowanie: przykład 37

Przykład użycia kolekcji z lambdami.

```
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

public class Main {

    public static void main(String[] args) {
        Map<Integer, String> userPerId = new HashMap<>();
        userPerId.put(1, "mark");
        userPerId.put(2, "elisabeth");
        userPerId.put(3, "george");

        System.out.println("Users: " + userPerId);

        // Function
        Function<Integer, String> defaultUser = key -> "unknown-" + key;
        String userId3 = userPerId.computeIfAbsent(3, defaultUser);
        System.out.println("userId3: " + userId3);

        String userId4 = userPerId.computeIfAbsent(4, defaultUser);
        System.out.println("userId4: " + userId4);

        System.out.println("Users: " + userPerId);

        // BiFunction
        userPerId.replaceAll((key, value) -> !value.endsWith("-" + key) ? value + "-" + key : value);

        System.out.println("Users: " + userPerId);

        // Consumer
        userPerId.values().forEach(userName ->  System.out.printf("User name: %s%n", userName));
    }
}
```

Język Java: strumienie

Strumienie (ang. streams) to narzędzia służące do procesowania sekwencji elementów, takich jak kolejki, listy czy mapy. Pozwalają na operacje takie jak: wyszukiwanie, filtrowanie czy mapowanie wartości.

```
ArrayList<String> names = new ArrayList<>();
names.add("Gary");
names.add("Jessica");
names.add("George");
names.add("Elisabeth");
names.add("George");

names.stream()
    .filter(name -> name.startsWith("G"))
    .map(name -> name.toLowerCase())
    .distinct()
    .forEach(name -> System.out.printf("User: %s%n", name));
```

Język Java: strumienie

Zestaw klas do operacji na strumieniach zdefiniowany jest w pakiecie `java.util.stream`. Główną klasą dostarczającą funkcjonalności przetwarzania strumieniowego jest `Stream<T>`.

Strumienie w dużym stopniu opierają się na wyrażeniach `lambda`, które pozwalają na manipulowanie elementami. Co istotnie, zdefiniowane operacje w postaci wyrażeń `lambda` nie są wykonywane zaraz po ich definicji, ale w momencie ich faktycznego użycia.

Strumienie używają podejścia `fluent API`. Polega ono na łączeniu obiektów za pomocą łańcucha wywołań metod. Celem takiego zapisu jest zwiększenie czytelności oraz używalności kodu.

Programowanie: przykład 38

Wywołania strumieni.

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class Main {

    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();
        names.add("Gary");
        names.add("Jessica");
        names.add("George");
        names.add("Elisabeth");
        names.add("George");

        names.stream()
                .map(name -> {
                    System.out.printf("Mapping 1: %s%n", name);
                    return name.toUpperCase();
                });

        names.stream()
                .map(name -> {
                    System.out.printf("Mapping 2: %s%n", name);
                    return name.toUpperCase();
                })
                .forEach(name -> {
                });

        List<String> upperCaseNames = names.stream()
                .map(name -> {
                    System.out.printf("Mapping 3: %s%n", name);
                    return name.toUpperCase();
                })
                .collect(Collectors.toList());
    }
}
```

Język Java: strumienie

Działanie strumieni możemy podzielić na 3 etapy:

- tworzenie strumienia,
- operacje pośrednie,
- operacje kończącą.

Po utworzeniu strumienia możemy wywołać wiele operacji pośrednik zwracających nowy, zmodyfikowany strumień. Każdy ze strumieni może zostać zakończony tylko przez jedną operację kończącą.

> Operacje na strumieniach zostają wykonywane dopiero w momencie, gdy są wymagane przez operację kończącą.

Język Java: strumienie

Podstawowe operacje na strumieniach:

`of()` - stworzenie strumienia na podstawie zadanych elementów.
`empty()` - stworzenie pustego strumienia.
`filter()` - filtrowanie elementów strumienia.
`map()` - mapowanie elementów strumienia.
`distinct()` - usuwanie duplikatów ze strumienia.
`sorted()` - sortowanie elementów w strumieniu.
`collect()` - redukcja strumienia.
`findFirst()` - pobranie pierwszego elementu.
`findAny()` - pobranie dowolnego elementu.

Język Java: strumienie

Strumienie mogą zostać zredukowane do innego typu. Najczęściej redukcja zachodzi to kolekcji. Strumienie Java wprowadzają podstawowe implementacje `Collectors` pozwalające na redukcję:

`Collectors.toList()` - redukcja do listy elementów.
`Collectors.toSet()` - redukcja do zbioru elementów.
`Collectors.toMap()` - redukcja do mapy elementów.
`Collectors.groupingBy()` - redukcja grupująca elementy na podstawie danej cechy.
`Collectors.joining()` - redukcja łącząca zwracająca ciąg znaków.

Programowanie: przykład 39

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.function.BiConsumer;
import java.util.function.BinaryOperator;
import java.util.function.Function;
import java.util.function.Supplier;
import java.util.stream.Collector;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Main {

    public static void main(String[] args) {
        List<String> names = Arrays.asList("Gary", "Jessica", "George", "Elisabeth", "George");

        forEach();
        customToListCollector(names);
        collectorsToList(names);
        groupingBy(names);
        joining(names);
    }

    public static void forEach() {
        Stream.of("Gary", "Jessica", "George", "Elisabeth", "George")
                .forEach(name -> System.out.println("Name: " + name));

        Stream.empty()
                .forEach(name -> System.out.println("Name: " + name));
    }

    public static void customToListCollector(List<String> initNames) {
        List<String> names = initNames.stream()
                .filter(name -> name.startsWith("G"))
                .map(name -> name.toUpperCase())
                .distinct()
                .sorted()
                .collect(new Collector<String, List<String>, List<String>>() {
                    @Override
                    public Supplier<List<String>> supplier() {
                        return () -> new ArrayList<String>();
                    }

                    @Override
                    public BiConsumer<List<String>, String> accumulator() {
                        return (items, item) -> items.add(item);
                    }

                    @Override
                    public BinaryOperator<List<String>> combiner() {
                        return (leftItems, rightItems) -> {
                            leftItems.addAll(rightItems);
                            return leftItems;
                        };
                    }

                    @Override
                    public Function<List<String>, List<String>> finisher() {
                        return items -> items;
                    }

                    @Override
                    public Set<Characteristics> characteristics() {
                        return new HashSet<>();
                    }
                });
        System.out.println("Names: " + names);
    }

    public static void collectorsToList(List<String> initNames) {
        List<String> names = initNames.stream()
                .filter(name -> name.startsWith("G"))
                .map(name -> name.toUpperCase())
                .distinct()
                .sorted()
                .collect(Collectors.toList());
        System.out.println("Names: " + names);
    }

    public static void groupingBy(List<String> initNames) {
        Map<String, List<String>> names = initNames.stream()
                .collect(Collectors.groupingBy(name -> name.substring(0, 1)));

        System.out.println("Names: " + names);
    }

    public static void joining(List<String> initNames) {
        String names = initNames.stream().collect(Collectors.joining(" and "));
        System.out.println("Names: " + names);
    }
}
```