Wyższa Szkoła Bankowa
we Wrocławiu

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: 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:

```
(paramers 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 36

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 37

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

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 38

```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);
    }
}
```

Język Java: Optional

Częstym problemem pojawiającym się w systemach jest błąd `NullPointerException`. Wynika to najczęściej z operacji na obiektach, które zakładają, że zmienna zawsze ma przypisaną wartość:

```
public static Integer percent(Integer value) {
    return value * 100;
}
```

Tak zdefiniowana metoda jest podatna na błędy. Wywołanie jej z `value = null` spowoduje błąd `NullPointerException`.

Język Java: Optional

Problem możemy rozwiązać, sprawdzając, czy `value` nie jest nullem.

```
public static Integer percent(Integer value) {
    if (value == null) {
        return null;
    }

   return value * 100;
}
```

Była to częsta praktyka do momentu wprowadzenia klasy `Optional`. Instrukcja warunkowa rozwiązuje problemy z `NullPointerException`.

Język Java: Optional

Głównym celem klasy `Optional` jest reprezentacja wartości opcjonalnej. Operacje są wykonywane jedynie w przypadku, w którym wartość zadana nie jest równa `null`. Wartość opcjonalną z pustą wartością możemy rozumieć jako zastępstwo referencji do `null`.

```
public static Integer percent(Integer value) {
    return Optional.ofNullable(value)
        .map(v -> v * 100)
        .orElse(null);
}
```

> Klasa `Optional` pozwala na łatwe radzenie sobie z wartościami `null` wymuszając dobre praktyki programistyczne.

Język Java: Optional

Do podstawowych operacji na klasie `Optional`:

`of()` - stworzenie obiektu z wartością na podstawie parametru lub błąd, gdy parametr jest równy `null`.
`ofNullable()` - stworzenie obiektu z wartością na podstawie parametru lub pustego obiektu, gdy parametr jest równy `null`.
`empty()` - stworzenie pustego obiektu.
`isPresent()` - sprawdzenie, czy wartość nie jest pusta.
`isEmpty()` - sprawdzenie, czy wartość jest pusta.
`get()` - pobranie wartości lub błąd, gdy wartość jest pusta.

Język Java: Optional

`orElse()` - pobranie wartości lub wartość domyślna, gdy wartość jest pusta.
`orElseGet()` - pobranie wartości lub wartość wygenerowanie wartości domyślnej, gdy wartość jest pusta.
`orElseThrow()` - pobranie wartości lub błąd, gdy wartość jest pusta.
`orElseThrow()` - pobranie wartości lub zadany błąd, gdy wartość jest pusta.
`or()` - pobranie innego obiektu z wartością, gdy wartość obecnego jest pusta.
`filter()` - filtrowanie wartości, jeżeli nie jest pusta.
`map()` - mapowanie wartości, jeżeli nie jest pusta.
`stream()` - konwersja obiektu na strumień.
`ifPresent()` - wykonanie operacji, jeżeli wartość nie jest pusta.

Programowanie: przykład 39

```
import java.util.Optional;

public class Main {

    public static void main(String[] args) {
        System.out.printf("First character: %s: %s%n", "bit", firstCharacter("bit").orElse('-'));
        System.out.printf("First character: %s: %s%n", "9bit", firstCharacter("9bit").orElse('-'));
        System.out.printf("First character: %s: %s%n", "", firstCharacter("").orElse('-'));
        System.out.printf("First character: %s: %s%n", null, firstCharacter(null).orElse('-'));

        System.out.printf("Starts with digit: %s: %s%n", "bit", startsWithDigit("bit"));
        System.out.printf("Starts with digit: %s: %s%n", "9bit", startsWithDigit("9bit"));
        System.out.printf("Starts with digit: %s: %s%n", "", startsWithDigit(""));
        System.out.printf("Starts with digit: %s: %s%n", null, startsWithDigit(null));
    }

    public static Optional<Character> firstCharacter(String value) {
        return Optional.ofNullable(value)
                .filter(v -> v.length() > 0)
                .map(v -> v.charAt(0));
    }

    public static boolean startsWithDigit(String value) {
        return firstCharacter(value).filter(c -> Character.isDigit(c)).isPresent();
    }
}
```

Programowanie: zadanie 23

Korzystając z klasy `Optional` zdefiniuj metodę sprawdzającą czy dana liczba jest cyfrą, której zapis przy użyciu instrukcji warunkowych byłby następujący:

```
public static boolean isDigitWithoutOptional(Integer value) {
    boolean isDigit = false;
    if (value != null && value >= 0 && value <= 9) {
        isDigit = true;
    }
    return isDigit;
}
```

Język Java: referencje do metod

Do tej pory tworzyliśmy wyrażenia lambda w postaci metod anonimowych.

```
Stream.of("one", null, "two").filter(value -> Objects.nonNull(value));
```

Istnieje specjalny typ wyrażeń `lambda`, które określane są mianem referencji do metod. To nic innego jak użycie danej funkcji w postaci wyrażenia `lambda`.

```
Stream.of("one", null, "two").filter(Objects::nonNull);
```

Istnieją 4 typy referencji.

Język Java: referencje do metod

Metody statyczne

```
Stream.of("one", null, "two").filter(value -> Objects.nonNull(value));
```

```
Stream.of("one", null, "two").filter(Objects::nonNull);
```

Język Java: referencje do metod

Metody poszczególnych obiektów

```
String prefix = "prefix-";
Stream.of("one", "two").map(value -> prefix.concat(value));
```

```
String prefix = "prefix-";
Stream.of("one", "two").map(prefix::concat);
```

Język Java: referencje do metod

Metody dowolnego obiektu określonego typu

```
Stream.of("one", "two").map(value -> value.toLowerCase());
```

```
Stream.of("one", "two").map(String::toLowerCase);
```

Język Java: referencje do metod

Konstruktory

```
Stream.of("one", "two").map(value -> new Item(value));
```

```
Stream.of("one", "two").map(Item::new);
```

Język Java: wyjątki

Wyjątek to nieoczekiwany błąd, który pojawia się podczas działania programu.

Przyczyn pojawiania się wyjątków jest wiele: zaczynając od błędów programistycznych, poprzez błędy w danych użytkowników aż po błędy związane z dostępem do danych. Powodują one nieoczekiwane zachowanie i możliwe niedeterministyczne zakończenie programu.

Język Java: wyjątki

Java wprowadza hierarchę klas wyjątków.

```
Throwable
|-----> Error
|-----> Exception
        |-----> RuntimeException
        |-----> Any other subclass of Exception
```

Throwable

Główna klasa w hierarchii definiująca dowolny błąd.

Error

Reprezentuje nieodwracalne zdarzenia takie jak: wycieki pamięci, przepełnienia stosu czy nieskończone wywołania rekurencyjne.

Język Java: wyjątki

Exception

Reprezentuje wyjątki, które mogą zostać obsłużone przez program. Zawiera informacje o zdarzeniu oraz stan programu, w którym dany błąd został zgłoszony.

RuntimeException

Reprezentuje wyjątki wynikające z błędów programistycznych.

Any other subclass of Exception

Reprezentuje wyjątki sprawdzane przez kompilator podczas kompilacji. Wymuszają obsługę z poziomu kodu źródłowego programu.

Język Java: typy wyjątków

Wyjątki możemy podzielić na 2 grupy, w zależności od konieczności ich obsługi.

Unchecked Exceptions

Wyjątki, które nie są sprawdzane podczas kompilacji. Do tej grupy należą `Error`, `RuntimeException` oraz ich wszystkie podklasy.

Przykłady `RuntimeException`:

- `NullPointerException`, zgłaszany podczas próby dostępu do niezainicjalizowanej zmiennej.
- `ArrayIndexOutOfBoundsException`, zgłaszany podczas próby dostępu do nieistniejącego indeksu tablicy lub kolekcji.
- `ArithmeticException`, zgłaszany podczas dzielenia przez 0.
- `IllegalArgumentException`, zgłaszany podczas niepopranego użycia danej klasy lub metody.

Język Java: typy wyjątków

Checked Exceptions

Wyjątki, które są sprawdzane podczas kompilacji. Do tej grupy należą wszystkie pozostałe wyjątki. Wyjątki te muszą zostać obsłużone.

Przykłady:

- `IOException`, zgłaszany podczas błędów w operacjach I/O.
- `FileNotFoundException`, zgłaszany podczas prób dostępu do nieistniejącego pliku.
- `InterruptedException`, zgłaszany podczas przerwania wątku, który jest oczekujący.

Język Java: typy wyjątków

Możemy tworzyć własne typy wyjątków poprzez rozszerzanie klas `Exception` oraz `RuntimeException`.

> `Unchecked Excetions` nie są zwyczajowo obsługiwane, gdyż wynikają z błędów programistycznych. Ich usunięcie powinno odbywać się przez naprawienie wadliwego kodu.

Język Java: zgłaszanie wyjątków

Do zgłoszenia wyjątku używamy słowa kluczowego `throw`.

```
throw new ExceptionType();
```

Dla przykładu:

```
public static Integer percent(Integer value) {
    if (value == null) {
        throw new IllegalArgumentException("Value may not be null");
    }
    return value * 100;
}
```

Język Java: obsługa wyjątków try

Do obsługi wyjątków używamy bloku `try...catch...finally`

```
try {
    /* ... */
} catch (ExceptionClass1 e1) {
    /* ... */
} catch (ExceptionClass2 | ExceptionClass3 e1) {
    /* ... */
} finally {
    /* ... */
}
```

- Wyjątek może zostać zgłoszony w bloku `try {}`
- Wyrażenie `catch` jest odpowiedzialne za obsługę wyjątku o typie `ExceptionClass1`. Dla wyjątków o takim samym sposobie obsługi możemy użyć uwspólnionego wyrażenia `ExceptionClass2 | ExceptionClass3`.
- Blok `finally {}` jest wykonywany po blokach `try {} catch {}` nawet w przypadku zgłoszenia błędu. Jest to blok opcjonalny.

Język Java: obsługa wyjątków try

Dla przykładu:

```
public static Integer safeParseInt(String value) {
    try {
        return Integer.parseInt("i");
    } catch (NumberFormatException ex) {
        System.out.println("Failed to parse integer: " + ex.getMessage());
    } finally {
        System.out.println("Finished integer parsing");
    }
    return null;
}
```

> Obsługa wyjątku typu `RuntimeException` posłużyła jedynie do prezentacji użycia bloku `try`. Kod powinien być napisany w sposób, który nie dopuszcza do takich sytuacji.

Język Java: obsługa wyjątków

Obsługa wyjątków odbywa się sekwencyjnie. Pierwszy napotkany blok `catch`, którego typ jest zgodny z typem zgłoszonego wyjątku, zostanie użyty do obsługi błędu.

Definicje `catch` powinny zaczynać się od typów bardziej specyficznych, a kończyć na bardziej ogólnych.

Język Java: deklaracja wyjątków

Do deklaracji typów wyjątków zwracanych przez metodę używamy słowa kluczowego `throws`.

Dla przykładu:

```
public static Integer percent(Integer value) throws IllegalArgumentException {
    if (value == null) {
        throw new IllegalArgumentException("Value may not be null");
    }
    return value * 100;
}
```

> Wyjątki typu `unchecked` nie muszą być zadeklarowany podczas definicji metody.

> W przypadku, gdy metoda zgłasza wyjątek typu `checked`, który nie został obsłużony w jej ciele, musi on zostać zadeklarowany podczas definicji metody.

Programowanie: przykład 40

Wyjątek typu `checked` wraz z jego obsługą.

```
import java.util.stream.Stream;

public class Main {

    public static void main(String[] args) {
        Stream
                .of(
                        new Person("Andrzej", 1984),
                        new Person("Andrzej", 2020),
                        new Person("       ", 1984),
                        new Person("       ", 2020))
                .forEach(Main::validate);
    }

    public static void validate(Person person) {
        try {
            Validator.validateName(person);
            Validator.validateAdult(person);
        } catch (ValidationException ex) {
            System.out.println("Validation failed: " + ex.getMessage());
        }
    }
}
```

Język Java: operacje wejścia/wyjścia (I/O)

Strumień wejściowy (input stream) jest odpowiedzialny za czytanie danych ze źródła. Strumień wyjściowy (output stream) jest odpowiedzialny za pisanie danych do źródła.

Istnieją dwa główne typy strumieni:

Strumienie binarne (byte streams)

Używane do odczytu pojedynczych bajtów danych. Do ich obsługi służą wszystkie podklasy `InputStream` oraz `OutputStream`.

Strumienie binarne (byte streams)

Używane do odczytu pojedynczych znaków danych. Do ich obsługi służą wszystkie podklasy `Reader` oraz `Writer`.

Język Java: operacje wejścia/wyjścia (I/O)

Do tej pory spotkaliśmy się ze strumieniami przy okazji czytania oraz pisania na konsolę.

`System.out` - obiekt typu `PrintStream`, rozszerzający `OutputStream`.
`System.in` - obiekt typu `InputStream`

Java posiada bardzo rozbudowaną strukturę strumieni wejścia i wyjścia udostępnioną w pakiecie `java.io`. Skupimy się jedynie na podstawowych przykładach pozwalających na odczyt danych z pliku oraz ich zapis do pliku.

W najnowszych wersjach Javy został wprowadzony pakiet `java.nio` (new I/O) rozszerzający strumienie o dodatkowe mechanizmy.

> Przy okazji strumieni warto wspomnieć, że w wielu miejscach używają klas opakowujących. Każda z kolejnych warstw dodaje nowe funkcjonalności, zwiększając poziom abstrakcji.

Programowanie: przykład 41

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.time.LocalDateTime;
import java.util.Optional;

public class Main {

    private static final String INPUT_FILE = "/tmp/input";
    private static final String OUTPUT_FILE = "/tmp/output";

    public static void main(String args[]) {
        readFile(INPUT_FILE).ifPresent(content -> writeFile(OUTPUT_FILE, content + LocalDateTime.now()));
    }

    public static Optional<String> readFile(String filename) {
        FileInputStream input = null;
        try {
            input = new FileInputStream(filename);
            StringBuilder stringBuilder = new StringBuilder();

            int i = input.read();
            while(i != -1) {
                stringBuilder.append((char) i);
                i = input.read();
            }

            input.close();
            return Optional.of(stringBuilder.toString());
        }  catch(Exception ex) {
            try {
                if (input != null) {
                    input.close();
                }
            } catch (IOException ioEx) {
            }
            System.out.println("Reading failed: " + ex.getMessage());
        }
        return Optional.empty();
    }

    public static void writeFile(String filename, String content) {
        FileOutputStream output = null;
        try {
            output = new FileOutputStream(filename);
            byte[] array = content.getBytes();
            output.write(array);
            output.close();
        } catch(Exception ex) {
            try {
                if (output != null) {
                    output.close();
                }
            } catch (IOException ioEx) {
            }
            System.out.println("Writing failed: " + ex.getMessage());
        }
    }
}
```

Język Java: obsługa wyjątków try-with-resource

Obsługa wyjątków przy pomocy `try-with-resource` pozwala na automatyczne zamykanie zasobów bez konieczności wywoływania metody `close()` explicite w bloku `finally`.

```
try (resource declaration) {
  /* ... */
} catch (ExceptionType e1) {
  * ... */
}
```

Zasób (jeden lub więcej) musi zostać zadeklarowany oraz zainicjalizowany w wyrażeniu `try()`.

Język Java: obsługa wyjątków try-with-resource

Dla przykładu:

```
try (FileReader input = new FileReader(filename)) {
    /* ... */
} catch(IOException ex) {
}
```

Dowolny obiekt, który implementuje `java.lang.AutoCloseable`, również te, które implementują `java.io.Closeable` mogą zostać użyte jako zasoby.

> Wyrażenie `try-with-resource` automatycznie zamyka wszystkie zasoby po wykonaniu bloku `try` nawet w przypadku wystąpienia błędu.

Programowanie: przykład 42

```
import java.io.FileReader;
import java.io.FileWriter;
import java.time.LocalDateTime;
import java.util.Optional;

public class Main {

    private static final String INPUT_FILE = "/tmp/input";
    private static final String OUTPUT_FILE = "/tmp/output";

    public static void main(String args[]) {
        readFile(INPUT_FILE).ifPresent(content -> writeFile(OUTPUT_FILE, content + LocalDateTime.now()));
    }

    private static Optional<String> readFile(String filename) {
        char[] array = new char[1];

        try (FileReader input = new FileReader(filename)) {
            StringBuilder stringBuilder = new StringBuilder();

            int result = input.read(array);
            while (result != -1) {
                stringBuilder.append(array);
                result = input.read(array);
            }

            return Optional.of(stringBuilder.toString());
        } catch(Exception ex) {
            System.out.println("Reading failed: " + ex.getMessage());
        }

        return Optional.empty();
    }

    private static void writeFile(String filename, String content) {
        try (FileWriter output = new FileWriter(filename)) {
            output.write(content);
        } catch (Exception ex) {
            System.out.println("Writing failed: " + ex.getMessage());
        }
    }
}
```

Język Java: obsługa systemu plików

W Javie podstawową klasą do operacji na plikach jest klasa `java.io.File`. Pozwala ona na zarządzanie systemem plików, ścieżkami plików oraz ich własnościami.

Wraz z pakietem `java.nio.file` zostały wprowadzone dodatkowe klasy do zarządzania ścieżkami:

- `java.nio.file.Path`, reprezentującą ścieżkę pliku,
- `java.nio.file.Files`, definiującą operacje na ścieżkach plików oraz systemie plików.

Istnieje szereg innych klas współpracujących oraz usprawniających działania na systemie plików.

Programowanie: przykład 43

Przykładowe użycie klas `File` oraz `Path` do przeglądania systemu plików.

```
import java.io.File;
import java.io.IOException;
import java.nio.file.FileVisitResult;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.SimpleFileVisitor;
import java.nio.file.attribute.BasicFileAttributes;
import java.util.Arrays;
import java.util.Optional;
import java.util.stream.Stream;

public class Main {

    public static void main(String[] args) throws IOException {

        File file = new File(".");
        System.out.println("Absolute path: " + file.getCanonicalFile().getAbsolutePath());
        System.out.println("Is directory: " + file.isDirectory());
        System.out.println("Is directory: " + file.isFile());
        System.out.println("Is directory: " + file.exists());

        Optional.ofNullable(file.listFiles()).stream().flatMap(Arrays::stream)
                .forEach(subFile -> {
                    try {
                        System.out.println("Sub file absolute path: " + subFile.getCanonicalFile().getAbsolutePath());
                    } catch (Exception ex) {
                        System.out.println("Something went wrong: " + ex.getMessage());
                    }
                });

        Path path = file.toPath().toAbsolutePath().normalize();
        System.out.println("Absolute path: " + path.toString());
        System.out.println("Parent absolute path: " + path.getParent().toString());

        try (Stream<Path> stream = Files.list(path)) {
            stream.forEach(subPath -> {
                System.out.println("Sub file absolute path: " + subPath.toAbsolutePath().toString());
            });
        }

        try (Stream<Path> stream = Files.walk(path, 2)) {
            stream.forEach(subPath -> {
                System.out.println("Sub file absolute path: " + subPath.toAbsolutePath().toString());
            });
        }

        Files.walkFileTree(path, new SimpleFileVisitor<>() {
            @Override
            public FileVisitResult visitFile(Path subPath, BasicFileAttributes attrs) {
                if (!Files.isDirectory(subPath) && subPath.toString().endsWith("java")) {
                    System.out.println("Sub file absolute path: " + subPath.toAbsolutePath().toString());
                }
                return FileVisitResult.CONTINUE;
            }
        });
    }
}
```

Język Java: obsługa czasu

Czas lokalny opisuje czas w danej strefie czasowej.

`2021-01-01T00:00`

Dodatkowo czas lokalny może zostać rozszerzony o informacje o:

- przesunięciu czasowym definiowanym w godzinach względem UTC
  `2021-01-01T00:00+02:00`
- strefie czasowej definiowanej identyfikatorem strefy czasowej oraz przesunięciem czasowym względem UTC
  `2021-07-01T00:00+02:00[Europe/Warsaw]`

Język Java: obsługa czasu

Jedna strefa czasowa może przyjmować dwa różne przesunięcia czasowe w zależności od pory roku.

Dla przykładu polska strefa czasowa: `Europe/Warsaw` przyjmuje przesunięcie +1h w zimie oraz +2h w lecie. Związane jest to ze zmianą czasu na czas letni (ang. Daylight Saving Time, DST).

```
Czas zimowy: 2021-01-01T00:00+01:00[Europe/Warsaw]
Czas letni: 2021-07-01T00:00+02:00[Europe/Warsaw]
```

Język Java: obsługa czasu

UTC (ang. Universal Time Coordinated) to wzorcowy czas koordynowany względem czasu słonecznego oraz nieuwzględniający zmian czasu.

Znany również pod swoją wojskową nazwą Zulu time (czas Zulu, Z). Litera "z" oznacza południk zerowy, czyli długość geograficzną 0.

Jest następcą GMT (ang. Greenwich Mean Time).

Język Java: obsługa czasu

Do zarządzenia czasem i datą służy pakiet `java.time`. Do głównych klas należą:

- `Instant`: reprezentuje dany moment w czasie.
- `LocalDate`: informacje o lokalnej dacie.
- `LocalDateTime`: informacje o lokalnej dacie wraz z czasem.
- `OffsetDateTime`: informacje o dacie wraz z czasem wzbogaconej o offset czasowy.
- `ZonedDateTime`: informacje o dacie wraz z czasem wzbogaconej o offset czasowy oraz strefę czasową.

Język Java: obsługa czasu

W systemach informatycznych używane jest pojęcie `Unix epoch`. Jest to ilość sekund, które upłynęły od 1970-01-01 00:00 UTC.

Obiekty `Instant` przechowują dane w postaci sekund i nanosekund w formacie `Unix epoch`. Obiekty te nie przechowują kontekstu przesunięcia czasowego czy strefy czasowej.

Stała `Instant.EPOCH` wewnętrznie reprezentowana jest przez 0 sekund i 0 nanosekund.

Programowanie: przykład 44

Przykładowe użycie klas `LocalDateTime`, `ZonedDateTime` oraz `ZonedDateTime`.

```
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;

public class Main {

    public static void main(String[] args) {
        System.out.printf("##### Local%n");

        // Winter
        LocalDateTime winterLocal = LocalDateTime.of(2021, 1, 1, 0, 0, 0);
        // Summer
        LocalDateTime summerLocal = LocalDateTime.parse("2021-07-01T00:00:00");

        System.out.println("Winter Local: " + winterLocal);
        System.out.println("Summer Local: " + summerLocal);

        System.out.printf("%n##### Same instant%n");

        // Zones
        ZoneId warsawZone = ZoneId.of("Europe/Warsaw");
        ZoneId dublinZone = ZoneId.of("Europe/Dublin");

        ZonedDateTime winterWarsawZone = ZonedDateTime.of(winterLocal, warsawZone);
        ZonedDateTime summerWarsawZone = ZonedDateTime.of(summerLocal, warsawZone);

        System.out.println("Winter Warsaw Zoned: " + winterWarsawZone);
        System.out.println("Summer Warsaw Zoned: " + summerWarsawZone);

        System.out.println("Winter Warsaw Offset: " + winterWarsawZone.toOffsetDateTime());
        System.out.println("Summer Warsaw Offset: " + summerWarsawZone.toOffsetDateTime());

        System.out.println("Winter Dublin Zoned: " + winterWarsawZone.withZoneSameInstant(dublinZone));
        System.out.println("Summer Dublin Zoned: " + summerWarsawZone.withZoneSameInstant(dublinZone));

        System.out.println("Winter Dublin Offset: " + winterWarsawZone.withZoneSameInstant(dublinZone).toOffsetDateTime());
        System.out.println("Summer Dublin Offset: " + summerWarsawZone.withZoneSameInstant(dublinZone).toOffsetDateTime());

        System.out.printf("%n##### Same local%n");

        System.out.println("Winter Warsaw Zoned: " + winterWarsawZone);
        System.out.println("Summer Warsaw Zoned: " + summerWarsawZone);

        System.out.println("Winter Dublin Zoned (same local): " + winterWarsawZone.withZoneSameLocal(dublinZone));
        System.out.println("Summer Dublin Zoned (same local): " + summerWarsawZone.withZoneSameLocal(dublinZone));
    }
}
```

Programowanie: przykład 45

Formatowanie klas opisujących datę oraz czas.

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.time.format.FormatStyle;

public class Main {

    public static void main(String[] args) {
        LocalDateTime value = LocalDateTime.parse("2021-07-01T01:21:31");

        System.out.println("toString: " + value.toString());
        System.out.println("ISO_DATE_TIME: " + value.format(DateTimeFormatter.ISO_DATE_TIME));
        System.out.println("ISO_LOCAL_DATE_TIME: " + value.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
        System.out.println("ISO_LOCAL_DATE: " + value.format(DateTimeFormatter.ISO_LOCAL_DATE));
        System.out.println("LocalizedDateTime MEDIUM: " + value.format(DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM)));
        System.out.println("LocalizedDate FULL: " + value.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)));
        System.out.println("custom: " + value.format(DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss")));
        System.out.println("custom: " + value.format(DateTimeFormatter.ofPattern("yy/M/d H:m:s")));
        System.out.println("custom: " + value.format(DateTimeFormatter.ofPattern("yy/M/d H:m:s")));
    }
}
```