Kolokwium 2

15 червня 2024 р. 19:30

PLAN.

- [x] Pakiety
- [x] Testy
- [x] Wielowątkowość
- [ ] Serwer webowy
- [ ] Połączenia sieciowe
- [ ] Próba kolokwium

PAKIETY

Pakiety (Packages)

Co to są pakiety?

Pakiety w Javie to sposób organizowania klas i interfejsów w logiczne grupy. Pomagają one zarządzać kodem i unikać konfliktów nazw.

Jak używać pakietów?

Pakiety deklaruje się na początku pliku źródłowego za pomocą słowa kluczowego package:

```
package com.example.myapp;
```

Następnie w obrębie pakietu można tworzyć klasy:

```
package com.example.myapp;

public class MyClass {
    // kod klasy
}
```

Aby użyć klasy z innego pakietu, używamy słowa kluczowego import:

import com.example.myapp.MyClass;

```
public class AnotherClass {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
    }
}
```

Do czego używać pakietów?

Pakiety są używane do:

- Organizowania klas w logiczne grupy.
- Unikania konfliktów nazw (dwie klasy mogą mieć tę samą nazwę, jeśli są w różnych pakietach).
- Kontroli dostępu (modyfikatory dostępu, jak public, protected, private mają różne znaczenia w kontekście pakietów).

Deployment

Co to jest deployment?

Deployment (wdrażanie) to proces przenoszenia aplikacji z fazy deweloperskiej do środowiska produkcyjnego, gdzie aplikacja będzie dostępna dla użytkowników końcowych.

Jak wygląda deployment aplikacji Java?

W przypadku aplikacji Java, deployment może obejmować:

- Kompilację kodu źródłowego do plików .class.
- Pakowanie aplikacji w pliki JAR (Java ARchive) lub WAR (Web Application Archive) dla aplikacji webowych.
- Przeniesienie tych plików na serwer produkcyjny.
- Uruchomienie aplikacji na serwerze.

Do czego używać deploymentu?

Deploymentu używa się, aby:

- Udostępnić aplikację użytkownikom końcowym.
- Przetestować aplikację w środowisku zbliżonym do produkcyjnego.
- Aktualizować aplikacje już działające w środowisku produkcyjnym.

Maven

Co to jest Maven?

Maven to narzędzie do zarządzania projektami i budowania aplikacji w Javie. Ułatwia kompilację, pakowanie, testowanie i zarządzanie zależnościami projektu.

Jak używać Maven?

Maven używa pliku konfiguracyjnego o nazwie pom.xml (Project Object Model), który zawiera informacje o projekcie i jego zależnościach. Oto przykładowy plik pom.xml:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>my-app</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Maven automatycznie pobiera zależności z centralnego repozytorium Maven i zarządza nimi za nas.

Do czego używać Maven?

Maven jest używany do:
- Zarządzania zależnościami projektów (automatyczne pobieranie bibliotek).
- Automatyzacji procesu budowania aplikacji (kompilacja, testowanie, pakowanie).
- Standaryzacji struktury projektów (projekty Maven mają wspólną strukturę katalogów).
- Łatwiejszej integracji z innymi narzędziami (np. Jenkins, Git).

TESTY

Czym są testy?

Testowanie oprogramowania to proces oceny funkcjonalności, wydajności, bezpieczeństwa oraz zgodności aplikacji z wymaganiami. Ma na celu wykrycie błędów i zapewnienie wysokiej jakości produktu końcowego. Istnieje wiele rodzajów testów, z których każdy koncentruje się na innym aspekcie oprogramowania:
- Testy jednostkowe
- Testy integracyjne
- Testy funkcjonalne
- Testy akceptacyjne
- Testy wydajnościowe
- Testy regresji
- Testy bezpieczeństwa

Testy jednostkowe (Unit Tests)

Testy jednostkowe sprawdzają pojedyncze "jednostki" kodu, takie jak funkcje, metody czy klasy, aby upewnić się, że działają one poprawnie. Każdy test jest zazwyczaj izolowany od innych testów i środowiska, co oznacza, że testuje tylko jedną jednostkę funkcjonalności.

- Przykład: Testowanie metody add w klasie Calculator, aby sprawdzić, czy poprawnie dodaje dwie liczby.
- Biblioteka: JUnit jest często używaną biblioteką w Javie do tworzenia i uruchamiania testów jednostkowych.

JUnit

JUnit to popularna biblioteka w Javie, używana do tworzenia i uruchamiania testów jednostkowych. Pozwala na definiowanie testów, które automatycznie sprawdzają poprawność działania kodu. Poniżej znajduje się przykładowy kod testu jednostkowego z użyciem JUnit:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class CalculatorTest {

    @Test
    void testAdd() {
        Calculator calc = new Calculator();
        int result = calc.add(2, 3);
        assertEquals(5, result, "2 + 3 should equal 5");
    }
}
```

Kluczowe funkcje JUnit:
- Testowanie: Łatwe definiowanie testów za pomocą adnotacji @Test.
- Assercje: Sprawdzanie wyników testów za pomocą metod takich jak assertEquals, assertTrue,

assertFalse.
- Organizacja testów: Grupowanie testów w klasy testowe.
- Automatyzacja: Integracja z narzędziami do ciągłej integracji (CI) jak Jenkins, GitLab CI itp.

Testy integracyjne (Integration Tests)

Testy integracyjne sprawdzają interakcje pomiędzy różnymi modułami lub komponentami oprogramowania, aby upewnić się, że współpracują one ze sobą zgodnie z oczekiwaniami.

- Przykład: Testowanie, czy moduł autoryzacji poprawnie współpracuje z bazą danych użytkowników.

Testy funkcjonalne (Functional Tests)

Testy funkcjonalne sprawdzają działanie systemu zgodnie z określonymi wymaganiami funkcjonalnymi. Koncentrują się na testowaniu zewnętrznego zachowania systemu w różnych scenariuszach użytkowania.

- Przykład: Testowanie, czy użytkownik może poprawnie zarejestrować się na stronie internetowej.

Testy akceptacyjne (Acceptance Tests)

Testy akceptacyjne są przeprowadzane, aby sprawdzić, czy system spełnia wymagania i oczekiwania użytkownika końcowego. Są to testy na poziomie aplikacji, często wykonywane przez klienta lub zespół QA.

- Przykład: Testowanie, czy wszystkie funkcjonalności w aplikacji są zgodne z wymaganiami przedstawionymi przez klienta.

Testy wydajnościowe (Performance Tests)

Testy wydajnościowe mierzą, jak dobrze system działa pod określonym obciążeniem. Mogą obejmować testy szybkości, skalowalności i stabilności systemu.

- Przykład: Testowanie, jak szybko strona internetowa ładuje się przy dużym ruchu użytkowników.

Testy regresji (Regression Tests)

Testy regresji sprawdzają, czy nowe zmiany w kodzie nie wprowadziły nieoczekiwanych błędów w już istniejących funkcjonalnościach. Są często automatyzowane, aby szybko wykrywać problemy po wprowadzeniu nowych zmian.

- Przykład: Ponowne testowanie wszystkich funkcjonalności aplikacji po dodaniu nowej funkcji, aby upewnić się, że nic nie zostało przypadkowo zepsute.

Testy bezpieczeństwa (Security Tests)

Testy bezpieczeństwa oceniają, czy system jest odporny na ataki i czy dane są chronione przed nieautoryzowanym dostępem. Skupiają się na wykrywaniu luk i podatności.

- Przykład: Testowanie, czy aplikacja jest odporna na ataki typu SQL Injection.

WIELOWĄTKOWOŚĆ

Wątki

Czym są wątki?

Wątki to niezależne ścieżki wykonania w programie, które pozwalają na wykonywanie wielu zadań jednocześnie. Każdy wątek działa w ramach tego samego procesu, dzieląc zasoby, takie jak pamięć, ale wykonując różne zadania równolegle.

Interfejs Runnable

Runnable jest funkcjonalnym interfejsem w Javie, który posiada jedną metodę run(). Implementując Runnable, definiujesz kod, który ma być wykonany w wątku. Przykład:

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Wątek działa!");
    }
}
```

Aby uruchomić wątek, tworzysz instancję Thread i przekazujesz do niej instancję Runnable:

```
public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable());
        thread.start();
    }
}
```

Stany wątku

Wątki w Javie mogą być w jednym z kilku stanów:

- New: Wątek został utworzony, ale jeszcze nie uruchomiony (start() nie zostało wywołane).
- Runnable: Wątek jest gotowy do wykonania i oczekuje na przydzielenie czasu procesora.
- Blocked: Wątek czeka na monitor, aby wejść do sekcji krytycznej.
- Waiting: Wątek czeka na inny wątek, aby zakończyć pewne działanie.
- Timed Waiting: Wątek czeka na określony czas.
- Terminated: Wątek zakończył wykonanie.

Jak i do czego używać wątki?

Wątki są idealne do zadań, które można wykonywać równolegle, takich jak:

- Przetwarzanie dużych zbiorów danych: Można podzielić dane na mniejsze fragmenty i przetwarzać je równocześnie.
- Operacje sieciowe: Można obsługiwać wiele połączeń sieciowych jednocześnie, np. serwery HTTP.
- Zadania wejścia/wyjścia: Można wykonywać operacje na plikach lub bazach danych bez blokowania głównego wątku.

Współdzielone zasoby

Kiedy wiele wątków ma dostęp do wspólnych zasobów, może dojść do konfliktów, takich jak wyścigi danych. Aby tego uniknąć, używa się mechanizmów synchronizacji, takich jak:

- synchronized: Służy do synchronizacji metod lub bloków kodu.
- Locks: Zaawansowane mechanizmy blokad z java.util.concurrent.locks.

Przykład użycia synchronized:

```
public class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public synchronized int getCount() {
        return count;
    }
}
```

Pula wątków

Executor

Pula wątków pozwala na zarządzanie grupą wątków, które mogą być ponownie używane do wykonywania zadań, co jest bardziej efektywne niż tworzenie nowych wątków dla każdego zadania. Executor jest interfejsem do zarządzania wykonaniem wątków.

Przykład użycia ExecutorService:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Main {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 10; i++) {
            executorService.submit(new MyRunnable());
        }
        executorService.shutdown();
    }
}
```

Future

Future jest interfejsem do reprezentowania wyniku asynchronicznego obliczenia. Umożliwia sprawdzenie, czy obliczenie zostało zakończone, pobranie wyniku lub anulowanie obliczenia.

Przykład użycia Future:

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Main {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(5);
        Future<Integer> future = executorService.submit(new
Callable<Integer>() {
            @Override
            public Integer call() {
                return 42;
```

```
            }
        });

        try {
            Integer result = future.get();
            System.out.println("Wynik: " + result);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        } finally {
            executorService.shutdown();
        }
    }
}
```

SERWER WEBOWY

Serwer Webowy

Serwer webowy to oprogramowanie, które obsługuje żądania HTTP/HTTPS od klientów (zazwyczaj przeglądarek internetowych) i zwraca odpowiedzi, najczęściej w postaci stron HTML, plików, danych JSON itp. W Javie popularnymi serwerami webowymi są Apache Tomcat, Jetty oraz serwery aplikacji jak JBoss czy GlassFish.

HTTP (Hypertext Transfer Protocol)

HTTP to protokół komunikacyjny używany do przesyłania danych w sieci WWW. Klient (np. przeglądarka) wysyła żądania HTTP do serwera, który zwraca odpowiedzi. Każde żądanie HTTP składa się z:

- Metody HTTP (GET, POST, PUT, DELETE itp.)
- URL (adres zasobu)
- Nagłówków (informacje dodatkowe jak typ treści, autoryzacja)
- Ciała żądania (w przypadku metod jak POST lub PUT)

Odpowiedź HTTP zawiera:

- Kod statusu (np. 200 OK, 404 Not Found)
- Nagłówki
- Ciało odpowiedzi (zawartość strony, dane JSON, pliki itp.)

Przykład żądania HTTP GET:

```
GET /api/items HTTP/1.1
Host: www.example.com
```

Przykład odpowiedzi HTTP:

```
HTTP/1.1 200 OK
Content-Type: application/json

[
    {"id": 1, "name": "Item1"},
    {"id": 2, "name": "Item2"}
]
```

REST (Representational State Transfer)

REST to architektoniczny styl projektowania systemów sieciowych, który korzysta z protokołu HTTP. RESTful API (Application Programming Interface) to interfejs, który pozwala na komunikację między

różnymi systemami przez sieć. Zasady REST obejmują:

- Zasoby identyfikowane przez URI (Uniform Resource Identifier).
- Stany reprezentacji zasobu, manipulowane za pomocą metod HTTP.
- Bezstanowość - każde żądanie od klienta do serwera musi zawierać wszystkie informacje potrzebne do zrozumienia i przetworzenia żądania.
- Kod statusu HTTP do informowania o wyniku operacji.

Przykład RESTful API:

```
Endpoint: GET /api/items
```

Odpowiedź:

```
[
    {"id": 1, "name": "Item1"},
    {"id": 2, "name": "Item2"}
]
```

Spring Framework

Spring to potężny framework dla Javy, który ułatwia tworzenie aplikacji webowych i enterprise. Składa się z wielu modułów, takich jak Spring Core, Spring MVC, Spring Data, Spring Security i inne.

Spring MVC (Model-View-Controller)

Spring MVC to część Spring Framework, która wspiera wzorzec projektowy Model-View-Controller (MVC). Wzorzec MVC rozdziela aplikację na trzy główne komponenty:

- Model: Reprezentuje dane aplikacji oraz logikę biznesową.
- View: Prezentuje dane użytkownikowi. W Spring MVC widok może być reprezentowany przez JSP, Thymeleaf, FreeMarker itp.
- Controller: Przechwytuje żądania HTTP, przetwarza je (może współpracować z Modelem), i zwraca odpowiednie widoki lub dane.

Jak działa Spring MVC

- DispatcherServlet: Wszystkie żądania HTTP są przechwytywane przez DispatcherServlet, który jest centralnym punktem w Spring MVC.
- Handler Mapping: DispatcherServlet korzysta z handler mapping, aby znaleźć odpowiedni kontroler dla żądania.
- Controller: Kontroler przetwarza żądanie, współpracuje z Modelem (logika biznesowa) i zwraca nazwę widoku lub dane.
- View Resolver: DispatcherServlet używa view resolver, aby znaleźć odpowiedni widok na podstawie nazwy widoku zwróconej przez kontroler.
- View: Widok renderuje odpowiedź, którą DispatcherServlet zwraca do klienta.

Tworzenie RESTful API w Spring

Konfiguracja Spring Boot
Spring Boot ułatwia konfigurację aplikacji Spring dzięki konwencji nad konfiguracją. Możemy stworzyć nowy projekt Spring Boot na stronie Spring Initializr.

Kontroler REST
W Spring, kontrolery REST definiuje się za pomocą adnotacji @RestController i @RequestMapping. Oto przykład prostego kontrolera REST:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
```

```
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.ArrayList;
import java.util.List;

@RestController
@RequestMapping("/api")
public class MyRestController {

    private List<Item> items = new ArrayList<>();

    @GetMapping("/items")
    public List<Item> getAllItems() {
        return items;
    }

    @GetMapping("/items/{id}")
    public Item getItem(@PathVariable int id) {
        return items.stream()
                    .filter(item -> item.getId() == id)
                    .findFirst()
                    .orElse(null);
    }

    @PostMapping("/items")
    public Item addItem(@RequestBody Item item) {
        items.add(item);
        return item;
    }
}
```

W tym przykładzie:

@RestController oznacza, że klasa jest kontrolerem REST i jej metody zwracają dane bezpośrednio w formacie JSON lub XML.
@RequestMapping("/api") ustawia podstawową ścieżkę dla wszystkich metod w tej klasie.
@GetMapping, @PostMapping itp. definiują, które metody HTTP są obsługiwane przez konkretne metody w kontrolerze.
@PathVariable służy do pobierania zmiennych ścieżki z URL, a @RequestBody do pobierania danych z ciała żądania.
Model
Model to klasa reprezentująca dane, np. Item:

```
public class Item {
   private int id;
   private String name;

   // Gettery i settery
}
```

Korzystanie z Spring MVC i REST

- Uruchomienie serwera: Tworząc aplikację Spring Boot, serwer webowy (np. embedded Tomcat) jest uruchamiany automatycznie.
- Tworzenie kontrolerów: Definiujemy klasy kontrolerów z odpowiednimi metodami obsługującymi żądania.
- Model: Tworzymy klasy modelu, które reprezentują dane.

- Konfiguracja widoków (opcjonalnie): W przypadku aplikacji MVC, definiujemy widoki (np. pliki JSP, Thymeleaf).

Spring Boot automatycznie konfiguruje aplikację i uruchamia serwer webowy, co pozwala skupić się na logice biznesowej i kontrolerach. RESTful API umożliwia komunikację między systemami za pomocą żądań HTTP, a Spring MVC ułatwia zarządzanie kontrolerami i widokami.

Przykład Aplikacji Spring Boot
Konfiguracja projektu
Stwórz projekt Spring Boot za pomocą Spring Initializr, wybierając zależności "Spring Web" i "Thymeleaf".

Plik pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Plik główny aplikacji

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

Kontroler MVC

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class MyMvcController {

    @GetMapping("/greeting")
    public String greeting(Model model) {
        model.addAttribute("message", "Hello, World!");
        return "greeting";
    }
}
```

Widok Thymeleaf (src/main/resources/templates/greeting.html)

```
html
```

```
Копировать код
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Greeting</title>
</head>
<body>
```

1) Add to main:
   importorg.springframework.boot.autoconfigure.SpringBootApplication;
   importorg.springframework.boot.SpringApplication;
2) Upper to main:
   @SpringBootApplication
3) In main
   SpringApplication.*run*(Main.class,args);