# Class

- În Java, o clasă este un model sau un șablon pentru crearea de obiecte. Definește proprietățile (câmpuri/variabile) și comportamentele (metode) pe care le va avea un obiect din acea clasă.

# Obiect-

Un obiect în Java este o instanță a unei clase care are o stare (câmpuri/variabile) (fields/variables) și un comportament (metode/funcții) (methods/functions). . Obiectele sunt create dintr-o clasă și reprezintă entități din lumea reală.

# Constructor

- Un constructor în Java este o metodă specială folosită pentru a inițializa obiecte. Este apelat automat atunci când este creat un obiect. Constructorul stabilește valorile inițiale pentru atributele obiectului și asigură configurarea corectă.

# Variable-

. Local Variable

Declared inside a method or block.

Only accessible within that method/block.

Must be initialized before use.

Instance Variable (Non-Static Variable)

Declared inside a class but outside methods.

Each object gets its own copy of instance variables.

Default values: 0 for numbers, null for objects, false for booleans.

Static Variable (Class Variable)

Declared using the static keyword.

Shared among all objects of a class (common memory).

Initialized only once at class loading time.

# Errori

Compile time error - greseala de sintaxa;

Runtime errors -exceptii

Logical errors -2+2=5

# Set

**Hash set**  accepta 1 element null, nu mentine ordinea de inregistrare

**LinkedHashSet** accepta 1 element null, mentinea odinea de inregistrare

**TreeSet** nu permite null valori si mentinea oridinea lor naturala adica asscendent

# MAP

**Hash map -**lafel ca hash set doar ca este key si value, is permite 1 null ca key

**LInkedHashMap** ca si linked hash set dor ca permite la key 1 valoare null

**TreeMap** ordoneaza in ordinea naturala si nu permite null key si nici value

**HashTable** implementare veche si este synchronized el nu permite null key si valoare

**De invatat:**

1. Red-Black Tre algoritm de sortare folosit in TreeSet
2. hashCode()
3. Explain the concept of WeakHashMap.

# Selenium

3 tipes of wait

* Implicit - **Definition**: It sets a default wait time for all elements before throwing a NoSuchElementException if they are not found immediately.

 **Usage**: Applied globally and applies to every findElement and findElements call

* explicit  **Definition**: It waits for a specific condition to be met before proceeding.

 **Usage**: Used for individual elements, making it more flexible than implicit wait.

 **Limitation**: Must be used every time synchronization is needed.

to specify conditions like visibility, clickability, presence

* fluent

 **Definition**: It is a type of explicit wait that allows polling intervals and ignoring specific exceptions.

 **Usage**: Best used when dealing with dynamic elements that load at unpredictable times.

 **Limitation**: Requires more configuration compared to explicit wait.

 Waits up to 15 seconds.

 Checks for the element every 2 seconds.

# Spring

# @Controller

 Indică faptul că **această clasă va gestiona rutele HTTP (GET, POST etc.)**.

 Poate returna:

* un **view** (pentru aplicații web tradiționale cu pagini HTML),
* sau date (dacă este combinat cu @ResponseBody sau se folosește @RestController).

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.GetMapping;

@Controller

public class HelloController {

@GetMapping("/hello")

public String sayHello(Model model) {

model.addAttribute("message", "Salut din Spring!");

return "hello"; // trimite către o pagină hello.html

}

}**Dacă vrei să trimiți direct text (nu pagină HTML):**

**Atunci folosești @RestController în loc de @Controller:**

java

CopyEdit

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class HelloRestController {

@GetMapping("/api/hello")

public String helloApi() {

return "Salut din API!";

}

}

|  |  |
| --- | --- |
| @Controller | Returnează pagini (ex: HTML, JSP) |

|  |  |
| --- | --- |
| @RestController | Returnează direct date (ex: JSON, text) |

# Ce este Model?

Model este un **obiect oferit de Spring MVC** care îți permite să trimiți **date din controller către view-ul HTML** (de exemplu, un fișier .html sau .jsp).

**🧠 Cum funcționează?**

* Model este un **container pentru date**.
* Când apelezi model.addAttribute("cheie", valoare), adaugi o pereche cheie-valoare.
* Apoi, în pagina HTML (de obicei un template Thymeleaf), poți accesa acea valoare.

**🔧 Exemplu concret:**

**1. Controller-ul Java:**

java

CopyEdit

@GetMapping("/hello")

public String sayHello(Model model) {

model.addAttribute("message", "Salut din Spring!");

return "hello"; // trimite către hello.html

}

**2. View-ul hello.html (Thymeleaf):**

html

CopyEdit

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<title>Salut</title>

</head>

<body>

<h1 th:text="${message}">Text implicit</h1>

</body>

</html>

**✅ Ce se întâmplă:**

* Controller-ul pune "Salut din Spring!" în Model sub cheia "message".
* View-ul (hello.html) accesează valoarea prin ${message}.
* Pe pagină se va afișa:

nginx

CopyEdit

Salut din Spring!

# Pe scurt: Ce este Thymeleaf?

* Este un **template engine** care permite combinarea **codului HTML cu date din Java**.
* Este folosit mai ales în aplicații **Spring MVC** pentru a crea **interfețe web dinamice**.
* Alternativ la JSP (Java Server Pages), dar mai modern, mai ușor de folosit și compatibil cu HTML standard.

**✅ Ce poate face Thymeleaf?**

* Afișează date din Java (Model) în HTML.
* Permite condiții (if, else), bucle (for).
* Poate include fișiere HTML una în alta.
* Are suport pentru formulare și binding cu obiecte Java.

**🧩 Exemplu simplu:**

**1. Controller Java:**

java

CopyEdit

@Controller

public class HelloController {

@GetMapping("/hello")

public String hello(Model model) {

model.addAttribute("name", "Ana");

return "hello"; // trimite către hello.html

}

}

**2. Fișierul hello.html (Thymeleaf template):**

html

CopyEdit

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<title>Pagina Hello</title>

</head>

<body>

<h1>Salut, <span th:text="${name}">utilizator</span>!</h1>

</body>

</html>

**🔄 Ce se întâmplă:**

* Controller-ul trimite "Ana" prin Model.
* În HTML, th:text="${name}" înlocuiește textul cu valoarea "Ana".
* Rezultatul afișat în browser: **"Salut, Ana!"**

**💡 De ce folosim Thymeleaf?**

| **Avantaj** | **Descriere** |
| --- | --- |
| ✅ Ușor de integrat cu Spring Boot | Funcționează perfect cu @Controller și Model |
| ✅ HTML valid | Template-urile Thymeleaf pot fi deschise în browser direct |
| ✅ Sintaxă clară și prietenoasă | Ușor de citit și de învățat |
| ✅ Suport pentru formulare și binding | Poți lega direct câmpurile unui formular la obiecte Java |

**🏗️ Unde punem fișierele Thymeleaf?**

Fișierele .html pentru Thymeleaf se pun în:

css

CopyEdit

src/main/resources/templates/

# @PathVariable si @RequestParam

**1. @RequestParam – pentru parametri din query string:**

java

CopyEdit

@GetMapping("/produse")

public String cautaProdus(@RequestParam String categorie) {

return "Ai căutat produse din categoria: " + categorie;

}

**Cerere HTTP:**

bash

CopyEdit

GET http://localhost:8080/produse?categorie=electronice

**Răspuns:**

r

CopyEdit

Ai căutat produse din categoria: electronice

**🔹 2. @PathVariable – pentru parametri din path-ul URL-ului:**

java

CopyEdit

@GetMapping("/produse/{id}")

public String detaliiProdus(@PathVariable int id) {

return "Detalii pentru produsul cu ID-ul: " + id;

}

**Cerere HTTP:**

bash

CopyEdit

GET http://localhost:8080/produse/42

**Răspuns:**

css

CopyEdit

Detalii pentru produsul cu ID-ul: 42

**🔄 Diferență:**

| **Situație** | **Adnotare** | **Exemplu URL** |
| --- | --- | --- |
| Parametru opțional, filtrare, sortare | @RequestParam | /produse?categorie=fructe&pretMax=10 |
| Parametru fix din URL | @PathVariable | /produse/1001 |

# @RequestBody

| Anotare | Folosit pentru... |
| --- | --- |
| @PathVariable | date din URL |
| @RequestBody | date JSON trimise în corpul cererii |
| @RequestParam | query parameters (opționali sau de filtrare) |

@RestController

@RequestMapping("/produse")

public class ProdusController {

@PostMapping("/{id}/review")

public String adaugaReview(

@PathVariable int id,

@RequestBody ReviewDTO review

) {

return "Review adăugat pentru produsul " + id + ": '" +

review.getComentariu() + "' (Rating: " + review.getRating() + ")";

}

}

public class ReviewDTO {

private String comentariu;

private int rating;

// Getters și setters (sau folosește Lombok: @Data)

public String getComentariu() {

return comentariu;

}

public void setComentariu(String comentariu) {

this.comentariu = comentariu;

}

public int getRating() {

return rating;

}

public void setRating(int rating) {

this.rating = rating;

}

}

Json body

{

"comentariu": "Produs foarte bun!",

"rating": 5

}

Răspuns:

nginx

Copy

Edit

Review adăugat pentru produsul 42: 'Produs foarte bun!' (Rating: 5)

# @Autowired

**Ce clase pot fi injectate cu @Autowired?**

Clase marcate cu:

* @Component
* @Service
* @Repository
* @Controller

@Autowired este o adnotare în **Spring Framework** folosită pentru **injectarea automată a dependențelor**. Practic, îi spui Spring-ului:  
➡️ *"Aici am nevoie de o instanță a acestei clase — caut-o și dă-mi-o!"*

**🔹 De ce este utilă?**

Spring gestionează "componentele" aplicației tale (servicii, repository-uri, controllere etc.) și le poate lega automat între ele fără ca tu să creezi manual obiecte cu new.

**🔹 Exemplu simplu**

java

CopyEdit

@Service

public class SalutService {

public String mesaj() {

return "Salut din serviciu!";

}

}

java

CopyEdit

@RestController

public class SalutController {

@Autowired

private SalutService salutService;

@GetMapping("/salut")

public String salut() {

return salutService.mesaj();

}

}

🔍 **Ce face @Autowired:**  
Spring detectează că SalutService este un @Service, îl creează automat, și îl "infuzează" în SalutController.

**🔹 Unde poți folosi @Autowired?**

1. Pe **câmpuri** (cel mai comun, dar nu recomandat pentru testare):

java

CopyEdit

@Autowired

private MyService service;

1. Pe **constructori** (recomandat):

java

CopyEdit

private final MyService service;

@Autowired

public MyController(MyService service) {

this.service = service;

}

1. Pe **setteri**:

java

CopyEdit

private MyService service;

@Autowired

public void setMyService(MyService service) {

this.service = service;

}

**🔹 Bună practică:**

* Pentru **testabilitate și claritate**, e recomandat să folosești **constructor injection** în loc de field injection.

**Deci ce face @Autowired:**

✔️ Caută o clasă MyService marcată ca bean (de ex. @Service)  
✔️ Creează automat un singur obiect ("singleton") din acea clasă  
✔️ Îl inserează acolo unde tu ai scris @Autowired — fără să mai scrii new  
✔️ Tu scrii doar declarația, nu și inițializarea

**Scenariu: Două implementări ale aceleiași interfețe**

Să presupunem că avem o interfață MesajService și două clase care o implementează:

🔹 Interfața:

java

CopyEdit

public interface MesajService {

String trimite();

}

🔹 Două implementări:

java

CopyEdit

@Service

public class EmailService implements MesajService {

@Override

public String trimite() {

return "Mesaj trimis prin EMAIL";

}

}

java

CopyEdit

@Service

public class SmsService implements MesajService {

@Override

public String trimite() {

return "Mesaj trimis prin SMS";

}

}

❗ Problema:

Dacă încerci asta:

java

CopyEdit

@Autowired

private MesajService mesajService;

➡️ Spring NU ȘTIE pe care dintre cele două să o injecteze  
🔴 Vei primi o eroare: NoUniqueBeanDefinitionException

✅ Soluții:

✅ 1. Folosește @Qualifier pentru a specifica exact care implementare vrei:

java

CopyEdit

@Autowired

@Qualifier("emailService") // numele bean-ului (implicit: numele clasei cu prima literă mică)

private MesajService mesajService;

În acest exemplu, emailService vine de la EmailService.

✅ 2. Sau definește un bean preferat cu @Primary:

java

CopyEdit

@Service

@Primary

public class SmsService implements MesajService {

@Override

public String trimite() {

return "Mesaj trimis prin SMS";

}

}

➡️ Acum, dacă ai:

java

CopyEdit

@Autowired

private MesajService mesajService;

➡️ Spring va alege automat SmsService.

✅ 3. Injectează toate implementările ca listă (opțional):

java

CopyEdit

@Autowired

private List<MesajService> toateServiciile;

➡️ Poți itera prin ele sau selecta dinamically.

✅ Extra: cu @Qualifier + constructor injection (cea mai bună practică)

java

CopyEdit

@RestController

public class MesajController {

private final MesajService mesajService;

public MesajController(@Qualifier("smsService") MesajService mesajService) {

this.mesajService = mesajService;

}

@GetMapping("/trimite")

public String trimite() {

return mesajService.trimite();

}

}

**🔚 Concluzie:**

| **Caz** | **Soluție** |
| --- | --- |
| **Mai multe implementări** | **@Qualifier sau @Primary** |
| **Ai nevoie de toate** | **List<Interface>** |
| **Vrei flexibilitate + testare ușoară** | **Constructor injection + @Qualifier** |

# @Bean

**CE ESTE UN BEAN?**

**Un *bean* este un obiect controlat de Spring.**  
Spring îl creează, îl configurează și îl dă mai departe în aplicație când ai nevoie de el.

**🔄 Gândește-te la Spring ca la un „șef de obiecte”:**

Tu nu mai faci:

java

CopyEdit

MyService service = new MyService();

Ci Spring zice:

"*Liniște, creez eu obiectul și ți-l dau deja făcut.*"

**✅ CUM DEFINESC UN BEAN?**

**🔸 Variante uzuale:**

**1. Cu adnotări (@Component, @Service, @Repository, @Controller)**

java

CopyEdit

@Component

public class SalutService {

public String spuneSalut() {

return "Salut!";

}

}

🔹 Acum SalutService este un *bean* Spring. Poți să-l folosești oriunde cu @Autowired.

**2. Sau manual, într-o clasă de configurare:**

java

CopyEdit

@Configuration

public class AppConfig {

@Bean

public SalutService salutService() {

return new SalutService();

}

}

🔹 Acesta este un *bean* declarat manual, dar Spring tot îl gestionează.

**🔍 EXEMPLU CLAR DE UTILIZARE**

**1. Avem clasa cu @Component:**

java

CopyEdit

@Component

public class Calculator {

public int aduna(int a, int b) {

return a + b;

}

}

**2. O folosim în controller:**

java

CopyEdit

@RestController

public class CalculatorController {

@Autowired

private Calculator calculator;

@GetMapping("/aduna")

public int aduna(@RequestParam int a, @RequestParam int b) {

return calculator.aduna(a, b);

}

}

➡️ Când accesezi /aduna?a=2&b=3, răspunsul va fi 5.

🧠 Tu NU ai scris new Calculator(). Spring a creat obiectul "bean" Calculator și ți l-a dat automat.

**📌 PE SCURT**

| **Termen** | **Ce înseamnă** |
| --- | --- |
| **Bean** | Un obiect creat și gestionat de Spring |
| **@Component** | Marcăm o clasă ca fiind un *bean* Spring |
| **@Bean** | Declarăm un *bean* manual într-o config |
| **@Autowired** | Cerem un bean să fie injectat într-o clasă |

# **@Component,**

# **@Service,**

# **@Repository,**

# **@Controller**

Toate acestea **marchează o clasă ca fiind un bean Spring**. Diferența dintre ele este mai ales de **semnificație (semantică)** și **ce comportamente speciale adaugă**.

**🔸 1. @Component**

🔹 Este cea mai **generală adnotare**. Orice clasă marcată cu @Component devine un bean Spring.

java

CopyEdit

@Component

public class Motor {

public void porneste() {

System.out.println("Motor pornit");

}

}

✅ E folosit când nu există un rol specific (ex: service, repo etc.)

**🔸 2. @Service**

🔹 Este un tip special de @Component, folosit pentru **logica de business**.

java

CopyEdit

@Service

public class PlatiService {

public void proceseaza() {

System.out.println("Plata procesată");

}

}

✅ Semnifică: "*Aceasta este o clasă de serviciu (business logic)*"  
✅ Spring nu face nimic diferit tehnic față de @Component, dar:

* E mai clar pentru alți programatori
* Poate primi comportamente extra în viitor (ex: AOP)

**🔸 3. @Repository**

🔹 Tot un @Component, dar folosit pentru **interacțiunea cu baza de date**.

java

CopyEdit

@Repository

public class ProdusRepository {

public void salveaza() {

System.out.println("Produs salvat în DB");

}

}

✅ În plus:

* Spring **interceptează erorile de DB** (ex: SQLException) și le transformă în excepții Spring (DataAccessException)

**🔸 4. @Controller**

🔹 E folosit pentru clasele care **gestionează request-uri web (HTTP)**.

java

CopyEdit

@Controller

public class PaginaController {

@GetMapping("/acasa")

public String acasa() {

return "index"; // pagina HTML din templates

}

}

✅ Este de fapt un @Component + suport pentru MVC (Model-View-Controller)

**🔸 BONUS: @RestController**

java

CopyEdit

@RestController

public class SalutController {

@GetMapping("/salut")

public String salut() {

return "Salut!";

}

}

✅ Este echivalent cu:

java

CopyEdit

@Controller

@ResponseBody

Adică trimite răspunsul direct ca text sau JSON, fără să încerce să returneze o pagină HTML.

**🧠 REZUMAT**

| **Adnotare** | **Rol principal** | **E un @Component?** | **Comportamente speciale** |
| --- | --- | --- | --- |
| @Component | General | ✅ | Nu |
| @Service | Logică de business | ✅ | Potențial pentru AOP |
| @Repository | Acces la baza de date | ✅ | Tratare automată excepții DB |
| @Controller | Controller pentru web (MVC) | ✅ | Routing + View rendering |
| @RestController | Controller pentru REST API | ✅ | @Controller + @ResponseBody |