AGENDA

1. Dependency Injection - Setter, Field and Constructor
2. Backward Compatibility
3. Type Erasures in Java - Data types during runtime.
4. Build all CRUD APIs for Product
5. Introduction to AOP
6. Exception Handling - @ControllerAdvice

1. Dependency Injection - Setter, Field and Constructor

Setter Injection -

Pros:

- Flexible: Allows for optional dependencies, as you don't have to set all properties.
- Readable: For beans with a lot of dependencies, setter methods can be more readable than a large constructor.

Cons:

- Mutable: Beans' properties can be changed after initialization, potentially leading to issues.
- Not Fail-Fast: The container won't fail at startup if a necessary dependency is not set; the error will be discovered at runtime when the setter is accessed.

```
Class ProductController {

        private ProductService   productService;
        private CategoryService  categoryService;
        ,
        ,
        .
        ,
@Autowired
public void  setProductService(ProductService productService){
            this.productService = productService;
 }
@Autowired
public void  setCategoryService(CategoryService categoryService){
            this.categoryService = categoryService
}
```

* fail fast ⇒ should fail as soon as possible,
  better fail at compile time than runtime

Constructor Injection:

Pros: • Easy to test

- Immutable: Once the dependencies are set, they can't be changed.
- Fail-Fast: Missing dependencies will cause the container to throw an exception at startup, not later.
- Clear Dependencies: It's straightforward to see all required dependencies of a bean just by looking at its constructor.

Cons:

- Verbose: Can become verbose when a class has many dependencies.

```
public class ProductController {
    private ProductService productService;
    private CategoryService categoryService;

    @Autowired
    public ProductController (Prod.Service ps, CatService cs) {
            this.ps = ps
            this. cs = cs;

    }
```

* Highly recommended to be used in production level projects.

Field Injection:

Pros:

- Concise: Eliminates the need for setter methods or constructors, leading to shorter code.

Cons:

- Not Testable: Field injection makes unit testing harder since you can't inject mock dependencies outside of the Spring context. This is one of the primary reasons it's discouraged.
- Inflexible: Can't have optional dependencies; every @Autowired field expects a bean to be available.

```
class ProductController {

        @Autowired
        private ProductService productService;
        ,
        |
        |
}
```

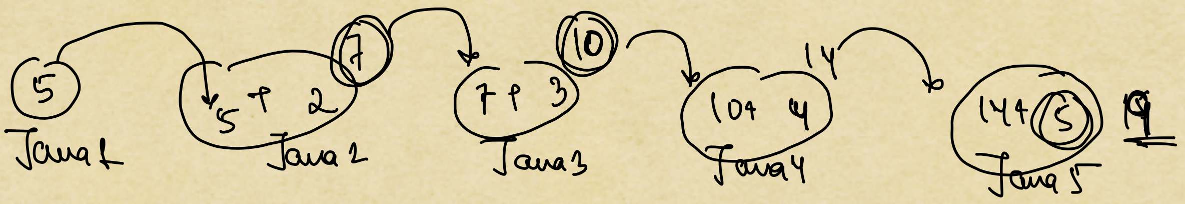
* NOT recommended

⇒ **Backward Compatibility:-**

* When adding new features | capabilities in a new version it should not lead to break down of existing features | capabilities.

$$\begin{tikzcd}
\text{Java 8} \arrow[r] & \text{Java 17} \\
x &
\end{tikzcd}$$

$$\begin{tikzcd}
\text{Java 1} \arrow[r] \arrow[r, bend left] & \text{Java 5} \\
x &
\end{tikzcd}$$



Java 1 ⇒ List, ArrayList... —

↓

Java 5 ⇒ Generics

↓

type erasure

↓

any internal type (generic) will be erased at runtime.

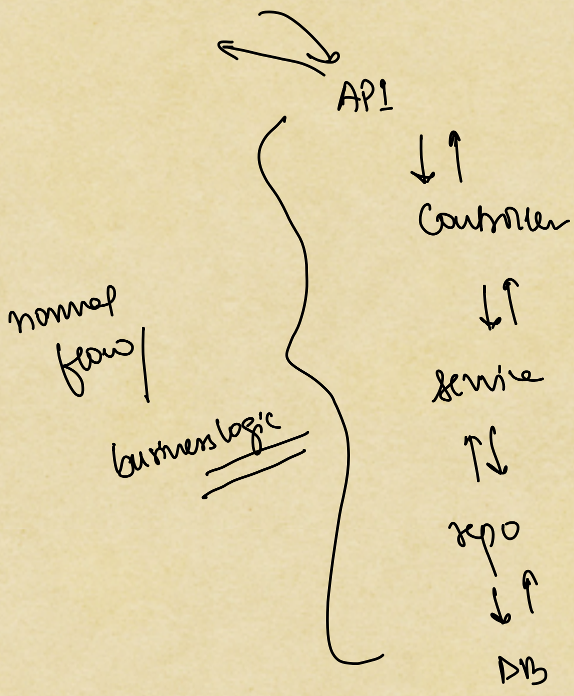
```
class A<E>{
    E e;
}
```

A<Int>

A<Boolean>

AOP :

Aspect-Oriented Programming (AOP) is a programming paradigm that focuses on the separation of cross-cutting concerns in a software application. Cross-cutting concerns are aspects of a program that affect multiple modules and are often difficult to modularize using traditional Object-Oriented Programming (OOP) techniques. Examples of cross-cutting concerns include logging, transaction management, security, and performance monitoring.



* track perf. of DB Queries
* error logging
* auth logging
* exception handling

⇒ @ControllerAdvice