Spring

Objectives

- Explain the advantages of the Spring Framework
- List the Spring Framework components
- Write/Modify an Application Based on Spring Core
- Describe Dependency Injection
- Identify the two distinct Spring containers:
  - BeanFactory
  - ApplicationContext

Spring Framework Background

- Simplified development of enterprise applications in Java technologies
- Started around 2002-2003 by Rod Johnson
- Open source application framework for Java platform
- Layered architecture; allows selection of components based on requirements
- Easy way to configure and resolve dependencies using Inversion of Control (IoC)

Problems with Traditional Approach

- *Most* JavaEE applications are complex and require a lot of effort to develop.
- Specific causes of complexity and other problems in JavaEE applications:
  - Contain excessive amounts of ‘plumbing’ code
  - Difficult to unit test
  - Certain JavaEE technologies have failed in performance, for example, EJB 2.x entity beans

Goals of Spring Framework

Reduced glue code/plumbing work:

- Dependencies described in separate file (xml), rather than mixing with business logic code itself, for better control over application
- Dependencies better managed

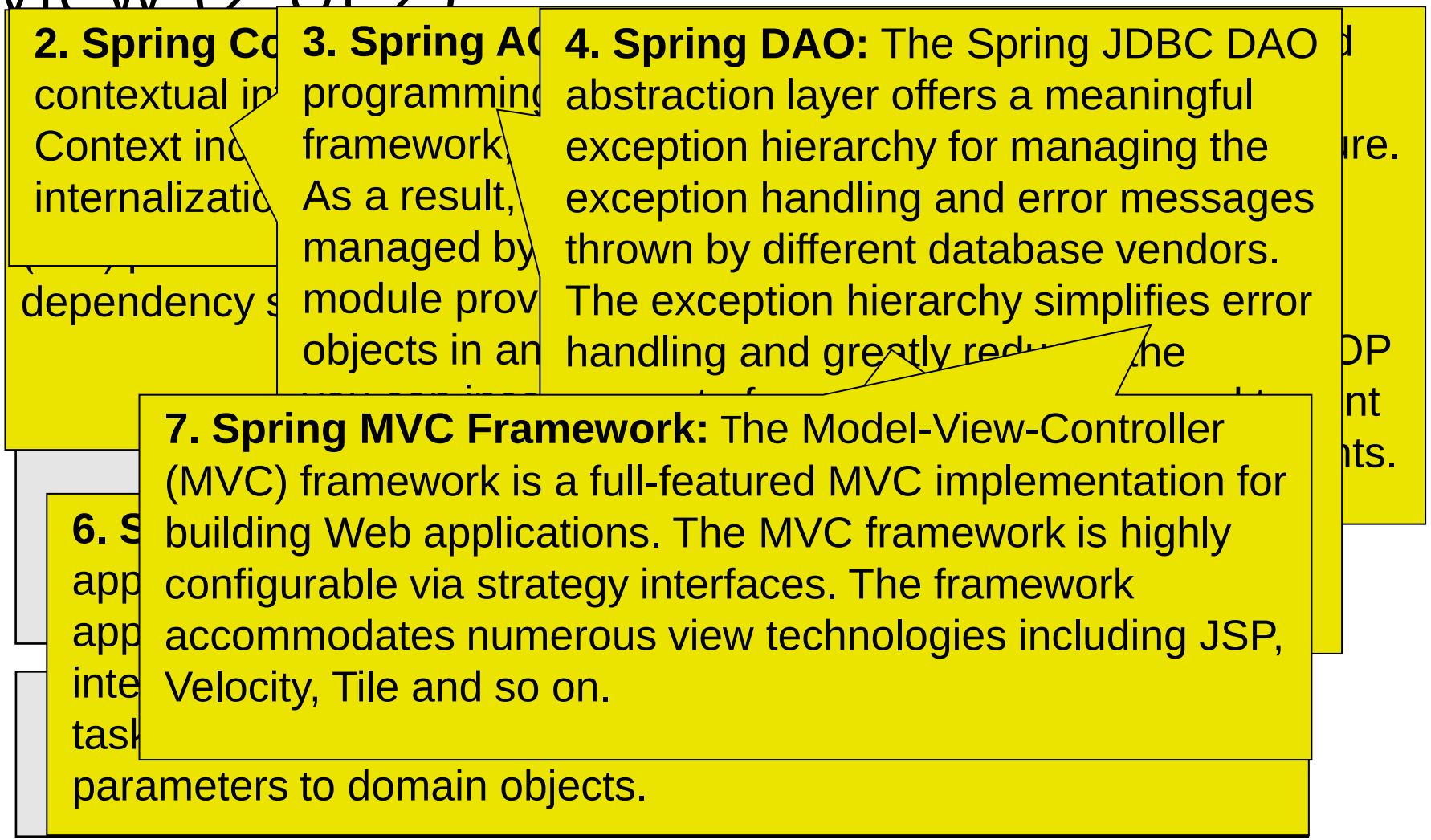
Flexibility:

- Programmers choose modules to suit their application
- Offers integration points with several other frameworks

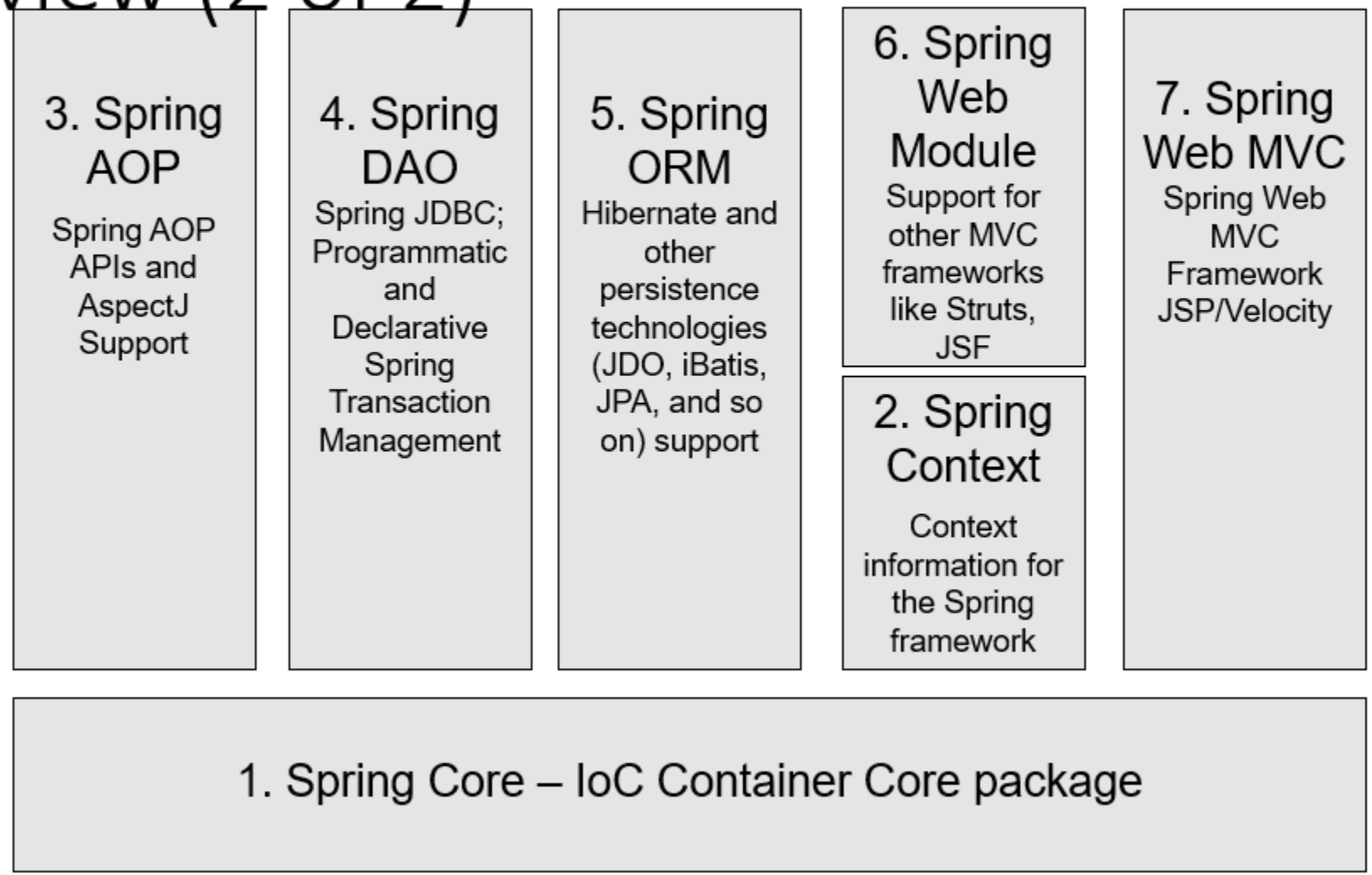
Spring Framework Components
Overview (1 of 2)

- Spring framework consists of several components/ modules.
- Each module has a defined set of functionality.
- Each module can be used independently.
- Spring provides integration points for every module to work with other frameworks.

Spring Framework Components Overview (2 of 2)

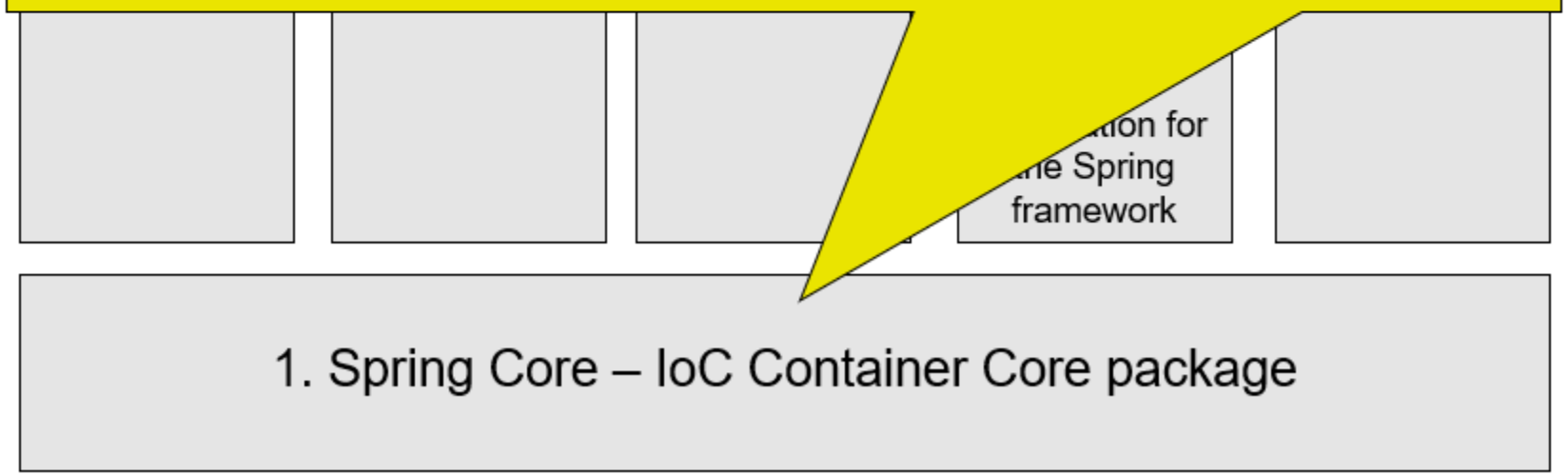


Spring Framework Components Overview (2 of 2)



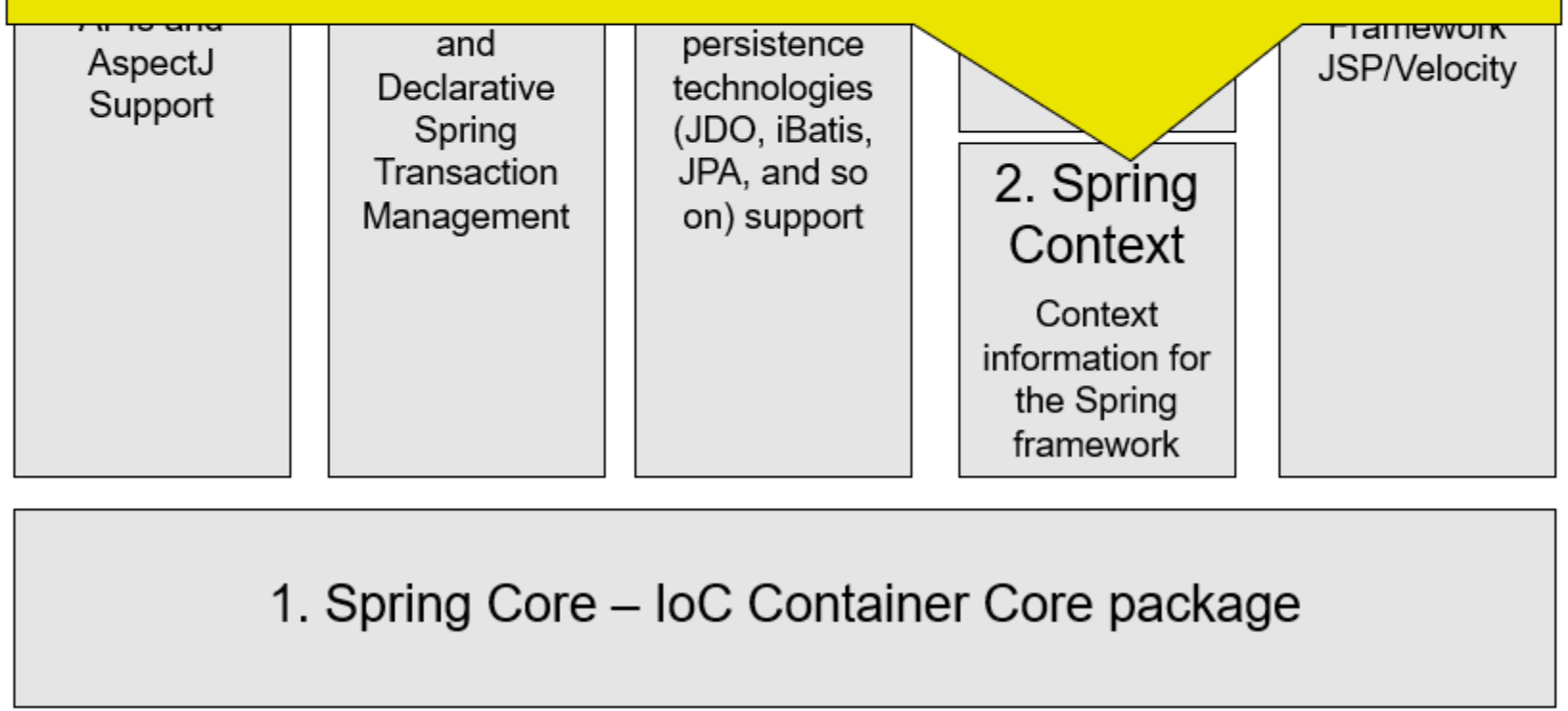
Spring Framework Components Overview (2 of 2)

1. Spring Core: The core container provides the essential functionality of the Spring framework. A primary component of the core container is the BeanFactory, an implementation of the Factory pattern. The BeanFactory applies the Inversion of Control (IoC) pattern to separate an application's configuration and dependency specification from the actual application code.

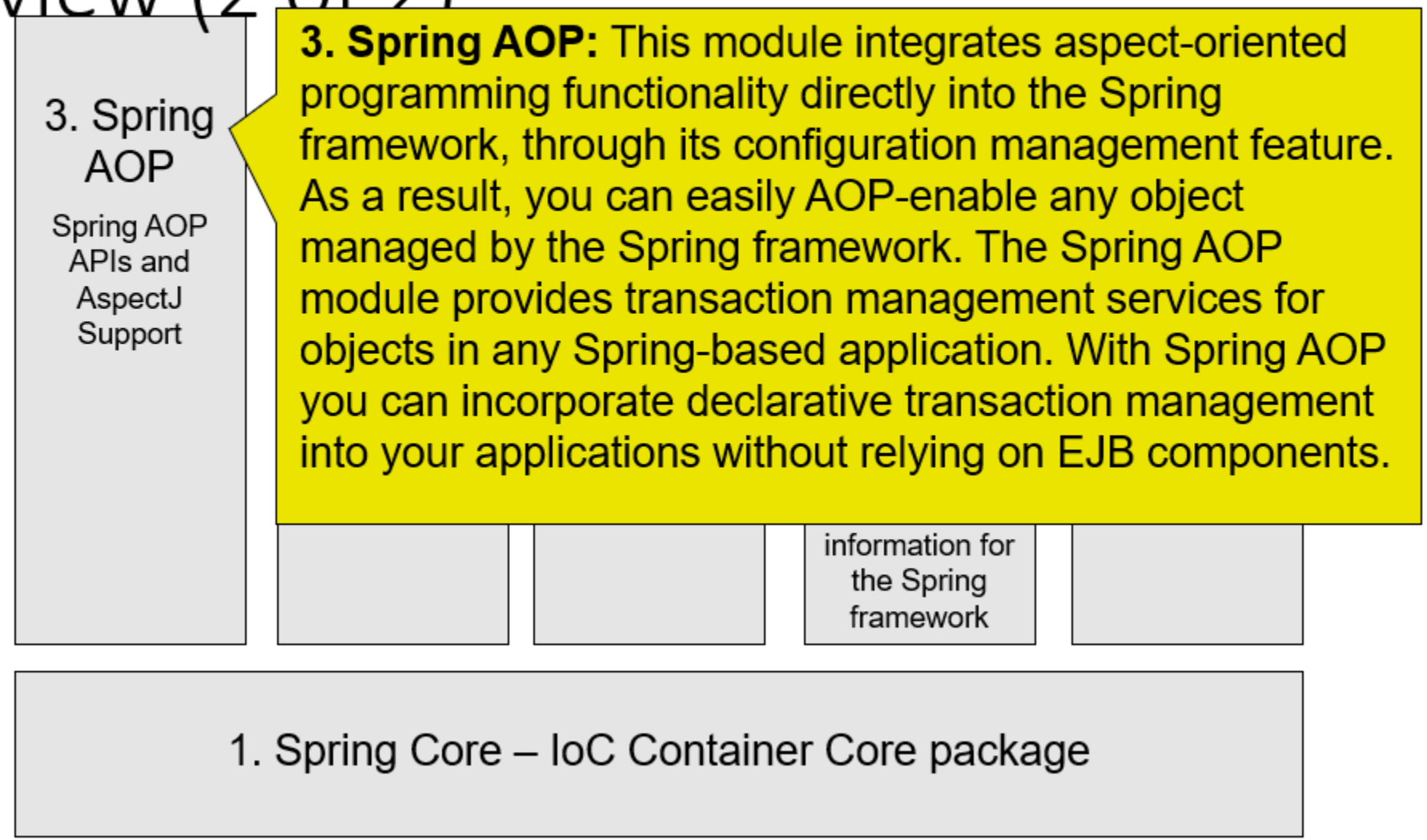


Spring Framework Components Overview (2 of 2)

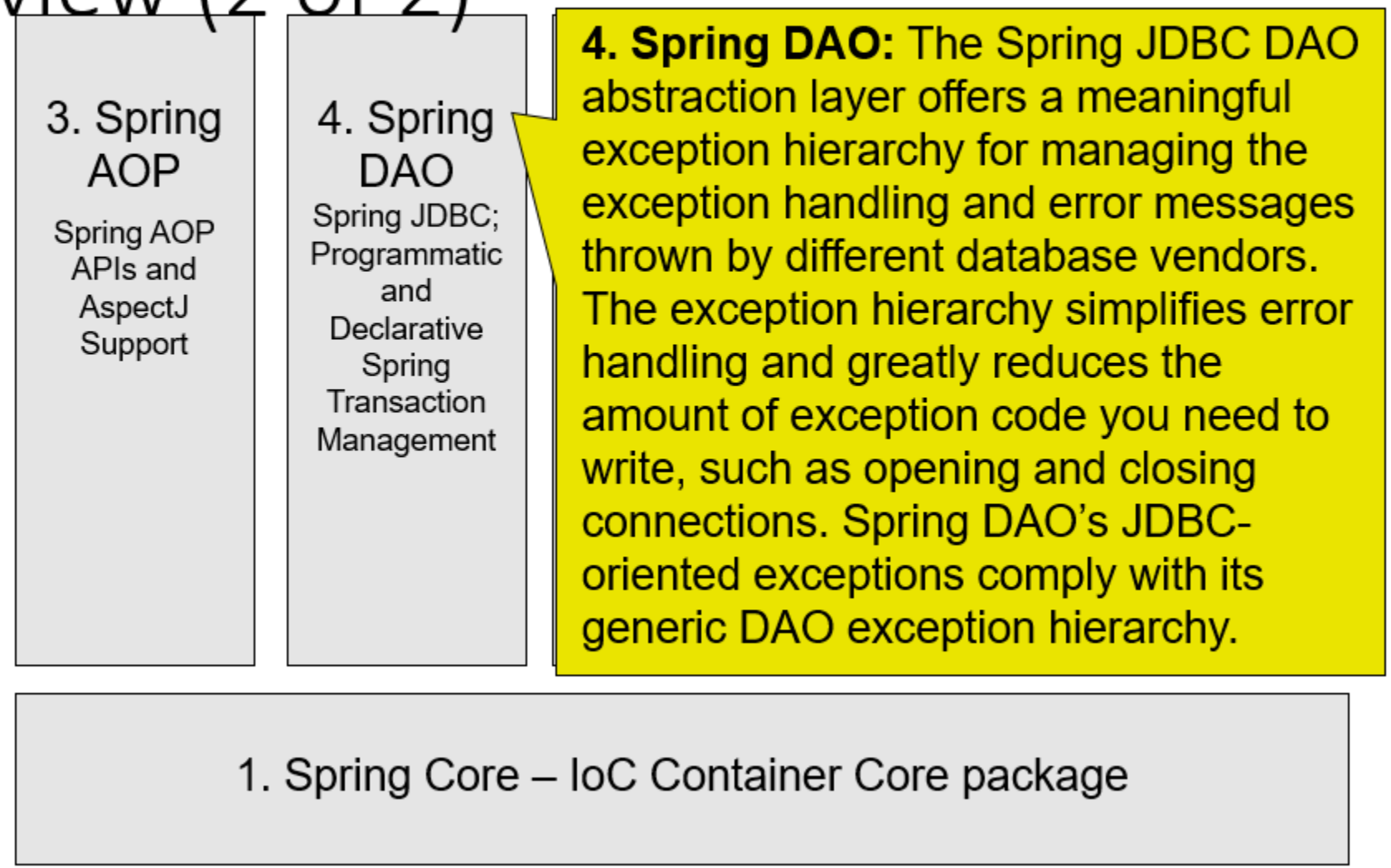
2. Spring Context: It is a configuration file that provides contextual information to the Spring framework. The Spring Context includes enterprise services such as JNDI, EJB, e-mail, internalization, validation and scheduling functionalities.



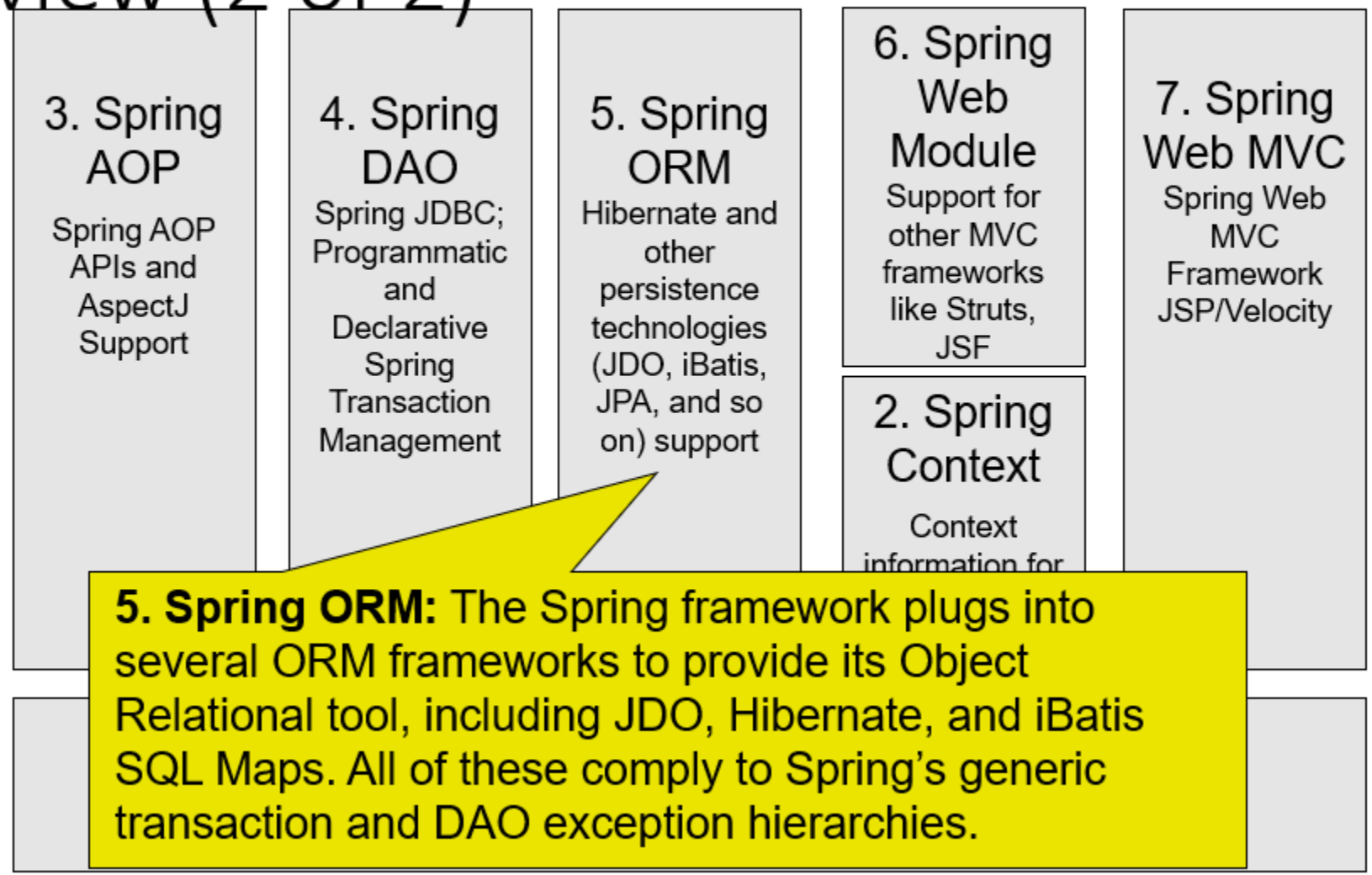
Spring Framework Components Overview (2 of 2)



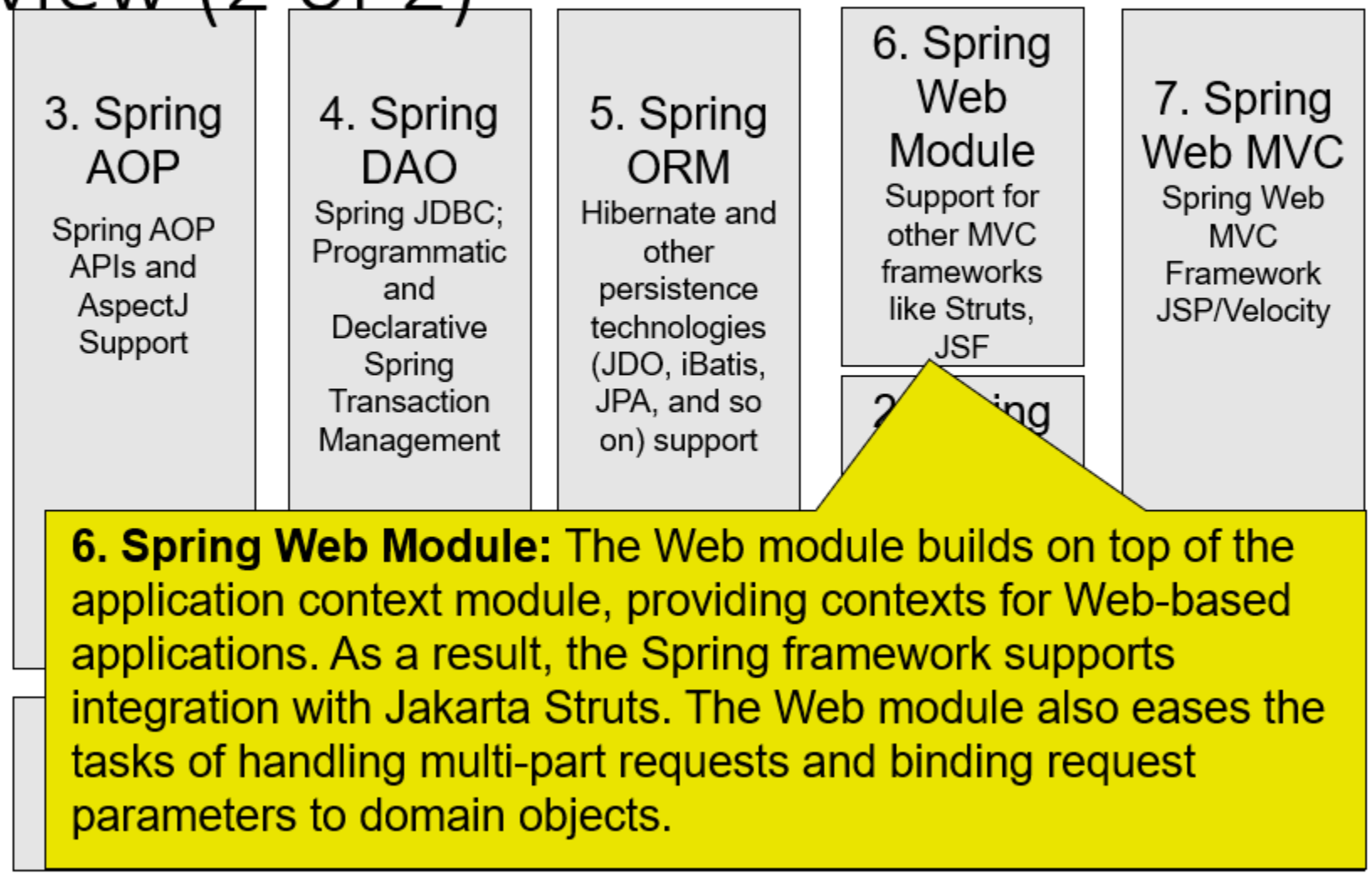
Spring Framework Components Overview (2 of 2)



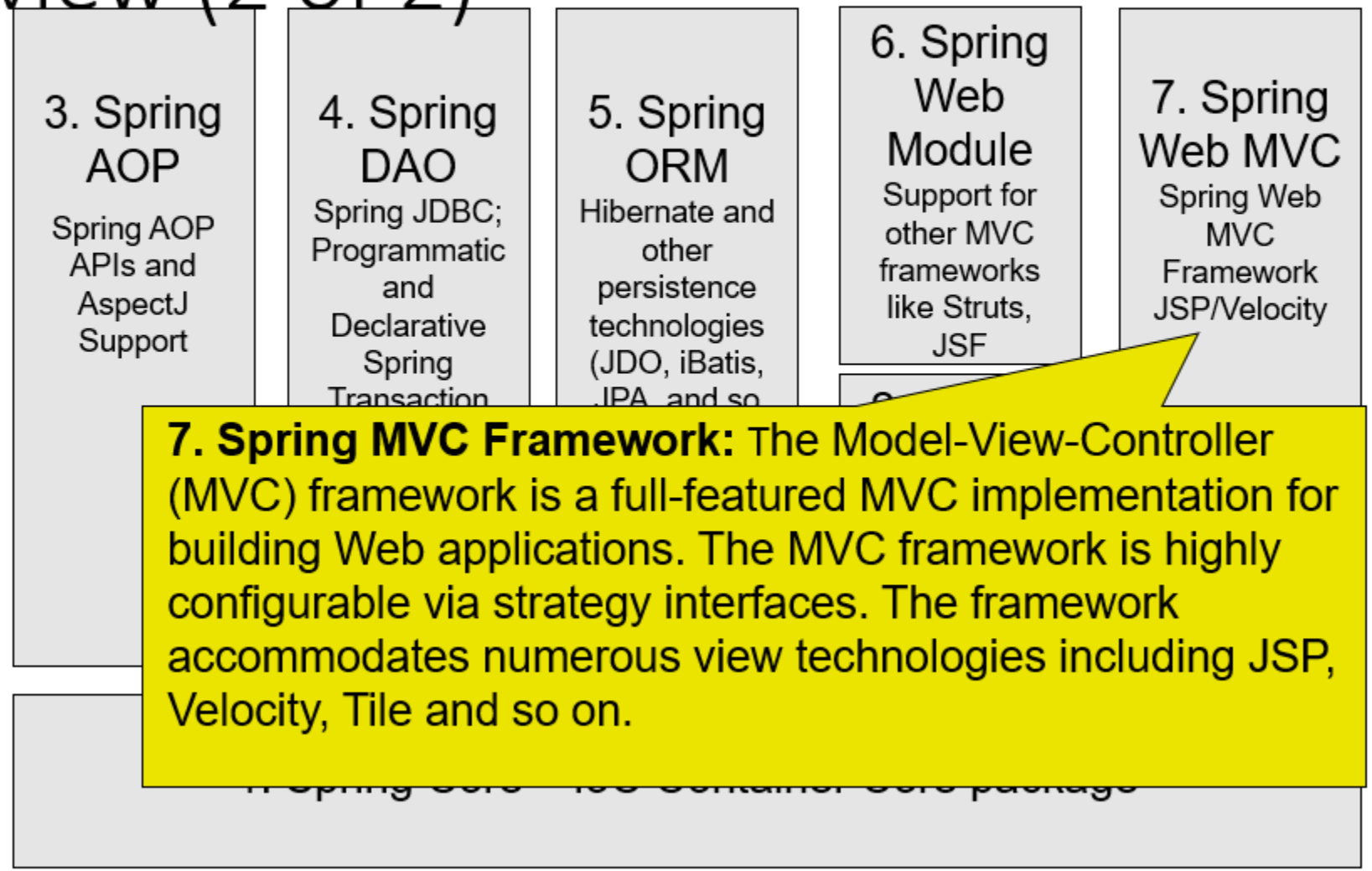
Spring Framework Components Overview (2 of 2)



Spring Framework Components Overview (2 of 2)



Spring Framework Components Overview (2 of 2)



Spring Framework Components Spring Core (1 of 2)

- All Spring modules rely on core components.
- Spring Core is also referred to as an IoC container.
  - Supports Dependency Injection into Spring components through Inversion of Control mechanism
  - Provides decoupling of configuration and dependency specifics from the actual program logic

Spring Framework Components Spring Core (2 of 2)

- Spring Core supports creation of and management of objects and other common applications services.
- Main packages include:
  - Core package: *BeanFactory*
  - Provides the basic functionality of creating beans
  - Context package: *ApplicationContext*
  - Superset of BeanFactory
  - More suitable for JavaEE applications

Spring Core Containers Overview

Spring's Container uses IoC to manage components of the application.

Spring has two distinct containers

- **Bean Factories:** (org.springframework.beans.factory.BeanFactory), provides support for Dependency Injection
- **Application contexts**: (org.springframework.context.ApplicationContext) provides application framework services

Spring Core Containers
Dependency Injection

- Java classes should be as independent as possible from each other.
- To decouple classes from one another, dependencies should be injected through:
  - Constructors
  - Setters
- Spring Framework injects these dependencies via their containers.
- A class should not configure itself, IoC uses dependency injection to:
  - Configure a class correctly from outside the class
  - Wire services or components

Spring Core Containers
Configuring Beans (1 of 2)

- Piecing together all beans in the Spring Container is called **wiring**.
- Wiring can be done through xml.
- Various BeanFactories and ApplicationContext objects that support wiring are:
  - XmlBeanFactory
  - ClassPathXmlApplicationContext
  - FileSystemXmlApplicationContext
  - XmlWebApplicationContext

Spring Core Containers
Configuring Beans (2 of 2)

- The beans are listed in the configuration file so that they can later be referred to by application programs.

Example:

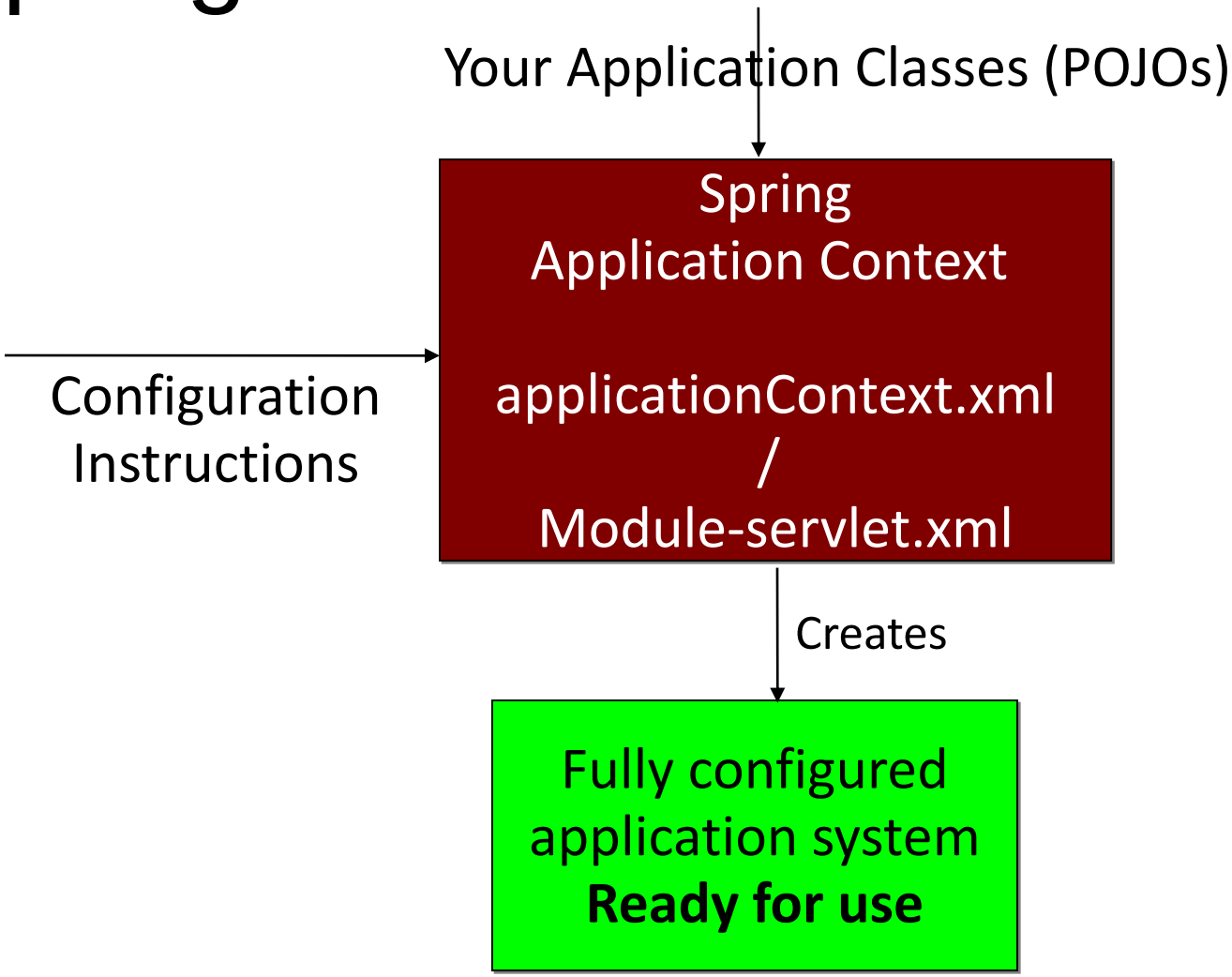
```
<? xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN// EN"
"http://www.springframework.org/dtd/spring-beans.dtd"?>
<beans>
<bean id="event" class="com.amrita.sample.Event"/>
</beans>
```

Spring Core Annotations

Objectives

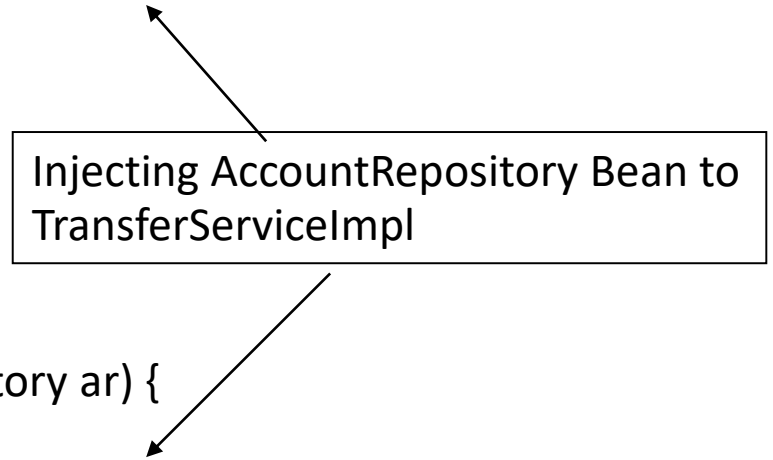
- Spring annotations
- Annotation Configuration
  - @Autowired
  - @Component
  - @Qualiifier

How Spring works



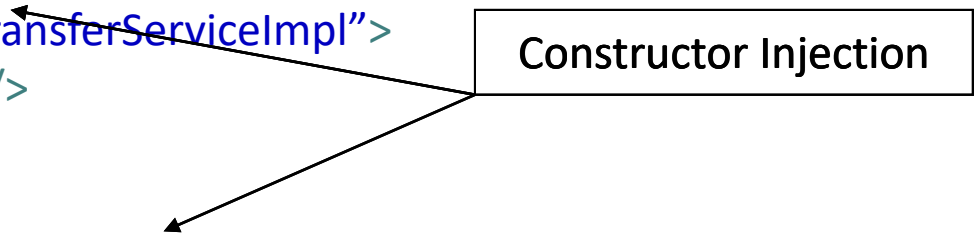
Bean Injection

```
public class TransferServiceImpl implements TransferService {

   // Constructor Injection
     public TransferServiceImpl(AccountRepository ar) {
              this.accountRepository = ar;
     }
   …

// OR – Setter Injection

AccountRepository accountRepository;

       public setAccountRepository (AccountRepository ar) {
             this.accountRepository = ar;
       }

}
```



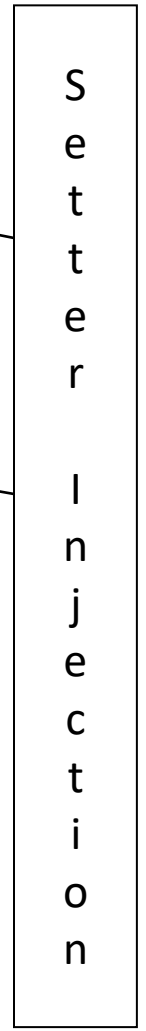
Constructor Injection – XML Configuration

```
<beans>

   <bean id="transferService" class="app.impl.TransferServiceImpl">
      <constructor-arg ref="accountRepository" />
   </bean>

   <bean id="accountRepository" class="app.impl.JdbcAccountRepository">
      <constructor-arg ref="dataSource" />
   </bean>

   <bean id="dataSource" class="com.mysql.jdbc.Driver">
      <property name="URL" value="jdbc:mysql://localhost:3306/codingtondb" />
      <property name="user" value="root" />
      <property name="password" value="abcd1234" />
   </bean>

</beans>
```



Setter Injection – XML Configuration

```
<beans>
  <bean id="transferService" class="app.impl.TransferServiceImpl">
    <property name="accountRepository" ref="accountRepository" />
  </bean>

  <bean id="accountRepository" class="app.impl.JdbcAccountRepository">
    <property name="dataSource" ref="dataSource" />
  </bean>

  <bean id="dataSource" class="com.mysql.jdbc.Driver">
    <property name="URL" value="jdbc:mysql://localhost:3306/codingtondb" />
    <property name="user" value="root" />
    <property name="password" value="abcd1234" />
  </bean>
</beans>
```



- Place holder (Setter – Getter methods) for injecting bean in parent class.

@Autowired

```
public class TransferServiceImpl implements TransferService {
    @Autowired
    public TransferServiceImpl(AccountRepository ar) {
        this.accountRepository = ar;
    }
    …
}
```

```
public class JdbcAccountRepository implements AccountRepository {
    @Autowired
    public JdbcAccountRepository(DataSource ds) {
        this.dataSource = ds;
    }
    …
}
```

@Autowired – XML Configuration

```
<beans>

  <bean id="transferService" class="app.impl.TransferServiceImpl" />

  <bean id="accountRepository" class="app.impl.JdbcAccountRepository" />

  <bean id="dataSource" class="com.mysql.jdbc.Driver">
    <property name="URL" value="jdbc:mysql://localhost:3306/codingtondb" />
    <property name="user" value="root" />
    <property name="password" value="abcd1234" />
  </bean>

<context:annotation-config/>

</beans>
```

No need to specify
constructor-args / Setter reference

looks for annotations on beans
only in the same application context
where it is defined

@Autowired

- @Autowired annotation can be applied on setter methods, constructors and fields.
- Autowired indicating “required dependencies”.
- Autowire **will fail** if no matching bean is available in the context.
- @Autowired(required=false) – indicating not a mandatory dependency. Defaults to true. Autowire **will not fail** if no matching bean is available in the context.

```
@Autowired(required=false)
private AccountRepository accountRepository;
```

@Component

- Indicates that the annotated class is a "component"
- Both identify POJOs as Spring Beans
- Removes the need to specify *almost anything* in XML
- Optionally pass it a String, which will be the bean name
- Default bean name is de-capitalized non-qualified name

```
@Component
public class TransferServiceImpl implements TransferService
   public TransferServiceImpl(AccountRepository ar) {
      this.accountRepository = ar;
   }
   …
}
```

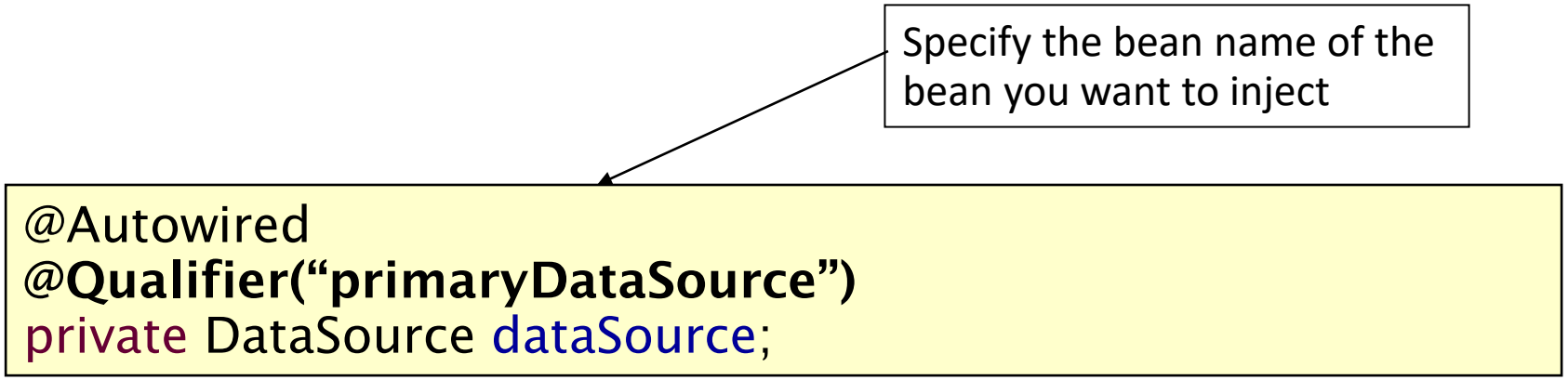
@Component

- @Component takes a String parameter that names the bean
- Arguably not a best practice to put bean names in your Java code
- **<context:component-scan** base-package="com.amrita.xx.xx.x" **/>** - required in configuration xml to enable annotation scan in mentioned package

```
@Component("myTransferService")
public class TransferServiceImpl implements TransferService
   public TransferServiceImpl(AccountRepository ar) {
      this.accountRepository = ar;
   } …
  }
```

@Qualifier

- To used on a field or parameter as a qualifier for a beans when autowiring
- Can be used in other annotations to that can be used as qulaifier
- Needed in case multiple instances of the same type exist, one of which needs to be autowired
- Using an @Qualifier annotation you can inject named beans



When to use What

- Start using annotations for small isolated parts of your application (Spring @MVC controllers)
- Annotations are spread across your code base
- XML is centralized in one (or a few) places
- XML for infrastructure and more 'static' beans
- Annotations for frequently changing beans

Develop Spring Application Using Java Base Configuration

Develop Spring Application Using Java Base Configuration

- Spring 3 onwards a spring application can be configured with almost no XML using pure java.
- Java base configuration allows moving bean definition and spring configuration out of XML file into a java classes.

Java Configuration Class

- @Configuration -The java base equivalent to <beans> in xml is a Java class annotated with @Configurations
- @ Bean -This annotation is used to define the beans.

Implementing
Bean Lifecycle Callbacks And Bean Scope

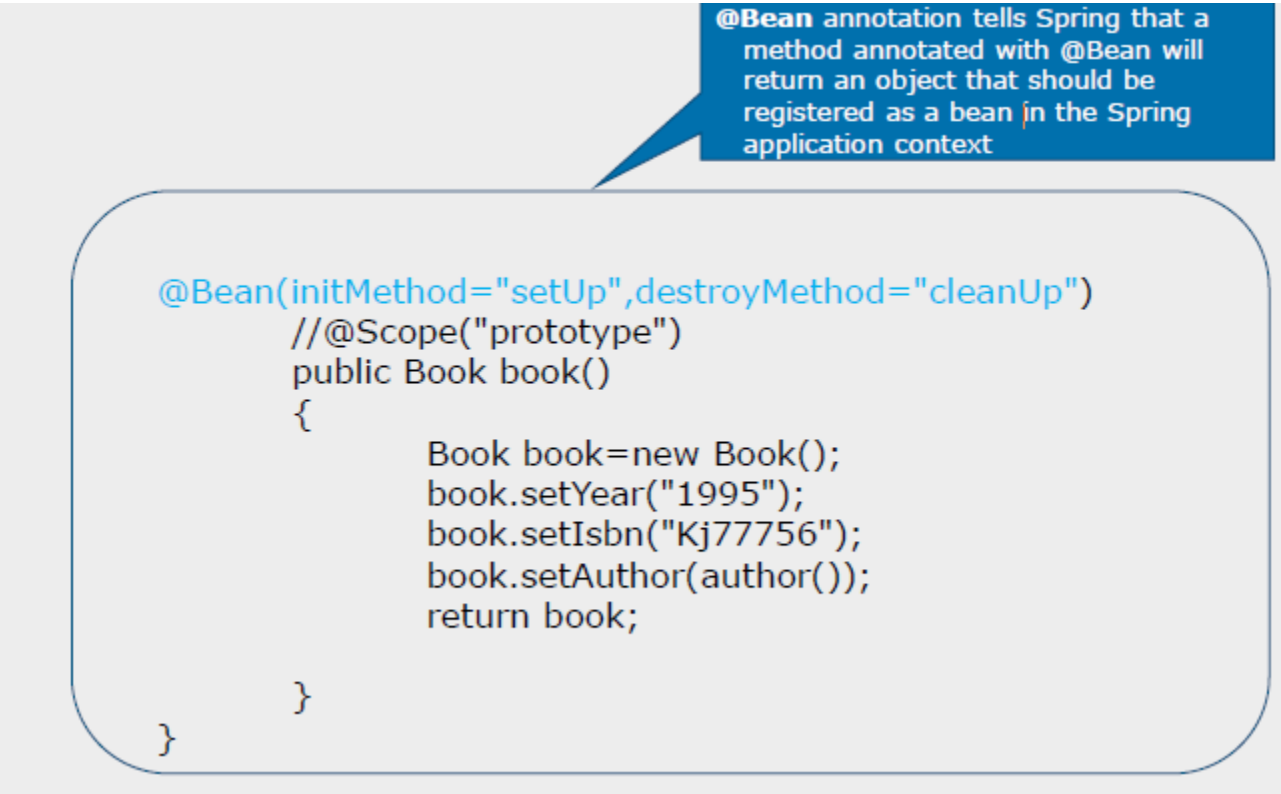
- @PostConstruct-This Annotation is used on a method that needs to be executed after dependency injection is done to perform any initialization.
- @PreDestroy-This Annotation is used on methods as a callback notification to signal that the instance is in the process of being removed by the container.
- @Scope -This annotation is used in java base configuration to define the scope of the bean

@Configuration

- Annotating a class with the **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions.

@Bean

- **@Bean** annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring



Example

- public class Author{
- private String authorName;
- private String address;
- public String getAuthorName() {return authorName;}
- public void setAuthorName(String authorName) {
- this.authorName= authorName; }
- public String getAddress() {return address;}
- public void setAddress(String address) {
- this.address= address;
- }

```
public Author(String authorName, String address)
{
super();
this.authorName= authorName;
this.address= address;
}
@Override
public String toString() {
return "Author [authorName=" + authorName+ ",
address=" + address+ "]";}
@PostConstruct
public void customAuthorInit()
{
System.out.println("Method customAuthorInit()
invoked...");        }
@PreDestroy
public void customAuthorDestroy()
{  System.out.println("Method
customAuthorDestroy() invoked...");  }}
```

Book.java

```
public class Book
{
private Author author;
private String isbn;
private String year;
@PostConstruct
public void customBookInit()
{    System.out.println("Method
customBookInit() invoked...");
}
@PreDestroy
public void customBookDestroy()
{
System.out.println("Method
customBookDestroy() invoked...");
}
```

```
public void setUp()throws Exception
{
System.out.println(" Initializing the Book Bean
with custom Config");
}
public void cleanUp()throws Exception
{
System.out.println(" Destroying the Book Bean
with custom
Config");
}
public Author getAuthor() {return author;}
public void setAuthor(Author author) {this.author=
author;}
public String getIsbn() {return isbn;}
public void setIsbn(String isbn) {this.isbn= isbn;}
public String getYear() {return year;}
public void setYear(String year) {this.year= year;}
@Override
public String toString()
{
return "Book [author=" + author + ", isbn=" +
isbn+ ", year=" + year+ "]";
}}
```

Registering Configuration Using AnnotationConfigApplicationContext -BookClient.java

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class BookClient{
public static void main(String[] args)
{
ApplicationContextctx= new  AnnotationConfigApplicationContext(MyBookConfig.class);
Book book1=(Book)ctx.getBean("book");
System.out.println(" Book HashCode: "+book1.hashCode());
System.out.println(" Book Info  : "+book1);
try
{
book1.cleanUp();
}
catch (Exception e) {
e.printStackTrace();  }}
```