Spring Annotations - Interview Revision Notes

🔑 Core Concepts

What is a Bean?

Bean = Object managed by Spring Container

- Spring creates, manages lifecycle, and injects dependencies
- Default scope: Singleton (one instance shared everywhere)
- Alternative to manual `new ObjectName()`

What is Classpath?

Classpath = List of locations where Java finds classes

```
POM Dependency → Maven Downloads JARs → JARs on Classpath → Spring Boot Auto-configuration
```

- Spring Boot scans classpath for marker classes to decide what to auto-configure
- Example: `spring-boot-starter-web` adds `DispatcherServlet.class` → Spring configures web MVC

🏗️ Evolution: Java → Spring → Spring Boot

Aspect	Pure Java	Spring Framework	Spring Boot
Object Creation	`new Service()`	`@Component` + XML/Java config	`@Component` + Auto-config
Configuration	Manual wiring	Manual `@Configuration`	Auto-configuration
Dependencies	Manual management	Manual dependency declaration	Starter dependencies
Server	External (Tomcat)	External	Embedded
Setup Time	Days	Hours	Minutes

Example Evolution:

```java
// Pure Java
EmailService emailService = new EmailService();
UserService userService = new UserService(emailService);

// Spring Framework
@Configuration
public class AppConfig {
    @Bean public EmailService emailService() { return new EmailService(); }
    @Bean public UserService userService() { return new UserService(emailService()); }
}

// Spring Boot
@Service public class EmailService { }
@Service public class UserService {
    @Autowired private EmailService emailService; // Auto-injected
}
```

📋 Core Stereotype Annotations

@Component, @Service, @Repository, @Controller

Annotation	Purpose	When to Use	Special Features
`@Component`	Generic Spring bean	Utility classes, when others don't fit	Base annotation
`@Service`	Business logic layer	Business operations, transactions	Often `@Transactional`
`@Repository`	Data access layer	Database operations	Exception translation
`@Controller`	Web request handler	HTTP requests → Java methods	Returns view names
`@RestController`	REST API endpoints	`@Controller` + `@ResponseBody`	Returns JSON/XML

```java
@Service
public class OrderService {
    @Autowired private OrderRepository repository;

    @Transactional
    public Order processOrder(Order order) { /* business logic */ }
}

@Repository
public class OrderRepository {
    // SQLException automatically translated to DataAccessException
}

@RestController
public class OrderController {
    @GetMapping("/orders/{id}")
    public ResponseEntity<Order> getOrder(@PathVariable Long id) { /* web logic */ }
}
```

⚙️ Configuration Annotations

@Configuration vs @Component

```java
// @Component: One class = One bean
@Component
public class EmailService { }

// @Configuration: One class = Multiple beans
@Configuration
public class AppConfig {
    @Bean public DataSource dataSource() { }
    @Bean public JdbcTemplate jdbcTemplate(DataSource ds) { }
}
```

@Value - Property Injection

```java
@Value("${app.name}")                    // From properties file
private String appName;

@Value("${app.timeout:30}")              // With default value
private int timeout;

@Value("#{systemProperties['user.dir']}") // SpEL expression
private String userDir;
```

@Profile - Environment-specific Beans

```java
@Configuration
@Profile("development")
public class DevConfig {
    @Bean public DataSource devDataSource() { /* H2 database */ }
}

@Configuration
@Profile("production")
public class ProdConfig {
    @Bean public DataSource prodDataSource() { /* MySQL database */ }
}
```

🎯 Advanced Annotations

@Primary and @Qualifier - Multiple Implementations

```java
public interface PaymentService { }

@Service
@Primary  // Default choice
public class CreditCardPaymentService implements PaymentService { }

@Service
@Qualifier("paypal")  // Specific identifier
public class PayPalPaymentService implements PaymentService { }

// Usage
@Autowired private PaymentService defaultPayment;  // Gets @Primary
@Autowired @Qualifier("paypal") private PaymentService paypalPayment;
```

@Conditional - Smart Bean Creation

```java
@Bean
@ConditionalOnProperty(name = "feature.enabled", havingValue = "true")
public FeatureService featureService() { }

@Bean
@ConditionalOnClass(RedisTemplate.class)  // Only if Redis on classpath
public CacheService redisCacheService() { }

@Bean
@ConditionalOnMissingBean(CacheService.class)  // Fallback
public CacheService defaultCacheService() { }
```

🔍 Component Scanning

Default Behavior

```java
@SpringBootApplication  // Scans current package + sub-packages only
public class Application { }
```

Custom Scanning - Why Needed?

```
Problem: Multi-module project
com.company.main.Application      ← @SpringBootApplication here
com.company.users.UserService     ← @Service here (NOT FOUND!)
com.company.orders.OrderService   ← @Service here (NOT FOUND!)
```

Solution: Custom @ComponentScan

```java
@SpringBootApplication
@ComponentScan(basePackages = {
    "com.company.users",    // Include user module
    "com.company.orders"    // Include order module
})
public class Application { }

// Advanced filtering
@ComponentScan(
    basePackages = "com.example",
    includeFilters = @Filter(type = FilterType.ANNOTATION, classes = Service.class),
    excludeFilters = @Filter(type = FilterType.REGEX, pattern = ".*Test.*")
)
```

🧪 Dependency Injection Patterns

Constructor Injection (Recommended)

```java
@Service
public class OrderService {
    private final PaymentService paymentService;

    // @Autowired optional for single constructor
    public OrderService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }
}
```

Field Injection (Avoid - Hard to Test)

```java
@Service
public class OrderService {
    @Autowired
    private PaymentService paymentService;  // Hard to mock in tests
}
```

Why Constructor Injection is Better:

- ✅ **Easy testing**: Simple to pass mock objects
- ✅ **Immutable**: Dependencies can't change after creation

- ✅ **Clear dependencies**: Constructor shows what's needed
- ✅ **Fail fast**: Missing dependencies cause immediate failure

---

🎭 Custom Annotations

Creating Custom Annotations

```java
// 1. Define annotation
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Auditable {
    String operation() default "";
    boolean logParams() default false;
}

// 2. Create aspect to handle it
@Aspect
@Component
public class AuditingAspect {
    @Around("@annotation(auditable)")
    public Object audit(ProceedingJoinPoint joinPoint, Auditable auditable) {
        System.out.println("Auditing: " + auditable.operation());
        return joinPoint.proceed();
    }
}

// 3. Use it
@Service
public class UserService {
    @Auditable(operation = "CREATE_USER", logParams = true)
    public User createUser(User user) { return userRepository.save(user); }
}
```

🎯 Common Interview Questions & Answers

Q1: "Difference between @Component, @Service, @Repository, @Controller?"

Answer: "All are functionally identical - they're all @Component underneath. The difference is semantic purpose and additional features:

- @Service: Business logic, often transactional
- @Repository: Data access, automatic exception translation
- @Controller: Web layer, handles HTTP requests

- @Component: Generic purpose, when others don't fit"

Q2: "Why avoid field injection?"

Answer: "Field injection makes testing difficult because you can't easily inject mock objects. Constructor injection is preferred because:

- Easy to pass mock dependencies in tests
- Clear visibility of dependencies
- Immutable objects
- Fail fast if dependencies missing"

Q3: "How does @Autowired work?"

Answer: "Spring scans for @Autowired annotations and automatically injects matching beans. It works by:

1. Finding beans of the required type
2. If multiple beans exist, uses @Primary or @Qualifier
3. If no beans found, throws exception (unless required=false)
4. Creates proxy objects when needed for aspects"

Q4: "Difference between @Configuration and @Component for beans?"

Answer: "@Component creates one bean per class. @Configuration allows creating multiple beans from one class using @Bean methods. Use @Configuration for:

- Complex bean setup logic
- Multiple related beans
- Third-party library configuration
- Conditional bean creation"

Q5: "How to handle multiple implementations?"

Answer: "Use @Primary for default choice and @Qualifier for specific selection:

```
java

@Service @Primary
public class DefaultPaymentService implements PaymentService { }

@Service @Qualifier("paypal")
public class PayPalPaymentService implements PaymentService { }

@Autowired private PaymentService default;  // Gets @Primary
@Autowired @Qualifier("paypal") private PaymentService paypal;
```"

Q6: "Explain component scanning process"
Answer: "Spring scans specified packages for classes annotated with @Component (an
1. Start from basePackages (default: @SpringBootApplication package)
2. Find all .class files
3. Apply include/exclude filters
4. Register matching classes as beans
5. Resolve dependencies between beans"

🔄 Bean Lifecycle

```java
Bean Creation → Dependency Injection → @PostConstruct → Ready for Use → @PreDestroy → 


java

@Service
public class DatabaseService {
    @PostConstruct  // Called after dependencies injected
    public void init() { /* setup database connection */ }

    @PreDestroy     // Called before bean destruction
    public void cleanup() { /* close database connection */ }
}
```

💡 Best Practices for Interviews

Do:

- ✅ Use constructor injection
- ✅ Prefer @Primary over @Qualifier when possible
- ✅ Use specific stereotypes (@Service vs @Component)

- ✅ Understand the difference between classpath and configuration
- ✅ Know when to use @Configuration vs @Component

Don't:

- ❌ Use field injection in production code
- ❌ Mix business logic in @Controller classes
- ❌ Create circular dependencies
- ❌ Overuse @Qualifier (redesign if too many needed)

Key Points to Emphasize:

1. **Spring's value**: Inversion of Control - "Don't call us, we'll call you"
2. **Auto-configuration**: Based on classpath detection
3. **Bean scope**: Default singleton, shared instances
4. **Testing**: Constructor injection makes mocking easy
5. **Separation of concerns**: Each layer has specific responsibility