```
class OrderService {
    private PaymentService paymentService = new PaymentService(); // tightly coupled
}
```

3 Major Problems with this code:

1. Tight coupling
2. Hard to test
3. Scattered Object Creation

Introduction To Dependency Injection

DI is a design pattern where an object receives its dependencies from the outside, instead of creating them itself.

```java
// BAD: Tight coupling
public class OrderService {
    private EmailService emailService = new EmailService();
}

// GOOD: Loose coupling via DI
public class OrderService {
    private final EmailService emailService;

    public OrderService(EmailService emailService) {
        this.emailService = emailService;
    }
}
```

Types of Dependency Injection

Type	Annotation	Recommended?
Constructor	@Autowired (optional)	YES – Best
Setter	@Autowired	OK for optional
Field	@Autowired	Avoid in production

How Spring Resolves Dependencies

1. By Type → EmailService
2. If multiple beans → use name or @Qualifier
3. If still ambiguous → use @Primary

```java
@Bean
@Qualifier("smtp")
public EmailService smtpEmailService() { ... }

@Bean
@Qualifier("sendgrid")
public EmailService sendgridEmailService() { ... }
```

```java
public OrderService(@Qualifier("sendgrid") EmailService emailService) { ... }
```

CODING SHUTTLE

Dependency Injection Misc.

Get all bean instances:

```
@Autowired
private Map<String, PaymentGateway> gateways;  // bean name → instance
```

Optional Beans

```
public OrderService(Optional<AnalyticsService> analytics) { ... }
```

```
@Autowired(required = false)
private BackupService backupService;  // null if not present
```