Dependency Injection & IoC

Inversion of Control, Dependency Injection, and constructor/setter/field injection in plain Java -- no Spring required.

Intermediate 45 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/dependency-injection

---

Dependency Injection & IoC

This lesson is the first Spring Boot topic on the site, but it doesn't touch Spring yet -- Dependency Injection (DI) and Inversion of Control (IoC) are two framework-independent design ideas that predate Spring by a long way. The goal is to first do, entirely by hand, in plain Java, what Spring's `@Autowired` does "magically" -- the next lesson (Spring IoC Container & Bean Lifecycle) covers the container that automates this by-hand work. We'll start from the tight coupling problem, compare all three of constructor/setter/field injection, and end with a short look at how Spring automates them.

What Are Dependency Injection and IoC?

At its simplest, Dependency Injection (DI) means an object receives the other objects it depends on from the outside, instead of creating them itself with `new`. Inversion of Control (IoC) is the more general idea behind it: "inverting control" -- normally a class manages its own dependencies and flow, but IoC hands that control to something outside the class (a hand-written "composition root," or a container like Spring). DI is the most common concrete way of implementing IoC:

```
// Without DI: OrderService decides and owns everything itself.
class OrderService {
    private final EmailSender sender = new EmailSender();
}

// With DI: OrderService only declares what it needs; someone else decides
// which EmailSender (or alternative) to hand it.
class OrderService {
    private final EmailSender sender;
    OrderService(EmailSender sender) { this.sender = sender; }
}
```

In the second version, `OrderService` has no idea where its `EmailSender` came from -- that's the common thread running through constructor/setter/field injection, which we'll cover one at a time in the sections ahead.

Why Does It Exist?

The core problem Dependency Injection solves is **tight coupling** -- a class embedding a concrete implementation of another class directly inside itself (a `new SomeClass()` line). That has three concrete costs: **untestability** (you're forced to test against a real email service, with no way to avoid the network call), **difficulty changing** (switching to SMS tomorrow means opening up `OrderService` and editing its code), and **mixed responsibility** (a class ends up owning both "what to do" and "how to construct its dependencies" at once).

DI solves all three by moving the dependency outside the class: `OrderService` doesn't know which `NotificationSender` it gets, only that it needs *a* `NotificationSender`. As we'll see in the sections ahead, that separation both speeds up tests and makes adding a new channel possible without touching existing code.

History

Dependency Injection existed before Spring -- the idea's roots go back to the "Inversion of Control" discussions of the 1990s. But it owes its name and popularity largely to Spring Framework: in his 2002 book *Expert One-on-One J2EE Design and Development*, Rod Johnson proposed a much lighter alternative to the heavy, complex EJB (Enterprise JavaBeans) model of the time -- those ideas took concrete shape as Spring Framework 1.0 in 2004.

That same year, Martin Fowler's article "Inversion of Control Containers and the Dependency Injection pattern" clarified the until-then vaguely used term "Inversion of Control" and proposed the name "Dependency Injection" -- most of today's terminology comes from that article. In 2009, JSR-330 (`javax.inject`, known today as `jakarta.inject`) standardized shared annotations like `@Inject`, taking DI beyond being Spring-specific -- Spring still prefers its own `@Autowired`, but supports `@Inject` too.

The Tight Coupling Problem

Let's see the problem we mentioned in "Why Does It Exist?" in concrete code -- an `OrderService` that creates its own `EmailSender`:

```
// The "before" picture: OrderService constructs its own dependency with `new`,
// so it is permanently welded to EmailSender -- no other channel, and no fake
// version for a test, can ever take its place.
class EmailSender {
    void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

class OrderService {
    private final EmailSender emailSender = new EmailSender();

    void placeOrder(String customerEmail, String item) {
        // Business logic and object construction are tangled together here.
        emailSender.send(customerEmail, "Your order for '" + item + "' has been placed.");
    }
}

class TightlyCoupledOrderService {
    public static void main(String[] args) {
        OrderService orderService = new OrderService();
        orderService.placeOrder("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.

        // There is no way to send this via SMS instead, and no way to replace
        // EmailSender with a fake for a test -- OrderService leaves us no seam.
    }
}
```

As long as `OrderService` embeds the line `new EmailSender()`, the only way to switch it to SMS -- or use a fake sender in a test -- is to open up `OrderService`'s source and change it. The problem isn't `EmailSender` itself -- it's that `OrderService` bundled the decision of *which* sender to use together with the logic of *using* one.

What Is Inversion of Control (IoC)?

IoC in its smallest form: moving the decision of object creation outside the class that uses it. Below, `OrderNotifier` no longer creates `EmailMessageSender` itself with `new` -- it hands that job to a separate factory:

```
// Inversion of Control, in its smallest possible form: OrderNotifier no
// longer decides HOW its dependency gets built -- a separate factory does,
// and OrderNotifier only asks for the finished object.
interface MessageSender {
    void send(String to, String message);
}

class EmailMessageSender implements MessageSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

// The factory is the one place that knows EmailMessageSender exists. Swapping
// the concrete implementation later means editing this one method, not every
// class that used to call `new` directly.
class MessageSenderFactory {
    static MessageSender create() {
        return new EmailMessageSender();
    }
}

class OrderNotifier {
    private final MessageSender messageSender;

    OrderNotifier() {
        // OrderNotifier still decides to CALL the factory itself here -- that
        // is the piece "Dependency Injection: Programming Against a Contract"
        // removes next: even the factory call moves outside this class.
        this.messageSender = MessageSenderFactory.create();
    }

    void notifyCustomer(String email, String item) {
        messageSender.send(email, "Your order for '" + item + "' has been placed.");
    }
}

class ManualFactoryExample {
    public static void main(String[] args) {
        OrderNotifier notifier = new OrderNotifier();
        notifier.notifyCustomer("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.
    }
}
```

`OrderNotifier` still calls the factory itself -- control hasn't fully inverted yet, it's just moved one step outward. In the next section we remove that last step too, arriving at a version where `OrderService` doesn't call anything itself and instead receives a ready-made object straight through its constructor.

> This idea is sometimes called the "Hollywood Principle": "Don't call us, we'll call you" -- a component doesn't reach out and fetch what it needs itself; it waits for it to be brought to it from outside when needed.

Dependency Injection: Programming to a Contract

Now we remove both the factory step and the concrete class dependency entirely -- `OrderService` depends on a `NotificationSender` **interface**, and which implementation gets used is decided from the outside, through the constructor:

```
// The "after" picture: OrderService now depends only on an abstraction
// (NotificationSender), never on a concrete class -- the same interface
// pattern from the "Interface" lesson, applied to the dependency problem.
interface NotificationSender {
    void send(String to, String message);
}

class EmailNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

class SmsNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[sms to " + to + "] " + message);
    }
}

class OrderService {
    private final NotificationSender notificationSender;

    // The dependency now arrives from OUTSIDE, through the constructor --
    // OrderService no longer contains the words "new EmailNotificationSender()"
    // anywhere. This is dependency injection: the caller decides, and injects.
    OrderService(NotificationSender notificationSender) {
        this.notificationSender = notificationSender;
    }

    void placeOrder(String customerContact, String item) {
        notificationSender.send(customerContact, "Your order for '" + item + "' has been
placed.");
    }
}

class NotificationSenderExample {
    public static void main(String[] args) {
        OrderService emailBackedService = new OrderService(new EmailNotificationSender());
        emailBackedService.placeOrder("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.

        // Same OrderService class, a completely different channel -- nothing
        // inside OrderService changed to make this possible.
        OrderService smsBackedService = new OrderService(new SmsNotificationSender());
        smsBackedService.placeOrder("+90 555 000 00 00", "Java 21 Book");
        // [sms to +90 555 000 00 00] Your order for 'Java 21 Book' has been placed.
    }
}
```

This is the "program to an interface, not an implementation" principle from the Interface lesson, applied to the dependency problem. `OrderService`'s source code never mentions

`EmailNotificationSender` or `SmsNotificationSender` by name -- as the two calls in `main` show, the same `OrderService` can be wired to two different channels without changing a single line.

Constructor Injection

The most common way to hand over a dependency is to take it as a constructor parameter and store it in a `final` field:

```
// Constructor Injection: the dependency is a required constructor parameter,
// stored in a `final` field. There is no way to end up with a half-built
// OrderService that is missing its NotificationSender -- the object simply
// cannot exist without one.
interface NotificationSender {
    void send(String to, String message);
}

class EmailNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

class OrderService {
    private final NotificationSender notificationSender;
    private final String storeName;

    // Multiple dependencies/parameters are injected the same way -- just more
    // constructor arguments. All of them are guaranteed to be set once the
    // constructor returns.
    OrderService(NotificationSender notificationSender, String storeName) {
        this.notificationSender = notificationSender;
        this.storeName = storeName;
    }

    void placeOrder(String customerContact, String item) {
        notificationSender.send(customerContact,
                "[" + storeName + "] Your order for '" + item + "' has been placed.");
    }
}

class ConstructorInjectionExample {
    public static void main(String[] args) {
        OrderService orderService = new OrderService(new EmailNotificationSender(), "Java
Kitabevi");
        orderService.placeOrder("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] [Java Kitabevi] Your order for 'Java 21 Book' has
been placed.

        // The line below would not compile if uncommented -- there is no
        // no-argument constructor, so "forgetting" the dependency is not an
        // option the compiler will allow.
        // OrderService broken = new OrderService();
    }
}
```

`notificationSender` and `storeName` are **always** populated for as long as the `OrderService` object exists -- there's no way to forget them, because the compiler won't let you create an `OrderService` without those parameters. We'll look more closely at why that guarantee matters in "Why Is Constructor Injection Recommended?".

Setter Injection

The second approach hands over the dependency **after** the object already exists, through an ordinary setter method:

```
// Setter Injection: the dependency is assigned through an ordinary setter
// method AFTER the object already exists -- useful for genuinely optional
// dependencies, but it also means the object can exist in a "half-wired"
// state until someone remembers to call the setter.
interface NotificationSender {
    void send(String to, String message);
}

class EmailNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

class OrderService {
    // Not final -- it has to stay reassignable so the setter can populate it
    // after construction.
    private NotificationSender notificationSender;

    void setNotificationSender(NotificationSender notificationSender) {
        this.notificationSender = notificationSender;
    }

    void placeOrder(String customerContact, String item) {
        // If setNotificationSender(...) was never called, this throws a
        // NullPointerException here -- at call time, not at construction time.
        notificationSender.send(customerContact, "Your order for '" + item + "' has been
placed.");
    }
}

class SetterInjectionExample {
    public static void main(String[] args) {
        OrderService orderService = new OrderService();
        orderService.setNotificationSender(new EmailNotificationSender());
        orderService.placeOrder("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.

        // A second OrderService, created but never wired -- this compiles fine
        // and only fails much later, when placeOrder() actually runs.
        OrderService forgotten = new OrderService();
        try {
            forgotten.placeOrder("mehmet@example.com", "Spring Boot Book");
        } catch (NullPointerException e) {
            System.out.println("Failed: notificationSender was never set.");
            // Failed: notificationSender was never set.
        }
    }
}
```

Here `notificationSender` can no longer be `final` -- it has to stay reassignable so the setter can populate it later. The second `OrderService` in `main` shows the cost of that: calling `placeOrder(...)` without first calling `setNotificationSender(...)` fails right there, at runtime, not when the object was created.

Field Injection

The third approach hands over the dependency through neither a constructor nor a setter -- it's "injected" straight into a field. In Spring you'd see this as an `@Autowired` field; here we simulate the same mechanism by hand, to see what a framework does behind the scenes:

```
import java.lang.reflect.Field;

// Field Injection: a framework (Spring's @Autowired on a field is the classic
// example) reaches directly into a private field and sets it via reflection --
// the same Field.setAccessible(true) + Field.set(...) mechanism from the
// Reflection lesson's "Accessing Private Fields and Methods" section, just driven
// by a framework instead of your own code.
interface NotificationSender {
    void send(String to, String message);
}

class EmailNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

class OrderService {
    // A real Spring field would be annotated @Autowired; there is no
    // constructor or setter here at all -- nothing but the bare field.
    private NotificationSender notificationSender;

    void placeOrder(String customerContact, String item) {
        notificationSender.send(customerContact, "Your order for '" + item + "' has been
placed.");
    }
}

class FieldInjectionExample {
    public static void main(String[] args) throws ReflectiveOperationException {
        OrderService orderService = new OrderService();

        // This is, in miniature, what a dependency injection framework does
        // for every @Autowired field: find it by reflection, force it
        // accessible, and set it -- no constructor call, no setter call.
        Field field = OrderService.class.getDeclaredField("notificationSender");
        field.setAccessible(true);
        field.set(orderService, new EmailNotificationSender());

        orderService.placeOrder("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.

        // Nothing in OrderService's own source code reveals how the field got
        // its value -- that opacity is exactly why "Common Mistakes" warns
        // against relying on field injection.
    }
}
```

`OrderService` has no constructor or setter at all -- the `field.set(...)` call writes directly into the `private` field from the outside, using the exact mechanism covered in the Reflection lesson's "Accessing Private Fields and Methods" section. In a real Spring app the container does this instead of you, but the result is the same: you can't tell from `OrderService`'s source how that field got filled.

> Field injection also requires reflection to test `OrderService` -- a plain `new OrderService(fakeSender)` call can't set the dependency at all, because there's no constructor that accepts it. "Common Mistakes" covers this and other reasons field injection is generally avoided.

Comparing the Injection Styles

Lined up side by side:

- **Constructor Injection:** the dependency is `final`, required, and guaranteed the moment the object is created. A missing dependency is caught **at compile time** (if the parameter is missing) or, at the latest, the instant the object is constructed.
- **Setter Injection:** the dependency is mutable and can genuinely be optional. A missing dependency only surfaces at runtime, on the exact line where it's actually used.
- **Field Injection:** the least code (no constructor or setter to write), but the least control -- where the dependency comes from isn't visible in the source, and testing it by hand (without a framework) requires reflection.

These three aren't mutually exclusive -- the same class could take one required dependency through the constructor and one optional one through a setter. But in practice, a single style is almost always preferred over the others; the next section covers why.

Why Is Constructor Injection Recommended?

Constructor injection being the recommended default isn't arbitrary -- it comes from its guarantees:

```
import java.util.Objects;

// Constructor injection lets every dependency be `final` -- once built, an
// OrderService can never end up pointing at a different (or missing)
// NotificationSender. Combined with an explicit null-check, a broken wiring
// attempt fails immediately and loudly, not with a mysterious NPE three
// method calls later (compare with "Setter Injection").
interface NotificationSender {
    void send(String to, String message);
}

class EmailNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

class OrderService {
    private final NotificationSender notificationSender;

    OrderService(NotificationSender notificationSender) {
        // Fail fast: if the caller passes null, we find out right here, at
        // the exact line that got it wrong -- not somewhere deep inside
        // placeOrder() later.
        this.notificationSender = Objects.requireNonNull(notificationSender,
"notificationSender must not be null");
    }

    void placeOrder(String customerContact, String item) {
        notificationSender.send(customerContact, "Your order for '" + item + "' has been
placed.");
    }
}

class ImmutableOrderService {
    public static void main(String[] args) {
        OrderService orderService = new OrderService(new EmailNotificationSender());
        orderService.placeOrder("ayse@example.com", "Java 21 Book");
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.

        try {
            new OrderService(null);
        } catch (NullPointerException e) {
            System.out.println("Failed immediately: " + e.getMessage());
            // Failed immediately: notificationSender must not be null
        }
    }
}
```

Thanks to `Objects.requireNonNull(...)`, trying to build an `OrderService` with a `null` dependency fails **immediately**, as `main` shows -- right where the mistake happened. With setter injection (see "Setter Injection"), that same failure could surface much later, on a line that looks completely unrelated.

> Constructor injection has a hidden benefit too: once a constructor's parameter list creeps up to five or six, that's usually an early sign the class has taken on too many responsibilities -- setter/field injection hides this "code smell," because a large number of dependencies gets spread across scattered lines instead of standing out at a glance.

Dependency Injection and Testability

DI's most immediate everyday payoff shows up in testing -- instead of a real `NotificationSender`, all it takes is a fake one that just records what it was asked to send:

```
import java.util.ArrayList;
import java.util.List;

// The payoff of depending on an interface: in a test, we can swap the real
// EmailNotificationSender for a tiny in-memory fake that just records what it
// was asked to send -- no real email is sent, and the test can assert on
// exactly what OrderService tried to do.
interface NotificationSender {
    void send(String to, String message);
}

class FakeNotificationSender implements NotificationSender {
    final List<String> sentMessages = new ArrayList<>();

    @Override
    public void send(String to, String message) {
        sentMessages.add(to + ": " + message);
    }
}

class OrderService {
    private final NotificationSender notificationSender;

    OrderService(NotificationSender notificationSender) {
        this.notificationSender = notificationSender;
    }

    void placeOrder(String customerContact, String item) {
        notificationSender.send(customerContact, "Your order for '" + item + "' has been
placed.");
    }
}

class TestableOrderServiceExample {
    public static void main(String[] args) {
        FakeNotificationSender fake = new FakeNotificationSender();
        OrderService orderService = new OrderService(fake);

        orderService.placeOrder("ayse@example.com", "Java 21 Book");

        // A hand-rolled assertion -- no test framework needed to see the point:
        // this check runs against memory, in milliseconds, without a real
        // email provider or network call anywhere in sight.
        if (fake.sentMessages.size() != 1) {
            throw new AssertionError("Expected exactly one message to be sent");
        }
        System.out.println("Test passed: " + fake.sentMessages.get(0));
        // Test passed: ayse@example.com: Your order for 'Java 21 Book' has been placed.
    }
}
```

`FakeNotificationSender` never touches a real email provider -- the test finishes in milliseconds, and checking `sentMessages` lets us verify exactly what `OrderService` tried to do. None of this was possible with `TightlyCoupledOrderService` from "The Tight Coupling Problem" -- there, the only way to replace `EmailSender` was to edit the source code.

Manual Dependency Injection Without Spring (Composition Root)

Every `main` method up to this point was actually a small "composition root" -- the one place the application knows its concrete classes. Let's make that clearer with a larger example that wires up more than one dependency at once:

```
// A "composition root": one single place in the whole application where
// `new` is allowed to wire concrete classes together. Every class below this
// point (OrderService) only ever sees interfaces -- exactly what a Spring
// container will automate in the next lesson, done here with nothing but
// plain constructors.
interface NotificationSender {
    void send(String to, String message);
}

class EmailNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

interface ReceiptPrinter {
    void print(String item, double price);
}

class ConsoleReceiptPrinter implements ReceiptPrinter {
    @Override
    public void print(String item, double price) {
        System.out.printf("[receipt] %s - %.2f TL%n", item, price);
    }
}

class OrderService {
    private final NotificationSender notificationSender;
    private final ReceiptPrinter receiptPrinter;

    OrderService(NotificationSender notificationSender, ReceiptPrinter receiptPrinter) {
        this.notificationSender = notificationSender;
        this.receiptPrinter = receiptPrinter;
    }

    void placeOrder(String customerContact, String item, double price) {
        notificationSender.send(customerContact, "Your order for '" + item + "' has been
placed.");
        receiptPrinter.print(item, price);
    }
}

class CompositionRootExample {
    // This method is the composition root: the one place that knows about
    // EmailNotificationSender and ConsoleReceiptPrinter by name. Nothing else
    // in the application does.
    static OrderService buildOrderService() {
```

```
        NotificationSender notificationSender = new EmailNotificationSender();
        ReceiptPrinter receiptPrinter = new ConsoleReceiptPrinter();
        return new OrderService(notificationSender, receiptPrinter);
    }

    public static void main(String[] args) {
        OrderService orderService = buildOrderService();
        orderService.placeOrder("ayse@example.com", "Java 21 Book", 349.90);
        // [email to ayse@example.com] Your order for 'Java 21 Book' has been placed.
        // [receipt] Java 21 Book - 349.90 TL
    }
}
```

Outside of `buildOrderService()`, neither `OrderService` itself nor the code calling it knows that `EmailNotificationSender` or `ConsoleReceiptPrinter` exist. In real applications this pattern is known as "Pure DI" or "Poor Man's DI" -- it gets you all the benefits of IoC using nothing but classes and constructors, no framework required; it's still a perfectly valid choice for small applications or whenever you want to avoid a framework dependency.

A Quick Look at How Spring Automates DI

Spring automates the composition root we just wrote by hand, using a container -- it scans classes (`@Component` / `@Service`), reads their constructors (`@Autowired`), and builds the objects itself, in the right order:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

// A preview only -- this file will not do anything useful on its own, since
// there is no container here to create these objects. It shows the same
// OrderService design from "Manual Dependency Injection Without Spring",
// now annotated so that a Spring container could build the composition root
// FOR us.
interface NotificationSender {
    void send(String to, String message);
}

@Component
class EmailNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

@Service
class OrderService {
    private final NotificationSender notificationSender;

    // @Autowired on a constructor is optional when there is only one
    // constructor (Spring uses it automatically) -- it is written explicitly
    // here to keep the intent visible, matching "Constructor Injection".
    @Autowired
    OrderService(NotificationSender notificationSender) {
        this.notificationSender = notificationSender;
    }

    void placeOrder(String customerContact, String item) {
        notificationSender.send(customerContact, "Your order for '" + item + "' has been
placed.");
    }
}

class SpringPreviewExample {
    // No output worth demonstrating here: without an ApplicationContext,
    // nobody scans for @Component/@Service or calls this constructor.
    // "Spring IoC Container & Bean Lifecycle" is where that container itself
    // gets built.
    public static void main(String[] args) {
        System.out.println("This class needs a Spring ApplicationContext to do anything --
see the next lesson.");
        // This class needs a Spring ApplicationContext to do anything -- see the next
lesson.
    }
}
```

This file does nothing on its own, since there's no running `ApplicationContext` to scan it, find the `@Autowired` constructor, and call it -- that container is exactly what we'll cover in "Spring IoC Container & Bean Lifecycle." What matters for now: the `OrderService` here is **identical**

in design to the one in "Manual Dependency Injection Without Spring (Composition Root)" -- Spring just does, by reading annotations, what `buildOrderService()` did by hand.

> Don't confuse this with the "Simple Dependency Injection Container" mini project from the Reflection lesson: the `SimpleContainer` we wrote there **discovered on its own, via reflection**, which parameters a type's constructor needed -- here, we explicitly mark which implementation wires to which interface with `@Component` / `@Service`. Spring's real container is a mix of both: it uses reflection, but it's also steered by annotations.

Best Practices

- **Default to constructor injection** -- it makes required dependencies `final` and catches a missing one at the earliest possible point (see "Why Is Constructor Injection Recommended?").
- **Reserve setter injection for genuinely optional dependencies** -- if a class can't work meaningfully without a dependency, it belongs in the constructor, not a setter.
- **Avoid field injection** -- it provides neither testability nor visibility into a class's dependencies (see "Field Injection" and "Common Mistakes").
- **Depend on interfaces, not concrete classes** ("Dependency Injection: Programming to a Contract") -- this lets you swap real implementations, or use a fake one in tests, without touching any calling code.
- **Read a growing constructor parameter list as a warning** -- it's usually a sign the class has taken on too many responsibilities; consider splitting the class instead of adding yet another parameter.
- **Fail fast with `Objects.requireNonNull(...)`** ("Why Is Constructor Injection Recommended?") -- finding out about a missing dependency immediately, when the object is built, always beats hitting an unrelated error much later.

Common Mistakes

1. Defaulting to field injection because it's "less code." Less code means less control -- where the dependency comes from isn't visible in the source, and testing it by hand requires reflection (see "Field Injection").

2. Looking for a missing setter-injected dependency on the line where the error appears. The real cause is usually a much earlier line where a `setX(...)` call was forgotten (see "Setter Injection").

3. Treating a five-or-six-parameter constructor as normal. That's an early sign the class has taken on more than one responsibility (see "Why Is Constructor Injection Recommended?").

4. Letting a `null` dependency be accepted silently, with no check like `Objects.requireNonNull(...)`. Such an object gets built successfully but blows up later, at its first real use, somewhere that looks unrelated (see "Why Is Constructor Injection Recommended?").

5. Assuming DI is a Spring-specific concept. As "Manual Dependency Injection Without Spring (Composition Root)" shows, DI is a design idea that works with no framework at all -- Spring just automates it.

6. Skipping interfaces and depending on concrete classes instead. This brings back the tight coupling problem from "Why Does It Exist?" and makes it impossible to use a fake implementation in tests.

Summary, Cheat Sheet, and Glossary

Dependency Injection is an object receiving its dependencies from the outside instead of creating them itself; Inversion of Control is the more general "hand control outward" idea behind it. Key points:

- Tight coupling (creating a concrete class directly with `new`) leads to untestability, difficulty changing, and mixed responsibility
- Three injection styles: **constructor** (required, `final`, earliest possible failure), **setter** (optional, reassignable later), **field** (least code, least control)
- Constructor injection should be the default -- guaranteed population, fail-fast validation, and a crowded parameter list works as an early design warning
- A "composition root": the one place an application knows its concrete classes, where all the `new` calls collect -- delivers IoC's benefits without Spring
- Spring automates the same idea with `@Component` / `@Service` scanning and `@Autowired` -- the container itself is the subject of the next lesson (Spring IoC Container & Bean Lifecycle)

Quick reference:

```
// Tight coupling (avoid this)
class OrderService {
    private final EmailSender sender = new EmailSender();
}

// Constructor injection (recommended default)
class OrderService {
    private final NotificationSender sender;
    OrderService(NotificationSender sender) {
        this.sender = Objects.requireNonNull(sender);
    }
}

// Setter injection (only for genuinely optional dependencies)
class OrderService {
    private NotificationSender sender;
    void setSender(NotificationSender sender) { this.sender = sender; }
}

// Field injection (Spring: @Autowired; no manual equivalent without a framework/reflection)
class OrderService {
    private NotificationSender sender; // set by a framework via reflection
}

// Composition root: the one place that knows the concrete classes
class AppComposition {
    static OrderService buildOrderService() {
        return new OrderService(new EmailNotificationSender());
    }
}
```

Glossary

Dependency Injection (DI) — An object receiving the dependencies it needs from the outside, instead of creating them itself.

Inversion of Control (IoC) — A component's flow/dependencies being managed by something outside itself (a composition root or a container) rather than by the component; DI is the most common concrete way of implementing IoC.

Tight coupling — A class being directly dependent (via `new`) on a concrete implementation of another class it needs.

Constructor Injection — A dependency taken as a required constructor parameter and stored in a `final` field.

Setter Injection — A dependency handed over, optionally, through a setter method after the object has already been created.

Field Injection — A dependency assigned directly to a field, without going through a constructor or setter, typically by a framework via reflection.

Composition root — The one place in an application where concrete classes are known and `new` calls collect; also known as Pure DI or Poor Man's DI.

Fail-fast — Throwing an error (e.g., for a missing dependency) at the earliest point it can be detected, usually when the object is constructed; makes the source of the error easy to find.

Test double / fake — An object that stands in for a real implementation in a test, with simplified or observable behavior.

Appendix: Mini Project — A Multi-Channel Notification Dispatcher

Let's combine what we've learned so far ("Constructor Injection", "Dependency Injection: Programming to a Contract") and take it one step further: instead of a single `NotificationSender`, the dependency becomes **every implementation at once**. The idea is simple -- `NotificationDispatcher` forwards the same message to every channel in the list it was given, without knowing how many channels there are or what they're called:

```
import java.util.List;

// Combines "Constructor Injection" (a required, final dependency) with a
// twist Spring uses constantly: injecting a WHOLE LIST of implementations at
// once, so every registered channel gets used without NotificationDispatcher
// ever naming a single concrete class.
interface NotificationSender {
    void send(String to, String message);
}

class EmailNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[email to " + to + "] " + message);
    }
}

class SmsNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[sms to " + to + "] " + message);
    }
}

class PushNotificationSender implements NotificationSender {
    @Override
    public void send(String to, String message) {
        System.out.println("[push to " + to + "] " + message);
    }
}

class NotificationDispatcher {
    private final List<NotificationSender> senders;

    // Every NotificationSender the caller decides to pass in gets used -- the
    // dispatcher itself has zero knowledge of how many channels exist or what
    // they are called.
    NotificationDispatcher(List<NotificationSender> senders) {
        this.senders = senders;
    }

    void dispatch(String to, String message) {
        for (NotificationSender sender : senders) {
            sender.send(to, message);
        }
    }
}
```

```
import java.util.List;

class NotificationDispatcherDemo {
    public static void main(String[] args) {
        // The composition root: this is the only place that lists the
        // concrete channels by name.
        NotificationDispatcher allChannels = new NotificationDispatcher(
                List.of(new EmailNotificationSender(), new SmsNotificationSender(), new
PushNotificationSender()));

        allChannels.dispatch("ayse@example.com", "Your order has shipped.");
        // [email to ayse@example.com] Your order has shipped.
        // [sms to ayse@example.com] Your order has shipped.
        // [push to ayse@example.com] Your order has shipped.

        // A second dispatcher, wired with a different (smaller) list -- same
        // NotificationDispatcher class, no code changes required.
        NotificationDispatcher emailOnly = new NotificationDispatcher(List.of(new
EmailNotificationSender()));
        emailOnly.dispatch("mehmet@example.com", "Your order has shipped.");
        // [email to mehmet@example.com] Your order has shipped.
    }
}
```

`NotificationDispatcher`'s constructor takes a `List<NotificationSender>` instead of a single `NotificationSender` -- the composition root decides how many elements go into the list (`allChannels` has three, `emailOnly` has one), and not a single line of `dispatch(...)` changes.

> When you inject a `List<NotificationSender>` in a real Spring container, the list's order is **undefined** by default (it can depend on bean definition order, or even class scanning order) -- if you need a specific order, use the `@Order` annotation or the `Ordered` interface. In the `List.of(...)` call we wrote by hand here, we control the order ourselves, but that guarantee doesn't come for free with Spring's automatic scanning.

Appendix: Mini Project — A Payment Processor

The final mini project shows the same ideas again in a different domain (payment processing), with one optional dependency (as mentioned in "Comparing the Injection Styles," not every dependency has to be required). `PaymentProcessor` depends on a required `PaymentGateway` (`Objects.requireNonNull`, see "Why Is Constructor Injection Recommended?") and an optional, nullable `FraudChecker`:

```
import java.util.Objects;

// A second, different domain to show the same ideas hold generally -- not
// just for notifications. PaymentProcessor requires a PaymentGateway
// (constructor injection, see "Why Is Constructor Injection Recommended?"),
// and accepts an OPTIONAL FraudChecker that may legitimately be null -- a
// reminder that not every collaborator needs Objects.requireNonNull.
interface PaymentGateway {
    boolean charge(String cardNumber, double amount);
}

class CreditCardGateway implements PaymentGateway {
    @Override
    public boolean charge(String cardNumber, double amount) {
        System.out.printf("[credit card] Charged %.2f TL to card ending in %s%n",
                amount, cardNumber.substring(cardNumber.length() - 4));
        return true;
    }
}

interface FraudChecker {
    boolean looksSuspicious(double amount);
}

class ThresholdFraudChecker implements FraudChecker {
    private final double threshold;

    ThresholdFraudChecker(double threshold) {
        this.threshold = threshold;
    }

    @Override
    public boolean looksSuspicious(double amount) {
        return amount > threshold;
    }
}

class PaymentProcessor {
    private final PaymentGateway gateway;
    private final FraudChecker fraudChecker; // may be null -- genuinely optional

    PaymentProcessor(PaymentGateway gateway, FraudChecker fraudChecker) {
        this.gateway = Objects.requireNonNull(gateway, "gateway must not be null");
        this.fraudChecker = fraudChecker;
    }

    boolean process(String cardNumber, double amount) {
        if (fraudChecker != null && fraudChecker.looksSuspicious(amount)) {
            System.out.println("[fraud-check] Blocked a suspicious payment of " + amount +
" TL");
            return false;
        }
        return gateway.charge(cardNumber, amount);
    }
}
```

```
class PaymentProcessorDemo {
    public static void main(String[] args) {
        // With fraud checking enabled.
        PaymentProcessor guarded = new PaymentProcessor(new CreditCardGateway(), new
ThresholdFraudChecker(5000));
        guarded.process("4242424242424242", 250.00);
        // [credit card] Charged 250.00 TL to card ending in 4242
        guarded.process("4242424242424242", 8000.00);
        // [fraud-check] Blocked a suspicious payment of 8000.0 TL

        // Without a fraud checker at all -- perfectly legal, since it is optional.
        PaymentProcessor unguarded = new PaymentProcessor(new CreditCardGateway(), null);
        unguarded.process("4242424242424242", 8000.00);
        // [credit card] Charged 8000.00 TL to card ending in 4242

        // A fake gateway swapped in, exactly like "Dependency Injection ve Test
        // Edilebilirlik" -- no real payment provider involved.
        PaymentGateway fakeGateway = (cardNumber, amount) -> {
            System.out.println("[fake] Pretending to charge " + amount + " TL, no real
network call made.");
            return true;
        };
        PaymentProcessor testable = new PaymentProcessor(fakeGateway, null);
        testable.process("4242424242424242", 100.00);
        // [fake] Pretending to charge 100.0 TL, no real network call made.
    }
}
```

Notice the three scenarios in `PaymentProcessorDemo`: with fraud checking, without it (passing `null`), and finally with a fake `PaymentGateway` written on the spot as a lambda, just like in "Dependency Injection and Testability." In all three, `PaymentProcessor`'s own code doesn't change by a single line.

> Instead of letting `FraudChecker fraudChecker` be `null`, the Spring world usually makes this more explicit with `Optional<FraudChecker>` or `@Autowired(required = false)` -- we'll look at how Spring expresses optional dependencies in "Spring IoC Container & Bean Lifecycle."