**JoinPoint — Deep and Simple Explanation**

**🔹 What exactly is a JoinPoint?**

A **JoinPoint** is a **specific moment** during the execution of your program  
where Spring AOP can **insert** some extra code (advice).

👉 In Spring, this "moment" is **always a method call** —  
because Spring AOP works only at the **method level** (not at variable access, not at constructor level — those belong to pure AspectJ).

So:

Whenever a method in your application gets called,  
that method call is a *potential JoinPoint* for Spring.

**💡 Example**

public void saveEmployee() {

System.out.println("Saving employee to database...");

}

When this method runs →  
the **moment right before it starts**, the **moment after it ends**,  
and the **moment when it throws an exception**  
are **JoinPoints** where Spring can "join" your extra logic.

**🧠 Why the word “Join”?**

Because at that point,  
Spring **joins** (connects) your cross-cutting logic (advice)  
with your main business logic (method execution).

That’s why it’s called a **JoinPoint** —  
it’s the **joining point** between two worlds:

* the **business logic** (core function)
* the **cross-cutting logic** (like logging, transaction, security, etc.)

**⚙️ How JoinPoint Works Internally in Spring**

Let’s see the behind-the-scenes step-by-step:

1️⃣ You start your Spring Boot app.  
2️⃣ Spring scans all your beans (classes marked with @Component, @Service, etc.).  
3️⃣ It finds any class with @Aspect.  
4️⃣ Inside that aspect, Spring reads all the advices (@Before, @After, etc.) and their pointcuts.  
5️⃣ It then marks every method that matches those pointcuts as **JoinPoints**.

👉 So Spring keeps track of **which methods** are “joinable” — i.e., methods where advice can run.

When you actually call one of those methods:

* Spring intercepts the call (through proxy)
* Creates a **JoinPoint object** (which contains runtime info like method name, parameters, etc.)
* Then passes it to the advice method (if declared).

**💻 Code Example (Simple View)**

@Before("execution(\* in.orcas.dao.\*.\*(..))")

public void logMethodDetails(org.aspectj.lang.JoinPoint jp) {

System.out.println("📍 Method Name: " + jp.getSignature().getName());

System.out.println("📦 Arguments: " + Arrays.toString(jp.getArgs()));

System.out.println("🎯 Target Object: " + jp.getTarget().getClass().getName());

}

**🔍 What JoinPoint gives you:**

| **Method** | **Meaning** |
| --- | --- |
| jp.getSignature().getName() | Name of the method being executed |
| jp.getArgs() | The arguments passed to the method |
| jp.getTarget() | The actual target object (business class) |
| jp.getThis() | The proxy object created by Spring |
| jp.getKind() | Type of joinpoint (in Spring AOP → always "method execution") |
| jp.getSignature().toLongString() | Complete method signature (return type, class, params) |

**🎯 Real-Life Analogy**

Let’s imagine you’re watching a movie 🎬.

Every time an actor starts or ends a scene —  
you could *pause* the movie and insert something extra —  
like showing behind-the-scenes footage or adding subtitles.

These *pause points* are like **JoinPoints**.

You can:

* Add an intro before the scene starts (@Before)
* Add a summary after the scene ends (@AfterReturning)
* Add a failure message if the scene goes wrong (@AfterThrowing)

So JoinPoint = The **exact place** where you can *cut into* the movie and *insert* your extra clip.

**⚙️ Visual Flow**

Caller

↓

Proxy

↓

[JoinPoint detected]

↓

→ Run @Before Advice

↓

→ Execute target method (business logic)

↓

→ Run @AfterReturning or @AfterThrowing Advice

↓

Back to caller

At the “JoinPoint detected” step, Spring AOP knows:

* Which method is being called
* Which advice to trigger
* What arguments were passed
* What target object it belongs to

That’s why JoinPoint is like the **gateway** for your advice to know everything about what’s happening.

**🧩 In short:**

| **Concept** | **Meaning** |
| --- | --- |
| **JoinPoint** | The exact moment during method execution where advice can be applied |
| **Example** | When EmployeeDao.saveEmployee() runs |
| **What Spring does** | Creates a JoinPoint object to describe this execution |
| **Use in advice** | Helps you log details, parameters, class names, etc. |
| **Analogy** | A checkpoint in a movie where you can pause and add extra footage |

**Weaving — (How Spring “injects” your extra logic into business logic)**

**🔹 Concept**

👉 **Weaving** means **combining** your **Aspect (advice logic)** with your **main business logic (target class)**.

In simple words:

Weaving is the process of **attaching your advice** to the **actual methods** that match the pointcut.

So whenever your business method runs,  
Spring makes sure your extra code (like logging, transaction, or security check)  
is also executed — **without changing your original class code**.

**💬 In very simple terms**

You wrote your business class (for example PaymentService)  
and separately wrote your aspect (LoggingAspect).

Now Spring’s job is to **mix** these two together —  
so that when your method runs, your logging (advice) also runs.

That “mixing process” is what we call **Weaving**.

**⚙️ What Actually Happens Internally**

Let’s understand step by step:

1️⃣ You start your Spring application.  
2️⃣ Spring finds your @Aspect classes (like LoggingAspect).  
3️⃣ Spring looks at all the pointcut expressions inside them.  
4️⃣ It scans your other beans (like PaymentService, EmployeeDao, etc.) to see which methods match those pointcuts.  
5️⃣ For every matching method, Spring creates a **proxy object** that wraps your original class.  
6️⃣ Inside that proxy, Spring “weaves” (connects) your advice code with your method.

Now, when you call your method:

* You are **actually calling the proxy**, not your real class.
* That proxy runs your advice code **before** or **after** your method (depending on annotation).

✅ This process of connecting advice with target = **Weaving**.

**🎯 Analogy: Tailor and Clothes**

Imagine you have a simple plain shirt 👕 (your **business class**).  
Now, you want to add a special design (your **advice**).

A tailor takes your shirt and *stitches* a pattern or embroidery on it  
without changing the shirt’s basic structure.

This stitching or attaching is **weaving**.

So your shirt (method) and your design (advice) are now combined —  
but still, you didn’t remake the whole shirt from scratch!

That’s exactly what Spring does:  
It weaves (stitches) your advice code into your business method using proxies.

**🧠 Types of Weaving (based on when it happens)**

| **Type** | **When It Happens** | **Explanation** |
| --- | --- | --- |
| **Compile-time Weaving** | During code compilation | Aspect code is merged into bytecode (.class file) while compiling. Used in pure AspectJ (not Spring). |
| **Load-time Weaving** | When class is loaded into JVM | Advice is weaved when class loader loads the class file into memory. |
| **Runtime Weaving** | When program runs | Proxy-based weaving happens dynamically at runtime (✅ This is what **Spring AOP** uses). |

So in **Spring AOP**, weaving happens **at runtime** using **dynamic proxies**.

That means —  
your class file on disk is not modified.  
Spring just wraps your bean in a proxy at runtime.

**🧩 Example Flow (Behind the Scenes)**

Let’s take a small example:

@Aspect

@Component

public class LoggingAspect {

@Before("execution(\* in.orcas.service.PaymentService.\*(..))")

public void logBefore() {

System.out.println("Logging before payment...");

}

}

@Service

public class PaymentService {

public void makePayment() {

System.out.println("Processing payment...");

}

}

When your app starts:

1️⃣ Spring sees LoggingAspect and reads the pointcut.  
2️⃣ It sees that PaymentService.makePayment() matches it.  
3️⃣ It doesn’t modify your class file — instead,  
it creates a **proxy** around it.  
That proxy is the **woven version** of your class.

Now when you call:

paymentService.makePayment();

This actually happens:

Proxy object → runs @Before advice

↓

runs real method makePayment()

↓

(optional @After advice)

Here, **weaving** happened at runtime → proxy “joined” advice and method together.

**🧩 Quick Summary Table**

| **Concept** | **Meaning** | **In Spring AOP** |
| --- | --- | --- |
| **Aspect** | A class that contains advice (extra logic) | @Aspect class |
| **Advice** | Actual logic to run before/after method | @Before, @After, etc. |
| **Pointcut** | Expression that selects methods | "execution(\* in.orcas..\*.\*(..))" |
| **JoinPoint** | Actual method execution matched by pointcut | When method runs |
| **Weaving** | Process of connecting advice to target method | Happens at runtime |
| **Proxy** | The generated wrapper that handles advice logic | Created automatically by Spring |

**🎬 Real-World Analogy (Full Picture)**

Let’s say you run a restaurant 🍽️.

* **Chef** → business class (real worker doing main task)
* **Manager** → aspect (extra checks or logs)
* **Spring** → the organizer who decides when the manager must intervene
* **Weaving** → the moment when the manager’s supervision is linked to the chef’s work
* **Proxy** → the system that ensures manager and chef work in sync.

You didn’t rewrite the chef’s recipe —  
you just told the manager,  
“Before every dish, check the ingredients.”

Spring does exactly that using weaving!

**🧩 Final Summary (in one line)**

🔸 **Weaving** = the process of *linking your advice* with *target methods* so that your extra code runs automatically — done by Spring at runtime using proxies.

**Target — The Real Worker Behind the Scene**

**🔹 Concept**

👉 The **Target** is simply your **original business class object** —  
the one that actually contains your main business logic.

It’s called “target” because this is the **object that AOP aims at**  
when applying the extra behavior (advice).

**💬 In simple words**

When you write a business class like:

@Service

public class PaymentService {

public void processPayment() {

System.out.println("Processing payment...");

}

}

This PaymentService is your **Target class**  
because this is where the **real work** happens (like saving to DB, processing payments, etc.).

Spring never changes this class — it stays clean and simple.  
All extra logic like logging, security, or transactions are added externally via **Aspects**.

So, the Target is your **core business object**,  
and Spring AOP adds support logic around it (before or after)  
without changing your original class.

**🧠 What Actually Happens Internally**

When you enable AOP, Spring does this:

1️⃣ Finds your Target bean (like PaymentService).  
2️⃣ Creates a **Proxy** object that “wraps” around this target.  
3️⃣ When you call a method on your bean —  
you’re actually calling it through the Proxy, not directly.  
4️⃣ The Proxy decides when to run advice and when to call the real target method.

So, the **Target** stays untouched —  
but the **Proxy** controls how it’s accessed.

**🎯 Analogy: Chef in a Restaurant**

Imagine the **Target** is a **Chef** 🍳 in your restaurant.  
He’s the one who actually cooks food.

But before the chef starts cooking:

* The **Manager** (Aspect) checks the ingredients (@Before advice),
* The **Cleaner** cleans up afterward (@After advice).

The chef doesn’t know these people exist —  
he just does his cooking.  
Still, every time he works, these extra steps automatically happen.

The chef = **Target**,  
and these checks = **Advices**,  
coordinated by **Spring’s AOP system**.

**💬 Note:**

* You can have multiple advices (security, transaction, logging) all targeting the same target class.
* Target objects are **normal beans** — annotated with @Component, @Service, etc.
* You never directly modify the target — Spring handles it automatically.

**Proxy — The Smart Middleman**

**🔹 Concept**

👉 A **Proxy** is the **object that Spring actually gives you**  
when you ask for your bean.

It **looks like** your original object (same methods, same name),  
but it secretly has **extra logic** inside.

When you call any method on this proxy —  
it first runs the **AOP advices**,  
then calls your **target method**,  
then runs any **post advices** if needed.

**💬 In very simple words**

When you write:

PaymentService service = context.getBean(PaymentService.class);

service.processPayment();

You might *think* you’re calling PaymentService directly —  
but actually, Spring gives you a **Proxy object** that **wraps** your real service.

So instead of:

You → PaymentService

It’s actually:

You → Proxy → PaymentService

The proxy is like a **gatekeeper** —  
it decides which advices should run before or after calling the actual business method.

**⚙️ How Proxy Works in Spring AOP**

There are **two types** of proxies Spring can create:

| **Type** | **Used For** | **Implementation** |
| --- | --- | --- |
| **JDK Dynamic Proxy** | If your target implements an interface | Uses Java’s built-in proxy (java.lang.reflect.Proxy) |
| **CGLIB Proxy** | If your target class does NOT implement any interface | Uses subclassing (code generation using CGLIB library) |

Spring automatically decides which proxy type to use — you don’t have to choose.

**🧠 Step-by-Step Flow (with Proxy)**

Let’s see what happens internally:

1️⃣ You start your app.  
2️⃣ Spring finds your AOP configuration (@Aspect classes).  
3️⃣ It checks which beans match your pointcuts.  
4️⃣ For every matched bean, it creates a **Proxy** around the original bean (Target).  
5️⃣ When you call a method:

* The Proxy runs all **@Before** advices.
* Then calls the **Target** method.
* If no exception → runs **@AfterReturning**.
* If exception → runs **@AfterThrowing**.
* Finally runs **@After** advice (always).  
  6️⃣ The user (you) only sees the final output,  
  unaware that it was handled through a proxy.

**🎯 Analogy: Security Guard at the Door**

Think of the Proxy as a **Security Guard** 🚪 standing before a VIP (your Target).

When you try to meet the VIP:

* The guard checks your ID (@Before advice).
* Lets you meet the VIP (method executes).
* After meeting, the guard might log your exit or thank you (@After advice).
* If something goes wrong, the guard raises an alert (@AfterThrowing advice).

You never meet the VIP directly —  
you always go **through the guard (proxy)**.

That’s how Spring ensures advices run properly without modifying your VIP (target).

**🧩 Relationship Summary**

| **Concept** | **Meaning** | **Example / Analogy** |
| --- | --- | --- |
| **Target** | The real class that performs business logic | The chef who cooks |
| **Proxy** | The middle object that controls method access and advice execution | The security guard who manages entry and exit |
| **Aspect** | Class that contains advice logic | The manager who decides what checks to run |
| **Advice** | Extra behavior added around methods | Checking, logging, cleaning |
| **Weaving** | Process of connecting advice and target method | Attaching manager’s rules to chef’s work |
| **JoinPoint** | Actual moment when target method runs | When the chef starts cooking |
| **Pointcut** | Rule to select which methods to target | "All cook() methods in kitchen package" |

**💬 Final Summary**

* **Target** → your real object (business logic).
* **Proxy** → a wrapper created by Spring that intercepts method calls.
* **Weaving** → process that connects advices to the target using proxy.
* **JoinPoint** → the actual method execution where advice runs.

🧠 **In one line:**

In Spring AOP, you never directly call your target object.  
You always call a proxy — which weaves the advice logic around your target methods dynamically.

### 🔁 ****Spring AOP Complete Flow****

1️⃣ **Developer writes a business class → Target**  
👉 This class contains the main business logic (like saving employee data).

2️⃣ **Developer writes cross-cutting logic → Aspect (contains Advice + Pointcut)**  
👉 This defines what extra logic to apply (Advice) and where to apply it (Pointcut).

3️⃣ **Spring reads the Pointcut → finds matching JoinPoints (methods)**  
👉 It identifies which methods match your defined expression.

4️⃣ **Spring creates a Proxy object for that Target**  
👉 The proxy wraps your target class to insert AOP logic dynamically.

5️⃣ **When client calls Target → Proxy intercepts it**  
👉 The call first goes to the proxy, not directly to the target.

6️⃣ **Proxy checks if the method matches Pointcut**  
👉 If the called method fits the rule, AOP will activate.

7️⃣ **If yes:**  
→ Runs the @Before advice (pre-logic)  
→ Calls the actual **Target method**  
→ Runs the @After, @AfterReturning, or @AfterThrowing advice (post-logic)

8️⃣ **Spring uses Weaving to connect these together at runtime**  
👉 This is the final stitching of Advice and Target method — all happens dynamically during execution.

💡 **In short:**

Target → Aspect(Pointcut + Advice) → JoinPoint → Proxy → Weaving