**🌱 Step 1: Let’s Recall the Big Picture**

In Spring AOP, we mainly have two parts working together:

1. **Pointcut** → tells *which* methods you want to apply extra logic to.
2. **Advice** → contains the *actual extra logic* (like logging, security, transactions, etc.).

Example:

@Aspect

@Component

public class LoggingAspect {

@Pointcut("execution(\* com.orcas.service.PaymentService.\*(..))")

public void paymentMethods() {}

@After("paymentMethods()")

public void logAfter() {

System.out.println("After Payment method finished.");

}

}

So, here:

* @Pointcut only selects which methods to watch.
* @After tells when to run the advice code (after the selected methods finish).

**🌾 Step 2: What Does execution(...) Mean in Simple Words**

The word **execution** means:

“When the selected method actually starts running.”

It’s like saying:

“Whenever this method begins to run, remember that this is the place where I want to attach my advice.”

It does **not** mean that advice runs during that time.  
It just marks those methods as “targets” for advice.

So, if your pointcut is:

@Pointcut("execution(\* com.orcas.service.PaymentService.makePayment(..))")

That means:

“Whenever the method makePayment() is executed, Spring should apply the advice(s) connected to it.”

It’s just a **rule or a condition**, not a piece of code that runs.

**🌼 Step 3: What Happens When You Call Your Business Method**

Let’s say you call:

paymentService.makePayment();

Now, Spring internally does something special.

It doesn’t directly call your real PaymentService class.  
Instead, it calls a **proxy object** created by Spring.  
This proxy is like a **middleman** between you and your actual business logic.

Why? Because this proxy helps Spring decide:

“Should I run any advice before or after this method?”

**🌻 Step 4: The Proxy Checks Your Pointcut Expression**

Spring’s proxy checks:

“Is this method (makePayment) matching any execution(...) rule?”

If **yes**, then Spring says:

“Okay, I have to run the advice(s) connected to that pointcut.”

If **no**, then Spring just runs your method normally, without any advice.

**🌷 Step 5: Now @After Comes into Action**

Since you used @After("paymentMethods()"),  
Spring understands that:

“Once the makePayment() method finishes, no matter success or failure, I must run this advice.”

So the flow is:

Step 1 → Check pointcut condition (execution rule)

Step 2 → Run the main business logic (makePayment)

Step 3 → After it finishes, run @After advice code

✅ Both work smoothly — no exception happens.

Because:

* execution(...) just tells where the advice should be applied.
* @After tells when the advice should run.

They **don’t fight** with each other — they **work together**.

**🌹 Step 6: Understanding the “Timing” Clearly**

Each advice type tells *when* to run your extra logic.

| **Advice Type** | **When It Runs** | **Example Use** |
| --- | --- | --- |
| @Before | Before the business method starts | Check security, validate data |
| @After | After the business method ends (success or fail) | Close DB connection, cleanup |
| @AfterReturning | Only if the method runs successfully (no exception) | Log “Payment done successfully” |
| @AfterThrowing | Only if the method throws an exception | Log error or alert admin |
| @Around | Before and after both | Measure time, control flow, modify return value |

So, in your example:

* execution(...) → marks the method (makePayment) as a “target”.
* @After → runs extra logic **after** that target method finishes.

✅ No conflict.  
✅ No exception.  
✅ Perfect timing.

**🌼 Step 7: Let’s Think in a Real-Life Example**

Imagine a restaurant:

* **Customer** = the user who calls your business method
* **Waiter** = the proxy created by Spring
* **Chef** = your actual business class (PaymentService)
* **CCTV camera** = the advice (extra logic, like logging)

Now:

1. The waiter receives the order (method call).
2. The waiter checks the CCTV rule (pointcut using execution(...)).  
   → “Should I record this customer?”
3. If yes, the CCTV is turned ON when the chef starts cooking (method runs).
4. After cooking is done, the CCTV stops and logs the entry (advice runs).

Here:

* The CCTV rule (execution) defines *where* to record.
* The actual recording logic (@After) defines *when* to record.

✅ Both work in sequence, not at the same time.  
❌ No conflict.

**🌻 Step 8: Important Internal Truth**

When you see:

@After("execution(\* com.orcas.service.PaymentService.makePayment(..))")

Spring first reads execution(...) → it finds **which method** is being talked about.

Then it attaches your advice logic to that method.

So the advice itself doesn’t “interrupt” your business method.  
It just **waits for the right moment** (as told by @After, @Before, etc.)  
and then executes its logic.

**🌺 Step 9: So, Will There Be Any Exception?**

👉 **No.**  
Not at all. Because both have different roles.

| **Part** | **Role** | **Runs When** |
| --- | --- | --- |
| execution(...) | Tells “where” the advice should be applied | Checked before method starts |
| @After (or any other advice) | Tells “when” the extra code should run | Runs according to its type (before/after/etc.) |

They never overlap or confuse Spring.

Instead, Spring uses both together to perfectly time your advice code.

**🌼 Step 10: Small Timeline Visualization (In Words)**

You call -> paymentService.makePayment()

↓

Spring proxy checks -> Does it match execution(...) rule? ✅

↓

Spring runs business logic -> makePayment()

↓

Method finished (success or failure)

↓

Spring runs @After advice (your extra logic)

↓

Done ✅

Everything happens in proper order.  
No mix-up. No exception. No double execution.

**🌻 Final Summary (Simple but Deep)**

| **Concept** | **What It Really Means** |
| --- | --- |
| execution | A **condition** that selects methods where advice should apply. It does not execute anything. |
| @Pointcut | A method that stores the execution expression for reusability. |
| @After / @Before / etc. | Decide **when** to run your extra logic — before, after, on success, or on error. |
| Together | execution (where) + @After (when) = Properly timed advice execution without errors. |
| Exception? | ❌ No. Spring handles the timing automatically using proxies. |

Eg: SpringAopExample

**⚙️ Real-Time Usage of AOP in Projects**

**🧩 Given Example**

interface EmployeeRepository extends CrudRepository<Employee, Long> {}

@Service

public class EmployeeServiceMgmtImpl implements IEmployeeService {

@Transactional

public void saveEmployee(Employee emp) {

// business logic to save employee

}

}

**🧠 What’s Happening Here**

At first glance, you see a simple method:

saveEmployee(Employee emp)

But because it is annotated with @Transactional,  
**Spring AOP automatically adds extra logic around it** — even though you didn’t write it yourself.

That’s where **AOP (Aspect Oriented Programming)** comes into play.  
Let’s break it down step by step.

**🚀 Step-by-Step Flow (Simplified)**

1. **You call the method:**
2. employeeService.saveEmployee(emp);
3. Spring doesn’t call the original saveEmployee() directly.  
   Instead, it creates a **Proxy object** for EmployeeServiceMgmtImpl.
4. That **Proxy** checks:

“Is there any aspect (extra logic) applied to this method?”

Yes — there is, because of the @Transactional annotation.

1. Before calling your real business logic, Spring runs **transaction management logic**, such as:
   * Opening a database transaction
   * Setting up connection context
2. Then it runs your **actual business logic**:
3. employeeRepository.save(emp);
4. After the method finishes:
   * If no exception → commit transaction
   * If exception occurs → rollback transaction

This entire behavior (begin, commit, rollback)  
is **automatically woven** around your business logic — using **Spring AOP**.

**⚡ So what’s really happening**

| **Phase** | **Handled by AOP** | **What it does** |
| --- | --- | --- |
| Before method runs | @Before (internally used by Spring for @Transactional) | Opens DB transaction |
| Method execution | Your business logic | Saves employee |
| After method completes | @AfterReturning | Commits transaction |
| If exception occurs | @AfterThrowing | Rolls back transaction |

✅ You didn’t write these manually — AOP does it for you automatically.

**🔍 So @Transactional = Built-in Aspect**

When you use:

@Transactional

You’re not using plain Java logic —  
you’re telling **Spring AOP** to attach a **Transaction Management Aspect** around your method.

It uses a **Pointcut** like:

execution(\* com.company.service.\*.\*(..))

to find where @Transactional is used, and then automatically applies advice such as:

* open connection before
* commit or rollback after

**💼 Why AOP is useful in real projects**

In real-time enterprise applications, we use **AOP** to separate *common repetitive logic* (called **Cross-Cutting Concerns**) from business logic.

**🔁 Common Cross-Cutting Concerns handled by AOP:**

| **Cross-Cutting Concern** | **Example** | **Description** |
| --- | --- | --- |
| **Transaction Management** | @Transactional | Begin/commit/rollback automatically |
| **Logging** | Custom @Before / @After | Log who called which method |
| **Security** | @PreAuthorize, @Secured | Check user permissions |
| **Performance Monitoring** | @Around advice | Measure method execution time |
| **Exception Handling** | @AfterThrowing | Catch and log exceptions globally |
| **Caching** | @Cacheable | Store/reuse frequently used data |

All these use **AOP under the hood** — even if you don’t write @Aspect yourself.

**🧩 Real-Life Analogy**

Imagine you run a **bank**.  
Every time someone transfers money, you want to:

* Validate user session (security)
* Begin a transaction
* Log the action
* Commit or rollback after success/failure

Instead of writing these 4 steps inside every method,  
you define them **once** in an **Aspect**, and Spring automatically applies them to all related methods.

That’s the **power of AOP**.

**✅ Summary**

| **Concept** | **Meaning** |
| --- | --- |
| **AOP (Aspect Oriented Programming)** | A technique to add extra behaviors (like logging, transactions) without touching actual business code |
| **@Transactional** | A built-in Spring Aspect that manages database transactions |
| **Spring creates proxy objects** | To insert extra logic before and after your real methods |
| **Why useful** | Keeps your code clean, reusable, and easy to maintain |

**🌱 Let’s Start From the Root: Why We Even Need AOP**

When we write business applications (like HR systems, banking apps, or shopping apps), we often have **common activities** that appear across multiple modules:

* Logging (“method started”, “method ended”)
* Transaction management (start, commit, rollback)
* Security checks (user permissions)
* Performance monitoring (how long method took)
* Exception handling (catch, log errors)

We call these **cross-cutting concerns**,  
because they *cut across* many parts of the application — they are not specific to one module.

Instead of writing this same code again and again in every class,  
Spring gives us **AOP** — Aspect-Oriented Programming — to **separate** these concerns cleanly.

**🧩 Think of the Whole Flow Like a Movie Scene 🎬**

Imagine your business method (saveEmployee()) is the **main actor** on stage.

But behind the camera, we have:

* Director (AOP proxy)
* Script instructions (Pointcut)
* Supporting crew (Advices: @Before, @AfterReturning, @AfterThrowing, @After)

Let’s see how the show runs 👇

**🎭 The Execution Flow — Step by Step**

**Step 1️⃣: The Proxy is the “Middleman”**

When your app starts, Spring doesn’t call your real EmployeeDao directly.  
It secretly creates a **proxy object** that wraps around your actual class.

So when you call:

employeeDao.saveEmployee();

you’re **not really calling your original class**,  
you’re calling the **proxy’s version**.

This proxy checks:

“Hey, is this method (saveEmployee) matched by any **Pointcut**?”

If yes, then it knows:

“Okay, I must also run some **Advice** along with it.”

**Step 2️⃣: The Pointcut — It’s the “Matcher”**

The **Pointcut** defines *which methods* should have the extra behaviour.

Example:

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

public void daoMethods() {}

Meaning:  
“Pick all methods of all classes inside the in.orcas.dao package,  
no matter what they return or what parameters they take.”

So if your method matches this — your advice will be triggered.

**Step 3️⃣: The Advices — The “Supporting Actors”**

These are **small logic blocks** that tell *when* to run and *what* to do.

| **Advice** | **When it runs** | **What it does** |
| --- | --- | --- |
| @Before | Before your business method runs | Setup work, logging, validation |
| @AfterReturning | After your method successfully finishes | Log result, audit, confirmation |
| @AfterThrowing | When your method throws an exception | Error logging, rollback, alert |
| @After | Always runs after method ends | Cleanup, closing resources |

Each advice type acts like a small helper entering at the right time.

**⚙️ Two Possible Paths When Method Runs**

Let’s now zoom into **what happens when your business method executes**.

**🟢 CASE 1: The Method Runs Successfully**

Imagine your method:

public String saveEmployee() {

System.out.println("Employee saved...");

return "Success";

}

Now what happens?

| **Step** | **What runs** | **Description** |
| --- | --- | --- |
| 1️⃣ | @Before advice | runs before business logic — e.g., “Start transaction” |
| 2️⃣ | saveEmployee() | actual logic executes — “Employee saved…” |
| 3️⃣ | @AfterReturning | runs only if no exception — can use return value "Success" |
| 4️⃣ | @After | runs always — “Close transaction” |

✅ So in a success case:  
**@Before → Business Logic → @AfterReturning → @After**

**🔴 CASE 2: The Method Fails (Throws Exception)**

Now suppose your method:

public void saveEmployee() {

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

if (new Random().nextInt(10) < 5)

throw new RuntimeException("Database error!");

}

Then what happens?

| **Step** | **What runs** | **Description** |
| --- | --- | --- |
| 1️⃣ | @Before advice | runs before method — “Start transaction” |
| 2️⃣ | saveEmployee() | throws exception midway |
| 3️⃣ | @AfterThrowing | runs because an exception occurred — can access that exception |
| 4️⃣ | @After | runs finally — “Close transaction, free resources” |

✅ So in a failure case:  
**@Before → Business Logic → @AfterThrowing → @After**

**Eg: SpringAopExample2**

**💡 Deep Insight — How Spring Decides This**

Spring’s proxy keeps track of:

* Whether the method finished successfully or threw an error
* Whether there is a return value or not
* Whether there’s any advice waiting to act on it

Depending on the situation, Spring **automatically triggers** the correct advice.  
You don’t write any try-catch or if manually — Spring manages that behind the proxy.

**🧠 Mnemonic to Remember Easily**

| **Advice** | **When?** | **Think of it as…** |
| --- | --- | --- |
| @Before | Before method runs | “Prepare the stage” |
| @AfterReturning | When method succeeds | “Applaud the success” |
| @AfterThrowing | When method fails | “Handle the error” |
| @After | Always at the end | “Pack up and close the stage” |

**🏢 Real-World Analogy**

Imagine a **banking transaction**:

1. Customer initiates payment (→ Business logic)
2. Bank system logs the request (→ @Before)
3. If payment succeeds (→ @AfterReturning), it sends a confirmation SMS.
4. If payment fails (→ @AfterThrowing), it rolls back and alerts the admin.
5. Either way, it closes the session (→ @After).

This is exactly how AOP ensures **clean, predictable behaviour** across your whole app without repeating code everywhere.

**✅ Summary Table**

| **Situation** | **Advices That Run** | **Purpose** |
| --- | --- | --- |
| Method succeeds | @Before, @AfterReturning, @After | Logging, committing transaction |
| Method fails (exception) | @Before, @AfterThrowing, @After | Rollback, logging error |
| Always runs | @After | Cleanup |