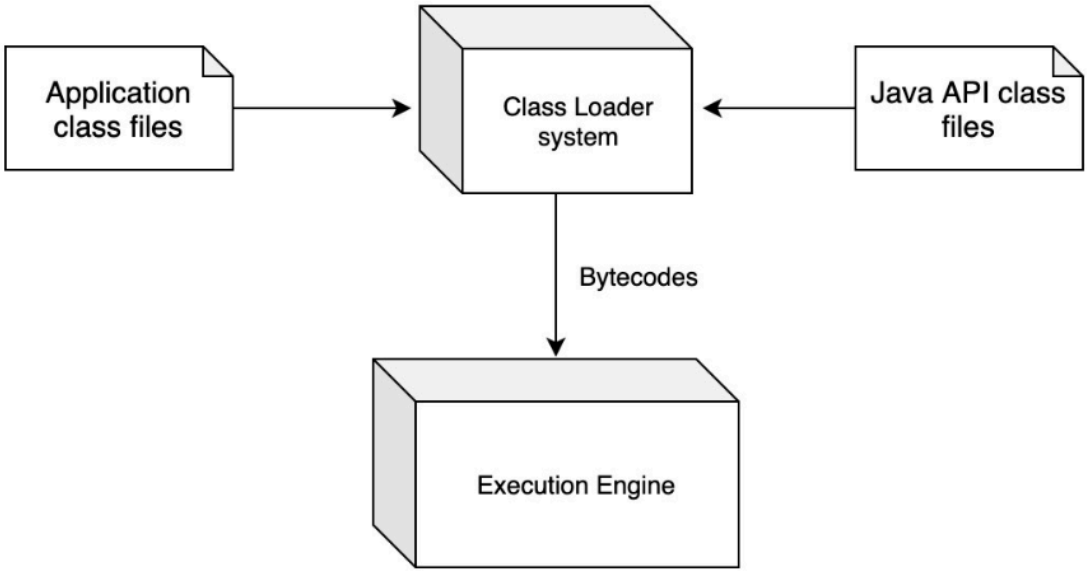
“This document shows what happens inside a Spring Boot application before it serves its first HTTP request.”

0. Class Loader

1. Class Loaders are basically used to load `.class` files into the memory.
2. CL is a part of the java runtime environment and it dynamically loads the java class into memory.
3. It doesn't load all files into the memory in one go. It loads the classes when it is required by the application.



Different types of Class Loader?

There are 3 types of class loader.

- **Bootstrap class loader** -> It is used to load the .class files which are present in <JAVA_HOME>/jre/lib folder.
- **Extension class loader** -> It is used to load the .class files which are present in <JAVA_HOME>/jre/lib/ext folder.
- **Application/System class loader** -> It is used to load the .class files which are present in class path.

1. Bootstrap ClassLoader (core JVM classes):

   It loads:

java.lang.*
java.util.*
java.io.*
java.math.*
java.net.*
java.nio.*
java.time.*
java.security.*
Internal JVM classes

Example:

```
System.class
String.class
Object.class
Thread.class
Math.class
List.class
```

```
System.out.println(String.class.getClassLoader()); // Output null for bootstrap class loader
```

2. Platform / Extension Class Loader (JDK modules that are NOT core):

   It loads:

```
jdk.* packages
JavaFX (for building GUI applications)
```

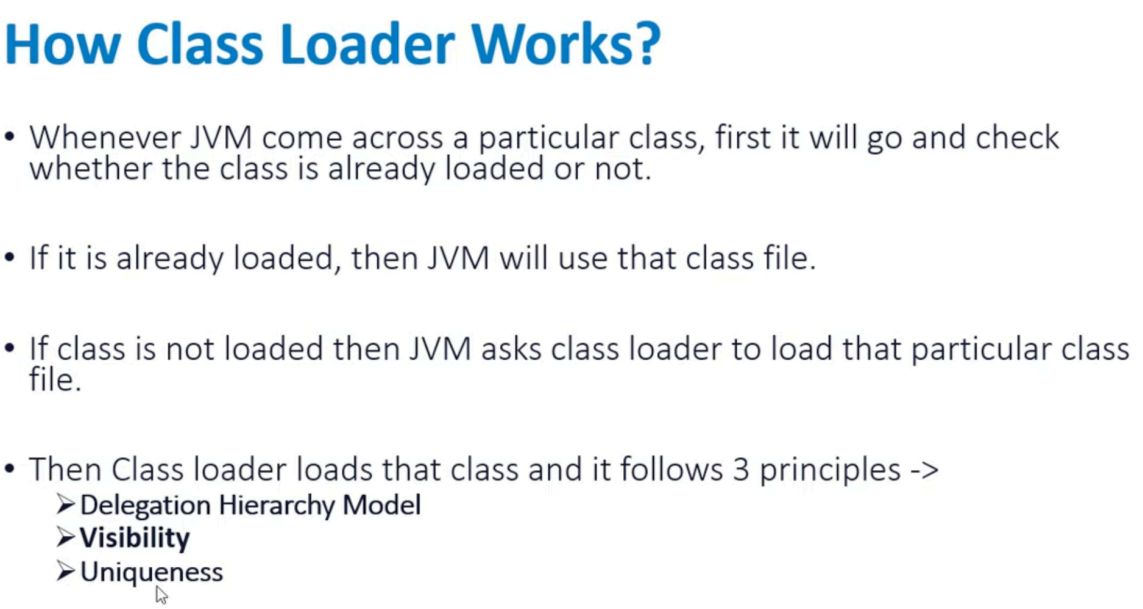
3. Application ClassLoader (application + external libraries):

   It loads:

```
Spring Boot app classes (com.example.*)
Classes inside target/classes
Classes inside jars in /libs
All Spring Framework classes
Hibernate classes
```

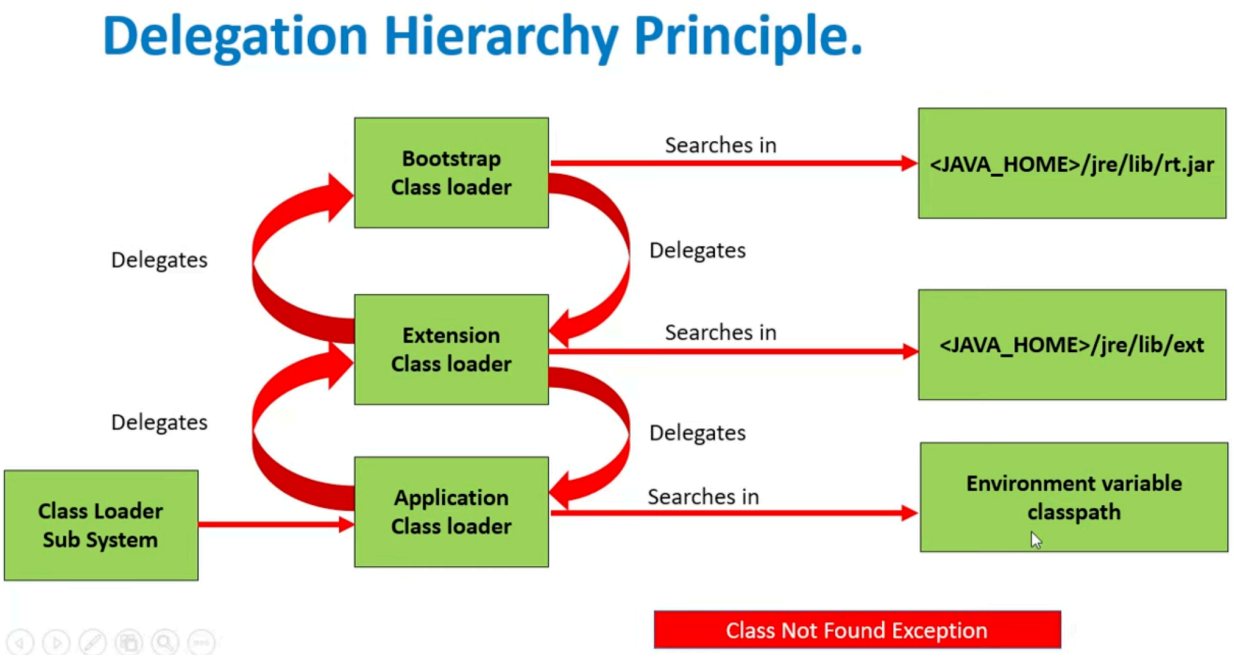
Third-party libraries

```
System.out.println(MyController.class.getClassLoader());  // Output: jdk.internal.loader.ClassLoaders$AppClassLoader@xxxx
```



Delegation Hierarchy Principle

A class loader always asks its parent first before attempting to load a class itself.



Example:

```
public class LoaderInfo {
    public static void main(String[] args) {
        ClassLoader cl = LoaderInfo.class.getClassLoader();
        System.out.println("Loader for this class: " + cl);
        while (cl != null) {
            System.out.println(" -> " + cl);
            cl = cl.getParent();
        }
        System.out.println("Bootstrap loader is represented as null.");
    }
}
```

Output:

Loader for this class:
jdk.internal.loader.ClassLoaders$AppClassLoader@...
 -> jdk.internal.loader.ClassLoaders$AppClassLoader@...
 -> jdk.internal.loader.ClassLoaders$PlatformClassLoader@...
Bootstrap loader is represented as null.

Visibility Principle

Visibility Principle.

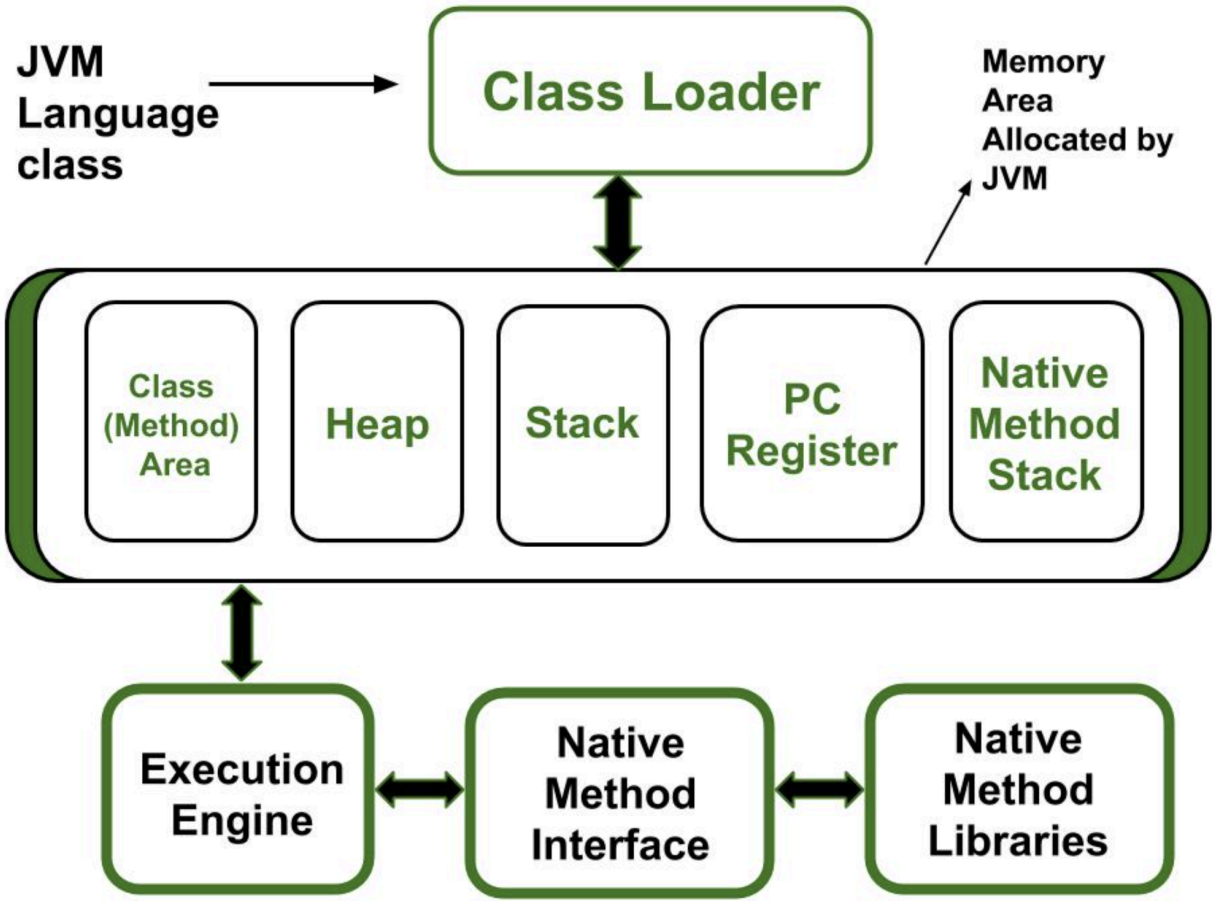
- It says that a class loaded by a parent class loader will be visible to child class loader, but a class loaded by child class loader will not be visible to parent class loader.
- Suppose, a class InterviewMania.class has been loaded by Extension class loader, then this class will only be visible to Extension and Application class loader and will not be visible to Bootstrap Class Loader.
- That is why Class loader follows Delegation Hierarchy Principle.

Uniqueness Principle

Uniqueness Principle.

- The Uniqueness principle says that the classes are unique and there is no repetition of classes.
- This also ensures that the classes which are loaded by parent will not be loaded by child class loader.
- If the parent class loader is not able to load the class then only the child class loader will try to load the class.

1. Method Area, Heap, Stack (JVM Memory)




```
@Service
class ProductService implements Product {
        static String category = "Electronics";
        int stock = 50;
        public int getStock() {
            return stock;
```

```
	}

}
public class MainApp {
	public static void main(String[] args) {
		ProductService service = new ProductService();
		int result = service.getStock();
		System.out.println(result);
	}
}
```

Method Area

Purpose: Stores all **class-level information**.
Loaded once per class when the class is loaded by the JVM.

Contains:

1. Class metadata
    - Class names: ProductService, MainApp
2. Package and type info
    - Parent class: java.lang.Object
    - Implemented interfaces: Product
3. Field definitions (structure only, not values for instances)
    - static String category
    - int stock
4. Static members
    - Static variables and static methods
    - category = "Electronics" (String literal lives in Runtime Constant Pool)

5. Method metadata
    - Method signatures
    - Access modifiers
    - Parameter types
    - Return types
    - Exception tables
6. Method bytecode
    - Bytecode for getStock()
    - Bytecode for constructor
    - Bytecode for main()
7. Annotations
    - @Service (Spring reads this from the Method Area)
8. Runtime Constant Pool
    - String literal "Electronics"
    - Method names
    - Field names

> Method Area does NOT store objects

> It does NOT store instance field values

> It is shared across all threads

Heap

Purpose: Stores **objects created at runtime**.

Contains:

1. ProductService instance
    - stock = 50

> Instance fields always live in the heap

> Each object has its own copy of instance fields

Does not contain:

- Static fields (category)
- Annotations
- Method bytecode

Heap = objects + their instance data only

Stack

Purpose: Stores **method execution data**.
Each method call gets its own stack frame.

When main() executes:

Stack Frame: main()

- service => reference to ProductService object in heap
- result = 50
- args => reference to String[]

Stack stores only:

- Primitive values (int, boolean, double, etc.)
- References (pointers to heap objects)

- Return addresses
- Temporary values used during execution

Stack NEVER stores:

- Objects
- Static fields
- Annotations
- Method bytecode

> Stack frames are destroyed when the method returns

> Stack memory is thread-specific

2. Object creation in memory

Heap is divided into:

1. Young Generation
    a. Eden Space
    b. Survivor Space S0
    c. Survivor Space S1

2. Old Generation (also called Tenured space)

Objects start in **Eden**. If they survive several Minor GCs, they move to Survivor spaces (S0 => S1) and increase their "age." Once an object survives enough GC cycles or if Survivor spaces run out of room, the JVM promotes it to the **Old Generation**.

> Singleton beans usually end up in Old Gen.
> Prototype/request-scoped beans may die young.

Reason for generation split:

Each generation has a size limit inside the heap, and we can configure it. Generations exist because most objects die quickly, and only a few live for a long time. By separating short-lived objects (Young Gen) from long-lived ones (Old Gen), Java makes garbage collection extremely fast and efficient. This is especially useful in Spring Boot, where beans live long (Old Gen) and request objects die fast (Young Gen).

Example:

```
@Service
class UserService {
    int count = 5;
}

public class MainApp {
    public static void main(String[] args) {

        UserService u1 = new UserService();  // long-lived object
        int x = 10;
        for (int i = 0; i < 3; i++) {
            UserService temp = new UserService(); // short-lived objects
        }
    }
}
```

1 long-lived object (**u1**)

3 short-lived objects (**temp inside loop**)

1. METHOD AREA (Only metadata):

Before any object is created, JVM loads class info

```
Class UserService:
    field: count
    annotation: @Service
    bytecode for methods
```

```
    constant pool: "UserService", "count".

Class MainApp:
    bytecode for main()
```

2. OBJECT CREATION (Eden (Young Gen)):

```
UserService u1 = new UserService();
```

What happens:

> JVM allocates memory in Eden

> Initializes fields (count = 5)

> Stores address of object in stack variable u1

HEAP (Eden):

```
UserService@0xA1
    count = 5
```

STACK:

```
main():
    u1 => 0xA1
    x = 10
```

3. TEMPORARY OBJECTS CREATED IN LOOP:

```
for (int i = 0; i < 3; i++) {
    UserService temp = new UserService();
```

```
}
```

Every time new is called, a new UserService object is created in Eden.

After loop (before GC):
HEAP:

```
Eden:
   u1              (long-lived)
   temp1
   temp2
   temp3
```

STACK (after loop ends):

```
main():
   u1 => 0xA1
   x = 10
```

4. Minor GC happens:
Minor GC cleans Eden, not Old Gen.

What happens during Minor GC:
> JVM scans Eden to find objects that are still referenced
> Moves surviving objects => Survivor S0
> Deletes unreferenced objects

Survivors:

> u1 (still referenced) => moved to Survivor S0

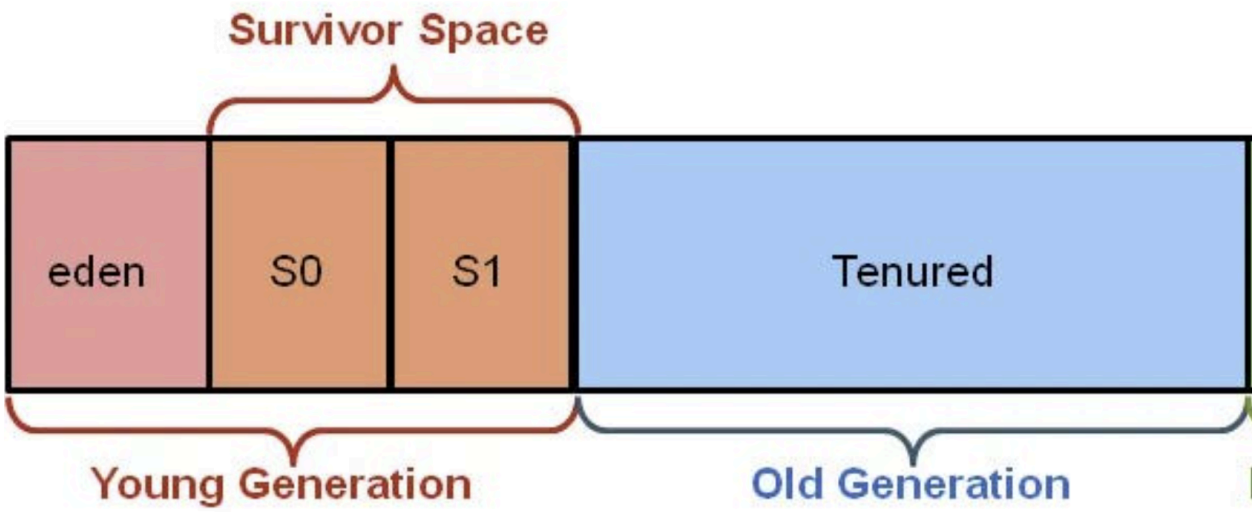
> temp1, temp2, temp3 => deleted

NEW MEMORY STATE:

```
Survivor S0:
   u1 (age = 1)

Eden:
   empty (after cleanup)
```

Objects start in Eden, survive a few GCs to move into Survivor spaces, and if they keep surviving (or Survivor space fills), the JVM promotes them into Old Gen. Class metadata stays in Method Area, and references stay in the stack.



3. Stack frames & method calls

A stack frame is a small memory block created every time a method is called.

Each frame contains:

1. Local variables
2. References to heap objects
3. Parameters
4. Return address (where to continue after method finishes)
5. Operand stack (temporary values the JVM uses for calculations)

When the method ends, the entire frame is deleted instantly.

Example:

```
class MathUtil {
    int add(int a, int b) {
        int sum = a + b;
        return sum;
    }
}
public class MainApp {
    public static void main(String[] args) {
```

```
        MathUtil m = new MathUtil();
        int result = m.add(10, 20);
        System.out.println(result);
    }
}
```

STEP 1 - main() STARTS => JVM CREATES FIRST STACK FRAME

STACK FRAME: main()

```
args => reference to String[]
m => reference to MathUtil object in heap
result = 30 (after method returns)
```

HEAP:

```
MathUtil@0xA1 => object
```

STEP 2 - m.add(10,20) IS CALLED

JVM creates another stack frame.

STACK FRAME: add()

```
a = 10
b = 20
sum = 30       (local variable)
return address => where to return inside main()
```

OPERAND STACK inside add():

```
10
20
30   (temporary calculation value)
```

STEP 3 - add() ENDS => its frame is deleted

Immediately removed. No memory leak ever.
Stack now has only the main() frame.

STEP 4 - main() receives the result

```
result = 30
```

The Method Area is created only once per JVM and never recreated for each method call or stack frame. Stack frames are temporary. Method Area is permanent during the entire JVM life.

Each method call creates a stack frame containing local variables, parameters, temporary values, and a return address.

Objects used inside the method live in the heap, but the references to them live in the stack.

When the method finishes, its frame is deleted instantly.

Spring Boot methods (controllers, services, repositories) all run with these stack frames, but the actual beans live in heap and their metadata lives in the Method Area.

4. Garbage Collection basics

Garbage Collection deletes objects that cannot be reached from **GC Roots**.

GC Roots include stack variables, static variables, running threads, and Spring's ApplicationContext.

Every new object starts in Eden, short-lived objects die there, while survivors move to Survivor spaces and eventually to Old Gen. Only objects not reachable from GC Roots are removed.

Spring beans are held by the ApplicationContext, which is a GC Root, so they are never deleted until the app shuts down.

GC roots

- GC roots are the only places the garbage collector starts looking.
- Everything reachable from them lives. Everything else dies

1. Stack variables (local variables):

What it is: Variables inside a running method.
Why leak happens: Method never finishes.

- Each running method has a stack frame.

- Local variables in that frame are GC roots while the method is executing.

```
void run() {
    Order order = new Order();
}
```

- order => stack => GC root
- Order => heap => alive

When run() finishes:

- Stack frame destroyed
- Root disappears
- Order becomes collectible

2. Static variables (class-level fields):

What it is: Class-level variables.
Why leak happens: Statics live forever

- Static fields live as long as the class is loaded
- Loaded class = permanent GC root.

```
class Cache {
    static List<byte[]> data = new ArrayList<>();
}
```

- Cache.data => GC root
- Everything inside data is immortal

3. Threads:

What it is: Running threads.
Why leak happens: Thread never stops. Live thread keeps everything it references.

4. ThreadLocal:

What it is: Data attached to a thread.
Why leak happens: Value not removed + thread reused.

5. Runtime Constant Pool

The Runtime Constant Pool is a table inside the Method Area that holds all symbolic and literal information a class needs such as

- names of fields,
- methods,
- classes,
- signatures,
- annotations,
- constants, and strings.

Each class has its own constant pool, and the JVM uses it to resolve method calls, load classes, run bytecode, and allow frameworks like Spring to read annotations and metadata. It is essential for linking, reflection, and class execution.

Runtime Constant Pool store:

1. String literals (“TEXT”)
2. Numeric constants (101)
3. Class names (“UserService”)
4. Field names (int age; // store age)
5. Method names & signatures
6. References to other classes/methods
7. Annotations (@Service)

Example:

```
@Service
class UserService {
    static String type = "SERVICE_LAYER";

    int age = 30;

    String getInfo() {
        return "User info";
    }
}
```

Runtime Constant Pool of UserService contains:

```
"UserService"
"type"
"age"
"SERVICE_LAYER"
"getInfo"
"User info"
"()Ljava/lang/String;"    // method signature
"@Service"
"java/lang/Object"
field reference: age
method reference: getInfo()
```

The JVM cannot store direct memory addresses in bytecode because Java is designed to be platform-independent, and real addresses depend on the CPU type, OS, memory layout, pointer size, and can change whenever

the garbage collector moves objects or when the JIT compiler optimizes code.

Instead, the JVM stores symbolic references, names of classes, fields, methods, and constants and resolves them at runtime based on the actual environment, class loader, and memory state.

This is exactly why the Runtime Constant Pool exists: it works as a lookup table containing all symbolic information the bytecode needs, allowing the JVM to translate symbolic names into real memory addresses safely and dynamically during execution. Without the constant pool, Java could not resolve methods, load classes, handle annotations, or maintain portability and safety.

Class Metadata vs Runtime constant pool:

Class Metadata Area:

The Class Metadata Area stores the structural definition of a class:

- what fields and methods exist,
- their types,
- modifiers, and
- how the class is laid out.

It does not store actual constant values or names as text.
It stores references (indexes) to them.

Runtime Constant Pool

The Runtime Constant Pool is a per-class table that stores all constants and symbolic information needed by bytecode at runtime, such as:

- String literals
- Class names
- Field names
- Method names
- Method signatures
- Symbolic references to fields and methods

This is where actual text and constants live.

```
class Calculator {

    int add(int a, int b) {
        return a + b;
    }
}
```

Runtime Constant Pool (actual data):

```
#1  Utf8        "Calculator"
#2  Utf8        "add"
#3  Utf8        "(II)I"
#4  Class       Calculator
#5  MethodRef   Calculator.add:(II)I
#6  Class       java/lang/Object
```

This is where strings and symbolic references are stored.

Class Metadata Area (structure + indexes):

Class:
  name_index => #1
  super_class => #6

Method:
  name_index => #2
  descriptor_index => #3
  bytecode => <address>

Method definitions (including bytecode) live in the Class Metadata Area, the Runtime Constant Pool only helps locate them.

The Runtime Constant Pool and Class Metadata are part of the Method Area.

6. Class.forName()

Class.forName() is a method that dynamically loads a class at runtime based on its name (as a string).

```
Class<?> c = Class.forName("com.example.User");
```

What happens internally?

> JVM asks Application ClassLoader to find User.class
> JVM loads the class into Method Area
> JVM initializes static fields
> JVM executes static blocks
> JVM returns a Class object representing this class

This is NOT object creation.
It is class-loading and class-initialization.

What Class.forName(String) actually does?

Locates the .class file using the caller's class loader (actually uses the caller's bootstrap behaviour historically, modern JVM uses the caller's class loader).

Loads the class into the JVM (reads bytecode and creates metadata in the Method Area).

Initializes the class: runs static initializers and initializes static fields.

Returns the single **java.lang.Class<?>** object that represents the class in this JVM/classloader.

Important: It does **not** create an instance of that class (not like new). It returns the Class object (metadata) only.

A Class object is created only once when the JVM loads a class, not every time we create an object. All instances of that class share this single Class object.
The Class object contains the runtime metadata of the class (methods, fields, annotations, bytecode, inheritance), allowing the JVM to allocate objects, call methods, perform reflection, load annotations, and support frameworks like Spring. It is the runtime blueprint that enables Java to execute classes dynamically and safely.

```
// 1. Basic
Class<?> c = Class.forName("com.example.MyClass");

// 2. Full control
Class<?> c = Class.forName(String name, boolean initialize, ClassLoader loader);
```

> initialize = true means run class initialization (static blocks, static field initializers) immediately.

> initialize = false means load the class (and link it) but do not run static initializers yet. (We can initialize later when the JVM needs it, for example, when we first access a static field or call a constructor.)

loader gives the ClassLoader to use.

Example:

> load with initializing

```
// A.java
public class A {
    static { System.out.println("A: static block"); }
}

// Main.java
public class Main {
    public static void main(String[] args) throws Exception {
        Class.forName("A");
        System.out.println("Main done");
    }
}
```

Output:

A: static block

Main done

> load without initializing

```
public class B {
    static { System.out.println("B: static"); }
}

public class Main {
    public static void main(String[] args) throws Exception {
```

```
        Class<?> c = Class.forName("B", false, Main.class.getClassLoader());
        System.out.println("B loaded but not initialized");
        // Now trigger initialization by accessing a static field or calling
Class.forName with initialize true:
        Class.forName("B", true, Main.class.getClassLoader());
    }
}
```

Output:

B loaded but not initialized

B: static

The first call loads but does not run a static block, the second call initializes it.

> Static block runs in these situations:

- When we create an object
- When we access a static field
- When we call a static method
- When we call Class.forName("A")

7. Lazy class loading

- JVM loads a class only when it is actually needed.

Example:

```
class A { }
class B { }

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

- Only Main loads.
- A and B are NOT loaded at all
- They will stay on disk as .class files until used

When does JVM load a class?

- When we create an object
- When we access a static field
- When we call a static method
- When we call Class.forName("A")

Spring eagerly initializes singleton beans by default.

But we have an option for lazy loading in spring (@Lazy).

8. Annotations

Marker annotations

Marker annotations are annotations that have no methods inside them. They exist only to mark a class, method, or field with meaning.

They’re basically empty tags the compiler or framework can check.

```
@interface MyMarker { }
```

This is a valid marker annotation.

The annotation is **not executed**. It’s **simply detected**.

Single-value Annotations

A single-value annotation is an annotation that has only one attribute, and that attribute is almost always named value().

Because of this special name, we don’t need to write “ value = “ when using it.

Example:

```
@interface MyTag {
    String value();   // only one attribute and it can be any type
}
```

Usage:

```
@MyTag("Hercules")
class TestClass { }
```

A single-value annotation stores its value in a method named value() inside the annotation type. The compiler puts this value into the class’s metadata in the Runtime Constant Pool.

At runtime, Spring reads this metadata using reflection (getAnnotation()) and retrieves the value through the value() method. Spring then uses this extracted value to configure mappings, dependency injection, or bean definitions.

Annotation element type must be one of these:

- Primitive types (int, boolean, etc.)
- String
- Class<?>
- Enum
- Another annotation
- Array of the above types

When Java compiles TestClass, it writes the annotation details into the Runtime Constant Pool:

- Annotation: MyTag
- Attribute: value
- Value: "Hercules"

At runtime, Java converts the annotation metadata into a proxy object that implements the annotation interface.

```
Class<TestClass> cls = TestClass.class;
MyTag tag = cls.getAnnotation(MyTag.class);
```

Now tag is a proxy object where:

```
tag.value();   // returns "Hercules"
```

Full (multi-value) Annotations

- These are annotations that have more than one attribute (method) inside them

Example:

```
@interface UserInfo {
   String name();
   int age();
}
```

Usage:

```
@UserInfo(name = "Hercules", age = 25)
class Person { }
```

```
   String name();
   int age();
```

They are annotation elements (annotation methods).
Inside an annotation, every method:

- has no body
- has no logic
- acts like a key/value pair
- is stored in the class's annotation metadata (not in heap, not in stack)

Each class stored in the Method Area includes its annotation metadata.

JVM reads annotation metadata from Method Area,
creates a fake “annotation object” (proxy) on the heap, and returns it.

Meta-Annotations

- These annotations used ON other annotations.
- They describe how an annotation behaves.

4 main meta-annotations:

1. @Target - Where can this annotation be used?

- It tells Java where our annotation is allowed:

```
@Target(ElementType.TYPE)       // class
@Target(ElementType.METHOD)     // method
@Target(ElementType.FIELD)      // field
```

- Example:

```
@Target(ElementType.TYPE)
@interface MyAnno { }
```

Now @MyAnno can be used only in classes

2. @Retention - How long does the annotation stay?

- RetentionPolicy.SOURCE
  - means the annotation exists only in code.
  - It is removed during compilation.
  - It never enters the .class file.
  - Example:

```
@Retention(RetentionPolicy.SOURCE)
```

```
@interface DebugInfo { }
```

- Visible only to the compiler
- Not available at runtime.
- Not present in .class file

- RetentionPolicy.CLASS (default)
  - Annotation is stored in the .class file
  - but JVM does NOT keep it at runtime.
  - Example

```
@Retention(RetentionPolicy.CLASS)
@interface LogInfo { }
```

    - Stored in class file
    - NOT accessible at runtime
    - Reflection cannot read it

- RetentionPolicy.RUNTIME:
  - Annotation is stored in the .class file AND JVM keeps it at runtime
  - Example:

```
@Retention(RetentionPolicy.RUNTIME)
@interface Service { }
```

    - Present in .class
    - JVM loads annotation
    - Spring can read it using reflection

- Frameworks can act based on it

3. @Documented - Should annotation appear in Javadoc?

- Just for documentation purposes
- Example:

```
@Documented
@interface MyAnno { }
```

4. @Inherited - Should child classes inherit the annotation?

- If a class has this annotation, subclasses automatically inherit it.
- Example

```
import java.lang.annotation.*;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@interface MyRole {
   String value();
}
```

Apply the annotation to a parent class:

```
@MyRole("ADMIN")
class Parent { }
```

Create a child class that extends the parent:

```
class Child extends Parent { }
```

Child has **no annotation**, but it will inherit the parent's annotation.

Repeatable Annotations

Repeatable annotations allow us to use **the same annotation multiple times** on the same class, method, or field.

- Every repeatable annotation needs
    - The annotation itself
    - A **container annotation** to hold the repeated values

Example:

```
@interface Schedules {
    Schedule[] value();
}

import java.lang.annotation.*;
@Repeatable(Schedules.class)
@interface Schedule {
    String day();
}
```

Use the repeatable annotation:

```
@Schedule(day = "Monday")
@Schedule(day = "Tuesday")
class Task { }
```

read them internally:

```
public class Demo {
    public static void main(String[] args) {
        Schedule[] schedules =
Task.class.getAnnotationsByType(Schedule.class);
        for (Schedule s : schedules) {
            System.out.println(s.day());
        }
    }
}
```

Type annotations

Type annotations allow us to place annotations directly on any use of a type - inside generics, arrays, casts, new expressions, or type declarations. They enable static analysis tools to detect null-safety issues, illegal assignments, and security problems. They are defined using @Target(TYPE_USE).

Example:

```
List<@NonNull String> names;
```

Each String inside the list must be non-null.
@NonNull is not a Java keyword; it is just a regular annotation defined by libraries like Spring or Lombok. The JVM doesn't enforce it

```
new @ReadOnly File("abc.txt");
```

Mark the created object's type with a special rule.

Custom Annotation Creation

Example 1:

UserInfo.java

```
import java.lang.annotation.*;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface UserInfo {
    String name();
    int age();
}
```

This annotation has two values => name and age.

Person.java

```
@UserInfo(name = "Hercules", age = 25)
public class Person {
}
```

Read annotation at runtime:

Main.java

```
public class Main {
    public static void main(String[] args) {
        UserInfo info = Person.class.getAnnotation(UserInfo.class);
```

```
        System.out.println("Name: " + info.name());
        System.out.println("Age: " + info.age());
    }
}
```

Example 2:

LogMe.java

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)

public @interface LogMe { }
```

Create a simple proxy that uses the annotation:

With interface (for JDK proxy):

```
public interface Payment {
    void pay();
}
public class PaymentService implements Payment {
    @LogMe
    public void pay() { System.out.println("Paying..."); }
}
```

Without interface (for CGLIB):

```
public class PaymentServiceNoIface {
    @LogMe
    public void pay() { System.out.println("Paying..."); }
}
```

Manual JDK Dynamic Proxy (requires interface):

```
import java.lang.reflect.*;

class LogHandler implements InvocationHandler {
    private final Object target;
    LogHandler(Object t){ this.target = t; }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
        if (method.isAnnotationPresent(LogMe.class))
System.out.println("LOG start " + method.getName());
        Object r = method.invoke(target, args);
        if (method.isAnnotationPresent(LogMe.class))
System.out.println("LOG end " + method.getName());
        return r;
    }
}

// usage:
Payment real = new PaymentService();
Payment proxy = (Payment) Proxy.newProxyInstance(
    Payment.class.getClassLoader(),
    new Class[]{Payment.class},
    new LogHandler(real)
);
```

```
proxy.pay();
```

Manual CGLIB Proxy (no interface)

```
import net.sf.cglib.proxy.*;

class CglibLogger implements MethodInterceptor {
    private final Object target;
    CglibLogger(Object t){ this.target = t; }

    @Override
    public Object intercept(Object obj, Method method, Object[] args,
MethodProxy proxy) throws Throwable {
        if (method.isAnnotationPresent(LogMe.class))
System.out.println("LOG start " + method.getName());
        Object r = proxy.invoke(target, args); // or proxy.invokeSuper(obj,
args) depending on setup
        if (method.isAnnotationPresent(LogMe.class))
System.out.println("LOG end " + method.getName());
        return r;
    }
}

// usage:
Enhancer e = new Enhancer();
e.setSuperclass(PaymentServiceNoIface.class);
e.setCallback(new CglibLogger(new PaymentServiceNoIface()));
PaymentServiceNoIface proxy = (PaymentServiceNoIface) e.create();
proxy.pay();
```

Java built-in annotations

JVM does nothing special for built-in annotations.

- The javac compiler is the one that reads them and takes actions.
- These annotations NEVER create proxies, never add behavior, and never run code.

Example:

```
@Override
public void run() { }
```

How Java handles it:

- Compiler checks: “Does this method really override a method from a superclass or interface?”
- If NOT, compilation error.

Runtime:

- Annotation is ignored - JVM does nothing with it.

Built-in annotations are generally SOURCE or CLASS retention

Examples:

Annotation	Retention	Meaning
@Override	SOURCE	Removed before .class generation
@SuppressWarnings	SOURCE	Removed before .class generation

@Deprecated	RUNTIME/CLASS	Stored but JVM doesn't treat it special

Only some (like @Deprecated) survive into the bytecode as metadata, but **JVM still ignores them**.

Java's built-in annotations like @Override, @Deprecated, @SuppressWarnings, and @FunctionalInterface are handled exclusively at compile time by the Java compiler.

They affect warnings, errors, documentation, and code validation. The JVM does not modify runtime behavior based on these annotations. They are never used to create proxies, intercept calls, or add logic, they exist only to help the compiler and IDE provide safety and clarity.

Spring built-in annotations

- Java built-in annotations are handled only by the compiler
- Spring annotations are handled at runtime by Spring's own code, not by Java

Spring reads annotations, interprets them, and executes logic because Spring is a **runtime framework**, not part of the JDK

Spring is a runtime framework

- Spring becomes alive only when the program is running, not before
- Spring does all of its work while our program is executing, not during compilation
- Meaning:
    - When we compile our code => Spring does nothing
    - When the JVM loads classes => Spring does nothing
    - When we start our Spring Boot app => Spring wakes up
    - Then Spring scans annotations, creates beans, and builds proxies
    - Then Spring controls how our code behaves at runtime

Flow:

- Java compiler (javac)
    - Handles things like @Override, @Deprecated
    - Stops errors before running code
    - Removes certain annotations completely

- Spring (runtime)
    - java -jar myapp.jar

- Then Spring begins:
    - Scanning the classes
    - Finding @Service, @Component, @RestController
    - Injecting dependencies (@Autowired)
    - Creating proxies for AOP (@Transactional, @Cacheable)
    - Setting up controllers (@RequestMapping)
    - Scheduling tasks (@Scheduled)
    - Starting background threads (@Async)
    - Managing lifecycle (@PostConstruct)

Spring does NOT let JVM handle annotations.
Spring writes its own code to read, interpret, and execute based on them.

Process:
1. Spring scans our classes:
Using:

- ClassPathScanner
- MetadataReader
- AnnotationUtils

Spring looks for annotations like:

- @Component
- @Service
- @Controller
- @Bean
- @Configuration
- @Transactional
- @Autowired

These annotations are simply metadata inside the Method Area.
Spring reads them using reflection.

2. Spring builds Bean Definitions:
When Spring sees:

```
@Service
public class UserService { }
```

Spring creates a bean definition::

```
Bean name: userService
Bean type: UserService
Scope: singleton
Wiring: automatic
Need a proxy? maybe
```

Java does NOT do this.
Spring does this manually with code.

3. Spring creates the bean instance:
Spring decides:

- How to instantiate it
- Whether to wrap it
- How to inject dependencies

Example: @Autowired

Spring sees:

```
@Autowired
private OrderService service;
```

Spring:

- looks at bean definitions
- finds OrderService
- injects it via constructor or field
- builds the final bean

Java never touches this, it's all Spring logic.

4. Spring may decide to wrap the bean with a proxy:

Example: @Transactional

```
@Transactional
public void save() { }
```

Spring:

- Detects the annotation
- Creates a proxy around the object
- Proxy intercepts our method call

Only Spring does, via:

- JDK Dynamic Proxy
- or CGLIB Proxy

This is runtime behavior

Feature	Java built-in annotations	Spring annotations
Who handles it?	Compiler	Spring framework
Time of effect	Compile-time	Runtime
Adds behavior?	No	Yes (proxy, AOP, DI)
Uses reflection?	No	Yes
Uses proxies?	No	Yes
Control beans/transactions?	No	Yes
JVM understanding?	Ignored	JVM just holds metadata; Spring interprets

9. Reflection

- Reflection = reading and controlling classes at runtime, not at compile time.
- Program learning about itself while running

1. Class
    a. Every loaded Java class has a Class object stored in the Method Area.
    b. Get it like:
        i. **Class<?> cls = User.class;**
        ii. **Class<?> cls = Class.forName("com.app.User");**

This Class object is the entry point for reflection.

2. Method
    a. Represents a method of a class.
    b. Method m = cls.getMethod("login");
    c. Method object lets us:
        - call the method
        - read annotations
        - check modifiers
        - read parameters

This is how Spring calls controller methods, bean methods, config methods, etc

3. Field
    a. Represents a class variable.
    b. Field f = cls.getDeclaredField("name");
    c. Field lets us:
        - read value
        - write value
        - break private access using setAccessible(true)

Spring uses this to inject dependencies into @Autowired fields.

4. Constructor
    a. Represents class constructors
    b. Constructor<?> c = cls.getConstructor(String.class);
    c. Spring uses constructor reflection when doing:
        - constructor injection
        - creating beans without new
        - creating proxies

Example:

```
public class User {
    private String name;

    public User(String name) {
        this.name = name;
    }

    private void greet() {
        System.out.println("Hello, " + name);
    }
```

```
}
```

Reflection operations:

```
Class<?> cls = User.class;

// 1. Constructor
Constructor<?> cons = cls.getConstructor(String.class);
Object obj = cons.newInstance("Hercules");

// 2. Field
Field field = cls.getDeclaredField("name");
field.setAccessible(true);          // access private
field.set(obj, "Zeus");

// 3. Method
Method m = cls.getDeclaredMethod("greet");
m.setAccessible(true);              // access private
m.invoke(obj);
```

Reflection lets Java inspect and manipulate a class at runtime. Class gives the structure, Constructor creates objects, Field accesses variables (**even private**), and Method calls methods (even private). Spring uses reflection everywhere: scanning beans, injecting dependencies, handling controllers, invoking @Bean methods, reading annotations, and applying AOP.

Bean - Short intro:

A bean is simply an object created, managed, and controlled by the Spring IoC container.

If Spring creates it => it's a bean.
If we create it with new => it is not a bean.

Examples of beans:

- Services (@Service)
- Controllers (@RestController)
- Repositories (@Repository)
- Configuration classes (@Configuration)
- Objects created inside @Bean methods

```
@Service
public class UserService { }
```

Spring sees @Service => creates an object of UserService.
That object is a bean.

A bean method is a method inside an @Configuration class that is annotated with @Bean.

```
@Configuration
public class AppConfig {

    @Bean
    public UserService userService() {
        return new UserService();
    }
}
```

When Spring calls this method at runtime:

- It creates an object (new UserService())
- Registers it as a bean
- Manages its lifecycle
- Injects it into dependent classes

So bean methods create beans.

Why do bean methods exist?
Because sometimes we need more control over how an object is created:

- building objects with parameters
- reading configs
- using factories
- creating third-party objects (e.g. ObjectMapper, DataSource, RestTemplate)

Spring can't scan them, so we tell Spring explicitly:

```
@Bean
public ObjectMapper mapper() {
   return new ObjectMapper();
}
```

Example showing both:

```
@Service
public class PaymentService { }
```

Spring automatically creates this bean.

```
@Configuration
public class MyConfig {
    @Bean
    public PaymentService payment() {
        return new PaymentService();   // bean method
    }
}
```

Both objects are beans - one via scanning, one via a bean method.

A bean is any object created and managed by Spring's IoC container. A bean method is a method inside an @Configuration class marked with @Bean, and Spring calls that method to create a bean. Bean methods give us full control over object creation, while component scanning automatically detects beans via annotations like @Service or @Controller.

getDeclaredMethods()

It returns all methods declared inside a class, including:

- private methods
- protected methods
- public methods
- default methods

But NOT inherited methods from parent classes.

```
Class<?> cls = UserService.class;
Method[] methods = cls.getDeclaredMethods();

for (Method m : methods) {
    System.out.println(m.getName());
}
```

Scanning methods with annotations

```
public class User {

    @LogMe
    public void pay() {}

    private void hidden() {}
}
```

```
Class<?> cls = User.class;

for (Method m : cls.getDeclaredMethods()) {
    if (m.isAnnotationPresent(LogMe.class)) {
        System.out.println("Found annotated method: " + m.getName());
    }
}
```

getDeclaredMethods() returns all methods declared in a class - public, private, protected without including inherited methods. Spring uses it to detect @Bean methods, controller endpoints, event handlers, AOP

annotations like @Transactional, and more. It allows Spring to fully understand what methods exist in a class and how to manage them at runtime.

setAccessible(true)

- setAccessible(true) allows Java to break access rules.
- Normally we cannot access:
    - private fields
    - private methods
    - private constructors
- But with reflection, we can bypass Java's access control

```
field.setAccessible(true);
method.setAccessible(true);
constructor.setAccessible(true);
```

Why does this exist?

Because frameworks like Spring need to:

- call methods even if they are private
- set fields even if they are private
- create objects even if the constructor is private

Example:

```
public class User {
    private String name = "Hercules";

    private void greet() {
        System.out.println("Hello " + name);
    }
}
```

```
Class<?> cls = User.class;
User obj = new User();

// Access private field
Field f = cls.getDeclaredField("name");
f.setAccessible(true); // important
f.set(obj, "Zeus");

// Access private method
Method m = cls.getDeclaredMethod("greet");
m.setAccessible(true); // important
m.invoke(obj);
```

What does setAccessible(true) actually do internally?

- It flips a permission flag in JVM:
- allowAccess = true
- Then Java doesn't check if the field/method is private.

Creating objects dynamically

- Reflection lets us create objects without using ‘new’ keyword.
- This is exactly how Spring creates all our beans internally.

Basic ways to create objects dynamically:

1. Using Class.newInstance()
2. Using Constructor.newInstance()
3. Using Spring’s BeanUtils.instantiateClass()

Creating object with default constructor:

```
Class<?> cls = User.class;
Object obj = cls.newInstance();
```

```
Constructor<?> c = cls.getDeclaredConstructor();
Object obj = c.newInstance();
```

Creating object using parameterized constructor:

```
public class User {
    private String name;

    public User(String name) {
        this.name = name;
    }
}
```

```
Class<?> cls = User.class;
Constructor<?> cons = cls.getConstructor(String.class);
Object obj = cons.newInstance("Hercules");
```

Creating object in dependency injection:

```
@Component
class OrderService {
    public OrderService(UserService user) { }
}
```

Spring creates the bean using:

```
Constructor<?> cons =
OrderService.class.getConstructor(UserService.class);
OrderService obj = (OrderService)
cons.newInstance(userServiceInstance);
```

10. Proxy Object

- A proxy object is a dynamically created object that stands in place of the real object.

It:

- Looks like the real object
- Has the same methods
- Sits between the caller and the real object
- Can execute extra logic before and/or after calling the real method

Logical flow:
User => Proxy => Real Object

Why Proxies Exist

- Java does not allow interception of method calls directly.

Spring needs to run logic like:

- start / commit / rollback transactions
- caching
- logging
- security checks
- async execution
- HTTP routing for controllers

Because Java cannot intercept method calls directly, Spring wraps the object inside a proxy.

Without a proxy, Spring cannot execute logic around method calls.

When Spring Creates A Proxy

Spring automatically creates proxies when it sees:

- @Transactional
- @Cacheable
- @Async
- AOP advices
- @Controller / @RestController route handling

What Actually Happens On A Method Call

```
@Service
public class UserService {
        public void save() {
                System.out.println("Saving user");
        }
}

....
@Autowired
UserService myService;
....
myService.save();
....
```

Real execution flow:

myService (proxy).save()

=> interceptor logic (transaction / cache / logging / etc.)
=> realService.save()

This is why adding annotations changes runtime behavior.

How Method Interception Works

When calling:
myService.save();

Actual flow:
proxy.save()
=> check annotations
=> run interceptors
=> call real save() on target object
=> optionally modify return value
=> return result to caller

Controller Flow With Proxy

```
@RestController
public class UserController {
	@Transactional
	public String saveUser() {
		return "saved";
	}
}
```

Request flow:
Client HTTP request
=> Spring MVC resolves controller
=> Spring registers a PROXY as the bean
=> proxy intercepts the method call
=> proxy resolves annotations
=> before logic
=> targetObject.method()
=> after logic
=> response returned to client

Actual call path:
Controller => Proxy => Target Object

What Spring Stores In The Container

Spring creates TWO objects:

1. Real object (target)
2. Proxy object (bean)

Only the PROXY is stored in the Spring container.

The real object is hidden inside the proxy.

@Autowired injects:
userService => proxy
NOT the real object

Internal Structure

```
ProxyObject {
        private targetObject;
        invoke(...) // interceptor logic
}
```

The application only knows about the proxy.

Proxy Creation Flow

1. Spring creates the real object
2. Spring scans for annotations (@Transactional, @Async, @Cacheable, etc.)
3. If interception is needed => Spring creates a proxy
4. Proxy wraps the real object
5. Proxy becomes the bean
6. Real object stays hidden inside the proxy

Even if only ONE method has an annotation:

- Spring still creates ONE proxy for the entire class
- Every method call goes through the proxy
- Extra logic runs ONLY for annotated methods

EXAMPLE

```
@Service
public class UserService {
        public void a() { }

        @Transactional
        public void b() { }

        public void c() { }
}
```

Spring registers:

Proxy(UserService)

        => target = real UserService

Call a():

proxy.a()

=> no annotation

=> directly calls real.a()

Call b():

proxy.b()

=> begin transaction

=> real.b()

=> commit transaction

Multiple Annotations

If a method has multiple annotations, Spring builds an interceptor chain.

Example:

```
@Cacheable
@Transactional
public String compute() {}
```

Execution order:

cache interceptor

=> transaction interceptor

=> real method

<= transaction completion

<= cache write

Self-Invocation Problem

Example:

```
@Service
public class UserService {
        public void a() { }

        @Transactional
        public void b() {
            a();  // internal call
        }
```

```
}
```

Output:
[TX Begin]
Method B
Method A (NO TRANSACTION)
[TX Commit]

Reason:

- userService.b() => proxy.b()
- proxy forwards to realObject.b()
- inside b(), this.a() is called
- this = real object, NOT proxy
- proxy is bypassed

AOP works ONLY for external method calls.

How Spring Chooses Proxy Type

Spring decision rules:

1. If proxyTargetClass = true
   => use CGLIB (class-based proxy)
2. Else if class implements an interface
   => use JDK dynamic proxy
3. Else
   => use CGLIB

Proxy Implementations

1. JDK Dynamic Proxy
    - Uses java.lang.reflect.Proxy
    - Implements InvocationHandler
2. CGLIB Proxy
    - Creates subclass at runtime
    - Uses MethodInterceptor

11. JDK Dynamic Proxy (Interface-Based)

InvocationHandler

- An InvocationHandler is a single method (invoke) that will run every time any method is called on the proxy.
    - The proxy **doesn't really have method logic**
    - It forwards everything to InvocationHandler.invoke().

Example:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// --- 1) define custom annotations ---

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface Tx {}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface LogMe {}
```

```
// --- 2) service interface ---

interface Service {
    void pay();          // no annotation
    void save();         // annotated with @Tx
    String compute(int x); // annotated with @LogMe
```

```
}
```

```
// --- 3) implementation (annotations on implementation methods) ---

class ServiceImpl implements Service {
    @Override
    public void pay() {
        System.out.println("ServiceImpl.pay(): doing payment work");
    }

    @Override
    @Tx
    public void save() {
        System.out.println("ServiceImpl.save(): persisting data");
    }

    @Override
    @LogMe
    public String compute(int x) {
        System.out.println("ServiceImpl.compute(): computing for " + x);
        return "result:" + (x * 2);
    }

    // self-invocation
    public void outer() {
        System.out.println("ServiceImpl.outer() calls save() internally:");
        save();             // this is a direct call on the real object (bypasses
proxy)
    }
}
```

```
// --- 4) InvocationHandler that inspects annotations on the
implementation method ---
class MultiBehaviorHandler implements InvocationHandler {

    private final Object target;

    public MultiBehaviorHandler(Object target) {
        this.target = target;
    }

    private Method getImplMethod(Method interfaceMethod) throws
NoSuchMethodException {
        // find method on implementation class with same name and
parameter types

        return target.getClass().getMethod(interfaceMethod.getName(),
interfaceMethod.getParameterTypes());
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
        // Handler ALWAYS runs for every method call on the proxy
        Method implMethod = getImplMethod(method);

        // Example behavior: if @Tx present -> wrap with transaction-like
prints
        if (implMethod.isAnnotationPresent(Tx.class)) {
            System.out.println("[HANDLER] TX begin for " +
```

```
method.getName());
            try {
                Object res = method.invoke(target, args);
                System.out.println("[HANDLER] TX commit for " +
method.getName());
                return res;
            } catch (InvocationTargetException ite) {
                System.out.println("[HANDLER] TX rollback for " +
method.getName());
                throw ite.getTargetException();
            }
        }

        // If @LogMe present -> log before/after
        if (implMethod.isAnnotationPresent(LogMe.class)) {
            System.out.println("[HANDLER] LOG start " +
method.getName());
            Object res = method.invoke(target, args);
            System.out.println("[HANDLER] LOG end " + method.getName());
            return res;
        }

        // No annotation -> plain passthrough
        return method.invoke(target, args);
    }
}
```

```
// --- 5) main demo —

public class ProxyDemo {
    public static void main(String[] args) {
```

```
        Service real = new ServiceImpl();

        Service proxy = (Service) Proxy.newProxyInstance(
                Service.class.getClassLoader(),
                new Class[]{Service.class},
                new MultiBehaviorHandler(real)
        );

        System.out.println("=== call proxy.pay() (no annotation) ===");
        proxy.pay();

        System.out.println("\n=== call proxy.save() (@Tx on impl) ===");
        proxy.save();

        System.out.println("\n=== call proxy.compute(5) (@LogMe on impl)
===");
        String r = proxy.compute(5);
        System.out.println("Returned: " + r);

        System.out.println("\n=== call real.outer() directly (shows
self-invocation bypass) ===");
        // calling on real object demonstrates internal call not intercepted
        real.outer();
    }
}
```

Output:

```
=== call proxy.pay() (no annotation) ===
ServiceImpl.pay(): doing payment work

=== call proxy.save() (@Tx on impl) ===
```

```
[HANDLER] TX begin for save
ServiceImpl.save(): persisting data
[HANDLER] TX commit for save

=== call proxy.compute(5) (@LogMe on impl) ===
[HANDLER] LOG start compute
ServiceImpl.compute(): computing for 5
[HANDLER] LOG end compute
Returned: result:10

=== call real.outer() directly (shows self-invocation bypass) ===
ServiceImpl.outer() calls save() internally:
ServiceImpl.save(): persisting data
```

- InvocationHandler.invoke(...) is called for every method invoked on the proxy (annotated or not).

- The handler inspects the implementation method (via target.getClass().getMethod(...)) to decide what behavior to add. That makes it work whether annotations are declared on the interface or on the implementation.

- Non-annotated methods still go through the proxy, but the handler usually just forwards them (no extra logic).

- Annotated methods trigger extra logic in the handler (transactions, logging, caching, etc.).

- Internal calls inside the real object (this.save() from outer()) bypass the proxy entirely - we saw that when real.outer() invoked save() directly (no handler prints).

Proxy.newProxyInstance()

- Proxy.newProxyInstance(...) is the factory method that creates a runtime proxy class and an instance that implements one or more interfaces we supply. The returned object delegates every method call to the provided InvocationHandler.invoke(...).

1. JVM generates a proxy class (e.g., com.sun.proxy.$Proxy123) that implements the provided interfaces.
2. Methods in the generated proxy call InvocationHandler.invoke(proxy, method, args).
3. We must provide a handler, the handler usually holds a reference to the real target and the advice chain.
4. The proxy instance is allocated and returned to callers.

```
Object Proxy.newProxyInstance(
    ClassLoader loader,
    Class<?>[] interfaces,
    InvocationHandler h
)
```

loader : the ClassLoader that will define the generated proxy class. Usually use the interface class loader (e.g. Service.class.getClassLoader()), or the current thread context class loader. If wrong, we will get ClassCastException or IllegalArgumentException.

interfaces : array of Class objects for the interfaces our proxy must implement. The generated proxy class implements ALL these interfaces.

h : our InvocationHandler that receives invoke(proxy, method, args) for every method call. method is the Method object representing the interface method invoked.

```
Service target = new ServiceImpl();

Service proxy = (Service) Proxy.newProxyInstance(
    Service.class.getClassLoader(),
    new Class[]{Service.class},
    (proxyObj, method, args) -> {
        System.out.println("before " + method.getName());
        Object res = method.invoke(target, args); // forward to real object
        System.out.println("after " + method.getName());
        return res;
    }
);
```

“The generated proxy class implements only interfaces”

```
interface Service {
    void pay();       // only 1 method in interface
}

class ServiceImpl implements Service {
    public void pay() {}
    public void hello() {}  // NOT in interface
    public void bye() {}    // NOT in interface
```

```
}
```

```
Service proxy = (Service) Proxy.newProxyInstance(
        Service.class.getClassLoader(),
        new Class[]{Service.class},
        handler
);
```

The generated proxy class looks like this inside JVM:

```
class $Proxy123 implements Service {   // ONLY implements the interface

    void pay() {
        handler.invoke(... pay ...);
    }

    // Notice: NO hello(), NO bye()
}
```

- This proxy does NOT have hello() or bye(),
- because they were not part of the interface.
- This is why JDK proxy cannot proxy methods that are not in interfaces.

```
proxy.hello();   //  compile-time error
proxy.bye();     //  compile-time error
```

Because the proxy doesn't have these methods.

So what happens to the real class's extra methods?
They still exist in the real object, but we can't call them through the proxy.

Non-interface methods:

- If JDK proxy is used: proxy only exposes interface methods. Methods declared only on the implementation class are not available via the proxy reference and cannot be intercepted by that JDK proxy. They still exist on the real target (and can be called internally or via direct access to the target), but external callers with only the proxy reference cannot call them.

In-depth Example:

```
// ServiceApi.java
public interface ServiceApi {
    void pay();
    void audit();
}

// ServiceImpl.java
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class ServiceImpl implements ServiceApi {

    @Override
    @Transactional          // Spring will create an advisor for this
```

```
    public void pay() {
        System.out.println("REAL: pay()");
    }

    @Override
    public void audit() {
        System.out.println("REAL: audit()");
    }

    // internal call example:
    public void doBoth() {
        pay();   // self-invocation: bypasses proxy
        audit();
    }
}
```

- Spring scans ServiceImpl and sees @Transactional on pay() => it builds an Advisor for that method (advisor contains a pointcut + transaction advice).
- Spring collects all advisors for the bean and forms an AdvisedSupport (target + advisors + proxied interfaces + proxyTargetClass flag).
- Decision: bean implements an interface and proxyTargetClass=false => use JDK proxy.
- Spring creates the handler: new JdkDynamicAopProxy(advisedSupport). This object implements InvocationHandler and holds:

    - reference to the target (or TargetSource),
    - advisor list / interceptor chain builder.

- Spring calls Proxy.newProxyInstance(classLoader, proxiedInterfaces, handler) and stores the returned proxy object as the bean in the container. External injection gets that proxy.

Simplified pseudo-code of what Spring executes internally:

```
AdvisedSupport advised = new AdvisedSupport();
advised.setTarget(new ServiceImpl());
advised.setInterfaces(ServiceApi.class);
advised.addAdvisor(transactionalAdvisorForPay);

InvocationHandler handler = new JdkDynamicAopProxy(advised);

Object proxy = Proxy.newProxyInstance(
    ServiceApi.class.getClassLoader(),
    new Class[]{ ServiceApi.class },
    handler
);

// store "proxy" as the bean; inject proxy to callers
```

JdkDynamicAopProxy is Spring's handler. It will receive every call on the generated proxy and run the advisor chain for matching methods.

```
// PSEUDO: com.sun.proxy.$Proxy123 implements ServiceApi

public final class $Proxy123 implements ServiceApi, java.io.Serializable {
    private final InvocationHandler h;
```

```
    public $Proxy123(InvocationHandler h) { this.h = h; }

    public void pay() {
        try {
            // obtain Method object for interface method ServiceApi.pay()
            Method m = ServiceApi.class.getMethod("pay");
            // forward the call to the handler
            h.invoke(this, m, null);
        } catch (Throwable t) { throw wrap(t); }
    }

    public void audit() {
        try {
            Method m = ServiceApi.class.getMethod("audit");
            h.invoke(this, m, null);
        } catch (Throwable t) { throw wrap(t); }
    }

    // equals/hashCode/toString also forward to handler
}
```

- The proxy implements only ServiceApi (so only pay() and audit() are present).
- Each proxy method calls h.invoke(this, method, args).
- h is the JdkDynamicAopProxy instance Spring created.

12. CGLIB Proxy (Class-Based)

If our class has no interface, Spring cannot use JDK proxy.
So Spring uses CGLIB.

CGLIB works by creating a new subclass of our class during Spring startup. It overrides our methods so that calls first go into a MethodInterceptor, where Spring applies AOP behaviors like @Transactional, @Cacheable, @Loggable, etc.

This approach does not need interfaces, but cannot proxy final classes or final methods.

While JDK proxies route calls using InvocationHandler.invoke(), CGLIB proxies route calls through MethodInterceptor.intercept() using invokeSuper() to reach the real method.

Final class => cannot proxy at all
Final method => method cannot be intercepted

- CGLIB does NOT wrap our object. It creates a new class that extends our class.
- CGLIB can intercept non-interface methods, which JDK cannot

How CGLIB interception works

```
class OrderService$$SpringCGLIB$$123abc extends OrderService {

    @Override
    void place() {
        // Spring's interceptor logic
        methodInterceptor.intercept(this, placeMethod, args, methodProxy);
    }
}
```

This new class replaces our bean in the Spring container.
So our actual bean is the CGLIB subclass, not the real one.

orderService.place();

Spring calls the CGLIB subclass:

OrderService$$CGLIB.place()

Flow:
User => CGLIB subclass => MethodInterceptor => real method

Example:

```
public class PaymentService {

    @Loggable
```

```
    @Transactional
    public void pay() {
        System.out.println("REAL: pay()");
    }

    @Transactional
    public void refund() {
        System.out.println("REAL: refund()");
    }

    public void audit() {
        System.out.println("REAL: audit()");
    }
}
```

```
public @interface Loggable {}
```

```
@Aspect
@Component
public class LoggingAspect {
    @Around("@annotation(Loggable)")
    public Object log(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("[LOG] before");
        Object r = pjp.proceed();
        System.out.println("[LOG] after");
        return r;
    }
}
```

1. READING ANNOTATIONS - REFLECTION

Spring does this:

```
Method payMethod = PaymentService.class.getMethod("pay");
Annotation[] annotations = payMethod.getAnnotations();
```

Results:

pay() => [Loggable, Transactional]

refund() => [Transactional]

audit() => []

Spring knows which annotation is on which method

2. CONVERT ANNOTATIONS => INTERCEPTOR OBJECTS

1. @Loggable

@Around aspect creates a helper class inside Spring.

Spring converts the log() method into a simple MethodInterceptor:

```
class LogInterceptor {
    invoke(invocation) {
        print "[LOG] before"
        result = invocation.proceed()   // call next step / real method
        print "[LOG] after"
        return result;
    }
}
```

2. @Transactional

Spring creates its built-in:

```
class TransactionInterceptor {
    invoke(invocation) {
        print "[TX] begin"
        result = invocation.proceed()
        print "[TX] commit"
        return result;
    }
}
```

3. Methods with NO annotation

CGLIB uses:

NoOp => does nothing, calls real method directly

3. BUILD METHOD => INTERCEPTOR LIST MAPPING (REAL MAP)

Spring builds:

```
Map<Method, List<MethodInterceptor>> m = new HashMap<>();
```

Method	Interceptors
pay()	[LogInterceptor, TransactionInterceptor]
refund()	[TransactionInterceptor]
audit()	[]

4. BUILD THE CGLIB PROXY

Spring uses CGLIB, which requires:

- a Callback array
- a CallbackFilter

These two control “which method uses which interceptor.”

CALLBACK ARRAY (stored inside proxy object):
Callback array contains only two objects:

```
callbackArray[0] = DynamicInterceptor   // handles pay + refund
callbackArray[1] = NoOp                 // handles audit
```

Inside DynamicInterceptor, Spring stores the method => interceptor list map from Phase 3.

This means:

```
DynamicInterceptor knows:
  pay() => [log, tx]
  refund() => [tx]
```

So only ONE callback object is needed for all advised methods.

CALLBACK FILTER:
This small object returns an index for each method:

```
pay()    => return 0
refund() => return 0
audit()  => return 1
```

WHAT CGLIB GENERATES:

```
class PaymentService$$CGLIB extends PaymentService {

   public void pay() {
      callbackArray[0].intercept(this, payMethod, args, methodProxy);
   }

   public void refund() {
      callbackArray[0].intercept(this, refundMethod, args, methodProxy);
   }

   public void audit() {
      callbackArray[1].intercept(this, auditMethod, args, methodProxy);
   }
}
```

- this: This is the proxy object, NOT the real PaymentService
- payMethod: This is a java.lang.reflect.Method object.
    - It represents: public void PaymentService.pay()
    - Why is it needed?
        - DynamicAdvisedInterceptor uses this as a key in its internal map

- args: This is an Object array containing the method arguments.
- methodProxy:
    - It is a CGLIB object that knows:

- how to call the real method (PaymentService.pay())

RUNTIME METHOD CALLS:

CALL 1: pay()

```
proxy.pay()
=> callbackFilter => index 0
=> callbackArray[0] = DynamicInterceptor
```

DynamicInterceptor looks up the method:

```
interceptorsFor(pay) = [log, tx]
```

Execution:

```
log.invoke()
   |
   tx.invoke()
       |
       methodProxy.invokeSuper() => REAL pay()
```

CALL 2: refund()

```
proxy.refund()
=> filter returns index 0
=> callbackArray[0] called
```

DynamicInterceptor:

```
refund() => [tx]
```

```
tx.invoke()
  |
  REAL refund()
```

CALL 3: audit()

```
proxy.audit()
=> filter returns index 1
=> callbackArray[1] = NoOp
```

@Around aspect is read by Spring, converted into an Advice object, wrapped into an Advisor, then adapted into a MethodInterceptor. Spring puts this MethodInterceptor into a method=>interceptor map. DynamicAdvisedInterceptor is the callback that reads this map at runtime. When the proxy calls callbackArray[0].intercept(...), DynamicAdvisedInterceptor fetches the interceptor chain for the method and executes each interceptor, ending with invokeSuper() to call the real method.

How an Aspect becomes an Interceptor

```
@Aspect
@Component
public class LoggingAspect {
```

```
    @Around("@annotation(Loggable)")
    public Object log(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("[LOG] before");
        Object r = pjp.proceed();
        System.out.println("[LOG] after");
        return r;
    }
}
```

Spring internally performs 3 steps:

1. Step 1 - Spring creates an “Advice” object from our @Around method

   Spring reads our method using reflection.
   It creates:

```
AspectJAroundAdvice advice
```

This object stores:

- the method log(...)
- the instance of our LoggingAspect
- the pointcut @annotation(Loggable)

2. Step 2 - Spring puts this advice inside an Advisor object

```
AspectJPointcutAdvisor advisor
    |── pointcut = @annotation(Loggable)
    |── advice   = AspectJAroundAdvice
```

Advisor = a container that holds “WHERE + WHAT”.

3. Step 3 - Spring converts this advice => MethodInterceptor
   Spring MUST convert advice to the common AOP interface:

```
public interface MethodInterceptor {
   Object invoke(MethodInvocation invocation) throws Throwable;
}
```

So Spring creates:

```
LogInterceptor (adapter around our aspect method)
```

```
public class LogInterceptor implements MethodInterceptor {

   private final Method aspectMethod;       // log(...)
   private final Object aspectInstance;     // new LoggingAspect()

   public LogInterceptor(Method aspectMethod, Object aspectInstance) {
      this.aspectMethod = aspectMethod;
      this.aspectInstance = aspectInstance;
   }

   @Override
   public Object invoke(MethodInvocation invocation) throws Throwable {

      System.out.println("[LOG] before");

      // call next interceptor or real method
```

```
        Object result = invocation.proceed();

        System.out.println("[LOG] after");

        return result;
    }
}
```

Inside LogInterceptor.invoke():

- call our log() method

So:

Aspect => Advice => Advisor => MethodInterceptor

That's how our @Around method becomes an interceptor

What is DynamicAdvisedInterceptor?

- DynamicAdvisedInterceptor is created by Spring's CGLIB proxy mechanism.
- It is the object that actually executes our interceptors in the correct order and finally calls the real method
- It lives inside - callbackArray[0]

WHY DynamicAdvisedInterceptor EXISTS?

Because:

- A method might have 0, 1, or 10 interceptors
- Spring must pick the correct interceptors at runtime
- Then run them one by one

- Then call the real method

So Spring needs one object to do this job.
That's DynamicAdvisedInterceptor.

What is inside DynamicAdvisedInterceptor?

```
public class DynamicAdvisedInterceptor implements
MethodInterceptor {

   private final Object target;     // original PaymentService
   private final AdvisedSupport config; // contains method =>
interceptor map

   @Override
   public Object intercept(
         Object proxy,
         Method method,
         Object[] args,
         MethodProxy methodProxy) throws Throwable {

      // get the interceptor list for the method
      List<Object> chain = config.getInterceptors(method);

      if (chain.isEmpty()) {
         // no advice => call real method
         return methodProxy.invokeSuper(proxy, args);
      }

      // wrap everything in MethodInvocation
      MethodInvocation invocation =
            new ReflectiveMethodInvocation(target, method, args,
```

```
chain, methodProxy);

        return invocation.proceed();
    }
}
```

target - This is the REAL object: new PaymentService()

config.getInterceptors(method) -

- This gives:
    - For pay(): [LogInterceptor, TransactionInterceptor]
    - For refund(): [TransactionInterceptor]
    - For audit(): []

If the list is empty => directly call real method

- methodProxy.invokeSuper(proxy, args)

If the list is not empty => build MethodInvocation

- MethodInvocation stores:
- which interceptor is next
- which method to call
- which object is the real target

Flow:

```
proxy.pay()
  |
DynamicAdvisedInterceptor.intercept()
```

```
 |
getInterceptors(pay) => [log, tx]
 |
MethodInvocation.proceed()
 |
log.invoke()
   |
   tx.invoke()
      |
      REAL pay()
      |
   tx.after
|
log.after
|
return
```

@Around aspect becomes a LogInterceptor (a MethodInterceptor). Spring also creates a TransactionInterceptor for @Transactional. DynamicAdvisedInterceptor holds the real PaymentService object and the method=>interceptor mapping. When the proxy calls callbackArray[0].intercept(), DynamicAdvisedInterceptor looks up the correct interceptor chain for the method and executes each interceptor in order via MethodInvocation. If there are no interceptors, it calls the real method directly using invokeSuper(). This is the exact engine behind Spring AOP proxying.

13. Cross-cutting concern

- A cross-cutting concern is any logic that must run for multiple unrelated methods, without rewriting that logic in each method.

```
public class PaymentService {

    public void pay() {
        System.out.println("[LOG] Before method");   // repeated
        System.out.println("Pay executed");
    }

    public void refund() {
        System.out.println("[LOG] Before method");   // repeated
        System.out.println("Refund executed");
    }

    public void audit() {
        System.out.println("[LOG] Before method");   // repeated
        System.out.println("Audit executed");
    }
}
```

Here logging is repeated three times.
That repeated logic is our cross-cutting concern (logging before each method).

```
@Aspect
```

```
public class LoggingAspect {

    @Before("execution(* PaymentService.*(..))")
    public void log() {
        System.out.println("[LOG] Before method");
    }
}
```

This says: Run log() before ALL methods in PaymentService.

Spring converts **@Aspect => Advice => Interceptor**, inserts this interceptor into the proxy, and the proxy calls the interceptor before invoking the real method.

Spring scans @Aspect => extracts advice

```
// Spring finds the method annotated with @Before
Method logMethod = LoggingAspect.class.getMethod("log");

// Create the aspect instance (normal Java object)
LoggingAspect aspectInstance = new LoggingAspect();

// Store metadata
AdviceMetadata metadata = new AdviceMetadata(
        aspectInstance,
        logMethod,
        "execution(* PaymentService.*(..))"
);
```

This produces an object representing our advice.
Spring wraps Advice => Interceptor

```
public class MethodBeforeAdviceInterceptor implements
MethodInterceptor {

    private final Object aspectInstance;
    private final Method logMethod;

    public MethodBeforeAdviceInterceptor(Object aspectInstance, Method
logMethod) {
        this.aspectInstance = aspectInstance;
        this.logMethod = logMethod;
    }

    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        // 1. run our aspect
        logMethod.invoke(aspectInstance);

        // 2. call the real method
        return invocation.proceed();
    }
}
```

Create the interceptor instance:

```
MethodBeforeAdviceInterceptor loggingInterceptor =
        new MethodBeforeAdviceInterceptor(aspectInstance, logMethod);
```

Spring generates a proxy subclass using CGLIB:

```
public class PaymentServiceProxy extends PaymentService {

    private MethodBeforeAdviceInterceptor interceptor;

    public PaymentServiceProxy(MethodBeforeAdviceInterceptor
interceptor) {
        this.interceptor = interceptor;
    }

    @Override
    public void pay() {
        try {
            interceptor.invoke(
                new MethodInvocation(
                    this,
                    PaymentService.class.getMethod("pay"),
                    new Object[] {}
                )
            );
        } catch (Throwable t) {
            throw new RuntimeException(t);
        }
    }

    @Override
    public void refund() {
        try {
            interceptor.invoke(
                new MethodInvocation(
                    this,
                    PaymentService.class.getMethod("refund"),
                    new Object[] {} // params
```

```
            )
        );
    } catch (Throwable t) {
        throw new RuntimeException(t);
    }
  }

  @Override
  public void audit() {
    try {
        interceptor.invoke(
            new MethodInvocation(
                this,
                PaymentService.class.getMethod("audit"),
                new Object[] {} // params
            )
        );
    } catch (Throwable t) {
        throw new RuntimeException(t);
    }
  }
}
```

What happens when the user calls the proxy?

```
PaymentService service =
        new PaymentServiceProxy(loggingInterceptor);

service.pay();
```

```
pay()
  |
proxy.pay()
  |
interceptor.invoke()
  |
logMethod.invoke(aspectInstance)  // calls our @Before method
  |
proceed() => super.pay()          // real method runs
```

Spring AOP supports five advice types:

1. @Before - Runs before the target method.
2. @AfterReturning - Runs after successful return.
3. @AfterThrowing - Runs only when the method throws an exception.
4. @After - Runs always (success or exception). Equivalent to finally
5. @Around - A method wrapper that controls before, after, and even skipping the method.

What is a pointcut?

"execution(* PaymentService.pay(..))" - This is a pointcut.

- Pointcut = method filter.
- A rule that selects which method(s) should be intercepted.
- The rule that decides where to apply the advice
- Execution Expressions:
    - syntax to define pointcuts
    - * => any return type
    - PaymentService => class
    - pay => method name
    - (..) => any arguments

What is a JoinPoint?

- A join point is the moment a method is about to run.'
- A Join Point is just a method call event.
- **paymentService.pay();** - This moment is a join point.

Why @Before and @After have NO CONTROL?

```
public Object invoke(MethodInvocation invocation) throws Throwable {

   logMethod.invoke(aspectInstance);  // BEFORE advice runs

   return invocation.proceed();       // Spring calls real method
}
```

The real method ALWAYS runs IMMEDIATELY after @Before.
We CANNOT stop it, delay it, skip it, replace it, or wrap it.

We don’t get the invocation object.
Spring controls the real method call.

That is why @Before has no control.

@Around vs @Before

```
public class PaymentService {

    public String pay(int amount) {
        System.out.println("Real pay: " + amount);
        return "SUCCESS";
    }
}
```

@Before advice:

```
@Aspect
public class BeforeAspect {

    @Before("execution(* PaymentService.pay(..))")
    public void beforeAdvice() {
        System.out.println("BeforeAdvice executed");
    }
}
```

BeforeInterceptor:

```
class BeforeInterceptor implements MethodInterceptor {

    private final LogBeforeAspect aspect;

    public BeforeInterceptor(LogBeforeAspect aspect) {
        this.aspect = aspect;
    }

    @Override
    public Object invoke(MethodInvocation inv) throws Throwable {
        aspect.beforeAdvice();       // BEFORE logic
        return inv.proceed();        // go to next (real method)
    }
}
```

Notice:

We do NOT control the real method

We do NOT see return value

We cannot wrap

We cannot skip

No ProceedingJoinPoint

Return type stays String because Spring calls the real method

@Around advice:

```
@Aspect
public class AroundAspect {
```

```
    @Around("execution(* PaymentService.pay(..))")
    public Object aroundAdvice(ProceedingJoinPoint pjp) throws
Throwable {

        System.out.println("Around BEFORE");

        // call the real method
        Object returnValue = pjp.proceed(new Object[]{200});  // overriding
args example

        System.out.println("Around AFTER");

        return returnValue; // must return Object!
    }
}
```

- WHY return type is Object?
    - Look at the real method: public String pay(int amount)
    - The return type is String.
    - But the interceptor signature must work for ANY return type:
        - String
        - int
        - boolean
        - void
        - List<Order>
        - PaymentResponse
    - So Spring uses the common parent: Object

```
class AroundInterceptor implements MethodInterceptor {

    private final LogAroundAspect aspect;

    public AroundInterceptor(LogAroundAspect aspect) {
        this.aspect = aspect;
    }

    @Override
    public Object invoke(MethodInvocation inv) throws Throwable {

        ProceedingJoinPoint pjp =
            new PJP(inv.target, inv.method, inv.args, inv.next);

        return aspect.aroundAdvice(pjp);
    }
}
```

What proxy looks like for @Around:

```
class PaymentServiceProxy extends PaymentService {

    private MethodInterceptor[] chain;

    public PaymentServiceProxy(MethodInterceptor[] chain) {
        this.chain = chain;
    }

    @Override
    public String pay(int amount) {
```

```
        MethodInvocation inv =
            new MethodInvocation(this, method_pay, new Object[]{amount},
chain, 0);

        return (String) inv.proceed();
    }
}
```

ProceedingJoinPoint:

- a small object Spring gives to @Around

```
@Around("execution(* PaymentService.pay(..))")
public Object log(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("Before");
    Object result = pjp.proceed();
    System.out.println("After");
    return result;
}
```

Spring creates the pjp object and passes it to our method.

What is inside ProceedingJoinPoint?

```
class PJP implements ProceedingJoinPoint {

    private final Object target;      // real PaymentService
    private final Method method;      // pay(int)
    private final Object[] args;      // { amount }
```

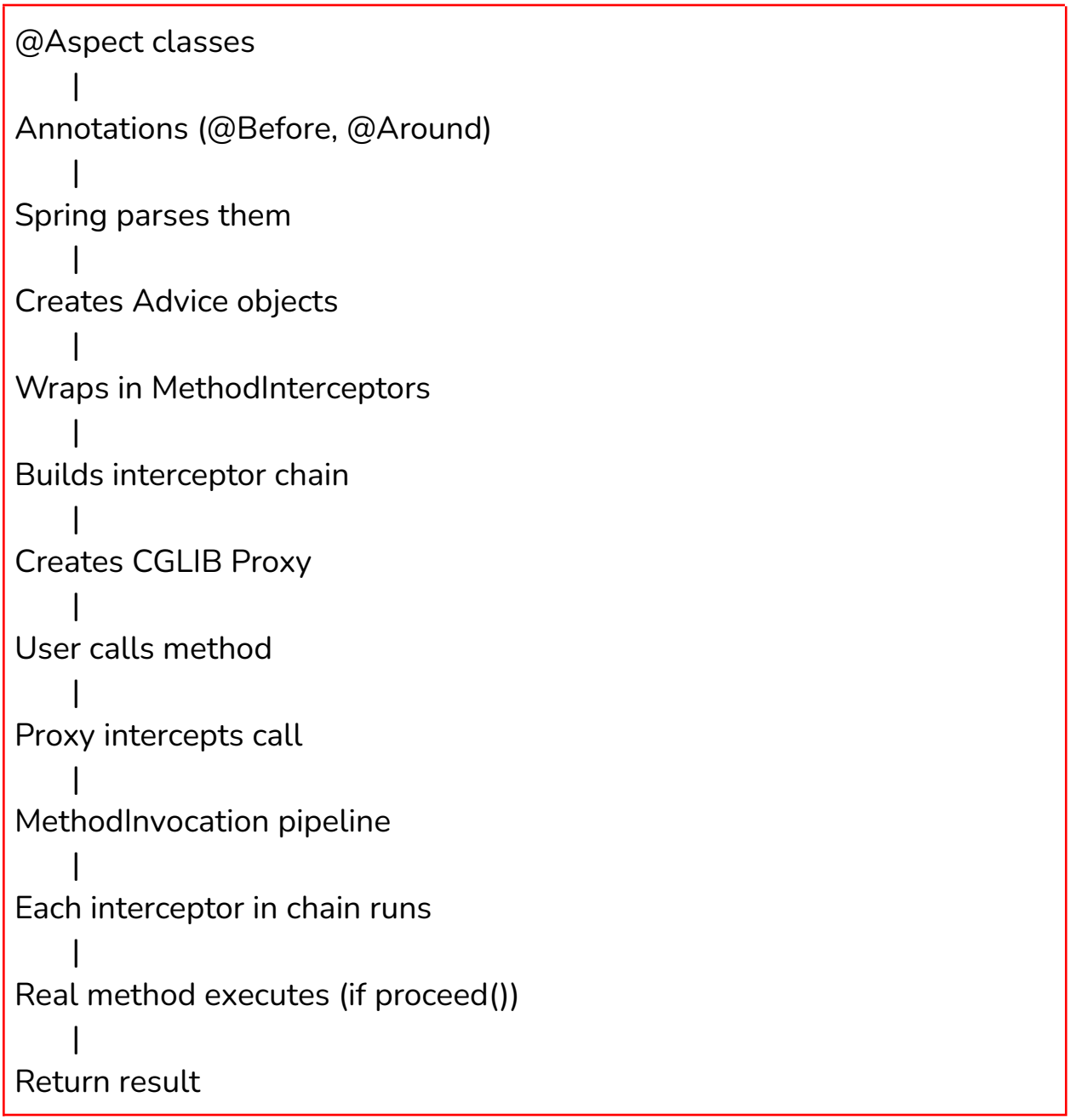
```
    public PJP(Object target, Method method, Object[] args) {
        this.target  = target;
        this.method  = method;
        this.args    = args;
    }

    @Override
    public Object proceed() throws Throwable {
        return method.invoke(target, args);
    }

    @Override
    public Object proceed(Object[] newArgs) throws Throwable {
        return method.invoke(target, newArgs);
    }
}
```

ProceedingJoinPoint: (target + method + args) + the ability to call the real method.

Flow:


@Aspect classes
|
Annotations (@Before, @Around)
|
Spring parses them
|
Creates Advice objects
|
Wraps in MethodInterceptors
|
Builds interceptor chain
|
Creates CGLIB Proxy
|
User calls method
|
Proxy intercepts call
|
MethodInvocation pipeline
|
Each interceptor in chain runs
|
Real method executes (if proceed())
|
Return result

14. Dependency Injection

What is a dependency?

- When class A needs class B to do its job, B is a dependency of A.

```
class PaymentService {
    private TaxService taxService;
}
```

PaymentService cannot work without TaxService.
PaymentService => depends on => TaxService

Traditional (BAD) approach – creating dependencies

```
class PaymentService {
    private TaxService taxService = new TaxService();
}
```

This causes problems:

- PaymentService is tightly coupled to TaxService
- Hard to test (We cannot easily replace TaxService with a fake/mock)
- Hard to change implementation (What if we want GSTService instead of TaxService?)
- We control object creation => no flexibility, no lifecycle management

This is why new is considered bad in large systems.

Dependency Injection

- Instead of a class creating its own dependencies, Spring creates them and gives them to the class.

```
class PaymentService {
    private final TaxService taxService;
    public PaymentService(TaxService taxService) {
        this.taxService = taxService;
    }
}
```

- PaymentService does not create TaxService
- It only receives whatever implementation Spring gives
- PaymentService is reusable, testable, clean, and loosely coupled

How Spring DI actually works internally:

Spring maintains a container (like a map):

```
beanMap = {
    TaxService.class => taxServiceInstance,
    PaymentService.class => paymentServiceInstance
}
```

When it sees: PaymentService(TaxService taxService)

Spring does:

- Check if TaxService is available
- Create TaxService if not already created

- Call: new PaymentService(taxServiceInstance)
  This is object graph building

Spring gives those dependencies can differ:

- Field injection
- Constructor injection
- Setter injection

Field Injection

```
@Service
class PaymentService {

   @Autowired
   private TaxService taxService;  // Inject into FIELD
}
```

What Spring actually does behind the scenes:

- Create object FIRST: PaymentService ps = new PaymentService();
- Then inject dependencies using reflection:
    - ps.taxService = taxServiceInstance;

Field injection creates the object first and injects dependencies later.

Constructor Injection

```
@Service
class PaymentService {

    private final TaxService taxService;

    @Autowired
    public PaymentService(TaxService taxService) {
        this.taxService = taxService;  // Inject via CONSTRUCTOR
    }
}
```

What Spring does behind the scenes:

- Create TaxService instance
    - TaxService ts = new TaxService();
    -
- THEN create PaymentService with the dependency
    - PaymentService ps = new PaymentService(ts);

constructor injection builds the object with its dependencies.
PaymentService cannot be created unless TaxService is provided.

Setter Injection

- Instead of injecting through
    - field (@Autowired private X x;)
    - or constructor (PaymentService(X x))

- Spring injects dependencies using a setter method.

```
@Service
class PaymentService {

   private TaxService taxService;

   @Autowired
   public void setTaxService(TaxService taxService) {
      this.taxService = taxService;
   }
}
```

When does setter injection run?

Spring steps:

```
1. new PaymentService()            <= object created (incomplete)
2. new TaxService()                <= dependency created
3. call setTaxService(taxService)    <= injection happens here
```

Why new is avoided in Spring

Because we used new, Spring cannot:

- inject dependencies into PaymentService
- apply AOP (no @Transactional, no @Around, no @Before)
- manage lifecycle (@PostConstruct, @PreDestroy)
- manage proxies
- detect circular dependencies
- apply caching

- apply security checks

Spring completely loses control.

So our object becomes outside the Spring container => a “dead” object.

Object Graph Wiring

Object graph:

- When our application starts, Spring creates many objects (beans)
- Each bean might depend on another bean.
- Spring connects (“wires”) these beans together into a graph
- This is called the object graph.

Object graph wiring is Spring’s process of creating all beans and connecting their dependencies together to form a working application.

15. IoC Container

- Object factory + Dependency manager + Lifecycle manager + Proxy factory
- The IoC Container is Spring's engine that creates objects, wires them together, manages their lifecycle, and applies AOP.

```
@Service
class PaymentService {
   private final TaxService taxService;
   public PaymentService(TaxService taxService) { ... }
}
```

IoC container timeline:

- Read as "bean definition"
- Creates TaxService bean
- Creates PaymentService bean with constructor injection
- Stores both in memory
- Returns PaymentService whenever we need it

When we start a Spring Boot app:

```
@SpringBootApplication
public class App {
   public static void main(String[] args) {
      ApplicationContext ctx = SpringApplication.run(App.class);
PaymentService ps = ctx.getBean(PaymentService.class);
   }
}
```

This method: SpringApplication.run(...) creates the IoC container internally.

The container is actually an ApplicationContext object created at runtime and stored in memory.

ApplicationContext:

- The IoC Container = ApplicationContext
- It is the object that holds the entire IoC container inside it.
- It contains:
    - a map of all beans
    - the BeanFactory engine
    - lifecycle managers
    - proxy creators
    - event broadcasters

BeanFactory:

- ApplicationContext delegates bean creation to BeanFactory
- ApplicationContext contains a BeanFactory
- BeanFactory creates beans AND handles their basic lifecycle
- BeanFactory responsibilities:
    - creating bean instances
    - injecting dependencies
    - managing scopes (singleton, prototype)
    - calling lifecycle methods (InitializingBean, DisposableBean)
    - managing singletons via a cache (a HashMap inside BeanFactory that stores bean instances.)

ApplicationContext = BeanFactory + all Spring features

16. Bean Definition

- BeanDefinition is a metadata object created by Spring during scanning, stored in the BeanFactory's map, and used as a blueprint (with factory info) to create real beans later.
- It is information ABOUT how to create the object

```
@Service
class PaymentService {
   public PaymentService(TaxService taxService) { }
}
```

A Bean Definition contains:

- beanName = "paymentService"
- beanClass = PaymentService
- scope = singleton
- dependencies = [TaxService]
- lazy = false
- proxy = false (unless AOP applies)
- factory = constructor(PaymentService)

When does Spring use the Bean Definition?

During object creation:
Step 1: Read Bean Definition
Step 2: Resolve dependencies
Step 3: Call constructor
Step 4: Inject fields/setters
Step 5: Wrap with proxy (if needed)

Step 6: Store in singleton cache

BeanDefinitions live inside the BeanFactory, which is part of the ApplicationContext.

A BeanDefinition is an object of **org.springframework.beans.factory.config.BeanDefinition** interface.

A BeanDefinition is a metadata object created by scanners and processors, stored inside DefaultListableBeanFactory, and used as a recipe so the BeanFactory can create real beans.

FULL FLOW:
STEP 1 - Spring Boot starts the ApplicationContext

When we run:

```
SpringApplication.run(App.class)
```

Spring creates:
ApplicationContext
|
DefaultListableBeanFactory (inside it)

BeanFactory contains an empty map:

```
beanDefinitionMap = {}
```

This is where all BeanDefinitions will be stored.

STEP 2 - Spring scans the classpath

THREE mechanisms create BeanDefinitions:

- Each source of a bean has different metadata and different creation rules

A. ClassPathBeanDefinitionScanner
- Scans classpath (@Component, @Service)
- Produces ScannedGenericBeanDefinition

B. AnnotatedBeanDefinitionReader
- Reads manually registered classes (@Configuration)
- Produces AnnotatedGenericBeanDefinition

C. ConfigurationClassPostProcessor
- Processes @Configuration (handles @Bean, @ComponentScan, @Import)
- Produces RootBeanDefinition for @Bean methods

If we provide the class => AnnotatedGenericBeanDefinition
If SPRING finds the class by scanning or import => ScannedGenericBeanDefinition
If class is from @Bean method => RootBeanDefinition

@Component => normal bean
@ComponentScan => where to look for beans
@Configuration => special bean containing @Bean methods
@Bean => method that creates another bean

@Import => include more configuration classes

STEP 3 - BeanDefinitions are stored

All BeanDefinition objects produced in step 2 are stored in:

DefaultListableBeanFactory.beanDefinitionMap

STEP 4 - Post-processors enhance BeanDefinitions
Before creating beans, Spring runs:

- ConfigurationClassPostProcessor
- AutowiredAnnotationBeanPostProcessor
- CommonAnnotationBeanPostProcessor
- BeanFactoryPostProcessors

These DO NOT create beans yet.
They only modify BeanDefinitions.

STEP 5 - BeanFactory creates real bean objects
Now the BeanFactory reads the stored BeanDefinitions one by one and creates real objects.

Creation steps:

- Look at BeanDefinition
- Determine how to create bean:
    - constructor?
    - @Bean factory method?
    - static factory method?

- Resolve dependencies
- Call the constructor / factory method
- Inject fields & setters
- Apply AOP proxies
- Store the final bean in singletonObjects map

17. Bean Creation strategy

1. Constructor Factory

```
@Service
class PaymentService {
    PaymentService(TaxService tax) { }
}
```

BeanDefinition stores:

factoryMethod = constructor(PaymentService)

factoryBeanName = null

factoryMethodName = null

- Create this bean by calling its constructor
- Spring does: new PaymentService(taxServiceInstance)

2. @Bean Factory Method

```
@Configuration
class AppConfig {
    @Bean
    PaymentService paymentService() {
        return new PaymentService(taxService());
    }
}
```

BeanDefinition stores:

factoryBeanName = "appConfig"

factoryMethodName = "paymentService"

- To create PaymentService, call appConfig.paymentService().
- Spring does:

```
AppConfig configObj = getBean("appConfig");
PaymentService ps = configObj.paymentService();
```

3. Static Factory Method

```
class PaymentFactory {
    public static PaymentService create() {
        return new PaymentService(new TaxService());
    }
}
```

BeanDefinition stores:

factoryBeanName = null

factoryMethodName = "create"

factoryClass = PaymentFactory

- Call the static method PaymentFactory.create() to build the bean.
- Spring does: PaymentService ps = PaymentFactory.create();

18. Bean Scopes

A scope decides how many times Spring creates the bean.

Singleton Scope (default):

Only one instance for the entire application.

When Spring creates a singleton bean:

- It creates one instance
- Stores it in the singleton cache
- Reuses the same instance everywhere

Prototype Scope

A new instance every time we request the bean.

Spring keeps scope info inside the BeanDefinition.
BeanFactory looks at this scope when creating the bean.

19. Bean Lifecycle

Instantiation:

- Spring calling our constructor (or factory method) to create an empty object.

```
@Component
class TaxService {
   public TaxService() {
      System.out.println("TaxService created");
   }
}
```

Spring reads BeanDefinition for TaxService
|
Spring picks constructor
|
Spring does new TaxService()
|
We see “TaxService created”

No injection yet.
No AOP yet.
No custom logic yet.

Code in constructors runs EARLY:

- before dependency injection
- before @PostConstruct
- before proxies

- before BeanPostProcessors

Dependency Injection

After instantiation, Spring takes our object and injects:

- constructor dependencies
- @Autowired fields
- @Autowired setters
- @Value fields

anything required to “complete” the object

Spring now resolves all needed beans and injects them.

Spring uses:

- populateBean()
- This method does:
    - Find dependencies from BeanDefinition
    - Resolve beans (create others if needed)
    - Inject into fields or setters

DI does NOT wrap our bean with AOP proxy

1. Instantiation:

PaymentService ps = new PaymentService();

2. Dependency Injection:

ps.tax = TaxService instance

ps.audit = AuditService instance

ps.repo = Repo bean instance

Now the object is “complete,” but not “initialized.”

Aware Interfaces

- Aware interfaces allow our bean to receive Spring’s internal objects (like BeanFactory or ApplicationContext) immediately after dependency injection and before initialization.

- Aware interfaces let our bean request Spring’s internal objects (like ApplicationContext). They are optional, rarely needed, and most applications use @Autowired instead.

Callback interface:

- An interface we implement
- Which Spring calls during lifecycle
- To give our bean some data

```
@Component
class PaymentService implements BeanNameAware {
```

```
    private String name;

    @Override
    public void setBeanName(String n) {
        this.name = n;
    }
}
```

1. Aware interfaces are callback interfaces.
2. Spring detects them using instanceof during bean creation.
3. If implemented, Spring calls the matching setter method.
4. The bean receives a Spring infrastructure object (ctx, factory, env...).
5. If the bean doesn't implement any Aware interface, nothing happens.
6. They are optional and rarely needed in normal app code.

If our bean implements an Aware interface,
Spring detects it using instanceof,
and calls the matching setter method to pass internal container objects.

BeanPostProcessor

A BeanPostProcessor (BPP) is an interface that Spring uses to

- Process a bean after DI, but before @PostConstruct, and
- Process a bean after initialization

It has two callback methods:
Object postProcessBeforeInitialization(Object bean, String name);
Object postProcessAfterInitialization(Object bean, String name);

Spring calls these for EVERY bean, including our services
This lets Spring:

- modify beans
- wrap beans
- validate beans
- replace beans
- create proxies

When does a BPP run in its lifecycle?

1. Instantiation (constructor)
2. Dependency injection (@Autowired)
3. Aware interfaces (setApplicationContext)
4. BeanPostProcessor BEFORE (postProcessBeforeInitialization)
5. @PostConstruct (or afterPropertiesSet)
6. BeanPostProcessor AFTER (postProcessAfterInitialization)
7. Bean ready

What does a BeanPostProcessor actually do?

- Inject @Autowired fields
- Handle @PostConstruct
- Create AOP proxies
- Implement @Transactional

Example:

```
@Component
class PaymentService {
```

```
    @Autowired
    TaxService taxService;

    @PostConstruct
    public void init() {
        System.out.println("Init method");
    }
}
```

```
@Component
class LoggingBPP implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String
name) {
        System.out.println("Before init: " + name);
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String name)
{
        System.out.println("After init: " + name);
        return bean;
    }
}
```

1. A BeanPostProcessor is a lifecycle hook.
2. Spring uses it to modify or replace beans automatically.

3. Runs after constructor + DI but before the bean is used.
4. BEFORE step: inject @Autowired, @Value.
5. AFTER step: create AOP proxies, apply @Transactional.

BeanPostProcessor = lifecycle hook that receives every bean after DI.

postProcessBeforeInitialization:
    Runs before @PostConstruct.
    Used for @Autowired, @Value, validation.

postProcessAfterInitialization:
    Runs after initialization.
    Used for AOP proxy creation.

If a BPP returns a proxy => the proxy becomes the actual bean used everywhere.

Without BPPs, Spring loses almost all advanced features.

Initialization

- Calling our initialization method. Nothing more
- After dependencies are injected, Spring runs our initialization logic.

Why do we need a separate initialization step?

Because the constructor runs too early.
At constructor time:

- @Autowired fields are not injected yet

- AOP proxies are not applied yet
- BeanFactory, Environment, Resources are not available yet

So constructor cannot safely do:

- cache loading
- DB connections
- Validation
- property reading
- logic that depends on injected beans

Spring needs a phase where:
Dependencies are injected => THEN our setup code runs.

That phase = Initialization.

Example:

```
@Component
class PaymentService {

    @Autowired
    TaxService tax;

    @PostConstruct
    public void init() {
        System.out.println("Test: " + tax);
    }
}
```

Lifecycle:

Object created
Dependencies injected
@PostConstruct called <= THIS is initialization
Bean ready

Spring Initialization Mechanisms:
1. @PostConstruct
   - Annotation on a method.
   - Runs first

```
@PostConstruct
public void init() { }
```

2. InitializingBean.afterPropertiesSet()

Implementing an interface.

```
class MyBean implements InitializingBean {
    @Override
    public void afterPropertiesSet() { }
}
```

Runs after @PostConstruct

3. Custom init method (XML or @Bean)

```
@Bean(initMethod = "startup")
public MyBean myBean() { return new MyBean(); }
```

Spring will call: myBean.startup();
Runs after afterPropertiesSet()

Order of execution:
1. @PostConstruct
2. InitializingBean.afterPropertiesSet()
3. Custom init-method

Destruction

- Destruction = run our cleanup code

When does bean destruction happen?
Only when:

- The Spring ApplicationContext is shutting down
- Or a scope ends (like request/session scope for web apps)

This does NOT happen on garbage collection.
This is Spring-managed destruction, not JVM-level destruction.

Destruction options:
1. @PreDestroy

- Annotation on a method.
- Runs first.

```
@PreDestroy
```

```
public void cleanup() { }
```

2. DisposableBean.destroy()

- Implementing the interface.
- Runs after @PreDestroy.

```
class MyBean implements DisposableBean {
    @Override
    public void destroy() { }
}
```

3. Custom destroy method

- Using @Bean.
- Runs last.

```
@Bean(destroyMethod = "shutdown")
public MyBean bean() { return new MyBean(); }
```

Spring will call - myBean.shutdown();

Order of destruction:

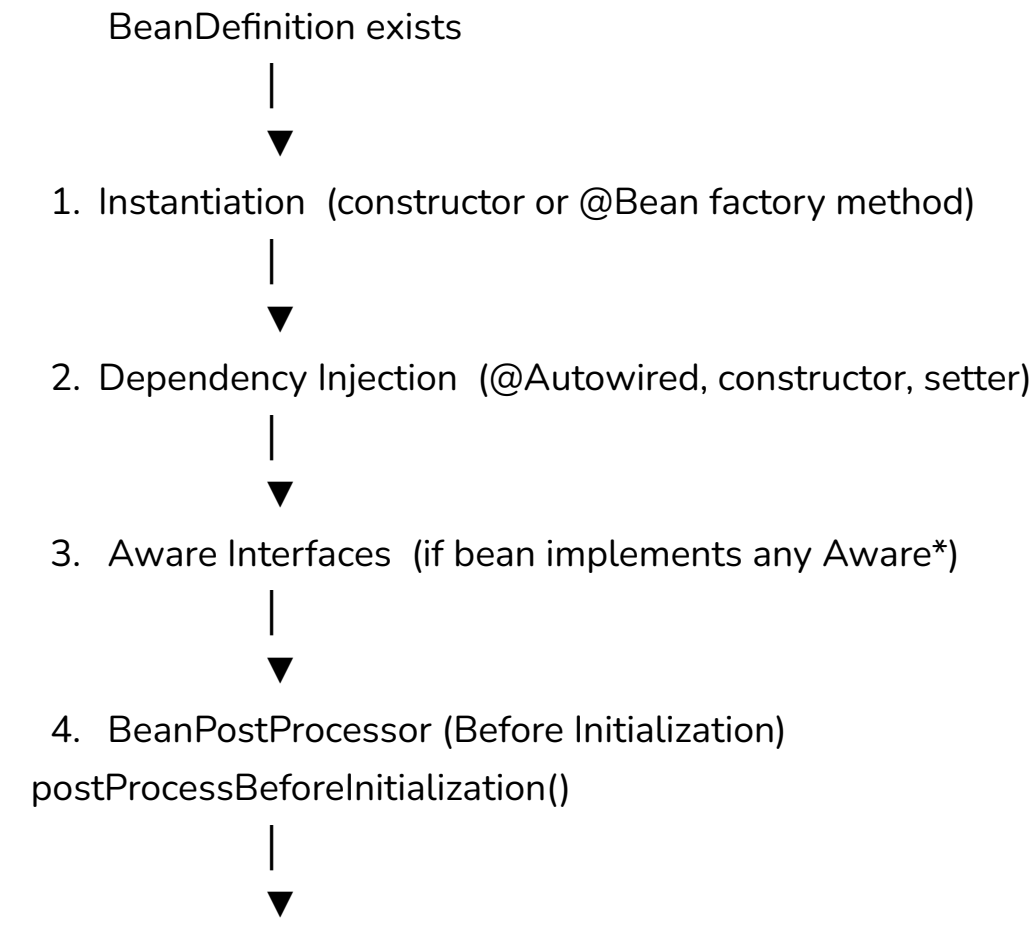
1. @PreDestroy
2. DisposableBean.destroy()
3. custom destroy-method

Flow:

1. Find all singleton beans

2. For each bean:
   - call @PreDestroy
   - call destroy() if DisposableBean
   - call custom destroy-method
3. Clear bean instances
4. JVM garbage collects later

Bean Lifecycle:



6. BeanPostProcessor (After Initialization)

postProcessAfterInitialization()

=> AOP proxy creation happens here

7. Bean is ready to use

8. Destruction (during context shutdown)

- @PreDestroy
- destroy()
- destroy-method

20. Autowiring Resolution Rules

By-Type Injection (Default)

```
@Autowired
PaymentService payment;
```

Spring performs the following algorithm:

- Determine required type: required type = PaymentService.class
- Search BeanFactory for beans of this type
    - If exactly ONE bean of that type exists: Inject
    - If NO bean is found: NoSuchBeanDefinitionException
    - If MORE THAN ONE bean exists: NoUniqueBeanDefinitionException

@Qualifier for exact selection

We can name the bean here

```
@Autowired
@Qualifier("taxServiceB")
TaxService tax;
```

tax = bean named "taxServiceB"

@Qualifier overrides by-type selection and forces by-name selection.

Rule 1: Spring injects by TYPE.

Rule 2: If exactly one bean matches => OK.
Rule 3: If zero beans match => NoSuchBeanDefinitionException.
Rule 4: If more than one bean matches => NoUniqueBeanDefinitionException.
Rule 5: @Primary resolves conflicts by marking one bean as default.
Rule 6: @Qualifier explicitly chooses the bean by NAME.

@Primary

When multiple beans match the required type, Spring needs to choose one

@Primary marks one bean as the default bean for that type.

```
@Component
@Primary
class TaxServiceA implements TaxService {}

@Component
class TaxServiceB implements TaxService {}
```

```
@Autowired
TaxService tax;
```

Algorithm:

Type needed => TaxService
Beans found => TaxServiceA, TaxServiceB
Which bean has @Primary? => TaxServiceA
Inject TaxServiceA

@Primary works ONLY for by-type injection

@Qualifier

When multiple beans match a type, @Qualifier lets us pick the bean by its name.

```
@Component("upi")
class UpiPayment implements Payment {}

@Component("card")
class CardPayment implements Payment {}
```

```
@Autowired
@Qualifier("card")
Payment payment;
```

By-Type vs By-Name Autowiring

```
@Autowired
Payment payment;
```

```
@Component
class UpiPayment implements Payment {}

@Component
class CardPayment implements Payment {}
```

2 beans matching Payment

Now Spring checks the field name:

- field is named: payment
- bean candidates named: upiPayment, cardPayment
- No match => still conflict.

If field was named upiPayment:

```
@Autowired
Payment upiPayment;
```

Now Spring can resolve uniquely

fieldName = upiPayment

beanName = upiPayment

1. Spring always matches by TYPE first.
2. If exactly one bean matches => inject.
3. If multiple beans match => try by field NAME.
4. If still ambiguous => use @Primary if available.
5. If @Qualifier is present => it overrides everything else.

21. ApplicationEvent

ApplicationEvent is a Spring framework event object that represents something that happened inside the application and can be published to notify other beans without direct coupling

```
public class PaymentCompletedEvent{

    private final String orderId;

    public PaymentCompletedEvent(String orderId) {
        this.orderId = orderId;
    }

    public String getOrderId() {
        return orderId;
    }
}
```

Spring allows any bean to publish events using ApplicationEventPublisher.

```
@Component
class PaymentService {

    @Autowired
    private ApplicationEventPublisher publisher;

    public void pay() {
```

```
        // business logic...

        publisher.publishEvent(new
PaymentCompletedEvent("ORDER123"));
    }
}
```

Spring delivers this event to all listeners.

```
@Component
class AuditListener {

    @EventListener
    public void handle(PaymentCompletedEvent event) {
        System.out.println("Audit: payment completed for " +
event.getOrderId());
    }
}
```

- publishEvent(event) is called
- Spring collects all beans that listen for that event type
- Spring calls each listener in order
- Listener methods execute
- **Control returns to publisher**

By default, the publisher waits until ALL event listeners finish execution. Spring uses synchronous method calls unless we explicitly enable async handling.

Conditional listeners:
We can filter events:

```
@EventListener(condition = "#event.orderId.startsWith('VIP')")
public void vipOrders(PaymentCompletedEvent event) { }
```

Only events matching the condition run this listener.

Context Lifecycle Events

- These are events automatically published by Spring when the ApplicationContext changes state

Lifecycle events (in order):
1. ApplicationStartingEvent:
When:
- JVM just started
- SpringApplication is created
- Context is NOT created yet

Use:
- Very early logging
- Environment inspection

```
@EventListener
public void onStart(ApplicationStartingEvent e) {
    System.out.println("Application is starting");
}
```

Beans are NOT available here

2. ApplicationEnvironmentPreparedEvent:

When:

- Environment is ready
- Properties loaded
- Context not created yet

Use:

- Read config properties
- Modify environment

3. ApplicationContextInitializedEvent:

When:

- ApplicationContext object created
- Bean definitions NOT loaded

4. ApplicationPreparedEvent:

When:

- Bean definitions loaded
- Beans NOT created yet

5. ContextRefreshedEvent:

When:

- All beans created
- Dependency injection done
- @PostConstruct executed

6. ApplicationReadyEvent:

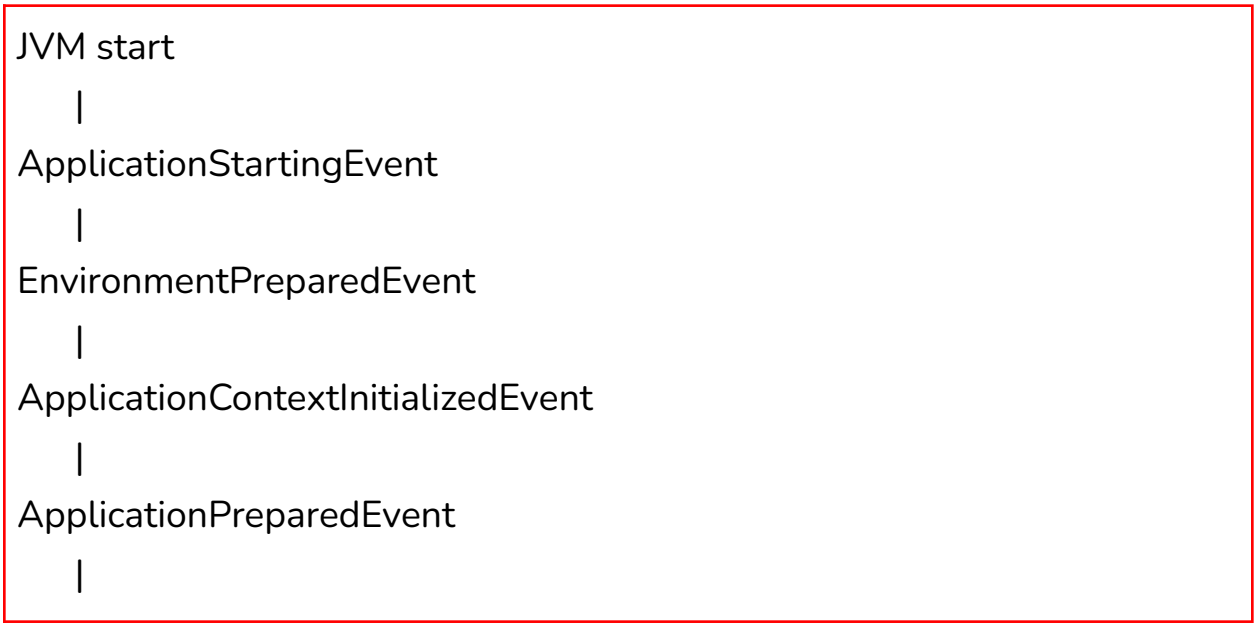
When:

- Application fully started
- Web server started
- App is ready to receive requests

7. ContextClosedEvent:

When:

- Application shutting down
- Before @PreDestroy



```
ContextRefreshedEvent
    |
ApplicationReadyEvent
    |
[Application running]
    |
ContextClosedEvent
    |
JVM exit
```

Async event handling

- Event listeners run in a different thread
- so the publisher does NOT wait for them to finish.

```
publishEvent()
 => submit listener1 to executor
 => submit listener2 to executor
 <= return to publisher immediately
```

Mark the listener as @Async

```
@Component
class AuditListener {

   @Async
   @EventListener
   public void handle(PaymentCompletedEvent event) {
      System.out.println("Async listener thread: " +
```

```
Thread.currentThread().getName());
    }
}
```

Internally:
Spring does NOT call the method directly.
Instead:

- Event is published
- Spring detects listener is async
- Spring wraps listener call in a Runnable
- Runnable is submitted to a TaskExecutor
- Publisher returns immediately
- Listener executes later in another thread

Where does the thread come from?

Case A - We did NOT define any executor
Spring uses a default executor:

SimpleAsyncTaskExecutor

Characteristics:

- Creates new threads
- No pooling

Case B - Custom executor

```
@Configuration
@EnableAsync
```

```
class AsyncConfig {

    @Bean
    Executor taskExecutor() {
        ThreadPoolTaskExecutor exec = new ThreadPoolTaskExecutor();
        exec.setCorePoolSize(5);
        exec.setMaxPoolSize(10);
        exec.setQueueCapacity(100);
        exec.initialize();
        return exec;
    }
}
```

Spring now uses this executor for all @Async methods.

What does @EnableAsync actually do?

@EnableAsync does NOT:

- create threads
- create executors
- run anything by itself

What it actually does:

- @EnableAsync registers an internal BeanPostProcessor.
- Specifically: AsyncAnnotationBeanPostProcessor
- This BPP does one job:
    - If a bean method is annotated with @Async, wrap that bean with a proxy. So method calls are executed via Executor.

-1. End-to-End Runtime Flow

```
@RestController
class OrderController {
    @Autowired
    OrderService orderService;

    @GetMapping("/order")
    public String place() {
        return orderService.place();
    }
}

@Service
class OrderService {
    @Transactional
    public String place() {
        return "OK";
    }
}
```

When the application starts,

- The JVM first loads these classes using the **Application ClassLoader**, storing their bytecode, method signatures, and annotation metadata in the **Method Area**. No objects exist yet.

- Spring then scans the loaded class metadata using **reflection**, detects @RestController and @Service, and creates bean definitions for OrderController and OrderService.

- While creating the OrderService bean, Spring notices @Transactional on place() and decides interception is required, so it creates a **proxy** (JDK proxy or CGLIB) and registers the proxy, not the real object, in the **IoC container**.The real OrderService instance lives inside the proxy on the heap.

- When the HTTP request /order arrives, Spring routes it to the controller proxy, a **stack frame** is created for place(), the proxy intercepts the call, starts a transaction, invokes the real method, commits the transaction, and returns the result.

- After the method returns, the stack frame is destroyed, temporary objects become eligible for **garbage collection**, and the application waits for the next request!

==========