Annotations in Java explanation

give me the deep and detailed explaination of Annotations in java so that i can able to understand it well and do not have any type of doubt in this.

so give me the deep and detailed explaination

Think of *annotations* in Java as **special labels / tags** that you attach to your code so that:

- The **compiler** can check or change behavior
- **Frameworks / libraries** (like Spring, Hibernate) can understand what to do
- **Tools** (like Lombok, validation frameworks, JSON mappers, etc.) can generate or handle extra code

They **do not execute by themselves** like methods. They're **metadata** (information *about* your code).

---

1. What is an Annotation?

Formal definition:
An annotation is a **special kind of interface** that provides metadata about your program. You attach it to classes, methods, fields, parameters, etc.

Basic syntax:

```java
@Override
public String toString() {
    return "Hello";
}
```

Here, `@Override` is an annotation.

You can think of it like a **sticker** on your code:
"Dear compiler/framework, here is extra information about this element."

Annotations:

- Don't change the *Java syntax* of code
- Are processed by compiler, tools, or runtime frameworks
- Can be checked and used via **reflection** (if kept at runtime)

---

2. Where Can We Use Annotations?

You can put annotations on:

- Class / interface / enum / record:

```java
@Deprecated
public class OldClass { }
```

- Methods:

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

- Fields:

```java
@Deprecated
private int oldField;
```

- Parameters:

```java
public void setName(@NotNull String name) { }
```

- Constructors:

```java
@Inject
public Service(Dependency d) { }
```

- Local variables:

```java
public void m() {
    @NonNull String s = "hello";
}
```

- Type use (Java 8+):

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

The possible places are controlled by another concept: **@Target** (we'll come to it).

---

3. Types of Annotations (Conceptually)

Annotations can be of these conceptual types:

1. **Marker annotations** – no values, just presence:

   ```java
   @Override
   void m() { }
   ```

   Only the presence of `@Override` matters.

2. **Single-value annotations** – one main value:

   ```java
   @SuppressWarnings("unchecked")
   void m() { }
   ```

3. **Full/normal annotations** – multiple key–value pairs:

   ```java
   @MyAnnotation(name = "Piyush", age = 21)
   class Student { }
   ```

We'll see how to define these later.

---

4. Important Built-in Annotations

4.1 `@Override`

Used on methods to tell compiler:
"This method is **overriding** a method from a superclass or interface."

```java
class Parent {
    void show() { }
}

class Child extends Parent {
    @Override
    void show() {   // OK, correctly overriding
        System.out.println("Child");
    }
}
```

If you write:

```java
class Child extends Parent {
    @Override
    void shw() { }  // compile-time error: no method to override
}
```

So `@Override` helps:

- Catch **typos**
- Make your intention clear

4.2 `@Deprecated`

Marks something as **old / not recommended**.

```java
@Deprecated
class OldService {
    @Deprecated
    void oldMethod() { }
}
```

When you use it, IDE/Compiler shows a **warning**:

```java
OldService s = new OldService(); // warning: OldService is deprecated
s.oldMethod();                   // warning: oldMethod is deprecated
```

New code should avoid using deprecated APIs.

4.3 `@SuppressWarnings`

Tells the compiler:
"I know this code may produce a specific warning; please ignore it."

```java
@SuppressWarnings("unchecked")
void m() {
    List list = new ArrayList(); // unchecked warning normally
    list.add("hello");
}
```

Common values:

- **`"unchecked"`**
- **`"deprecation"`**
- **`"rawtypes"`**
- or `"all"` to suppress everything (not recommended).

4.4 `@FunctionalInterface`

Used on interfaces to say:
"This interface should have **exactly one abstract method**" (to be used as lambda).

```java
@FunctionalInterface
interface MyFunction {
    int apply(int x);
    // If you add another abstract method -> compile-time error
}
```

This is helpful for lambda expressions.

4.5 Other Common JDK Annotations

- `@SafeVarargs` – for methods using varargs with generics, to suppress unchecked warnings.
- `@SuppressWarnings` – we saw.
- `@Retention`, `@Target`, `@Documented`, `@Inherited`, `@Repeatable` – these are **meta-annotations** (used on annotations themselves).

---

5. Meta-Annotations (Annotations on Annotations)

Meta-annotations tell Java **how the annotation itself behaves**.

Important meta-annotations:

1. `@Retention`
2. `@Target`
3. `@Documented`
4. `@Inherited`
5. `@Repeatable`

Let's go one by one.

5.1 `@Retention`

Defines **how long** the annotation is kept:

```java
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation { }
```

`RetentionPolicy` has 3 options:

1. `SOURCE`
    - Annotation exists only in **source code**
    - Compiler **discards** it; not in `.class` file
    - Cannot be read by reflection at runtime

- Example: `@Override`

2. `CLASS` (default)
   - Annotation stored in `.class` file
   - Not visible at runtime (not loaded into JVM for reflection)
   - Usually used for tools working on bytecode, not runtime.
3. `RUNTIME`
   - Stored in `.class` file
   - Available at **runtime via reflection**
   - Used by frameworks like Spring, Hibernate, validation, etc.

```java
@Retention(RetentionPolicy.RUNTIME)
@interface MyRuntimeAnnotation { }
```

If you want to **read annotation at runtime**, always use `RUNTIME`.

5.2 `@Target`

Specifies **where** this annotation can be applied.

```java
@Target(ElementType.METHOD)
public @interface MyMethodAnnotation { }
```

`ElementType` values (important ones):

- `TYPE` → class, interface, enum, record
- `FIELD` → fields
- `METHOD` → methods
- `PARAMETER` → method parameters
- `CONSTRUCTOR` → constructors
- `LOCAL_VARIABLE` → local variables
- `ANNOTATION_TYPE` → another annotation
- `TYPE_PARAMETER` → generic type parameters `<T>`
- `TYPE_USE` → any use of a type

Example – annotation usable on class **and** method:

```java
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface MyAnnotation { }
```

If you try to put it somewhere else (like on a field), compiler will show an error.

5.3 `@Documented`

Indicates that this annotation should appear in the **generated Javadoc**.

```java
@Documented
public @interface MyAnnotation { }
```

Good for library authors so users see annotations in documentation.

5.4 `@Inherited`

Controls **inheritance of annotations** on classes.

```java
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface MyInheritedAnnotation { }

@MyInheritedAnnotation
class Parent { }

class Child extends Parent { }
```

Now:

```java
Child.class.isAnnotationPresent(MyInheritedAnnotation.class); // true
```

So the annotation on `Parent` is also considered present on `Child` (for `TYPE` level only).

5.5 `@Repeatable`

Allows you to use **same annotation multiple times** on one element.

Old style:

```java
@interface Role {
    String name();
}

@interface Roles {
    Role[] value();
}

@Roles({
    @Role(name="ADMIN"),
    @Role(name="USER")
})
class User { }
```

With **`@Repeatable`** (Java 8+):

```java
@Repeatable(Roles.class)
@interface Role {
    String name();
}

@interface Roles {
    Role[] value();
}

@Role(name = "ADMIN")
@Role(name = "USER")
class User { }
```

Simpler to read & write.

---

6. Defining Your Own Annotation

You define an annotation using **`@interface`** (instead of **`interface`**).

6.1 Basic Custom Annotation

```java
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Info {
    String author();
    String date();
    int version() default 1;   // default value
}
```

Points to notice:

- Use **`public @interface Info { ... }`**
- Inside, you define **methods without body** → these are called **elements** or **members** of the annotation.
- **`version()`** has a **default value**.

Using it:

```java
@Info(author = "Piyush", date = "2025-12-10", version = 2)
public class MyService {
}
```

If default is present, you can omit:

```java
@Info(author = "Piyush", date = "2025-12-10")
public class MyService {
}
```

6.2 Types Allowed Inside Annotations

Annotation elements can have only specific types:

- Primitive types: **`int`**, **`long`**, **`boolean`**, etc.
- **`String`**
- **`Class<?>`**
- Enums
- Other annotations
- Arrays of above types

Examples:

```java
public @interface Config {
    String name();
    int timeout() default 1000;
    String[] tags() default {};
    Class<?> type() default Object.class;
}
```

Illegal types: no **`List`**, no custom objects like **`User`**.

6.3 Marker Annotations

No elements at all:

```java
public @interface Marker { }
```

Use:

```java
@Marker
class Test { }
```

6.4 Single-Value Annotation Special Case

If an annotation has only **one element** named `value`, you can use shorter syntax.

```java
public @interface Author {
    String value();
}

// Using it:
@Author("Piyush")
class MyClass { }
```

Instead of:

```java
@Author(value = "Piyush")
```

---

7. Accessing Annotations at Runtime (Reflection)

If an annotation has `@Retention(RetentionPolicy.RUNTIME)`, you can read it via reflection.

Example:

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

// 1. Define annotation
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface Info {
    String author();
    int version() default 1;
}

// 2. Use annotation
@Info(author = "Piyush", version = 2)
class MyService {
}

// 3. Read annotation
public class Main {
    public static void main(String[] args) {
        Class<MyService> clazz = MyService.class;

        if (clazz.isAnnotationPresent(Info.class)) {
            Info info = clazz.getAnnotation(Info.class);
            System.out.println("Author: " + info.author());
            System.out.println("Version: " + info.version());
        } else {
            System.out.println("No Info annotation present");
        }
    }
}
```

Output:

```text
Author: Piyush
Version: 2
```

This is exactly what frameworks like Spring do internally:

- Scan classes
- Check which annotations are present
- Decide what to do (create beans, map URLs, validate fields, etc.)

---

8. Annotations in Frameworks (Spring Example)

You already use Spring Boot, so these will be familiar:

```java
@RestController
@RequestMapping("/users")
public class UserController {

    @GetMapping("/{id}")
    public User getUser(@PathVariable Long id) { ... }

    @PostMapping
    public User createUser(@RequestBody User user) { ... }
}
```

What's happening:

- **`@RestController`**
  Tells Spring: "This class handles REST requests and returns JSON/text responses."
- **`@RequestMapping("/users")`**
  Base URL for all endpoints in this controller.
- **`@GetMapping`**, **`@PostMapping`**
  Map specific HTTP methods + paths.
- **`@PathVariable`**, **`@RequestBody`**
  Tell Spring how to bind URL and body to method parameters.

Spring uses **reflection + runtime annotations** to:

- Detect all controllers
- Build a mapping: **`(HTTP_METHOD, PATH) → method`**
- Resolve parameters, validations, etc.

So annotations are the *language* through which you **communicate with the framework**.

---

9. Compile-Time Annotation Processing (Advanced but Important)

Some tools use **annotation processors**: classes that run **at compile time** and generate code or perform checks.

Example: Lombok

```java
@Data
public class User {
    private String name;
    private int age;
}
```

`@Data` is processed at compile time → generates getters, setters, `toString()`, etc., in the compiled `.class` file.

This is done with `javax.annotation.processing` package (or `jakarta.*` in newer Java EE):

- You create a class extending `AbstractProcessor`
- The compiler calls it and passes elements with certain annotations
- You can write new `.java` or `.class` files

You don't need to implement this now, but understanding:
Annotations can affect both **compile-time** and **runtime** behavior.

---

10. Type Annotations (Java 8+)

Earlier you could only annotate declarations (class, method, field).
Now you can annotate **any use of a type**.

```java
List<@NonNull String> names;
@NonNull String s = methodReturningString();
```

Used in:

- Nullness checking tools
- Static analysis tools
- More precise code checks

To enable this, use `@Target(ElementType.TYPE_USE)`.

---

11. Annotations vs Comments vs Interfaces

11.1 Annotations vs Comments

- **Comments**: ignored by compiler, no structure, no meaning for tools (unless custom parser reads them).
- **Annotations**: structured, type-checked, available in bytecode and at runtime (if RUNTIME).

So annotations are like "machine-readable comments".

11.2 Annotations vs Interfaces / Code

- Interfaces / classes contain **behavior** (methods, logic).
- Annotations contain **metadata** (values, tags, configuration).

Annotations **describe** what should happen;
Actual behavior is implemented by compiler/framework.

---

12. Summary in Simple Words

- Annotation = **tag/label/metadata** on code.
- They are written with `@Name`.
- They don't do anything alone; **compiler/framework** reads them.
- You can:
  - Use built-in ones (`@Override`, `@Deprecated`, `@FunctionalInterface`, etc.).
  - Create your own using `@interface`.
- Meta-annotations (`@Retention`, `@Target`, `@Inherited`, `@Documented`, `@Repeatable`) describe:
  - How long annotation is kept
  - Where it can be used
  - Whether it is inherited
  - Whether it is repeatable
- `RUNTIME` retention + reflection = frameworks like Spring working.
- Annotations are everywhere in modern Java development (Spring, Hibernate, JPA, validation).

If you want, next I can:

- Give you a **small project-style example** using custom annotations (like a mini `@Test` framework), and show step-by-step how to scan annotated methods and call them using reflection.