You're asking an excellent and deeper question about **how annotations technically work under the hood** in Java and how they inform the compiler and runtime environment. Let's break it down step by step:

**1. Annotations and the Compiler**

Annotations interact with the compiler through **metadata** and optional **annotation processors**. Here's what happens:

1. **Metadata Recording**
   * When you use annotations in your code, they are stored as metadata in the compiled .class files.
   * Depending on the annotation's **retention policy** (SOURCE, CLASS, or RUNTIME):
     + SOURCE: The annotation is discarded by the compiler after processing.
     + CLASS: The annotation is included in the .class file but not available at runtime.
     + RUNTIME: The annotation metadata is stored in the .class file and is available during runtime through reflection.

Example of a Retention Policy:

@Retention(RetentionPolicy.RUNTIME)

public @interface MyAnnotation { }

1. **Annotation Processors**
   * During compilation, the Java Compiler (javac) invokes **annotation processors** (tools based on the javax.annotation.processing package) to process annotations and generate additional code or perform checks.
   * Annotation processors implement the Processor interface and operate on annotations with the @SupportedAnnotationTypes and @SupportedSourceVersion directives.

Example of an Annotation Processor:

@SupportedAnnotationTypes("MyAnnotation")

public class MyProcessor extends AbstractProcessor {

@Override

public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {

for (Element element : roundEnv.getElementsAnnotatedWith(MyAnnotation.class)) {

System.out.println("Processing: " + element);

}

return true;

}

}

1. **Static Analysis**
   * The compiler performs checks or transformations based on annotations like @Override or @FunctionalInterface. For example:
     + @Override: Ensures the annotated method matches a superclass method.
     + @FunctionalInterface: Ensures the interface contains exactly one abstract method.

**2. Annotations at Runtime**

Annotations inform the runtime environment via the **Reflection API**, which allows inspection and manipulation of classes, methods, fields, and their annotations.

1. **Annotation Metadata in Bytecode**
   * The compiler writes annotation metadata into the .class file.
   * At runtime, the JVM loads the .class file, including the metadata, into memory.
2. **Accessing Annotations with Reflection**
   * Using java.lang.reflect classes like Class, Method, or Field, you can query annotations at runtime.

Example:

@Retention(RetentionPolicy.RUNTIME)

public @interface MyAnnotation {

String value();

}

@MyAnnotation("Hello, World!")

public class MyClass { }

public class Main {

public static void main(String[] args) {

Class<MyClass> clazz = MyClass.class;

MyAnnotation annotation = clazz.getAnnotation(MyAnnotation.class);

System.out.println("Annotation value: " + annotation.value());

}

}

Output:

Annotation value: Hello, World!

1. **Framework Behavior**
   * Frameworks like Spring or Hibernate dynamically process annotations using reflection and bytecode manipulation libraries like ASM or Javassist.
   * Example:
     + Spring scans for @Component annotations during the application startup and registers those classes as beans in the application context.

**3. Behind-the-Scenes in Spring**

Spring Boot, for instance, uses annotations heavily to configure and manage application behavior. Here's how it works internally:

1. **Classpath Scanning**:
   * Spring scans the classpath for specific annotations (e.g., @RestController, @Service) during application startup using a mechanism like ClassPathScanningCandidateComponentProvider.
2. **Reflection & Proxies**:
   * When Spring detects @Transactional, it creates a **proxy object** around the annotated method. This proxy intercepts calls to the method and adds transactional logic before and after the actual method execution.
3. **Custom Annotations**:
   * Spring processes custom annotations using BeanPostProcessor or HandlerMethodArgumentResolver for finer-grained control.

**4. How Annotations Work in the Compiler**

**Bytecode Representation:**

Annotations are represented in bytecode with **attributes**. For example:

@Retention(RetentionPolicy.RUNTIME)

public @interface MyAnnotation {

String value();

}

When applied:

@MyAnnotation("example")

public class MyClass { }

The bytecode (javap -c -v MyClass) will show something like this:

RuntimeVisibleAnnotations:

0: #11(#12=s#13)

#11: MyAnnotation

#12: value

#13: example

**Summary**

1. **Compilation Phase:**
   * Metadata is stored in .class files.
   * Annotation processors perform compile-time tasks (e.g., code generation, validations).
2. **Runtime Phase:**
   * JVM uses reflection to read annotations.
   * Frameworks (e.g., Spring) process annotations dynamically to configure behaviors like dependency injection, AOP, etc.

Annotations are more than just tags—they act as the bridge between code and meta-logic in Java, enabling frameworks and tools to execute custom behavior seamlessly.