

## 1. What Is the Interpreter Design Pattern?

The **Interpreter Pattern** defines a way to evaluate language grammar or expressions by representing rules as a class hierarchy.

In simple terms:

It allows you to **interpret and evaluate sentences in a custom language or rule syntax**, by modeling grammar rules as objects.

---

### Example Analogy

Think of how a calculator interprets  $5 + 3 * 2$ .

Each number and operator (+, \*) is part of a *language grammar*.

An **interpreter** pattern allows you to:

- Represent each part (Expression) as an object.
  - Build a tree (Expression Tree).
  - Evaluate it by calling `.interpret()`.
- 

## 2. Core Participants

Component	Role
<b>AbstractExpression</b>	Declares the <code>interpret()</code> interface
<b>TerminalExpression</b>	Represents constants/variables
<b>NonTerminalExpression</b>	Represents operators or rules combining expressions
<b>Context</b>	Contains global information needed during interpretation

---

### Example (Simplified Expression Evaluator)

```
interface Expression {
    int interpret();
}

// Terminal Expression
class NumberExpression implements Expression {
    private int number;
    public NumberExpression(int number) { this.number = number; }
    public int interpret() { return number; }
}

// Non-Terminal Expression (Addition)
class AddExpression implements Expression {
    private Expression left, right;
    public AddExpression(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }
    public int interpret() {
        return left.interpret() + right.interpret();
    }
}
```

```

}

// Client
public class InterpreterDemo {
    public static void main(String[] args) {
        // Represents (5 + 3) + (2 + 1)
        Expression result = new AddExpression(
            new AddExpression(new NumberExpression(5), new
NumberExpression(3)),
            new AddExpression(new NumberExpression(2), new
NumberExpression(1))
        );

        System.out.println("Result: " + result.interpret());
    }
}

```

---

### 3. Why It's a Behavioral Pattern

Because it focuses on **how objects collaborate to interpret and execute meaning (behavior)** of sentences in a language.

It's not about structure or creation, but about **interpreting behavior** defined by rules — hence **behavioral**.

---

### 4. Real Power — Rule-Based Frameworks

Now let's connect it to **Spring Boot** and real-world use.

A **rule engine** evaluates conditions dynamically, for example:

```

IF (customer.age > 60 AND customer.accountBalance > 50000)
THEN discount = 10%

```

You could:

- Write Java `if-else` blocks everywhere 😬
- OR build a **rule interpreter** that can evaluate dynamic rule expressions at runtime ✅

That's the **Interpreter Pattern** in action.

---

### 5. Interpreter Pattern in a Rule Engine (Spring Boot Example)

We'll build a simple **rule interpreter** that evaluates logical expressions defined as text.

---

#### Step 1: Define the Expression Interface

```

public interface Expression {
    boolean interpret(Context context);
}

```

---

## Step 2: Define Context (stores runtime data)

```
public class Context {
    private Map<String, Object> data = new HashMap<>();

    public void put(String key, Object value) {
        data.put(key, value);
    }

    public Object get(String key) {
        return data.get(key);
    }
}
```

---

## Step 3: Implement Terminal and Non-Terminal Expressions

```
// Terminal Expression
public class VariableExpression implements Expression {
    private String key;
    private String operator;
    private Object value;

    public VariableExpression(String key, String operator, Object value) {
        this.key = key;
        this.operator = operator;
        this.value = value;
    }

    @Override
    public boolean interpret(Context context) {
        Object obj = context.get(key);
        if (obj instanceof Integer) {
            int var = (Integer) obj;
            int val = Integer.parseInt(value.toString());
            return switch (operator) {
                case ">" -> var > val;
                case "<" -> var < val;
                case "==" -> var == val;
                default -> false;
            };
        }
        return false;
    }
}

// Non-terminal Expression for logical AND
public class AndExpression implements Expression {
    private Expression left, right;
    public AndExpression(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }
    @Override
    public boolean interpret(Context context) {
        return left.interpret(context) && right.interpret(context);
    }
}

// Non-terminal Expression for logical OR
public class OrExpression implements Expression {
```

```

private Expression left, right;
public OrExpression(Expression left, Expression right) {
    this.left = left;
    this.right = right;
}
@Override
public boolean interpret(Context context) {
    return left.interpret(context) || right.interpret(context);
}
}

```

---

## Step 4: Test in a Spring Boot Runner

```

@SpringBootApplication
public class RuleInterpreterApp implements CommandLineRunner {

    public static void main(String[] args) {
        SpringApplication.run(RuleInterpreterApp.class, args);
    }

    @Override
    public void run(String... args) {
        Context ctx = new Context();
        ctx.put("age", 65);
        ctx.put("balance", 70000);


        Expression ageRule = new VariableExpression("age", ">", 60);
        Expression balanceRule = new VariableExpression("balance", ">", 50000);
        Expression rule = new AndExpression(ageRule, balanceRule);

        boolean eligible = rule.interpret(ctx);
        System.out.println("Eligible for discount: " + eligible);
    }
}

```

### Output:

Eligible for discount: true

 Congratulations — you’ve built a *mini rule engine* using the Interpreter pattern.

---

## 6. How This Scales

In a real-world **Spring Boot rule-based framework**, you could:

- Store rule definitions in a database or YAML file.
- Parse them dynamically.
- Build an **Expression Tree** (similar to above).
- Evaluate dynamically for each request (no code change).

For example:

```

rules:
- name: SeniorDiscount
  expression: "(age > 60) AND (balance > 50000)"

```

```
action: "applyDiscount(10)"
```

You'd then parse and interpret `expression` at runtime.

This is how **Drools**, **MVEL**, or **Spring Expression Language (SpEL)** work under the hood — **Interpreter Pattern at scale**.

---

## 7. Spring Expression Language (SpEL) — Built-in Interpreter

Spring already implements the Interpreter pattern internally via **SpEL**.

Example:

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("age > 60 and balance > 50000");

StandardEvaluationContext ctx = new StandardEvaluationContext();
ctx.setVariable("age", 65);
ctx.setVariable("balance", 70000);

boolean result = exp.getValue(ctx, Boolean.class);
System.out.println("Eligible: " + result);
```

✅ Same idea — interpret expression at runtime → **Interpreter pattern**.

---



## 8. Real Use Cases in Enterprise Apps

Use Case	Interpreter Role
<b>Rule Engines</b> (Drools, SpEL)	Evaluate business rules dynamically
<b>Workflow Engines</b>	Interpret workflow DSL (e.g., BPMN)
<b>Search Query Builders</b>	Parse query strings like <code>(name=John AND age&gt;30)</code>
<b>Custom Validation Engines</b>	Define dynamic validation logic
<b>Feature Toggles</b>	Interpret runtime conditions for feature activation

---



## 9. Summary Table

Concept	Description
<b>Pattern Type</b>	Behavioral
<b>Goal</b>	Define grammar and evaluate expressions dynamically
<b>Key Benefit</b>	Flexibility — no hardcoding of logic
<b>Spring Boot Use</b>	Dynamic rule evaluation using SpEL or custom interpreters
<b>Example Frameworks</b>	Drools, SpEL, MVEL, Hibernate Criteria

---



## 10. Key Takeaways

- **Interpreter pattern** = turning *rules/expressions* into executable objects.
- It's **behavioral** because it controls **how expressions behave at runtime**.

- In **Spring Boot**, it's fundamental for:
    - **SpEL**
    - **Rule-based services**
    - **Workflow or Policy engines**
  - It's the foundation behind “configurable intelligence” in enterprise systems.
- 

Would you like me to extend this into a **complete mini rule engine project** (with YAML rule definitions, parser, and Spring Boot REST endpoints)?

That would show a *real-world application* of Interpreter pattern integrated with Spring Boot architecture (Controller → Service → Interpreter Engine).

**You said:**

give me annotation based example of spel

ChatGPT said:

Excellent 👍 — SpEL (**Spring Expression Language**) can be used **directly inside annotations**, and this is where it becomes extremely powerful and elegant — no XML, no manual parsing — just expressive, dynamic logic baked right into your code.

Let's go step-by-step with a **clear, working, annotation-based example** (Spring Boot style) to understand how it works.

---

## **Goal**

We'll build a **Spring Boot example** that:

- Uses **SpEL expressions inside annotations** (@Value, @PreAuthorize, @Cacheable).
  - Demonstrates **dynamic evaluation** at runtime.
  - Explains **how Spring interprets** these expressions under the hood.
- 

## **1. Basic Setup**

```
@SpringBootApplication
public class SpelAnnotationApp {
    public static void main(String[] args) {
        SpringApplication.run(SpelAnnotationApp.class, args);
    }
}
```

---

## **2. Example 1 — Using @Value with SpEL**

This is the most common use case.

You can evaluate dynamic expressions inside the @Value annotation.

```

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class ValueExampleBean {

    // Literal expression
    @Value("#{2 + 3}")
    private int sum;

    // Access system properties
    @Value("#{systemProperties['user.name']}")
    private String userName;

    // Access environment variables
    @Value("#{systemEnvironment['OS']}")
    private String operatingSystem;

    // Conditional expression
    @Value("#{2 > 1 ? 'Greater' : 'Smaller'}")
    private String comparisonResult;

    public void printValues() {
        System.out.println("Sum = " + sum);
        System.out.println("User Name = " + userName);
        System.out.println("Operating System = " + operatingSystem);
        System.out.println("Comparison Result = " + comparisonResult);
    }
}

```

---

## Explanation

Expression	Meaning
<code>#{2 + 3}</code>	Inline arithmetic
<code>#{systemProperties['user.name']}</code>	Access JVM system property
<code>#{systemEnvironment['OS']}</code>	Access OS environment variable
<code>#{2 &gt; 1 ? 'Greater' : 'Smaller'}</code>	Conditional (ternary) evaluation

---

## 3. Example 2 — SpEL in @Bean methods

You can use SpEL directly inside a `@Bean` definition to set dynamic values.

```

import org.springframework.context.annotation.*;

@Configuration
public class AppConfig {

    @Bean
    public String
    dynamicMessage(@Value("#{T(java.time.LocalDate).now().toString()}") String date)
    {
        return "Today is " + date;
    }
}

```

The expression `T(java.time.LocalDate).now()` calls a **static method** using the `T()` operator.



## 4. Example 3 — SpEL in Security (@PreAuthorize)

If you use **Spring Security**, you can evaluate expressions to **control method access dynamically**.

```
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.stereotype.Service;
```

```
@Service
public class AccountService {

    // Only allow if principal.username == 'admin'
    @PreAuthorize("#username == authentication.name")
    public void deleteAccount(String username) {
        System.out.println("Account deleted: " + username);
    }
}
```

Here:

- `#username` → method parameter.
- `authentication.name` → logged-in user from `SecurityContext`.
- Expression evaluated dynamically before method invocation.



Useful for **role-based or user-based access control** without hardcoding.

---



## 5. Example 4 — SpEL in Caching Annotations

```
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;
```

```
@Service
public class ProductService {

    // Cache key is dynamically evaluated from method parameter
    @Cacheable(value = "products", key = "#id")
    public String getProductById(int id) {
        System.out.println("Fetching product from DB for ID = " + id);
        return "Product-" + id;
    }
}
```

Here:

- `#id` → refers to the method argument.
  - `@Cacheable` uses SpEL to evaluate the cache key dynamically.
- 



## 6. Example 5 — Combining Bean References

SpEL can refer to other beans using `@` syntax.

```
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Value;
```



```

@Component
public class DiscountCalculator {

    @Value("#{priceCalculator.basePrice * 0.9}")
    private double discountedPrice;

    public double getDiscountedPrice() {
        return discountedPrice;
    }
}

@Component
class PriceCalculator {
    public double basePrice = 1000.0;
}

```

- `@Value("#{priceCalculator.basePrice * 0.9}")` → accesses another bean's field (`priceCalculator`) and computes a dynamic value.
- Result → `discountedPrice = 900.0`.

## 7. Example 6 — Full Spring Boot Demo

```

import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class SpelRunner implements CommandLineRunner {

    private final ValueExampleBean bean;
    private final ProductService productService;
    private final DiscountCalculator discountCalculator;

    public SpelRunner(ValueExampleBean bean, ProductService productService,
DiscountCalculator discountCalculator) {
        this.bean = bean;
        this.productService = productService;
        this.discountCalculator = discountCalculator;
    }

    @Override
    public void run(String... args) {
        bean.printValues();
        System.out.println("Discounted Price: " +
discountCalculator.getDiscountedPrice());
        System.out.println("Product: " + productService.getProductById(101));
        System.out.println("Product (cached): " +
productService.getProductById(101)); // won't fetch again
    }
}

```

## 8. Supported Expression Features

Feature	Example	Description
Arithmetic	<code>#{10 + 5}</code>	Math operations

Feature	Example	Description
Logical	<code>{age &gt; 18 and country == 'India'}</code>	Logical ops
String manipulation	<code>"Hello " + name}</code>	Concatenation
Static method call	<code>T(java.lang.Math).random()</code>	Using T()
Bean reference	<code>{@beanName.property}</code>	Access other beans
Collections	<code>{list[0]}, {map['key']}</code>	Access collection values



## 9. How It Works Internally

Spring's **Annotation Bean Post-Processor** intercepts annotations like `@Value`, `@Cacheable`, etc.

It uses the **SpEL parser** (`SpELExpressionParser`) to parse the expression inside `{}` and evaluate it against:

- BeanFactory context (beans)
- System properties
- Environment variables
- Method parameters (if applicable)



## 10. Real Use Cases of SpEL with Annotations

Use Case	Annotation	Purpose
Inject dynamic config	<code>@Value</code>	Compute or read runtime data
Security control	<code>@PreAuthorize</code> , <code>@PostAuthorize</code>	Role/parameter-based access
Caching logic	<code>@Cacheable</code> , <code>@CachePut</code> , <code>@CacheEvict</code>	Custom cache keys
Scheduling	<code>@Scheduled(cron = "{scheduler.cronExpression}")</code>	Read cron dynamically
Feature toggles	<code>@ConditionalOnExpression("{environment['featureX.enabled']})"</code>	Conditional bean loading



## Summary

Concept	Example	Description
<b>Pattern</b>	Interpreter (Behavioral)	SpEL internally uses the Interpreter pattern
<b>Annotation</b>	<code>@Value("{expression}")</code>	Embeds expressions directly
<b>Use</b>	Dynamic injection, logic, condition, security, cache	
<b>Spring Feature</b>	Built-in in core container and Spring Security	
<b>Advantage</b>	No code changes for logic tweaks — just change expression or config	

---

## ❏ In short

**SpEL** is Spring's *Interpreter pattern* in action, letting you embed dynamic logic into annotations, properties, or beans — evaluated at runtime, safely and declaratively.